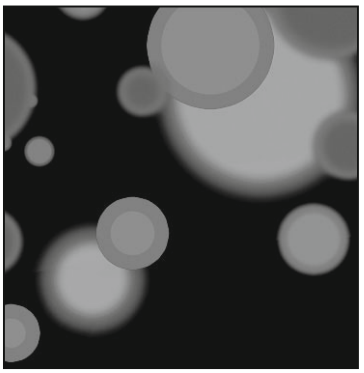


Chapter 6

Uploading Files

PHP's ability to handle forms isn't restricted to text. It can also be used to upload files to a server. For instance, you could build a real estate website for clients to upload pictures of their properties or a site for all your friends and relatives to upload their holiday photos. However, just because you can do it, doesn't necessarily mean that you should. Allowing others to upload material to your website could expose you to all sorts of problems. You need to make sure that images are the right size, that they're of suitable quality, and that they don't contain any illegal material. You also need to ensure that uploads don't contain malicious scripts. In other words, you need to protect your website just as carefully as your own computer.

PHP makes it relatively simple to restrict the type and size of files accepted. What it cannot do is check the suitability of the content. Think carefully about security measures, such as restricting uploads to registered and trusted users by placing the upload form in a password-protected area.

Until you learn how to restrict access to pages with PHP in Chapters 9 and 17, use the PHP solutions in this chapter only in a password-protected directory if deployed on a public website. Most hosting companies provide simple password protection through the site's control panel.

The first part of this chapter is devoted to understanding the mechanics of file uploads, which will make it easier to understand the code that follows. This is a fairly intense chapter, not a collection of quick solutions. But by the end of the chapter, you will have built a PHP class capable of handling single and multiple file uploads. You can then use the class in any form by writing only a few lines of code.

You'll learn about the following:

- Understanding the `$_FILES` array
- Restricting the size and type of uploads
- Preventing files from being overwritten
- Organizing uploads into specific folders
- Handling multiple uploads

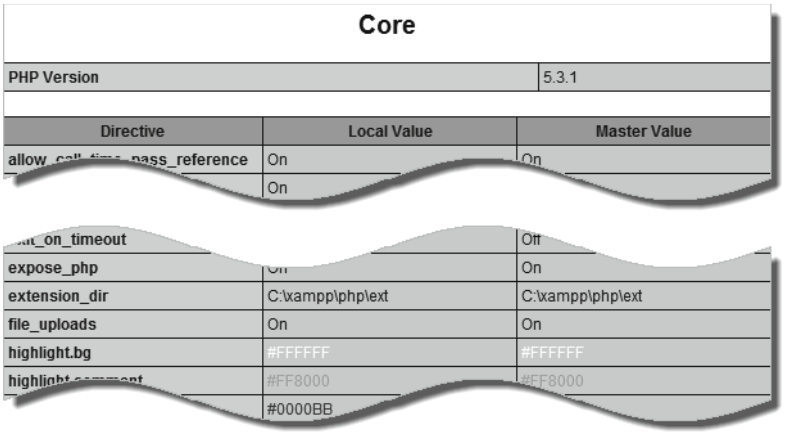
How PHP handles file uploads

The term “upload” means moving a file from one computer to another, but as far as PHP is concerned, all that’s happening is that a file is being moved from one location to another. This means you can test all the scripts in this chapter on your local computer without the need to upload files to a remote server.

PHP supports file uploads by default, but hosting companies can restrict the size of uploads or disable them altogether. Before going any further, it’s a good idea to check the settings on your remote server.

Checking whether your server supports uploads

All the information you need is displayed in the main PHP configuration page that you can display by running `phpinfo()` on your remote server, as described in Chapter 2. Scroll down until you find `file_uploads` in the **Core** section, as shown in the following screenshot.



Core

PHP Version	5.3.1

Directive	Local Value	Master Value
allow_call_time_pass_reference	On	On
	On	
_on_timeout		Off
expose_php	On	On
extension_dir	C:\xampp\php\ext	C:\xampp\php\ext
file_uploads	On	On
highlight.bg	#FFFFFF	#FFFFFF
highlight.comment	#FF8000	#FF8000
	#0000BB	

If the **Local Value** is **On**, you’re ready to go, but you should also check the other configuration settings listed in Table 6-1.

Table 6-1. PHP configuration settings that affect file uploads

Directive	Default value	Description
`max_execution_time`	30	The maximum number of seconds that a PHP script can run. If the script takes longer, PHP generates a fatal error.
`max_input_time`	60	The maximum number of seconds that a PHP script is allowed to parse the `$_POST` and `$_GET` arrays and file uploads. Very large uploads are likely to run out of time.
`post_max_size`	8M	The maximum permitted size of all `$_POST` data, *including* file uploads. Although the default is 8MB, hosting companies may impose a smaller limit.

Directive	Default value	Description
`upload_tmp_dir`		This is where PHP stores uploaded files until your script moves them to a permanent location. If no value is defined in `php.ini`, PHP uses the system default temporary directory (`C:\Windows\Temp` or `/tmp` on Mac/Linux).
`upload_max_filesize`	2M	The maximum permitted size of a single upload file. Although the default is 2MB, hosting companies may impose a smaller limit. A number on its own indicates the number of bytes permitted. A number followed by K indicates the number of kilobytes permitted.

The default 8MB value of `post_max_size` includes the content of the `$_POST` array, so the total size of files that can be uploaded simultaneously is less than 8MB, with no single file greater than 2MB. These defaults can be changed by the server administrator, so it's important to check the limits set by your hosting company. If you exceed those limits, an otherwise perfect script will fail.

If the **Local Value** of `file_uploads` is **Off**, uploads have been disabled. There is nothing you can do about it, other than ask your hosting company if it offers a package with file uploading enabled. Your only alternatives are to move to a different host or to use a different solution, such as uploading files by FTP.

After using `phpinfo()` to check your remote server's settings, it's a good idea to remove the script or put it in a password-protected directory.

Adding a file upload field to a form

Adding a file upload field to an HTML form is easy. Just add `enctype="multipart/form-data"` to the opening `<form>` tag, and set the `type` attribute of an `<input>` element to `file`. The following code is a simple example of an upload form (it's in `file_upload_01.php` in the `ch06` folder):

```
<form action="" method="post" enctype="multipart/form-data" id="uploadImage">
  <p>
    <label for="image">Upload image:</label>
    <input type="file" name="image" id="image">
  </p>
  <p>
  <input type="submit" name="upload" id="upload" value="Upload">
  </p>
</form>
```

Although this is standard HTML, how it's rendered in a web page depends on the browser (see Figure 6-1). Some browsers insert a text input field with a **Browse** button on the right. In older browsers, the text input field is editable, but most modern browsers make it read-only or launch the operating system's file navigation panel as soon as you click inside the field. Browsers that use the WebKit engine, such as Safari and Google Chrome, display a **Choose File** button with a status message or name of the selected

file on the right. These differences don't affect the operation of an upload form, but you need to take them into account when designing the layout.

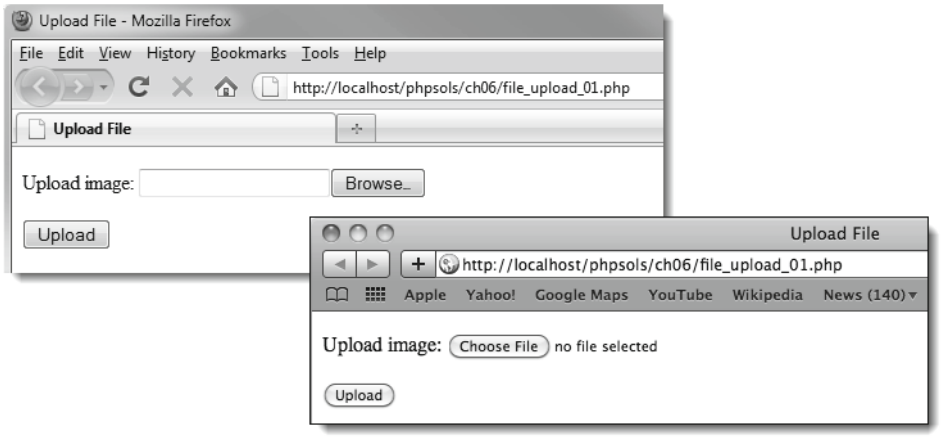


Figure 6-1. The look of a file input field depends on the browser.

Understanding the $_FILES array

What confuses many people is that their file seems to vanish after it has been uploaded. This is because you can't refer to an uploaded file in the `$_POST` array in the same way as with text input. PHP transmits the details of uploaded files in a separate superglobal array called, not unreasonably, `$_FILES`. Moreover, files are uploaded to a temporary folder and are deleted unless you explicitly move them to the desired location. Although this sounds like a nuisance, it's done for a very good reason: you can subject the file to security checks before accepting the upload.

Inspecting the $_FILES array

The best way to understand how the `$_FILES` array works is to see it in action. If you have installed a local test environment, you can test everything on your computer. It works in exactly the same way as uploading a file to a remote server.

1. Create a new folder called `uploads` in the `phpsols` site root. Create a new PHP file called `file_upload.php` in the `uploads` folder, and insert the code from the previous section. Alternatively, copy `file_upload_01.php` from the `ch06` folder, and rename the file `file_upload.php`.

2. Insert the following code right after the closing `</form>` tag (it's also in `file_upload_02.php`):

```
</form>
<pre>
<?php
if (isset($_POST['upload'])) {
  print_r($_FILES);
```

```
}
?>
</pre>
</body>
```

This uses `isset()` to checks whether the `$_POST` array contains `upload`, the `name` attribute of the submit button. If it does, you know the form has been submitted, so you can use `print_r()` to inspect the `$_FILES` array. The `<pre>` tags make the output easier to read.

3. Save `file_upload.php`, and load it into a browser.

4. Click the **Browse** (or **Choose File**) button, and select a file on your hard disk. Click **Open** (or **Choose** on a Mac) to close the file selection dialog box, and then click **Upload**. You should see something similar to Figure 6-2.

   You can see that the `$_FILES` array is actually a multidimensional array—an array of arrays. The top-level array contains just one element, which gets its key (or index) from the `name` attribute of the file input field—in this case, `image`.

Figure 6-2. The `$_FILES` array contains the details of an uploaded file.

The `image` element contains another array (or subarray) that consists of five elements, namely:

- `name`: The original name of the uploaded file
- `type`: The uploaded file's MIME type
- `tmp_name`: The location of the uploaded file
- `error`: An integer indicating the status of the upload
- `size`: The size of the uploaded file in bytes

Don't waste time searching for the temporary file indicated by `tmp_name`: it won't be there. If you don't save it immediately, PHP discards it.

5. Click **Upload** without selecting a file. The `$_FILES` array should look like Figure 6-3.

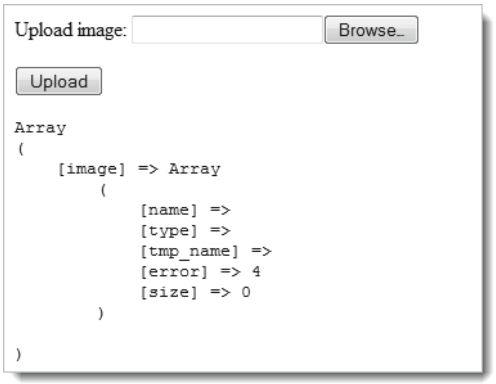


Figure 6-3. The `$_FILES` array still exists when no file is uploaded.

An error level of 4 indicates that no file was uploaded; 0 means the upload succeeded. Table 6-2 later in this chapter lists all the error codes.

6. Select a program file, and click the **Upload** button. In many cases, the form will happily try to upload the program and display its type as `application/zip` or something similar. This is a warning that it's important to check the MIME type of uploaded files.

Establishing an upload directory

Another source of confusion is the question of permissions. An upload script that works perfectly locally may confront you with a message like this when you transfer it to your remote server:

```
Warning:  move_uploaded_file(/home/user/htdocs/testarea/kinkakuji.jpg)
[function.move-uploaded-file]: failed to open stream: Permission denied in ➥
 /home/user/htdocs/testarea/upload_test.php on line 3
```

Why is permission denied? Most hosting companies use Linux servers, which impose strict rules about the ownership of files and directories. In most cases, PHP doesn't run in *your* name, but as the web server—usually `nobody` or `apache`. Unless PHP has been configured to run in your name, you need to give global access (`chmod 777`) to every directory to which you want to upload files.

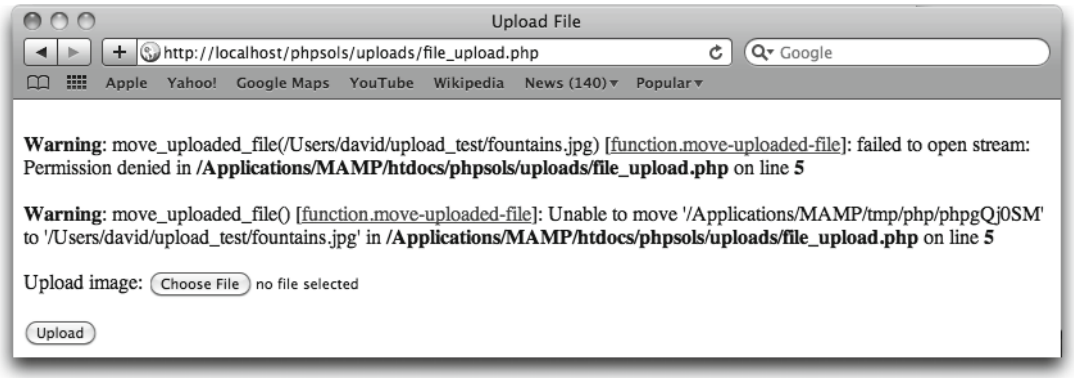
Since 777 is the least secure setting, begin by testing uploads with a setting of 700. If that doesn't work, try 770, and use 777 only as a last resort. The upload directory doesn't need to be within your site root. If your hosting company gives you a private directory outside the site root, create a subdirectory for uploads inside the private one. Alternatively, create a directory inside your site root, but don't link to it from any web pages. Give it an innocuous name, such as `lastyear`.

Creating an upload folder for local testing on Windows

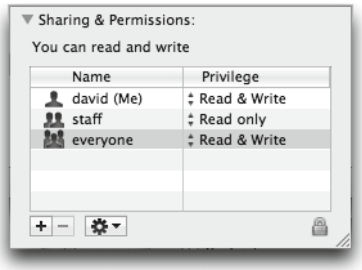
For the following exercises, I suggest you create a folder called `upload_test` at the top level of the C drive. There are no permissions issues on Windows, so that's all that you need to do.

Creating an upload folder for local testing on Mac OS X

Mac users need to do a little more preparation, because file permissions are similar to Linux. Without changing the permissions, you'll be confronted with a screen like this:



1. Create a folder called `upload_test` within your home folder.
2. Select `upload_test` in Finder, and select **File ➤ Get Info** (Cmd-I) to open its **Info panel**.
3. In **Sharing & Permissions**, click the padlock icon at the bottom right to unlock the settings, and change the setting for **everyone** from **Read only** to **Read & Write**, as shown in the following screenshot.



   In older versions of Mac OS X, **Sharing & Permissions** is called **Ownership & Permissions**, and **everyone** is called **Others**.
4. Click the padlock icon again to preserve the new settings, and close the **Info** panel. Your `upload_test` folder is now ready for use.

Uploading files

Before building the file upload class, it's a good idea to create a simple file upload script to make sure that your system handles uploads correctly.

Moving the temporary file to the upload folder

The temporary version of an uploaded file has only a fleeting existence. If you don't do anything with the file, it's discarded immediately. You need to tell PHP where to move it and what to call it. You do this with the `move_uploaded_file()` function, which takes the following two arguments:

- The name of the temporary file
- The full pathname of the file's new location, including the filename itself

Obtaining the name of the temporary file itself is easy: it's stored in the `$_FILES` array as `tmp_name`. Because the second argument requires a full pathname, it gives you the opportunity to rename the file. For the moment, let's keep things simple and use the original filename, which is stored in the `$_FILES` array as `name`.

PHP Solution 6-1: Creating a basic file upload script

Continue working with the same file as in the previous exercise. Alternatively, use `file_upload_03.php` from the `ch06` folder. The final script for this PHP solution is in `file_upload_04.php`.

1. If you are using the file from the previous exercise, delete the code highlighted in bold between the closing `</form>` and `</body>` tags:

```
</form>
<pre>
<?php
if (isset($_POST['upload'])) {
  print_r($_FILES);
  }
?>
</pre>
</body>
```

2. In addition to the automatic limits set in the PHP configuration (see Table 6-1), you can specify a maximum size for an upload file in your HTML form. Add the following line highlighted in bold immediately before the file input field:

```
<label for="image">Upload image:</label>
<input type="hidden" name="MAX_FILE_SIZE" value="<?php echo $max; ?>">
<input type="file" name="image" id="image">
```

   This is a hidden form field, so it won't be displayed onscreen. However, it's vital to place it *before* the file input field; otherwise, it won't work. The `name` attribute, `MAX_FILE_SIZE`, is fixed. The `value` attribute sets the maximum size of the upload file in bytes. Instead of specifying a numeric value, I have used a variable, which needs to be defined next. This value will also be used in the server-side validation of the file upload, so it makes sense to define it once, avoiding the possibility of changing it in one place, but forgetting to change it elsewhere.

3. Define the value of `$max` in a PHP block above the `DOCTYPE` declaration like this:

```
<?php
// set the maximum upload size in bytes
```

```
$max = 51200;
?>
<!DOCTYPE HTML>
```

This sets the maximum upload size to 50kB (51,200 bytes).

4. The code that moves the uploaded file from its temporary location to its permanent one needs to be run after the form has been submitted. Insert the following code in the PHP block you have just created at the top of the page:

```
$max = 51200;
if (isset($_POST['upload'])) {
  // define the path to the upload folder
  $destination = '/path/to/upload_test/';
  // move the file to the upload folder and rename it
  move_uploaded_file($_FILES['image']['tmp_name'], $destination . ➥
    $_FILES['image']['name']);
}
?>
```

Although the code is quite short, there's a lot going on. The entire code block is enclosed in a conditional statement that checks whether the **Upload** button has been clicked by checking to see if its key is in the `$_POST` array.

The value of `$destination` depends on your operating system and the location of the upload folder.

- If you are using Windows, and you created the `upload_test` folder at the top level of the C drive, it should look like this:

  ```
  $destination = 'C:/upload_test/';
  ```

  Note that I have used forward slashes instead of the Windows convention of backslashes. You can use either, but if you use backslashes, the final one needs to be escaped by another backslash, like this (otherwise the backslash escapes the quote):

  ```
  $destination = 'C:\upload_test\\';
  ```

- On a Mac, if you created the `upload_test` folder in your home folder, it should look like this (replace *username* with your Mac username):

  ```
  $destination = '/Users/username/upload_test/';
  ```

- On a remote server, you need the fully qualified filepath as the second argument. On Linux, it will probably be something like this:

  ```
  $destination = '/home/user/private/upload_test/';
  ```

The final line inside the `if` statement moves the file with the `move_uploaded_file()` function. Since `$_FILES['image']['name']` contains the name of the original file, the second argument, `$destination . $_FILES['image']['name']`, stores the uploaded file under its original name inside the upload folder.

You may come across scripts that use `copy()` instead of `move_uploaded_file()`. Without other checks in place, `copy()` can expose your website to serious security risks. For example, a malicious user could try to trick your script into copying files that it should not have access to, such as password files. Always use `move_uploaded_file()`; it's much more secure.

5. Save `file_upload.php`, and load it into your browser. Click the **Browse** or **Choose File** button, and select a file from the `images` folder in the `phpsols` site. If you choose one from elsewhere, make sure it's less than 50kB. Click **Open** (**Choose** on a Mac) to display the filename in the form. In browsers that display a file input field, you might not be able to see the full path. That's a cosmetic matter that I'll leave you to sort out yourself with CSS. Click the **Upload** button. If you're testing locally, the form input field should clear almost instantly.
6. Navigate to the `upload_test` folder, and confirm that a copy of the image you selected is there. If there isn't, check your code against `file_upload_04.php`. Also check that the correct permissions have been set on the upload folder, if necessary.

The download files set `$destination` to `C:/upload_test/`. Adjust this to your own setup.

7. If you get no error messages and cannot find the file, make sure that the image didn't exceed `upload_max_filesize` (see Table 6-1). Also check that you didn't leave the trailing slash off the end of `$destination`. Instead of `myfile.jpg` in `upload_test`, you may find `upload_testmyfile.jpg` one level higher in your disk structure.
8. Change the value of `$max` to `3000`, save `file_upload.php`, and test it again by selecting a file bigger than 2.9kB to upload (any file in the `images` folder will do). Click the **Upload** button, and check the upload folder. The file shouldn't be there.
9. If you're in the mood for experimentation, move the `MAX_FILE_SIZE` hidden field below the file input field, and try it again. Make sure you choose a different file from the one you used in step 6, because `move_uploaded_file()` overwrites existing files of the same name. You'll learn later how to give files unique names.

   This time the file should be copied to your upload folder. Move the hidden field back to its original position before continuing.

The advantage of using `MAX_FILE_SIZE` is that PHP abandons the upload if the file is bigger than the stipulated value, avoiding unnecessary delay if the file is too big. Unfortunately, users can get around this restriction by faking the value submitted by the hidden field, so the script you'll develop in the rest of this chapter will check the actual size of the file on the server side, too.

Creating a PHP file upload class

As you have just seen, it takes just a few lines of code to upload a file, but this is not enough on its own. You need to make the process more secure by implementing the following steps:

- Check the error level.
- Verify on the server that the file doesn't exceed the maximum permitted size.
- Check that the file is of an acceptable type.
- Remove spaces from the filename.
- Rename files that have the same name as an existing one to prevent overwriting.
- Handle multiple file uploads automatically.
- Inform the user of the outcome.

You need to implement these steps every time you want to upload files, so it makes sense to build a script that can be reused easily. That's why I have chosen to use a custom class. Building PHP classes is generally regarded as an advanced subject, but don't let that put you off. I won't get into the more esoteric details of working with classes, and the code is fully explained. Although the class definition is long, using the class involves writing only a few lines of code.

A **class** is a collection of functions designed to work together. That's an oversimplification, but it's sufficiently accurate to give you the basic idea behind building a file upload class. Each function inside a class should normally focus on a single task, so you'll build separate functions to implement the steps outlined in the previous list. The code should also be generic, so it isn't tied to a specific web page. Once you have built the class, you can reuse it in any form.

If you're in a hurry, the finished class is in the `classes/completed` folder of the download files. Even if you don't build the script yourself, read through the descriptions so you have a clear understanding of how it works.

Defining a PHP class

Defining a PHP class is very easy. You use the `class` keyword followed by the class name and put all the code for the class between a pair of curly braces. By convention, class names normally begin with an uppercase letter and are stored in a separate file. It's also recommended to prefix class names with an uncommon combination of 3–4 letters followed by an underscore to prevent naming conflicts (see `http://docs.php.net/manual/en/userlandnaming.tips.php`). All custom classes in this book use `Ps2_`.

PHP 5.3 introduced the concept of namespaces to avoid naming conflicts. At the time of this writing, many hosting companies have not yet migrated to PHP 5.3, so namespaces may not be supported on your server. PHP Solution 6-7 converts the scripts to use namespaces.

PHP Solution 6-2: Creating the basic file upload class

In this PHP solution, you'll create the basic definition for a class called `Ps2_Upload`, which stores the `$_FILES` array in an internal property ready to handle file uploads. You'll also create an instance of the class (a `Ps2_Upload` object), and use it to upload an image.

1. Create a subfolder called `Ps2` in the `classes` folder.
2. In the new `Ps2` folder, create a file called `Upload.php`, and insert the following code:

```
<?php
```

```
class Ps2_Upload {

}
```

That, believe it or not, is a valid class called `Ps2_Upload`. It doesn't do anything, so it's not much use yet, but it will be once you start adding code between the curly braces. This file will contain only PHP code, so you don't need a closing PHP tag.

3. In many ways, a class is like a car engine. Although you can strip down the engine to see its inner workings, most of the time, you're not interested in what goes on inside, as long as it powers your car. PHP classes hide their inner workings by declaring some variables and functions as protected. If you prefix a variable or function with the keyword `protected`, it can be accessed only inside the class. The reason for doing so is to prevent values from being changed accidentally.

> *Technically speaking, a protected variable or function can also be accessed by a subclass derived from the original class. To learn about classes in more depth, see my* PHP Object-Oriented Solutions *(friends of ED, 2008, ISBN: 978-1-4302-1011-5).*

The `Ps2_Upload` class needs protected variables for the following items:

- `$_FILES` array
- Path to the upload folder
- Maximum file size
- Messages to report the status of uploads
- Permitted file types
- A Boolean variable that records whether a filename has been changed

Create the variables by adding them inside the curly braces like this:

```
class Ps2_Upload {

  protected $_uploaded = array();
  protected $_destination;
  protected $_max = 51200;
  protected $_messages = array();
  protected $_permitted = array('image/gif',
                                'image/jpeg',
                                'image/pjpeg',
                                'image/png');
  protected $_renamed = false;

}
```

I have begun the name of each protected variable (or **property**, as they're normally called inside classes) with an underscore. This is a common convention programmers use to remind themselves that a property is protected; but it's the `protected` keyword that restricts access to the property, not the underscore.

By declaring the properties like this, they can be accessed elsewhere in the class using `$this->`, which refers to the current object. For example, inside the class definition, you access `$_uploaded` as `$this->_uploaded`.

When you first declare a property inside a class, it begins with a dollar sign like any other variable. However, you omit the dollar sign from the property name after the -> operator.

With the exception of `$_destination`, each protected property has been given a default value:

- `$_uploaded` and `$_messages` are empty arrays.
- `$_max` sets the maximum file size to 50kB (51200 bytes).
- `$_permitted` contains an array of image MIME types.
- `$_renamed` is initially set to `false`.

The value of `$_destination` will be set when an instance of the class is created. The other values will be controlled internally by the class, but you'll also create functions (or **methods**, as they're called in classes) to change the values of `$_max` and `$_permitted`.

4. When you create an instance of a class (an **object**), the class definition file automatically calls the class's constructor method, which initializes the object. The constructor method for all classes is called `__construct()` (with two underscores). Unlike the properties you defined in the previous step, the constructor needs to be accessible outside the class, so you precede its definition with the `public` keyword.

*The `public` and `protected` keywords control the **visibility** of properties and methods. Public properties and methods can be accessed anywhere. Any attempt to access protected properties or methods outside the class definition or a subclass triggers a fatal error.*

The constructor for the `Ps2_Upload` class takes the path to the upload folder as an argument and assigns it to `$_destination`. It also assigns the `$_FILES` array to `$_uploaded`. The code looks like this:

```
protected $_renamed = false;

public function __construct($path) {
  if (!is_dir($path) || !is_writable($path)) {
    throw new Exception("$path must be a valid, writable directory.");
```

```
    }
    $this->_destination = $path;
    $this->_uploaded = $_FILES;
  }

}
```

The conditional statement inside the constructor passes `$path` to the `is_dir()` and `is_writable()` functions, which check that the value submitted is a valid directory (folder) that is writable. If either condition fails, the constructor throws an exception with a message indicating the problem.

If `$path` is OK, it's assigned the `$_destination` property of the current object, and the `$_FILES` array is assigned to `$_uploaded`.

Don't worry if this sounds mysterious. You'll soon see the fruits of your efforts.

5. With the `$_FILES` array stored in `$_uploaded`, you can access the file's details and move it to the upload folder with `move_uploaded_file()`. Create a public method called `move()` immediately after the constructor, but still inside the curly braces of the class definition. The code looks like this:

```
public function move() {
  $field = current($this->_uploaded);
  $success = move_uploaded_file($field['tmp_name'], $this->_destination . ➥
    $field['name']);
  if ($success) {
    $this->_messages[] = $field['name'] . ' uploaded successfully';
  } else {
    $this->_messages[] = 'Could not upload ' . $field['name'];
  }
}
```

To access the file in the `$_FILES` array in PHP Solution 6-1, you needed to know the `name` attribute of the file input field. The form in `file_upload.php` uses `image`, so you accessed the filename as `$_FILES['image']['name']`. But if the field had a different name, such as `upload`, you would need to use `$_FILES['upload']['name']`. To make the script more flexible, the first line of the `move()` method passes the `$_uploaded` property to the `current()` function, which returns the current element of an array—in this case, the first element of the `$_FILES` array. As a result, `$field` holds a reference to the first uploaded file regardless of name used in the form. This is the first benefit of building generic code. It takes more effort initially, but saves time in the end.

So, instead of using `$_FILES['image']['tmp_name']` and `$_FILES['image']['name']` in `move_uploaded_file()`, you refer to `$field['tmp_name']` and `$field['name']`. If the upload succeeds, `move_uploaded_file()` returns `true`. Otherwise, it returns `false`. By storing the result in `$success`, you can control which message is assigned to the `$_messages` array.

6. Since `$_messages` is a protected property, you need to create a public method to retrieve the contents of the array. Add this to the class definition after the `move()` method:

```
public function getMessages() {
  return $this->_messages;
}
```

This simply returns the contents of the `$_messages` array. Since that's all it does, why not make the array public in the first place? Public properties can be accessed—and changed—outside the class definition. This ensures that the contents of the array cannot be altered, so you know the message has been generated by the class. This might not seem such a big deal with a message like this, but it becomes very important when you start working with more complex scripts or in a team.

7. Save `Upload.php`, and change the code at the top of `file_upload.php` like this:

```
<?php
// set the maximum upload size in bytes
$max = 51200;
if (isset($_POST['upload'])) {
  // define the path to the upload folder
  $destination = 'C:/upload_test/';
  require_once('../classes/Ps2/Upload.php');
  try {
    $upload = new Ps2_Upload($destination);
    $upload->move();
    $result = $upload->getMessages();
  } catch (Exception $e) {
    echo $e->getMessage();
  }
}
?>
```

This includes the `Ps2_Upload` class definition and creates an instance of the class called `$upload` by passing it the path to the upload folder. It then calls the `$upload` object's `move()` and `getMessages()` methods, storing the result of `getMessages()` in `$result`. Because the object might throw an exception, the code is wrapped in a `try/catch` block.

At the moment, the value of `$max` in `file_upload.php` affects only `MAX_FILE_SIZE` in the hidden form field. Later, you'll also use `$max` to control the maximum file size permitted by the class.

8. Add the following PHP code block above the form to display any messages returned by the `$upload` object:

```
<body>
<?php
if (isset($result)) {
  echo '<ul>';
  foreach ($result as $message) {
```

```
    echo "<li>$message</li>";
  }
echo '</ul>';
}
?>
<form action="" method="post" enctype="multipart/form-data" id="uploadImage">
```

This is a simple `foreach` loop that displays the contents of `$result` as an unordered list. When the page first loads, `$result` isn't set, so this code runs only after the form has been submitted.

9. Save `file_upload.php`, and test it in a browser. As long as you choose an image that's less than 50kB, you should see confirmation that the file was uploaded successfully, as shown in Figure 6-4.

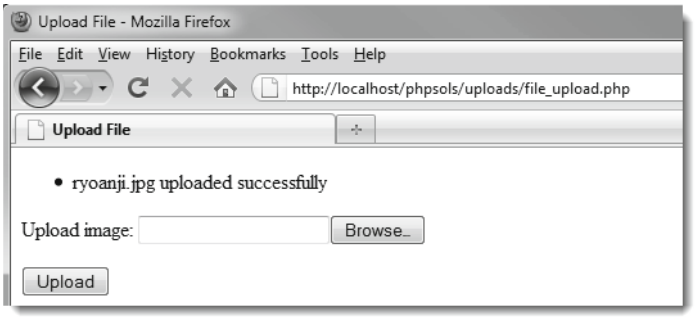


Figure 6-4. The `Ps2_Upload` class reports a successful upload.

You can compare your code with `file_upload_05.php` and `Upload_01.php` in the `ch06` folder.

The class does exactly the same as PHP Solution 6-1: it uploads a file, but it requires a lot more code to do so. However, you have laid the foundation for a class that's going to perform a series of security checks on uploaded files. This is code that you'll write once. When you use the class, you won't need to write this code again.

If you haven't worked with objects and classes before, some of the concepts might seem strange. Think of the `$upload` object simply as a way of accessing the functions (methods) you have defined in the `Ps2_Upload` class. You often create separate objects to store different values, for example, when working with `DateTime` objects. In this case, a single object is sufficient to handle the file upload.

Checking upload errors

As it stands, the `Ps2_Upload` class uploads any type of file indiscriminately. Even the 50kB maximum size can be circumvented, because the only check is made in the browser. Before handing the file to `move_uploaded_file()`, you need to run a series of checks to make sure the file is OK. And if a file is rejected, you need to let the user know why.

PHP Solution 6-3: Testing the error level, file size, and MIME type

This PHP solution shows how to create a series of internal (protected) methods for the class to verify that the file is OK to accept. If a file fails for any reason, an error message reports the reason to the user.

Continue working with `Upload.php`. Alternatively, use `Upload_01.php` in the `ch06` folder, and rename it `Upload.php`. (Always remove the underscore and number from partially completed files.)

1. The first test you should run is on the error level. As you saw in the exercise at the beginning of this chapter, level 0 indicates the upload was successful and level 4 that no file was selected.

   Table 6-2 shows a full list of error levels. Error level 8 is the least helpful, because PHP has no way of detecting which PHP extension was responsible for stopping the upload. Fortunately, it's rarely encountered.

Table 6-2. Meaning of the different error levels in the `$_FILES` array

Error level*	Meaning
0	Upload successful
1	File exceeds maximum upload size specified in `php.ini` (default 2MB)
2	File exceeds size specified by `MAX_FILE_SIZE` (see PHP Solution 6-1)
3	File only partially uploaded
4	Form submitted with no file specified
6	No temporary folder
7	Cannot write file to disk
8	Upload stopped by an unspecified PHP extension

**Error level 5 is not currently defined.*

2. Add the following code after the definition of `getMessages()` in `Upload.php`:

```
protected function checkError($filename, $error) {
  switch ($error) {
    case 0:
      return true;
    case 1:
    case 2:
      $this->_messages[] = "$filename exceeds maximum size: " . ➥
        $this->getMaxSize();
      return true;
    case 3:
```

```
        $this->_messages[] = "Error uploading $filename. Please try again.";
        return false;
      case 4:
        $this->_messages[] = 'No file selected.';
        return false;
      default:
        $this->_messages[] = "System error uploading $filename. Contact ➥
          webmaster.";
        return false;
    }
  }
```

Preceding the definition with the `protected` keyword means this method can be accessed only inside the class. The `checkError()` method will be used internally by the `move()` method to determine whether to save the file to the upload folder.

It takes two arguments, the filename and the error level. The method uses a `switch` statement (see "Using the switch statement for decision chains" in Chapter 3). Normally, each `case` in a `switch` statement is followed by the `break` keyword, but that's not necessary here, because `return` is used instead.

Error level 0 indicates a successful upload, so it returns `true`.

Error levels 1 and 2 indicate the file is too big, and an error message is added to the `$_messages` array. Part of the message is created by a method called `getMaxSize()`, which converts the value of `$_max` from bytes to kB. You'll define `getMaxSize()` shortly. Note the use of `$this->`, which tells PHP to look for the method definition in this class.

Logic would seem to demand that `checkError()` should return `false` if a file's too big. However, setting it to `true` gives you the opportunity to check for the wrong MIME type, too, so you can report both errors.

Error levels 3 and 4 return `false` and add the reason to the `$_messages` array. The `default` keyword catches other error levels, including any that might be added in future, and adds a generic reason.

3. Before using the `checkError()` method, let's define the other tests. Add the definition for the `checkSize()` method, which looks like this:

```
protected function checkSize($filename, $size) {
  if ($size == 0) {
    return false;
  } elseif ($size > $this->_max) {
    $this->_messages[] = "$filename exceeds maximum size: " . ➥
      $this->getMaxSize();
    return false;
  } else {
    return true;
  }
}
```

Like `checkError()`, this takes two arguments—the filename and the size of the file as reported by the `$_FILES` array—and returns `true` or `false`.

The conditional statement starts by checking if the reported size is zero. This happens if the file is too big or no file was selected. In either case, there's no file to save and the error message will have been created by `checkError()`, so the method returns `false`.

Next, the reported size is compared with the value stored in `$_max`. Although `checkError()` should pick up files that are too big, you still need to make this comparison in case the user has managed to sidestep `MAX_FILE_SIZE`. The error message also uses `getMaxSize()` to display the maximum size.

If the size is OK, the method returns `true`.

4. The third test checks the MIME type. Add the following code to the class definition:

```
protected function checkType($filename, $type) {
  if (!in_array($type, $this->_permitted)) {
    $this->_messages[] = "$filename is not a permitted type of file.";
    return false;
  } else {
    return true;
  }
}
```

This follows the same pattern of accepting the filename and the value to be checked as arguments and returning `true` or `false`. The conditional statement checks the type reported by the `$_FILES` array against the array stored in `$_permitted`. If it's not in the array, the reason for rejection is added to the `$_messages` array.

5. The `getMaxSize()` method used by the error messages in `checkError()` and `checkSize()` converts the raw number of bytes stored in `$_max` into a friendlier format. Add the following definition to the class file:

```
public function getMaxSize() {
  return number_format($this->_max/1024, 1) . 'kB';
}
```

This uses the `number_format()` function, which normally takes two arguments: the value you want to format and the number of decimal places you want the number to have. The first argument is `$this->_max/1024`, which divides `$_max` by 1024 (the number of bytes in a kB). The second argument is 1, so the number is formatted to one decimal place. The `. 'kB'` at the end concatenates kB to the formatted number.

The `getMaxSize()` method has been declared public in case you want to display the value in another part of a script that uses the `Ps2_Upload` class.

6. You can now check the validity of the file before handing it to `move_uploaded_file()`. Amend the `move()` method like this:

```
public function move() {
  $field = current($this->_uploaded);
  $OK = $this->checkError($field['name'], $field['error']);
  if ($OK) {
    $success = move_uploaded_file($field['tmp_name'], $this->_destination ➥
      . $field['name']);
    if ($success) {
      $this->_messages[] = $field['name'] . ' uploaded successfully';
    } else {
      $this->_messages[] = 'Could not upload ' . $field['name'];
    }
  }
}
```

The arguments passed to the `checkError()` method are the filename and the error level reported by the `$_FILES` array. The result is stored in `$OK`, which a conditional statement uses to control whether `move_uploaded_file()` is called.

7. The next two tests go inside the conditional statement. Both pass the filename and relevant element of the `$_FILES` array as arguments. The results of the tests are used in a new conditional statement to control the call to `move_uploaded_file()` like this:

```
public function move() {
  $field = current($this->_uploaded);
  $OK = $this->checkError($field['name'], $field['error']);
  if ($OK) {
    $sizeOK = $this->checkSize($field['name'], $field['size']);
    $typeOK = $this->checkType($field['name'], $field['type']);
    if ($sizeOK && $typeOK) {
      $success = move_uploaded_file($field['tmp_name'], $this->_destination ➥
        . $field['name']);
      if ($success) {
        $this->_messages[] = $field['name'] . ' uploaded successfully';
      } else {
        $this->_messages[] = 'Could not upload ' . $field['name'];
      }
    }
  }
}
```

8. Save `Upload.php`, and test it again with `file_upload.php`. With images smaller than 50kB, it works the same as before. But if you try uploading a file that's too big and of the wrong MIME type, you get a result similar to Figure 6-5.

   You can check your code against `Upload_02.php` in the `ch06` folder.

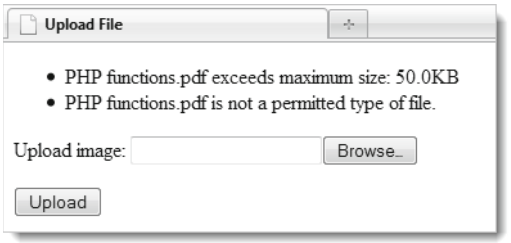


Figure 6-5. The class now reports errors with invalid size and MIME types.

Changing protected properties

The `$_permitted` property restricts uploads to images, but you might want to allow different types. Instead of diving into the class definition file every time you have different requirements, you can create public methods that allow you to make changes to protected properties on the fly.

You can find definitions of recognized MIME types at `www.iana.org/assignments/media-types`. Table 6-3 lists some of the most commonly used ones.

Table 6-3. Commonly used MIME types

Category	MIME type	Description
Documents	`application/pdf`	PDF document
	`text/plain`	Plain text
	`text/rtf`	Rich text format
Images	`image/gif`	GIF format
	`image/jpeg`	JPEG format (includes `.jpg` files)
	`image/pjpeg`	JPEG format (nonstandard, used by Internet Explorer)
	`image/png`	PNG format
	`image/tiff`	TIFF format

An easy way to find other MIME types not listed in Table 6-3 is to use `file_upload_02.php` and see what value is displayed for `$_FILES['image']['type']`.

PHP Solution 6-4: Allowing different types and sizes to be uploaded

This PHP solution shows you how to add one or more MIME types to the existing `$_permitted` array and how to reset the array completely. To keep the code relatively simple, the class checks the validity of only

a few MIME types. Once you understand the principle, you can expand the code to suit your own requirements. You'll also add a public method to change the maximum permitted size.

Continue working with `Upload.php` from the previous PHP solution. Alternatively, use `Upload_02.php` in the `ch06` folder.

1. The `Ps2_Upload` class already defines four permitted MIME types for images, but there might be occasions when you want to permit other types of documents to be uploaded as well. Rather than listing all permitted types again, it's easier to add the extra ones. Add the following method definition to the class file:

```
public function addPermittedTypes($types) {
  $types = (array) $types;
  $this->isValidMime($types);
  $this->_permitted = array_merge($this->_permitted, $types);
}
```

   This takes a single argument, `$types`, which is checked for validity and then merged with the `$_permitted` array. The first line inside the method looks like this:

```
$types = (array) $types;
```

   The highlighted code is what's known as a **casting operator** (see "Explicitly changing a data type" after this PHP solution). It forces the following variable to be a specific type—in this case, an array. This is because the final line of code passes `$types` to the `array_merge()` function, which expects both arguments to be arrays. As the function name indicates, it merges the arrays and returns the combined array.

   The advantage of using the casting operator here is that it allows you to use either an array or a string as an argument to `addPermittedTypes()`. For example, to add multiple types, you use an array like this:

```
$upload->addPermittedTypes(array('application/pdf', 'text/plain'));
```

   But to add one new type, you can use a string like this:

```
$upload->addPermittedTypes('application/pdf');
```

   Without the casting operator, you would need an array for even one item like this:

```
$upload->addPermittedTypes(array('application/pdf'));
```

   The middle line calls an internal method `isValidMime()`, which you'll define shortly.

2. On other occasions, you might want to replace the existing list of permitted MIME types entirely. Add the following definition for `setPermittedTypes()` to the class file:

```
public function setPermittedTypes($types) {
  $types = (array) $types;
  $this->isValidMime($types);
  $this->_permitted = $types;
}
```

This is quite simple. The first two lines are the same as `addPermittedTypes()`. The final line assigns `$types` to the `$_permitted` property, replacing all existing values.

3. Both methods call `isValidMime()`, which checks the values passed to them as arguments. Define the method now. It looks like this:

```
protected function isValidMime($types) {
  $alsoValid = array('image/tiff',
                     'application/pdf',
                     'text/plain',
                     'text/rtf');
  $valid = array_merge($this->_permitted, $alsoValid);
  foreach ($types as $type) {
    if (!in_array($type, $valid)) {
      throw new Exception("$type is not a permitted MIME type");
    }
  }
}
```

The method begins by defining an array of valid MIME types not already listed in the `$_permitted` property. Both arrays are then merged to produce a full list of valid types. The `foreach` loop checks each value in the user-submitted array by passing it to the `in_array()` function. If a value fails to match those listed in the `$valid` array, the `isValidMime()` method throws an exception, preventing the script from continuing.

4. The public method for changing the maximum permitted size needs to check that the submitted value is a number and assign it to the `$_max` property. Add the following method definition to the class file:

```
public function setMaxSize($num) {
  if (!is_numeric($num)) {
    throw new Exception("Maximum size must be a number.");
  }
  $this->_max = (int) $num;
}
```

This passes the submitted value to the `is_numeric()` function, which checks that it's a number. If it isn't, an exception is thrown.

The final line uses another casting operator—this time forcing the value to be an integer—before assigning the value to the `$_max` property. The `is_numeric()` function accepts any type of number, including a hexadecimal one or a string containing a numeric value. So, this ensures that the value is converted to an integer.

PHP also has a function called `is_int()` that checks for an integer. However, the value cannot be anything else. For example, it rejects `'102400'` even though it's a numeric value because the quotes make it a string.

5. Save `Upload.php`, and test `file_upload.php` again. It should continue to upload images smaller than 50kB as before.

6. Amend the code in `file_upload.php` to change the maximum permitted size to 3000 bytes like this:

```
$max = 3000;
if (isset($_POST['upload'])) {
// define the path to the upload folder
$destination = 'C:/upload_test/';
require_once('../classes/Ps2/Upload.php');
try {
  $upload = new Ps2_Upload($destination);
  $upload->setMaxSize($max);
  $upload->move();
```

   By changing the value of `$max` and passing it as the argument to `setMaxSize()`, you affect both `MAX_FILE_SIZE` in the form's hidden field and the maximum value stored inside the class. Note that the call to `setMaxSize()` *must* come before you use the `move()` method. There's no point changing the maximum size in the class after the file has already been saved.

7. Save `file_upload_php`, and test it again. Select an image you haven't used before, or delete the contents of the `upload_test` folder. The first time you try it, you should see a message that the file is too big. If you check the `upload_test` folder, you'll see it hasn't been transferred.

8. Try it again. This time, you should see a result similar to Figure 6-6.

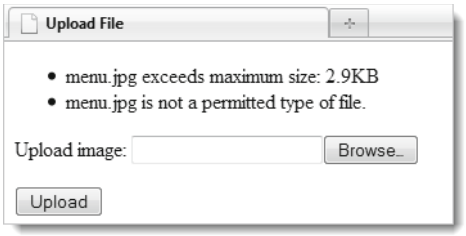


Figure 6-6. The size restriction is working, but there's an error in checking the MIME type.

   What's going on? The reason you probably didn't see the message about the permitted type of file the first time is because the value of `MAX_FILE_SIZE` in the hidden field isn't refreshed until you reload the form in the browser. The error message appears the second time because the updated value of `MAX_FILE_SIZE` prevents the file from being uploaded. As a result, the type element of the `$_FILES` array is empty. You need to tweak the `checkType()` method to fix this problem.

9. In `Upload.php`, amend the `checkType()` definition like this:

```
protected function checkType($filename, $type) {
  if (empty($type)) {
```

```
      return false;
    } elseif (!in_array($type, $this->_permitted)) {
      $this->_messages[] = "$filename is not a permitted type of file.";
      return false;
    } else {
      return true;
    }
  }
```

This adds a new condition that returns `false` if `$type` is empty. It needs to come before the other condition, because there's no empty value in the `$_permitted` array, which is why the false error message was generated.

10. Save the class definition, and test `file_upload.php` again. This time, you should see only the message about the file being too big.

11. Reset the value of `$max` at the top of `file_upload.php` to **51200**. You should now be able to upload the image. If it fails the first time, it's because `MAX_FILE_SIZE` hasn't been refreshed in the form.

12. Test the `addPermittedTypes()` method by adding an array of MIME types like this:

```
$upload->setMaxSize($max);
$upload->addPermittedTypes(array('application/pdf', 'text/plain'));
$upload->move();
```

MIME types must always be in lowercase.

13. Try uploading a PDF file. Unless it's smaller than 50kB, it won't be uploaded. Try a small text document. It should be uploaded. Change the value of `$max` to a suitably large number, and the PDF should also be uploaded.

14. Replace the call to `addPermittedTypes()` with `setPermittedTypes()` like this:

```
$upload->setMaxSize($max);
$upload->setPermittedTypes('text/plain');
$upload->move();
```

You can now upload only text files. All other types are rejected.

If necessary, check your class definition against `Upload_03.php` in the `ch06` folder.

Hopefully, by now you should be getting the idea of how a PHP class is built from functions (methods) that are dedicated to doing a single job. Fixing the incorrect error message about the image not being a permitted type was made easier by the fact that the message could only have come from the `checkType()` method. Most of the code used in the method definitions relies on built-in PHP functions. Once you learn which functions are the most suited to the task in hand, building a class—or any other PHP script—becomes much easier.

Explicitly changing a data type

Most of the time, you don't need to worry about the data type of a variable or value. Strictly speaking, all values submitted through a form are strings, but PHP silently converts numbers to the appropriate data type. This automatic **type juggling**, as it's called, is very convenient. There are times, though, when you want to make sure a value is a specific data type. In such cases, you can **cast** (or change) a value to the desired type by preceding it with the name of the data type in parentheses. You saw two examples of this in PHP Solution 6-4, casting a string to an array and a numeric value to an integer. This is how the value assigned to `$types` was converted to an array:

```
$types = (array) $types;
```

If the value is already of the desired type, it remains unchanged. Table 6-4 lists the casting operators used in PHP.

Table 6-4. PHP casting operators

Operator	Alternatives	Converts to
`(array)`		Array
`(bool)`	`(boolean)`	Boolean (`true` or `false`)
`(float)`	`(double)`, `(real)`	Floating-point number
`(int)`	`(integer)`	Integer
`(object)`		Object
`(string)`		String
`(unset)`		Null

To learn more about what happens when casting between certain types, see the online documentation at `http://docs.php.net/manual/en/language.types.type-juggling.php`.

Preventing files from being overwritten

As the script stands, PHP automatically overwrites existing files without warning. That may be exactly what you want. On the other hand, it may be your worst nightmare. The class needs to offer a choice of whether to overwrite an existing file or to give it a unique name.

PHP Solution 6-5: Checking an uploaded file's name before saving it

This PHP solution improves the `Ps2_Upload` class by adding the option to insert a number before the filename extension of an uploaded file to avoid overwriting an existing file of the same name. By default, this option is turned on. At the same time, all spaces in filenames are replaced with underscores. Spaces should never be used in file and folder names on a web server, so this feature isn't optional.

Continue working with the same class definition file as before. Alternatively, use `Upload_03.php` in the `ch06` folder.

1. Both operations are performed by the same method, which takes two arguments: the filename and a Boolean variable that determines whether to overwrite existing files. Add the following definition to the class file:

```
protected function checkName($name, $overwrite) {
  $nospaces = str_replace(' ', '_', $name);
  if ($nospaces != $name) {
    $this->_renamed = true;
  }
  if (!$overwrite) {
    // rename the file if it already exists
  }
  return $nospaces;
}
```

   This first part of the method definition takes the filename and replaces spaces with underscores using the `str_replace()` function, which takes the following three arguments:

   - The character(s) to replace—in this case, a space
   - The replacement character(s)—in this case, an underscore
   - The string you want to update—in this case, `$name`

   The result is stored in `$nospaces`, which is then compared to the original value in `$name`. If they're not the same, the filename has been changed, so the `$_renamed` property is reset to `true`. If the original name didn't contain any spaces, `$nospaces` and `$name` are the same, and the `$_renamed` property—which is initialized when the `Ps2_Upload` object is created—remains `false`.

   The next conditional statement controls whether to rename the file if one with the same name already exists. You'll add that code in the next step.

   The final line returns `$nospaces`, which contains the name that will be used when the file is saved.

2. Add the code that renames the file if another with the same name already exists:

```
protected function checkName($name, $overwrite) {
  $nospaces = str_replace(' ', '_', $name);
  if ($nospaces != $name) {
    $this->_renamed = true;
  }
  if (!$overwrite) {
    // rename the file if it already exists
    $existing = scandir($this->_destination);
    if (in_array($nospaces, $existing)) {
      $dot = strrpos($nospaces, '.');
      if ($dot) {
```

```
        $base = substr($nospaces, 0, $dot);
        $extension = substr($nospaces, $dot);
      } else {
        $base = $nospaces;
        $extension = '';
      }
      $i = 1;
      do {
        $nospaces = $base . '_' . $i++ . $extension;
      } while (in_array($nospaces, $existing));
      $this->_renamed = true;
    }
  }
  return $nospaces;
}
```

The first line of new code uses the `scandir()` function, which returns an array of all the files and folders in a directory (folder), and stores it in `$existing`.

The conditional statement on the next line passes `$nospaces` to the `in_array()` function to determine if the `$existing` array contains a file with the same name. If there's no match, the code inside the conditional statement is ignored, and the method returns `$nospaces` without any further changes.

If `$nospaces` is found the `$existing` array, a new name needs to be generated. To insert a number before the filename extension, you need to split the name by finding the final dot (period). This is done with the `strrpos()` function (note the double-r in the name), which finds the position of a character by searching from the end of the string.

It's possible that someone might upload a file that doesn't have a filename extension, in which case `strrpos()` returns `false`.

If a dot is found, the following line extracts the part of the name up to the dot and stores it in `$base`:

```
$base = substr($nospaces, 0, $dot);
```

The `substr()` function takes two or three arguments. If three arguments are used, it returns a substring from the position specified by the second argument and uses the third argument to determine the length of the section to extract. PHP counts the characters in strings from 0, so this gets the part of the filename without the extension.

If two arguments are used, `substr()` returns a substring from the position indicated by the second argument to the end of the string. So this line gets the filename extension:

```
$extension = substr($nospaces, $dot);
```

If `$dot` is false, the full name is stored in `$base`, and `$extension` is an empty string.

The section that does the renaming looks like this:

```
$i = 1;
```

```
do {
  $nospaces = $base . '_' . $i++ . $extension;
} while (in_array($nospaces, $existing));
```

It begins by initializing `$i` as 1. Then a `do. . . while` loop builds a new name from `$base`, an underscore, `$i`, and `$extension`. Let's say you're uploading a file called `menu.jpg`, and there's already a file with the same name in the upload folder. The loop rebuilds the name as `menu_1.jpg` and assigns the result to `$nospaces`. The loop's condition then uses `in_array()` to check whether `menu_1.jpg` is in the `$existing` array.

If `menu_1.jpg` already exists, the loop continues, but the increment operator (++) has increased `$i` to 2, so `$nospaces` becomes `menu_2.jpg`, which is again checked by `in_array()`. The loop continues until `in_array()` no longer finds a match. Whatever value remains in `$nospaces` is used as the new filename.

Finally, `$_renamed` is set to true.

Phew! The code is relatively short, but it has a lot of work to do.

3. Now you need to amend the `move()` method to call `checkName()`. The revised code looks like this:

```
public function move($overwrite = false) {
  $field = current($this->_uploaded);
  $OK = $this->checkError($field['name'], $field['error']);
  if ($OK) {
    $sizeOK = $this->checkSize($field['name'], $field['size']);
    $typeOK = $this->checkType($field['name'], $field['type']);
    if ($sizeOK && $typeOK) {
      $name = $this->checkName($field['name'], $overwrite);
      $success = move_uploaded_file($field['tmp_name'], $this->_destination ➥
        . $name);
      if ($success) {
        $message = $field['name'] . ' uploaded successfully';
        if ($this->_renamed) {
          $message .= " and renamed $name";
        }
        $this->_messages[] = $message;
      } else {
        $this->_messages[] = 'Could not upload ' . $field['name'];
      }
    }
  }
}
```

The first change adds `$overwrite = false` as an argument to the method. Assigning a value to an argument in the definition like this sets the default value and makes the argument optional. So, using `$upload->move()` automatically results in the `checkName()` method assigning a unique name to the file if necessary.

The `checkName()` method is called inside the conditional statement that runs only if the previous checks have all been positive. It takes as its arguments the filename transmitted through the `$_FILES` array and `$overwrite`. The result is stored in `$name`, which now needs to be used as part of the second argument to `move_uploaded_file()` to ensure the new name is used when saving the file.

The final set of changes assign the message reporting successful upload to a temporary variable `$message`. If the file has been renamed, `$_renamed` is `true` and a string is added to `$message` reporting the new name. The complete message is then assigned to the `$_messages` array.

4. Save `Upload.php`, and test the revised class in `file_upload.php`. Start by amending the call to the `move()` method by passing `true` as the argument like this:

```
$upload->move(true);
```

5. Upload the same image several times. You should receive a message that the upload has been successful, but when you check the contents of the `upload_test` folder, there's only one copy of the image. It has been overwritten each time.

6. Remove the argument from the call to `move()`:

```
$upload->move();
```

7. Save `file_upload.php`, and repeat the test, uploading the same image several times. Each time you upload the file, you should see a message that it has been renamed.

8. Repeat the test with an image that has a space in its filename. The space is replaced with an underscore, and a number is inserted in the name after the first upload.

9. Check the results by inspecting the contents of the `upload_test` folder. You should see something similar to Figure 6-7.

You can check your code, if necessary, against `Upload_04.php` in the `ch06` folder.

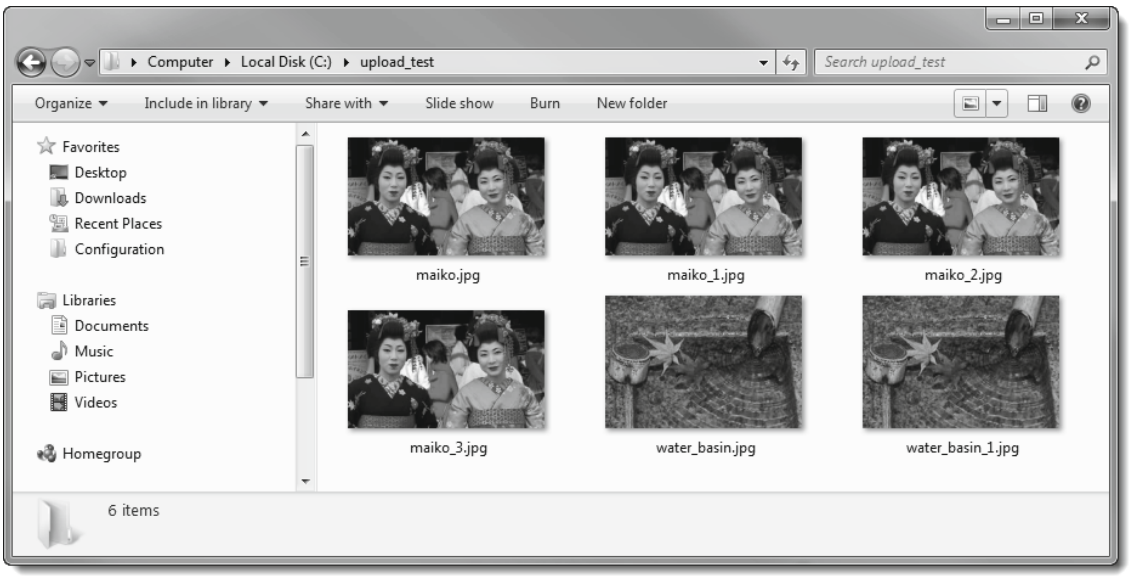


Figure 6-7. The class removes spaces from filenames and prevents files from being overwritten.

Uploading multiple files

You now have a flexible class for file uploads, but it can handle only one file at a time. Adding the `multiple` attribute to the file field's `<input>` tag permits the selection of multiple files in an HTML5-compliant browser. Older browsers also support multiple uploads if you add extra file fields to your form.

The final stage in building the `Ps2_Upload` class is to adapt it to handle multiple files. To understand how the code works, you need to see what happens to the `$_FILES` array when a form allows multiple uploads.

How the $_FILES array handles multiple files

Since `$_FILES` is a multidimensional array, it's capable of handling multiple uploads. In addition to adding the `multiple` attribute to the `<input>` tag, you need to add an empty pair of square brackets to the `name` attribute like this:

```
<input type="file" name="image[]" id="image" multiple>
```

Support for the `multiple` attribute is available in Firefox 3.6, Safari 4, Chrome 4, and Opera 10. At the time of this writing, it is not supported in any version of Internet Explorer, but that might change once the final version of IE9 is released. If you need to support older browsers, omit the `multiple` attribute, and create separate file input fields for however many files you want to upload simultaneously, Give each `<input>` tag the same `name` attribute followed by square brackets.

As you learned in Chapter 5, adding square brackets to the `name` attribute submits multiple values as an array. You can examine how this affects the `$_FILES` array by using `file_upload_06.php` or `file_upload_07.php` in the `ch06` folder. Figure 6-8 shows the result of selecting four files in an HTML5-

compliant browser. The structure of the `$_FILES` array is the same when a form uses separate input fields that share the same `name` attribute.

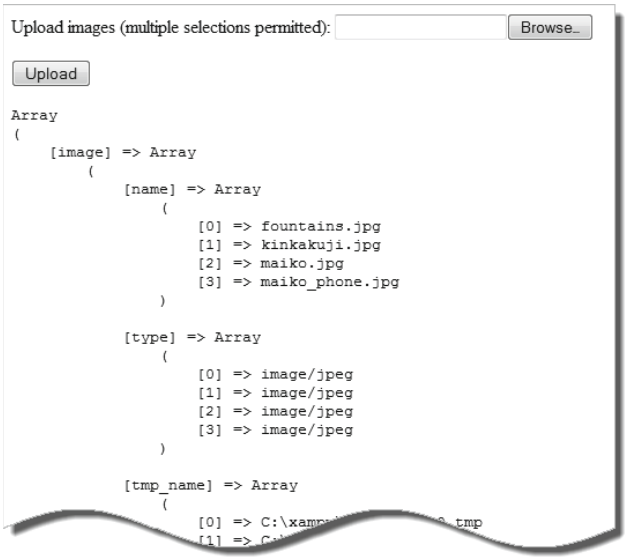


Figure 6-8. The `$_FILES` array can upload multiple files in a single operation.

Although this structure is not as convenient as having the details of each file stored in a separate subarray, the numeric keys keep track of the details that refer to each file. For example, `$_FILES['image']['name'][2]` relates directly to `$_FILES['image']['tmp_name'][2]`, and so on.

When you use the HTML5 `multiple` attribute on file input fields, older browsers upload a single file using the same structure, so the name of the file is stored as `$_FILES['image']['name'][0]`.

PHP Solution 6-6: Adapting the class to handle multiple uploads

This PHP solution shows how to adapt the `move()` method of the `Ps2_Upload` class to handle multiple file uploads. The class detects automatically when the `$_FILES` array is structured like Figure 6-8 and uses a loop to handle however many files are uploaded.

Continue working with your existing class file. Alternatively, use `Upload_04.php` in the `ch06` folder.

1. When you upload a file from a form designed to handle only single uploads, the `$_FILES` array stores the name like this (see Figure 6-2 earlier in this chapter):

   ```
   $_FILES['image']['name']
   ```

   When you upload a file from a form capable of handling multiple uploads the name of the first file is stored like this (see Figure 6-8):

   ```
   $_FILES['image']['name'][0]
   ```

In Figures 6-2 and 6-8, both refer to `fountains.jpg`. `$_FILES['image']['name']` is a string in Figure 6-2, but in Figure 6-8 it's an array.

So, by detecting whether the `name` element is an array, you can decide how to process the `$_FILES` array. If it's an array, you need to loop through it, passing the appropriate values to the `checkError()`, `checkSize()`, `checkType()`, and `checkName()` protected methods before passing it to `move_uploaded_file()`. The problem is that you need to add the index number for a multiple upload, but not for a single upload.

One solution is to require the upload form to use square brackets at the end of the `name` attribute, even for single uploads. This forces the form to submit the `$_FILES` array in the same format as shown in Figure 6-8. However, that's far from ideal.

The solution I have adopted is to split the `move()` method into two.

2. In the `move()` method select the code highlighted in bold, and cut it to your clipboard.

```
public function move($overwrite = false) {
  $field = current($this->_uploaded);
  $OK = $this->checkError($field['name'], $field['error']);
  if ($OK) {
    $sizeOK = $this->checkSize($field['name'], $field['size']);
    $typeOK = $this->checkType($field['name'], $field['type']);
    if ($sizeOK && $typeOK) {
      $name = $this->checkName($field['name'], $overwrite);
      $success = move_uploaded_file($field['tmp_name'], ➥
        $this->_destination . $name);
      if ($success) {
        $message = $field['name'] . ' uploaded successfully';
        if ($this->_renamed) {
          $message .= " and renamed $name";
        }
        $this->_messages[] = $message;
      } else {
        $this->_messages[] = 'Could not upload ' . $field['name'];
      }
    }
  }
}
```

3. Create a new protected method called `processFile()`, and paste the code from the `move()` method between the curly braces like this:

```
protected function processFile() {
  $OK = $this->checkError($field['name'], $field['error']);
  if ($OK) {
    $sizeOK = $this->checkSize($field['name'], $field['size']);
    $typeOK = $this->checkType($field['name'], $field['type']);
    if ($sizeOK && $typeOK) {
      $name = $this->checkName($field['name'], $overwrite);
```

```
        $success = move_uploaded_file($field['tmp_name'], ➥
          $this->_destination . $name);
        if ($success) {
          $message = $field['name'] . ' uploaded successfully';
          if ($this->_renamed) {
            $message .= " and renamed $name";
          }
          $this->_messages[] = $message;
        } else {
          $this->_messages[] = 'Could not upload ' . $field['name'];
        }
      }
    }
}
```

At the moment, this new method won't do anything because the arguments to `checkError()`, `checkSize()`, and so on are dependent on the `move()` method. To activate the `processFile()` method, you need to call it from the `move()` method, and pass the following values as arguments:

- `$field['name']`
- `$field['error']`
- `$field['size']`
- `$field['type']`
- `$field['tmp_name']`
- `$overwrite`

4. Amend the `move()` method like this:

```
public function move($overwrite = false) {
  $field = current($this->_uploaded);
  $this->processFile($field['name'], $field['error'], $field['size'], ➥
                     $field['type'], $field['tmp_name'], $overwrite);
}
```

5. Next, fix the arguments in the `processFile()` definition. Although, you could use the same variables, the code is cleaner and easier to understand if you change them both in the arguments declared between the parentheses and in the body of the method. Amend the code like this:

```
protected function processFile($filename, $error, $size, $type, ➥
                               $tmp_name, $overwrite) {
  $OK = $this->checkError($filename, $error);
  if ($OK) {
    $sizeOK = $this->checkSize($filename, $size);
    $typeOK = $this->checkType($filename, $type);
    if ($sizeOK && $typeOK) {
      $name = $this->checkName($filename, $overwrite);
```

```
        $success = move_uploaded_file($tmp_name, $this->_destination . $name);
        if ($success) {
          $message = "$filename uploaded successfully";
          if ($this->_renamed) {
            $message .= " and renamed $name";
          }
          $this->_messages[] = $message;
        } else {
          $this->_messages[] = "Could not upload $filename";
        }
      }
    }
  }
```

In other words, `$field['name']` has been converted to `$filename`, `$field['error']` to `$error`, and so on.

6. Splitting the functionality like this gives you a method that handles individual files. You can now use it inside a loop to handle multiple files one by one. Update the `move()` method like this:

```
public function move($overwrite = false) {
  $field = current($this->_uploaded);
  if (is_array($field['name'])) {
    foreach ($field['name'] as $number => $filename) {
      // process multiple upload
      $this->_renamed = false;
      $this->processFile($filename, $field['error'][$number], ➥
                         $field['size'][$number], $field['type'][$number], ➥
                         $field['tmp_name'][$number], $overwrite);
    }
  } else {
    $this->processFile($field['name'], $field['error'], $field['size'], ➥
                       $field['type'], $field['tmp_name'], $overwrite);
  }
}
```

The conditional statement checks if `$field['name']` is an array (`$field` is the current element of the `$_FILES` array, so `$field['name']` stores `$_FILES['image']['name']`). If it is an array, a `foreach` loop is created to handle each uploaded file. The key of each element is assigned to `$number`. The value of each element is assigned to `$filename`. These two variables give you access to each file and its details. Using the example in Figure 6-8, the first time the loop runs, `$number` is 0 and `$filename` is `fountains.jpg`. The next time, `$number` is 1 and `$filename` is `kinkakuji.jpg`, and so on.

Each time the loop runs, the `$_renamed` property needs to be reset to false. The values extracted from the current element of the `$_FILES` array are then passed to the `processFile()` method.

The existing code is wrapped in an `else` block that runs when a single file is uploaded. Don't forget the extra curly brace to close the `else` block.

7. Save `Upload.php`, and test it with `file_upload.php`. It should work the same as before.

8. If you're using an HTML5-compliant browser, add a pair of square brackets at the end of the `name` attribute in the file field, and insert the `multiple` attribute like this:

```
<input type="file" name="image[]" id="image" multiple>
```

   You don't need to make any changes to the PHP code above the `DOCTYPE` declaration. The code is the same for both single and multiple uploads.

9. Save `file_upload.php`, and reload it in your browser. Test it by selecting multiple files. When you click **Upload**, you should see messages relating to each file. Files that meet your criteria are uploaded. Those that are too big or of the wrong type are rejected.

   You can check your code against `Upload_05.php` in the `ch06` folder.

Using namespaces in PHP 5.3 and later

Prefixing the class names with `Ps2_` to avoid potential name clashes is a minor inconvenience when you're using only a handful of classes. But third-party libraries of PHP classes, such as the Zend Framework (`http://framework.zend.com/`), often consist of thousands of files in hundreds of folders. Naming the classes can become a major headache. The Zend Framework uses the convention of naming classes after the folder structure, so you can end up with unwieldy class names such as `Zend_File_Transfer_Adapter_Http`.

This led to the decision to implement namespaces in PHP 5.3. The idea is to prevent name collisions and very long class names. Instead of using underscores to indicate the folder structure, namespaces uses backslashes. So, instead of `Ps2_Upload`, the namespaced class name becomes `Ps2\Upload`. Although that doesn't sound like much of a gain, the advantage is that instead of referring all the time to `Ps2\Upload`, you can shorten it to `Upload`.

To declare a namespace, just use the `namespace` keyword followed by the name like this:

```
namespace Ps2;
```

This *must* be the first line of code after the opening PHP tag.

PHP Solution 6-7: Converting the class to use a namespace

This PHP solution shows how to convert the `Ps2_Upload` class to use a namespace. Your server must be running PHP 5.3 or later. It will not work in earlier versions of PHP.

1. Open your copy of `Upload.php` in the `Ps2` folder.

2. Declare the `Ps2` namespace immediately after the opening PHP tag, and change the class name from `Ps2_Upload` to `Upload` like this:

```
<?php
namespace Ps2;
class Upload {
```

3. Save `Upload.php`. That's all you need to do to the class definition.

4. Open `file_upload.php` in the `uploads` folder.

5. Locate the following line:

```
$upload = new Ps2_Upload($destination);
```

Change it to this:

```
$upload = new Ps2\Upload($destination);
```

6. Save `file_upload.php`, and test it. It should work as before.

7. Add the namespace declaration immediately after the opening PHP tag in `file_upload.php`:

```
<?php
namespace Ps2;
```

8. Change the code that instantiates the upload object like this:

```
$upload = new Upload($destination);
```

9. Save `file_upload.php`, and test it again. It should continue to work as before.

You can find examples of the code in `file_upload_ns.php` and `Upload_ns.php` in the `ch06` folder.

This has been a relatively trivial example, which sidesteps many subtleties of using namespaces. To learn more about using namespaces, see `www.phparch.com/2010/03/29/namespaces-in-php/`, as well as `http://docs.php.net/manual/en/language.namespaces.faq.php`.

Using the upload class

The `Ps2_Upload` class is simple to use. Just include the class definition in your script, and create a `Ps2_Upload` object by passing the file path to the upload folder as an argument like this:

```
$destination = 'C:/upload_test/';
$upload = new Ps2_Upload($destination);
```

The path to the upload folder must end in a trailing slash.

The class has the following public methods:

- `setMaxSize()`: Takes an integer and sets the maximum size for each upload file, overriding the default 51200 bytes (50kB). The value must be expressed as bytes.
- `getMaxSize()`: Reports the maximum size in kB formatted to one decimal place.
- `addPermittedTypes()`: Takes an array of MIME types, and adds them to the types of file accepted for upload. A single MIME type can be passed as a string.
- `setPermittedTypes()`: Similar to `addPermittedTypes()`, but replaces existing values.
- `move()`: Saves the file(s) to the destination folder. Spaces in filenames are replaced by underscores. By default, files with the same name as an existing file are renamed by inserting a number in front of the filename extension. To overwrite files, pass `true` as an argument.
- `getMessages()`: Returns an array of messages reporting the status of uploads.

Points to watch with file uploads

Uploading files from a web form is easy with PHP. The main causes of failure are not setting the correct permissions on the upload directory or folder, and forgetting to move the uploaded file to its target destination before the end of the script. Letting other people upload files to your server, however, exposes you to risk. In effect, you're allowing visitors the freedom to write to your server's hard disk. It's not something you would allow strangers to do on your own computer, so you should guard access to your upload directory with the same degree of vigilance.

Ideally, uploads should be restricted to registered and trusted users, so the upload form should be in a password-protected part of your site. Also, the upload folder does not need to be inside your site root, so locate it in a private directory whenever possible unless you want uploaded material to be displayed immediately in your web pages. Remember, though, there is no way PHP can check that material is legal or decent, so immediate public display entails risks that go beyond the merely technical. You should also bear the following security points in mind:

- Set a maximum size for uploads both in the web form and on the server side.
- Restrict the types of uploaded files by inspecting the MIME type in the `$_FILES` array.
- Replace spaces in filenames with underscores or hyphens.
- Inspect your upload folder on a regular basis. Make sure there's nothing in there that shouldn't be, and do some housekeeping from time to time. Even if you limit file upload sizes, you may run out of your allocated space without realizing it.

Chapter review

This chapter has introduced you to creating a PHP class. If you're new to PHP or programming, you might have found it tough going. Don't be disheartened. The `Ps2_Upload` class contains more than 170 lines of code, and some of it is complex, although I hope the descriptions have explained what the code is doing at each stage. Even if you don't understand all the code, the `Ps2_Upload` class will save you a lot of time. It implements the main security measures necessary for file uploads, yet using it involves as little as ten lines of code:

```
if (isset($_POST['upload'])) {
  require_once('classes/Ps2/Upload.php');
  try {
    $upload = new Ps2_Upload('C:/upload_test/');
    $upload->move();
    $result = $upload->getMessages();
  } catch (Exception $e) {
    echo $e->getMessage();
  }
}
```

If you found this chapter a struggle, come back to it later when you have more experience, and you should find the code easier to understand.

In the next chapter, you'll learn some techniques for inspecting the contents of files and folders, including how to use PHP to read and write text files.