Listing 10.1.2 **Continued**

```
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>What city/town do you hail from? <input name="city" type="text" /></p>
<p><input type="submit" /></p>
</form>
```

One additional superglobal is created in regards to this data. The `$_REQUEST` variable is created and contains all the data from both the `GET` and `POST`. (It also contains cookie data.) This can be a handy shortcut when you don't know which of the two methods are going to provide the data you expect. Unless modified by the `variables_order` initialization directive, `POST` data overrides `GET` data in `$_REQUEST`.

In normal use, it is usually preferable to state whether you want the `GET` or `POST` data specifically. This makes for more robust code, and no unintended side effects can occur because you ended up with a different value for the variable than you expected. Also, you can sometimes do interesting things using both `GET` and `POST` data separately, as Section 10.3, "Using `GET` and `POST` Form Data Together," discusses later in this chapter.

10.2 Obtaining Multidimensional Arrays of Form Data

What makes the automatic form data handling described in the preceding section, "Easily Obtaining Form Data," even more powerful, is PHP's capability to understand multidimensional data coming from a form. By naming form fields with PHP-styled array constructors, PHP actually creates them as arrays for you.

So having a set of fields all with the same name: '`comment[]`', for example, causes an array indexed by the string `comment` to be created, as shown in Listing 10.2.1.

Listing 10.2.1 **Retrieving Multidimensional Data from a Form**

```
<?php
// If we have a 'cities' from the POST, then loop over and display:
if (isset($_POST['cities'])) {
    echo "<p>You claim to have lived in the following cities:";

    // Loop over each city and output
    foreach ($_POST['cities'] as $num => $val) {
        echo '<br />', ($num + 1), ". {$val}\n";
    }
    echo "</p>\n";
}
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>In order, what are the most recent cities you have lived in?</p>
<ol>
<?php
```

Listing 10.2.1 **Continued**

```
// Create a dozen form entries to hold cities:
echo str_repeat('<li><input name="cities[]" type="text" /></li>', 12);
?>
</ol>
<p><input type="submit" /></p>
</form>
```

In this example, there will always be a dozen entries for the cities because any entries that you leave blank still get submitted as a blank string. Therefore, if someone enters data into the seventh position alone, it still comes back in the seventh position.

This is useful for processing large amounts of similar data. It is also possible to specify the array index, instead of just using `[]` to have it automatically created. This means that you could, for example, use `address[state]`, and it would return to you as such (see Listing 10.2.2).

Listing 10.2.2 **Multidimensional Form Data Using Explicit Indexes**

```
<?php
// If we have a POST, then:
if (count($_POST)) {
    echo "
<p>Your address is:<br />
{$_POST['address']['line1']}<br />
{$_POST['address']['line2']}<br />
{$_POST['address']['line3']}<br />
{$_POST['city']}, {$_POST['state']}  {$_POST['zip']}
</p>
";
}
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>Please provide your address:<br />
Line 1: <input name="address[line1]" type="text" /><br />
Line 2: <input name="address[line2]" type="text" /><br />
Line 3: <input name="address[line3]" type="text" /><br />
City: <input name="city" type="text" /><br />
State: <input name="state" type="text" /><br />
Zip: <input name="zip" type="text" />
</p>
<p><input type="submit" /></p>
</form>
```

Finally, one other case allows for you to receive arrays full of data, and that refers to multiple select fields. In XHTML you can create a multiple select field that allows the user to select any number of items from it. To access these in PHP, again give them a name ending in `[]`, and all the selections that are made will arrive as an array (see Listing 10.2.3).

Listing 10.2.3 **Retrieving Multiselect Form Input**

```
<?php
// If we have a POST, then:
if (isset($_POST['beer'])) {
    // Output all beers that they liked:
    $beers = implode(', ', $_POST['beer']);
    echo "<p>Wow, you like the following beers: {$beers}</p>\n";
}
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>What are your favorite types of beer?</p>
<select name="beer[]" multiple="multiple">
  <option value="stout">Stout</option>
  <option value="porter">Porter</option>
  <option value="hefeweizen">Hefeweizen</option>
  <option value="brown ale">Brown Ale</option>
  <option value="lambic">Lambic</option>
  <option value="double bock">Double Bock</option>
</select>
<input type="submit">
</form>
```

As can be seen, with all these variations available, PHP has an amazingly powerful, flexible, and yet simple mechanism to get form data into the script.

10.3 Using `GET` and `POST` Form Data Together

Because `GET` and `POST` variables are both passed in separately, you can use them both at the same time, even having the same variable names in both cases. The most common practice of this is to pass some configuration variables to your script via `GET`, while allowing user input to come as a `POST`.

This can be especially useful in some situations because `GET` parameters are saved in bookmarks, and `POST` parameters are not. Therefore you can place a few items in the `GET` string that will affect whether someone makes a bookmark, giving you the information you need to display the page properly. This is demonstrated in Listing 10.3.1, where we use the `GET` string to indicate that the page has been posted but send the user's data via a `POST`. This way if someone bookmarks the "posted" page, the user will be presented a custom error if he returns to the page.

Listing 10.3.1 **Using** `GET` **and** `POST` **Data Simultaneously**

```
<?php
// If we have a 'GET' parameter called state and it is set to 'save'
if (isset($_GET['state']) && ($_GET['state'] == 'save')) {
    // Check if the POST data is blank:
    if (count($_POST) == 0) {
        // Looks like a bookmark, the GET existed but no post:
        die("ERROR:  This page has been improperly accessed.");
    }

    // Otherwise, output some data:
    echo "<p>You claim to live in state: {$_POST['state']}</p>\n";
    echo "<p>The script in is the '{$_GET['state']}' state.</p>\n";
}
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>?state=save"
  method="post" name="f1">
<p>In what state do you live? <input name="state" type="text" /></p>
<p><input type="submit" /></p>
</form>
```

10.4 Accepting Uploaded Files

PHP also provides a straightforward manner in which to handle file uploads as well. Anytime that you have a file upload field on your web page, the `$_FILES` array is automatically filled in for you with various pieces of data. For example, if you had named your upload field `'attachment'`, the following data would exist:

- `$_FILES['attachment']['name']`—The original name of the file from the user's machine.
- `$_FILES['attachment']['type']`—The mime type of the file (for example: 'text/plain'), but only if the browser provided the type.
- `$_FILES['attachment']['size']`—The size of the file in bytes.
- `$_FILES['attachment']['tmp_name']`—The full filename of the uploaded file.
- `$_FILES['attachment']['error']`—An error code, if there were any problems with the upload.

Using this capability, Listing 10.4.1 accepts an uploaded file and saves it in the same directory as itself, using its original name from the user's machine.

Listing 10.4.1 **File Uploading**

```
<?php
// If we had any files
if (count($_FILES)) {
    // Doublecheck that we really had a file:
    if (!($_FILES['attachment']['size'])) {
        echo "<p>ERROR:  No actual file uploaded</p>\n";
    } else {
        // Determine the filename to which we want to save this file:
        $newname = dirname(__FILE__) . '/' .
                basename($_FILES['attachment']['name']);

        // Attempt to move the uploaded file to it's new home:
        if (!(move_uploaded_file($_FILES['attachment']['tmp_name'],
                $newname))) {
            echo "<p>ERROR:  A problem occurred during file upload!</p>\n";
        } else {
            // It worked!
            echo "<p>Done!  The file has been saved as: {$newname}</p>\n";
        }
    }
}
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post"
    enctype="multipart/form-data" name="f1">
<input type="hidden" name="MAX_FILE_SIZE" value="8388608" />
<p>Why don't you upload a file? <input type="file" name="attachment" /></p>
<p><input type="submit" /></p>
</form>
```

The function `move_uploaded_file()` does a lot. Not only does it take care of moving the file to where we really want it to live, but it also performs some sanity checks on the uploaded file to make sure that a hijacking attempt has not occurred. You still need to check that the size of the file is not zero to determine whether a file was really uploaded. If someone submits a filename that doesn't actually exist on her machine, everything will look like a file was uploaded, except that the file size will be zero.

One thing to note is the "special" hidden form field named `MAX_FILE_SIZE`. This field serves two purposes. First, some web browsers actually pick up on this field and will not allow the user to upload a file bigger than this number (in bytes). This makes for a good user interface because if you are planning on ignoring file uploads that are bigger than a certain size anyway, setting this means that the user doesn't try to sit through a long upload only to find out that it was too big.

Second, PHP itself looks for this value, and it ignores all files bigger than it—setting an error code of `UPLOAD_ERR_FORM_SIZE` if the file is bigger. Granted, this can be fooled easily by manipulating the form data, so if you really want to make sure that no larger of a file is uploaded, you need to check the file size yourself as well before moving it. At the very least, you should set this value to coincide with the maximum upload size that is set in your PHP's configuration file. It is set with the `upload_max_filesize` directive and the default is 8MB, or 8388608 bytes.

In addition to making sure that the PHP directive `upload_max_filesize` and the form field `MAX_FILE_SIZE` are set appropriately, a number of other parameters need to be set, both on the server and in the HTML form. On the server side, the PHP directive `post_max_size` must be greater than `upload_max_filesize`. Also, the time it takes a file to upload is counted against the script execution time. Hence, the PHP directive `max_execution_time` needs to be set long enough to upload the expected files sizes. This can also be set within your script itself via the function `set_time_limit()`. There is also a directive called `max_input_`time, which is the maximum number of seconds that the script is allowed to parse input data. If the user is on a slow connection, or has a large file to upload, this limit may be exceeded. A small detail often overlooked is to make sure that permissions on the upload directory on the server are set such that the web server can actually write to it.

On the client side, there are two rules to follow: First, make sure that the form uses the `POST` method. Second, the form needs the following attribute: `enctype="multipart/form-data"`. Without all these requirements, your file upload will not work.

10.5 Generating Select Statements

Web applications often need to have a list of options repeated many times on the same web page. These might be a list of countries, states, classes of data, and so on. Even more often, this information may come from a database in the first place, so it is imperative for performance reasons to generate the list of options once and reuse it. Listing 10.5.1 demonstrates a method of creating such repeating options.

Listing 10.5.1 **Dynamically Generating Select Options**

```
<?php
// If this file was a submission, then output the values
if (count($_POST)) {
    echo "<p>You submitted the following music ID numbers:<br />
1st - {$_POST['1st']}, 2nd - {$_POST['2nd']}, 3rd - {$_POST['3rd']}</p>
";
}

// Array of options in a format that might have come from a database query:
$types = array(
    array('id' => 10, 'desc' => 'Rock'),
    array('id' => 11, 'desc' => 'Alternative'),
    array('id' => 12, 'desc' => 'Metal'),
    array('id' => 20, 'desc' => 'Classical'),
```

Listing 10.5.1 **Continued**

```
    array('id' => 33, 'desc' => 'Jazz'),
    array('id' => 34, 'desc' => 'Blues'),
    array('id' => 42, 'desc' => 'Ska'),
    array('id' => 44, 'desc' => 'Folk'),
    array('id' => 49, 'desc' => 'Blue Grass')
    );

// Now, generate the options portion of a select string, based upon this:
// Start with a blank option:
$options = "<option value=\"\">Select one ...</option>\n";

// Now loop over all possibilities, adding them in:
foreach ($types as $m) {
    $options .= "<option value=\"{$m['id']}\">{$m['desc']}</option>\n";
}

// And now we can output our page, using this multiple times:
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>Please choose your favorite types of music:</p>
<p>1st: <select name="1st"><?= $options ?></select></p>
<p>2nd: <select name="2nd"><?= $options ?></select></p>
<p>3rd: <select name="3rd"><?= $options ?></select></p>
<p><input type="submit" /></p>
</form>
```

This technique works fine when you have a list that will always be exactly the same. However, often you will need to have the proper option in the list preselected, based on some saved data. When you have the option list already created, this is not possible (except after the fact with JavaScript) because the HTML will need to change to set a new default value.

This is easily solvable, though. If you already have all the data stored in an array as we have, instead of generating the list once, you can create a function that generates the list of options on-the-fly, but preselecting the proper value based on a parameter. Listing 10.5.2 is a rewrite of Listing 10.5.1 to use this technique to have the select lists show your selection after submission.

Listing 10.5.2 **Dynamic Select Generation with Changing Data**

```
<?php
// If this file was a submission, then output the values
if (count($_POST)) {
    echo "<p>You submitted the following music ID numbers:<br />
1st - {$_POST['1st']}, 2nd - {$_POST['2nd']}, 3rd - {$_POST['3rd']}</p>
```

Listing 10.5.2 **Continued**

```
";
}

// Create a function that will take an ID, and an array of ID's & Desc's as
// parameters, and will output the options of a select list with the
// proper one highlighted.
function create_option_list($id, $data) {
    // The blank option - automatically selected if nothing else is
    $options = "<option value=\"\">Select one ...</option>\n";

    // Now loop over all possibilities, adding them in:
    foreach ($data as $x) {
        // Determine if this one should be selected
        $selected = ($id == $x['id']) ? ' selected' : '';

        // Now add in the option
        $options .=
            "<option value=\"{$x['id']}\"{$sel}>{$x['desc']}</option>\n";
    }

    // Now return the option list
    return $options;
}

// Array of options in a format that might have come from a database query:
$types = array(
    array('id' => 10, 'desc' => 'Rock'),
    array('id' => 11, 'desc' => 'Alternative'),
    array('id' => 12, 'desc' => 'Metal'),
    array('id' => 20, 'desc' => 'Classical'),
    array('id' => 33, 'desc' => 'Jazz'),
    array('id' => 34, 'desc' => 'Blues'),
    array('id' => 42, 'desc' => 'Ska'),
    array('id' => 44, 'desc' => 'Folk'),
    array('id' => 49, 'desc' => 'Blue Grass')
    );

// And now we can output our page, using this multiple times:
// We will use an @ in front of the $_POST references in order to suppress
//  the PHP notice that will occur due to it not being initialized at first.
?>
<form action="<?= $_SERVER['PHP_SELF'] ?>" method="post" name="f1">
<p>Please choose your favorite types of music:</p>
<p>1st: <select name="1st">
<?= create_option_list(@$_POST['1st'], $types) ?>
```

Listing 10.5.2 **Continued**

```
</select></p>
<p>2nd: <select name="2nd">
<?= create_option_list(@$_POST['2nd'], $types) ?>
</select></p>
<p>3rd: <select name="3rd">
<?= create_option_list(@$_POST['3rd'], $types) ?>
</select></p>
<p><input type="submit" /></p>
</form>
```

As one final option to explore, Listing 10.5.3 looks at the common need of generating lists of years and months. When looking at different people's code you may often find hard-coded lists of months or arrays built to hold them. PHP provides the capability to create these on-the-fly through creative use of the `date()`, `mktime()`, and `range()`.

Listing 10.5.3 **Generating Date-Based Options**

```
<?php
// Create a function that will return an option list of months:
function month_options() {
    $retval = '';

    // For all twelve months:
    foreach(range(1,12) as $mon) {
        // Make a timestamp for this month:
        $time = mktime(0, 0, 0, $mon);
        // Use date to return the month name as text:
        $name = date('F', $time);
        // Create the select option:
        $retval .= "<option value=\"{$mon}\">{$name}</option>\n";
    }

    return $retval;
}

// Create a function, that will output a list of years
// The first parameter is how many years in the past to start from
// The second parameter is how many years into the future to go:
function year_options($before, $after) {
    $retval = '';

    // From the current year:
    $cyear = date('Y');

    // Loop over the range requested:
    foreach(range($cyear - $before, $cyear + $after) as $y) {
```

Listing 10.5.3 **Continued**

```
        // Store the select option:
        $retval .= "<option value=\"{$y}\">{$y}</option>\n";
    }

    return $retval;
}

// Now we can just output a simple form using these:
?>
<form>
<p>Month:<select name="month"><?= month_options() ?></select></p>
<p>Year:<select name="year"><?= year_options(5, 10) ?></select></p>
</form>
```

10.6 Requiring Certain Fields to Be Filled Out

Often when accepting data from a user, certain fields on a form must be filled out. A common tactic to ensure this is to use JavaScript to check/require that every field is filled before submitting the form. The problem with this approach, however, is that the user might have JavaScript turned off or be using a browser that doesn't support JavaScript.

It is therefore better to use pure PHP and check the result values. You could have the form submit to a separate page, and on that page, if the values are not filled out, display an error. If you do that, the user needs to click the Back button. An interesting solution to remove the need of the back button is to have the page submit the data back to itself. This way, if the required data is not there, the warning can be given, plus any data that the user has already filled out can be repopulated. Using an array of required values makes it easier to do your checking, as shown in Listing 10.6.1.

Listing 10.6.1 **An HTML Page That Submits Back to Itself**

```
<?php
// An array of our required elements, and their real names:
$required = array('phone' => 'Phone Number', 'state' => 'State');

// If we had a POST, then check the data:
if (count($_POST)) {
    $errors = '';
    // Loop over all required fields, and ensure they exist:
    foreach ($required as $field => $desc) {
        // If it is not even set, or is blank after trimming:
        if (!isset($_POST[$field]) || (trim($_POST[$field]) === '')) {
            $errors .= "<br />{$desc} is a required field!\n";
        }
    }
```