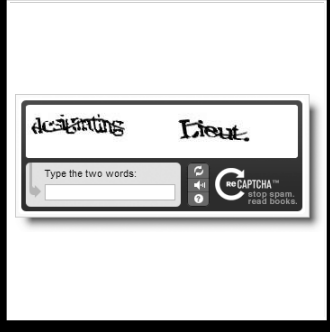
11 USING PHP TO PROCESS A FORM



In Chapter 9, I showed you how to build a feedback form and validate the input on the client side with Spry validation widgets. In this chapter, we'll take the process to its next stage by validating the data on the server side with PHP. If the data is OK, we'll send the contents by email and display an acknowledgment message. If there's a problem with any of the data, we'll redisplay it in the form with messages prompting the user to correct any errors or omissions. Figure 11-1 shows the flow of events.

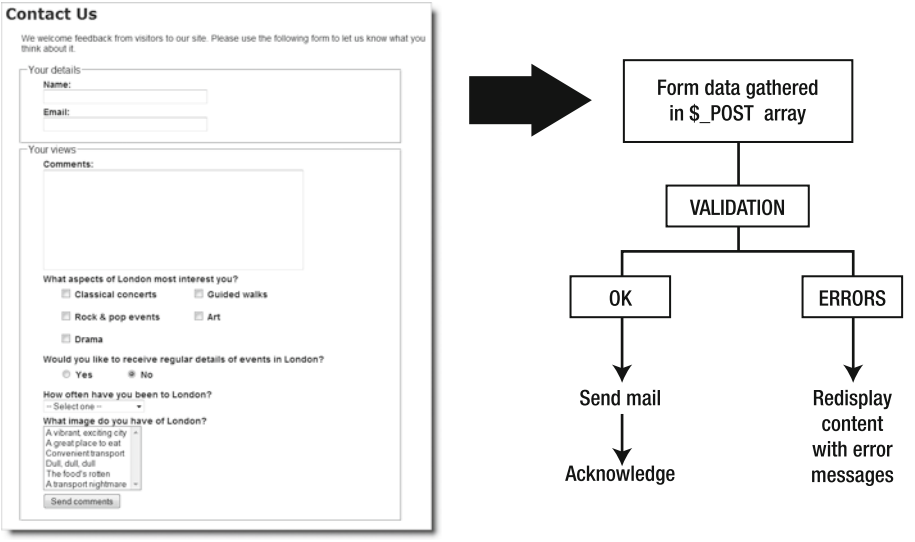


Figure 11-1. The flow of events in processing the feedback form

Sending an email from an online form is just the sort of task that Dreamweaver should automate, but unfortunately it doesn't. Commercial extensions are available to automate the process for you, but not everyone will have—or want to buy—a commercial extension in addition to Dreamweaver CS4, so I think it's important to show you how to hand-code this vital feature. At the same time, it gives you practical experience working with PHP code, which is essential unless you are willing to be limited to very basic tasks. The Dreamweaver server behaviors and data objects that you will use in later chapters take a lot of the hard work out of creating dynamic applications, but like the CSS layout that you used in Chapter 5, they lay a solid foundation for you to build on, rather than do absolutely everything for you.

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

- Gathering user input and sending it by email
- Using PHP conditional logic to check required fields
- Displaying errors without losing user input
- Saving frequently used code as a snippet

- Filtering out suspect material
- Avoiding email header injection attacks
- Processing multiple-choice form elements
- Blocking submission by spam bots

The flow of events shown in Figure 11-1 is controlled by a series of conditional statements (see "Making decisions" in the previous chapter). The PHP script will be in the same page as the form, so the first thing it needs to know is `if` the form has been submitted. `If` it has, the contents of the `$_POST` array will be checked. `If` it's OK, the email will be sent and an acknowledgment displayed, `else` a series of error messages will be displayed. In other words, everything is controlled by `if . . . else` statements.

Activating the form

As you saw in Chapter 9, data entered into the form can be retrieved by using `print_r($_POST);` to inspect the contents of the `$_POST` array. This is one of PHP's so-called superglobal arrays. They're such an important part of PHP that it's worth pausing for a moment to take a look at what they do.

Getting information from the server with PHP superglobals

Superglobal arrays are built-in associative arrays that are automatically populated with really useful information. They all begin with a dollar sign followed by an underscore. The most important superglobal arrays are as follows:

- **`$_POST`**: This contains values sent through the `post` method.
- **`$_GET`**: This contains values sent through the `get` method or a URL query string.
- **`$_SERVER`**: This contains information stored by the web server, such as file name, pathname, hostname, and so on.
- **`$_SESSION`**: This stores information that you want to preserve so that it's available to other pages. Sessions are covered in Chapter 15.
- **`$_FILES`**: This contains details of file uploads. File uploads are not covered in this book. See `http://docs.php.net/manual/en/features.file-upload.php` or my book *PHP Solutions: Dynamic Web Design Made Easy* (friends of ED, ISBN: 978-1-59059-731-6) for details.

The keys of `$_POST` and `$_GET` are automatically derived from the names of form elements. Let's say you have a text input field called `address` in a form; PHP automatically creates an array element called `$_POST['address']` when the form is submitted by the `post` method or `$_GET['address']` if you use the `get` method. As Figure 11-2 shows, `$_POST['address']` contains whatever value a visitor enters in the text field, enabling you

to display it onscreen, insert it in a database, send it to your email inbox, or do whatever you want with it.

Figure 11-2. The `$_POST` array automatically creates variables with the same name and value as each form field.

It's important to realize that variables like `$_POST['address']` or `$_GET['address']` don't exist until the form has been submitted. So, before using `$_POST` or `$_GET` variables in a script, you should always test for their existence with `isset()` or wrap the entire section of script in a conditional statement that checks whether the form has been submitted. You'll see both of these techniques in action in this chapter and the rest of this book.

You may come across old scripts or tutorials that tell you PHP automatically creates variables with the same name as form fields. In this example, it would be `$address`. This relies on a setting called `register_globals` being on. The default for this setting has been off since 2002, because it leaves your site wide open to malicious attacks. Most hosting companies now seem to have turned it off, but don't be tempted to try to find a way to turn it back on. It has been removed from PHP 6, so scripts that rely on `register_globals` will break in future.

Some scripts also recommend the use of `$_REQUEST`, which is another PHP superglobal. It's much less secure. Always use `$_POST` for data submitted using the `post` method and `$_GET` for the `get` method or when values are passed through a query string at the end of a URL.

> *Don't forget that PHP is case-sensitive. All superglobal array names are written in uppercase.* `$_Post` *or* `$_Get`*, for example, won't work.*

Dreamweaver code hints make it easy to type the names of superglobals. As soon as you type the underscore after the dollar sign, it displays a list of the array names; and for arrays such as `$_SERVER` with predefined elements, a second menu with the predefined elements is also displayed, as you'll see when you start scripting the form.

Sending email

To send an email with PHP, you use the `mail()` function, which takes up to five arguments, as follows (the first three are required):

- **Recipient(s)**: The email address(es) to which the message is being sent. Addresses 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>'
```

- **Subject**: A string containing the subject line of the message.
- **Body**: This is the message being sent. It should be a single string, regardless of how long it is. However, the email standard imposes a maximum line length. I'll describe how to handle this later.
- **Additional headers**: This is an optional set of email headers, such as `From`, `Cc`, `Reply-to`, and so on. They must be in a specific format, which is described later in this chapter.
- **Additional parameters**: As an antispam measure, some hosting companies require verification that the email originates from the registered domain. I'll explain how to use this argument later in the chapter.

It's important to understand that `mail()` isn't an email program. It passes data to the web server's mail transport agent (MTA). PHP's responsibility ends there. It has no way of knowing whether the email is delivered to its destination. It doesn't handle attachments or HTML email. Still, it's efficient and easy to use.

These days, most Internet service providers (ISPs) enforce Simple Mail Transfer Protocol (SMTP) authentication before accepting email for relay from another machine. However, `mail()` was designed to communicate directly with the MTA on the same machine, without the need for authentication. This presents a problem for testing `mail()` in a local testing environment. Since `mail()` doesn't normally need to authenticate itself, it's not capable of doing so. More often than not, when you attempt to use `mail()` on your local computer, it can't find an MTA or the ISP rejects the mail without authentication.

> *Although I normally recommend testing everything locally before uploading PHP scripts to a remote server, it's usually not possible with* `mail()`*, especially if you need to log into your normal email account. Some parts of the following script can be tested locally, but when it comes to the sections that actually send the mail, the overwhelming majority of readers will need to upload the script to their website and test it from there.*

Scripting the feedback form

To make things simple, I'm going to break up the PHP script into several sections. To start off, I'll concentrate on the text input fields and sending their content by email. Then I'll move onto validation and the display of error messages before showing you how to handle checkboxes, radio buttons, menus, and multiple-choice lists.

Most readers should be able to send a simple email after the following exercise, but even if you are successful, you should implement the server-side validation described later in the chapter. This is because, without some simple security precautions, you risk turning your online forms into a spam relay. Your hosting company might suspend your site or close down your account altogether.

This involves a lot of hand-coding—much more than you'll encounter in later chapters. To reduce the amount of typing you need to do, I have created an extension that contains several PHP functions stored as Dreamweaver snippets (small pieces of code that can be easily inserted into any page). I suggest you install them now so they're ready for use in this and subsequent chapters.

Installing the PHP snippets

To install the snippets, you need to have installed the Extension Manager when you originally installed Dreamweaver CS4. If you accepted the default options when installing Dreamweaver, you should have access to the Extension Manager. However, if you deselected all the optional programs and components, you will need to install the Extension Manager from your Dreamweaver or Creative Suite 4 DVD. The extension file is called `dwcs4_snippets.mxp` and is in the `extras` folder of the download files for this book.

1. Launch the Extension Manager as described in Chapter 8.
2. Click the Install button, navigate to `dwcs4_snippets.mxp`, and install it.
3. Close and relaunch Dreamweaver.
4. The snippets should have been installed in a folder called `PHP-DWCS4` in the Dreamweaver Snippets panel (see Figure 11-3). They are now accessible for use in any site.

I'll show you how to insert a snippet in a page later in this chapter.

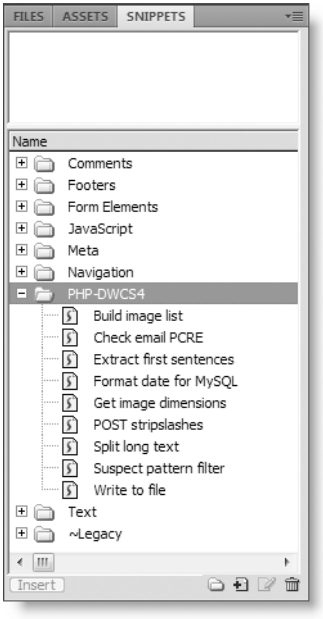


Figure 11-3. The extension installs a set of useful PHP scripts.

This is a long script. Give yourself plenty of time to absorb the details. You can check your progress at each stage with the files in `examples/ch11`. *The final code is in* `feedback_12.php`. *Even if you don't want to do a lot of PHP programming, it's important to get a feel for the flow of a script, because this will help you customize the Dreamweaver code once you start working with a database. The script uses a lot of PHP's built-in functions. I explain the important ones but don't always go into the finer points of how they work. The idea is to give you a working solution, rather than overwhelm you with detail. In the next chapter, I'll show you how to put the main part of the script in an external file so that you can reuse it with other forms without the need to hand-code everything from scratch every time.*

Processing and acknowledging the message

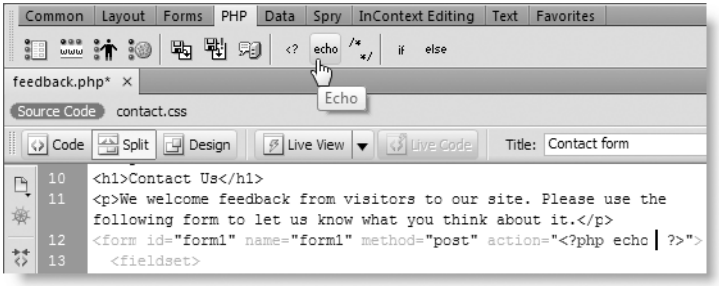
The starting point is in `feedback_01.php` in `examples/ch11`. It's the same as `feedback_fieldsets.php` from Chapter 9 but with the small block of PHP code removed from the bottom of the page. If you want to use your own form, I suggest you remove any client-side validation from it, because the client-side validation makes it difficult to check whether the more important server-side validation with PHP is working correctly. You can add the client-side validation back at the final stage.

1. Copy `feedback_01.php` and `contact.css` from `examples/ch11` to `workfiles/ch11`. Rename `feedback_01.php` to `feedback.php`. If Dreamweaver asks you whether to update links, click No.
2. Select `contact.css` in the Related Files toolbar to open it in Split view, and add the following style rule:

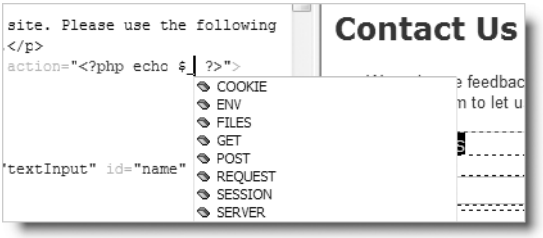
```
.warning {
  font-weight:bold;
  color:#F00;
}
```

   This adds a class called `warning`, which displays text in bold red. Save `contact.css`.
3. Select Source Code in the Related Files toolbar to display the underlying code of `feedback.php` in Split view, click anywhere in the form, and use the Tag selector at the bottom of the Document window to select the entire form. This should bring the opening tag of the form into view in Code view. Click in Code view so that your cursor is between the quotes of the `action` attribute. Although you can set the `action` for the form through the Property inspector, doing so in Code view greatly reduces the possibility of making a mistake.
4. Select the PHP tab on the Insert bar, and click the Echo button (the menu option is Insert ➤ PHP Objects ➤ Echo). This will insert a pair of PHP tags followed by echo

between the quotes of the `action` attribute, and Dreamweaver positions your cursor in the correct place to start typing, as shown in the following screenshot:



5. To set the `action` attribute of the form to process itself, you need to use a variable from the `$_SERVER` superglobal array. As noted before, superglobals always begin with `$_`, so type just that at the current position. Dreamweaver automatically presents you with a pop-up menu containing all the superglobals, as shown here:



You can navigate this pop-up menu in several ways: continue typing `server` in either uppercase or lowercase until `SERVER` is highlighted or use your mouse or the arrow keys to highlight it. Then double-click or press Enter/Return. Dreamweaver will present you with another pop-up menu. Locate `PHP_SELF` as shown here, and either double-click or press Enter/Return:

6. Although it's not strictly necessary for a single command, get into the habit of ending all statements with a semicolon, and type one after the closing square bracket

([]) of the superglobal variable that's just been entered. The code in the opening <form> tag should look like this (new code is highlighted in bold type):

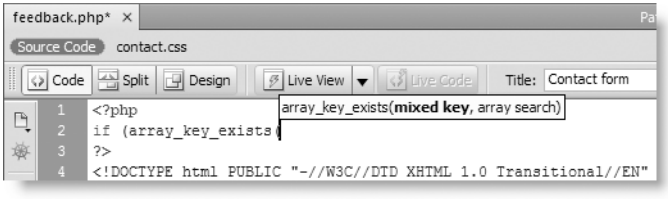
```
<form id="form1" name="form1" method="post" action="<?php echo ➥
$_SERVER['PHP_SELF']; ?>">
```

The predefined variable $_SERVER['PHP_SELF'] always contains the name of the current page, so using echo between the quotes of the action attribute automatically sets it to the current page, making this a self-processing form. As you saw in Chapter 9, leaving out the value of action also results in the form attempting to process itself. So, technically speaking, this isn't 100-percent necessary, but it's common practice in PHP scripts, and it's useful to know what $_SERVER['PHP_SELF'] does.

7. You now need to add the mail-processing script at the top of the page. As you saw in Chapter 9, the $_POST array contains not only the data entered into the form but also the name and value of the submit button. You can use this information to determine whether the submit button has been clicked. From this point onward, it will be easier to work in Code view. Switch to Code view, and insert the following block of PHP code immediately above the DOCTYPE declaration:

```
<?php
if (array_key_exists('send', $_POST)) {
  // mail processing script
  echo 'You clicked the submit button';
}
?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" ➥
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

This uses the PHP function array_key_exists() to check whether the $_POST array contains a key called send, the name attribute of the form submit button. If you don't want to type the function name yourself, you can press Ctrl+Space to bring up an alphabetical list of all PHP functions. Type just the first few letters, and then use your arrow keys to select the right one. When you press Tab or Enter/Return, Dreamweaver finishes the rest of the typing and pops up a code hint. Alternatively, just type the function name directly, and the code hint appears as soon as you enter the opening parenthesis after array_key_exists, as shown here:



The mixed data type refers to the fact that array keys can be either numbers or strings. In this case, you are using a string, so enclose send in quotes, and then after

a comma, type $_POST. Because it's a superglobal, you are presented with the same pop-up menu as in step 5. If you select POST, Dreamweaver assumes you want to add the name of an array key and will automatically add an opening square bracket after the T. On this occasion, you want to check the whole $_POST array, not just a single element, so remove the bracket by pressing Backspace. Make sure you use two closing parentheses—the first belongs to the function array_key_exists(), and the second encloses the condition being tested for by the if statement.

If the send array key exists, the submit button must have been clicked, so any script between the curly braces is executed. Otherwise, it's ignored. Don't worry that echo will display text above the DOCTYPE declaration. It's being used for test purposes only and will be removed eventually.

> *Remember, an* if *statement doesn't always need to be followed by* else *or* elseif. *When the condition of a solitary* if *statement isn't met, PHP simply skips to the next block of code.*

8. Save feedback.php, and test it in a browser. It should look no different from before.
9. Click the Send comments button. A message should appear at the top of the page saying "You clicked the submit button."
10. Reload the page without using the browser's reload button. Click inside the address bar, and press Enter/Return. The message should disappear. This confirms that any code inside the curly braces runs only if the submit button has been clicked.
11. Change the block of code you entered in step 7 so it looks like this:

```
<?php
if (array_key_exists('send', $_POST)) {
  //mail processing script
  $to = 'me@example.com'; // use your own email address
  $subject = 'Feedback from Essential Guide';

  // process the $_POST variables
  $name = $_POST['name'];
  $email = $_POST['email'];
  $comments = $_POST['comments'];

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

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

  // send it
  $mailSent = mail($to, $subject, $message);
}
?>
```

The code that does the processing consists of five stages. The first two lines assign your email address to `$to` and the subject line of the email to `$subject`.

Next, `$_POST['name']`, `$_POST['email']`, and `$_POST['comments']` are reassigned to ordinary variables to make them easier to handle.

The shorter variables are then used to build the body of the email message, which must consist of a single string. As you can see, I have used the combined concatenation operator (`.=`) to build the message and escape sequences to add carriage returns and newline characters between each section (see "Adding to an existing string" and "Using escape sequences in strings" in Chapter 10).

Once the message body is complete, it's passed to the `wordwrap()` function, which takes two arguments: a string and an integer that sets the maximum length of each line. Although most mail systems will accept longer lines, it's recommended to limit each line to 70 characters.

After the message has been built and formatted, the recipient's address, subject line, and body of the message are passed to the `mail()` function. There is nothing magical about the variable names `$to`, `$subject`, and `$message`. I chose them to describe what each one contains, making much of the script self-commenting.

The `mail()` function returns a Boolean value (`true` or `false`) indicating whether it succeeded. By capturing this value as `$mailSent`, you can use it to redirect the user to another page or change the contents of the current one.

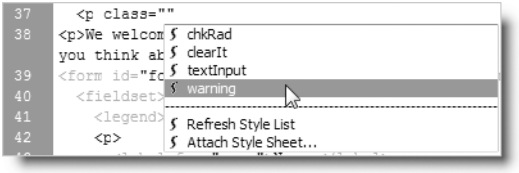
> *The official format for email is described in a document known as Request For Comments (RFC) 2822 (*`http://tools.ietf.org/html/rfc2822`*). Among other things, it says that carriage returns and newline characters must not appear independently in the body of a message; they must always be together as a pair. It also sets the maximum length of a line in the body at 998 characters but recommends restricting lines to no more than 78. The reason I have set* `wordwrap()` *to a more conservative 70 characters is to avoid problems with some mail clients that automatically wrap messages. If you set the value too high, you end up with alternating long and short lines.*

12. For the time being, let's keep everything in the same page, because the rest of the chapter will add further refinements to the basic script. Scroll down, and insert the following code just after the page's main heading (new code is highlighted in bold):

```
<h1>Contact us</h1>
<?php
if ($_POST && !$mailSent) {
?>
  <p class="warning">Sorry, there was a problem sending your message.
Please try later.</p>
<?php
} elseif ($_POST && $mailSent) {
?>
  <p><strong>Your message has been sent. Thank you for your feedback.
</strong></p>
<?php } ?>
<p>We welcome feedback from visitors . . .</p>
```

Many beginners mistakenly think you need to use `echo` or `print` to display HTML in a PHP block. However, except for very short pieces of code, it's more efficient to switch back to HTML, as I've done here. Doing so avoids the need to worry about escaping quotes. Also, Dreamweaver code hints and automatic tag completion speed things up for you. As soon as you type a space after `<p` in the first paragraph, Dreamweaver pops up a code hint menu like this:

Select `class`. As soon as you do so, Dreamweaver checks the available classes in the attached style sheet and pops up another code hint menu, as shown in the next screenshot, so you can choose `warning`:



This makes coding much quicker and more accurate. Dreamweaver's context sensitivity means you get the full range of HTML code hints only when you're in a section of HTML code. When you're in a block of PHP code, you get a list of HTML tags when you type an opening angle bracket, but there are no attribute hints or autocompletion. So, it makes more sense to use PHP for the conditional logic but keep the HTML separate. The only thing you need to watch carefully is that you balance the opening and closing curly braces correctly. I'll show you how to do that in "Using Balance Braces" a little later in the chapter.

So, what does this code do? It may look odd if you're not used to seeing scripts that mix HTML with PHP logic, but it can be summarized like this:

```
<h1>Contact us</h1>
<?php
if ($_POST && !$mailSent) {
  // display a failure message
} elseif ($_POST && $mailSent) {
  // display an acknowledgment
 }
?>
<p>We welcome feedback from visitors . . .</p>
```

Both parts of the conditional statement check the Boolean values of `$_POST` and `$mailSent`. Although the `$_POST` array is always set, it doesn't contain any values unless the form has been submitted. Since PHP treats an empty array as `false` (see "The truth according to PHP" in Chapter 10), you can use `$_POST` on its own to test whether a form has been submitted. So, the code in both parts of this conditional statement is ignored when the page first loads.

However, if the form has been submitted, `$_POST` equates to `true`, so the next condition is tested. The exclamation mark in front of `$mailSent` is the negative operator, making it the equivalent of *not* `$mailSent`. So, if the email hasn't been sent, both parts of the test are `true`, and the HTML containing the error message is displayed. However, if `$mailSent` is `true`, the HTML containing the acknowledgment is displayed instead.

13. Save `feedback.php`, and switch to Design view. The top of the page should now look like this:

There are three gold shields indicating the presence of PHP code, and both the error and acknowledgment messages are displayed. You need to get used to this sort of thing when designing dynamic pages.

If you don't see the gold shields, refer to "Passing information through a hidden field" in Chapter 9 for details of how to control invisible elements in Design view.

14. To see what the page looks like when the PHP is processed, click the Live View button in the Document toolbar. Dreamweaver will ask whether you want to update the copy on the testing server. Click Yes.

If you have coded everything correctly, the error message and acknowledgment should disappear. Click the Live View button to toggle it off again.

If you got a PHP error message, read "Using Balance Braces," and then check your code against `feedback_02.php`.

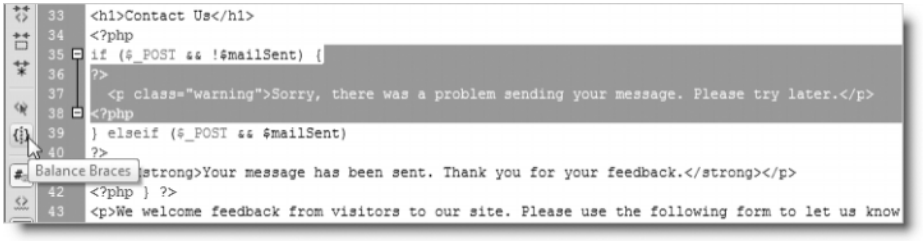
The script in step 11 is theoretically all you need to send email from an online form. Don't be tempted to leave it at that. Without the security checks described in the rest of the chapter, you run the risk of turning your website into a spam relay.

Using Balance Braces

Even if you didn't encounter a problem in the preceding exercise, Balance Braces is a tool that you definitely need to know about. Like quotes, curly braces must always be in matching pairs, but sometimes the opening and closing braces can be dozens, even hundreds, of lines apart. If one of a pair is missing, your script will collapse like a house of cards. Balance Braces matches pairs in a highly visual way, making troubleshooting a breeze.

Let's take a look at the code in step 12 that I suspect will trip many people up. I deliberately removed an opening curly brace at the end of line 39 in the following screenshot. That triggered a parse error, which reported an unexpected closing curly brace on line 42. Now, that could mean either of the following:

- There's a missing opening brace to match the closing one.
- There's an extra closing brace that shouldn't be there.



The way to resolve the problem is to place your cursor anywhere between a pair of curly braces, and click the Balance Braces button in the Coding toolbar. This highlights the code between the matching braces. I started by placing my cursor on line 37. As you can see, it highlighted all the code between the braces on lines 35 and 38.

Next, I positioned my cursor on line 41. When I clicked the Balance Braces button again, nothing was highlighted, and my computer just beeped. So there was the culprit. All I needed to work out was where the opening brace should go. My first test showed that I had a logical block on lines 35–38 (the closing brace is at the beginning of line 39), so it was just a process of elimination tracking down the missing brace. If the problem had been an extra curly brace that shouldn't have been there, the code would have been highlighted, giving me a clear indication of where the block ended.

Although it can't tell you whether your code logic is right or where a missing brace should go, you'll find this tool a great time-saver. It works not only with braces but also with square brackets and parentheses. Just position your cursor inside any curly brace, square bracket, or parenthesis, and click the Balance Braces button to find the other one of the pair. You may need to test several blocks to find the cause of a problem, but it's an excellent way of visualizing code blocks and the branching logic of your scripts.

You can also access Balance Braces through the Edit menu, and if you're a keyboard shortcut fan, the combination is Ctrl+'/Cmd+' (single quote).

Testing the feedback form

Assuming that you now have a page that displays correctly in Live view, it's time to test it. As mentioned earlier, testing `mail()` in a local PHP testing environment is unreliable, so I suggest you upload `feedback.php` to a remote server for the next stage of testing. Once you have established that the `mail()` function is working, you can continue testing locally.

Upload `feedback.php` and `contact.css` to your remote server. Enter some text in the Name, Email, and Comments fields. Make sure your input includes at least an apostrophe or quotation mark, and click Send comments. The form should clear, and you should see a confirmation message, as in Figure 11-4.

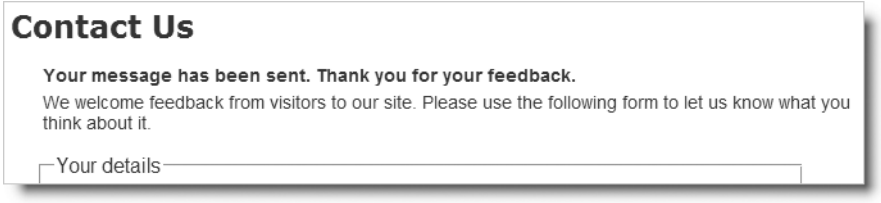


Figure 11-4. Confirmation that the mail() function has passed the message to the server's mail transport agent

Shortly afterward, you should receive the message in your inbox. Most of the time, it should work, but there are several things that might go wrong. The next section should help you resolve the problem.

> *If you see an error message saying that the* `From` *header wasn't set or that* `sendmail_from` *isn't defined in* `php.ini`*, keep building the script as described in each section, and I'll tell you when you can test your page on the remote server again. If you get a blank page, it means you have a syntax error in your PHP script, use the File Compare feature (see Chapter 2) to compare your code with* `feedback_02.php` *in* `examples/ch11`.

Troubleshooting mail()

If you don't receive anything, the first thing to check is your spam trap, because the email may appear to come from an unknown or a suspicious source. For example, it may appear to come from Apache or a mysterious nobody (the name often used for web servers). Don't worry about the odd name; that will be fixed soon. The main thing is to check that the mail is being sent correctly.

Improving the security of the mail-processing script

As the preceding exercise showed, the basic principles of processing the contents of a form and sending it by email to your inbox are relatively simple. However, you can improve the script in many ways; indeed, some things must be done to improve its security. One of the biggest problems on the Internet is caused by insecure scripts. As the mail processing script currently stands, it's wide open to abuse. If you received an error

message about the `From` header not being set, it indicates that your hosting company has taken measures to increase security and prevent poorly written mail scripts from being used as spam relays. Your script won't work until you implement the security measures in the following sections.

Most of the rest of this chapter is devoted to improving the security and user experience of the existing script. First I'll deal with unwanted backslashes that might appear in your email.

Getting rid of unwanted backslashes

Some day back in the mists of time, the PHP development team had the "brilliant" idea of creating a feature known as magic quotes . . . only it wasn't so brilliant after all. When inserting data into a database, it's essential to escape single and double quotes. So, the idea of magic quotes was to make life simpler for beginners by doing this automatically for all data passed through the `$_POST` and `$_GET` arrays, and cookies. While this seemed like a good idea at the time, it has caused endless problems. To cut a long story short, magic quotes are being officially phased out of PHP (they'll be gone in PHP 6), but they're still enabled on a lot of shared servers. You will know whether your server uses them if your test email has backslashes in front of any apostrophes or quotes, as shown in Figure 11-5.

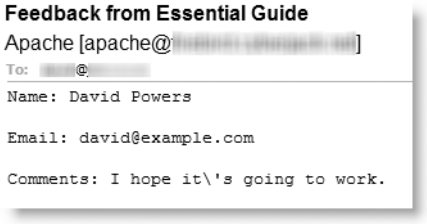


Figure 11-5. PHP magic quotes insert unwanted backslashes in the email.

Dreamweaver's server behaviors automatically handle magic quotes by stripping the backslashes, if necessary, and preparing data for database input. However, when you're hand-coding like this, you need to deal with the backslashes yourself.

I have created a Dreamweaver snippet so that you can drop a ready-made script into any page that needs to get rid of unwanted backslashes. It automatically detects whether magic quotes are enabled, so you can use it safely on any server. If magic quotes are on, it removes the backslashes. If magic quotes are off, it leaves your data untouched. It's part of the collection of snippets that you should have installed as a Dreamweaver extension at the beginning of this chapter.

Using the POST stripslashes snippet

These instructions continue the creation of the form processing script. So, continue working with `feedback.php` from the previous section. They also show you how to insert code from the Snippets panel.

1. Open `feedback.php` in Code view. Position your cursor at the beginning of line 4, just under the `mail processing script` comment, and insert a couple of blank lines.

   Move your cursor onto one of the blank lines, and open the Snippets panel by clicking the Snippets tab in the Files panel group or selecting Window ➤ Snippets.

On Windows, you can also use the keyboard shortcut Shift+F9, but this doesn't work on the Mac version.

Highlight the POST stripslashes snippet in the PHP-DWCS4 folder, and double-click it, or click the Insert button at the bottom of the panel.

2. This inserts the following block of code into your page:

```
// remove escape characters from $_POST array
if (PHP_VERSION < 6 && get_magic_quotes_gpc()) {
  function stripslashes_deep($value) {
    $value = is_array($value) ? array_map('stripslashes_deep', ➥
$value) : stripslashes($value);
    return $value;
    }
  $_POST = array_map('stripslashes_deep', $_POST);
  }
```

Lying at the heart of this code is the PHP function `stripslashes()`, which removes the escape backslashes from quotes and apostrophes. Normally, you just pass the string that you want to clean up as the argument to `stripslashes()`. Unfortunately, that won't work with an array. This block of code checks whether the version of PHP is prior to PHP 6 and, if so, whether magic quotes have been turned on (magic quotes have been removed from PHP 6); and if they have, it goes through the `$_POST` array and any nested arrays, cleaning up your text for display either in an email or in a web page.

3. Save `feedback.php`, and send another test email that includes apostrophes and quotes in the message. The email you receive should be nicely cleaned up. This won't work yet if you weren't able to send the first test email.

If you have any problems, check your page against `feedback_03.php`.

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. The following instructions make use of arrays and the `foreach` loop, both of which are described in Chapter 10. So if you're new to PHP, you might find it useful to refer to the relevant sections in the previous chapter before continuing.

Checking required fields

In this part of the script, you create three arrays to hold details of variables you expect to receive from the form, those that are required, and those that are missing. This not only helps identify any required items that haven't been filled in; it also adds an important security check before passing the user input to a loop that converts the names of `$_POST` variables to shorter ones that are easier to handle.

1. Start by creating two arrays: one listing the `name` attribute of each field in the form and the other listing all *required* fields. Also, initialize an empty array to store the names of required fields that have not been completed. 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 just before the section that processes the $_POST variables:

```
$subject = 'Feedback from Essential Guide';

// list expected fields
$expected = array('name', 'email', 'comments');
// set required fields
$required = array('name', 'comments');
// create empty array for any missing fields
$missing = array();

// process the $_POST variables
```

2. At the moment, the $_POST variables are assigned manually to variables that use the same name as the $_POST array key. With three fields, manual assignment is fine, but it becomes a major chore with more fields. Let's kill two birds with one stone by checking required fields and automating the naming of the variables at the same time. Replace the three lines of code beneath the $_POST variables comment as follows:

```
// process the $_POST variables
foreach ($_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, $required)) {
    array_push($missing, $key);
  } elseif (in_array($key, $expected)) {
    // otherwise, assign to a variable of the same name as $key
    ${$key} = $temp;
  }
}

// build the message
```

If studying PHP code makes your brain hurt, you don't need to worry about how this works. As long as you create the $expected, $required, and $missing arrays in the previous step, you can just copy and paste the code for use in any form.

So, what does it do? In simple terms, this foreach loop goes through the $_POST array, strips out any whitespace from user input, 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.

The code uses several built-in PHP functions, all of which have intuitive names:

- is_array() tests whether a variable is an array.
- trim() trims whitespace from both ends of a string.
- empty() tests whether a variable contains nothing or equates to false.

- `in_array()` checks whether the first argument is part of the array specified in the second argument.
- `array_push()` adds a new element to the end of an array.

At this stage, you don't need to understand how each function works, but you can find details in the PHP online documentation at `http://docs.php.net/manual/en/index.php`. Type the name of the function in the search for field at the top right of the page (see Figure 11-6), and click the right-facing arrow alongside function list. The PHP documentation has many practical examples showing how functions and other features are used.

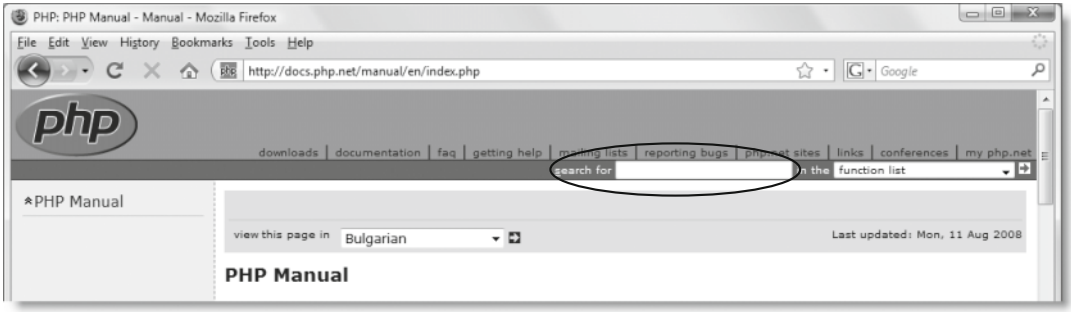


Figure 11-6. Refer often to the excellent PHP online documentation, and your skills will increase rapidly.

> *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.*

3. You want to build the body of the email message and send it only if all required fields have been filled in. Since `$missing` starts off as an empty array, nothing is added to it if all required fields are completed, so `empty($missing)` is `true`. Wrap the rest of the script in the opening PHP code block like this:

```
// go ahead only if all required fields OK
if (empty($missing)) {

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

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

```
    // send it
    $mailSent = mail($to, $subject, $message);
    if ($mailSent) {
      // $missing is no longer needed if the email is sent, so unset it
      unset($missing);
    }
  }
}
```

This ensures that the mail is sent only if nothing has been added to `$missing`. However, `$missing` will be used to control the display of error messages in the main body of the page, so you need to get rid of it if the mail is successfully sent. This is done by using `unset()`, which destroys a variable and any value it contains.

4. Let's turn now to the main body of the page. You need to display a warning if anything is missing. Amend the conditional statement at the top of the page content like this:

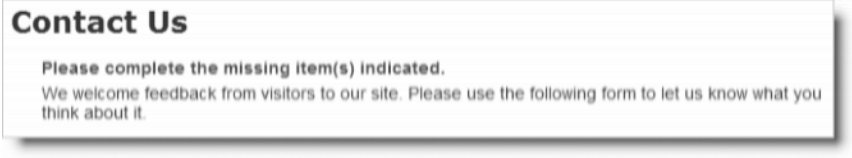
```
<h1>Contact us</h1>
<?php
if ($_POST && isset($missing) && !empty($missing)) {
?>
  <p class="warning">Please complete the missing item(s) indicated.</p>
<?php
} elseif ($_POST && !$mailSent) {
?>
  <p class="warning">Sorry, there was a problem sending your message.
Please try later.</p>
```

This adds a new condition. The `isset()` function checks whether a variable exists. If `$missing` doesn't exist, that means that all required fields were filled in and the email was sent successfully, so the condition fails, and the script moves on to consider the `elseif` condition. However, if all required fields were filled in but there was a problem sending the email, `$missing` still exists, so you need to make sure it's empty. An exclamation mark is the negative operator, so `!empty` means "not empty."

On the other hand, if `$missing` exists and *isn't* empty, you know that at least one required field was omitted, so the warning message is displayed.

I've placed this new condition first. The `$mailSent` variable won't even be set if any required fields have been omitted, so you must test for `$missing` first.

5. To make sure it works so far, save `feedback.php`, and load it in a browser. You don't need to upload it to your remote server, because you want to test the message about missing items. Don't fill in any fields. Just click Send comments. The top of the page should look like this (check your code against `feedback_04.php` if necessary):



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

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

Since the $missing array is created only after the form has been submitted, you need to check first with isset() that it exists. If it doesn't exist—such as when the page first loads or if the email has been sent successfully—the is never displayed. If $missing does exist, the second condition checks whether the $missing array contains the value name. If it does, the is displayed as shown in Figure 11-7.

7. Insert a similar warning for the comments field like this:

```
<label for="comments">Comments: <?php
if (isset($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.

8. Save feedback.php, and test the page again locally by entering nothing into any of the fields. The page should look like Figure 11-7. Check your code against feedback_05.php if you encounter any problems.

Figure 11-7. The PHP script displays alerts if required information is missing, even when JavaScript is disabled.

9. Try one more test. Open Code view, and amend the line that sends the email like this:

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

This temporarily sets the value of $mailSent to false and comments out the code that actually sends the email.

10. Reload `feedback.php` into your browser, and type something in the Name and Comments fields before clicking Send comments. This time you should see the message telling you there was a problem and asking you to try later.
11. Reverse the change you made in step 9 so that the code is ready to send the email.

Preserving user input when a form is incomplete

Imagine you have just 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.

When the page first loads or the email is successfully sent, 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. So, that's the key: if the `$missing` variable exists, you want the content of each field to be redisplayed. You can set default text for a text input field by setting the `value` attribute of the `<input>` tag.

At the moment, the `<input>` tag for `name` looks like this:

```
<input name="name" type="text" class="textInput" id="name" />
```

To add the `value` attribute, all you need is a conditional statement that checks whether `$missing` exists. If it does, you can use `echo` to display `value=""` and put the value held in `$_POST['name']` between the quotes. It sounds simple enough, but this is one of those situations where getting the right combination of quotes can drive you mad. It's made even worse by the fact that the user input in the text field might also contain quotes. Figure 11-8 shows what happens if you don't give quotes in user input special treatment. The browser finds the first matching quote and throws the rest of the input away.

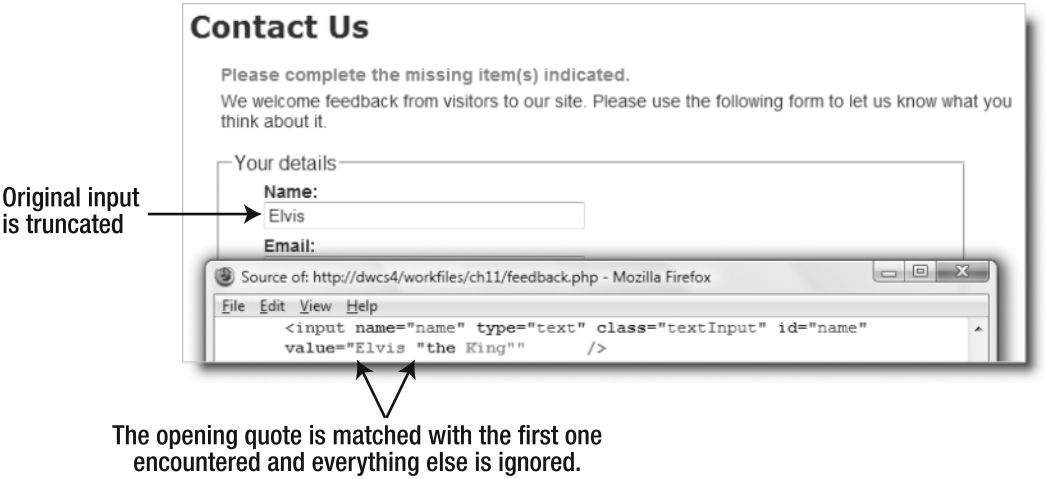


Figure 11-8. Quotes within user input need special treatment before form fields can be redisplayed.

You might be thinking that this is a case where magic quotes would be useful. Unfortunately, they won't work either. If you don't use the POST stripslashes snippet, this is what you get instead:

Magic quotes work only with input into a database (and not very well, either, which is why they are being phased out). The browser still sees the first matching quote as the end of the value attribute. The solution is simple: convert the quotes to the HTML entity equivalent ("), and PHP has a function called—appropriately—htmlentities(). Passing the $_POST array element to this function converts all characters (except space and single quote) that have an HTML entity equivalent to that entity. As a result, the content is no longer truncated. What's cool is that the HTML entity " is converted back to double quotes when the form is resubmitted, so there's no need for any further conversion.

The htmlentities() function was created in the days before widespread support for Unicode (UTF-8), so it uses Latin-1 or Western European encoding (ISO-8859-1) as its default. Since Dreamweaver uses UTF-8 as the default encoding for web pages, you need to pass an argument to htmlentities() to tell it to use the correct encoding. Unfortunately, to set the encoding argument, you need to pass a total of three arguments to htmlentities(): the string you want converted, a PHP constant describing how to handle quotes, and a string containing the encoding. Tables 11-1 and 11-2 list the available values for the second and third arguments.

Table 11-1. PHP constants for handling quotes in htmlentities()

Constant	Meaning
ENT_COMPAT	Converts double quotes to " but leaves single quotes alone. This is the default. You don't need to use this unless you also pass a third argument to htmlentities().
ENT_QUOTES	Converts double quotes to " and single quotes to '.
ENT_NOQUOTES	Leaves both double and single quotes alone.

Table 11-2. Character encodings supported by htmlentities()

Encoding	Aliases	Description
ISO-8859-1	ISO8859-1	Western European (Latin-1). This is the default and not normally required as an argument to htmlentities().
ISO-8859-15	ISO8859-15	Western European (Latin-9). Includes the euro symbol and characters used in French and Finnish.
UTF-8		Unicode, the default encoding in Dreamweaver CS4.

Continued

Table 11-2. *Continued*

Encoding	Aliases	Description
cp866	ibm866, 866	DOS-specific Cyrillic character set.
cp1251	Windows-1251, win-1251, 1251	Windows-specific Cyrillic character set.
cp1252	Windows-1252, 1252	Windows-specific character set for Western European.
KOI8-R	koi8-ru, koi8r	Russian.
GB2312	936	Simplified Chinese; national standard character set used in People's Republic of China.
BIG5	950	Traditional Chinese; mainly used in Taiwan.
BIG5-HKSCS		Big5 with Hong Kong extensions.
Shift_JIS	SJIS, 932	Japanese.
EUC-JP	EUCJP	Japanese.

If you are using the Dreamweaver default encoding, passing a value to `htmlentities()` involves using all three arguments like this:

```
htmlentities(value, ENT_COMPAT, 'UTF-8');
```

This converts double quotes but leaves single quotes alone. More importantly, it preserves accented characters and any other characters outside the Latin-1 character set.

If you are using a different encoding for your web pages or want quotes to be handled differently, substitute the appropriate values for the second and third arguments using Tables 11-1 and 11-2. The third argument must be a string, so it should be enclosed in quotes. The aliases in Table 11-2 are alternative spellings or names for the character encoding supported by PHP.

So, to summarize, the way you redisplay the user's input in the Name field if one or more required fields are missing is like this:

```
<input name="name" type="text" class="textInput" id="name"
<?php if (isset($missing)) {
  echo 'value="'.htmlentities($_POST['name'], ENT_COMPAT, 'UTF-8').'"';
} ?>
/>
```

This code is quite short, but the line inside the curly braces contains a tricky combination of quotes and periods. 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. You can break down the rest of the line into three sections, as follows:

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

The first section outputs `value="` as text and uses the concatenation operator (a period—see "Joining strings together" in Chapter 10) to join it to the next section, which passes `$_POST['name']` and the two arguments from Tables 11-1 and 11-2 to the `htmlentities()` function. The final section uses the concatenation operator again to join the next string, which consists solely of a double quote. So if `$missing` has been set and `$_POST['name']` contains `Joe`, you'll end up with this inside the `<input>` tag:

```
<input name="name" type="text" class="textInput" id="name" ➥
value="Joe" />
```

Whenever you need this code, the only thing that changes is the name of the `$_POST` array element. So, rather than laboriously type it out every time, it's a good idea to convert it into a snippet.

Saving frequently used code as a snippet

Although I have provided some snippets for you in the extension that you installed at the beginning of this chapter, it's easy enough to create snippets of your own. The following instructions show you how to turn the code described in the previous section into a snippet that leaves your cursor in the correct position to insert the name of the `$_POST` array element. You can use the same technique to create a snippet from any frequently used code.

1. Open the Snippets panel, right-click, and select New Folder from the context menu. Name the new folder PHP.
2. With the PHP folder selected in the Snippets panel, right-click, and select New Snippet from the context menu. This opens the Snippet dialog box shown in Figure 11-9.

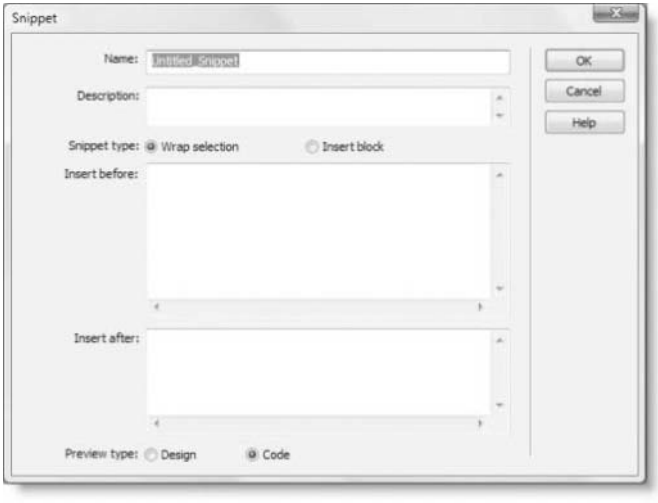


Figure 11-9.
Snippets are a useful way of storing short pieces of frequently used code.

The Snippets panel has the following options:

- Name: This is the name that appears in the Snippets panel. Choose a short but descriptive name. Spaces are permitted.
- Description: This is for a more detailed description. It appears alongside the name in the Snippets panel (you might need to expand the panel horizontally to see it). When you mouse over it, the full description appears as a tooltip.
- Snippet type: Wrap selection creates a wraparound snippet that leaves the cursor in the correct place to type in a value. Insert block inserts a single block of code.
- Insert before: This is the first section of code in a wraparound snippet. The cursor will be positioned immediately after the code you enter here. If you select the Insert block radio button, the Insert before and Insert after fields are merged into a single one labeled Insert code.
- Insert after: This applies only to a wraparound snippet. Enter the code you want to appear after the cursor.
- Preview type: This determines how the snippet is displayed in the preview pane at the top of the Snippets panel. Snippets can consist of HTML code. So, selecting Design shows the output rather than the underlying code in the preview pane.

3. Enter Sticky input value in the Name field.
4. Type a brief description in the Description field. I used this: Redisplays user input by inserting value attribute when $missing is set.
5. Select the Wrap selection radio button, and enter the following code in the Insert before field:

```
<?php if (isset($missing)) {
  echo 'value="' . htmlentities($_POST['
```

6. Enter the following code in the Insert after field:

```
'], ENT_COMPAT, 'UTF-8') . '"';
} ?>
```

7. Select the Code radio button for Preview type, and click OK to save the snippet.

That's all there is to creating a snippet. As well as typing in the code directly as you have done here, you can also select existing code in the Document window and choose Create New Snippet from the context menu. Dreamweaver automatically inserts the selected code in the Insert before field.

To edit a snippet, select it in the Snippets panel, right-click, and select Edit from the context menu.

Creating sticky form fields

Now that you have a new snippet to cut down on the coding, let's put it to work.

1. Insert a blank line just before the closing /> of the <input> tag for the Name text field like this:

```
<input name="name" type="text" class="textInput" id="name"

/>
```

2. With your cursor on the blank line, double-click Sticky input value in the Snippets panel. Alternatively, select the snippet, and click the Insert button at the bottom of the Snippets panel, or right-click and select Insert from the context menu.

   Dreamweaver should insert the code in the snippet and leave your cursor in the right position to type the name of the $_POST array element, as shown here:

```
89        <input name="name" type="text" class="textInput" id="name"
90        <?php if (isset($missing)) {
91    echo 'value="' . htmlentities($_POST['], ENT_COMPAT, 'UTF-8') . '"';
92  } ?>
93        />
```

> *In my testing, I found that Dreamweaver inserted the cursor after the pair of single quotes instead of between them. The only way I could correct this problem was by adding a space before the closing quote in the* Insert after *field of the* Snippets *dialog box (see Figure 11-9). I decided that moving the cursor one character to the left with my arrow keys was still a lot easier than typing the full code every time.*

3. Make sure your cursor is between the single quotes of $_POST[''], and enter name so that it reads $_POST['name'].
4. Repeat steps 1–3 to amend the email input field in the same way, entering email instead of name between the quotes of $_POST['']
5. 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="45" rows="5"><?php
  if (isset($missing)) {
    echo htmlentities($_POST['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 in the text area.

6. Save `feedback.php`, and test the page. If the first test message earlier in the chapter was successful, you can upload it to your remote server. If any required fields are omitted, the form displays the original content along with any error messages. However, if the form is correctly filled in, the email is sent, an acknowledgment is displayed, and the input fields are cleared.

   If your remote server test didn't succeed earlier in the chapter, just test locally. You'll probably get a PHP error message if all required fields are filled in, but that's nothing to worry about. We're almost at the stage to get your remote server working.

   You can check your code with `feedback_06.php`.

You might want to save the PHP code inserted in step 5 as another snippet. The easiest way is to highlight the whole section of PHP code, right-click, and select New Snippet from the context menu. You can then cut and paste the code inside the Snippets panel to split it between the Insert before and Insert after fields.

Filtering out potential attacks

A particularly nasty exploit known as **email header injection** emerged in mid-2005. It 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:", because these are email headers that the attacker injects into your script in an attempt 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.

Blocking emails that contain specific phrases

In this section, we'll create a pattern to check for suspect phrases and pass the form input to a custom-built function that checks for any matches. The function is one of the snippets that you installed earlier in the chapter, so the most complex part of the coding is already done for you. If a match is found, a conditional statement prevents the email from being sent.

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. Insert the code for both of these just above the section that processes the `$_POST` variables:

```
// create empty array for any missing fields
$missing = array();

// assume that there is nothing suspect
$suspect = false;
```

```
// create a pattern to locate suspect phrases
$pattern = '/Content-Type:|Bcc:|Cc:/i';

// process the $_POST variables
```

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.

2. You can now use `$pattern` to filter out any suspect user input from the `$_POST` array. At the moment, each element of the `$_POST` array contains only a string. However, multiple-choice form elements, such as checkboxes, return an array of results. So, you need to tunnel down any subarrays and check the content of each element separately. In the snippets collection you installed earlier in the chapter, you'll find a custom-built function to do precisely that.

Insert two blank lines immediately after the `$pattern` variable from step 1. Then open the Snippets panel, and double-click Suspect pattern filter in the PHP-DWCS4 folder to insert the code shown here in bold:

```
// create a pattern to locate suspect phrases
$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;
    }
  }
}
```

3. I won't go into detail about how this code works. All you need to know is that calling the `isSuspect()` function is very easy. You just pass it three values: the `$_POST` array, the pattern, and the `$suspect` Boolean variable. Insert the following code immediately after the code in the previous step:

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

4. If any suspect phrases are detected, the value of `$suspect` changes to `true`, so you need to set `$mailSent` to `false` and delete the `$missing` array to prevent the email from being sent and to display an appropriate message in the form. There's also no

point in processing the $_POST array any further. Wrap the code that processes the $_POST variables in the second half of an `if . . . else` statement like this:

```
if ($suspect) {
  $mailSent = false;
  unset($missing);
} else {
  // process the $_POST variables
  foreach ($_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, $required)) {
      array_push($missing, $key);
      }
    // otherwise, assign to a variable of the same name as $key
    elseif (in_array($key, $expected)) {
      ${$key} = $temp;
      }
    }
  }
```

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

5. If suspect content is detected, you don't want the code that builds and sends the email to run, so amend the condition in the opening `if` statement like this:

```
// go ahead only if not suspect and all required fields OK
if (!$suspect && empty($missing)) {
  // build the message
```

6. Save feedback.php, and check your code against feedback_07.php.

Because the `if` statement in step 4 sets `$mailSent` to `false` and unsets `$missing` if it detects any suspect pattern, the code in the main body of the page displays the same message that's displayed if there's a genuine problem with the server. A neutral message reveals nothing that might assist an attacker. It also avoids offending anyone who may have innocently used a suspect phrase.

You can use `isSuspect()` with any array or pattern, but it always requires the following three arguments:

- An array that you want to filter. If the array contains other arrays, the function burrows down until it finds a simple value against which it can match the pattern.
- A regular expression containing the pattern(s) you want to search for. There are two types of regular expression, Perl-compatible regular expression (PCRE) and Portable Operating System Interface (POSIX). You must use a PCRE. This function won't work with a POSIX regular expression. A good online source is `http://regexlib.com`.
- A Boolean variable set to `false`. If the pattern is found, the value is switched to `true`.

Safely including the user's address in email headers

Up to now, I've avoided using one of the most useful features of the PHP `mail()` function: the ability to add extra email headers with the optional fourth argument. A popular use of extra headers 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 convenient, but it provides a wide open door for an attacker to supply a spurious set of headers. With the `isSuspect()` function in place, you can block attacks and safely use the fourth argument with the `mail()` function.

The most important header you should add is `From`. Email sent by `mail()` is often identified as coming from `nobody@`*`servername`*. Adding the `From` header not only identifies your mail in a more user-friendly way, but it also solves the problem you might have encountered on the first test of there being no setting for `sendmail_from` in `php.ini`.

You can find a full list of email headers at `http://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 (often essential for languages other than Western European ones).

Like the body of the email message, headers must be passed to the `mail()` function as a single string. Each new header, except the final one, must be on a separate line terminated by a carriage return and newline character. This means using the `\r` and `\n` escape sequences in double-quoted strings.

Let's say you want to send copies of messages to other departments, plus a copy to another address that you don't want the others to see. This is how you pass those additional email headers to `mail()`:

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

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

The default encoding for email is `iso-8859-1` (English and Western European). If you want to use a different encoding, set the `Content-Type` header. Dreamweaver uses Unicode (`UTF-8`) as its default, so you need to add a header like this:

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

The web page that the form is embedded in must use the same encoding (usually set in a `<meta>` tag). The preceding code assumes other headers will follow. If it's the final header, omit the `\r\n` sequence at the end of the line.

Hard-coded additional headers present no security risk, but anything that comes from user input must be filtered before it's used.

Adding email headers and automating the reply address

This section incorporates the user's email address into a `Reply-To` header. Although `isSuspect()` should sanitize user input, it's worth subjecting the email field to a more rigorous check to make sure that it doesn't contain illegal characters or more than one address.

1. At the moment, the `$required` array doesn't include `email`, and you may be happy to leave it that way. So, to keep the validation routine flexible, it makes more sense to handle the email address outside the main loop that processes the `$_POST` array.
   - If `email` is required but has been left blank, the loop will have already added `email` to the `$missing` array, so the message won't get sent anyway.
   - If it's not a required field, you need to check `$email` only if it contains something. So, you need to wrap the validation code in an `if` statement that uses `!empty()`.

   Insert the code shown in bold after the loop that processes the `$_POST` array.

```
    // otherwise, assign to a variable of the same name as $key
    elseif (in_array($key, $expected)) {
      ${$key} = $temp;
    }
  }
}

// validate the email address
if (!empty($email)) {

}

// go ahead only if not suspect and all required fields OK
if (!$suspect && empty($missing)) {
```

2. Position your cursor on the blank line between the curly braces of the conditional statement you have just inserted. Open the Snippets panel, and double-click Check email PCRE in the PHP-DWCS4 folder. This inserts the following regular expression:

```
$checkEmail = '/^[^@]+@[^\s\r\n\'";,@%]+$/';
```

   Designing a regular expression to recognize a valid-looking email address is notoriously difficult. So, instead of striving for perfection, `$checkEmail`, takes a negative approach by rejecting characters that are illegal in an email address. However, to make sure that the input resembles an email address in some way, it checks for an `@` mark surrounded by at least one character on either side.

3. Now add the code shown in bold to check `$email` against the regular expression:

```
// validate the email address
if (!empty($email)) {
  // regex to ensure no illegal characters in email address
  $checkEmail = '/^[^@]+@[^\s\r\n\'";,@%]+$/';
```

```
  // reject the email address if it doesn't match
  if (!preg_match($checkEmail, $email)) {
    $suspect = true;
    $mailSent = false;
    unset($missing);
  }
}
```

The conditional statement uses the `preg_match()` function, which takes two arguments: a PCRE and the string you want to check. If a match is found, the function returns `true`. Since it's preceded by the negative operator, the condition is `true` if the contents of `$email` *don't* match the PCRE.

If there's no match, `$suspect` is set to `true`, `$mailSent` is set to `false`, and `$missing` is unset. This results in the neutral alert saying that the message can't be sent and clears the form. This runs the risk that someone who has accidentally mistyped the email address will be forced to enter everything again. If you don't want that to happen, you can omit `unset($missing);`. However, the PCRE detects illegal characters that are unlikely to be used by accident, so I have left it in.

> *Many popular PHP scripts use pattern-matching functions that begin with* `ereg`*. These work only with POSIX regular expressions. I recommend you always use the PCRE functions that begin with* `preg_`*. Not only is PCRE more efficient, support for the* `ereg` *family of functions has been removed from PHP 6.*

4. Now add the additional headers to the email. Place them immediately above the call to the `mail()` function, and add `$headers` as the fourth argument like this:

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

// create additional headers
$headers = "From: Essential Guide<feedback@example.com>\r\n";
$headers .= 'Content-Type: text/plain; charset=utf-8';
if (!empty($email)) {
  $headers .= "\r\nReply-To: $email";
}

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

Use your own email address in the first header, rather than the dummy one shown here.

The second header assumes you are using the Dreamweaver default encoding. If you are using a different character encoding on your page, you need to change `charset=utf-8` to the appropriate value for your character set. You can find the correct value by inspecting the `Content-Type` `<meta>` tag in the `<head>` of your web page.

If you don't want email to be a required field, there's no point in using a nonexistent value in the Reply-To header, so I have wrapped it in a conditional statement. Since you have no way of telling whether the Reply-To header will be created, it makes sense to put the carriage return and newline characters at the beginning of the second header. It doesn't matter whether you put them at the end of one header or the start of the next one, as long as a carriage return and newline character separate each header. For instance, if you wanted to add a Cc header, you could do it like this:

```
$headers = "From: Essential Guide<feedback@example.com>\r\n";
$headers .= "Content-Type: text/plain; charset=utf-8\r\n";
$headers .= 'Cc: admin@example.com';
if (!empty($email)) {
  $headers .= "\r\nReply-To: $email";
}
```

Or like this:

```
$headers = "From: Essential Guide<feedback@example.com>\r\n";
$headers .= 'Content-Type: text/plain; charset=utf-8';
$headers .= "\r\nCc: admin@example.com";
if (!empty($email)) {
  $headers .= "\r\nReply-To: $email";
}
```

5. Save feedback.php, upload it to your remote server, and test the form. When you receive the email, click the Reply button in your email program, and you should see the address that you entered in the form automatically entered in the recipient's address field. You can check your code against feedback_08.php.

> *When building your own forms, don't forget to add the* name *of each text field to the* $expected *array. Also add the* name *of required fields to the* $required *array, and add a suitable alert as described in "Checking required fields."*

What if you still don't get an email?

For security reasons, some hosting companies require a fifth argument to mail(). Normally, it takes the form of a string comprised of -f followed by your email address like this:

```
'-fdavid@example.com'
```

Add it to the line of code that sends the mail like this:

```
$mailSent = mail($to,$subject,$message,$headers,'-fdavid@example.com');
```

If using this fifth argument does not work, ask your hosting company for a sample script for sending email. Some companies tell you to use ini_set() to adjust a setting called sendmail_from. The $headers in the previous section should avoid the need to do this.

However, if you still get an error message about sendmail_from, amend the preceding code like this (use your own email address instead of david@example.com):

```
ini_set('sendmail_from', 'david@example.com');
$mailSent = mail($to,$subject,$message,$headers,'-fdavid@example.com');
```

Handling multiple-choice form elements

You now have the basic knowledge to process text input from an online form and email it to your inbox. The principle behind handling multiple-choice elements is exactly the same: the name attribute is used as the key in the $_POST array. However, as you saw in Chapter 9, checkboxes and multiple-choice lists don't appear in the $_POST array if nothing has been selected, so they require different treatment.

The following exercises show you how to handle each type of multiple-choice element. If you're feeling punch drunk at this stage, come back later to study how to handle multiple-choice elements when you need to incorporate them into a script of your own.

Getting data from checkboxes

In Chapter 9, I showed you how to create a checkbox group, which stores all checked values in a subarray of the $_POST array. However, the subarray isn't even created if all boxes are left unchecked. So, you need to use isset() to check the existence of the subarray before attempting to process it.

1. Add the name of the checkbox group to the $expected array like this:

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

   In the form, interests is followed by square brackets like this:

   ```
   <input type="checkbox" name="interests[]" . . .
   ```

   The square brackets in the form tell the $_POST array to store all checked values in a subarray called $_POST['interests']. However, *don't* add square brackets to interests in the $expected array. Doing so would bury the checked values in a subarray one level deeper than you want. See "Using arrays to store multiple values" in Chapter 10 for a reminder of how arrays are created.

2. If you want the checkboxes to be required, add the name of the checkbox group to the $required array in the same way.

3. Because the checkbox array might never be created, you need to set a default value before processing the $_POST variables. You need to do this even if you're not making the checkbox group required, because it affects the way the message is built. The following code in bold goes after the $missing array is initialized:

   ```
   // create empty array for any missing fields
   $missing = array();
   // set default values for variables that might not exist
   if (!isset($_POST['interests'])) {
     $_POST['interests'] = array();
   }
   ```

This uses a conditional statement to check whether `$_POST['interests']` has been set. If it hasn't, it's initialized as an empty array. This will trigger the code that processes the `$_POST` variables to add `interests` to the `$missing` array if no checkbox has been selected.

4. If you want more than one checkbox to be required, you need to add another test immediately after the code in the previous step like this:

```
// minimum number of required checkboxes
$minCheckboxes = 2;
// if fewer than required add to $missing array
if (count($_POST['interests']) < $minCheckboxes) {
  $missing[] = 'interests';
}
```

This sets a variable containing the minimum number of required checkboxes (I'm using a variable so the number can be reused in the error message) and then compares it with the number of elements in `$_POST['interests']`. The `count()` function, as you might expect, counts the number of elements in an array.

5. To extract the values of the checkbox array, you can use the oddly named `implode()` function, which joins array elements. It takes two arguments: a string to be used as a separator and the array. So, `implode(', ', $interests)` joins the elements of `$interests` as a comma-separated string. Add the following code shown in bold to the script that builds the body of the email:

```
$message .= "Comments: $comments\r\n\r\n";
$message .= 'Interests: '.implode(', ', $interests);
```

Note that I added two newline characters at the end of the line that adds the user's comments to the email. On the following line, I put `Interests:` in single quotes because there are no variables to be processed, and I used the concatenation operator to join the result of `implode(', ', $interests)` to the end of the email message. You cannot include a function inside a string.

6. If you have made the checkbox group required, add an alert like this:

```
<p><strong>What aspects of London most interest you?</strong>
<?php  if (isset($missing) && in_array('interests', $missing)) { ?>
<span class="warning">Please choose at least
<?php echo $minCheckboxes; ?></span><?php } ?>
</p>
```

This assumes you have set a value for `$minCheckboxes` in step 4. If you want only one checkbox selected, you can replace `<?php echo $minCheckboxes; ?>` with the word "one" inside the `<span>`.

7. The next listing shows the code for the first two checkboxes in the body of the page. The code in bold preserves the user's checkbox selections if any required field is missing.

```
<label>
<input type="checkbox" name="interests[]" value="Classical concerts" ➥
id="interests_0"
```

```
<?php
if (isset($missing) && in_array('Classical concerts', ➥
$_POST['interests'])) {
  echo 'checked="checked"';
} ?>
/>Classical concerts</label>
<br />
<label>
<input type="checkbox" name="interests[]" value="Rock/pop" ➥
id="interests_1"
<?php
if (isset($missing) && in_array('Rock/pop', $_POST['interests'])) {
  echo 'checked="checked"';
} ?>
/>Rock & pop events</label>
```

The PHP code for each checkbox tests whether the `$missing` variable exists and whether the value of the checkbox is in the `$_POST['interests']` subarray. If both are `true`, `echo` inserts `checked="checked"` into the `<input>` tag. (If you're using HTML instead of XHTML, use just `checked`.) Although it looks like a lot of hand-coding, you can copy and paste the code after creating the first one. Just change the first argument of `in_array()` to the `value` of the checkbox. The complete code is in `feedback_09.php`.

Getting data from radio button groups

Radio button groups allow you to pick only one value. This makes it easy to retrieve the selected one. All buttons in the same group must share the same `name` attribute, so the `$_POST` array contains the `value` attribute of whichever radio button is selected. However, if you don't set a default button in your form, the radio button group's `$_POST` array element remains unset.

1. Add the name of the radio button group to the `$expected` array.
2. If you haven't set a default button and you want a choice to be compulsory, also add it to the `$required` array. This isn't necessary if a default choice is set in the form.
3. If you haven't set a default button, you need to set a default value before building the body of the email message. You do this in a similar way to a checkbox group, but since a radio button group can have only one value, you set the default as an empty string, not an array, as shown in this example:

   ```
   if (!isset($_POST['radioGroup'])) {
     $_POST['radioGroup'] = '';
   }
   ```

4. Add the value of the radio button group to the body of the message like this:

   ```
   $message .= 'Interests: '.implode(', ', $interests)."\r\n\r\n";
   $message .= "Subscribe: $subscribe";
   ```

5. Assuming a default button has been defined, amend the radio button group like this:

```
<label>
  <input type="radio" name="subscribe" id="subscribeYes" value="y"
  <?php
  if (isset($missing) && $_POST['subscribe'] == 'y') {
    echo 'checked="checked"';
  } ?>
  />
Yes</label>
<label>
  <input name="subscribe" type="radio" id="subscribe-no" value="n"
  <?php
  if (!$_POST || isset($missing) && $_POST['subscribe'] == 'n') {
    echo 'checked="checked"';
  } ?>
  />
No</label>
```

   The conditional statement for the default radio button begins with `!$_POST ||`, which means "if the `$_POST` array is empty *or* . . ." So, if the form hasn't been submitted or if the user has selected No and the form is incomplete, this button will be checked.

   The completed script is in `feedback_10.php`.

You need to add a required alert only if no default has been defined in the original form.

Getting data from a drop-down menu

Drop-down menus created with the `<select>` tag normally allow the user to pick only one option from several. One item is always selected, even if it's only the first one inviting the user to select one of the others. Setting the `value` of this first `<option>` to `0` has the advantage that the `empty()` function, which is used to check required fields, returns `true` when `0` is passed to it either as a number or string.

1. Add the name of the drop-down menu to the `$expected` array. Also add it to the `$required` array if you want a choice to be compulsory.
2. Add the value of the drop-down menu to the email message like this:

```
$message .= "Subscribe: $subscribe\r\n\r\n";
$message .= "Visited: $visited";
```

   One option will always be selected, so this doesn't need special treatment. However, change the `value` of the first `<option>` tag in the menu to `No response` if it isn't a required field. Leave it as `0` if you want the user to make a selection.
3. The following code shows the first two items of the drop-down menu in `feedback.php`. The PHP code highlighted in bold assumes that the menu has been made a required field and resets the selected option if an incomplete form is

submitted. 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 it.

```
<label for="visited">How often have you been to London? <?php
if (isset($missing) && in_array('visited', $missing)) { ?>
  <span class="warning">Please select a value</span><?php } ?></label>
  <select name="visited" id="visited">
    <option value="0"
    <?php
    if (!$_POST || $_POST['visited'] == '0') {
      echo 'selected="selected"';
    } ?>
    >-- Select one --</option>
    <option value="Never"
    <?php
    if (isset($missing) && $_POST['visited'] == 'Never') {
      echo 'selected="selected"';
    } ?>
    >Never been</option>
  . . .
  </select>
```

When setting the second condition for each <option>, it's vital that you use the same spelling and mixture of uppercase and lowercase as contained in the value attribute. PHP is case-sensitive and won't match the two values if there are any differences.

The finished code is in feedback_11.php.

Getting data from a multiple-choice list

Multiple-choice lists are similar to checkboxes: they allow the user to choose zero or more items, so the result is stored in an array. If no items are selected, the $_POST array contains no reference to the list, so you need to take that into consideration both in the form and when processing the message.

1. Add the name of the multiple-choice list to the $expected array. Also add it to the $required array if you want a choice to be compulsory.
2. Set a default value for a multiple-choice list in the same way as for an array of checkboxes.

```
// set default values for variables that might not exist
if (!isset($_POST['interests'])) {
  $_POST['interests'] = array();
}
if (!isset($_POST['views'])) {
  $_POST['views'] = array();
}
```

3. When building the body of the message, use `implode()` to create a comma-separated string, and add it to the message like this:

```
$message .= "Visited: $visited\r\n\r\n";
$message .= 'Impressions of London: '.implode(', ', $views);
```

4. The following code shows the first two items from the multiple-choice list in `feedback.php`. The code works in an identical way to the checkboxes, except that you echo `'selected="selected"'` instead of `'checked="checked"'`. It also assumes you have made at least one selection required.

```
<label for="views">What image do you have of London? <?php
if (isset($missing) && in_array('views', $missing)) { ?>
  <span class="warning">Please select a value</span><?php } ?>
</label>
<select name="views[]" size="6" multiple="multiple" id="views">
  <option value="Vibrant/exciting"
  <?php
  if (isset($missing) && in_array('Vibrant/exciting', ➥
$_POST['views'])) {
    echo 'selected="selected"';
  } ?>
  >A vibrant, exciting city</option>
  <option value="Good food"
  <?php
  if (isset($missing) && in_array('Good food', $_POST['views'])) {
    echo 'selected="selected"';
  } ?>
  >A great place to eat</option>
  . . .
</select>
```

The completed code is in `feedback_12.php`.

If you want to make more than one item in the multiple-choice list required, create a variable for the minimum number of items, and use `count()` to add the name of the multiple-choice list to the `$missing` array in the same way as in step 4 of "Getting data from checkboxes."

Redirecting to another page

Everything has been kept within the same page, even if the message is sent successfully. To redirect the visitor to a different page, change the code at the end of the message-processing section like this:

```
// send it
$mailSent = mail($to, $subject, $message, $headers);
if ($mailSent) {
```

```
            // redirect the page with a fully qualified URL
            header('Location: http://www.example.com/thanks.php');
            exit;
        }
    }
}
```

The HTTP/1.1 protocol stipulates a fully qualified URL for a redirect command, although most browsers will perform the redirect correctly with a relative pathname.

When using the `header()` function, you must be careful that no output is sent to the browser before PHP attempts to call it. If, when testing your page, you see an error message warning you that headers have already been sent, check that there are no characters, including newline characters, spaces, or tabs, ahead of the opening PHP tag.

Blocking submission by spam bots

The battle against spam is never ending. The filters used in this chapter should prevent turning your form into a spam relay, but they won't stop spammers from using the form to send unwanted mail to your inbox. Most spam is sent by automated bots, so several techniques have been developed to try to prevent forms from being submitted automatically.

Using a CAPTCHA

One of the most common methods of combating spam is to use a CAPTCHA (Completely Automated Public Turing Test to Tell Computers and Humans Apart). This requires the user to decipher distorted text and type it into a field before submitting the form. Figure 11-10 shows a typical example. The idea is that such text is easy for humans to read but that it's beyond the capability of current computer programs.

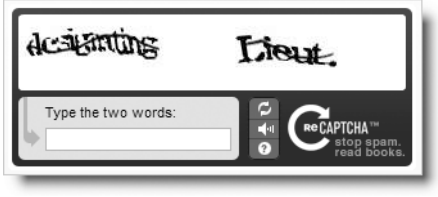


Figure 11-10.
If a CAPTCHA is hard to read, it deters humans just as much as spam bots.

The problem with using a CAPTCHA is that, for the text to defeat optical character recognition, it needs to be difficult to read. I have pretty good eyes, but I found the first word in Figure 11-10 difficult to make out. If it weren't a real word, I would have difficulty guessing some of the letters. For anyone who is not a native speaker of English or with poor eyesight, it would be a major challenge. This particular example also has an audio button, but—for me at least—the audio test was even more difficult. It also assumes that the user has an audio player and speakers installed.

In theory, CAPTCHA is a good idea, but its major drawback lies in the need to make the test too difficult for computer programs to solve. As a result, it becomes more difficult for humans. So, there's a danger that using a CAPTCHA will prevent not only the spammers but also the people you want to use the form.

You can learn more about CAPTCHA at `http://www.captcha.net`.

Using a question in plain text

A simpler form of CAPTCHA asks a question in plain text, for example "What is the sum of three plus four?" If the answer is entered in a text input field called `test`, you could add the following code to the script in `feedback.php` after the `$missing` array is initialized:

```
if (empty($_POST['test']) || $_POST['test'] != 7 || ➥
strtolower($_POST['test']) != 'seven') {
  $missing[] = 'test';
}
```

This checks whether any value has been entered in the `test` field and whether its value gives the right answer either as a number or in words (`strtolower()` converts the answer to lowercase, so any combination of uppercase and lowercase is acceptable). If the field hasn't been filled in or the answer is wrong, `test` is added to the `$missing` array, preventing the mail from being processed.

The problem with this sort of solution is that its simplicity makes it relatively easy to break. It could be strengthened by rotating the questions on a random basis, but that makes the code much more complex.

Using a honeypot

Another technique is based on the principle that just like bears can't resist honey, spam bots can't resist filling in every field they find. A honeypot is a form field that's hidden from view in normal browsers, so it shouldn't contain any input. Give the field a `name` attribute that a spammer is likely to want to fill in, and give its surrounding paragraph an ID that gives no indication that you're using it as a honeypot, for example:

```
<p id="website">
  <label for="url">Website: </label>
  <input type="text" name="url" id="url" />
</p>
```

In your style sheet, create a style rule for the surrounding paragraph's ID like this:

```
#website {
  display: none;
}
```

In the form processing script, check whether `$_POST['url']` contains a value. If it does, the form has almost certainly been filled in by a spam bot, so can be rejected. If you want to check that genuine form submissions aren't rejected by mistake, change the address and subject line of the email like this:

```
if (!empty($_POST['url'])) {
  $to = 'spamtrap@example.com';
  $subject = 'Suspected bot submission from feedback form';
} else {
  $to = 'me@example.com'; // use your own email address
  $subject = 'Feedback from Essential Guide';
}
```

On the other hand, if you want to dump all suspect submissions, set `$suspect` to true when `$_POST['url']` contains a value like this:

```
// check the $_POST array and any subarrays for suspect content
isSuspect($_POST, $pattern, $suspect);
if (!empty($_POST['url'])) {
  $suspect = true;
}
```

No doubt, spammers will get wise to the existence of honeypots, but this technique appears to be reasonably successful at the time of this writing.

Chapter review

If that was your first encounter with PHP, your head will probably be reeling. This has been a tough chapter. In the next chapter, you'll adapt this script so that it can be reused as an external file with most forms. The external file never changes, and the hand-coding is cut down to about a dozen lines. I could, of course, have given you the external file without explanation, but if you don't understand the code, you can't adapt it to your own requirements. Even if you never write an original PHP script of your own, you should know what the code in your page is doing. If you don't, you're storing up trouble for the future.

What makes PHP pages dynamic—and so powerful—is the fact that your code makes decisions, even though you have no way of knowing in advance what is going to be input into the form. The Dreamweaver code that you'll encounter in subsequent chapters tries to anticipate a lot of these unknown factors, but its beauty lies in the fact that it's configurable. If you know how to hand-code, you can get Dreamweaver to do a lot of the hard work for you and then take it beyond the basics.

However, it's no fun spending all your time churning out code. Life becomes simpler if you can reuse code. So, that's what the next chapter is about—saving time with PHP includes.