SEMESTER – VI - III BSc IT

18BIT61C – Programming in PHP

Prepared by Dr.N.Thenmozhi

UNIT III: Creating Classes: Introducing classes and objects-defining and using classes-using advanced OOPs concepts-using constructors and destructors-extending classes-adjusting visibility settings-working with files and directories: reading local file-remote file-specific segments of a file-writing files-processing directories-performing other file and directory operations.

TEXT BOOKS

1. Vikram Vaswani, "PHP: A beginners guide", TMH Hill, 1stedition, 2010 (Unit-I to IV).

REFERENCE BOOKS

1. Matt Doyle, " Beginning PHP 5.3", Wiley India pvt. Ltd, First edition, 2010.
2. Luke welling and Laura Thomson, "PHP and MySQL Web Development", 5th Edition, 2016.

Creating Classes

In addition to allowing you to create your own functions, PHP also allows you to group related functions together using a *class.* Classes are the fundamental construct behind object-oriented programming (OOP), a programming paradigm that involves modeling program behavior into "objects" and then using these objects as the basis for your applications.

Up until recently, PHP's support for OOP was limited; however, PHP 5.0 introduced a new object model that allowed programmers significantly greater flexibility and ease of use when working with classes and objects.

Introducing Classes and Objects

Think of a *class* as a miniature ecosystem: it's a self-contained, independent collection of variables and functions, which work together to perform one or more specific (and usually related) tasks. Variables within a class are called *properties*; functions are called *methods.* Classes serve as templates for *objects,* which are specific instances of a class. Every object has properties and methods corresponding to those of its parent class. Every object instance is completely independent, with its own properties and methods, and can thus be manipulated independently of other objects of the same class.

To put this in more concrete terms, consider an example: an Automobile class that contain properties for color and make, and methods for acceleration, braking, and turning. It's possible to derive two independent objects from this Automobile class, one representing a Ford and the other a Honda. Each of these objects would have methods for acceleration, braking, and turning, as well as specific values for color and make. Each object instance could also be manipulated independently: for example, you could change the Honda's color without affecting the Ford, or call the Ford's acceleration method without any impact on the Honda.

Defining and Using Classes

In PHP, classes are defined much like functions: a class definition begins with the class keyword, which is followed by the class name and a pair of curly braces. Figure 3.1 illustrates the relationship between a class and its object instances visually.

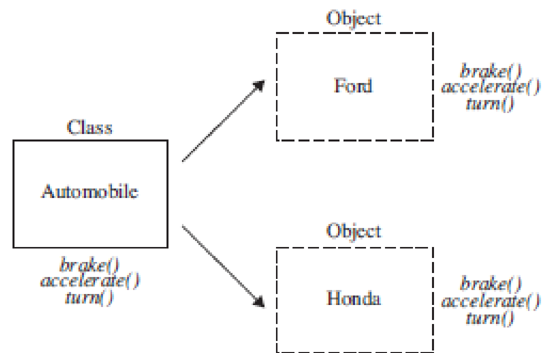

Figure 3.1 The relationship between classes and objects

The complete class definition must be enclosed within these braces; in most cases, this definition consists of property (variable) definitions followed by method (function) definitions. To see what a class definition looks like, review the following listing: it contains a definition for an Automobile class, with two properties named $color and $make and methods named accelerate(), brake(), and turn():

```
<?php
	// class definition
	class Automobile
	{
	// properties
		public $color;
		public $make;
	// methods
		public function accelerate()
		{
			echo 'Accelerating...';
		}
		public function brake()
		{
			echo 'Slowing down...';
		}
		public function turn()
		{
			echo 'Turning...';
		}
	}
?>
```

Once a class has been defined, objects can be created from the class with the new keyword. Class methods and properties can directly be accessed through this object instance. Here’s an example, which creates an instance of the Automobile class and assigns it to $car, and then sets the object’s properties and invokes object methods (note the -> symbol used to connect objects to their properties or methods):

```
<?php
// instantiate object
	$car = new Automobile;
```

```
// set object properties
        $car->color = 'red';
        $car->make = 'Ford Taurus';
// invoke object methods
        $car->accelerate();
        $car->turn();
?>
```

To access or change a class method or property from within the class itself, it's necessary to prefix the corresponding method or property name with $this, which refers to "this" class. To see how this works, consider this revision of the preceding example, which sets a class property named $speed and then modifies this property from within the accelerate() and brake() functions:

```
<?php
// class definition
        class Automobile
        {
        // properties
                public $color;
                public $make;
                public $speed = 55;
        // methods
                public function accelerate()
                {
                        $this->speed += 10;
                        echo 'Accelerating to ' . $this->speed . '...';
                }
                public function brake()
                {
                        $this->speed -= 10;
                        echo 'Slowing down to ' . $this->speed . '...';
                }
                public function turn()
                {
                        $this->brake();
                        echo 'Turning...';
                        $this->accelerate();
                }
        }
?>
```

And now, when you invoke these functions, you'll see the effect of changes in $speed:

```
<?php
// instantiate object
        $car = new Automobile;
// invoke methods
// output: 'Accelerating to 65...'
// 'Slowing down to 55...'
// 'Turning...'
```

```
// 'Accelerating to 65...'
        $car->accelerate();
        $car->turn();
?>
```

Encrypting and Decrypting Text

This next listing defines a class named Jumbler, which allows users to encrypt (and decrypt) text using a simple encryption algorithm and a user-supplied numeric key. Take a look at the class definition (*jumbler.php*):

```
<?php
// class definition
        class Jumbler
        {
        // properties
                public $key;
        // methods
        // set encryption key
                public function setKey($key)
                {
                        $this->key = $key;
                }
        // get encryption key
                public function getKey()
                {
                        return $this->key;
                }
        // encrypt
                public function encrypt($plain)
                {
                for ($x=0; $x<strlen($plain); $x++)
                {
                $cipher[] = ord($plain[$x]) + $this->getKey() + ($x * $this->getKey());
                }
                return implode('/', $cipher);
                }
// decrypt
        public function decrypt($cipher)
        {
                $data = explode('/', $cipher);
                $plain = '';
                for ($x=0; $x<count($data); $x++) {
                $plain .= chr($data[$x] - $this->getKey() - ($x * $this->getKey()));
                }
                return $plain;
                }
        }
?>
```

This class has a single property and four methods:

- The $key property holds the numeric key entered by the user. This key is used to perform the encryption.
- The setKey() method accepts an argument and sets the $key property to that value.
- The getKey() method returns the value of the $key property.
- The encrypt() function accepts a plain-text string and “jumbles” it using the key.
- The decrypt() function accepts an encrypted string and restores the original plaintext string from it using the key.

A quick word about the internals of the encrypt() and decrypt() methods, before proceeding to a usage example. When encrypt() is invoked with a plain-text string, it steps through the string and calculates a numeric value for each character. The numeric value is the sum of

- The character’s ASCII code, as returned by the ord() function
- The numeric key set by the user through the setKey() method
- The product of the numeric key and the character’s position in the string

Each number returned after this calculation is added to an array, and once the entire string is processed, the elements of the array are joined into a single string, separated by slashes, with implode().

Table 3-1 has a brief illustration of how the word 'Ant' is converted into the encrypted string '410/800/1151' using this method.

The decryption routine reverses this process: it first splits the encrypted ciphertext string into individual numbers using the slashes as delimiters, and adds them to an array. It then obtains the character corresponding to each number, by subtracting

- The numeric key set by the user through the setKey() method; and
- The product of the numeric key and the character’s position in the string

Table 3-1 An Example Encryption Run

Character $c	Position $p	ord($c)	Key $key	$key *$p	Total
'A'	0	65	345	0	410
'n'	1	110	345	345	800
't'	2	116	345	690	1151

from the number, and then using the chr() function to retrieve the ASCII character corresponding to the remainder. The characters returned through this process are concatenated into a single string and returned to the caller.

Using Advanced OOP Concepts

PHP’s object model also supports many more advanced features, giving developers a great deal of power and flexibility in building OOP-driven applications.

Using Constructors and Destructors

PHP makes it possible to automatically execute code when a new instance of a class is created, using a special class method called a *constructor*. You can also run code when a class instance ends using a so-called *destructor*. Constructors and destructors can be implemented by defining functions named __construct() and __destruct() within the class, and placing object (de)initialization code within them.

Here’s a simple example illustrating how this works:

```
<?php
```

```
    // define class
    class Machine
    {
    // constructor
        function __construct()
        {
            echo "Starting up...\n";
        }

        // destructor
        function __destruct()
        {
            echo "Shutting down...\n";
        }
    }
    // create an object
    // output: "Starting up..."
    $m = new Machine();
    // then destroy it
    // output: "Shutting down..."
    unset($m);
?>
```

Extending Classes

For most developers, extensibility is the most powerful reason for using the OOP paradigm. Put very simply, *extensibility* implies that a new class can be derived from an existing one, inheriting all the properties and methods of the parent class and adding its own new properties and methods as needed. Thus, for example, a Human class could extend a Mammal class, which is itself an extension of a Vertebrate class, with each new extension adding its own features as well as inheriting the features of its parent. In PHP, extending a class is as simple as attaching the extends keyword and the name of the class being extended to a class definition, as in the following example:

```
<?php
    class Mammal
    {
        / class definition
    }
    class Human extends Mammal
    {
        // class definition
    }
?>
```

With such an extension, all the properties and methods of the parent class become available to the child class and can be used within the class' program logic. To illustrate, consider the following listing:

```
<?php
    // parent class definition
    class Mammal
    {
        public $age;
```

```
		function __construct()
		{
			echo 'Creating a new ' . get_class($this) . '...';
		}
		function setAge($age)
		{
			$this->age = $age;
		}
		function getAge()
		{
			return $this->age;
		}
		function grow()
		{
			$this->age += 4;
		}
	}
	// child class definition
	class Human extends Mammal
	{
		public $name;
		function __construct()
		{
			parent::__construct();
		}
		function setName($name)
		{
			this->name = $name;
		}
		function getName()
		{
			return $this->name;
		}
		function grow()
		{
			$this->age += 1;
			echo 'Growing...';
		}
	}
?>
```

This listing contains two class definitions:

1. Mammal, the parent class, which contains the $age property and the methods setAge(), getAge(), and grow();

2. Human, which extends Mammal and inherits all of Mammal's properties and methods. Human contains the additional $name property and the additional methods setName() and getName(); its grow() method overrides the method of the same name in the parent class.

Notice also that Human's __construct() method internally calls Mammal's __construct() method; the special keyword parent provides an easy shortcut to refer to the current class' parent.

Here's an example of how you'd use the extended class:

```
<?php
        $baby = new Human;
        $baby->setAge(1);
        $baby->setName('Tonka');
        echo $baby->getName() . ' is now ' . $baby->getAge() . ' year(s) old...';
        $baby->grow();
        $baby->grow();
        echo $baby->getName() . ' is now ' . $baby->getAge() . ' year(s) old.';
?>
```

Following figure shows the output of this listing. From the output, it should be clear that even though the class definition for Human doesn't explicitly contain getAge() and setAge() methods, nor the $age property, instances of the Human class can still use these methods, as they are inherited from the parent Mammal class.

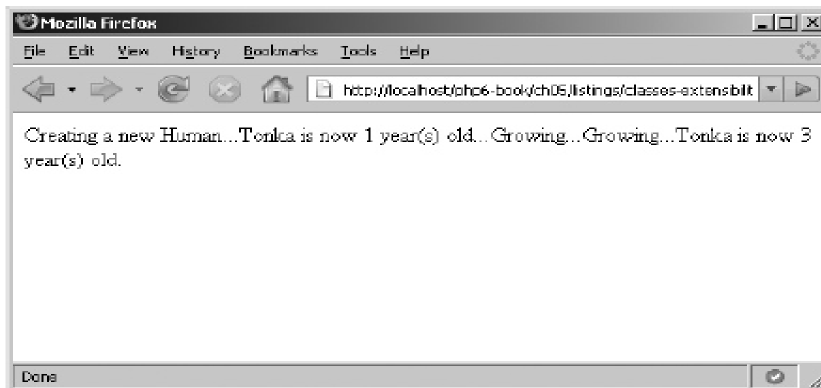

Figure 3.2 Using inherited methods of a class

Once you begin working with inheritance and extensibility, it's easy to quickly become overwhelmed with long and complex class trees. For this reason, PHP provides an instance of operator, to test if an object is an instance of a particular class. This operator returns true if the object instance has the named class anywhere in its parent tree.

Here's an example of it in action:

```
<?php
// class tree
        class Vertebrate
        {
        }
        class Mammal extends Vertebrate
        {
        }
        class Human extends Mammal
        {
```

```
	}
	$baby = new Human;
	// output: true
	echo ($baby instanceof Vertebrate) ? 'true' : 'false';
?>
```

Adjusting Visibility Settings

The difference between "local" and "global" scope, how variables used inside a function are invisible to the main program. When working with classes, PHP allows you to exert even greater control over the *visibility* of object properties and methods. Three levels of visibility exist, ranging from most visible to least visible; these correspond to the public, protected, and private keywords.

All the property and method definitions in earlier examples are prefixed with the keyword public; this sets the corresponding class methods and properties to be "public" and allows a caller to manipulate them directly from the main body of the program. This "public" visibility is the default level of visibility for any class member (method or property) under PHP.

You can explicitly mark a particular property or method as private or protected, depending on how much control you want to cede over the object's internals. "Private" methods and properties are only visible within the class that defines them, while "protected" methods and properties are visible to both their defining class and any child (inherited) classes. Attempts to access these properties or methods outside their visible area typically produce a fatal error that stops script execution.

Consider the following example:

```
<?php
// class tree
	class Mammal
	{
		public $name;
		protected $age;
		private $species;
	}
	class Human extends Mammal
	{
	}
	$mammal = new Mammal;
	$mammal->name = 'William'; // ok
	$mammal->age = 3; // fatal error
	$mammal->species = 'Whale'; // fatal error
	$human = new Human;
	$human->name = 'Barry'; // ok
	$human->age = 1; // fatal error
	$human->species = 'Boy'; // undefined
?>
```

Working with Files and Directories

In reality, though, your PHP script will need to work with data retrieved from disk files, SQL result sets, XML documents, and many other data sources. PHP comes with numerous built-in functions to access these data sources.

Reading Files

PHP's file manipulation API is extremely flexible: it lets you read files into a string or into an array, from the local file system or a remote URL, by lines, bytes, or characters.

Reading Local Files

The easiest way to read the contents of a disk file in PHP is with the file_get_contents() function. This function accepts the name and path to a disk file, and reads the entire file into a string variable. Here's an example:

```
<?php
	// read file into string
	$str = file_get_contents('example.txt') or die('ERROR: Cannot find file');
	echo $str;
?>
```

An alternative method of reading data from a file is PHP's file() function, which accepts the name and path to a file and reads the entire file into an array, with each element of the array representing one line of the file. To process the file, all you need do is iterate over the array using a foreach loop. Here's an example, which reads a file into an array and then displays it using such a loop:

```
<?php
	// read file into array
	$arr = file('example.txt') or die('ERROR: Cannot find file');
	foreach ($arr as $line)
	{
		echo $line;
	}
?>
```

Reading Remote Files

Both file_get_contents() and file() also support reading data from URLs, using either the HTTP or FTP protocol. Here's an example, which reads an HTML file off the Web into an array:

```
<?php
	// read file into array
	$arr = file('http://www.google.com') or die('ERROR: Cannot find file');
	foreach ($arr as $line)
	{
		echo $line;
	}
?>
```

In case of slow network links, it's sometimes more efficient to read a remote file in "chunks," to maximize the efficiency of available network bandwidth. To do this, use the fgets() function to read a specific number of bytes from a file.

```
<?php
	// read file into array (chunks)
	$str = '';$fp = fopen('http://www.google.com', 'r') or die('ERROR: Cannot open file');
	while (!feof($fp))
```

```
	{
		$str .= fgets($fp,512);
	}
	fclose($fp);
	echo $str;
?>
```

This listing introduces four new functions, First, the fopen() function: it accepts the name of the source file and an argument indicating whether the file is to be opened in read ('r'), write ('w'), or append ('a') mode, and then creates a pointer to the file. Next, a while loop calls the fgets() function continuously in a loop to read a specific number of bytes from the file and append these bytes to a string variable; this loop continues until the feof() function returns true, indicating that the end of the file has been reached. Once the loop has completed, the fclose() function destroys the file pointer.

Reading Specific Segments of a File

Reading only a specific block of lines from a lines. This can be accomplished with a combination of PHP's fseek() and fgets() functions. Consider the next example, which sets up a user-defined function named readBlock() and accepts three arguments: the filename, the starting line number, and the number of lines to return from the starting point:

```
<?php
	// function definition
	// read a block of lines from a file
	function readBlock($file, $start=1, $lines=null)
	 {
		// open file
		$fp = fopen($file, 'r') or die('ERROR: Cannot find file');
		// initialize counters
		$linesScanned = 1;
		$linesRead = 0;
		$out = '';
		// loop until end of file
		while (!feof($fp))
		{
			// get each line
			$line = fgets($fp);
			// if start position is reached
			// append line to output variable
			if ($linesScanned >= $start)
			{
				$out .= $line;
				$linesRead++;
			// if max number of lines is defined and reached
			// break out of loop
			if (!is_null($linesRead) && $linesRead == ($lines))
			{
				break;
			}
		}
```

```
            $linesScanned++;
        }
            return $out;
        }
        echo readBlock('example.txt', 3, 4);
?>
```

Within readBlock(), a loop iterates through the file line by line, until the starting line number is reached (a line counter named $linesScanned keeps track of the current line number, incrementing by 1 on each iteration of the loop). Once the starting line is reached, it (and all subsequent lines) are read into a string variable until the specified maximum number of lines are processed or until the end of the file is reached.

Writing Files

The flip side of reading data from files is writing data to them. And PHP comes with a couple of different ways to do this as well. The file_put_contents() function, accepts a filename and path, together with the data to be written to the file, and then writes into the file. Here's an example:

```
<?php
        // write string to file
        $data = "A fish \n out of \n water\n";
        file_put_contents('output.txt', $data)

        or die('ERROR: Cannot write file');
        echo 'Data written to file';
?>
```

If the file specified in the call to file_put_contents() already exists on disk, file_put_contents() will overwrite it by default. If, instead, you'd prefer to preservethe file's contents and simply append new data to it, add the special FILE_APPEND flag to your file_put_contents() function call as a third argument. Here's an example:

```
<?php
        // write string to file
        $data = "A fish \n out of \n water\n";
        file_put_contents('output.txt', $data, FILE_APPEND)

        or die('ERROR: Cannot write file');
        echo 'Data written to file';
?>
```

An alternative way to write data to a file is to create a file pointer with fopen(), and then write data to the pointer using PHP's fwrite() function. Here's an example of this technique:

```
<?php
        // open and lock file
        // write string to file
        // unlock and close file
        $data = "A fish \n out of \n water\n";
        $fp = fopen('output.txt', 'w') or die('ERROR: Cannot open file');
        flock($fp, LOCK_EX) or die ('ERROR: Cannot lock file');
        fwrite($fp, $data) or die ('ERROR: Cannot write file');
```

```
	flock($fp, LOCK_UN) or die ('ERROR: Cannot unlock file');
	fclose($fp);
	echo 'Data written to file';
?>
```

Notice the flock() function from the preceding listing: this function "locks" a file before reading or writing it, so that it cannot be accessed by another process. Doing this reduces the possibility of data corruption that might occur if two processes attempt to write different data to the same file at the same instant. The second parameter to flock() specifies the type of lock: LOCK_EX creates an exclusive lock for writing, LOCK_SH creates a non-exclusive lock for reading, and LOCK_UN destroys the lock.

Processing Directories

PHP also allows developers to work with directories on the file system, iterating through directory contents or moving forward and backward through directory trees. Iterating through a directory is a simple matter of calling PHP's DirectoryIterator object, as in the following example, which uses the DirectoryIterator to read a directory and list each file within it:

```
<?php
	// initialize iterator with name of
	// directory to process
	$dit = new DirectoryIterator('.');
	// loop over directory
	// print names of files found
	while($dit->valid())
	{
		if (!$dit->isDot())
		{
			echo $dit->getFilename() . "\n";
		}
		$dit->next();
	}
	unset($dit);
?>
```

Here, a DirectoryIterator object is initialized with a directory name, and the object's rewind() method is used to reset the internal pointer to the first entry in the directory. A while loop, which runs so long as a valid() entry exists, can then be used to iterate over the directory. Individual filenames are retrieved with the getFilename() method, while the isDot() method can be used to filter out the entries for the current (.) and parent (..) directories. The next() method moves the internal pointer forward to the next entry. You can also accomplish the same task with a while loop and some of PHP's directory manipulation functions as in the following listing:

```
<?php
	// create directory pointer
	$dp = opendir('.') or die ('ERROR: Cannot open directory');
	// read directory contents
	// print filenames found
	while ($file = readdir($dp))
	{
```

```
		if ($file != '.' && $file != '..')
		{
			echo "$file \n";
		}
	}
	// destroy directory pointer
	closedir($dp);
?>
```

Here, the opendir() function returns a pointer to the directory named in the function call. This pointer is then used by the readdir() function to iterate over the directory, returning a single entry each time it is invoked. It's then easy to filter out the . and .. directories, and print the names of the remaining entries. Once done, the closedir() function closes the file pointer.

TIP scandir() function : accepts a directory name and returns an array containing a list of the files within that directory together with their sizes. It's then easy to process this array with a foreach loop. In some cases, you might need to process not just the first-level directory, but also its subdirectories and sub-subdirectories.

```
<?php
	// function definition
	// print names of files in a directory
	// and its child directories
	function printDir($dir, $depthStr='+')
	{
		if (file_exists($dir))
		{
		// create directory pointer
		$dp = opendir($dir) or die ('ERROR: Cannot open directory');
		// read directory contents
		// print names of files found
		// call itself recursively if directories found
		while ($file = readdir($dp))
		{
			if ($file != '.' && $file != '..')
			{
				echo "$depthStr $dir/$file \n";
				if (is_dir("$dir/$file"))
				{
					printDir("$dir/$file", $depthStr.'+');
				}
			}
		}
		// close directory pointer
		closedir($dp);
		}
	}
	// print contents of directory
	// and all children
```

```
	if (file_exists('.'))
	{
		echo '<pre>';
		printDir('.');
		echo '<pre>';
	}
?>
```

The printDir() function in this listing might appear complex, but it's actually quite simple. It accepts two arguments: the name of the top-level directory to use, and a "depth string," which indicates, via indentation, the position of a particular file or directory in the hierarchy. Using this input, the function opens a pointer to the named directory and begins processing it with readdir(), printing the name of each directory or file found. In the event that a directory is found, the depth string is incremented by an additional character and the printDir() function is itself recursively called to process that subdirectory. This process continues until no further files or directories remain to be processed.

Performing Other File and Directory Operations

PHP comes with a whole range of file and directory manipulation functions, which allow you to check file attributes; copy, move, and delete files; and work with file paths and extensions.

Checking if a File or Directory Exists

If you try reading or appending to a file that doesn't exist, PHP will typically generate a fatal error. To access or read/write from/to a directory that doesn't exist also generate errors. To avoid these error messages, always check that the file or directory you're attempting to access already exists, with PHP's file_exists() function.

Table 2: Common PHP File and Directory Functions

Function	What It Does
file_exists()	Tests if a file or directory exists
filesize()	Returns the size of a file in bytes
realpath()	Returns the absolute path of a file
pathinfo()	Returns an array of information about a file and its path
stat()	Provides information on file attributes and permissions
is_readable()	Tests if a file is readable
is_writable()	Tests if a file is writable
is_executable()	Tests if a file is executable
is_dir()	Tests if a directory entry is a directory
is_file()	Tests if a directory entry is a file
copy()	Copies a file
rename()	Renames a file
unlink()	Deletes a file
mkdir()	Creates a new directory
rmdir()	Removes a directory
include() / require()	Reads an external file into the current PHP script

```
<?php
	// check file
```

```
	if (file_exists('somefile.txt'))
	{
		$str = file_get_contents('somefile.txt');
	}
	else
	{
		echo 'Named file does not exist. ';
	}
	// check directory
	if (file_exists('somedir'))
	{
		$files = scandir('somedir');
	}
	else
	{
		echo 'Named directory does not exist.';
	}
?>
```

Calculating File Size

To calculate the size of a file in bytes, call the filesize() function with the filename as argument:

```
<?php
	// get file size
	// output: 'File is 1327 bytes.'
	if (file_exists('example.txt'))
	{
		echo 'File is ' . filesize('example.txt') . ' bytes.';
	}
	else
	{
		echo 'Named file does not exist. ';
	}
?>
```

Finding the Absolute File Path

To retrieve the absolute file system path to a file, use the realpath() function, as in the next listing:

```
<?php
	// get file path
	// output: 'File path: /usr/local/apache/htdocs/
	// /php-book/ch06/listings/example.txt'
	if (file_exists('example.txt'))
	{
		echo 'File path: ' . realpath('example.txt');
	}
	else
	{
		echo 'Named file does not exist. ';
```

```
    }
?>
```

You can also use the pathinfo() function, which returns an array containing the file's path, name, and extension. Here's an example:

```
<?php
    // get file path info as array
    if (file_exists('example.txt'))
    {
        print_r(pathinfo('example.txt'));
    }
    else
    {
        echo 'Named file does not exist. ';
    }
?>
```

Retrieving File Attributes

You can obtain detailed information on a particular file, including its ownership, permissions, and modification and access times, with PHP's stat() function, which returns this information as an associative array. Here's an example:

```
<?php
    // get file information
    if (file_exists('example.txt'))
    {
        print_r(stat('example.txt'));
    }
    else
    {
        echo 'Named file does not exist. ';
    }
?>
```

You can check if a file is readable, writable or executable with the is_readable(), is_writable(), and is_executable() functions. The following example illustrates their usage:

```
<?php
    // get file information
    // output: 'File is: readable writable'
    if (file_exists('example.txt'))
    {
        echo 'File is: ';
        // check for readable bit
        if (is_readable('example.txt'))
        {
            echo ' readable ';
        }
        // check for writable bit
        if (is_writable('example.txt'))
        {
            echo ' writable ';
```

```
        }
        // check for executable bit
        if (is_executable('example.txt'))
        {
            echo ' executable ';
        }
    }
    else
    {
        echo 'Named file does not exist. ';
    }
?>
```

The is_dir() function returns true if the argument passed to it is a directory, while the is_file() function returns true if the argument passed to it is a file. Here’s an example:

```
<?php
    // test if file or directory
    if (file_exists('example.txt'))
    {
        if (is_file('example.txt'))
        {
            echo 'It\'s a file.';
        }
        if (is_dir('example.txt'))
        {
            echo 'It\'s a directory.';
        }
    }
    else
    {
        echo 'ERROR: File does not exist.';
    }
?>
```

Creating Directories

To create a new, empty directory, call the mkdir() function with the path and name of the directory to be created:

```
<?php
    if (!file_exists('mydir'))
    {
        if (mkdir('mydir'))
        {
            echo 'Directory successfully created.';
        }
        else
        {
            echo 'ERROR: Directory could not be created.';
        }
    }
    else
```

```
	{
		echo 'ERROR: Directory already exists.';
	}
?>
```

Copying Files

You can copy a file from one location to another by calling PHP's copy() function with the file's source and destination paths as arguments. Here's an example:

```
<?php
	// copy file
	if (file_exists('example.txt'))
	{
		if (copy('example.txt', 'example.new.txt'))
		{
			echo 'File successfully copied.';
		}
		else
		{
			echo 'ERROR: File could not be copied.';
		}
	}
	else
	{
		echo 'ERROR: File does not exist.';
	}
?>
```

It's important to note that if the destination file already exists, the copy() function will overwrite it.

Renaming Files or Directories

To rename or move a file (or directory), call PHP's rename() function with the old and new path names as arguments. Here's an example that renames a file and a directory, moving the file to a different location in the process:

```
<?php
	// rename/move file
	if (file_exists('example.txt'))
	{
		if (rename('example.txt', '../example.new.txt'))
		{
			echo 'File successfully renamed.';
		}
		else
		{
			echo 'ERROR: File could not be renamed.';
		}
	}
	else
	{
```

```
		echo 'ERROR: File does not exist.';
	}
	// rename directory
	if (file_exists('mydir'))
	{
		if (rename('mydir', 'myotherdir'))
		{
			echo 'Directory successfully renamed.';
		}
		else
		{
			echo 'ERROR: Directory could not be renamed.';
		}
	}
	else
	{
		echo 'ERROR: Directory does not exist.';
	}
?>
```

As with copy(), if the destination file already exists, the rename() function will overwrite it.

CAUTION

PHP will only allow you to copy, delete, rename, create, and otherwise manipulate a file or directory if the user "owning" the PHP script has the privileges necessary to perform the task.

Removing Files or Directories

To remove a file, pass the filename and path to PHP’s unlink() function, as in the following example:

```
<?php
	// delete file
	if (file_exists('dummy.txt'))
	{
		if (unlink('dummy.txt'))
		{
			echo 'File successfully removed.';
		}
		else
		{
			echo 'ERROR: File could not be removed.';
		}
	}
	else
	{
		echo 'ERROR: File does not exist.';
	}
?>
```

To remove an empty directory, PHP offers the rmdir() function, which does the reverse of the mkdir() function. If the directory isn't empty, though, it's necessary to first remove all its contents (including all subdirectories) and only then call the rmdir() function to remove the directory. You can do this manually, but a recursive function is usually more efficient—here's an example, which demonstrates how to remove a directory and all its children:

```
<?php
	// function definition
	// remove all files in a directory
	function removeDir($dir)
	{
		if (file_exists($dir))
		{
		// create directory pointer
			$dp = opendir($dir) or die ('ERROR: Cannot open directory');
		// read directory contents
		// delete files found
			// call itself recursively if directories found
			while ($file = readdir($dp))
			{
				if ($file != '.' && $file != '..')
				{
					if (is_file("$dir/$file"))
					{
						unlink("$dir/$file");
					}
					else if (is_dir("$dir/$file"))
					{
						removeDir("$dir/$file");
					}
				}
			}
			// close directory pointer
			// remove now-empty directory
			closedir($dp);
			if (rmdir($dir))
			{
				return true;
			}
			else
			{
				return false;
			}
		}
	}
	// delete directory and all children
	if (file_exists('mydir'))
	{
		if (removeDir('mydir'))
```

```
			{
				echo 'Directory successfully removed.';
			}
			else
			{
				echo 'ERROR: Directory could not be removed.';
			}
		}
		else
		{
			echo 'ERROR: Directory does not exist.';
		}
?>
```

Here, the removeDir() function is a recursive function that accepts one input argument: the name of the top-level directory to remove. The function begins by creating a pointer to the directory with opendir() and then iterating over the directory's contents with a while loop. For each directory entry found, the is_file() and is_dir() methods are used to determine if the entry is a file or a sub-directory; if the former, the unlink() function is used to delete the file and if the latter, the removeDir() function is called recursively to again process the subdirectory's contents. This process continues until no files or subdirectories are left; at this point, the top-level directory is empty and can be removed with a quick rmdir().

Reading and Evaluating External Files

To read and evaluate external files from within your PHP script, use PHP's include() or require() function. A very common application of these functions is to include a standard header, footer, or copyright notice across all the pages of a Web site. Here's an example:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
	<head>
		<title></title>
	</head>
	<body>
		<?php require('header.php'); ?>
		<p/>
		This is the page body.
		<p/>
		<?php include('footer.php'); ?>
	</body>
</html>
```

Here, the script reads two external files, *header.php* and *footer.php,* and places the contents of these files at the location of the include() or require() call. It's important to note that any PHP code to be evaluated within the files included in this manner must be enclosed within <?php ... ?> tags.