PHP 7 & MySQL: An Object Oriented Introduction

специален Български PHP издание

William B. Sanders, Ph.D.
Multimedia Design and Development Program
The University of Hartford
# Table of Contents

Chapter 1: Object Oriented PHP 7 Programming  
Chapter 2: Data and Data Types  
Chapter 3: PHP 7 Operators  
Chapter 4: The Atoms that Make Objects  
Chapter 5: Basic Language Structures  
Chapter 6: Functions and Methods  
Chapter 7: Interfaces, Abstract Classes, Methods and Properties  
Chapter 8: Data Input and Passing Values to a PHP 7 Class  
Chapter 9: The Client-Request Model  
Chapter 10: Web Pages as PHP Objects and PHP File Work  
Chapter 11: Communicating with Databases  
Chapter 12: Working with Tables  
Chapter 13: Data Manipulation  
Chapter 14: Applying Programming Principles in PHP 7  
Appendix A: Installing PHP/MySQL

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5-14</td>
</tr>
<tr>
<td>2</td>
<td>15-30</td>
</tr>
<tr>
<td>3</td>
<td>31-45</td>
</tr>
<tr>
<td>4</td>
<td>46-65</td>
</tr>
<tr>
<td>5</td>
<td>66-80</td>
</tr>
<tr>
<td>6</td>
<td>81-93</td>
</tr>
<tr>
<td>7</td>
<td>94-103</td>
</tr>
<tr>
<td>8</td>
<td>104-135</td>
</tr>
<tr>
<td>9</td>
<td>136-145</td>
</tr>
<tr>
<td>10</td>
<td>146-155</td>
</tr>
<tr>
<td>11</td>
<td>156-163</td>
</tr>
<tr>
<td>12</td>
<td>164-173</td>
</tr>
<tr>
<td>13</td>
<td>174-195</td>
</tr>
<tr>
<td>14</td>
<td>196-xx</td>
</tr>
<tr>
<td>A</td>
<td>xxx</td>
</tr>
</tbody>
</table>
Chapter 1: Object Oriented PHP 7 Programming

You can learn just about any programming language by learning different key words and operations. If you stack up those statements, they will execute sequentially so that something happens. Most non-professional developers learn how to program using this method.

At some point, you find that you have to re-use certain operations again and again, and you may move on to another technique where you create different procedures in the form of functions. The same procedure can be used wherever you want in a program to perform the group of operations within the function simply by calling it from anywhere in the program. That’s a big improvement over the first method.

Using either of the above two methods, called *sequential* and *procedural* programming, you will find that your programs get big quickly. The bigger the program, the more complex things get, and at some point you end up dealing with a briar patch of a program where adding and changing code becomes so cumbersome you either have to give up or start over. Eventually, you may discover *Object Oriented Programming* (OOP) to solve these problems.

Getting Started with OOP and PHP 7

Suppose that instead of learning programming either by stacking statements on top of one another or by writing procedures and mixing them in with sequences, you learn OOP from the very beginning. It’s like learning French, Chinese, Portuguese, Hebrew, English, Arabic, Spanish or Swahili. If you’re born into a family that speaks *any* language, you’ll pick up that language in a fashion you consider *natural*. The same is true with OOP. If you learn PHP 7 using OOP, it seems like the *natural* way of programming. So what’s the point of learning PHP 7 one way and then *unlearn* it only to relearn in OOP? Why not just start with OOP? Then you only have to learn it once.

**Objects**

We live in a world of objects. Your computer is an object with different characteristics and things that it does. It has a keyboard, a mouse or touch pad, a screen, storage and a processor. You interact with it using a keyboard, touch and mouse and get feedback on your screen. You may have additional interaction devices such as a microphone, a camera, a drawing pad and stylus. When you press the letter “R” it sends a message to the current program that an “R” has been pressed. Figure 1-1 illustrates a computer as an object with different properties:
When you do something with your computer, you may not know exactly the path the electrons take when you press a key, touch the screen, or move the mouse. You probably don’t even think about it. You just know that if you press the “R” key, something will happen based on the program you’re running. The action in a word processor program, places the letter “R” on the screen, but in a game program, the same action makes a character turn to the right.

Likewise, your car is an object. You send messages in your car by pressing on the accelerator pedal, the brake pedal and turning the steering wheel. Shifting the gears makes it go forward or backwards or puts it in park. It’s an object that has different properties that send messages to make it do what you want. When we think of our car or computer we think of them as entities, but we know that they’re made up of several different components.
PHP Objects: Classes to Objects
An object in computer programs is much like any other object. It has properties (characteristics) and things that it does (actions). It sends and receives messages, and works in conjunction with other objects. All objects are built as classes. When we use one of the classes, we refer to it as *an instance of a class* or simply as *an object*. Let’s build your first PHP object to see how easy it is. Enter the code in Listing 1-1 and save it as *FirstClass.php* in a root directory.

**Be careful.** The constructor function `__construct()` has two (2) underscores. It should look like this:

```
__construct()
```

but the two underscores run together to display one big one. *If you put in just one underscore, your program will not work.*

**Listing 1-1: FirstClass.php**

```php
<?php

class FirstClass
{
    function __construct()
    {
        $hi="Hello from PHP";
        echo $hi;
    }
}

//Create instance
$test=new FirstClass();
?>
```

**Note:** Before you can test this, you must set up PHP on a server, either on your computer or use a remote server. *Appendix A* shows you how. *Macintosh* computers have PHP and an Apache Web server built in already, but you must configure your Mac to use them. Also, your Mac may have an older version of PHP (pre-PHP 7), and you will need to download and install PHP 7 if the pre-loaded version of PHP is older than PHP 7. You can freely download and use Apache servers for computers using both Windows and Linux operating systems (including *Raspberry Pi*). You can also download and install PHP 7 and MySQL at no cost. *If you test your program on an older version of PHP, some of these examples will not work.*

To test it, you must do it from a server. That’s because PHP is a *server side* language. Unlike HTML, you cannot double-click the file on the desktop and expect it to run in your browser. You have to enter the http address. If you set up your basic root
directory as *localhost* on your computer and place the file in the *root* directory, you would enter in your browser window:

[http://localhost/FirstClass.php](http://localhost/FirstClass.php)

If you're using the root on Mac OS, your address would be something like:

[http://localhost/~computername/FirstClass.php](http://localhost/~computername/FirstClass.php)

As soon as you run the program, your screen shows:

```
Hello from PHP
```

You may be underwhelmed by this example, but you have just completed your first Object Oriented PHP program!

**Analyzing a Class**

Keep in mind that in looking at what goes to make up a class, you're looking at an object. So, let's step through the whole little program:

**PHP Wrapper:** All PHP programs have beginning and ending tags—a PHP wrapper:

```
<?php

   //PHP code goes here

?>
```

You can enter a PHP program in the middle of an HTML listing, and the tags tell the browser that the processing is now an operation handled by the server. The PHP server processes the code and returns HTML to the client (your computer). Then the browser handles the HTML. In this book, we'll try to *avoid* mixing up PHP and HTML, though. You'll see how an HTML document can be part of a PHP class.

**Class Name:** All classes begin with a name, and typically the name begins with a capital letter. The word “class” precedes the name or label of the class. This is followed by an open curly brace.

```
class FirstClass
{
```

Some programmers prefer to place the opening curly brace on the same line as the class declaration, and so you might see a class begin with,

```
class FirstClass {
```
Either way is fine. I prefer the former because it helps to see the structure of the class better.

**Constructor Function:** A constructor function is a function that automatically launches as soon as the class is used (instantiated.) In PHP, a special function name, `__construct`, indicates that the function is a constructor function.

```php
function __construct()
{
    // Code goes here
}
```

Using `__construct` makes it very easy to distinguish the constructor method from all other methods in the class. (In older versions of PHP, and other programming languages, you can use the name of the class as the constructor name, but in PHP 7, that throws an error.)

**Declaring a Variable:** All variables in PHP begin with a dollar sign ($). A variable *in a class* is called a *property*. You assign a value to a variable, which can be a *literal*—actual text, numbers or Boolean value,—another variable or constant, a function or an object (class). In the following example, two variables are declared:

```php
$hi="Hello from PHP";
and
$test=new FirstClass();
```

In the first instance, the *literal*, “Hello from PHP,” is assigned to the variable `$hi`, and in the second instance the object (class) `FirstClass()` is assigned to the `$test` variable. *As soon as $test is assigned new FirstClass(), the constructor function launches.*

**Adding an echo Statement:** After assigning a value to the `$hi` variable, the next line is an *echo* statement that sends the contents of `$hi` to the screen. PHP uses “echo” and “print” (as well as some variations of print) to send data to the screen.

```php
echo $hi;
```

You can see that within the constructor function are two statements. The first one assigns a value to a variable, and the second one sends that value to the screen.

**Close those Curly Braces:** After you place everything you want in the constructor function, end it with a closing curly brace (`}`). Likewise, with your class definition, you must remember to *place a closing curly brace at the end of it*. The following shows the curly braces you need for a class and methods in that class:

```php
class MyClass
```
function __construct()
{
  //PHP code
}

function anyMethod()
{
  //PHP code
}

All functions within a class must have its second curly brace closed before the closing class curly brace.

Launching the Class: The final operation within the PHP wrapper is to launch the class. As you have seen, all that is required is that the class be instantiated (assigned with a new statement) to a variable. The operation must occur outside of the class—outside of the curly braces where the class is defined. The lines,

$test=new MyClass();

instantiates the class as an object (instance of the class) and because the constructor function launches as soon as the class is instantiated, you see the results on the screen.

Adding Methods
As you know, a variable inside a class is called a property. Functions in classes are called methods. (The constructor function is sort of a method because it’s inside a class, but it launches with class’ instantiation so we’ll just consider it a constructor function even though it technically has method-like features.) In many cases (perhaps most), we really don’t need to specify a constructor function. All classes automatically generate a constructor function, but other than initializing properties, they do not automatically launch a method unless instructed to do so in a specified constructor function. Often you may want to instantiate a class, but you don’t want to fire any functions just yet. For example, suppose you have a salutations class. This class does nothing more than provide output for greetings—coming and going. When you want a salutation, you just call the greeting you want; however, you do not launch any of the other methods until you need them.

Defining Methods
A PHP method is made up of:

1. **Visibility** (optional): Any of the following visibility modifiers may be used: public, protected or private. If no modifier is used, the default is public.
2. **Name:** Generally beginning with a lowercase letter to differentiate them from classes.

3. **Parentheses:** The definition requires parentheses whether or not they endose parameters.

4. **Parameters** (optional): The parameter list may or may not include type declarations (data types).

5. **Method body:** Enclosed in curly braces, place the code, including defining local variables.

6. **Return type:** New in PHP 7 is the (optional) addition of a *return type*. You may specify the type of data that a method returns. (e.g., a string, an integer, an object of a certain kind.) If you have declared *strict typing*; you must include a return type.

A method’s *signature* is made up of the method’s name and parameter types. In PHP, where typing can be dynamic, you can include type declarations for user-defined classes and interfaces, build-in classes and interfaces and arrays. For example, the following method includes type declarations (array, Iterator) and a return type (int): 

```php
private function doAnalyze(array $groupA, Iterator $counter): int
{
    //code
}
```

The method’s *signature* is made up of its name, parameters and return type:

```php
doAnalyze(array $groupA, Iterator $counter): int{}
```

So, *doAnalyze()* is a method with the signature:

- doAnalyze (name)
- $groupA (parameter)
- $counter (parameter)
- int (return type)

Figure 1-2 illustrates another view of a signature:
You will see references to a method’s signature in OOP discussions, and just remember that the signature only consists of function name, (type declaration for parameter, parameter name, and return type) of a method. Probably the most useful way to think of a PHP signature is in terms of its name, parameters (and parameter type if type declarations are used) and return type (if a return type is used.). In Figure 1-2, most of the elements of a signature in parenthesis are optional. You will see plenty of methods with nothing but a name and empty parameters; but that is perfectly normal and acceptable.

**Tip to Remember:** You can have a type declaration for each parameter in a function; but only a single return type for that function.

### Accessing Properties and Methods

To access a method or property in a class instance (object), PHP uses the -> symbol made up of a dash (-) and greater-than symbol (>). You use the format:

```php
$myClassInstance = new MyClass();
$myClassInstance->myMethod();
```

You can use the same instance to call (launch) as many methods from the class as you want. You can call the same method with the same instance more than once as well.

In Listing 1-2, you see no constructor class, and so when the class is instantiated as an object, nothing happens. In order to make something happen, you need to specify what function you want by calling it from an instance of the class.

### Listing 1-2: Salutations.php

```php
<?php

```
class Salutations
{
    public function hello()
    {
        $hi="Hello from PHP";
        echo $hi;
        echo "<br>";
    }
    public function goodbye()
    {
        $bye="PHP bids you farewell.";
        echo $bye;
    }
}

//This is outside the class definition
//but it is inside the PHP wrapper
 тест=new Salutations();
 тест->hello();
 тест->goodbye();
?>

When you test the program, you will see:

Hello from PHP
PHP bids you farewell.

When declaring a class with no constructor function in the code, you must call a method for something to happen.

**Designating Class Properties and Visibility Keywords**

To designate a variable as a class property it is first declared with a given *visibility*. In PHP, visibility is defined by the following keywords:

- **private**
- **protected**
- **public**

Each one refers to how the property may be accessed.

**Private:** The property may be used only from within the same class.

**Protected:** The property may be used from the same class or a child of the class. (In Chapter 2, you will learn about inheritance and parent and child classes.)
**Public:** Any object from any class may use the property.

You may have noticed in Listing 1-2 that two methods were designated as *public*. Public visibility is necessary so that an instance outside of the class can use it. If you use the *private* visibility keyword in PHP, it has the effect of *encapsulating* the property and it can only be accessed from within the class with the following reference:

```
$this->propertyName;
```

For example, the following snippet declares a variable as a property and then another line assigns the property a value.

```php
private $myProperty; //Declare a property
$this->myProperty="Private Property!"; //Use the property
```

If you place a property with private visibility in a class method with a public visibility, other classes or programs can use the property but they cannot change it. Consider this next class and how the class’s property is employed:

- **Listing 1-3: PrivateProperty.php**

```php
<?php
class PrivateProperty {
    private $greeting;

    public function greet() {
        $this->greeting="Welcome to my PHP program";
        echo $this->greeting;
        echo "<br>";
    }
}

$test=new PrivateProperty();
$test->greet();
?>
```

When you test the program, you will see:

```
Welcome to my PHP program
```
As you can see in Listing 1-3, the $test variable is able to access the greet() method because greet() is public. In turn, the greet() method was able to access the property $greeting because the method and property are part of the same class.

**It’s Just Objects, Properties and Methods**

OOP is no big mystery. Classes are made up of properties and methods. Other programs and classes can use the classes, their objects and methods. Organizing your programs using OOP techniques helps to create them in small, sensible modules that can be used and re-used. The best statement that describes OOP is that by David Chelimsky:

> In procedural programming, a process is expressed in one place in which a series of instructions are coded in order. Whereas in [object oriented programming], a process is expressed as a succession of messages across objects. One object sends a message to another which does part of the process and then sends a new message off to another object, which handles part of the process, etc. You modify a process by reorganizing the succession of messages rather than changing a procedure.


Unlike either sequential or procedural programming, the larger your OOP programs get, the easier it is to control them and use or re-use different parts. The classes are instantiated as objects, and their operations get done what you want your program to do.

**Do-It-Yourself Exercises**

The following exercises help you to move ahead on your own.

- Create a class that automatically puts your name on the screen as soon as the class is instantiated.
- Create a class with a public property and assign that property a value from outside of the class and display it on the screen.
- Create a class with a private method and a public method. Have the public method launch the private method.
Chapter 2: Data and Data Types

Every programming language can handle different types of data, and PHP is no different. Some data types are best used for storing text while other data types are better for storing numbers. This chapter examines the different types of data and how they can be used as properties in classes and used between classes.

PHP Primitives are Easy

The built-in types of data supported by a language are generally referred to as primitives. Typical primitives include types for storing text, numbers and Boolean (true/false) values. This section examines the primitives in PHP.

Before looking at the individual types of data, you need to know that in the past PHP used only weak typing. That means the same variable or property can change the type of data it stores simply by exchanging a value. For example, consider the following value assigned to a variable:

```php
$myVar = "Easy street";
```

The variable `$myVar` stores the string value “Easy street.” (Keep in mind that all PHP variables begin with a dollar sign —$.) You can change it to:

```php
$myVar = 66;
```

The variable `$myVar` now contains a number. Generally, changing data types with the same variable label (name) is a poor practice; however, you can do it in PHP. However, remember,

Just because you can do something doesn’t mean it’s a good idea.

The examples in this book avoid changing the data type of a variable—even with dynamic typing. Some programming languages such as Java, C# and C++ are statically typed and do not allow the same variable to change data types. Try running the class in Listing 2-1 to see how weak typing works:

Listing 2-1: WeakType.php

```php
<?php
class WeakType
{
    function __construct()
    {
```
You will see the following output:

Easy street
66

That’s what you can do with weakly typed variables but not statically typed variables. Note also that the script assigned an HTML tag ("<br>") to the variable. As you will see throughout the book, HTML tags can be used to format output in PHP. Remember, though, changing the type of a variable is not a good idea as a general practice. Changing the contents of a variable, though, is an important standard programming practice.

**Scalar Data Types**

The most basic kind of data type can store only a single piece of data at any one time. These are called scalar types. Scalar data constitute the basic building blocks of PHP and include the following:

- **Integer**: Whole numbers only. 5; 22; 8,442
- **Float**: Numbers with floating point decimals. 3.432; .04; 22.19
- **String**: Any sequence of characters enclosed in quotation marks. “I love PHP”; “123 Elm Street”; “Yikes!”; “<strong>Be Bold</strong>”
- **Boolean**: Only the values true or false or 1 or 0.

Fire up the script in Listing 2-2 to see how each type can be used.

**Listing 2-2: Scalar.php**

```php
<?php
class Scalar
{
    public function __construct()
    {
        $cr="<br>";
        $wholeNumber=250;
    
```
$fraction = 67.44;
$myString="Scalar sounds funny";
$justTheFacts=true;

echo $wholeNumber;
echo $cr;
echo $fraction;
echo $cr;
echo $myString;
echo $cr;
echo $justTheFacts;
}
}

$scalarTest=new Scalar();
?>

When you test the program, you will see the following:

250
67.44
Scalar sounds funny
1

You may be surprised to see the value 1 instead of true in the last output character, but in PHP the Boolean value true is treated as 1 and false as 0. The next several subsections examine each of these data types in more detail.

**Integers**

Many objects can only be expressed as values without fractions. For example, a room may have a capacity of 103 people. Similarly, a car carrier may only be able to carry 7 mid-size cars, 10, compact cars and 5 large cars. We could also say that the car carrier could handle 2 large cars and 1 mid-size car and 4 compact cars. Each car, no matter what size, is still a single unit.

As with most languages, you can mix integers with floating point numbers. For example, the following snippet would resolve in a valid result:

$val1 = 7; //integer
$val2 = 3.5; //float
$total = $val1 + $val2;//Add the two variables
echo $total; //float

So when working with both floating point numbers and integers, you don’t have to worry that they will not work together.
Floats
Because virtually all financial transactions require decimal points, floating point numbers are essential for any language that can be used for business. However, you may need to control the precision of the floating-point number. For example, PHP returns 6.2264150943396 when the float, 5.3, divides the integer, 33. In financial transactions, you only need a precision of 2 when expressing values in dollars and cent. So the value,

$6.2264150943396

is awkward. Fortunately, PHP has a built-in function to control float precision. Listing 2-3 provides an example of using the round() function to set the precision of data output to two numbers after the decimal point:

Listing 2-3: DollarsAndCents.php

```php
<?php
class DollarsAndCents
{
    public function __construct()
    {
        $cost=22.99;
        $tax = .06; // Set $tax at 6% (.06)
        $cost += $cost * $tax; // Add $cost to quotient of $cost multiplied $tax
        $total=$cost;
        echo $total;
        echo "<br>";
        echo round($total,2); // rounds to 2 decimal pts.
    }
}
$DCobject=new DollarsAndCents();
?>
```

When you test the class you will see the following output:

31.712406
31.71

The top value is what you get when you multiply 22.99 by .06 and add it to 22.99, and the bottom value shows the effect of rounding to two decimal places.

The way the round() function works can be seen most clearly when rounding two similar numbers. If the rounding value is 5 or higher, the value goes to the next value, while 4 or lower rounds to the lower value. For example, look carefully at the following:
class Rounder
{
    public function __construct()
    {
        $a = 1.23456;
        $b = 1.23500;
        echo round($a, 2) . "<br/>";
        echo round($b, 2) . "<br/>";
    }
}
$worker = new Rounder();
?>

When you test Rounder you will see the following output:

1.23
1.24

The round() function is set to "2" and so the next value after the second decimal determines whether it will be rounded up to the next value or remain at the same value. The values "23" followed by a "4" results in a decimal of "23," but when the second decimal is followed by a "5" it rounds up to the next decimal value, "24."

Strings
Strings can be understood as literal displays of characters. It treats numbers, text and symbols the same. For example, you may have an address, 123 Elm Street. The number "123" is just an identifier and not a real number subject to math operations. (It wouldn't make any sense to add 123 Elm Street to 148 Elm Street!)
To join one or more strings, PHP uses the dot character (.)—the period. The process works something like adding and is called concatenation. Consider, the following snippet:

$firstString = "123";
$secondString = "&nbsp;Elm Street";
$concat = $firstString . $secondString;
echo $concat;

The output would be,

123 Elm Street

Notice the space before Elm Street in the variable $secondString. By adding an HTML space symbol (&nbsp;) between the quotation marks results in a space “character” in the string—in fact you can add as many as you want. You can do the
same thing by just pressing the space bar for each space you want, but then the number of spaces is more difficult to determine.

You can also use &nbsp; in a variable for spacing such as the following snippet of PHP code shows:

```php
//Assign =&nbsp; to a variable
$sp =&nbsp;
$sp5 =&nbsp;$sp$sp$sp$sp$sp
$sp3 =&nbsp;$sp$sp$sp
echo "a $sp5 b" . "<br />";
echo "a $sp3 b" . "<br />";
echo "a $sp b" . "<br />"
```

The output shows the space between string “a” and “be.”

```
 a    b
 a  b
 a  b
```

Keep in mind that &nbsp; is a string (just like letters are strings), and it must include the ampersand (&) at the beginning and the semi-colon (;) at the end, all encased in quotation marks. (Be careful, the semi-colon (;) in &nbsp; cannot be used to signal the end of a line.)

**Booleans**

The final PHP scalar is the Boolean. As a data type, Boolean values have a long and important role in programming. In the early days of computer programming, Booleans were important because they are both small (they take up little memory) and they’re fast (easy to process). Likewise, in conditional statements, Boolean values determine branches.

PHP Boolean values are stored as 0 or 1, and while they can be assigned either `true` or `false` values (without using quotation marks on the values) they work just like integers. Chapter 5 explains conditional statements, in detail and you can better see the value of Boolean scalars there.

**Coercive and Strict Typing in PHP 7**

The help the computer process data and generally speed up that process, PHP has introduced two types of strong typing; **coercive** (default) and **strict**. Whenever you add the data type before a parameter name or issue a return type, the program enforces that type.
Coercive Typing
Coercive typing is the default strong typing in PHP 7. If you make an error of type, your will get a "Notice" but not a "Fatal Error." Of course, you want to have neither a notice nor error. All you have to do is to make sure that the data type you use matches that specified in the type declaration. For example, consider Listing 2-4:

Listing 2-4: Team.php

```php
<?php
class Team {
    public function showMembers($stem, $mems) {
        echo "The $stem team has $mems members.";
    }
}
$worker = new Team();
$worker->showMembers("soccer", 11);?
```

The first parameter must be a `string` and the second an `integer`. When this program is run, the following appears on the screen,

The soccer team has 11 members.

If you put in something other than a string and integer, you’ll get (at least) a notice or an error message. To see how the data type is enforced, make the following changes:

```php
$worker->showMembers(rugby, 14.7);
```

As you will see, the results throw a Notice:

Notice: Use of undefined constant rugby - assumed 'rugby' in /Users/wdsanders/Sites/php/MWD310/MWD310_s16/Programs/Team.php on line 10

The rugby team has 14 members.

By omitting the quote marks around "rugby" at first the program considers it a constant instead of a string but decides that you really meant 'rugby' and so makes it into a string, and then chops off the 0.7 from 14.7 to change it from a floating point (float) to an integer (int). Note also, that floating point values are always rounded down where you have a coercive integer. (If they were rounded up, you’d have the correct number of players on a rugby team.)

The same coercive typing rules are applied to return types in functions. For example,
function myFunction() : int
{
    return 45.434;
}

will only return 45 because the return type is an integer.

### Strict Typing

Unlike coercive typing, you have to declare strict typing as the first statement in a file. Further, a strict typing directive applies only to that file in which it has been declared. Use the following directive to declare the strict mode:

```php
<?
declare(strict_types=1);
```

Once in a file with the strict mode, it is not as forgiving as the coercive mode. If, for example, you declare an integer return type and then use a floating point value, the strict mode throws an error. Consider the following:

```php
<?php
declare(strict_types=1);
class StrictMode
{
    public function elStricto(): int
    {
        return 4.892;
    }
}
$worker=new StrictMode();
echo $worker->elStricto();
?>
```

Fatal error: Uncaught TypeError: Return value of StrictMode::elStricto() must be of the type integer, float returned in /Users/wdsanders/Sites/php/MWD310/MWD310_s16/Programs/Strict.php:7 Stack trace:
#0 /Users/wdsanders/Sites/php/MWD310/MWD310_s16/Programs/Strict.php(12): StrictMode->elStricto()
#1 {main} thrown in /Users/wdsanders/Sites/php/MWD310/MWD310_s16/Programs/Strict.php on line 7

You will find that if you want to become accustomed to strong typing, use the strict typing directive in all of your files. Dynamic typing is easier (and far more sloppy) than strict typing, but ultimately, your programs will run faster and more efficiently with strict typing.
Strict typing helps you pay attention to what you’re creating so that as your program grows, you won’t have problems down the line. Also, you still have a good deal of flexibility with strict typing. Consider the following that uses three named data types (float, int and string) and creates and returns a single string that uses all of the data:

```php
<?php
declare(strict_types=1);
class ShoppingCart {
    public function shopper(float $price, int $quant) : string {
        $total = ($price * $quant) * 1.06;
        $display = "Your total bill is 
round($total,2);
        return $display;
    }
}
$worker = new ShoppingCart();
echo $worker->shopper(5.95, 4);
?>
```

The results show:

```
Your total bill is $25.23
```

It multiplied the price (float) by the number purchased (int) and then added the sales tax of 6%. Finally, by combining the rounded value down to two decimal points and the string literal, it created a new string stored in the variable, `$display`. (Whenever you combine [concatenate] a string with a non-string, the combo results in a new string.)

**Compound Data Types: Objects and Arrays**

PHP has two kinds of compound data types; *objects* and *arrays*. While scalar data types can only hold one value at a time, compound data types can handle many. You’ve seen several examples of object data types already and may not have known it. Each time that an example instantiated a class that variable holds more than one value. It contains all of the properties and methods of the class. For instance, in Listing 2-3, the line,

```php
$DCobject=new DollarsAndCents();
```

instantiates an instance of the class *DollarsAndCents*. The variable `$DCobject` is an *object*. The other compound data type is an *array*. The next two subsections introduce both compound data types. Before going on, though, take a look at Figure
2-1 that illustrates the differences between scalar and compound data types:

![Scalar data types can only store a single value at any one time.](image)

```
$myScalar="briefcase";
```

```
$myArray[0]="heart";
$myArray[1]="disk";
$myArray[2]="house";
```

```
$myArray=array("heart", "disk", "briefcase");
```

![Compound data types can store multiple values simultaneously.](image)

**Figure 2-1: Scalar and Compound Data Types**

Both compound and scalar data types can hold the *different types of values*, but compound types can hold multiple values and different types at one time but scalar variables only hold a single value. Both can change the values they store to different values. In keeping with the idea that *you should not change value types*, once a value type has been established in either a variable or array element, it should stay the same.

**Objects**

A class is nothing but a collection of properties and methods that accomplish a task. When the class is instantiated (declared in a variable using the `new` statement), it is an object. In examples in Chapter 1 and this chapter, you have seen how simple it is to create a class and create objects from a class.

Classes and the objects created by instantiating a class make it easy to manage different operations. One example in Chapter 1 shows how a single object can hold different greetings. Suppose you wanted to greet people in different languages. You could create a class that displays “Hello” is several different language and all you’d have to specify is the name of the language. Listing 2-4 illustrates how the variable (object) `$greetings` has different values:
Listing 2-4: HelloTranslator

```php
<?php

class HelloTranslator
{
    private $hello;
    private $cr="<br>";

    public function french(): string
    {
        $this->hello="bonjour";
        return $this->hello . $this->cr;
    }

    public function spanish(): string
    {
        $this->hello="hola";
        return $this->hello . $this->cr;
    }

    public function german(): string
    {
        $this->hello="Servus";
        return $this->hello . $this->cr;
    }

    public function indonesian(): string
    {
        $this->hello="halo";
        return $this->hello . $this->cr;
    }

    public function portuguese(): string
    {
        $this->hello="ola";
        return $this->hello . $this->cr;
    }
}

$greetings=new HelloTranslator();
echo $greetings->french();
echo $greetings->spanish();
echo $greetings->german();
echo $greetings->indonesian();
echo $greetings->portuguese();
?>
```
Because the object $greetings has multiple values it is a compound data type. The scalar data types can have only a single value. When you test an object, you can see the multiple values in the output:

bonjour
hola
Servus
halo
ola

The example shows the different methods in the object. However, if you look at the code in Listing 2-4, you can see that each method has a property, $this->hello. Each method assigns the property a different value that it displays on the screen using the echo statement in the call to the object’s method. The $cr property with a value of “<br>” adds a linefeed.

Note: Where possible, you do not want to have an output statement (e.g., echo, print, printf) within a method. Rather, you should use a return statement, and then have the output statement in the calling statement—the statement that calls the method as shown in Listing 2-4. However, for purposes of illustration in code snippets and example classes, you will find echo used throughout the discussions and examples in this book.

Arrays
An array is a special type of object that does not have to instantiate a class. You can store several elements in an array. The elements are stored using either an integer or string value. The default base value for an array is 0.

Numeric Array. One way to create an array is to use the array constructor and list the items you want in the array object. For example, the following line creates an array with three elements,

$myArray= array("stuff", "more stuff", "still more");

Each of the three strings you placed into the array is an element. To use any one of those elements, you simply address it by its position in the array (from left to right) beginning with 0. The following shows how to access the data in the array:

$cr="<br>";
echo $myArray[0] . $cr;
echo $myArray[1] . $cr;
echo $myArray[2] . $cr;

That’s all there is to it. You will see more sophisticated ways to access the elements in an array, but the easiest way to simply to address each element with its index
(numeric position in the array). As long as you remember to start counting at 0 instead of 1, it’s simple. The above output is:

```
stuff
more stuff
still more
```

You can mix numbers, strings and Booleans in the same array. Unlike scalar data, compound data types are not restricted to one data type at a time. Listing 2-5 illustrates how an array can store different types of data:

*Listing 2-5: ArrayClass.php*

```php
<?php
class ArrayClass {
    public function __construct() {
        $cr="<br>";
        $myArray=array("first", "second", 3, true, "cow", "kitchen sink");
        echo $myArray[0] . $cr;
        echo $myArray[1] . $cr;
        echo $myArray[2] . $cr;
        echo $myArray[3] . $cr;
        echo $myArray[4] . $cr;
        echo $myArray[5] . $cr . $cr;
        echo count($myArray) . " items are in the array.";
    }
}
$test=new ArrayClass();
?>
```

As you can see, strings, a number and a Boolean value are all included in the array named `$myArray`. When you test the program, you will see:

```
first
second
3
1
cow
kitchen sink
6 items are in the array.
```
The six items in parenthesis make up the contents of the array. (The 1 reflects the Boolean value true.) Using the count() function, you can see how many elements the array contains. In this case it's 6.

**Associative Array.** A second type of array is an associative array. Instead of using numeric keys to specify each array element, an associative array uses strings. For example, suppose you want to generate a list of Greek first names with an equal mix of male and female. Each will have a named letter from the Greek alphabet as a key (instead of a number).

- alpha: Adonia
- beta: Marcos
- gamma: Soterios
- delta: Zena

In an associative array, you use the name key => (equal-arrow) symbol and the data value. For example,

```php
$greek = array('alpha' => 'Adonia', 'beta' => 'Marcos', 'gamma' => 'Soterios', 'delta' => 'Zena');
```

assigns the key delta with the value Zena. To access the array element, use the string key to specify the element.

```php
echo $greek['delta'];
```

Making an associative array involves creating a list of keys with a list of values. These are key-value pairs. To display all of the elements in the array, you can use a foreach loop. This special loop iterates through any object and pulls out the properties that you can send to the screen using the echo statement. (You can find more details on the foreach loop in Chapter 5.) Listing 2-6 shows how:

**Listing 2-6: EzAssociative.php**

```php
<?php
class EzAssociative {
    private $greek;

    private function showGreeks() : array {
        $this->greek = array(
            'alpha' => 'Adonia',
            'beta' => 'Marcos',
            'gamma' => 'Soterios',
            'delta' => 'Zena'
        );
    }
}
```
return $this->greek;
}

public function getGreeks()
{
    $lambda = $this->showGreeks();
    foreach($lambda as $name)
    {
        echo $name . "<br />";
    }
}

$worker = new EzAssociative();
(worker)->getGreeks();
?>

When you test the code, you will see the following output:

Adonia
Marcos
Soterios
Zena

The big advantage of associative arrays will become clear when you begin retrieving data from a table in a database. The field names are associative keys, and so understanding associative arrays makes it easier to retrieve and use data stored in your database.

**Resources and Null**
Two special data types complete the PHP set of data types. A *resource* is something like a method in a class in that it can be used to perform operations. A resource variable holds a reference to an external resource that *does something*. Most often what resources do is with operations involving databases access and handlers for open databases.

A *null* data type is much clearer. Essentially, a null data means that a variable has no value. This can be due to one of three factors:

- Nothing has been assigned
- The NULL (case-sensitive) constant has been assigned
- The value has been subject to the unset() function.

You will find a surprising number of operations that use null to un-assign values (setting NULL automatically does that), act as a trigger to indicate that a process is ready—usually when the value is no longer null.
Do-It-Yourself Exercises

The following exercises help you to move ahead on your own.

- Create a class with four public methods. Each method will return a different type of scalar variable: string, float, integer, Boolean.
- Create a class with 5 properties with each assigned a string value. Create 5 scalar variables, each with a different name. Assign the 5 properties to the 5 variables and use the `return` statement to send the values to the screen.
- Create a class with two public methods with each method having a return type, `array`. Create a `numeric` array with 5 elements that is returned by one method and an `associative` array with 5 elements returned by the other method. Create a Client class that requests each Class->method and uses a `foreach` statement to output the results.
Chapter 3: PHP Operators

Operators in any language are the symbols that act as littler commands to do something. When you include an operator in an expression of code, you change a property or state. However, operators do more than just change properties, and by examining the different operators in PHP, you can see the many different functions operators perform. This chapter focuses on the most commonly used and important operators, while providing

**PHP’s Unique Operators**
If you know virtually any other computer language ranging from JavaScript to Java to Python, you know something about operators. You will find most of the same operators and operator use in PHP. However, some are different, and once you get used to using them in a PHP fashion, you may actually like them better. In the previous two chapters you saw operators used in a number of different examples; so this chapter will help fill in the blanks.

Operator precedence is at the end of the chapter so that you have a chance to review the PHP operators aligned in precedence before examining precedence order.

**Using Operators**
In their use, operators can be divided into three general categories:

- **Unary** — operates on one value
- **Binary** — operates on two values
- **Ternary** — select between two expressions that depend on a third expression.

Most operators are binary, but before looking at the full gamut of PHP operators each of these three types needs to be considered.

**Unary Operators**
When an operator affects only a single value, it is considered *unary*. For example, the logical NOT operator is an exclamation point (!). It is often used with conditional statements in the following format:

```php
if(!$someVar)
{
    //do something
}
```
The statement basically says "If it’s NOT the value of $someVar, then do something." The statement ! $someVar is essentially a Boolean because it can only be equal to the value (false) or not equal to it (true).

Other common unary operators are ++ (for incrementing a value) and -- (for decrementing a value). Whenever an operator requires only a single value to initial a change, it’s a unary operator.

**Binary Operators**

As noted, most operators you use require two values and may make a change in a third value. For example, consider the following snippet:

```php
$ace = 1;
$jack = 10;
$total = $ace + $jack;
```

The equal sign (=) is an assignment operator and requires a value (#1) and a variable (#2) to assign the value. That’s two things, and so is binary. The expression,

```
$total = $ace + $jack;
```

combines two values with the add operator (+) and assigns (=) the total to a third value.

**Ternary Operators**

This third kind of operator works more like a shortcut for a conditional expression than an operator. The ternary operator is a question mark and colon (?:), but an expression is wedged between the ? and : symbols. The following shows a typical expression using a ternary operator:

```
($a==10) ? "good" : "bad";
```

The statement is interpreted as: If $a is equal to 10 then (?) assign “good” otherwise (:) assign “bad.” The following snippet expands this example:

```php
$a=10;
$outCome = ($a==10) ? "good" : "bad";
echo $outCome;
```

Here, the variable $outCome would be assigned the value “good” because it has been assigned the test value (10). If $a were assigned 7 (or any value other than 10), for example, the assigned value would be “bad.”

For the time being don’t worry about working with ternary operators. They can be very handy for saving space in lines of code, and you’ll see some examples further on in the book. However, initially, they will not be used.
Data Types and Operators

Because PHP variables are weakly typed, almost any kind of data will work in some way with most operators. For example, consider the following snippet:

```
$first="one";
$second="two";
$third= $first + $second;
echo $third;
```

outcome=0

The plus operator (+) is arithmetic one that adds two values. In this case, the first two values are strings. However, because they are weakly typed, PHP assumes that they must be zero values that are added. As a result, instead of throwing an error, the variable $third is assigned a value of zero (0). In some computer languages the plus (+) operator is used for both addition and concatenation. A dot ($) is used for concatenation in PHP.

By the same token, if numeric values are used and their values are joined by concatenating them the result can be unexpected as the following snippet shows:

```
$first=1;
$second=2;
$third= $first . $second;
echo $third;
```

outcome=12

You need to be careful when using operators that do not throw errors when used incorrectly. About the only way you know that an operator has been used incorrectly is by recognizing unexpected output.

Scope Resolution Operator

The scope resolution operator indicated by the double-colon (::) has diverse uses. It resolves constant, static and overridden class properties methods in a class. Listing 3-1 shows the general format for the operator:

```
Listing 3-1: ScopeRes.php
<?php
ini_set("display_errors","1");
ERROR_REPORTING(E_ALL);
class ScopeRes
{
    const ALWAYS="true love<br>";
    public static $hope = "Sunrise";
}```
function __construct()
{
    echo self::ALWAYS;
}

$testClass=new ScopeRes();
echo ScopeRes::$hope;
?>

(Output)
true love
Sunrise

When working with constants and static variables, the double-colon operator is required to get the stored value. If a constant or static variable within the class in which it is defined must be accessed the keyword self is used to access the values. As you work more with classes and their properties, you will find the scope resolution operator is very handy. The discussion of constants in Chapter 4 further discusses the scope resolution operator. (In further inquiries about the scope resolution operator in PHP you may run across the term, Paamayim Nekudotayim, which is Hebrew for double-colon.)

Arithmetic Operators
Arithmetic operators are straight forward, representing the different basic math operations. Table 1 shows each symbol’s use and name:

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negation</td>
<td>–</td>
<td>–$gravity</td>
</tr>
<tr>
<td>Addition</td>
<td>+</td>
<td>$price + $tax</td>
</tr>
<tr>
<td>Subtraction</td>
<td>–</td>
<td>$price - $discount</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
<td>$mass * $density</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
<td>$total / $members</td>
</tr>
<tr>
<td>Modulus</td>
<td>%</td>
<td>$uneven % $even</td>
</tr>
</tbody>
</table>

The modulus operator divides the left entry by the right entry and returns the remainder. For example, the following snippet divides 10 by 8 and returns the remainder:

$first=10;
$second=8;
$third= $first % $second;
echo $third;
outcome=2

The other arithmetic operators work in the order of standard math operations.

Assignment Operators
With a few twists, PHP assignment operators basically assign a given value to a variable, constant or property. Table 2 provides a list of the assignment operators.

Table 2: Assignment Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic assignment</td>
<td>=</td>
<td>$total = (30 * $unitPrice)</td>
</tr>
<tr>
<td>Assign add</td>
<td>+=</td>
<td>$amount = $total</td>
</tr>
<tr>
<td>Assign concatenate</td>
<td>.=</td>
<td>$price += $shipping</td>
</tr>
<tr>
<td>Array assign</td>
<td>=&gt;</td>
<td>array('firstElem' =&gt; 'firstVal')</td>
</tr>
<tr>
<td>Assignment by reference</td>
<td>=&amp;</td>
<td>$second =&amp; $first;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$first = 20;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$second = 88;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>// $second is now 88;</td>
</tr>
</tbody>
</table>

One of the more interesting assignments is assignment by reference. Here, one value is defined by another. If the value of the assigned variable changes, so too does the variable assigned the value. Consider the following snippet:

```php
$cr="<br>";
$first=10;
$second=&$first;
echo $second . $cr;
$first=37.5;
echo $second;
```

(Output)

10
37.5

Once the variable `$second` is assigned by reference to the variable `$first`, any changes in `$first` are duplicated in `$second`.

Comparison Operators
All comparison operators resolve as TRUE or FALSE with the exception of the PHP 7 spaceship operator (`<=>`), which resolves as -1, 0 or 1. First, with two variables with different values, you can see how they resolve as TRUE or FALSE:
$jellybeans = 10;
$raisins = 20;
$raspberries = 5;
$blueberries = 10;

$jellybeans == $raisins; //FALSE
$jellybeans != $raisins; //TRUE
$jellybeans > $raisins; //FALSE
$jellybeans >= $raisins; //FALSE
$jellybeans < $raisins; //TRUE
$jellybeans <= $raisins; //TRUE

The spaceship operator (<=), introduced in PHP 7, returns three values: -1 (less than), 0 (equal to) or 1 (greater than).

$jellybeans <=> $raisins; // -1 (less than)
$jellybeans <=> $raspberries; // 1 (greater than)
$jellybeans <=> $blueberries; // 0 (equal to)

In conditional statements you will find frequent use of comparison operators. Further, you can define a Boolean value using comparison operators. For instance,

$myBoolean = ($jellybeans != $raisins);

is a perfectly valid statement. The variable $myBoolean is true only if the variables $jellybeans and $raisins are not equal. The data type can be strings, numbers or other Booleans. Table 3 shows all of the PHP comparison operators:

**Table 3: Comparison Operators**

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal</td>
<td>==</td>
<td>$first == $second</td>
</tr>
<tr>
<td>Identical</td>
<td>===</td>
<td>$twin1 === $twin2</td>
</tr>
<tr>
<td>Not equal</td>
<td>!=</td>
<td>$dogs != $cats</td>
</tr>
<tr>
<td>Not equal</td>
<td>&lt;&gt;</td>
<td>$apples &lt;&gt; $oranges</td>
</tr>
<tr>
<td>Not identical</td>
<td>!==</td>
<td>$paternalTwin1 !== $paternalTwin2</td>
</tr>
<tr>
<td>Less than</td>
<td>&lt;</td>
<td>$income &lt; $debt</td>
</tr>
<tr>
<td>Greater than</td>
<td>&gt;</td>
<td>$happiness &gt; $sorrow</td>
</tr>
<tr>
<td>Less than or</td>
<td>&lt;=</td>
<td>$orange &lt;= $grapefruit</td>
</tr>
<tr>
<td>equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Greater than or</td>
<td>=&gt;</td>
<td>$bear =&gt; $bigDog</td>
</tr>
<tr>
<td>equal to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Less than,</td>
<td>&lt;= =&gt;</td>
<td>$better &lt;= $best</td>
</tr>
<tr>
<td>equal to or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>greater than</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Incrementing/Decrementing Operators

PHP operators that add or subtract one from a value are *incrementing* and *decrementing* variables, respectively. They are most commonly found in for loops where an index or counter variable is automatically incremented with each iteration through the loop. Table 4 shows such operators available in PHP:

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment</td>
<td>++</td>
<td>$count++</td>
</tr>
<tr>
<td>Decrement</td>
<td>--</td>
<td>$down--</td>
</tr>
<tr>
<td>Pre-increment</td>
<td>++</td>
<td>++$count</td>
</tr>
<tr>
<td>Pre-decrement</td>
<td>--</td>
<td>--$down</td>
</tr>
</tbody>
</table>

The pre- and post- decrement/increment determine whether the value is added before or after the value has been returned. Consider the following snippet and its output:

```php
$cr="<br>";
$count=10;
echo $count++ . $cr;
echo $count . $cr;
echo $count-- . $cr;
echo --$count . $cr;
echo ++$count;
```

(Output)
```
10
11
9
10
```

The variable `$count` is assigned a value of 10. The first time it is printed with a post-increment, it displays 10. However, the value has been changed to 11 and so the next time it is displayed on the screen, changes to 11. The next `echo` statement uses post-decrement, and so it is still 11. The next output shows 9 because the value was changed to 10 in the post-decrement and the pre-decrement drops it one more point. Finally, using a pre-increment, the value is returned to 10 in the last `echo` statement.

The increment and decrement operators do not affect Boolean variables. For example, if you change the value of the `$count` variable above to `true`, all of the output will be 1.
**Logical Operators**

Logical operators in PHP examine two values and determine if both, one or the other, only one or neither are true. A true condition depends on the operator in use. Table 5 shows the different relations and the logical operators. In the case of the logical AND and logical OR operators, PHP has a double set.

<table>
<thead>
<tr>
<th>Name</th>
<th>Symbol</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Both are true</td>
<td>and</td>
<td>$first and $second</td>
</tr>
<tr>
<td>(AND)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Both are true</td>
<td>&amp;&amp;</td>
<td>$first &amp;&amp; $second</td>
</tr>
<tr>
<td>One or the other is true</td>
<td>or</td>
<td>$dogs or $cats</td>
</tr>
<tr>
<td>OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>One or the other is true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not true</td>
<td>!</td>
<td>$a = false;</td>
</tr>
<tr>
<td>(XOR)</td>
<td></td>
<td>$b = (! $a); //Result 1</td>
</tr>
<tr>
<td>One is true but not the other</td>
<td>xor</td>
<td>$dogs xor $cats</td>
</tr>
</tbody>
</table>

The following snippet illustrates the different logical operators and what they return. The output is rendered as true or false using the `var_dump()` function.

```php
$cr="<br>";
$yes=true;
$no = false;
$a = ($yes and $no); //False
$b = ($yes && $no); //False
$c = ($yes or $no); //True
$d = ($yes || $no); //True
$e = ($yes xor $no) ; //True
$f = (! $no); //True
var_dump($a,$b,$c,$d,$e,$f);
```

The output is from left to right in the order of the six variables beginning with $a.

    bool(false) bool(false) bool(true) bool(true) bool(true) bool(true)

Logical operators are used frequently in conditional statements.
String Operators
PHP has two operators considered string operators. We've discussed both elsewhere. The dot (.) is the concatenation operator and the dot-equal (.=) is a compound operator that concatenates and assigns values.

Other Operators
In addition to the operators discussed above in this chapter, PHP has many other operators that may need from time to time, but they are not as commonly employed. They've been given a more summary treatment.

Bitwise Operators
For operations requiring binary math, bitwise operators are essential. To fully understand how they work, either find a good text on binary math or see if you can find an online document on binary math. Table 6 summarizes the binary operators.

Table 6: Bitwise Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Bits in both are set</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Xor bits set in one but not the other</td>
</tr>
<tr>
<td>~</td>
<td>Bits set in one are not set in other</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Shift right</td>
</tr>
</tbody>
</table>

Error Control Operators
To use the error control operators, you have to set your php.ini file so that track_errors is true. This separate file can be found in the PHP root folder. Then using the at sign (@) in front of the variable you wish to record errors. For example if you have a file named “myFile” and you accidentally enter the name as “myFil” the file will be found not to exist. The following example shows the format of using this operator:

```
$fileVar = @file('myFil') or die ('Ooops! File problem','$php_errormsg');
```

Execution Operators
The execution operators are made up of back ticks (`)—not single quotes. A back tick is the key to the left of the 1 key in the top left corner of your keyboard. You can make system commands directly in your PHP program that you normally would use in a command line in your terminal. A common command, ls (Unix) or dir (Windows) lists your current directory.
Mac or Linux
$terminal = `ls`;
echo "<pre>$terminal</pre>";

Windows
$terminal = `dir`;
echo "<pre>$terminal</pre>";

On a Macintosh or Linux system (including Raspian Linux on Raspberry Pi computers), the output shows what is in the folder containing the program just executed:

- ArithmeticOps.php
- BackTick.php
- BitwiseSwap.php
- Chapter3.docx
- TestBench.php

The name of the testing program is **BackTick.php** and the other files are those in the same directory path.

The same program using the `dir` command on a Windows server delivers a similar output, albeit a different format:

Volume in drive D is SQL
Volume Serial Number is H2FU-C123

Directory of D:\Homes\Yucca\wsanders\oopPHP\SimpleClass

08/03/2014  06:09 AM

08/03/2014  06:09 AM

08/03/2014  06:09 AM  57 BackTickWin.php
04/23/2014  03:05 PM   302 Client.php
04/23/2014  03:15 PM   328 Namer.html
04/23/2014  03:04 PM   136 Namer.php
  4 File(s)     823 bytes
  2 Dir(s) 263,449,239,552 bytes free

**WARNING!** This will use terminal commands on your server or computer; so be careful!
Array Operators

Other than the union operator (+), all of the other array operators are comparison (logical) ones resulting in true or false resolutions, or in the case of the spaceship operator, -1, 0 or 1. Table 7 shows the true conditions for array logical operators.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Both arrays have the same key/value pairs</td>
</tr>
<tr>
<td>===</td>
<td>Both arrays have the same key/value pairs in the same order and same types</td>
</tr>
<tr>
<td>!=</td>
<td>Arrays are unequal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Arrays are unequal</td>
</tr>
<tr>
<td>!==</td>
<td>Arrays are not identical</td>
</tr>
<tr>
<td>&lt;=</td>
<td>-1 Array1 is less than Array2; 0 Arrays are equal; 1 Array1 is greater than Array2. (See spaceship operator.)</td>
</tr>
</tbody>
</table>

In comparing arrays, both the keys and their values are part of the logical comparison process.

The union process of arrays is not mathematical addition—that is, two arrays in a union is not the sum of the two arrays. In either a simple array where numeric keys are assigned values sequentially or in associative arrays using string keys, duplicated keys are not overwritten when the two arrays are joined. For example, consider the following snippet:

```php
$first = array("Bud","Lou");
$second =array("Larry","Mo","Curly");
$third = $first + $second;
var_dump($third);
```

When the two arrays are joined, instead of having five elements, the joined array ($third) only has three. That’s because the 0 (Bud) and 1 (Lou) elements are not overridden. The third element, 2 (Curly) in the $second array does not have to override an element in the $first array because the $first array only has two elements. So the variable dump shows the following output:

```php
array(3) { [0]=> string(3) "Bud" [1]=> string(3) "Lou" [2]=> string(5) "Curly" }
```

The dump shows that the 0 key is “Bud”, the 1 key is “Lou” and the 2 key is “Curly.” However, the values “Larry” and “Mo” are lost because they had the same keys (0 and 1) as “Bud” and “Lou” and so were not included in the union.
With an associative array, you can use unique key names to avoid values being overridden and lost. The following snippet shows the same values in an associative array with unique string keys for each element in the two arrays that are joined:

```php
$first = array("b"=>"Bud","l"=>"Lou");
$second =array("la"=>"Larry","m"=>"Mo","c"=>"Curly");
$third= $first + $second;
var_dump($third);
```

Because none of the string keys are duplicated, the two arrays joined into a third now have five elements instead of just three as the variable dump shows:

```
array(5) { 
    ["b"] => string(3) "Bud"
    ["l"] => string(3) "Lou"
    ["la"] => string(5) "Larry"
    ["m"] => string(2) "Mo"
    ["c"] => string(5) "Curly"
}
```

Depending on the purpose of the array union, you need to be aware that any identical key names whether numeric or string will not be overridden when multiple arrays are joined in a union.

**Type Operators**

You may still have occasion to check the type of your object as being a member of one class (object) or another. Using the `instanceof` statement, you can test for any class you create. Listing 3-1 provides examples using both Boolean variables and the `var_dump` statement.

**Listing 3-1: CheckInstance.php**

```php
<?php

class MyClass
{
    public $myProp="From MyClass";
}

class MyOtherClass
{
    public $myOtherProp="From MyOtherClass";
}

$a = new MyClass;
echo $a->myProp . "<br>";
// Returns 1 or blank

$myType =($a instanceof MyClass);
echo $myType . " (1=true blank=false) <p">
// Returns bool(true) or bool(false)

var_dump($a instanceof MyClass);
echo "<br>";
```
var_dump($a instanceof MyOtherClass);

```php
$myTypeNow=($a->myProp instanceof MyClass);
var_dump($myTypeNow);
?>
```

The `instanceof` statement only seems to work with user created properties. For example,

```php
$myBoo=true;
echo var_dump($myBoo instanceof Boolean);
```

or

```php
$myBoo=true;
echo var_dump($myBoo instanceof bool);
```

result in `bool(false)`.

**Precedence**

Precedence refers to the order in which statements containing operators are executed. For example, multiplication and division have a higher precedence than addition and subtraction. So in the same line, the multiplication and division occur *before* any addition and subtraction. Consider the following snippet:

```php
echo 5 + 10 / 3;
```

You might think that the 5 and 10 will be added to result in 15 and divided by 3 to render the result of 5. However, because of precedence, the 10 is first divided by 3 resulting in 3.333 and then 5 is added for the result 8.333. You can re-order precedence by using parentheses. The snippet,

```php
echo (5 + 10) / 3;
```

results in 5. Expressions in parenthesis are evaluated first no matter which operators are involved. Further, nested parentheses work their way from the innermost to the ones on the outside.

In precedence associativity refers to the direction of precedence. *Left associativity* means that expressions are evaluated from left to right, and *right associativity* means that expressions are evaluated from right to left. Some operators like `instanceof` have no associativity. Table 8 summarizes the order of precedence (from
top to bottom) and indicates whether the associativity is left, right or is non-associative (non-a).

**Table 8: Precedence**

<table>
<thead>
<tr>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-a</td>
<td>clone new</td>
</tr>
<tr>
<td>left</td>
<td>[</td>
</tr>
<tr>
<td>non-a</td>
<td>++ --</td>
</tr>
<tr>
<td>right</td>
<td>~ - (int) (float) (string) (array) (object) (bool) @</td>
</tr>
<tr>
<td>non-a</td>
<td>instanceof</td>
</tr>
<tr>
<td>right</td>
<td>!</td>
</tr>
<tr>
<td>left</td>
<td>* / %</td>
</tr>
<tr>
<td>left</td>
<td>+ - .</td>
</tr>
<tr>
<td>left</td>
<td>&lt;&lt;= &gt;&gt;=</td>
</tr>
<tr>
<td>non-a</td>
<td>&lt;= &gt;= &lt;&gt; &gt;</td>
</tr>
<tr>
<td>non-a</td>
<td>== != === !==</td>
</tr>
<tr>
<td>left</td>
<td>&amp;</td>
</tr>
<tr>
<td>left</td>
<td>^</td>
</tr>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>left</td>
<td></td>
</tr>
<tr>
<td>left</td>
<td>?:</td>
</tr>
<tr>
<td>right</td>
<td>+= -= *= /= .= %= &amp;=</td>
</tr>
<tr>
<td>left</td>
<td>and</td>
</tr>
<tr>
<td>left</td>
<td>xor</td>
</tr>
<tr>
<td>left</td>
<td>or</td>
</tr>
<tr>
<td>left</td>
<td>,</td>
</tr>
</tbody>
</table>

Working with precedence is crucial to understanding PHP programming, and it becomes clearer with practice. If you have unexpected results, take a look at precedence in an operation to make sure everything is in the order you intend. When in doubt, add parentheses to re-order the statement to make it do what you want.

**Do-It-Yourself Exercises**

The following exercises are designed to help you work out some problems with different operators and better understand precedence, associativity and the operators.

- Create a method that adds 5 numbers and returns their average.
- Create a method in a class that uses the spaceship operator (<=>) to determine 1st, 2nd and 3rd place in a computer programming contest. Use the following scores:
The output should display:

First: Evergreen Community College
Second: CalTech
Third: MIT

• The following script shows a trick you can do with bitwise operators to amaze your friends—the values of the two variables are swapped!

```php
<?php
class Trick {
    public function __construct() {
        $alpha = 65;
        $beta = 92;

        echo "\$alpha =\$alpha <br>";
        echo "\$beta = \$beta <p/>";

        $alpha ^= $beta;
        $beta ^= $alpha;
        $alpha ^= $beta;

        echo "\$alpha =\$alpha <br>";
        echo "\$beta =\$beta";
    }
}
$worker=new Trick();
?>
```

(Output)
$alpha =65
$beta = 92
$alpha =92
$beta =65

**Notice** that in order to print a variable name instead of the variable’s value a backslash is placed in front of the variable name.

• Create two arrays each with four elements (key/value pairs). Use numeric keys for the first one and string keys for a second. Then join them using the
union (+) operator into a third string so that the third variable has 8 elements.
Chapter 4: The Atoms that Make Objects

In this book, PHP is used for communication between different objects created in PHP. The essence of an object-oriented approach to computer coding is to create objects and then ask those objects to do things. In the same way that you get into your car and give it a set of instructions through the steering wheel to take you where you want to go. You know that your steering mechanism is made up of links, gears, servos and other gizmos about which you understand little. You just want the car to go left when you communicate \textit{left} by steering left and \textit{right} when you steer right.

In object oriented programming (OOP), we focus on the \textit{object} and communication with it and other objects. However, we need to program each object with a series of sequential statements to create the objects and their methods and properties. They make up the atomic matter of PHP OOP.

\textbf{Statements}

In the first three chapters you’ve seen plenty of statements and references to the term \textit{statement}, but we really never got around to defining what a statement does. Well, in programming a statement is a collection of code that \textit{does something}. It’s like a basic sentence in language. (Do this. Do that. \textit{Do something}.) Just like in grammar, PHP has rules, but nowhere near the number you find in grammar.

\textbf{Definition, Declarations and Assignments}

Most definitions in PHP are simply a matter of assigning a value to a variable. The statements generally include an assignment operator and a variable. You define a variable by assigning it a value. For example, the following statement defines a variable by assigning it a string literal:

\texttt{$\textit{doggy}="\textit{Rover}\";$}

All statements, no matter how complex or simple, \textit{end with a semi-colon (;)}.

You can declare a variable without assigning it anything. Typically, you find a declaration without an assigned value in a creating a property in a class. For example, in Listing 4-1 you can see several statements, including the first one that just declares a property (variable) for the class:

\textit{Listing 4-1: \textit{TestClass.php}}

\texttt{<?php}


definition, declarations and assignments

most definitions in php are simply a matter of assigning a value to a variable. the statements generally include an assignment operator and a variable. you define a variable by assigning it a value. for example, the following statement defines a variable by assigning it a string literal:

\texttt{$\textit{doggy}="\textit{Rover}\";$}

all statements, no matter how complex or simple, \textit{end with a semi-colon (;)}.
A good class only does one thing, and while that one thing may be relatively complex, it doesn’t have to be. Besides the statements that make up the class are usually pretty simple as you saw in Listing 4-1.

**Case Sensitivity and Insensitivity**

Some computer languages are case sensitive; others are case insensitive. *PHP is both!* The term, *case sensitivity* refers to letters being uppercase (A, B, C, D, etc.) or lowercase (a,b,c,d, etc.) If it doesn’t matter what case the label on the code is, it is *case insensitive*, while if the case of letter differentiates on term from another, it is *case sensitive*.

Classes defined by the developer (that’s you!) are insensitive as are built-in constructs and keywords you’ll be using. We’ve been using the keyword **class** to define our classes. However, we could define our classes and keywords as shown in Listing 4-2:

**Listing 4-2: TestClass**

```php
<?php
class TestClass
{
    //Declaration statement
    private $myProperty;

    function __construct()
    {
        //Assignment statement
        $this->myProperty="I am a property!";

        //Action statement
        echo $this->myProperty;
    }
}
//Declare+assignment statement
$testClass=new TestClass();
?>
```
The class in Listing 4-2 works exactly like the one in Listing 4-1. Note that the following are all *keywords* that were written without regard for case:

- class
- private
- function
- __construct()
- $this
- new

Because the *rule* allows case insensitivity does not mean it’s a good idea. As a general convention, write all class declarations beginning with a capital letter and all keywords using lowercase. That helps keep everything clear, and when you’re dealing with several objects, clarity is important.

While keywords are case insensitive, variables are case sensitive. For example,

```php
$myVar
$Myvar
$MyVar
$MYVAR
```

are all *different* variables. For example, the following statements would result in a blank space or generate an “Undefined variable” notice.

```php
$doggy="Rover";
echo $doGGy;
```

That’s because *$doggy* and *$doGGy* are as different as *$doggy* and *$kitty*. So, the *rule for variables is that they are case sensitive*. However, differentiating variables on the basis of their casing does not promote clarity. Your code will be far clearer if you use names like *$doggy1* and *$doggy2* instead of *$doggy* and *$doGGy* to differentiate your variables. (Imagine how much time you’d waste trying to debug a program because you *accidentally* used the wrong case in a variable name? Imagine how much more time you’d waste if you did it intentionally.)

**Expressions**

*Expressions* in statements are evaluated to produce some value. For example, the expression,

\[ 2 + 2 \]
can be evaluated to produce 4. The expression can be used in different statements.

```php
echo 2 + 2;
$sum = 2 + 2;
$formula = (2 + 2) / $algorithm;
$complex = ((7 ^ 3) * (2 + 2)) + $formula;
return 2 + 2;
```

As you can see, the expression `2 + 2` can be used in a variety of ways, including as part of other expressions. Expressions are created with literal values, constants, objects, properties, methods, operators and variables. (Chapter 3 has a complete list of PHP operators to use to create expressions and the order to set up operators in expressions to produce the effect you intend.)

**Literals, Constants and Variables**

In programming, values are either stored in variables (including properties) or constants. Those values can be generated dynamically through expressions, sent from external sources (such as database or file) or they can be generated as literals—actual numbers, strings or Booleans. This section examines the syntax of all of these in PHP.

**Literals**

Any actual data value placed in a program is a literal. Generally, literals are divided into four types:

- String literals
- Numeric literals
- Booleans
- Null

Each is slightly different and will be examined individually.

**String Literals**

A string literal is any alphanumeric grouping of characters. String literals must be placed in single or double quotes. The following are examples:

```
'Hello world'
"Goodbye Columbus"
'123 Elm Street'
"805 555 4343"
"Why you little !#$&&!"
```

As long as the characters are within quotation marks, the characters are considered part of the string literal. However, PHP has some important exceptions.
One unique characteristic of PHP is that variables can be placed inside quotation marks and still be treated as variables and not part of the string literal. For example, consider the following code snippet:

```
$myNum = 66;
echo "Jane's lucky number is $myNum ";
echo "that she stores in the variable \\$myNum.";
```

The outcome of that snippet is:

```
Jane's lucky number is 66 that she stores in the variable $myNum.
```

Because all variables begin with a dollar sign ($), PHP recognizes variables inside of string literals and treats them as variables. To treat variable names within strings literals as **literals**, place the backslash (\) escape character in front of the variable name, and then the characters are treated as literals.

You can place double or single quotes inside of string literals as long as the outside quote marks are the opposite of the inside quotes. For example, the following line of code,

```
echo 'John said, "Wow!" to that.';
```

results in the output:

```
John said, "Wow!" to that.
```

However, the following would create an error:

```
echo 'John said, "Wow!" to Jane's number.';
```

That’s because PHP cannot differentiate between an apostrophe (‘) and a single quote (’). In order to use apostrophes (or other single quotes) within a quotation enclosed by single quotes, simply use the escape character (\) before the second single quote. The following line has the desired output:

```
echo 'John said, "Wow!" to Jane\'s number.';
```

Your screen shows,

```
John said, "Wow!" to Jane’s number.
```

You can also use the backslash escape character with double quotes.
**Numeric Literals**

Numeric literals can be any valid PHP number format expressed without quotation marks. The following are typical numeric literals:

5  
23.18  
.0342  
0xA3CC4B

The last value in the list is a hexadecimal value. Any value preceded by “0x” (zero-x) is interpreted as hexadecimal. However, the output is in decimal. For instance, the following statement results in 255:

```
$myHex = 0xff;
echo $myHex;
255 //output
```

Any value expressed in hexadecimal is *not* case sensitive. The following are all the same:

```
0xFF2BCC;  
0xff2bcc;  
0XFF2BCC;  
```

Keep in mind that hexadecimals are all **numeric** values; *not* strings.

**Booleans**

A Boolean assignment occurs *only* when a value of **true** or **false** is assigned. The output may be 1, but it must be assigned **true** and not 1 to be a Boolean variable. With Booleans that resolve to false, the list of conditions to be considered Boolean false is a bit longer. First, though, to test a data type for a Boolean, you can use the `is_bool()` function as the following snippet shows:

```
$cr=\"<br>\";
$myBool=false;
echo is_bool($myBool) . $cr;
$myBool=true;
echo is_bool($myBool) . $cr;
```

Both of those `echo` statements will resolve `true` and 1 is printed to the screen. However, in a conditional statement far more `false` conditions are recognized as Boolean false. In the following snippet, all of the outcomes are resolved as false:

```
$myNothing;
//$myBool=0;
//$myBool="";
```
If you uncomment (remove the double slashes //) from the different $myBool variables so that all but one assignment has no comment slashes in front of it, you will find that each resolves to false and False! is printed to the screen. However, if you use the is_bool() function, these other Boolean conditions are unrecognized as Boolean false.

**NULL**
The NULL value can be assigned to a variable if you want it to be empty and not throw an error that it is unassigned. For example, the following two variables are both empty:

$myNada;
$myNothing=NULL;
echo $myNothing;
echo $myNada;

The second echo statement will generate an error. Both will generate a Boolean false in a conditional statement. When working with databases, developers often assign unassigned variables a NULL value so that the NULL is stored in the database table rather than nothing (blank.)

**Constants**
Constants store unchanging values. They can store scalar values only. *Within a class definition* the keyword const indicates a term will be a constant. Declare constants outside of a class definition using the define() function rather than assignment operators as with variables. By convention, constants are in all caps, making them easy to spot in a program. The next two subsections examine using constants within a class or interface and outside of classes and interfaces.

**Constants in Classes and Interfaces**
One of the surprising features of PHP is that constants can be used in interfaces as well as classes. Further, they work exactly alike, and an actual value can be passed from an implemented interface to the class that implements it. The scope resolution operator (::) must be used to extract the constant’s value from an interface or class. For example, Listing 4-3 shows a constant in an interface that is passed to the class that implements the interface:

*Listing 4-3: InterfaceConstant.php*

```php
<?php
interface ITest
```
By allowing constants to be used by implementing an interface, developers can place all of the connection information in the interface and use it by all of the classes that implement the interface. Typically, interfaces are all abstract and such a practice may be seen as defeating the purpose of an interface. However, it can be very handy for storing immutable values that will be used by all implementations of the interface.

**Constants Outside of Classes and Interfaces**

Dealing with constants outside of a class or interface is more like standard variables except that constants do not change. (Think of constants of a process that simply renames the literal values to the constant name.) In using the define() function, the constant identifier must be in single or double quotes, while the value is in quotes only if it is a string literal. The following examples illustrate defining different constants outside of a class:

```php
define("MY_DATABSE", "billzDB"); // String
define("FREEZE_F", 32); //Integer
define("FREEZE_C", 0); //Integer
define("POLITICIAN", false); //Boolean
define("TAX", .06); // Float
define('HEALTHY', 98.6); //Float
```

Unlike variables, constants are not recognized as `constants` within a string literal. If constants are used within a string literal, they must either be concatenated or put into a variable. The following snippet illustrates how constants are employed with string literals:

```php
define('HEALTHY', 98.6);
echo "Jack's temperature is HEALTHY.";"
echo "Jack's temperature is " . HEALTHY . ".<br>";
$healthy = HEALTHY;
echo "Jack's temperature is $healthy.";

(Output)
Jack's temperature is HEALTHY.
Jack's temperature is 98.6.
Jack's temperature is 98.6.

The value of constants lies in their capacity to be immutable.

Variables

PHP variables are unique in that they are indicated by a dollar sign ($) prefix. Variables are weakly typed in PHP, and they can hold any data type and the data type can change whenever a new value of a different type is assigned to the variable.

Variables can be placed in the middle of a string literal and still be read as variables. In PHP variables are amenable to just about anything the developer wants to do with them and with a few exceptions, data types are mutable. For example, the following “Vars” class starts with a variable ($myVar) and assigns it floating-point value, then changes it to an expression that contains a Boolean added to an integer concatenated to a string. Finally, it changes it again to an associative array.

```php
<?php
class Vars
{
    public function __construct()
    {
        $myVar = 22.34;
        $myVar =true + 17 . " Whatzup?<br>";
        echo $myVar;
        $myVar = array("hi" => "Greetings!");
        echo $myVar['hi'];
    }
}
$worker = new Vars();
?>
```

(Output)
18 Whatzup?
Greetings!

In PHP, variables are mutable in every sense of the word.

In Chapter 3, one of the operators assigned values by reference to another variable. If the assigned variable changes, then the variable it is assigned to changes to the
new value as well. The $ operator is used for variable assignment. The following class provides an example:

```php
class Amper
{
    public function __construct()
    {
        $myVar = "Eat healthy.<br>");
        $myNewVar =& $myVar;
        echo $myNewVar;
        $myVar = "Don't forget the veggies!";
        echo $myNewVar;
    }
}
$worker = new Amper();

(Output)
Eat healthy.
Don't forget the veggies!
```

**Identifiers**

Labeling variables, constants, classes, interfaces, and functions is simply a matter of providing them with a name. Basically, any alphabet character in upper- or lowercase or the underscore (_) character will do the trick. All variable names must begin with a dollar sign ($). Numbers can be used in an identifier (label), but not as the first character. For example, the following are all legitimate identifiers:

```php
$myNumber1Var=5;
$_privateVar =@ $myNumber1Var;
define('MYCONSTANT', 22); //constant outside a class
cost MYCONSTANT = 22; //constant in a class
class MySweetClass..
interface IMySweetInterface...
function doItNow() ...
```

Because case sensitivity in PHP is variable depending on whether you're using a keyword or a user constructed term, you need to follow some set of conventions. When working with OOP you have to be especially careful because you are using objects and other classes you've created.


**Scope**

In computing languages, *scope* is the part of the program where your variable or function can be accessed and used. A closely related concept in PHP is called *visibility*, which refers to usage of a property or method within and outside of a class where the property or method is declared. Scope and visibility are related but need to be separated to show their differences.

**Program Scope**

In the context of discussing *program scope* the term “program” refers to the code in a single file that does not use objects outside of the current file. Any variable declared in a program that has no functions can be used anywhere in the file after the declaration. In general, program scope can be broken down into four parts, *global, local, parameters* and *static variables*.

**Global variables**

All variables created outside of functions are considered *global*. The variables may only be used *after* they have been assigned a value. Listing 4-4 shows an example of where and how global variables can be employed:

**Listing 4-4: Scope.php**

```php
<?php

class Scope
{
    public function __construct()
    {
        $myVar1=5;
        $myVar2= " duck";
        $plural= "s";
        $myVar3= "visited my house.";
        echo "I said, $myVar1 $myVar2 $plural $myVar3 
<br">
        //Variable $myVar4 has not been declared or assigned value
        echo $myVar4;
        $this->myScope();
        $myVar4="<br>There's more.</br">
        echo $myVar4;
    }

    public function myScope()
    {
        echo "He hit the ball and said $myVar2! but nobody listened.";
    }
}

$worker = new Scope();
```
I said, 5 ducks visited my house.
He hit the ball and said! but nobody listened.
There's more.

In Listing 4-4, you can see that the declared variables are all available in the first line of output. However, in the second line of output, the variable $myVar2 does not show up because it is inside of a function and so the word duck does not appear in the second line. Likewise, the value of $myVar4 does not appear when it is sent to the screen with an echo statement. That's because no value has been assigned to the variable yet. In the last echo statement at the end of the program, $myVar4 has been assigned a value, and so now there's something to see on the screen.

The use of global variables is discouraged in functions. In object oriented programming the purpose of methods is to encapsulate operations, and so putting global variables in a method (created as a function) defeats the purpose of encapsulation. Likewise, many of the global settings in PHP have been changed or deprecated to discourage global variable use. However, if you have occasion where a global variable is needed in a function when you're not using a class, the following program shows you how to use both the keyword globals and the $GLOBALS array to general such global variables from within a function:

```php
<?php
$functionVar="Good";
$everywhere=" everywhere!<br>";

$moreGlobal= "What? More?";
$stuff =" Junk I don't need.";

function globalVars()
{
    global $functionVar, $everywhere;
    $functionVar .= $everywhere;
    echo $functionVar;

    $GLOBALS['moreGlobal'] .= $GLOBALS['stuff'];
}

globalVars();
echo $functionVar;
echo $moreGlobal;
?>
```

(Output)
Good everywhere!
As noted, you're unlikely to use any kind of global variable within a function except those properties set as public and then used within a method in a class.

**Superglobals**
In the PHP manual in the path *PHP Manual > Language Reference > Predefined Variables* you will find the *Superglobals* section. In it is a list of nine superglobals, but this subsection deals with only two—superglobals typically used in HTML to pass form data, *post* and *get*. To further simplify this introduction to superglobals, only *post* is examined here. (Chapter 8 has a more detailed discussion of superglobals.)

To send data from HTML to PHP the `<form>` tag expects a *method* parameter where the developer indicates “post” (or “get”). The following tag shows a typical form tag:

```html
<form action="Client.php" method="post">
</form>
```

Whatever data the user enters or selects is then sent to the PHP file using the post method. From PHP, the superglobal, `$_POST['inputval']` is an associative array that holds the element values of each input in the form. For example, consider the following HTML snippet:

```html
<form action="Client.php" method="post">
    <input type="text" name="fname">
    <input type="radio" name="gender" value="male">
    <input type="radio" name="gender" value="female">
</form>
```

The PHP program in the file named Client.php now has access to the text input named “fname” and the radio button group named “gender.” The superglobal `$_POST['fname']` holds the value of the text input and `$_POST['gender']`, the value of the radio button selected. The following snippet shows how the HTML entered values are passed to PHP variables:

```php
$firstName = $_POST['fname'];
$gender = $_POST['gender'];
```

A text input type passes the value the user types in, and in the radio button, the user makes a mutually exclusive choice. The choice is stored in the *value* attribute as part of the `<input type="radio" >` tag. So, for example, the value of `fname` might be “Michelle” and the gender value, “female.”
What’s Super about Superglobals?
The scope of a superglobal transcends a file or class in PHP. For example, if you have your HTML document action attribute assigned to Client.php, you can access the superglobals from the Client class (assuming your class and file name are the same). However, even if you do not access a superglobal (e.g., $_POST['gender'] ) from the Client class you can still access them from classes instantiated through Client. Figure 4-1 illustrates the truly extraordinary global scope of these variables:

![Superglobal Scope Diagram](image)

**Figure 4-1: Scope of superglobals transcends classes**

Having such a scope makes the HTML-as-I/O and PHP-as-processor pairing quite useful.

Unsetting a Superglobal
While the good feature of superglobals is that they can be accessed from different classes within a program, you need to remember that they can run around like free-range chickens and wreck havoc on your program if not carefully attended.

One way to deal with superglobals is to place their values into an encapsulated property and then clear them. For example, consider the Listings 4-5 and 4-6: HTML and PHP documents calling a private static function:

**Listing 4-5: SuperGlobals.html**
<!DOCTYPE html>
<html>
<head>
    <title>Clean My Superglobals!</title>
    <style>
        body { font-family: arial; color: #434055; background-color:#FDF8E5; }
        h2 {color:#7CA4A4;background-color:#434055; width:350px; text-indent: 1em;}
        iframe {background-color:#B29C87;}
    </style>
</head>
<body>
<h2>Enter Superglobal Data</h2>
<form action="ClientCleaner.php" method="post" target="feedback">
    First Name:<br>
    <input type="text" name="fname">
    <p>
        Gender:<br />
        <input type="radio" name="gender" value="male"><nbsp;Male<br />
        <input type="radio" name="gender" value="female"><nbsp;Female
    </p>
    <input type="submit" value="Send Data">
</form>
<iframe name="feedback" width=250 height=150></iframe>
</body>
</html>

The HTML file passes the superglobals to the PHP program shown in Listing 4-6. It first places the superglobals into variables and displays them, and then it clears the superglobals and shows that if the superglobals are not set, they are passed an “unset” string.

**Listing 4-6: ClientCleaner.php**

```php
<?php
class ClientCleaner
{
    //client request
    public static function request()
    {
        echo self::getSuperglobals();
        echo self::cleanSuperglobals();
    }
```
private static function getSuperglobals() : string
{
    $fname=$_POST['fname'];
    $gender=$_POST['gender'];
    $package = $fname . " is a $gender.";
    return $package;
}

private static function cleanSuperglobals() :string
{
    unset($_POST['fname']);
    unset($_POST['gender']);
    // *Uses ternary statements
    isset($_POST['fname']) ? $fname=$_POST['fname'] : $fname="<em>unset</em>";
    isset($_POST['gender']) ? $fname=$_POST['fname'] : $gender="<em>unset</em>``;
    return "<p>Test: $fname is the first name <br />
    $gender is the gender";
}

ClientCleaner::request();
?>

*Note: Ternary statements are a compact version of if/then/else statements. See Chapter 8. They have the form:

    condition ? doTrue : doFalse;
    isset(T/F) ? useNameA : useNameB;

Next, the program calls the **cleanSuperglobals**() which erases them using the **unset()** function. Using local variables, values are assigned, and then the program attempts to re-assign superglobals. However, it finds they have been cleared—they're NULL. A conditional statement checks to see if they're set, and since they are not, the values remain the same. Only a first name and gender value are passed from the HTML document. Figure 4-2 shows what appears:
Chapter 9 goes into more details about using superglobals, and you will see how you can use superglobal values to instantiate classes and methods. They can save a lot of time and avoid unnecessary conditional statements when making a request. Further, in Chapter 13, you will learn about how injection attacks can be made using superglobals and how to defend against them.

**Local Variables**
Within a function, all variables created in that function are treated as local to that function. Attempting to use variables created in a function will result in an undefined variable error. Listing 4-7 provides a class with examples of both the
correct use of local variables in functions and attempt to use the same variable outside of the function.

**Listing 4-7: Locals.php**

```php
<?php
error_reporting(E_ALL);
ini_set("display_errors", 1);
class Locals
{
    public function __construct()
    {
        echo $this->localVars();
        echo $inside;
    }

    private function localVars(): string
    {
        $inside = "This is from the inside of a function.<br>";
        return $inside;
    }
}
$worker = new Locals();
?>
```

(Output)
This is from the inside of a function.
Notice: Undefined variable: inside in
/Users/wdsanders/Sites/php/MWD310/MWD310_s16/Programs/Locals.php on line 9

In PHP visibility refers to the accessibility of a class (as an object), a method or a property of a different class. In Chapter 1 (in the section Designating Class Properties and Visibility), you saw that PHP has three levels of visibility: **private**, **protected** and **public**. In some respects the concept of scope and visibility are similar in that both deal with access to a variable, function (or in terms of a class, a property or method.)

Rather than rehashing what you read in Chapter 1, Listing 4-8 provides an example of all of the levels of visibility and you can see how each can be used.

**Listing 4-8: VisibilityDemo.php**

```php
<?php

class VisibilityDemo
```
private $classProperty="This belongs to VisibilityDemo<br>";

function __construct()
{
    //May only be used in same class
    echo $this->classProperty;
    $this->showStuff();
}

//showStuff() may be used in same class or child class
protected function showStuff()
{
    echo "This is from a protected function<br>";
}

//easyMe() can be called from any instance of VisibilityDemo
public function easyMe()
{
    echo "I can be picked up by any instance of VisibilityDemo<br>";
}
}

class VisChild extends VisibilityDemo
{
    function __construct()
    {
        //showStuff() inherited
        $this->showStuff();
    }
}

$vizTest=new VisibilityDemo();
$vizTest->easyMe();
$childTest= new VisChild();
?>

(Output)
This belongs to VisibilityDemo
This is from a protected function
I can be picked up by any instance of VisibilityDemo
This is from a protected function
Notice that both the private property and the private method were launched from within the VisibilityDemo class. The class itself is public, and so is the constructor function, and so the instance of the class ($visTest) can directly access only the public method—easyMe(). The only reason that the private property and protected method could be used was because it was from within the constructor function. The instance of the child class ($childTest) can use the protected method because it inherited it from the VisibilityDemo class directly.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand the basic building blocks in OOP PHP code. (Do not use `var_dump` in these exercises.)

- Create a small program that uses a single variable name with the identifier at least 6 letters long. Make each variable unique by using different casing (e.g., $thisVar, $ThisVar, etc.). Make a variable for 5 different data types. This exercise is to show you what you should not do—you’ll see how difficult it is to keep the unique variables separate using this technique!
- Write an expression that will do the following in the following order:
  - Add your age to today's date (just the month day number 0-31)
  - Multiply the sum by 3
  - Divide the quotient by 4
  - Add the year divided by 3 modulo
- Create a class that has three methods and three properties. The three methods and properties will be one each private, protected and public. Add a child class that uses the protected property and method. Display the contents of all properties and the method to the screen.
- Create a class that uses two of four superglobals passed from an HTML5 document, and the first class calls a second class that uses the remaining two superglobals. All output will appear in an iframe window.
Chapter 5: Basic Language Structures

Every computer language has a set of control structures, and PHP is no different. Working with PHP in an OOP or non-OOP context makes no difference—the language structures are the same. What makes OOP different is how the developer uses the structures. So in learning PHP language structures, all of the examples will be shown in an OOP context.

The three basic structures in writing code are sequential statements, conditional statements and loops. That’s it. These three structures are used within interfaces, classes and functions (which become methods in classes). This chapter shows how to use these basic structures to create OOP programs in PHP.

Sequential Statements
Sequential statements are a series of statements are executed from the top to the bottom. Sequential statements exist within classes and methods, and they are used for most of the inner programming statement in all programming all objects. A sequence is simply two or more statements in a program that may be executed as part of a general program, another programming structure (a loop or conditional statement) and in any kind of OOP element. Listing 5-1 shows sequential statements used in creating properties and within methods:

Listing 5-1: Sequences.php

```php
<?php
class Sequences {
    //Two line sequence
    private $prop1="Cats";
    private $prop2="Dogs";

    function __construct() {
        //Two line sequence
        echo "Here it comes!<br>";
        echo $this->shoutOut();
    }

    private function shoutOut(): string {
        //Three line sequence
        $bigCat = "Manx " . $this->prop1;
        $bigDog = "Greater Swiss Mountain " . $this->prop2;
        return "Yikes! Look at those $bigCat and $bigDog";
    }

    //One line sequence
}
```
$fireOffSequences = new Sequences();
?>

(Output)
  Here it comes!
  Yikes! Look at those Manx Cats and Greater Swiss Mountain Dogs

If we break any class or operation down into sequential statements, OOP is really pretty easy. Also, the sequences are shorter and more compact than non-OOP sequences. So while the overall structure of an OOP program is clearly not a sequential one, all of the parts of a PHP OOP program are created using little sequences of code.

**Conditional statements**

A statement that includes a branch of some sort is considered a *conditional* one. The program branches one way in one condition and a different way in a different condition. Typically, the condition is expressed as a Boolean, and the program branches depending whether the Boolean true or false. The range of behavior can be from *do nothing* to load up another class and use all of its methods.

**The If Statement**

The if statement evaluates a condition and decides what to do next. For example, in a login object, an object may contain a method to determine whether the password is correct or not. Listing 5-2 shows a class with a login method that evaluates a password.

Listing 5-2: Login.php

```php
<?php
class Login
{
    function __construct()
    {
        $test="secret";
        echo $this->password($test);
    }

    private function password($pass) : string
    {
        if($pass == "secret")
        {
            return "You may enter the goldmine.";
        }
    }
}
$testClass=new Login();
?>
```
(Output)
You may enter the goldmine.

In Listing 5-2, the segment:

```php
if($pass == "secret")
```

sets up a Boolean outcome. The `$pass` variable either contains the value “secret” or it doesn’t; so, it’s either `true` (value is “secret”) or `false` (the value is not “secret”).

Suppose that the login requires both a user name and a password. (You’ve only done that about a thousand times!) The double ampersand (`&&`) operator is a logical AND that is true only if both conditions are true. For example, the segment,

```php
if($uname == "lumpy" && $pass == "secret")
```

evaluates to `true` only if both conditions are true. If only part of the expression one is false, the whole expression evaluates to false.

Now consider a situation where login requires only the username or the password to be true. In PHP, double pipes (`||`) represent the logical OR operator. Using the logical OR operator requires only one part of an expression to be true. For instance, the same segment with a logical OR operator, means that if the user name or the password is true, the expression is evaluated as true:

```php
if($uname == "lumpy" || $pass == "sugar")
```

So if the user enters `lumpy` for the user name and `sugar` for the password, the expression will be considered `true` because only one condition has to be true. You can enter several logical AND or logical OR operators in a conditional statement for more complex conditions.

**The “invisible” comparisons**
In some PHP conditional statements you will see a line like the following:

```php
if($check)
{
    echo "That is correct!";
}
```

The “invisible” comparison is a query to determine if the variable is true. It’s the same as writing:

```php
if($check == true)...
```
Likewise, you may see something like,

\[
\text{If}(! \ $\text{check}) \ldots
\]

Which is the same as,

\[
\text{If}(\text{check} == \text{false}) \ldots
\]

It’s a handy shortcut and you will run into it frequently.

**The else clause**

An if statement can contain an else clause. The else clause provides an alternative route for the program where the if condition evaluates to false. For example, the following method uses an else clause to let the user know he had the wrong password:

```php
<?php
class Login
{
   function __construct()
   {
      $test="secret";
      echo $this->password($test);
   }

   private function password($pass) : string
   {
      if($pass == "secret")
      {
         return "You may enter the goldmine.";
      }
      else
      {
         return "Please leave. Now.";
      }
   }
}
$testClass=new Login();
?>
```

If you’re making a Web site where the user must choose between a mutually exclusive choice, such as *man* or *woman*, the paths can open new objects or classes. In fact, the conditional outcomes can be *any* legitimate statement.
The else if and elseif clauses

If three or more outcomes are possible, PHP provides the `else if` and `elseif` clauses. Both of these clauses generate similar outcomes but are different in how they are written. Listing 5-3 illustrates a common use of `else if`.

Listing 5-3: Diplomat.php

```php
<?php

class Diplomat
{
    function __construct()
    {
        $india = "Hindi";
        $bulgaria = "Bulgarian";
        $canada = "English";
        echo $this->interpreter($bulgaria);
    }

    private function interpreter(string $region) : string
    {
        if($region == "Hindi")
        {
            return "Suprabhaat!";
        }
        elseif($region == "Bulgarian")
        {
            return "Dobro utro!";
        }
        else
        {
            return "Good morning!";
        }
    }
}

$worker = new Diplomat();
?>
```

(Output)

Dobro utro!

Note that after the `else if` clause is another `else` clause. It works like a residual condition—nothing else is left so it must be this. In Listing 5-3, the residual condition is the morning greeting in English.

PHP has another format called `if else`—no spaces between “if” and “else.” This second format processes a bit faster, and while the same as `if else` where curly braces are used is a bit different if colons are employed. The `elseif` statement works
the same where the original if statement results in false. For instance, the line from Listing 5-3,

    else if($region=="Bulgarian")

can be written as,

    elseif($region=="Bulgarian")

with identical results.

When elseif is used with the colon format, the conditional sequence must end with an endif statement as the following class, Conditions, shows:

```php
<?php
class Conditions {
  function __construct() {
    $first=10;
    $second=11;
    $third = $first + $second;
    if($third==22):
      echo "The sum is 22";
    elseif($third==21):
      echo "The sum is $third";
    else:
      echo "Something else";
    endif;
  }
}$testClass=new Conditions();
?>
```

(Output)

The sum is 21.

Either the colon or curly brace format works, and while the elseif form executes a bit faster, when several conditions are considered in a conditional statement, a better statement may be switch which is considered next.

**The switch Statement**

When multiple conditions are evaluated, a tidier way to set up the conditional statements in PHP is using the switch statement. The switch statement has a single parameter value and several cases evaluate that value to be equal with their own. For example, Listing 5-4 shows a simple string variable being evaluated:
Listing 5-4: PetSound.php

```php
<?php
class PetSound
{
    public function __construct()
    {
        echo $this->speak("Mouse");
    }

    private function speak(string $critter) : string
    {
        switch($critter)
        {
        case "Dog":
            return "Bark bark!";
            break;

        case "Cat":
            return "Meow purrrr";
            break;

        case "Mouse":
            return "Squeek eeeek!";
            break;

        case "Turtle":
            return "Gurgle gurgle ";
            break;

        default:
            return "Couldn't find your pet";
        }
    }
}
$worker = new PetSound();
?>

(Output)
    Gurgle gurgle....
```

The **switch** statement queries the state of an object and then looks for a match with that state in several different cases. If a match occurs, the statements in the case are executed. *If* a **break** statement is added after the case actions, the program ceases going down the list of cases. When the developer does not include a **break** statement, the program will continue to the rest of the cases. If no match is found, the developer can choose to let the **switch** to end when it runs out of cases or it runs into the **default** case. *(Note: Since a **return** statement is after each case, we really
did not need the `break` statements because as soon as a program encounters a `return`, it terminates the method making the `return`—otherwise, `break` statements are essential.)

To provide a bit more practical example, Listing 5-5 uses a switch statement to decide which table query to make to execute in a database table. For this example, it just shows how to call a private method in a class using a switch statement that illustrates the process.

Listing 5-5: TableWork.php

```php
<?php

class TableWork
{
    function setTheTable(string $job) : string
    {
        switch ($job)
        {
            case "dataEntry":
                return $this->doEntry();
            case "retrieveData":
                return $this->doRetrieval();
            case "searchData":
                return $this->doSearch();
            case "changeData":
                return $this->doChange();
            default:
                return "Unable to process.";
        }
    }

    private function doEntry() : string
    {
        return "Entering data....";
    }

    private function doRetrieval()
    {
        return "Getting data....";
    }

    private function doSearch()
    {
        return "Searching data....";
    }

    private function doChange()
    {
        return "Changing data....";
    }
}

$host = new TableWork();
```
echo $host->setTheTable("searchData");
?>

(Output)
Searching data....

The search string for the switch statement is introduced through the parameter in the setTheTable method, which expects a string argument. The little snippet that called the TableWork class simply added the parameter for the SQL statement it wanted to invoke. Later in the book when you learn about using MySql tables, you will see the practicality of this particular class and ones like it.

**Loops**

PHP has three loop structures and each is important for different kinds of operations. However, while the choice of one type of loop over another can be a matter of taste and style, program conditions often require selecting one over another.

Unlike other programming languages you may know, loops in PHP are especially important for iterating through records in a database table. A single record in a database may have several different fields a developer needs to examine, and the process for looking at a single record is the same for all records. Rather than writing a new procedure for each record, the same procedure is used many times in a loop. The three loop structures are:

- while
- for
- foreach

Loops can be used for non-database operations as well—iterating through an array is a typical use. However, as soon as you begin learning about searching for a single record in a database containing thousands of records, you'll really appreciate loops in a new way.

**The while and do...while Loops**

Like all loops, the while loop works with a Boolean value. The structure of the loop is quite simple. The while term has a condition that evaluates to true or false. As long as it's true, the loop keeps on going.

```
while (condition)
{
    //do something
}
```
For example the following class keeps repeating *until the while condition is no longer true*:

```
<?php
class WhileTrack
{
    function __construct()
    {
        $tracker = 20;
        while($tracker >= 0)
        {
            echo $tracker . "<br>";
            $tracker--;
        }
    }
}
$testClass=new WhileTrack();
?>
```

In iterating through a database table, the condition the loop examines is the end of the file (the table no longer has any records) or a match has been found.

The **do...while** loop is similar to the **while** loop except it is guaranteed to make at least one iteration. The loop condition is at *the end of the loop*. The general structure is,

```
do
{
    //do something
}
while(condition);
```

In general, use the **while** loop, but if you must have at least one iteration, use the **do...while** loop. To get a full comparison, Listing 5-6 shows both types of loops in a PHP class example:

**Listing 5-6: CompareWhile.php**

```
<?php
class CompareWhile
{
    function __construct()
    {
        $this-&gt;whileDemo();
    }

    private function whileDemo()
    {
        $counter=5;
    }

    function doWhile()
    {
        do
        {
            //do something
        }
        while(condition);
    }
}
```
```php
while($counter)
{
    echo "Number: " . $counter . "<br>";
    $counter--;
}
//The do...while loop always goes 
//through the loop at least once.
```

```php
do
{
    echo "Number: " . $counter . "<br>";
    $counter--;
}
while($counter>=0);
```

$testClass=new CompareWhile();

(Output)
while Loop
Number: 5
Number: 4
Number: 3
Number: 2
Number: 1
do...while Loop Number: 0

Notice that in the **while** loop the statement,

```php
while($counter)
```

is used in to check the termination condition. Because a 0 (zero) is treated the same as `false`, the statement can be written without a comparison operator. However, in the **do...while** loop, the `$counter` variable is already 0 (zero) so we must put in a comparative operator (>=) to avoid an infinite loop.

**The for Loop**

Most languages have what is often called a “for-next” loop. This kind of loop sets up a starting point, a conditional expression and a change expression (typically increment or decrement). In PHP the format is,

```php
for(begin; condition; change)
{
    //do something"
}
```
The for loop is very compact in that the starting point, termination condition and change mechanism can be placed into a single parenthetical statement. The following snippet shows a simple example:

```php
for($tracker=0;$tracker < 20; $tracker++)
{
    echo $tracker . "<br>";
}
```

Often the condition to terminate the for loop will be a variable or the property of an object that can be passed to the termination condition. Other times, the condition is unknown, and so requires a different kind of loop described in the next section.

**The foreach Loop**

When working with objects you may want to iterate through all of the contents, but you're not sure what properties the object contains you will find the foreach loop very helpful. The basic format is:

```php
foreach($objName as $someElement)
{
    // $someElement is current value of property
}
```

Besides iterating through an unknown object, when you have an object with lots of properties, it's a much faster way to pull out all of properties and their values. Listings 5-7 and 5-7a show a class with four public properties (FullOfProperties) and another class (GetAll) that instantiates an instance of it and extracts all of the property values. The foreach loop allows a quick way to extract all of the properties.

*Listing 5-7: FullOfProperties.php*

```php
<?php
class FullOfProperties
{
    public $alpha;
    public $beta;
    public $gamma;
    public $delta;

    public function __construct()
    {
        $this->alpha="apples";
        $this->beta="bananas";
        $this->gamma="grapes";
        $this->delta="dates";
    }
}
?>
```
Listing 5-7a: GetAll.php

```php
<?php
ini_set("display_errors","1");
ERROR_REPORTING(E_ALL);
include_once("FullOfProperties.php");
class GetAll{
    private $objNow;
    public function __construct()
    {
        //Create an object from a class
        $objNow=new FullOfProperties();
        //Iterate through properties of the object
        foreach($objNow as $someElement)
        {
            echo $someElement . "<br>";
        }
    }
}
$worker = new GetAll();
?>

(Output)
apples
bananas
grapes
dates
```

As Listings 5-7/7a illustrate, the loop in the **GetAll** class does not specify a key of any sort, but instead iterates through the entire object and pulls out all of the property values.

In some cases you may want to know the property name as well as its value. For example, suppose you want to find the first and last names of clients and put them into a single string variable. In order to do that, you’d need to know the names of the variables.

To see how to find the property names you can specify each property and its current value as a key/value pair. For example, change the **foreach** statement in Listing 5-7a to the following:

```php
foreach($objNow as $key => $someValue)
{
    echo $key . "=" . $someValue . "<br>";
}
```
Now when you test it, you find the following output:

\[
\begin{align*}
\text{alpha} &= \text{apples} \\
\text{beta} &= \text{bananas} \\
\text{gamma} &= \text{grapes} \\
\text{delta} &= \text{dates}
\end{align*}
\]

As you can see, the property names are available along with their current values.

If you want to preserve encapsulation using the \texttt{private} visibility modifier, one way you can do that is to transfer the values of the properties to an array with private visibility and let the constructor function or some other public method display the values using the \texttt{foreach} loop to work through the array. Listing 5-8 shows one way to do this:

\textit{Listing 5-8: PrivateProps.php}

```php
<?php

class PrivateProps {
    private $alpha;
    private $beta;
    private $gamma;
    private $delta;
    private $arrayHolder;

    function __construct() {
        $this->assignPrivate();
        $this->arrayHolder=array($this->alpha,$this->beta,$this->gamma,$this->delta);

        foreach($this->arrayHolder as $someValue) {
            echo $someValue . "<br>";
        }
    }

    private function assignPrivate() {
        $this->alpha="apples";
        $this->beta="bananas";
        $this->gamma="grapes";
        $this->delta="dates";
    }
}
$objName=new PrivateProps();
?>
```
In the new output, the keys have been changed from the property names to numeric keys in the array. The values of the private properties were transferred to a private array that displays the value through the public constructor function.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand the basic building blocks in OOP PHP code.

- Write a simple sequential program that has a conditional statement that is a right or wrong “guess” based on the value assigned to a test variable. Create two variables. One will be the `$correct` and the other will be `$incorrect`. Use the conditional statement to evaluate whether they match or not. (Further on in this book, we’ll look at a real guessing game based on dynamic user input.)
- Create a program that will count to your age and display each year on a single line using a `for` loop. Your age should be placed into a variable that is used as the conditional value in a `for` loop.
- Create an associative array using the following keys and values: (See Chapter 3 to see how to create an associative array).

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>quarterback</td>
<td>Matt Smith</td>
</tr>
<tr>
<td>rtTackle</td>
<td>Adam Jones</td>
</tr>
<tr>
<td>ltTackle</td>
<td>Osmund Keller</td>
</tr>
<tr>
<td>rtGuard</td>
<td>Jared Finster</td>
</tr>
<tr>
<td>ltGuard</td>
<td>Mike Jackson</td>
</tr>
<tr>
<td>rtEnd</td>
<td>Jeff Jefferson</td>
</tr>
</tbody>
</table>

Use a `foreach` loop to display all of the keys and associated values using `echo` statements.
Chapter 6: Functions and Methods

In virtually all of the previous chapters, you've seen examples of functions at work. Functions created by the developer are called user functions. (Maybe they should be called developer functions.) In the flow of control, functions represent blocks of code that can be launched only by some kind of reference to the function label. The functions in classes are called methods, and they are a building block in all object-oriented programming. This chapter examines different features of methods and their use in an OOP environment.

Creating and Launching Functions

In the most fundamental sense, a function is nothing more than a block of code identified by the keyword function, a name (label) for the function and a set of operations within the curly braces that define the boundaries of the function. To launch a function, the function label is placed in a statement outside of the function. The following two subsections describe the fundamental types of functions.

Basic Function

Figure 6-1 shows the parts of the basic function. It is a block of code that is launched when it is called by the function and not when it is initially processed.

```
function label(): return
{
    //operations
}
```

Figure 6-1: Basic Function

The return type (: return) is optional and represents what can be expected from the operation.

When your PHP code is being processed, none of the code inside the function’s curly braces ( {...} ) launches. Instead, a statement must call the function. For example, the statement `greetings()` calls the function with the label `greetings` in Listing 6-1:

```
Listing 6-1: Greet.php
class Greet
{
    public function greetings(): string
    {
        $message="Your method greets you.";
        echo $message;
    }
}
```
$greeter=new Greet();
$greeter->greetings();

(Output)
Your method greets you.

The code that makes up the function is a list of statements that do nothing until the function is called. A basic function may contain other functions or any other code you can find in any other part of the program. Because functions are objects' basic building blocks, they play a key role in OOP.

**Constructor Function**
One function that has a special status is the constructor function. In PHP, you've seen that the function with the special name `__construct`, is the constructor function.

Listing 6-2 shows how the constructor function is correctly used:

```php
Listing 6-2: ShowCon.php
<?php
class ShowCon
{
    function __construct()
    {
        echo "This uses __construct <br>";
    }
}

$showOff=new ShowCon();
?>
```

(Output)
This uses __construct

The important feature of the constructor function is that it automatically launches as soon as an instance of the class is instantiated. If you do not include a constructor function in your class, one is automatically set up but does not launch any actions. It only initializes your properties.

**Adding Parameters and Arguments**
Functions are made far more flexible by adding parameters to the function. A parameter is an abstract value the developer uses to pass actual values called arguments when the function is used. The argument list in the function’s parameter section provides a list of expected values to be passed when the function is used.
Figure 6-2 shows a function with parameters:

```php
function label(type $a, type $b) {
    //operations using $a and $b
}
```

Figure 6-2: Function with parameters

While Figure 6-2 shows only two parameters, PHP does not specify a limit for the number of parameters. Generally, the number of values that must be passed dictates the number of parameters in a function. With PHP 7 you can add data types of strings, integers, Booleans and floats to the required data types. You can also add object data types such as interface names, class names, and arrays.

**Parameters**

Because PHP does not have strictly typed data, parameters in a function do not have a data type. However, when you build your parameters, you generally assume a data type. For example, suppose you make a function that calculates the price of an item and its sales tax. You expect a numeric value place into the parameters you have reserved for a price and a tax percentage as shown in Listing 6-3:

**Listing 6-3: Taxman.php**

```php
class Taxman {
    function priceTax(float $price, float $tax)
    {
        $tax=$tax/100;
        $total=$price + ($price * $tax);
        $total=round($total,2);
        echo "Your total is $" . $total;
    }
}
$doTax=new Taxman();
$doTax->priceTax(19.95,6);
```

(Output)

Your total is $20.15

The function employs the built-in `round()` function to round the value to 2 decimal places, illustrating that a function can contain other functions.
Parameter by Reference
You may also add a parameter by reference. As shown in Chapter 3, an assignment by reference uses the format:

```php
$variableA = 20;
$variableB = &$variableA;
```

If the referenced variable changes ($variableA), so too does the value of the variable that is assigned a value by reference ($variableB). Listing 6-4 shows how such an assignment works with a function:

```
Listing 6-4: TaxRef.php
<?php

class TaxRef {
    function taxByReference(float &$price) {
        $price += round($price * .06, 2);
        echo "Your total is $" . $price . "<br>";
    }
}

$bigSale = 54;
$taxNow = new TaxRef();
$taxNow->taxByReference($bigSale);
$bigSale = 66;
$taxNow->taxByReference($bigSale);
?>
```

(Output)
Your total is $57.24
Your total is $69.96

As the referenced variable changes so too does the value of the argument.

Default Arguments in Parameters
In many situations when using PHP, you may wish to use a default argument. For instance, if you create a questionnaire asking about the type of fuel people use in their cars, chances are that most still use gasoline. So you, may have a function that looks like Listing 6-5:

```
Listing 6-5: Fuel.php
<?php

class Fuel {
    function typeOfFuel(string $fuel="gasoline") {
```
As you can see from the output, when the default value is used, no arguments are placed in the function parenthesis.

Arguments
Differentiating parameters from arguments is a matter of distinguishing the abstract from the concrete. The parameters expect an actual value for each variable placed in the argument list. The actual value is the argument. In the example,

```
function priceTax($price, $tax)...
```

The variables $price and $tax represent parameters that make up the argument list and the statement,

```
priceTax(19.95, 6);
```

contains the arguments 19.95 and 6.

In the construction of the priceTax() function, every instance of $tax and $price in the definition of the function becomes the actual value of the arguments in the use of the function. To see this, consider how the function looks if we had a function X-ray machine that could see the insides of a function once it is supplied with argument as shown in Listing 6-6.

```
Listing 6-6: PriceTax.php
<?php
class PriceTax {
    public function taxedPrice(float $price, float $tax) {
```
$tax /= 100;
$total=$price + ($price * $tax);
$total=round($total,2);
echo "Your total is $" . $total;
}
}$calc=new PriceTax();
$calc->taxedPrice(19.95,6);
?>

(Output)
Your total is $21.15

As you can see the variable $total is not a parameter, but it uses the values of the arguments to store the sum of the price plus the amount of tax added.

Don’t let the differences between parameters and arguments throw you. They are often used interchangeably, but as long as you remember that the parameters are placeholders for the actual values (arguments) that will be used in a function you should be able to work with them.

Values that are passed as arguments in a function can be any PHP value. Besides using literals, you can pass arrays, variables, constants, Booleans—anything that holds a value. Also, besides using built-in functions like round(), you can use functions created by the developer in the body of the function.

**Delivering Results with Return**

As a rule of thumb in OOP, use return for generated data instead of an output statement. The general model in OOP that works the best is to make requests for something from an object instead of having the object decide what to do with the data it generates. There will be exceptions, but where possible and sensible, use (for example) return instead of echo.

In object oriented programming, often you will need a value generated by a function. Thus far, the values generated in the function examples have been sent to the screen using the echo statement. However, using the return statement, you can send the value generated in the function’s operations to a variable. Figure 6-3 shows the general format using the return statement:

```php
function label(type $a, type $b): return
{
    //operations to generate value
    return value
}
```
Figure 6-3: Using the return statement in a function

You don’t have to use parameters with a return statement, but you may if you wish. In general the function with a return acts something like a variable storing information. When you do have a return value in a class method in PHP 7 you can specify a return data type. For example, consider Listing 6-7:

Listing 6-7 ReturnInfo.php

```php
<?php
class ReturnInfo
{
    public function welcome(string $name): string
    {
        $greet="The Acme Corporation welcomes our new vice president, $name";
        return $greet;
    }
}
$gimme= new ReturnInfo();
//Returned value is stored in variable, $showme
$showme=$gimme->welcome("Henry Smith");
echo $showme;
?>

(Output)
The Acme Corporation welcomes our new vice president, Henry Smith
```

In the next section, and in OOP operations in general, you will find that the return statement is an essential way that objects communicate with one another. Inside methods, the return statement acts as a communicator between its class and a requesting client.

Methods: Functions in Classes

Objects contain packets of code that can be launched from an instance of a class that requires little work on the part of the developer. All he has to do is to instantiate an object and request the method. A well-organized program using and re-using methods is far easier to maintain than a single file with all of the code stuffed in one place.

A general OOP principle is that a class should only do one thing. Known as the single responsibility rule, it reminds us to keep classes simple and not to see how many different properties and methods we can cram in a single class. By having a single
responsibility, we make re-use much easier and changing a program is less likely to become entangled in a web of dependencies.

**Visibility**

One of the essential OOP structures in PHP is the visibility structure. Both properties and methods in a class can be assigned one of three visibility modifiers:

- **private**: Only accessed from within a class that creates the method
- **protected**: Only accessed from parent class or child of parent that defines method
- **public**: Accessible from anywhere

Regardless of visibility all methods, if launched from within the class that creates them, must use the `$this` statement.

```php
$this->classMethod();
```

The following subsections provide explanations and examples of all method visibility. Each is important for object-oriented programming, and each has a unique role.

**Private Methods: Encapsulation**

One of the key principles of OOP is *encapsulation*. What does it mean? All too often, encapsulation is described in terms of *restricting access*, and while that is a key feature, a more important image of encapsulation is that of an object with internally working mechanisms.

Take for example a toy robot. The robot has a switch to make it go forwards, backwards, left and right. **The switch represents access.** While you may not think of the switch as protecting the integrity of the robot’s inner works, it does. Without a switch, the user would have to break apart the robot and pull wires to make different connections. That would not only screw up the robot, the procedure would also be a lot more difficult to work the thing.

Another example is your automobile. If you’re driving your car, you have access to the steering wheel, brake pedal, accelerator and the dial on your radio. Suppose that someone driving by stuck his hand out the window and grabbed your steering wheel and steered it where he wanted to go. That would break encapsulation—you don’t want him or any other nutcase to grab your steering wheel.

What does this have to do with programming? Well, just like the guy who grabs your steering wheel, you don’t want other objects in your program misbehaving by changing a property’s value. Further, an encapsulated object is a unit unto itself—it’s got a complete interworking system that you can depend on to do only certain things. If it’s not encapsulated, you cannot rely on it behaving as expected.
The easiest way in the world to encapsulate a property (variable, constant) or method (function) in a class is to make it private. This doesn’t mean that the class with the private properties and methods cannot be accessed. Rather, it can only be accessed the way you want it to be accessed—just like the switch on the toy robot or the steering wheel in your car. Listing 6-8 shows a class with several private methods launched by the constructor function:

Listing 6-8: PrivateMethods.php

```php
<?php
class PrivateMethods {
    function __construct() {
        $this->methodA();
        echo $this->methodB("Nancy");
        echo $this->methodC(5);
    }

    private function methodA() {
        $radius = 7;
        $diameter = $radius * pi();
        echo "This circle's diameter is $diameter centimeters.<br>";
    }

    private function methodB(string $someString): string {
        $happyBirthday = $someString . " is having a birthday today!<br>";
        return $happyBirthday;
    }

    private function methodC(int $someNumber): float {
        $cubic = pow($someNumber, 3);
        $message = "The cubic feet of the square box is $cubic.<br>";
        return $message;
    }
}

$privateTest = new PrivateMethods();
?>
```

(Output)
This circle's diameter is 21.991148575129 centimeters.
Nancy is having a birthday today!
The cubic feet of the square box is 125.

When using a class’ own methods or properties, they must be preceded with the $this-> statement. If you do not, you’ll see the following error message:

Fatal error: Call to undefined function...

The error message is a bit of a drama queen, but when you see undefined function where you are certain that you have defined the function, check to see if it needs the $this-> statement.

Listing 6-8 also uses a couple of built-in functions, pow(arg1, arg2) and pi(). The pow(arg1, arg2) function returns the value of the first argument raised to the power of the second argument. The pi() function is the constant value of mathematical pi or 3.1415926535898. Finally, note that the function parameter type is an integer (int) and the expected return is a floating-point (float) value.

Protected Methods: Close Family Members Only

Protected methods may also be used to more loosely encapsulate operations. The class that creates it and all child classes can access a method that is protected. A child class is simply any class that extends the parent class or its children. When a child class uses a protected method, it must access it through its constructor function or some other public function for it to be generally accessible for use. For example, Listing 6-9 shows a child class (ProtectedChild) that uses its inherited protected methods through a constructor function and the public function viewProtected().

Listing 6-9: ProtectedMethods

```php
<?php
class ProtectedMethods {
    protected function money() {
        echo "Junior inherits all the money!<br>";
    }

    protected function realEstate(string $house) : string {
        $lulu="Lulu gets the $house. <br>";
        return $lulu;
    }

    protected function auto($car) {
        $dobs="Dobs, our chauffer, gets the $car.";
        return $dobs;
    }
}
```
class ProtectedChild extends ProtectedMethods
{
    function __construct()
    {
        $this->money();
        echo $this->realEstate("chicken coop");
    }

    public function viewProtected()
    {
        echo $this->auto("Rolls-Royce");
    }
}

$privateTest=new ProtectedChild();
$privateTest->viewProtected();

<?

(Output)
Junior inherits all the money!
Lulu gets the chicken coop.
Dobs, our chauffer gets the Rolls-Royce.

Using protected visibility with methods allows for both encapsulation and inheritance of the methods. The best use of inheritance is with abstract classes so that the child classes can add their own details to the inherited materials. Chapter 7 examines how to work with abstract classes and interfaces.

Public Methods
Remember the switch on the toy robot? You are able to make the robot go left, right, forward and backward by manipulating the switch. However, you know that motors, servos, and mechanisms encapsulated inside the robot actually make the moves possible. The switch just makes accessing those inner-workings accessible (or visible).

To see how to public methods interact with public methods that can be accessed from other classes and generally outside of a class, Listing 6-10 shows a class that simulates a robot with the switch that makes it do different things.

Listing 6-10: Robot.php
<?php
class Robot
{
    //Private methods to encapsulate
    private function goForward()
private function goBackwards()
{
    echo "Moving backwards Master...whirrr <br>

private function goLeft()
{
    echo "Moving left Master...whirrr <br>

private function goRight()
{
    echo "Moving right Master...whirrr <br>

// Public methods to access private methods
public function switchForward()
{
    $this->goForward();
}

public function switchBackwards()
{
    $this->goBackwards();
}

public function switchLeft()
{
    $this->goLeft();
}

public function switchRight()
{
    $this->goRight();
}

$fred=new Robot();
$fred->switchForward();
$fred->switchBackwards();
$fred->switchLeft();
$fred->switchRight();
?>
In looking at the program in Listing 6-10, you may be thinking that it’d be a lot easier just to have four `echo` statements generate four strings that describe what the “robot” is doing without using any functions at all. You’d be right. However, the point is to show how public methods work while keeping the actual inner-workings encapsulated (private.) When you begin creating more sophisticated programs, this approach (which is just plain OOP programming) will be welcomed. You will be able to prevent unwanted values changed in the private methods because the only way to access them is through a public method that can only work if correctly formed. In this way, your program is far less likely to become knotted up in a big tangled mess that looks like an unruly fishing line ball.

Anonymous Functions

Functions with no identifier (name) are descriptively called *anonymous functions*. Anonymous functions can never be methods, but they can be used in the makeup of methods.

Anonymous functions, sometimes called *lambda functions* (λ functions), can be used within methods for certain operations that would otherwise require named functions. They are also used for *callbacks*. One convention worth using with anonymous functions, especially to prevent them being mistaken for methods, is to put them on a single line. For example, the following class has a method named `firstLambda()`, and the method includes an anonymous function. The anonymous function is assigned to a variable, `$priceNship`.

```php
<?php
class Lambda
{
    public function firstLambda(float $cost): float
    {
        $priceNship = function ($x) {return $x + ($x * .1);};

        /*the anonymous function is return value of the firstLambda method*/

        return $priceNship($cost);
    }
}

$worker=new Lambda();
echo $worker->firstLambda(23.50);
```
The operation that makes up the anonymous function has the form of a mathematical expression more than a PHP statement, and so another useful way to think of anonymous functions is in terms of programmable expressions. Of course, an anonymous function can take a more statement-like set of behavior if necessary. In order to format anonymous functions correctly it needs to be set up with the right properties. Figure 6-2 details the function uses in the firstLambda() method in the Lambda class.

```
function ($x) {return $x + ($x * .1);};
```

**Figure 6-2: Anonymous Function**

The word `function` initiates the definition followed by parentheses with or without a parameter. One convention, borrowed from functional programming, is to place all of the function on a single line. Following the parentheses is the first curly brace followed by a `return` keyword. Instead of using `return`, you can use an I/O statement such as `echo`, `printf` or `var_dump`, but throughout this book, the preference will be to use the `return` statement. Next comes the operation or expression to be returned, followed by a `semi-colon` and then followed by the closing `curly brace` and a second `semi-colon`.

**Anonymous functions inheriting anonymous functions**

In order to keep anonymous functions compact and reusable, they can be inherited. Just like a class can inherit from a parent class, an anonymous function can inherit another anonymous function.

```php
<?php
class InheritLambda
{
    public function loadedLambda(float $cost): float
    {
```
$priceNship = function ($x) {return $x + ($x * .1);};

$priceShipNtax = function ($x) use ($priceNship) {return $priceNship($x) + ($x * .06);};

/*anonymous function is return value of loadedLambda method*/

return $priceShipNtax($cost);

$worker=new InheritLambda();
echo $worker->loadedLambda(23.50);
?>

The new anonymous function is stored in the variable $priceShipNtax. It uses the anonymous function stored in the variable $priceNship and includes it in its definition with the keyword use. Figure 6-3 provides the details for anonymous function inheritance.

![Use statement and anonymous function to use](image)

**Function ($x) use ($func1) {return $func1($x) + ($x * .06);};**

- Inherited function now part of new function.
- Added expression
- Note: 2 semi-colons

*Figure 6-3: Anonymous Function with Inheritance*

You can incorporate as many inherited anonymous functions as you need, using the format where $a, $b and $c are added anonymous functions:

```php
function ($x) use($a, $b, $c) {return $a() $b() $c() + $x;};
```

Keep in mind that adding more and more anonymous functions can lead to a complexity that may be difficult to unravel in the debugging process. Like all things OOP, reuse and clarity require judicious choices.

**Callbacks**

Callbacks typically are created with an anonymous function as an argument in another function. You can even think of callbacks as variables containing
information needed in the execution of the function. However, instead of a value stored in a variable, the value is the return value of the function. The following code snippet shows a callback in a function:

```php
$lambda = function($x) { return $x * $x;};

function useCallBack($callback)
{
    return "The square of callback value is $callback .";
}
return useCallBack($lambda(7));
```

The anonymous function is stored in the $lambda variable. When invoked, the `useCallBack($callback)` function expects a function as an argument. The following shows the callback in the context of a class method:

```php
<?php

class SimpleCallback
{
    public function doCallback()
    {
        $lambda = function($x) { return $x * $x;};

        function useCallBack($callback)
        {
            return "The square of callback argument is $callback .";
        }
        return useCallBack($lambda(7));
    }
}

$worker = new SimpleCallback();
echo $worker->doCallback();
?>
```

(Output)

The square of callback argument is 49.

In PHP 7, typically, callbacks are associated with events that invoke the anonymous function expected as an argument. For example, selection of an element for output can be controlled by a callback when iterating through an array. PHP has built-in functions that expect a callback as a parameter. One built-in function that expects a callback in its parameter is the `array_filter()` function. It creates a new array based on the callback function acting as a filter. Figure 6-4 shows its structure:
Figure 6-4: The array_filter() function includes a callback parameter.

For example, the following anonymous function filters out values ($fig) less than 100:

```php
function($fig) { return $fig >= 100; }
```

The anonymous function is not stored in a variable, but instead is created directly in the array_filter() function as a callback:

```php
array_filter($data, function($fig) { return $fig >= 100; });
```

This illustrates that even though anonymous function are often stored in PHP variables, they can be entered directly as callback arguments. Further, when you do not plan on reusing a function as a method, placing them directly into callbacks makes perfect sense. The following class shows how a class can use the array_filter() function with an anonymous function callback:

```php
<?php
class CallbackFilter
{
    public function doCallback() : string
    {
        $bundle="Accounts of 100 or greater:<br />";
        $data = array(25,713,91,34,56,902,821,48,100);
        $evenNumbers = array_filter($data, function($fig) { return $fig >= 100; });
        //Uses array_filter
        foreach($evenNumbers as $value)
        {
            $bundle .= $value . "<br />";
        }
    }
}
```
Anonymous functions are not new to PHP (introduced in PHP 5.3), but they are increasingly important because of their use in functional programming. They can add an elegant tough to PHP programs in an OOP context.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand working with different kinds of methods in an OOP environment.

- Create a class with three private functions (methods) that are all launched by a constructor function. The first method requires no arguments, the second method requires one string argument, and the third performs determines that area of a circle with the radius as a numeric argument. (Remember \( \text{pi()} \times \text{pow(r,2)} \) is the area of a circle where \( r= \) radius value.)
- Make a parent class with three protected methods, each one doing a different simple math problem requiring a single parameter. The first will add the argument value to itself; the second will subtract the argument value from 100, and the third determines the square of the argument value using the `pow()` function. The create three child classes with each using only one of the protected methods.
- Create a class with a private method that has a single parameter for age that subtracts the user’s age from the retirement age (65) to determine how many years they have until requirement. The private method can be accessed through a public method in the class through which the argument may be passed.
- Create a class with a single method that incorporates two anonymous functions. One of the two anonymous functions incorporates the other function employing the `use` keyword. The combined function calculates square footage of a wall with 15 x 8 (w/h) feet and the cubic feet with a room with walls of that dimension enclosing an area.
Chapter 7: Interfaces, Abstract Classes, Methods and Polymorphism

One of the most important elements in object-oriented programming is the use of abstract classes and interfaces. In Chapter 6, polymorphism was briefly mentioned, and in this chapter, we will examine the concept in more detail and the role that both abstract classes and interfaces play in polymorphism.

Abstract Classes

An abstract class is a class that cannot be instantiated, but instead it is used to host both abstract and concrete methods and properties. The basic use of an abstract class is to provide both an interface and a commonly used set of methods and/or properties that can be extended for different but related concreted uses. For example, in Chapter 11, you will be looking at ways to access a database. Every database has a common login sequence requiring the name of a hosting server, a username and password and the name of a database. Rather than having to re-code the values each time you write a different SQL command, you can store them in an abstract class and then use them with different SQL commands.

At the same time that you want to use a common set of values for the login procedures, you also want to have some flexibility about the SQL commands that you need to issue through PHP. However, you know you have to issue them. So, you can use an abstract method in the abstract class. That means the function has only the signature of the method leaving the actual implementation to the child classes. For example, Listing 7-1 shows an example the abstract class is named, AbstractSQLClass, and it cannot be instantiated. Rather, it is sub-classed by a class named ReadData. The ReadData class then issues “SQL commands” in PHP to “read” the data in a table specified in the doSQL() method’s argument. The value of using an abstract class in this manner is that you can extend it with several different specific implementations while using the same common property values.

Listing 7-1: AbstractSQLClass.php

```php
<?php
abstract class AbstractSQLClass
{
    protected $host = "myLocalHost";
    protected $uName = "myUserName";
    protected $dbName = "myDataBase";
    protected $pw = "myReallySecretPassword";

    abstract protected function doSQL($mySQLelement);
}
```
Listing 7-1a ReadData.php
<?php
include_once(
class ReadData extends AbstractSQLClass
{
    //Provide details to inherited method
    protected function doSQL($mySQLelement)
    {
        echo "Use the username '$this->uName' and password, \'<br> '$this->pw', and log on to $this->host to \'<br>";
        echo "open $mySQLelement in the $this->dbName database to read the data.";
    }
    public function useSQL($sql)
    {
        $this->doSQL($sql);
    }
}
$snoop = new ReadData();
$snoop->useSQL("UserSurvey");
?>

(Outcome)
Use the username 'myUserName' and password, 'myReallySecretPassword', and log on to myLocalHost to open UserSurvey in the myDataBase database to read the data.

Listing 7-1 is a trivial example, but in Chapters 9 and 10 when you start working with databases and tables, you'll find that it is a way to organize your work with SQL using the PHP mysqli() objects.

Abstract Methods
If a class has at least one abstract method, it must be an abstract class. As you saw in Listing 7-1, the doSQL($sql) method is defined as abstract. If you created a class without the abstract modifier and included an abstract method, your program would not work.

The use of an abstract method is to allow you flexibility while at the same time letting you use a common method. Listing 7-2 shows an abstract class with one common (non-abstract) method and an abstract method.

Listing 7-2: AbstractMethod.php
<?php
abstract class AbstractMethod
{
    abstract public function doWhatever($whatever);

    public function alwaysDoThis()
    {
        echo "<br>Whacka whacka doo!";
    }
}

class DoSomething extends AbstractMethod
{
    public function doWhatever($whatever)
    {
        echo $whatever;
    }
}

class DoSomethingElse extends AbstractMethod
{
    public function doWhatever($whatever)
    {
        return $whatever;
    }
}

$myThing= new DoSomething;
$myThing->doWhatever("Take it one step at a time.");
$myThing->alwaysDoThis();

$myOtherThing= new DoSomethingElse;
$myString= $myOtherThing->doWhatever("<p>Try to be patient.");
echo $myString;
$myOtherThing->alwaysDoThis();
?>

(Output)
Take it one step at a time.
Whacka whacka doo!

Try to be patient.
Whacka whacka doo!

The common method always does the same thing. However, if you look carefully at the implementation of the abstract method in the two child classes, each has a
different implementation. In the first one, the argument is sent to the screen using the `echo` statement, while in the second child class, the `return` statement makes the argument available as a value. The same signature is used in both child classes, and so it doesn’t matter what is in the content (between the curly braces).

**Limiting Hierarchy with Final**

One could (and, alas, some do) keep extending from an abstract parent class into grandchildren and great-grandchildren *ad infinitum* classes. To stop both overriding a concrete method and further extension from a concrete class, you can employ the `final` keyword. Listings 7-3a to 7-3e show examples with what works and what does not work. (Note that the keyword `require` is used with the two grandchild classes instead of `include_once()`. It throws a `fatal error` to add some drama to those classes that try to go beyond the `final` limitation.)

**Listing 7-3a: AbsClass.php**

```php
<?php
abstract class AbsClass
{
    abstract public function echoThis($myEcho);
}
?>
```

**Listing 7-3b: OneChild.php**

```php
<?php
include_once('AbsClass.php');
final class OneChild extends AbsClass
{
    public function echoThis($myEcho)
    {
        echo $myEcho;
    }
}
$test=new OneChild();
$test->echoThis("This works fine.<br>");
?>
```

**Listing 7-3c: GrandChildOne.php**

```php
<?
//This will not work because it attempts to extend a final class
require('OneChild.php');
class GrandChildOne extends OneChild
{
    function __construct()
    {
        echo "This will never be seen...";
    }
```
Listing 7-3d: TwoChild.php

```php
<?php
include_once('AbsClass.php');
class TwoChild extends AbsClass
{
    final public function echoThis($myEcho)
    {
        echo "This is a final function: $myEcho";
    }
}

$test2=new TwoChild;
$test2->echoThis("...so don't try and override it!<br>");
?>
```

Listing 7-3e: GrandChildTwo.php

```php
<?php
//This will not work because it attempts to override a final method
require('TwoChild.php');
class GrandChildTwo extends TwoChild
{
    public function echoThis($myEcho)
    {
        echo "This too will never see the light of day...$myEcho";
    }
}

$elBomb2=new GrandChildTwo();
$elBomb2->echoThis("...so don't try and override it!<br>");
?>
```

In the outcomes, you will see the first part of the two "grandchild" classes seem to work and then break down. Keep in mind that they have inherited the concrete parent classes, which operate correctly. The class fails when it begins to incorporate those parts of the class that go beyond the fatal point. So while they appear to be working correctly, they throw a Fatal error after the initial `echo` output.

(Output) OneChild, and TwoChild
This works fine.
This is a final function: ...so don't try and override it!

This works fine. (GrandChildOne)
Fatal error: Class GrandChildOne may not inherit from final class (OneChild) in /Users/wdsanders/Sites/php/PHPbook/Examples/GrandChildOne.php on line 5

This is a final function: ...so don't try and override it! (GrandChildTwo)

Fatal error: Cannot override final method TwoChild::echoThis() in /Users/wdsanders/Sites/php/PHPbook/Examples/GrandChildTwo.php on line 5

While inheritance is an essential part of OOP, you can create a tangled web if you keep sub-classing from an abstract parent class. Likewise, because you can override an abstract method, if you keep overriding the method in grandchildren classes, the whole purpose of inheritance (re-use) is lost. Hierarchy works best when it is shallow and not too broad.

**Interfaces**

Where you have no concrete properties or methods but you want to have a common set of methods for multiple classes, you can use an interface. In this way, you can create different methods and common signatures and use them in different implementations, depending on what you want to do. To declare an interface, simply use the keyword interface instead of class. Then add all of the method signatures you want. All methods in an interface must be public. Listings 7-4a and 7-4b show a simple example:

*Listing 7-4a: Interface.php*

```php
<?php

interface IBasicInterface
{
    public function simple();
    public function simpleArg($myParam);
}
?>
```

*Listing 7-4b: Interface.php*

```php
<?php
include_once('IBasicInterface');
class BasicClass1 implements IBasicInterface
{
    public function simple()
    {
        $alpha = "Let's take a trip on Route";
        $beta = 66;
    }
}````
Both the method with the argument, `simpleArg($myParam)` and the one without, `simple()`, can have any variety of content. As long as the signatures match the interface, they will work fine. When it comes to changes in the methods’ content, all that needs to be changed is the implementation details and the rest of the program—including all related classes and objects will function just fine.

**Constants in PHP Interfaces**

While an interface cannot have properties, it can have constants. If you’re familiar with other programming languages, you cannot include properties in an interface, constants or otherwise. However, PHP allows for that feature. They can be very handy where you have some common values you’ll need, such as database logon settings. Listings 7-5a and 7-5b show how to implement an interface with constants:

**Listing 7-5a: ConstantInterface.php**

```php
<?php

interface IConstantInterface
{
    const HOST = "myLocalHost";
    const UNAME = "myUserName";
    const DBNAME = "myDataBase";
    const PW = "myReallySecretPassword";

    public function sqlWork($table);
}
?>
```

**Listing 7-5b: DataIn.php**
<?
include_once('IConstantInterface.php');
class DataIn implements IConstantInterface
{
    public function sqlWork($table)
    {
        echo "Query table $table in the following:<p>");
        echo IConstantInterface::HOST . "<br>");
        echo IConstantInterface::DBNAME . "<br>");
        echo IConstantInterface::UNAME . "<br>");
        echo IConstantInterface::PW . "<br>");
    }
}
$useConstant = new DataIn();
$useConstant->sqlWork("Customers");
?>

(Output)
Query table Customers in the following:

  myLocalHost
  myDataBase
  myUserName
  myReallySecretPassword

With each implementation of the interface, you have both the detail in the constants and the flexibility of the methods to implement.

**Why Use Interfaces?**

As you begin to create larger applications, you will see the advantage of using interfaces. If you imagine a PHP program that you want to use to enter and read data into a database and nothing more, and you are positive you'll never change it, you probably won't have a great deal of use for interfaces. However, if you begin to have larger and more complex programs, interfaces work as an organizing device. When you separate your interface from your implementation (what your class actually does), you have far more flexibility.

Going back to the example of a PHP program that you build to enter and read data, you have to set up certain features so that everything works together. The data that the user enters must be passed to PHP variables or parameters, you have to open and prepare the database, select the table and enter the SQL commands. If your client goes through an interface, you can add and change services that the client can request without having to re-write the whole program. For example, the following list shows the kinds of requests typically sent to a MySQL database with PHP:
• Insert data
• Read data
• Update data
• Search for data
• Delete data

Typically, you need an implementation to insert data so that you have some data to use in a MySQL database. Then you'll need to read the data and that requires another module. Likewise, you may have several other modules for working with a database. However, if you create an interface, you've predefined the signatures for the methods that the implementation must use. You may initially find it a bit inconvenient, but as your program grows, you'll be glad you have an interface. Likewise, if you change it or even create a whole new table, you can easily reuse the same structure.

Polymorphism

The term polymorphism refers to a key concept in OOP. In the most specific sense polymorphism can be seen in different implementations of methods from an interface. In this chapter, you've seen many examples of polymorphism because the implementation of a method from a single interface may have many (poly) different forms (morphism.) For example, Listings 7-6a to 7-6c show a simple yet clear example of polymorphism.

Listing 7-6a: IPoly.php
<?php
interface IPoly
{
    public function easyMe($behavior);
}
?>

Listing 7-6b: Good.php
<?php
include_once('IPoly.php');
class Good implements IPoly
{
    public function easyMe($behavior)
    {
        return "If you $behavior you will be loved by all<br>";
    }
}

$polyUp = new Good();
$sweetness = $polyUp->easyMe("are kind to animals");
echo $sweetness;
Listing 7-6c: Bad.php

```php
<?php
include_once('IPoly.php');
class Bad implements IPoly
{
    private $kindPoints=20;
    public function easyMe($behavior)
    {
        echo "$behavior is bad! You're going to confuse the poor creatures!<br>
        $this->kindPoints -=10;
        return $this->kindPoints;
    }
}
$polyDown = new Bad();
$lout = $polyDown->easyMe("Tricking squirrels");
echo "You now only have $lout kindness points left!";
?>

(Output)

Good.php
If you are kind to animals you will be loved by all!

Bad.php
Tricking squirrels is bad! You're going to confuse the poor creatures!
You now only have 10 kindness points left!

The method `easyMe($behavior)` behaves very differently when called from different instances of different classes. However, the method has the identical signature. Polymorphism is at work in many places. First of all, using identical methods with identical implementations, you have polymorphism in the parameter of the method. You can add anything you want as an argument. So, the method has a built-in polymorphism—in fact any method with an argument implies polymorphic structures.

Second, the same method with the same signature is implemented in two very different ways. However, it is called though an instance in the same way—`$instanceName->easyMe("string")`. In programming polymorphism let's you make fewer assumptions about a method. As long as you attend to the signature, it doesn't matter what instance is using it. If you change the implementation, it still works!
Both abstract classes and interfaces allow polymorphic structures in your program. In fact, you can think of abstract classes as interfaces—they just have added features, like common methods and properties.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand working with abstract classes and interfaces.

- Create an abstract class that has both a common method and an abstract method. Create two subclasses that use the common method but implement the abstract method in different ways.
- Create an abstract class with protected properties and two abstract methods. Create two subclasses that use the properties in the implemented abstract methods differently.
- Create an interface with three different methods and three constants. Create an implementation that implements all of the methods so that they incorporate the constants.
Chapter 8: Data Input and Passing Values to a PHP Class

PHP and HTML5 have a strong relationship. Data input and data queries are made through HTML5. Of course earlier versions of HTML can be used with PHP, but HTML5 has some new and very interesting form input types that can validate certain kinds of input, such as correctly formed emails and Web addresses. Once the data are entered you need a way to pass it from the HTML5 form to PHP.

Please Pass the Data
HTML5 forms and related input tags have a name attribute. When you press a submit button, all of values entered in the form are sent to the PHP file named in the form’s action attribute. For example, when you use the form tag,

```
<form action="catcher.php" method="post">
   <input type="text" name="customer">
   <input type="submit" value="Send data">
</form>
```

the action attribute instructs that all form data be sent to the PHP file named catcher.php. The method used is generally going to be either POST or GET. For the purpose of simplicity and consistency, we will be using the POST method.

In each form container, a submit button provides a way for the user to send the information to the file to the PHP file. If a Web page has multiple forms, it can have multiple submit buttons, but each button only sends that data contained in its own form; not all the forms on the Web page. The following snippet shows a submit button in a form container:

```
<form action="catcher.php" method="post">
   <input type="text" name="customer">
   <input type="submit" value="Send data">
</form>
```

If the same page had another form, it would need another submit button to send the data. The name on the button itself is assigned to the value attribute on the button. If no value is assigned, it defaults to “submit.” (Nobody wants to submit; so interface designers suggest providing a friendlier name like “send data” or “send information”.)

On the PHP side, to “catch” the data sent from an HTML page using POST, you have a predefined variable, $_POST. This predefined variable is an associative array of all variables passed using the POST method from a given form. A submit button only
sends the data in its form—not every form on the Web page. Figure 8-1 shows the relationship between the HTML form and the PHP program:

**HTML5**

```html
<input type="text" name="customer">
```

All data in the form are sent when user presses the Submit button.

**PHP**

```php
$ cus = $_POST['customer'];
```

Whatever the user types in the text input field named 'customer' is transferred to a variable in PHP via)$_POST['customer']. The PHP variable assigned the POST value then holds the value the user entered.

*Figure 8-1: Passing values between HTML and PHP*

To get started, build an HTML5 data input page. Listing 8-1 shows a data entry page using HTML5 input types, email and url:

**Listing 8-1: enterData.html**

```html
<!DOCTYPE HTML>
<html>
<head>
<style type="text/css">
    h1 {
        font-family: "Arial Black", Gadget, sans-serif;
        text-align: center;
        color: #1B8C4C;
        background-color: #D9B036;
    }
    body {
        font-family: Verdana, Geneva, sans-serif;
        background-color: #BFD9CA;
    }
</style>
<meta charset="UTF-8">
<title>Passing Data</title>
</head>
<body>
<h1>Please Pass the Data</h1>
</body>
</html>
```
<h3>Enter the following data and Press the Send data button</h3>
<form action="ShowMailInfo.php" method="post">
    <input type="text" size=30 name="customer">
    &nbsp;Enter Name <br>
    <input type="email" size=30 name="cMail">
    &nbsp;Enter Email <br>
    <input type="url" size=30 name="cWeb">
    &nbsp;Enter Web site address <br>
    <input type="submit" name="sender" value="Send data">
</form>

When you enter data into an HTML5 form in the Opera browser, it has some validation checking for both email and url input forms. Figure 8-2 shows what happens when the user clicks the submit button if the http:// is omitted from a Web address:

![Image showing validation error for URL input](image.png)

_figure 8-2: Validation checking in Opera of url input_

Once the user adds the missing http:// the PHP program data to be passed can be called by the program shown in Listing 8-2. Data validation is also possible on the server side using PHP code and you can write JavaScript routines in HTML to customize your data input validation.

_listing 8-2: ShowMailInfo.php_

```php
<?php

```
class ShowMailInfo
{
    private $cuName;
    private $cuMail;
    private $cuWeb;

    public function __construct()
    {
        $this->cuName=$_POST['customer'];
        $this->cuMail=$_POST['cMail'];
        $this->cuWeb=$_POST['cWeb'];
        $this->display();
    }

    private function display()
    {
        $showNow= "Information for customer,";
        $showNow .= " $this->cuName, with email ";
        $showNow .= ": $this->cuMail and a Website ";
        $showNow .="at $this->cuWeb has been received.";
        $showNow .="<br> Thank you.";
        echo $showNow;
    }
}
$mrPostMan=new ShowMailInfo();
?>

(Output)
Information for customer, Bill, with email: bill@here.com and a Website at http://www.php5dp.com has been received. Thank you.

The output reflects the data entered in the HTML5 entry page. (Note that the Web address is now http://www.php5dp.com indicating the necessary adjustment was made in the input by adding http:// to the URL address.)

**Passing Class**

With the advent of the predefined variable $_POST, PHP provided a more secure data transfer. However, the values are not encapsulated unless the variables are. One way to deal with encapsulation is with getters and setters. However, many have criticized using a getter/setter model because public visibility is used so that values can be passed to the setters. The problem with any public access to setters is that external objects can change the setter values in ways that may be problematic. However, public visibility is important for getters so that external objects can get the values needed in the program.
Listing 8-3 establishes private variables that receive their values through a private method. Essentially, the private method is a setter for all of the data passed from the HTML5 form. The public getter methods are functions that make the values available to other objects. However, they derive the getter values from private variables (properties) so that the values assigned to the variables are restricted to the class operations.

In order to test this class, first change the HTML5 form action values in Listing 8-3 to:

```html
<form action="SetterGetter.php" method="post">
```

Then, save the `SetterGetter.php` file (Listing 8-3 below) in the same directory as the HTML5 file.

### Listing 8-3: SetterGetter.php

```php
<?php

class SetterGetter
{
    private $cuName;
    private $cuMail;
    private $cuWeb;

    public function __construct()
    {
        $this->setter();
    }

    private function setter()
    {
        if(isset($_POST['sender']))
        {
            $this->cuName=$_POST['customer'];
            $this->cuMail=$_POST['cMail'];
            $this->cuWeb=$_POST['cWeb'];
        }
    }

    public function getCustomer()
    {
        return $this->cuName;
    }

    public function getEmail()
    {
        return $this->cuMail;
    }
}
```

public function getWeb()
{
    return $this->cuWeb;
}

$getData=new SetterGetter();
echo $getData->getCustomer() . "<br>";
echo $getData->getEmail() . "<br>";
echo $getData->getWeb() . "<br>

(Output)
Bill
bill@here.com
http://www.php5dp.com

The output, instead of merely being sent to the screen, can be used by another class. So the class can take care of transferring the HTML5 data into PHP variables and make them available for use by another class (object). Also, the individual values passed from the HTML page can be accessed individually.

**Passing Data to a Mail Class**

In setting up an e-commerce site, one of the essential elements that most such sites have is a contact form. This form collects information from the user and sends it on in an email to the site owner. Unlike emails that can be launched from HTML, PHP sends emails directly from the server so that you do not need an email client one the computer you are using.

The key PHP function for sending email is **mail** with the following format:

mail(recipient, subject, message contents, optional)

For example the following would send an email to the address **joe@asdf.com** with the subject line, **Important Mail**, and the message, “Hello world!” embedded in the variable **$message**.

$message ="Hello world!";
mail("joe@asdf.com", "Important Mail", $message);

In a fourth parameter, you can add more, but the basic email form only has three parameters—the recipient’s email address, a subject line and the contents of the message.

To get started, use the HTML5 page in Listing 8-4 to enter the comments to be sent as an email: 
Listing 8-4: mailEntry.html

<!DOCTYPE HTML>
<html>
<head>
<style type="text/css">
h2 {
    font-family:"Arial Black", Gadget, sans-serif;
    text-align:center;
    color:#C7B07B;
    font-style:italic;
    background-color:#570026;
    margin-left:-10px;
}
body {
    font-family:Verdana, Geneva, sans-serif;
    background-color:#FFF6D9;
    color:#705B35;
    margin-left:20px;
}
fieldset {
    color:#570026;
}
input,textarea {
    color:#570026;
}
</style>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8">
<title>Email Form</title>
</head>
<body>
<header>
<h2>Acme Email</h2>
</header>
<h4>Type in your thoughts and send them to us:</h4>
<form action="SetMailData.php" method="post">
    <input type="text" size=20 name="customer">
    &nbsp;First name <br>
    <input type="email" size=20 name="cMail">
    Email address <p>
    <fieldset>
        <legend>Please enter your message here</legend>
        <textarea name="cComments" cols="50"
row="10"></textarea>
    </fieldset>
</form>
The HTML5 data entry is almost identical to the one in Listing 8-1 except that it includes a text area element and does not include a request for a Web page address. Like the name and email input text fields, the text area element is identified with a name attribute. Figure 8-3 shows what the data input page looks like:

![Data input using a textarea element](image)

**Figure 8-3: Data input using a textarea element**

Using two classes, the HTML page calls the `SetMailData.php` file that contains the `SetMailData` class (object). This is where the data are passed to PHP. An important OOP principle is that *no class should have more than a single responsibility*, and so
the **SetMailData** single responsibility is to encapsulate the data sent from the HTML5 forms and pass on the information to the **SimpleMailer** class. It passes on the information sent from the HTML form with an instance of the **SimpleMailer** class. It sends the information to the mailer using the `sendMail()` method (of the SimpleMailer class) with the included values as parameters. Listing 8-5 shows how the class retrieves the values from HTML using the predefined `$_POST[]` and then places the values in private variables.

**Listing 8-5: SetMailData.php**

```php
<?php
include_once "SimpleMailer.php";
class SetMailData
{
    private $cuName;
    private $cuMail;
    private $cuComments;

    public function __construct()
    {
        $this->setter();
        $mailNow=new SimpleMailer();
        $mailNow->sendMail($this->cuName,$this->cuMail,$this->cuComments);
    }

    private function setter()
    {
        if(isset($_POST['sender']))
        {
            $this->cuName=$_POST['customer'];
            $this->cuMail=$_POST['cMail'];
            $this->cuComments=$_POST['cComments'];
        }
    }
}
$test=new SetMailData();
?>
```

Note how the line,

```
$mailNow->sendMail($this->cuName,$this->cuMail,$this->cuComments);
```

sends values in encapsulated private variables. These are used in the **SimpleMailer** class to send the appropriate email. Listing 8-6 shows how the variables are passed to the built-in `mail` method:
Listing 8-6: SimpleMailer.php

```php
class SimpleMailer
{
    public function sendMail($cus, $cMail, $comments)
    {
        mail("yourMail@yourHost.com", "Mail Test", $comments);
        echo "Mail from $cus with email address, $cMail, has been sent";
    }
}
?>
```

As you can see, the `SimpleMailer` class has a single method, `sendMail()` expecting a value for the customer's name, an email address and comments for the e-mail to be sent. In this particular example, the recipient and subject are hard coded as arguments in the `mail()` parameters, and only the passed `$comments` variable from the input is actually used. Figure 8-4 shows what the email recipient sees:

![Figure 8-4: Email received from PHP](image)

With a single address to send the email to (the business who's getting feedback) hard-coding an address is common. You can simultaneously send an email to different places using more than a single `mail()` function, and in those cases you'll often want to include an email address sent by the user. The `From:` header of the email is the default from the file on the PHP server (`php.ini` file.) That can be changed either by changing the `php.ini` file or adding a fourth parameter to the
**mail()** function. In the section *Adding Optional Headers and Multiple Mails* you will see how PHP can simultaneously send multiple emails and how to customize headers.

### Passing Data from Radio Buttons and Check boxes

Passing data from some kind of HTML5 text input field (e.g., text, email, url) requires the name of the input element and as soon as the submit button is clicked, the current value of the user input is sent to PHP where the actual value can be retrieved using the **$_POST[]** predefined variable. However, how do you retrieve values from radio buttons and check boxes? In this section, you will see how each is handled and how they might be used.

#### Radio Buttons: Mutually exclusive choice

Radio buttons in HTML5 are easy to work with and easy to use to pass data to PHP. Because radio buttons represent a mutually exclusive choice (any one choice negates other choices from the same button group) no matter how many buttons are used in a group, you only have to deal with a single value from that group. For example, if you create a set of radio buttons for a single breed of dog, you only get to select one breed. If you select a bulldog, you cannot select an Afghan hound as well. (A dog cannot be two breeds simultaneously—it must be a single or mixed breed. A cross between a bulldog and an Afghan hound is neither, but belongs to another category we call “mixed.”)

Within a form, a radio button input is identified by a name, just like text input, but you must supply a value. Several buttons can have the same name, and the one selected becomes the value that is passed to PHP. The following snippet illustrates how this works:

```html
<input type="radio" name ="pet" value="dog">
<input type="radio" name ="pet" value="cat">
<input type="radio" name ="pet" value="other">
```

Every radio button in a **form** container with the same **name** value is treated as a single attribute. On the PHP side, the “pet” attribute holds the value of the selected radio button. So, for example, if a user selects the radio button with the value “cat,” on the PHP side, you simply have to write,

```php
$_someName=$_POST['pet'];
```

The several radio elements named ‘pet’ pass only the value of the one that has been selected. In this case, the user selects the “cat” choice; so the value of **$_someName** is “cat.”

Listing 8-6 is the CSS file for Listing 8-7. Keep them together in the same folder.
Listing 8-6: radio.css
@charset "UTF-8";
/* CSS Document */

h2 {
  font-family: "Gill Sans", "Arial Black", Arial, sans-serif;
  text-align: center;
  color: #ffffff;
  background-color: #666666;
}

body {
  font-family: Verdana, Geneva, sans-serif;
  background-color: #CCCCCC;
  color: #881111;
  margin-left: 20px;
}

legend {
  color: #333333;
}

fieldset {
  display: table;
}

#colA {
  display: table-cell;
  width: 250px;
}

#colB {
  display: table-cell;
  width: 250px;
}

input {
  color: #881111;
}

In order to cut down on the code for any single file, any CSS that’s more than a few lines will be paced in a separate file. In this way, it’s easier to focus on the relevant code and have a decent looking UI. Figure 8-5 shows the data input in the Web page:
Listing 8-7 shows where both the text and email text input are placed in the same form. An important feature of the radio button input is to set the value attribute as shown in the listing.

Listing 8-7: radioEntry.html
<!DOCTYPE HTML>
<html>
<head>
<link rel="stylesheet" type="text/css" href="radio.css">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Radio Buttons</title>
</head>
<body>
<header>
<h2>Radio Mail</h2>
</header>
<h4>Fill in the forms and make your selections:</h4>
<form action="SetRadioData.php" method="post">
<input type="text" size=20 name="customer">
First name  
<br>
<input type="email" size=20 name="cMail">
Email address
<p>
<fieldset>
Once the data have been entered, the action in the form passes the data to the `<section id="colA">` **Gender**<br> `<input type="radio" name="gender" value="male">` &nbsp;Man <br> `<input type="radio" name="gender" value="female">` &nbsp;Woman </section>`<br> `<section id="colB">` **Pet**<br> `<input type="radio" name="pet" value="dog">` &nbsp;Dog <br> `<input type="radio" name="pet" value="cat">` &nbsp;Cat<br> `<input type="radio" name="pet" value="other">` &nbsp;Other<br> </section>`<br> `</fieldset>`<br> `<br>`<br> `<input type="submit" name="sender" value="Send">`<br> `</form>`<br> `</body>`<br> `</html>`

Once the data have been entered, the action in the form passes the data to the **SetRadioData** class. This class is exactly like the class that set the data for the text in mailer shown Listing 8-5. The only difference is the names of the attributes passed.

### Listing 8-8: SetRadioData.php
```php
<?php
include_once "RadioMailer.php";
class SetRadioData
{
    private $cuName;
    private $cuMail;
    private $cuGender;
    private $cuPet;

    public function __construct()
    {
        $this->setter();
        $mailNow=new RadioMailer();
        $mailNow->sendMail(
            $this->cuName,
            $this->cuMail,
            $this->cuGender,
            $this->cuPet);
    }

    private function setter()
```
The **RadioMailer** class called by the SetRadio class is very similar to the mailer shown in Listing 8-6 (SimpleMailer) but with different parameters.

Listing 8-9: RadioMailer.php

```php
<?php

class RadioMailer
{
    public function sendMail($cus, $cMail, $cGender, $cPet)
    {
        $message = "$cus is a $cGender who has a $cPet for a pet";
        //echo $message;
        mail("yourMail@yourHost.com", "Mail Test", $message);
    }

}?
```

In testing this and other mail projects, a good practice is to place an echo statement to test the formatting in the message you’re sending. While testing, comment out the `mail()` object, and when you’re ready, just comment out the echo statement as shown in Listing 8-9.

**Checkbox: More than one choice**

While radio buttons are easy to use and pass to a PHP class, check boxes are a bit more complicated. With a radio button group that share the same name, all you have to do is to pass the radio name attribute to a `$POST[]` element, and no matter how many buttons are in the group, it only passes a single value. Your design expects a single value with radio buttons.

With check boxes, you expect multiple selections. Further, each check box needs a different name because when you submit the form it needs a unique name for each of the `multiple` values that may be passed in a check box. However, with check boxes,
you also expect some of the boxes to be unchecked. So, you need to handle them as well.

**Using Unique Name Values with Check boxes**

Because check boxes are not given a single name for all boxes in a grouping, each check box must have a unique name. For example, the following snippet shows a simple naming practice so that check box entries can be uniquely passed to PHP:

```html
<input type="checkbox" name="html5" value="html5" />
<input type="checkbox" name="php" value="serverSide" />
<input type="checkbox" name="javascript" value="clientSide" />
```

As you can see, the name of the element and its value can be the same or different. The name is how the element is uniquely identified, and the value is the content you want to pass to the PHP script. So, whether you want element and value names the same or different depends on what you plan to use them for when passed to PHP. In Listing 8-11, you will see that element names and values are the same since there was no reason to make them different. Listing 8-10 is the CSS and Listing 8-11 the HTML5 file.

**Listing 8-10:checkbox.css**

```css
@charset "UTF-8";
/* CSS Document */

h2 {
    font-family:"Arial Black", Gadget, sans-serif;
    text-align:center;
    color:#693C12;
    background-color:#F2BF27;
}

body {
    font-family:Verdana, Geneva, sans-serif;
    background-color:#8DB99E;
    color:#A60303;
    margin-left:20px;
}

legend {
    color:#F2BF27;
    background-color:#898C20;
}

fieldset {
    display:table;
}

#colA { 
    display:table-cell;
    width:250px;
}

```
Place the CSS file (checkbox.css) in the same directory as the checkEntry.html file. Again, the styling has been separated from the HTML to reduce clutter. Figure 8-6 shows the check box data entry page with the CSS applied:

As noted, each check box has a unique name; so the two groupings of check boxes (Web development skills and Web design skills) cannot be set up with only two unique names for the checkbox elements. Instead, each element needs a unique name. Listing 8-11 shows how check boxes are placed into an HTML5 data-entry form:

Listing 8-11: checkEntry.html
<!DOCTYPE HTML>
<html>
<head>

#colB {
  padding-left: 20px;
  display: table-cell;
  width: 300px;
}
input {
  color: #A60303;
}

Figure 8-6: Data entry using checkbox elements
<form action="SetCheckData.php" method="post">
  &nbsp;First name <br>
  &nbsp;Email address <p>
  <fieldset>
    <legend>Checkbox Corral</legend>
    <section id="colA"> Which Web development skills do you have? (check all that apply)<br>
      &nbsp;HTML5<br>
      &nbsp;PHP<br>
      &nbsp;JavaScript<br>
      &nbsp;ActionScript 3.0<br>
    </section>
    <section id="colB"> Which Web design skills do you have? (check all that apply)<br>
      &nbsp;Graphic design <br>
      &nbsp;Digital photography <br>
      &nbsp;User interface design (HCI)<br>
      &nbsp;Information design<br>
    </section>
  </fieldset>
</form>
With all of those unique names for the data being sent, you will need a lot of variables to handle them. The class in Listing 8-12 declares private variables for all of the text input (2) and check boxes (8). Then, all of the passed data are added as arguments in the `CheckMail.sendMail()` method.

Note that the submit button at the end of Listing 8-11 (above), it has a name, “sender.” The purpose or giving it a name is so that it can be recognized in the `SetCheckData` class in Listing 8-12. The line,

```php
if(isset($_POST['sender']))…
```

looks to see whether the data really have been sent from the HTML5 data entry page. By examining the submit button, it can verify the sent data using the `$_POST` predefined variable. Because the `$_POST` variable needs a name in its brackets, the object it is checking needs a name. By providing the submit button with a name, we are able to ensure that the correct submit button has been clicked and the data sent is what we expect.

**Listing 8-12: SetCheckData.php**

```php
<?php
include_once "CheckMailer.php";
class SetCheckData
{
    private $cuName;
    private $cuMail;
    private $html5;
    private $php;
    private $javascript;
    private $actionscript3;
    private $java;
    private $photo;
    private $hci;
    private $info;

    public function __construct()
    {
        $this->setter();
        $mailNow=new CheckMailer();
        $mailNow->sendMail(
            $this->cuName,
            $this->cuMail,
            $this->html5,
```
$this->php,
$this->javascript,
$this->actionscript3,
$this->java,
$this->graphic,
$this->photo,
$this->hci,
$this->info);

private function setter()
{
    if(isset($_POST['sender']))
    {
        $this->cuName=$_POST['customer'];
        $this->cuMail=$_POST['cMail'];
        empty($_POST['html5']) ? $this->html5="NULL" : $this->html5=$_POST['html5'];
        empty($_POST['php']) ? $this->php="NULL" : $this->php=$_POST['php'];
        empty($_POST['javascript']) ? $this->javascript="NULL" : $this->javascript=$_POST['javascript'];
        empty($_POST['actionscript3']) ? $this->actionscript3="NULL" : $this->actionscript3=$_POST['actionscript3'];
        empty($_POST['java']) ? $this->java="NULL" : $this->java=$_POST['java'];
        empty($_POST['graphic']) ? $this->graphic="NULL" : $this->graphic=$_POST['graphic'];
        empty($_POST['photo']) ? $this->photo="NULL" : $this->photo=$_POST['photo'];
        empty($_POST['hci']) ? $this->hci="NULL" : $this->hci=$_POST['hci'];
        empty($_POST['info']) ? $this->info="NULL" : $this->info=$_POST['info'];
    }
}
The `empty()` function returns a Boolean. A `true` value indicates that the check box is unchecked—it’s empty. So, it sets a `NULL` value in the PHP variable. Otherwise, it assigns the value passed from the checkbox input element. Remember the above ternary conditional statement is the same as:

```php
If(empty($_POST['name']))
{
$this->variable="NULL";
}
else
{
$this->variable=$_POST['name'];
}
```

Ternary statements just take up less room and the code is more readable.

All of the processed data are now sent to the CheckMailer class. Listing 8-13 shows that the date are passed in a manner that is almost identical to the mailer class that sent the radio and text data.

**Listing 8-13: CheckMailer.php**

```php
<?php
class CheckMailer
{
    public function sendMail($cus,$cMail,$html5,$php,$javascript,$actionscript3,$java,$photo,$hci,$info)
    {
        $seperator="\n---------------------\n";
        $message="$cus whose email is $cMail the following development skills:";
        $message .="\n$html5 \n$php \n$javascript \n$actionscript3 \n$java\n";
        $message .=$seperator;
        $message .="$cus has these design skills:\n$photo\n$hci\n$info";
        //echo $message;
        mail("wdsanders@comcast.net", "Mail Test", $message);
    }
}
?>
```

Check
Using HTML arrays for check boxes
Because check boxes may be grouped where users may have more than a single choice for each grouping (e.g., hobbies, programming languages), you may want to use array names for each grouping. In HTML5, an array name looks like the following:

```html
<input type="checkbox" name="sports[]" value="baseball" ">
<input type="checkbox" name="sports[]" value="football" ">
<input type="checkbox" name="sports[]" value="cricket" ">
```

All three checkboxes have the same array name, `sports[]`, but each has a different value. If a user chooses two sports, such as baseball and cricket, PHP must have a way of differentiating those selected and those not selected.

As you can see, arrays are passed just like single variables. To find the selected check boxes when the array values are passed to PHP, your script can iterate through the array using a simple `foreach` loop:

```php
<?php
$sport=$_POST['sports'];
foreach ($sport as $sp)
{
    echo $sp."<br">;
}
?>
```

If you change the checkboxes in Listing 8-11 to arrays, as in the following snippet, you can see how to pass these values on to another object:

```html
<fieldset>
    <legend>Checkbox Corral</legend>
    <section id="colA">
        Which Web development skills do you have? (check all that apply)<br>
        <input type="checkbox" name="dev[]" value="html5" 
        &nbsp;HTML5<br>
        <input type="checkbox" name="dev[]" value="php" 
        &nbsp;PHP<br>
        <input type="checkbox" name="dev[]" value="javascript" 
        &nbsp;JavaScript<br>
        &nbsp;ActionScript 3.0<br>
        <input type="checkbox" name="dev[]" value="java" 
        &nbsp;Java </section>
```
<section id="colB"> Which Web design skills do you have? (check all that apply)<br>
<input type="checkbox" name ="design[]" value="graphic" >
&nbsp;Graphic design <br>
<input type="checkbox" name ="design[]" value="photo" >
&nbsp;Digital photography<br>
<input type="checkbox" name ="design[]" value="hci" >
&nbsp;User interface design (HCI)<br>
<input type="checkbox" name ="design[]" value="info" >
&nbsp;Information design<br>
</section>
</fieldset>

Now, there are only two named check boxes; dev[] (for development) and design[] (for design).

The two classes being used to get the data and to mail it are going to have to make fundamental changes to accept and process the check box arrays. For example, on the PHP side, the dev[] and design[] arrays only include elements of the choices that were checked. As a result, the array sizes will vary depending on how many boxes the user checks. As a result, the methods that require a list of parameters and process incoming data will have to be changed. Listing 8-14 shows the changes in processing the code. As you will see, you have fewer lines to code because instead of individual variables, the checkbox values have been placed in two arrays.

Listing 8-14: SetArrayCheckData.php

```php
<?php
include_once "CheckArrayMailer.php";
class SetArrayCheckData
{
    private $cuName;
    private $cuMail;
    private $dev=array();
    private $des=array();

    public function __construct()
    {
        $this->setter();
        $mailNow=new CheckArrayMailer();
        $mailNow->sendMail(
            $this->cuName,
            $this->cuMail,
            $this->dev,
            $this->des);
```
private function setter()
{
    if(isset($_POST['sender']))
    {
        $this->cuName=$_POST['customer'];
        $this->cuMail=$_POST['cMail'];
        $this->dev=$_POST['dev'];
        $this->des=$_POST['design'];
    }
}

$test=new SetArrayCheckData();
?>

Instead of having 9 separate variables (5 for development and 4 for design), only the two arrays are pulled from the $_POST[] variables. To pass these values to the CheckArrayMailer class using the sendMail() method, both the SetArrayCheckData class and the CheckArrayMailer classes must be changed. As you can see, the sendMail() method only sends four arguments. However, remember that you do not know how many elements are in the arrays.

Now, when we look at the mailer class, CheckArrayMailer in Listing 8-15, you can see that the two arrays are opened and processed in the same way as

Listing 8-15: CheckArrayMailer.php

<?php
class CheckArrayMailer
{
    private $development;
    private $design;

    public function sendMail($cus,$cMail,$dev,$des)
    {
        foreach($dev as $devSkill)
        {
            $this->development .= $devSkill ."\n";
        }

        foreach($des as $desSkill)
        {
            $this->design .= $desSkill ."\n";
        }

        $separator="\n---------------------\n";
    }
}
$message="\n$cus whose email is $cMail has the following development skills:\n$this->development";
$message .=$separator;
$message .="$cus has these design skills:\n$this->design";

$sendTo="yourcompany@domain.com";
//echo $message;
mail($sendTo, "Check Box Skills", $message);
echo "Mail Sent to: $sendTo";
}
}
?>

When the email is processed, only the categories that have been selected are included in the array. So, you need two loops to iterate through arrays; one each for the two arrays. Figure 8-7 shows what the user receives:

![Email received by user](image)

**Figure 8-7: Email received by user**

The check box data is aligned below the two different skill sets. However, unlike the check box data that were passed as individual variables, the null values do not appear. For this kind of application where the values are sent as part of an email message, the null values are unimportant. However, when we begin looking at data to store in a database, we’ll need a method to get information for each check box—including null data.
Adding Optional Headers and Multiple Mails

All of the mail() examples thus far have focused on passing the data between the input source (HTML5) and sending out an email to a single source using the bare minimum parameters—to-address, subject and message. In this section, you will learn how to add an additional argument to send mail that has a return address and to send multiple mailings in a single class. Using the CheckArrayMailer class (Listing 8-15) you will see first how to add a return address and then how to send out an automatic reply.

Adding Extra Headers

Up to this point, all that you have seen in the mail() function is the required default arguments required for sending mail—address, subject and content. To add a header for a return address, a copy (cc:) or hidden copy (bcc:), you may add a forth argument. Figure 8-8 provides a general overview of the mail() function parameters with the appropriate arguments:

Mail Requirements and Options

```php
$sentTo="someOne@here.com";
$subject="Header Parameters";
$content="Hello, how are you?";
$optional="From: Customer <cus@here.com>";

mail($sendTo,$subject,$content,$optional);
```

![Figure 8-8: Arguments in mail() function](image)

The optional fourth parameter is especially important for adding an automatic return email to the sender. If such a parameter does not exist, you cannot just click on the “reply” button to send a return reply. Of course this requires the email address of the person who uses the email form.

The optional format may include and of the following or a combination of them:
- 'From: Name/title <specific@sample.net>'
- 'Cc: Name/title <specific@sample.net>'
- 'Bcc: Name/title <specific@sample.net>'
- 'Reply-To: Name/title <specific@sample.net>'
• 'To: NameA <specificA@sample.net, NameB <specificB@sample.net >'
• 'X-Mailer: PHP/

Figure 8-8 above provides an example of using a single header. However, if you want to use multiple headers in the fourth parameter you need to concatenate a variable with the additional headers. The following format is typical:

```
$headers .= 'To: NameA <A@sample.net, NameB <B@sample.net >' . "\r\n";
$headers .= 'From: Name/title <specific@sample.net>' . "\r\n";
$headers .= 'Cc: Name/title <specific@sample.net>' . "\r\n";
$headers .= 'Bcc: Name/title <specific@sample.net>' . "\r\n";
$headers .= 'Reply-To: Name/title <specific@sample.net>' . "\r\n";
$headers .= 'Importance: High\n';
```

Then, all of the headers would be placed in the mail() function as the following:

```
mail($sendTo, $subject, $contents, $headers);
```

In this way, a single parameter can hold multiple features in an argument ($headers).

You may wonder why a “To:” element would be included in the optional headers parameter since the first parameter is supposed to handle the send to address. That’s because the mail() function’s first parameter expects a single email address for an argument. If you have multiple recipients, you can place one in the first parameter and the others in the header in the fourth optional parameter.

In some applications you may have need for a fifth parameter. Listed under additional_parameters, a fifth parameter can be especially useful for passing additional flags as command line options. However, you need to explore much deeper into working with PHP mail and system settings to effectively employ this parameter.

**Multiple Mails**

One of the easiest and most useful features of the PHP mail() function is to set it up with an auto-reply to the sender while simultaneously sending it to the contact source. Without changing anything in the class in Listing 8-14, and using the HTML input with the check boxes named as arrays, Listing 8-16 shows how multiple emails can be sent and the optional headers are incorporated in a mailer. The emails are configured as an automatic reply to a job application so that the business
Listing 8-16: CheckArrayMailer.php (updated version)

```php
<?php

class CheckArrayMailer
{
    private $development;
    private $design;
    private $message;
    private $autoReply;

    public function sendMail($cus,$cMail,$dev,$des)
    {
        foreach($dev as $devSkill)
        {
            $this->development .= "    ".
$devSkill ."\n";
        }
        foreach($des as $desSkill)
        {
            $this->design .= "    ". $desSkill ."\n";
        }
        $businessMail="sandApp@sandlight.com";
        $applicant="From: Sandlight Inquiry<$businessMail>\n";
        $subjectNow="Position application at Sandlight";
        $this->message= "\nApplicant name: $cus\n";
        $this->message .="Applicant email: $cMail\n";
        $this->message .="\nDevelopment skills: \n$development\n";
        $this->message .="Design skills: \n$design";
        $this->autoReply ="Dear $cus, \nThank you for expressing an interest in a position ";
        $this->autoReply .="at Sandlight Productions. We will be going over your ";
        $this->autoReply .="qualifications and evaluating them in terms of our requirements.";
        $this->autoReply .="If you have any questions or would like to provide more information ";
        $this->autoReply .="please don't hesitate to contact us.";
    }
}```
Two of the echo statements are commented out. They are used to debug the output prior to using the information in the `mail()` method. The first mail is the automatic reply to the person who sent it. The second mail is send to the business (or person) that hosts the page with the PHP contact form. Both include a fourth parameter that includes the sender’s return address in an email parameter. In this way the sender and host can simply use the reply button on their email client.

Depending on the users’ email client, the auto-reply will be displayed differently. Even the same email client with different configurations can affect the way an email appears. Figure 8-9 shows the auto-reply the sender gets.

![Auto-reply email in Web-based email client sent to applicant](image)

*Figure 8-9: Auto-reply email in Web-based email client sent to applicant*
Note that in Figure 8-9 the settings in the upper right corner show possible view alternatives that the user may have set differently. As a rule of thumb, keep your auto-responses as simple as possible without adding graphics or other non-text elements that may conflict with the email client’s settings.

Figure 8-10 shows the email that the host receives. It is focused on the key elements of the job requirements and lacks any formalities of salutations. Instead the contents focus on the precise information needed to pre-screen candidates and contact the applicant.

![Email sent to host from applicant](image)

In an actual job application screening, an employer would ask for more details (like the applicant’s last name and phone contact), but the check boxes help to focus the email on the information about a specified skill-set. Importantly, though, the same PHP email application can generate very different emails.

**Do-It-Yourself Exercises**

While this chapter has been considerably longer than other chapters in the book, the key elements boil down to two PHP features: processing data from an HTML form and using the `mail()` function. These exercises focus on the basics for each.

- Create an HTML5 Web page with a form that sends the following to a PHP program that prints the HTML5 form data to the screen using the `echo` statement:
  - One text input field
- Two radio buttons with different values but the same name
- Two check boxes each with a different name and value

- Create a dual set of check boxes representing two modes of transportation (air travel and train travel) with different selections for each. Name the check box groups with two different array names.
- Make a simple order confirmation HTML5 form that sends the user an auto-response listing all of the products or services they have ordered and the mailing options:
  - Input put forms include
    - Text input for user name and email address using HTML5 input
    - Check boxes for items/services to select
    - Checkbox attributes include a price
  - PHP program calculate shipping costs and sales tax. (You can make up the shipping costs and use 6% tax) and include in auto-reply for total.
Chapter 9: The Client—Request Model

At the heart of OOP is some system of communication. The simplest way to think about communication between objects is a request-fulfill model. A client makes a request to an object to get something. The request can originate in the user UI, and it is passed to a client who finds the correct class and method to fulfill the request. Figure 9-1 shows a simple diagram with an overview of this model:

**PHP**

**Client—Request Model**

<table>
<thead>
<tr>
<th>HTML</th>
<th>Client.php</th>
<th>Worker.php</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;form action&gt;</code></td>
<td><code>class Client</code></td>
<td><code>class Worker</code></td>
</tr>
<tr>
<td><code>&lt;form request&gt;</code></td>
<td><code>{ new Worker();</code></td>
<td><code>{ request1();</code></td>
</tr>
<tr>
<td><code>&lt;form submit&gt;</code></td>
<td><code>worker-&gt;req();</code></td>
<td><code>request2();</code></td>
</tr>
<tr>
<td></td>
<td><code>new Client();</code></td>
<td><code>return $req;</code></td>
</tr>
</tbody>
</table>

*Request originates in HTML Document*  

*Client receives request from HTML and chooses appropriate class and method to fulfill request.*  

*Request fulfilled and returned.*

**Figure 9-1: Client Request Model**

Starting with a simple example, you can see how this model helps to clarify the OOP process and the role of HTML. It also simplifies setting up your program. Keep in mind that other, simpler, methods are available to arrive to generate the same results, but as your projects become more complex, you can use this same model to help simplify your tasks.

**Start With What you Want**

Often developers start of with questions like, “What should my UI look like? Or should I use Plan A or Plan B? Actually, you need to start with the goal. What will all of this coding result in that you need. So, we start off with the and we must ask ourselves,

*What do I want?*
In other words, rather than just tossing together some HTML with a form and some PHP to work with superglobals sent from the HTML UI, we need to do a little planning.

**What Object will Get Me What I want?**

Let's say I want to get the square and square root of a number returned. That’s easy enough with PHP using the `sqrt()` function. The following function finds and returns the square root of 10.

```php
public function doSquareRoot()
{
    $someValue=10;
    $answer = sqrt($someValue);
    return $answer;
}
```

Now, thinking about where the request originates in an HTML form, adding the appropriate superglobal provides a far more flexible class method:

```php
public function doSquareRoot()
{
    $someValue =$_POST['num'];
    $answer = sqrt($someValue);
    return $answer;
}
```

Thinking further, this function should be a method in a class that does certain math operations. So, now, we’ll embed it in a PHP class.

```php
<?php
class SquareWork
{
    private $valNow;
    private $squareRoot;

    public function doSquareRoot()
    {
        $this->valNow=$_POST['num'];
        $this->squareRoot = sqrt($this->valNow);
        return $this->squareRoot;
    }
}
?>
```
Notice that the class includes two private variables $valNow and $squareRoot. Why is that? You may want to use the square root value to do something else in the class and want your variables encapsulated as private variables.

Now that there’s a way to calculate the square root, the remaining task requires another method to calculate the square. That’s easy enough using the built-in pow() function, and so the final class deals with both requests for square roots and squares of values as shown in Listing 9-1:

Listing 9-1: SquareWork.php

```php
<?php
//SquareWork.php
class SquareWork
{
    private $valNow;
    private $squareRoot;
    private $square;

    public function doSquare()
    {
        $this->valNow=$_POST['num'];
        $this->square=pow($this->valNow, 2);
        return $this->square;
    }

    public function doSquareRoot()
    {
        $this->valNow=$_POST['num'];
        $this->squareRoot = sqrt($this->valNow);
        return $this->squareRoot;
    }
}
?>
```

Now, you have a class that returns both square roots and squares. In order for a user to interact with that class, you need a UI for entering values.

**HTML5 User Interface (UI)**

Throughout the book, you will see a reference to HTML, and in all cases, I refer to HTML5. The “5” is added every now and again just to remind you of that fact. For now, the HTML component is simply a way to allow the user to interact with the PHP that does all of the real work. This is not to say that the UI is unimportant but rather to point out that it is important only insofar as it can send and receive data for processing in PHP.
To get started, consider what the UI must send to PHP. Most obviously, it must send a value to be processed. However, since we’re dealing with more than a single process, we need a way to indicate what process we want—one that returns the square root or the square of a value. We know that the process must be through an HTML form and that form can pass information to PHP. So let’s consider the elements we need to pass on to a PHP client:

- Value of a number (A decimal or integer—does not matter)
- Name of the class to do the processing (SquareWork)
- Name of the method for calculating squares (doSquare())
- Name of the method for calculating square roots (doSquareRoot())

The value of the number needs to be part of some kind of user input (text input) and so too do the calculations—a choice of square or square root (radio buttons since they’re mutually exclusive). Only a single class is being used, so that can be placed in a hidden form and passed on to the Client class as shown in Listing 9-2:

**Listing 9-2: MathWiz.html**

```html
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet" type="text/css" href="request.css">
  <title>Request</title>
</head>

<body>
  <h3>MathWiz:<br /> The UI Class & Method Requester</h3>
  <form name='require' action='Client.php' method='post' target='feedback'>
    <!--Place SquareWork class name in hidden form -->
    <input type='hidden' name='class' value='SquareWork'>&nbsp;SquareWork Class<br />
    <input type='text' name='num' size='6'>&nbsp;Enter value <br />
    <fieldset>
      <legend>Methods</legend>
      <!-- Mutually exclusive choice: either return square or square root -->
      <!-- doSquare is the name of squaring method -->
      <input type='radio' name='method' value='doSquare'>&nbsp;Square the value<br />
      <!-- doSquareRoot is the name of square root method -->
      <input type='radio' name='method' value='doSquareRoot'>&nbsp;Square the value<br />
    </fieldset>
  </form>
</body>
```

While the **Client** class is launched by the action assignment in the `<form>` tag, all superglobal values in the form elements can be accessed from *any* of the PHP classes. So while the ‘num’ input text field is available to the **Client** class as a superglobal ($_POST['num']), the **Client** has no need to use it and does not. However, the superglobal generated from the hidden form element named ‘class’ and the radio button group, ‘method’ have values that the Client will need to make the request to the **SquareWork** class. You will see that use in the Client listing.

**Simple Style**
The link to the CSS style sheet and the fieldset make it the UI a little easier to work with, but they’re mainly design elements. Listing 9-3 shows the style sheet.

*Listing 9-3: request.css*

```css
@charset "UTF-8";
/* CSS Document */
/*595241(dark mud) B8AE9C (dove gray), FFFFFF (white), A3C1AD (Cambridge blue), 8A0917 (dark firebrick) */
body
{
    background-color:#FFF;
    color:#8A0917;
}
h3
{
    color:#595241;
    font-family:Segoe, "Segoe UI", "DejaVu Sans", "Trebuchet MS", Verdana, sans-serif;
}
iframe
{
    background-color: #A3C1AD;
}
```
The commented out list of values at the top is a color combination employed. Generally, in setting up a UI, you want to use colors that go together to provide a better user experience (UX).

**What About the `<iframe>`?**

Some developers still think that using `<iframe>` tags is a cardinal sin. *That was before HTML5.* Honest, in HTML5 the problems (real and imagined) with iframes were addressed. They are a simple and direct way of dealing with feedback from PHP requests without having to resort to Ajax or some convolution with JavaScript. They’re simple, they work, and they don’t cause problems. HTML5 has been out way too long for developers to still be fretting over iframes.

**The Client Makes a Request**

All links in this model from HTML go to the PHP class, *Client*. Those with experience in other languages may wonder why name a class “Client” instead of the more commonly used name, “Main.” First of all, “Main” is used out of convention and habit and does not signify any kind of other naming necessity. (Some languages have incorporated a *main* object as a necessary component.) Second, in the definitive book on computer programming design patterns, *Design Patterns: Elements of Reusable Object-Oriented Software* by Gamma, Helm, Johnson and Vlissides, the authors use the term “Client” in identifying participants in a pattern that make requests from the pattern. Finally, *Client* identifies the object that is a *request-maker*, and that gives a better sense of what clients do than the term *Main*. Besides in an object-oriented environment, the *main* aspect of the program is not in the *client* but in the interacting objects—*the classes that make up the program*.

**Registering Autoload**

The Client class is a good place to add an *autoload* function so that you don’t have to keep adding class *include* statements to insure that a class in an external file will be loaded. As programs grow in size and number, more classes that use other classes in external files are likely to be part of a new program. That is, a class from a previous program might be used in a new program. In that previously used class file, some kind of include statement may still be part of that file. By using the `spl_autoload_register()` function, you will not get an error if one of the classes used in the program already has an include statement (e.g., `include_once 'xyz.php'`). Besides, PHP is considering deprecating of the `__autoload()`
function and so it is a function that may not be recognized in the future versions of PHP and throw an error if a class used in the program has an include statement.

From UI to Client Requests
Using class and method names in the form values has simplified the process of converting UI input into specific requests. First, the class name from the hidden form is passed to the $reqClass property (a private variable.) Listing 9-4 shows the Client class “capturing” the names sent from the HTML form:

Listing 9-4: Client.php

```php
<?php
error_reporting(E_ALL | E_STRICT);
ini_set("display_errors", 1);
// Autoload given function name.
function includeAll($className)
{
    include_once($className . '.php');
}
// Register
spl_autoload_register('includeAll');

class Client
{
    private $reqClass;
    private $reqMethod;
    // client request
    public function request()
    {
        // Class name from hidden form
        $this->reqClass=$_POST['class'];

        // Method name from radio forms
        $this->reqMethod=$_POST['method'];

        // Instantiate new class instance (object)
        $classNow=new $this->reqClass();

        // Use object method
        echo $classNow->{$this->reqMethod}();
    }
}
$worker=new Client;
$worker->request();
?>
```
When instantiating a class object, you can use the name passed directly from the superglobal (a string) to a variable, and you can do the same with a method name in a string. However, with the method name you need to place a set of curly braces around the variable with the method name and put the parentheses outside of the curly braces as used in Listing 9-4. Figure 9-2 shows a detailed diagram of how to create class instances and methods from strings:

**Class from string:** (In private variable $reqClass)

```
$className = new $this->reqClass();
```

**Method from string:** (In private variable $reqMethod)

```
$className->{$this->reqMethod}();
```

By using class and method names in HTML form values, you are able to better link the HTML UI to the PHP for processing requests.

**The Client—Server Side Link**

One of the constant processes that occur when using PHP is that between the server side processing of requests and the I/O of client side request entry and display. All PHP processing returns code that can be interpreted by the browser on the client’s computer. Making the link between client and server side domains is eased and more easily understood in the Client-Request model. The closer a client-side document (an HTML5 document) is to a server-side class the smoother the transition in terms of writing code in an OOP environment. In looking at the output shown in Figure 9-3, you can understand the HTML—OOP connection. We might say that the HTML document *speaks* to the OOP-based PHP.
From a development point of view, the class name is hidden in a form, but the “SquareWork Class” tells the name of the class summoned by the request. The two methods (highlighted in a fieldset with the legend, “Methods”) describe what each method will do for the two different radio buttons. The submit button is labeled, “Make Request” to emphasize the fact that the request is forwarded to a PHP client class (named Client). Finally, the iframe with a Cambridge blue background shows the return value from PHP.

**Do-It-Yourself Exercises**

This chapter brings focus to a simple OOP model: Client—Request. The following exercises provide some practice on implementing the model.

- Make a *Client—Request* program that takes a text input and returns results that return the text as all lowercase or all uppercase letters.
  - A Single class
  - Two methods in the class
  - The names of the class and two methods are passed from HTML as superglobals
  - Text input is
  - The results are returned in an iframe
Tip: Use `strtoupper()` and `strtolower()` built-in PHP functions in your methods.

• Create a Client—Request program using two classes and each class has two methods. One class handles text and the other handles financial calculations. Your HTML UI needs a way to choose the methods. One set of methods will choose a type of car, and another set will calculate price depending on you want a new one or used one of the same make and model. Figure 9-4 shows the HTML UI, and see if you can create the program. The used price will always be 65% of the new one for both types of cars. (You can used any pricing you want.)

![Figure 9-4: HTML UI for dual classes and methods](image)
Chapter 10: Web Pages as Objects & PHP

File Work

The entire notion of “dynamic” in discussing Web pages has a range of meanings. Here, *dynamic* refers to using PHP to change the content of Web pages without having to rewrite the code for the page. For example, to change the video displayed on a Web page may involve nothing more than changing the URL link to *YouTube*. A programmer can quickly go into the page, change the literal value for the link and overwrite the old HTML file. A much easier way would be to use an administrative tool that changes the content in a database. All you’d have to do is to enter the administrative module, enter the new URL to change the contents of the table with the field where the HTML file gets its data.

**A PHP HTML Wrapper**

You can think of an encapsulated HTML document as one that is “wrapped” in a PHP shell. The PHP shell can be a simple container or a more robust PHP class; but to encapsulate HTML, *the PHP shell must be formatted as a class*. This part is very simple. The following listing shows the PHP shell or *wrapper*.

```php
<?php
class RequestUI
{
    private static $ui;

    static function request()
    {
        //Heredoc wrapper
        self::$ui=<<<UI
            //UI Wrapper
            UI;
        echo self::$ui;
    }
}
RequestUI::request();
?>
```

As you can see, the *RequestUI* class contains a static method (*request()*)) so that the object can be instantiated and used in a single line, *RequestUI::request()*). The static method simply makes it a bit easier to use the method, and if you prefer using non-static methods and properties, you can. It just takes two lines of code instead of one for self-launching the class.

So, what HTML can you place in the UI Wrapper? You can place any HTML document in the wrapper, and it works just like a regular HTML document outside the wrapper. The advantage is that you can now use private/protected methods from
within the class or inherited from another class, including abstract class and interfaces. (With interfaces, you can use constants with set values.)

A closer look at the HEREDOC string formatted HTML page
All HEREDOC strings have the following format:

```php
<<<DOCNAME
//Any content
//including HTML String
//To close a HEREDOC string
//The string name must go flush
//left-no spaces and terminate
//with a semi-colon (;)
DOCNAME;
```

All of the HTML should be well formed HTML5 (preferably), but any HTML document can be placed inside the HEREDOC wrapper.

Encapsulating a Web page in a PHP object
When part of a PHP class, the HTML document has all of the advantages of a PHP class, including private and protected methods that disallow unwanted data. You can think of a PHP class containing HTML as a PHP UI closely associated with a PHP client.

```php
<?php
class RequestUI {
    private static $ui;

    static function request() {
        //Heredoc wrapper
        self::$ui=<<<UI
        <!DOCTYPE html>
        <html>
        <head>
        <meta charset="UTF-8">
        <title>User Interface</title>
        <link rel="stylesheet" href="basicoop.css">
        </head>
        <body>
        <h3>What Request Do You Have?</h3>
        <form name="classNow" action="Client.php" method="post" target ="feedback">
        <input type="radio" name="request" value="Calculate">
        </form>
        UI
        </body>
    }
}
```
Multiply two numbers:<p>
<input type="text" name="first" size="6" value="1">

First number<br />
<input type="text" name="second" size="6" value="1">

Second number<p>
<input type="radio" name="request" value="Display">

Display a story<p>
<textarea name="story" cols="33" rows="5">Tell story: </textarea>
<input type="submit" value="Send Request"> <p>
<iframe name="feedback" width="250" height="250" sandbox="allow-same-origin">Feedback</iframe>
</body>
</html>

UI;
  echo self::$ui;
}
}
RequestUI::request();
?>

The easiest and best way to work with an object wrapper with HTML is to create and debug the HTML page first, and then just drop it in the HEREDOC wrapper. Of course if you’re creating a series of HTML pages, you don’t want to wrap them all, but only those pages used to make requests to a PHP page.

**Requesting a Class and a Method with HTML Form Data**

In making requests in an OOP environment through an HTML UI and PHP client, you can use form data to ask for specific classes and methods. Radio buttons are especially useful because they can pass the exact name of both the class and method to the client class to meet the user’s request without the user having to enter the class name or the method name.

To request a specific class, the following shows the process:

1. *From UI* (Encapsulated HTML in PHP class):
   
   `<input type="radio" name="request" value="Calculate">`

2. *To Client*:

   ```
   $classNow=$_POST['request'];
   $oop=new $classNow();
   ```
$oop->doTask();

The HTML form uses the name “request” to pass the value “Calculate.” Calculate is passed as a string, but PHP can use a string in declaring a class and create an object of that class. So, the line,

$oop=new $classNow();

instantiates an instance of the class Calculate.

In cases where the requests are to classes that all implement a common interface, you can simply use the actual name of the method, knowing that it will be differently and appropriately implemented depending on the class requested. So, the line,

$oop->doTask();

calls the method common to all of the classes that implement the same interface, assuming it only uses a single method.

**Calling different Methods though HTML**

While the problem of calling methods specifically related to a common interface might be solved by the simple expedient of calling the same method. However, when you want one specific class with different methods or different classes, each with a different method name, you need a different strategy.

**Using an Object String**

The easiest way to create a method reference from a string is to place the string with the name of the method in “object braces" (a pair of curly braces) and add a pair of parentheses in the form:

```
$classObject->{$method}();
```

For example, if you have a class named RequestedClass and a method named requestMethod, you would be able to call the method using the following sequence:

```php
$someMethod = "requestMethod"; //String with method name
$classNow = new RequestedClass();
$classNow->{$someMethod}();
```

The following example shows how a class with two methods can be accessed from an embedded HTML request through a client class. The names of both the class and method are passed directly from the values stored in radio button tags.

The HTML has been encapsulated in a PHP wrapper class:

```php
<?php
```
class RequestUI
{
    private static $ui;

    static function request()
    {
        //Heredoc wrapper
        self::$ui=<<<UI
<!DOCTYPE html>
<html>
<head>
    <link rel="stylesheet" type="text/css" href="request.css">
    <title>Request</title>
</head>
<body>
<h3>Mathster Mind:<br /> The UI Class & Method Requester</h3>
<form name='require' action='Client.php' method='post' target='feedback'>
    <input type='radio' name='class' value='MathsterMind' checked='checked'>&nbsp;MathsterMind<br />
    <input type='text' name='num' size='6'>&nbsp;Enter value<br />
    <fieldset>
        <legend>Methods</legend>
        <input type='radio' name='method' value='doSquare'>&nbsp;Square the value<br />
        <input type='radio' name='method' value='doSquareRoot'>&nbsp;Find the square root of the value<br />
    </fieldset><br />
    <input type='submit' name='send' value='Make Request'>
</form>
<iframe name='feedback'>Feedback</iframe>
</body>
</html>
UI;
    echo self::$ui;
}

RequestUI::request();
?>
The Client class processes the superglobals with the names of the class and methods. Since the requests are made using radio button forms, the class and methods names are immutable:

```php
<?php
error_reporting(E_ALL | E_STRICT);
ini_set("display_errors", 1);
function __autoload($class_name)
{
    include $class_name . '.php';
}
class Client
{
    private static $reqClass;
    private static $reqMethod;
    //client request
    public static function request()
    {
        self::$reqClass=$_POST['class'];
        self::$reqMethod=$_POST['method'];
        $classNow=new self::$reqClass();
        echo $classNow->{self::$reqMethod}();
    }
}
Client::request();
?>
```

The class that returns the requested actions, MathsterMind, simply provides the results from one of two methods: one that squares a value and one that finds the square root. Both methods use simple lambda calculus functions borrowed from functional programming.

```php
<?php
class MathsterMind
{
    private $valNow;
    public function doSquare()
    {
        $this->valNow=$_POST['num'];
        $lambda=function($num)
        {
            return $num * $num;
        };
        return $lambda($this->valNow);
    }
    public function doSquareRoot()
```
158

Notice that the numeric value is taken directly from a superglobal and placed in an encapsulated variable (private variable).

Using call_user_func()

PHP has a handy built-in method for making a call that can also take string values and turn them into classes and methods, call_user_func(). This function can accept class and method named strings, the first being a class name (as a string) and the second being a method name (also as a string). For making a calls to

```php
$lambda = function($num) {
    return sqrt($this->valNow);
};
return $lambda($this->valNow);
```

By changing the Client slightly, you can see how the same program used above in the subsection, Using an Object String, you can achieve the same results with call_user_func(). The following listing shows this different implementation of the client class, renamed, Client2.

```php
<?php
error_reporting(E_ALL | E_STRICT);
ini_set("display_errors", 1);
function __autoload($class_name) {
    include $class_name . '.php';
}
class Client2 {
    private static $reqClass;
    private static $reqMethod;
    private static $requestObject;
    //client request
    public static function request() {
        self::$reqClass = $_POST['class'];
        self::$reqMethod = $_POST['method'];
        self::$requestObject = new self::$reqClass();
        
```
echo call_user_func(array(self::$requestObject, 
    self::$reqMethod));
}
}
Client2::request();
?>

Whether you use one or the other is up to you. The first method, an object string, is both simpler and a bit faster than the second, using the built-in `call_user_func()`. In either case, Figure 10-1 shows what you will see on the screen:

![Mathster Mind: The UI Class & Method Requester](image)

*Figure 10-1: UI with selection of class and method*

The UI illustrates seeing the class and methods, but obviously, your programs will not display either classes or methods as such. Instead a UI displays just what the user wants to see and use. In this very simple case, the user wants to quickly find the square or square root of a value. This particular display, uses the following CSS file:

```css
@charset "UTF-8";
/* CSS Document */
/*595241 (dark mud) B8AE9C (dove gray), FFF (white), ACCFCC (robin's egg blue), 8A0917 (dark firebrick) */
body {
    background-color:#FFF;
```
Of course you can use any CSS you want. This one is selected to highlight the selection of a class (automatic) and one of two methods.

**Files: Non-MySQL**

- Text
- XML

**CSV**

**JSON**

Json stores arrays *only*. Scalar data can be stored as array elements, but to use Json, think “arrays.” In working with Json, you have four key methods:

1. `json_encode (array, [options])`: Numeric or associative array
2. `json_decode(jsonData, [options])`: Encoded jsonData
3. `file_put_contents(jFileName, jData)`: File name to store, encoded Json data
4. `file_get_contents(jsonFile, [options])`: This is the data retrieved to be decoded and used as persistent data.

To see how these can work, the following little program has a client with three methods: a `request()` to call either a Json write or read method; a `writeJ()` method and a `readJ()` method.

The `writeJ()` method creates an associative array and sends it as an argument in a call to the `WriteJson` class. The `readJ()` method calls the `ReadJson` class and expects a return with the original array containing the data stored there.

```php
<?php
//Client.php
error_reporting(E_ALL | E_STRICT);
in_set("display_errors", 1);
// Autoload given function name.
function includeAll($className)
{
    include_once($className . '.php');
}
```
spl_autoload_register('includeAll');

class Client
{
    //client request
    public function request()
    {
        $this->writeJ();
        $this->readJ();
    }

    private function writeJ()
    {
        $pageTags=array('header' => 'PHP Combines!',
            'subhead' => 'Json Stores',
            'body' => 'Instead of storing in text files use Json.');
        $writePage=new WriteJson();
        echo $writePage->doJson($pageTags);
    }

    private function readJ()
    {
        $readPage=new ReadJson();
        $jCatch=$readPage->getJson();
        echo $jCatch['header'] . "<br />";
        echo $jCatch['subhead'] . "<br />";
        echo $jCatch['body'] . "<br />";
    }
}

$worker=new Client();
$worker->request();

Before you can read a Json file, you need something in the file to read. Further, before you run this, be sure that the directory that you run it in has first been given full read/write permissions. A setting of 777 does the trick, but for some that may seem to be a little insecure; however, whatever permissions you provide, you need to provide enough for write to disk privileges to be invoked.

<?php
//WriteJson.php
class WriteJson
{
    private $jcodedData, $jFile;
    public function doJson($data)
    {
$this->jFile="jcode.json";
$this->jcodedData = json_encode($data, JSON_PRETTY_PRINT);
try {
    file_put_contents($this->jFile, $this->jcodedData);
} catch (Exception $e) {
    echo 'Problem : ' , $e->getMessage();
} return "success";
?>

Finally, you need an object to read the contents of the Json file. Remember, the decoded stored materials are an associative array. (See the Client.php above).

<?php
//ReadJson.php
class ReadJson
{
    private $jdecodedData, $jFile;
    public function getJson()
    {
        $this->jFile="jcode.json";
        try {
            $this->jdecodedData=json_decode(file_get_contents($this->jFile),true);
        } catch (Exception $e) {
            echo 'Problem : ' , $e->getMessage();
        } return $this->jdecodedData;
    }
}
?>

Once you have the values returned from the Json stored file, you can then use them just like any other contained values.
Do-It-Yourself Exercises

This chapter has three important points. First, you can encapsulate all HTML so that it can be treated as another class with all of the OOP advantages of a class. Second, you can make requests for a specific class and method through the encapsulated HTML via a client. Doing so better integrates the UI with the rest of the program. Finally, you can use different kinds of non-SQL options in storing data. The important feature of each is to load it into a page to be used dynamically.

• Create an HTML5 Web page with a form that has user input through a text field. (Make it simple with a single text field.)
  o Encapsulate the HTML document in a PHP wrapper using a HEREDOC string.
  o Create a client class that calls another class that places the text form value on the screen.
  o Use an iframe to display feedback.
• Create a PHP-encapsulated HTML5 UI that has options for more than one class and more than a single method for each class.
  o Create a Client class that uses the string values in the superglobals to launch the user-selected class and method.
Chapter 11: Communicating with Databases

To work with databases and data stored in tables residing in databases PHP uses Structured Query Language (SQL). The SQL commands place data into the tables, retrieves it and changes it. Then PHP takes the information via SQL and communicates with HTML. This chapter explores the MySQL database management system—a freely available database that will run on several different servers.

Setting Up MySQL
Before going on further, you will need to set up a MySQL relational database management system (RDBMS) or obtain access to one. The easiest way is to sign up for a hosting service that includes MySQL. The advantage of having a hosted service is that you can actually create PHP/MySQL applications that you can use worldwide. The basic services are inexpensive. For example JTLnet (www.jtl.net) accounts begin at $7.50/month but you will find hosting services with lower and higher prices. The quality and depth of services depend on how much you want to spend and how reliable the service is. However, even with a basic account like JTLNet’s, you’ll have everything you need including a PHP account and administrative tools.

A second alternative is to install a package on your computer that includes MySQL. Appendix A explains everything about turning your computer into a local hosting service. What’s more everything you need is free. This is quite handy for learning how to work with MySQL because everything you need is at hand and you don’t even have to connect to the Internet, but sooner or later you should look for a hosting service.

Whatever choice you make you must have a working MySQL account available to you before proceeding. The documentation for using your MySQL account should be available at your hosting service, at http://dev.mysql.com/doc/ or with the materials that you downloaded to set up your own locally hosted account.

Connection to a Database
PHP provides the MySQLi class to connect to a MySQL RDBMS. To use this class, you simply declare an instance of it with four parameters:

- Server name (e.g., myserver.here.com, localhost, 127.0.0.1)
- User name (e.g., jack, jill, underLordOfDarkness)
- Password (e.g., secret, rumplestillskin)
• Database name (e.g., myDB)

For example, the following snippet shows an object-oriented technique for connecting to a database named “billzDB:

```php
$server="localhost";
$user="bill";
$pass="billzsecret";
$currentDB="billzDB";
$hookup=new mysqli($server, $user, $pass, $currentDB);
```

Once you have the connection to the database, you're all set. Typically, in addition to connecting to MySQL, you also want to check to see if you have any problems and to let the user know that he/she is connected. Listings 11-1 to 11-3 show two reusable connection objects and a little test class. When learning how to use PHP with MySQL, using the mysqli error checking saves a great deal of time. In working with these files, but sure to substitute the HOST, UNAME, PW and DBNAME constant values with your own.

**Listing 11-1: IConnectInfo.php**

```php
<?php
//Filename: IConnectInfo.php
interface IConnectInfo
{
    const HOST ="userHost";
    const UNAME ="userName";
    const PW ="passWord";
    const DBNAME = "dataBaseName";

    public static function doConnect();
}
?>
```

The UniversalConnect class uses static properties and method. This allows you to use the returned value of the **doConnect()** method without having to instantiate a UniversalConnect object.

**Listing 11-2: UniversalConnect.php**

```php
<?php
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('IConnectInfo.php');

class UniversalConnect implements IConnectInfo
{
    private static $server=IConnectInfo::HOST;
    private static $currentDB= IConnectInfo::DBNAME;
```
private static $user = IConnectInfo::UNAME;
private static $pass = IConnectInfo::PW;
private static $hookup;

public static function doConnect()
{
    self::$hookup = mysqli_connect(self::$server,
        self::$user, self::$pass, self::$currentDB);
    try
    {
        self::$hookup;
        //Comment out / Uncomment following line for 
        //develop/debug
        echo "Successful MySql connection:<br />
    }
    catch (Exception $e)
    {
        echo "There is a problem: " . $e->getMessage();
        exit();
    }
    return self::$hookup;
}
?>

If the connection fails, you want to know why, and the catch function has a built-in Exception type that you can access through the exception variable $e. (You can use any variable name you want, but most developers just use $e.) For example, suppose you accidentally put in the wrong password. Instead of showing the success message, you’d get something like the following:

(Output)
Warning: mysqli::mysqli(): (28000/1045): Access denied for user
'bill'@'localhost' (using password: YES) in
/Library/WebServer/Documents/php/OOPphp/Ch11/connectOOP.php on line 10
Your error number is: 1045

Access denied for user 'bill'@'localhost' (using password: YES)

With an incorrect password, the error number is 1045, and while the error code is ambiguous you get a clue but not much more. For example, the MySQL documentation at http://dev.mysql.com/doc/refman/5.5/en/error-messages-server.html shows the following information for Error 1045:

Error: 1045 SQLSTATE: 28000 (ER_ACCESS_DENIED_ERROR)
Message: Access denied for user '%s'@'%s' (using password: %s)

That’s not much more information than that provided by the `mysqli_connect_error()` function. To get used to the typical kinds of errors you can expect, try intentionally entering incorrect connection information. For example, by removing the last letter of the database name results in the following error message:

(Output)

Warning: mysqli::mysqli(): (42000/1049): Unknown database 'sqlwor' in /Library/WebServer/Documents/php/OOPphp/Ch11/connectOOP.php on line 10

Your error number is: 1049

Unknown database 'sqlwor'

That error message is much clearer because it identifies the error specifically to be an unknown database. The MySQL error number 1049 shows,

Error: 1049 SQLSTATE: 42000 (ER_BAD_DB_ERROR)
Message: Unknown database '%s'

In comparing the error messages generated by the PHP `mysqli` functions and those listed by number in the MySQL documentation, the `mysqli` description provides at least as much information if not more. In any event look at error messages as learning tools.

If you are familiar with older versions of PHP, you may have worked with the `mysql_connect` function instead of the `MySQLi` class and the `mysqli()` function to connect to the MySQL server. The “i” stands for “improved,” and it is far more amenable to OOP programming. So, any function beginning with `mysql_` instead of the improved `mysqli_` are being treated as deprecated and not used for the purposes of this book.

Finally, you need a simple test to see if the connection works. Listing 11-3 provides a simple test. Note that the `UniversalConnect` class is not instantiated, but because of the global nature of static properties and methods, the connection is made through a direct reference to the `doConnect()` method.

**Listing 11-3: ConnectTest.php**

```php
<?php
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('UniversalConnect.php');
class ConnectTest
{
    private $test;
    public function __construct()
```
$worker=new ConnectTest();

If you see your success message, you know that all of the connection information you put into the interface is correct.

**Structured Query Language (SQL)**

Up to this point in the book, the code use has been limited to HTML5, PHP and some CSS. Now, you will be introduced to yet another language—Structured Query Language (SQL). Lest this news dismay you because it is yet another language to master, be advised that it’s quite easy and is integrated with PHP. Also, we’re only going to examine a small part of the MySQL version of SQL. MySQL has eight different types of SQL statements, and we are only going to look at key terms used for data definition and data manipulation. This will not delve into the database management aspect of SQL such as adding new accounts.

Besides MySQL you should know that several versions of SQL are available for different database management systems, such as Oracle, Microsoft/Sybase and PostgreSQL. In general, the syntax and data types of these different versions of SQL are very similar, and if you know SQL for one RDBMS, you’ll know most of the terms and how to use them for MySQL. However, in all versions you will find certain features that differ from the others. This discussion is solely on SQL for MySQL, and you may find differences between it and other SQL versions you may know.

Finally, one convention I’d like to use is to put all SQL statements in ALL CAPS. In this way, you can separate SQL statements, keywords and clauses from the PHP variables and literals. Keep in mind that constants are also in ALL CAPS; but in this chapter no constants are employed; so this should not be a problem. In later chapters both constants and SQL statements are employed, but by then you should be able to differentiate between the two.
MySQL Data Types

Most of the work done with SQL is moving, retrieving, changing and deleting data stored in a table. While PHP is very flexible with data types assigned to variables and arrays, MySQL is very specific. You can take data stored in a variable and pass it to a table for storage it must be compatible with the data types defined for the table. For example, a PHP variable can be assigned and odd integer value. If an odd number is divided by two, the result is no longer an integer because it will now have a fraction. However, MySQL data types do distinguish between integers and floating-point values. By looking at the different types of MySQL data, you will be able to see what types of data you can put into a table.

Numeric Data Types

Unlike PHP, numeric values in MySQL reserves space in a table in a database. In order to get just the right amount of space and not waste any, MySQL tables require that each numeric entry be of a specific type. This type expects some value, and fits a given value.

Table 11-1: MySQL Numeric Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT(P)</td>
<td>P=precision: 1 to 64</td>
</tr>
<tr>
<td>TINYINT</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>BOOL, BOOLEAN</td>
<td>0 or 1 (False/True)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>-32768 to 32767</td>
</tr>
<tr>
<td>MEDIUMINT</td>
<td>-8388608 to 8388607</td>
</tr>
<tr>
<td>INT, INTEGER</td>
<td>-2147483648 to 2147483647</td>
</tr>
<tr>
<td>BIGINT</td>
<td>-9223372036854775808 to 9223372036854775807</td>
</tr>
<tr>
<td>SERIAL</td>
<td>BIGINT UNSIGNED NOT NULL AUTO_INCREMENT UNIQUE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>-3.402823466E+38 to -1.175494351E-38, 0</td>
</tr>
<tr>
<td></td>
<td>1.175494351E-38 to 3.402823466E+38</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>-1.7976931348623157E+308 to -2.2250738585072014E-308, 0</td>
</tr>
<tr>
<td></td>
<td>2.2250738585072014E-308 to 1.7976931348623157E+308</td>
</tr>
<tr>
<td>DECIMAL(P,D)</td>
<td>P= precision:0-65, D=decimal points: 0-30</td>
</tr>
</tbody>
</table>

The history behind the different type and size of numeric data types reflects how little memory early computers had in relation to contemporary storage sizes. In order to use every little bit of persistent memory (hard-drive space), programmers didn’t want to use any more memory than they absolutely needed. For example, if 100 people were in a survey, programmers would be sure that they would use a TINYINT for IDs instead of the INT data type so as not to waste space they didn’t need. TINYINT handles values up to 127 while an INT can handle up to 2,147,483,647 values.

String Data Types

While numeric values are in some relatively familiar types (e.g., integers, float), strings in MySQL offer what may be new terms. The key differences are in the size of the allowable string and whether they are text or binary. The most basic string data
types are CHAR(n) and VARCHAR(n). The main differences between the two is that VARCHAR(n) handles variable sizes while CHAR(n) has a set size that cannot vary. Table 11-2 provides an overview of the string data types:

### Table 11-2: MySQL String Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR(n)</td>
<td>n=0-255 column width</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>variable up to n=65,535 columns</td>
</tr>
<tr>
<td>BINARY(n)</td>
<td>similar to CHAR but stores binary byte strings</td>
</tr>
<tr>
<td>VARCHARINARY(n)</td>
<td>similar to VARCHAR but stores binary byte strings</td>
</tr>
<tr>
<td>TINYBLOB</td>
<td>BLOB column with max length 255 bytes</td>
</tr>
<tr>
<td>TINYTEXT</td>
<td>TEXT column with max length of 255 characters</td>
</tr>
<tr>
<td>BLOB(n)</td>
<td>BLOB column with max length 65,535 bytes. The n value is optional length.</td>
</tr>
<tr>
<td>TEXT</td>
<td>TEXT column with max length of 65,535 characters</td>
</tr>
<tr>
<td>MEDIUMBLOB</td>
<td>BLOB column with max length 16,777,215 bytes</td>
</tr>
<tr>
<td>MEDIUMTEXT</td>
<td>TEXT column with max length of 16,777,215 characters</td>
</tr>
<tr>
<td>LONGBLOB</td>
<td>BLOB column with max length 4,294,967,295 or 4GB bytes</td>
</tr>
<tr>
<td>LONGTEXT</td>
<td>TEXT column with max length 4,294,967,295 or 4GB characters</td>
</tr>
<tr>
<td>ENUM('v1','v2',...)</td>
<td>String object with a single value from selected from list 'v1', 'v2' etc.</td>
</tr>
<tr>
<td>SET('v1','v2',...)</td>
<td>String object with 0 or more values selected from list 'v1', 'v2' etc.</td>
</tr>
</tbody>
</table>

The different kinds of BLOB data types are storage for binary strings (byte strings). They are different than TEXT data types because TEXT treats values as character strings rather than byte strings. When using large amounts of text as values, such as may be the case in storing posts or comments on an online blog, both the BLOB and TEXT data types may be used.

### Date and Time Data Types

The final type of MySQL data to be introduced here are date and time. These data types have a very rich set of formats, but MySQL has only three kinds of such data. Table 11-3 shows each:

### Table 11-3: Date and Time Data Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATETIME</td>
<td>YYYY-MM-DD HH:MM:SS</td>
</tr>
<tr>
<td>DATE</td>
<td>YYYY-MM-DD</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>1970-01-01 00:00:01' UTC to '2038-01-19 03:14:07' UTC</td>
</tr>
</tbody>
</table>
The DATETIME type has both the date and time, and the DATE type is the same except that it does not have the time. The TIMESTAMP is a value that is generated between January 1, 1970 and the current date and time.

One of the nice MySQL date and time features is that the current date and time may be automatically entered without having to assign a value through PHP. Built-in MySQL functions like NOW(), CURRENTTIME(), CURRENTDATE() and CURRENT_TIMESTAMP() handle setting date and time in MySQL without any need to use PHP variables or values. In the section, Creating a Table, further on in this chapter, you will see how to automate date, time and timestamp.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand working with different kinds of methods in an OOP environment.

- Create a new connection either on your computer or on a remote server and write the code to make the connection.
Chapter 12: Working with Tables

Any real work with databases begins with creating a table. In fact, some people confuse “tables” and “databases” because tables are so fundamental to working with databases. Tables are the “places” where you create your fields (column categories for different types of data) and store records (rows of filled-in columns). This chapter shows how to create and use tables with PHP and MySQL.

Table Basics
A table is nothing more than a grid. The columns are the fields and the rows the records. You first create a table by specifying the name of each of the fields and the type of data to store in the field. As each new record is entered, another row is added to the table. Table 12-1 shows a simple table with records. It has seven fields:

1. **ID**—each record needs a unique ID so that fields with the same information can be distinguished from other records with the same information.
2. **Name**—a text field used to store names.
3. **Member**—a Boolean field indicating whether the person is a member
4. **City**—text field for storing a city name
5. **State**—a two character field for storing state abbreviations
6. **Zip**—a five characters field for storing zip codes.
7. **Pet**—a field for indicating whether the person has a dog or cat

Chapter 11 shows the different data types used by MySQL and the amount of space each takes. As a general rule of thumb, you don’t want to use any more space than you have to in a table, but you will find that for most tables you will be using some data types a good deal and others, not so much or at all. The main focus in creating a table is to be sure to have the necessary fields for storing the data you will need.
Table 12-1

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Member</th>
<th>City</th>
<th>State</th>
<th>Zip</th>
<th>Pet</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>Jack Smith</td>
<td>Yes</td>
<td>Yuma</td>
<td>AZ</td>
<td>85364</td>
<td>dog</td>
</tr>
<tr>
<td>1002</td>
<td>Sue Gomez</td>
<td>Yes</td>
<td>Boston</td>
<td>MA</td>
<td>02201</td>
<td>cat</td>
</tr>
<tr>
<td>1003</td>
<td>Tyrone Ebbs</td>
<td>Yes</td>
<td>Chicago</td>
<td>IL</td>
<td>60604</td>
<td>cat</td>
</tr>
<tr>
<td>1004</td>
<td>Yusef Bata</td>
<td>Yes</td>
<td>Ocala</td>
<td>FL</td>
<td>34482</td>
<td>dog</td>
</tr>
<tr>
<td>1005</td>
<td>Bill Jackson</td>
<td>No</td>
<td>What Cheer</td>
<td>IA</td>
<td>50268</td>
<td>dog</td>
</tr>
<tr>
<td>1006</td>
<td>Betsy Kelly</td>
<td>Yes</td>
<td>Rome</td>
<td>GA</td>
<td>30163</td>
<td>dog</td>
</tr>
<tr>
<td>1007</td>
<td>Diego Lopez</td>
<td>No</td>
<td>Seattle</td>
<td>WA</td>
<td>98117</td>
<td>cat</td>
</tr>
</tbody>
</table>

Creating a Table
Even the most simple database and table make all the difference in the world when you’re a developer. You can now store and retrieve data from someplace other than a cookie on someone’s computer. A table is an organization of rows (records) and columns (fields) that can store user information from anywhere and retrieve it from anywhere. The first step in that process is creating a table using some simple SQL commands.

SQL Data Definitions
MySQL provides several different data definition statements. All of them revolve around working with a table. Included are the following:

- CREATE
You're very likely to be using all of these data defining statements with a table editor, but even the editors expect a certain set of rules to be followed. In the next subsections, you'll learn how to use each in conjunction with PHP.

**CREATE TABLE**

Of the data creation statements, the most important is CREATE because you use it to make your table and specify which data types it accepts. Figure 12-1 shows the basics of creating a table using several different data types:

```
CREATE TABLE tableName
(
  id SERIAL,
  dtime DATETIME DEFAULT NULL,
  name VARCHAR(30),
  item DECIMAL(7,2),
  PRIMARY KEY (id)
)
```

An important clause that you can include with an integer numeric variable is the AUTO_INCREMENT clause. When you create a table, you may need a unique identifier so that each record can be accessed individually. An integer variable may be used to automatically increment each entry with a unique value. The following statement is an example where “id” is the name of the field.

**Figure 12-1: Creating a table and adding data types.**
id INT NOT NULL AUTO_INCREMENT,

A similar and simpler way to create an automatic field is to use the SERIAL command.

id SERIAL,

That does the same thing except the data type is BIGINIT instead of INT. (See Figure 1.)

To make the field the primary key, the definition should also include a line that indicate that includes the key words and field name, such as the following:

PRIMARY KEY (id)

The field with the auto-increment is typically used as the primary key, as is the case in this example. The SERIAL data type encompasses all of the auto-increment information as can be seen in Figure 1 and in Listing 2.

In order to use MySQL statements with PHP, the statements are placed into a PHP variable. For example, the following snippet shows how a creation statement is placed into a variable:

```php
$this->sql = "CREATE TABLE $this->tableMaster ( 
    id SERIAL, 
    orderdate DATETIME DEFAULT NULL, 
    cuslast NVARCHAR(15), 
    cusfirst NVARCHAR(15), 
    cusemail NVARCHAR(40), 
    service DECIMAL(9,2), 
    PRIMARY KEY (id))";
```

The PHP variable referenced as $this->tableMaster is simply the name of a private variable containing the table's name. Likewise, the $this->sql variable is a private variable name to remind the developer that the variable has been assigned an SQL statement.

To get started, use the following class to create a table for a Web business. It uses the IConnect interface and UniversalConnect classes; so be sure to have them in the same directory as the WebBus.php file in Listing 4. (There is no input yet, but you need a table first for something to input into!)

*Listing 12-4: WebBus.php*

```php
<?php
```
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('UniversalConnect.php');
class CreateTable
{
    private $tableMaster;
    private $hookup;
    private $drop;
    private $sql;

    public function __construct()
    {
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        $this->dropTable();
        $this->makeTable();
        $this->hookup->close();
    }

    private function dropTable()
    {
        $this->drop = "DROP TABLE IF EXISTS $this->tableMaster";

        try
        {
            $this->hookup->query($this->drop) === true;
            printf("Old table %s has been dropped.<br/>",$this->tableMaster);
        }
        catch (Exception $e)
        {
            echo "Here is why it did not work:  $e->getMessage() <br />";
        }
    }

    private function makeTable()
    {
        $this->sql = "CREATE TABLE $this->tableMaster (id SERIAL,
        orderdate DATETIME DEFAULT NULL,
        cuslast NVARCHAR(15),
        cusfirst NVARCHAR(15),
        cusemail NVARCHAR(40),
        service DECIMAL(9,2),
        PRIMARY KEY (id))";
    }
}
try
{
    $this->hookup->query($this->sql);
}

    catch (Exception $e)
    {
        echo 'Here is why it did not work: ', $e->getMessage(), "<br />
    }
    echo "Table $this->tableMaster has been created successfully.";
}
$worker=new CreateTable();
?>

DROP and ALTER TABLE
The DROP TABLE statement deletes a table. Ironically, one of the most-used implementation of DROP TABLE is in creating a table. By checking first to see if a table exists, your program will not run into a “Table Exists” error if it first uses the DROP statement before using the CREATE statement. Before dropping a table, along with all of its contents, you want to be sure that you really want to delete it. Sometime, it’s easy to make one little mistake and then find that you cannot re-create the table because it already exists. So, using the DROP-CREATE sequence insures that you’re starting off with a clean slate as shown in Listing 12-4.

An important step is to print to the screen the fact that the old table was removed and that the new table has been created. Also, notice the clause IF EXISTS in the DROP TABLE statement. The clause is just like a conditional statement and only will execute if a table exists. Finally, in the PHP if statements;

if($hookup->query($drop) === true)…
...
if($hookup->query($sql) === true)…

note the use the identical operator (===). Basically, it’s finding whether the operation took place—it’s true.

When you have found a minor change you want to make in a table, such as the name of a column, you can use the ALTER statement instead of DROP. This is especially important if you have content already in the table. For instance, suppose your start finding surnames like van Oldenbarnevelt (Dutch), von der Reingruben (German), Fernández de Calderón (Spanish) and Kamakanamaikalani (Hawaiian) in your customer list. They will not fit in the 15-character field. You decide that you’d better
make the *cuslast* field longer and change the name of the field to *cusSurname* while you’re at it. Listing 12-3 shows how to make the change:

*Listing 12-3: AlterTable.php*

```php
<?php
// AlterTable.php
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('UniversalConnect.php');
class AlterTable
{
    private $tableMaster;
    private $hookup;
    private $alter;

    public function __construct()
    {
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        $this->alter = "ALTER TABLE $this->tableMaster
CHANGE cuslast cusSurname VARCHAR(26)";

        if($this->hookup->query($this->alter) === true)
        {
            printf(" Column 'cuslast' is now 'cusSurname' in
            $this->tableMaster <br/>");
        }
        else
        {
            printf("Here's what went wrong: %s
", $this->hookup->error);
        }
        $this->hookup->close();
    }
$worker=new AlterTable();
?>
```

The most important feature of the ALTER statement is that it does not delete the existing records. Further, if all you want to change is the name of a column, putting in the size and type of the data to be entered insures that your original data is not changed. For instance, if NVARCHAR(15) is changed to NVARCHAR(30), the existing data would be changed from 15 to 30 columns as well.
RENAMER and TRUNCATE
The RENAME statement is very simple. The statement is used to change the name of a table. All of the records are preserved and the only change is the table’s name. Listing 12-4 shows how to rename an existing table:

Listing 12-4: RenameTable.php
<?php
// RenameTable.php
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('UniversalConnect.php');
class RenameTable
{
    private $tableMaster;
    private $renamedTable;
    private $hookup;
    private $rename;

    public function __construct()
    {
        $this->tableMaster="webbus";
        $this->renamedTable="webwork";
        $this->hookup=UniversalConnect::doConnect();
        $this->rename = "RENAME TABLE $this->tableMaster TO $this->renamedTable";

        if($this->hookup->query($this->rename) === true)
        {
            printf(" The table $this->tableMaster is now named $this->renamedTable .  <br/>");
        }
        else
        {
            printf("Here's what went wrong: %s\n", $this->hookup->error);
        }
        $this->hookup->close();
    }
    $worker=new RenameTable();
?>

That's about as straightforward and clear as you can get. The same table just has a different name. (If you're going to use the "webbus" table, just use the same program to rename "webwork" back to "webbus."
Finally the TRUNCATE statement removes all of the records from a table. When developing a table, you may place several different “test” records in to debug it, and once you have everything just right, you want to get rid of all of the records. Of course you may have other reasons for deleting all of the records in a table but preserving its structure. Using TRUNCATE you can delete all of the records in one fell swoop rather than having to delete records one-at-a-time. Listing 12-5 shows how to use the TRUNCATE statement:

Listing 12-5: TruncateTable.php
<?php
    // TruncateTable.php
    ini_set("display_errors","1");
    ERROR_REPORTING( E_ALL | E_STRICT );
    include_once('UniversalConnect.php');
    class TruncateTable
    {
        private $tableMaster;
        private $hookup;
        private $truncate;

        public function __construct()
        {
            $this->tableMaster="webbus";
            $this->hookup=UniversalConnect::doConnect();
            $this->truncate = "TRUNCATE TABLE $this->tableMaster";

            if($this->hookup->query($this->truncate) === true)
            {
                printf(" All of the records in $this->tableMaster have been removed .  \n");
            }
            else
            {
                printf("Here's what went wrong: %s\n", $this->hookup->error);
            }
            $this->hookup->close();
        }
    }
    $worker=new TruncateTable();
?>

A further advantage of using TRUNCATE instead of deleting all of the rows is that it resets any auto-increments to 0. For example, if you delete a record, the auto-increment in the SERIAL data type increment to the next record. If you have 342
records and use auto-increment, if you delete all of the records individually and start with a new record, it will be 343, but with TRUNCATE, it will be 0.

**Do-It-Yourself Exercises**

The following exercises are designed to help you understand working with different kinds of methods in an OOP environment.

- Create a table with at least one data type from strings, dates and numbers
- Change one feature in a table using the ALTER statement
Chapter 13: Data Manipulation

Once you have your connection established and your table created, you’re all set to start using your database. If you stop and think about what you can do with a database table the following come to mind:

- Add data to the table
- Retrieve data from the table
- Change data in the table
- Delete records from the table

The first two, adding data to and retrieving it from a table, are the most common usages you’ll have. Users will type in information, and it will be stored to be retrieved and perhaps acted upon by the hosting company. The following scenario is a typical one:

A company wants to sell customers a product online. The customer enters the product desired, payment information and shipping information. The company retrieves this information from the table, ships the product, and indicates that the transaction is complete.

If you have some picture like the above scenario in mind when developing and using tables, you’ll find that the process is much clearer. The SQL statements that you’ll learn in this chapter show how to manipulate that table data in a useful and practical manner.

Inserting Data into a Table

When inserting data into a table, the first step is to make an HTML5 UI so that the user can enter data. Then, we need to make a table that can accept everything the user enters. Finally, we need a PHP program that inserts data into the table.

Of course the real starting point is determining what you can offer the user from your Web site. We’ll make it simple but realistic enough to be useful. In this example, the user wants a Web site, and you have several to offer. Each different type of Web site will have a different price. You’ll need a name (first and last) of the customer and the person’s email address. You will need a date and time of the order and some kind of unique key to identify the user. That’s it.

Setting Up HTML5 Data Entry

The data entry should be simple for the user. Clients want to order a Web site from you, and you can deliver, and you want to make it look nice; so you’ll have a CSS file
that makes it nice and clear. Listing 13-1 is the CSS file that uses a brown tint color scheme.

Listing 13-1: webser.css
@charset "UTF-8";
/* CSS Document */
/*CBB8A6,CC9769,958679,7F5E41,4C3827—color palette*/
body {
    background-color:#CBB8A6;
    color:#4C3827;
    font-family:Verdana, Geneva, sans-serif;
}

h1 {
    text-align:center;
    font-family:"Arial Black", Gadget, sans-serif;
    background-color:#CC9769;
    color:#4C3827;
}

h2 {
    color:#958679;
    background-color:#4C3827;
    width:340px;
}

.chooser {
    color:#7F5E41;
    font-size:11pt;
}

legend {
    color:#958679;
}

fieldset {
    width:230px;
}

iframe {
    background-color: #CC9769;
}

Next, the data input won’t take much time for the client because it requires only a name and email address to be typed in, and the rest can be done by clicking checkbox and submit buttons. Listing 13-2 shows how the data entry page is set up:

Listing 13-2: WebServices.html
<!DOCTYPE HTML>
<html>
<head>
<link rel="stylesheet" href="webser.css">

184
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Acme Web Sites</title>
<body>
<h1>Acme Web Services</h1>
<form action="DataEntry.php" method="post" target="feedback">
  <input type="text" name="fname"><br>
  First name <br>
  <input type="text" name="lname"><br>
  Last name<br>
  <input type="email" name="cusEmail"><br>
  Email address<br>
  <fieldset>
    <legend>Choose Your Web Service</legend>
    <section class="chooser">
      <input type="checkbox" name="service[]" value=100.95>
      Design $100.95<br>
      <input type="checkbox" name="service[]" value=162.75>
      Social Media $162.75<br>
      <input type="checkbox" name="service[]" value=227.55>
      Add Mobile $227.55<br>
      <input type="checkbox" name="service[]" value=318.45>
      Database $318.45<br>
    </section>
    <input type="submit" name="send" value="Send us your choice">
  </fieldset>
  <iframe name="feedback" seamless width="300" height="120">php</iframe>
</form>
</body>
</html>

You will remember from Chapter 8 that to pass data from HTML to PHP, you need both a name and a value to send. With text input, the value is whatever the user types in, but with checkbox buttons you need to add a value. In this case instead of a string value describing the type of Web site requested, the cost is used so that it can be entered directly as a decimal value. Figure 13-1 shows what the HTML data entry looks like to the user:
Each time a user enters data and sends it to the PHP file the process creates a new record. You might think of each user entry as a unique record. The same user might enter lots of records by clicking the “submit” (Send us your choice) button several times. As far as the table is concerned, it’s just another record whether it’s from the same or different user. The ID is with the record.

Creating the Table
Once you know the data you want to capture, create a table that has the necessary columns to store the data. Listing 13-3 shows the same table created in Chapter 9, and it is repeated here for convenience:

Listing 13-3: CreateTable.php
<?php
ini_set("display_errors","1");
ERROR_REPORTING( E_ALL | E_STRICT );
include_once('UniversalConnect.php');
class CreateTable
{
  private $tableMaster;
  private $hookup;
  private $drop;
  private $sql;

  public function __construct()
  {
    $this->tableMaster="webbus";
    $this->hookup=UniversalConnect::doConnect();
    $this->dropTable();
    $this->makeTable();
    $this->hookup->close();
  }

  private function dropTable()
  {
    $this->drop = "DROP TABLE IF EXISTS $this->tableMaster";
    try
    {
      $this->hookup->query($this->drop) === true;
      printf("Old table %s has been dropped.<br/>",$this->tableMaster);
    }
    catch (Exception $e)
    {
      echo "Here is why it did not work: $e->getMessage()<br/>";
    }
  }

  private function makeTable()
  {
    $this->sql = "CREATE TABLE $this->tableMaster (
      id SERIAL,
      orderdate DATETIME DEFAULT NULL,
      cuslast NVARCHAR(15),
      cusfirst NVARCHAR(15),
      cusemail NVARCHAR(40),
      service DECIMAL(9,2),
      PRIMARY KEY (id))";
    try
    {
      $this->hookup->query($this->sql);
    }
    catch (Exception $e)
    {
      echo 'Here is why it did not work: ', $e->getMessage(), "<br />";
    }
    echo "Table $this->tableMaster has been created successfully.<br/>";
  }
}
There’s nothing new in the table named **webbus** that you haven’t seen in Chapter 9. It has columns from the data passed from the HTML page and for automatically generated unique IDs and date/time data.

**Using the SQL INSERT Statement**

Once you have the table successfully created, you’re all set to enter the data from the HTML5 page created in Listing 13-2. Basically, you want to pass the HTML object names and values to PHP variables that are in turn used in an SQL statement. The SQL statement to enter data is illustrated in Figure 13-2:

```
INSERT INTO "tableName"
    (col1, col2, col3) VALUES ('$val1', '$val2', '$val3')
```

While the statement in Figure 13-2 may look a bit daunting, it’s really very simple in this basic form. In a simplified statement you have,

```
INSERT INTO "myTable" (col1, col2) VALUES ('$val1', '$val2')
```

The SQL statements are placed into a variable for simplicity and then acted up on by a **mysqli** object. Listing 13-4 shows the PHP that stores the data from the HTML into the table “webbus.” Also, please note that you will need the **UniversalConnect.php** and **IConnect.php** files from Chapter 9 again. Place them into the same folder as your other files that need connection to the MySQL database.

**Listing 13-4: DataEntry.php**
```
<?php
include_once("UniversalConnect.php");
class DataEntry
{
```
//Variables for MySql connection
private $hookup;
private $sql;
private $tableMaster;

//Field Variables
private $fname;
private $lname;
private $email;
private $service;
private $total;

public function __construct()
{
    //Get table name and make connection
    $this->tableMaster="webbus";
    $this->hookup=UniversalConnect::doConnect();

    //Get data from HTML form
    $this->fname=$_POST['fname'];
    $this->lname=$_POST['lname'];
    $this->email=$_POST['cusEmail'];
    $this->service=$_POST['service'];
    $counter=0;
    foreach ($this->service as $amount)
    {
        $counter += $amount;
    }
    $this->total=$counter;

    //Call private methods for MySql operations
    $this->doInsert();
    $this->hookup->close();
}

private function doInsert()
{
    $this->sql = "INSERT INTO $this->tableMaster (orderdate,cuslast,cusfirst,cusemail,service) VALUES (NOW(),'$this->lname','$this->fname','$this->email','$this->total');"

    try
    {
        $this->hookup->query($this->sql);
        printf("Data for %s %s was entered successfully:",$this->fname,$this->lname);
    }
    catch (Exception $e)
    {
        echo "There is a problem: " . $e->getMessage();
        exit();
    }
}

$worker=new DataEntry();
?>
In the INSERT statement assigned to the $sql variable the following data sources can be identified:

- **id**: Automatically generated primary key (MySQL)
- **orderdate**: Provided by MySQL NOW() function
- **lname**: Passed from HTML to PHP
- **fname**: Passed from HTML to PHP
- **email**: Passed from HTML to PHP
- **total**: Passed from HTML as “service” and calculated in PHP for “total”

So while data are entered into the table only when the PHP program is launched by a call from the HTML page, not all of the data inserted into the table are from the HTML page. In this example, two data sources are generated solely by the MySQL functions—the id and date and time.

**Protecting Against an Injection Attack**

Injection attacks occur when attackers enter certain types of special characters that can circumvent login names and passwords.

First, set your character set to one that will limit what can be entered as recognized characters using the API function $mysqli->set_charset:

```php
$mysqli->set_charset("utf8");
```

Next, use the mysqli->real_escape_string() method to filter out any strings that might be used in an injection attack.

```php
//Protection against injection attack
$this->fname = $this->hookup->real_escape_string($_POST['fname']);
$this->email = $this->hookup->real_escape_string($_POST['cusEmail']);
$this->service = $this->hookup->real_escape_string($_POST['service']);
```

You might wonder why anyone would want to enter harmful data into your database, but either by intent or accident, it can happen. If you have a login system with username and passwords, you definitely want to use the real_escape_string() function.

**No Guarantees Against Hacks into PHP**

While using the mysqli->real_escape_string() method will stop unsophisticated hackers and honest mistakes, don’t expect them to stop all malicious attacks. Throughout the rest of the book, you won’t see the use of mysqli->real_escape_string() because the focus is on OOP programming and not cyber security. As you can see, however, adding the method is a simple matter, and you should consider using it as an ounce of prevention. When dealing with actual customers beyond development practice, even a little security is better than none at all, and don’t forget to add the $mysqli->set_charset API function!
Retrieving Data From a Table
When requesting data from a table, the main SQL statement is SELECT. It has the basic format,

```
SELECT columnName FROM tableName
```

You can select as many columns as you want with each column name separated by a comma in the following format:

```
SELECT columnA, columnB, columnC FROM tableName
```

A common request is for all columns, and instead of having to specify all of the columns, the wildcard symbol (*) is used instead. The following SQL line requests all of the columns from a table:

```
SELECT * FROM tableName
```

Beginning with these simple SQL statements, you can see how the `mysqli` tools in PHP transfer the data to useful output.

Queries and Results
A more realistic understanding of storing and using data and a database can be seen with an administrative module. People with businesses want users to store their Web orders, but they do not want their customers to see what data other customers may have stored. Use the “administrative module” shown in Listing 13-5 for viewing and changing data in a MySQL database:

Listing 13-5: WebAdmin.html

```html
<!DOCTYPE HTML>
<html>
<head>
<link rel="stylesheet" href="webser.css">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Acme Administrator</title>
</head>
<body>
<h1>Acme Administrator</h1>
<h2>&nbsp;Change Email Address</h2>
<form action="DataUpdate.php" method="post" target="feedback">
&nbsp;New Email Address: <br />
<input type="text" name="newEmail">
<br />
&nbsp;ID Number <br />
<input type="text" name="id" size="6">
<br />
<input type="submit" name="addr" value="Change email">
</form>
<h2>&nbsp;Delete Record</h2>
<form action="DeleteRecord.php" method="post" target="feedback">
```
As you go through the rest of the chapter, look for the names of the files and associated classes referenced in the forms that make up the administrative module. Figure 13-3 Shows the administrative module as it will appear on your computer:
Figure 13-3 shows the results of using the `DataDisplay` class shown in Listing 13-5.
The most useful aspect of PHP and MySQL comes together because *PHP treats the table rows as arrays*. Each row (or each record) is an array, and each field (column) is an element. So if a table has 20 fields, each of the rows is an array with 20 elements. For instance, in the `webbus` table each record has six elements: id, fname, lname, email, website, and orderdate. As an array, we can imagine it as:

```
$row[0]
$row[1]
$row[2]
$row[3]
$row[4]
$row[5]
```

Even better, PHP will return the contents of a table row as an associative array with the keys being the field names. In an associative array, the elements would look like the following:

```
$row['id']
$row['orderdate']
$row['cuslast']
$row['cusfirst']
$row['cusemail']
$row['service']
```

In either case, the MySQLi class provides methods to handle the array that result from the query.

In the examples, the MySQLi instance has been placed into a variable (object) named `$hookup` (indicating a connection). A `query()` method sends the SQL statement to the database to retrieve the data.

To see how this works, Listing 13-5 shows how all of the data are retrieved from a table.

*Listing 13-5 DataDisplay.php*

```php
<?php
include_once("UniversalConnect.php");
class DataDisplay
{
    //Variable for MySql connection
    private $hookup;
    private $sql;
    private $tableMaster;

    public function __construct()
    {
```
//Get table name and make connection
$this->tableMaster="webbus";
$this->hookup=UniversalConnect::doConnect();
$this->doDisplay();
$this->hookup->close();

private function doDisplay()
{
    //Create Query Statement
    $this->sql = "SELECT * FROM $this->tableMaster";

    try
    {
        $result = $this->hookup->query($this->sql);
        printf("Select returned %d rows.<p />", $result->num_rows);
        while ($row = $result->fetch_assoc())
        {
            printf("ID: %s Name: %s %s <br /> Email: %s <br />
                    Total: %s <br /> Orderdate: %s <br />, $row['id'],
                    $row['cusfirst'], $row['cuslast'],$row['cusemail'],
                    $row['service'],$row['orderdate']);
        }
        $result->close();
    }
    catch(Exception $e)
    {
        echo "Here's what went wrong: " . $e->getMessage();
    }
}
$worker = new DataDisplay();
?>

In breaking down the crucial code, the SQL statement is placed into a variable ($this->sql). This is done to make it easier to place the statement into the mysqli->query() method ($this->hookup->query($this->sql)). The MySQLi query statement is assigned to an object, $result. Now the result object has all of the properties and methods of the MySQLi object. (Keep in mind that all of the necessary setup for your mysqli operations is done in the UniversalConnect class.) Finally, a new object, $row is assigned the contents of the current row within the while statement:

while ($row = $result->fetch_assoc())
Because each $row is an array made up of the current iteration through the table records, it can be treated as a unique array made up of the row’s column values. The fetch_redo() method, means that all of the keys for the associative array will be the names of the columns. If the fetch_array() method were used, the table rows would be returned as both/either associative or numeric arrays.

**Filtering and Ordering**

While MySQL has many different methods for searching the contents of a table and filtering the results, one of the most common uses the **WHERE** clause in the format:

```
SELECT column(s) FROM tableName WHERE condition
```

For instance, you might want to find the names of the customers who have purchased Web sites with values in excess of $500. The following SQL statement would find the first and last names of such clients:

```
SELECT id, cuslast, cusfirst FROM $this->tableMaster WHERE service > 500
```

In this case, it only looks at four columns: the ID, the first and last name and the value of the Web services. Listing 13-6 shows the full PHP program:

**Listing 13-6: SelectDisplay.php**

```php
<?php
include_once("UniversalConnect.php");
class SelectDisplay
{
    //Variable for MySql connection
    private $hookup;
    private $sql;
    private $tableMaster;

    public function __construct()
    {
        //Get table name and make connection
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        $this->doDisplay();
        $this->hookup->close();
    }

    private function doDisplay()
    {
        //Create Query Statement
        $this->sql ="SELECT id, cuslast, cusfirst FROM $this->tableMaster WHERE service > 500";
```
try
{
    $result = $this->hookup->query($this->sql);
    printf("Select returned %d rows.<p/>", $result->num_rows);
    while ($row = $result->fetch_assoc())
    {
        printf("ID: %s Name: %s %s <p/>", $row['id'], $row['cusfirst'], $row['cuslast']);
    }
    $result->close();
}
catch(Exception $e)
{
    echo "Here's what went wrong: " . $e->getMessage();
}
$
worker = new SelectDisplay();
?>

The SELECT statement has many other clauses for further refinements in a search of a table. For example, with the WHERE clause, you can include logical operators. The following shows a simple example:

SELECT * FROM $tableName WHERE service > 100 && service < 400

Only records that have a website value greater than 100 and less than 400 will be selected.

Along with the WHERE clause, you may optionally use the ORDER BY clause. By default, the records are ordered by the automatically generated id values. However, suppose you wanted to order the output by the value of the websites with the highest values first (descending order.) The ORDER BY clause default is in ascending order. However, it has a DESC option to order by descending order. The following statement shows an ordering by the service column in descending order:

SELECT * FROM $tableName ORDER BY service DESC

Even more sophisticated searches are possible using the different SELECT clauses, but the ones we covered should be enough to get started and encourage exploration.
Changing Data in a Table
In a dynamic environment, the content of a single record or even all records often need to be changed. The UPDATE statement is one that effects changes in record. It has the following syntax:

UPDATE $tableName SET colName='$newInfo'

The change is made using the SET keyword to indicate the column and assign it the new value. In cases where all columns need to have the same changes made, that’s pretty easy. For more specific changes in single records and single fields, more refined UPDATE clauses are required.

Updating an Entire Table
Sometimes you need some feature of your entire table changed. For example, suppose that you decide that the automatic user id needs to be increased by 1000. So instead of,

1
2
3
etc.

the id values are

1001
1002
1002

Take for example the following table output values:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Email</th>
<th>Purchase Amount</th>
<th>Date and time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jack Bequick</td>
<td><a href="mailto:zoom@fast.com">zoom@fast.com</a></td>
<td>200.00</td>
<td>2014-01-03 08:24:46</td>
</tr>
<tr>
<td>2</td>
<td>Nancy Moss</td>
<td><a href="mailto:mossy@nice.com">mossy@nice.com</a></td>
<td>100.00</td>
<td>2014-01-03 08:26:43</td>
</tr>
<tr>
<td>3</td>
<td>Benny Jet</td>
<td><a href="mailto:bball@fld.com">bball@fld.com</a></td>
<td>200.00</td>
<td>2014-01-03 08:27:09</td>
</tr>
<tr>
<td>4</td>
<td>Judy Galore</td>
<td><a href="mailto:jg@tx.com">jg@tx.com</a></td>
<td>300.00</td>
<td>2014-01-03 08:27:44</td>
</tr>
<tr>
<td>5</td>
<td>Freddy Wille</td>
<td><a href="mailto:fastf@foolis.net">fastf@foolis.net</a></td>
<td>150.00</td>
<td>2014-01-03 08:29:00</td>
</tr>
<tr>
<td>6</td>
<td>John Goode</td>
<td><a href="mailto:jd@jail.com">jd@jail.com</a></td>
<td>300.00</td>
<td>2014-01-04 15:42:58</td>
</tr>
<tr>
<td>7</td>
<td>Chester Gunn</td>
<td><a href="mailto:gunny@usmc.gov">gunny@usmc.gov</a></td>
<td>150.00</td>
<td>2014-01-05 05:26:01</td>
</tr>
</tbody>
</table>
You want the ID to start with 1001; so all you need to do is to add 1000 to each id value. The general UPDATE statement will do the trick:

```
UPDATE $tableName SET id = id + 1000
```

For a full example, Listing 13-7 shows how:

**Listing 13-7: IdUpdate.php**

```php
<?php
include_once("UniversalConnect.php");
class IdUpdate
{
    private $hookup;
    private $tableMaster;
    private $sql;
    public function __construct()
    {
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        //Call update
        $this->changeId();
        //Close
        $this->hookup->close();
    }
    private function changeId()
    {
        $this->sql ="UPDATE $this->tableMaster SET id = id +1000";
        try
        {
            $result = $this->hookup->query($this->sql);
            echo "All IDs have been changed.<br />";
        }
        catch(Exception $e)
        {
            echo "Here's what went wrong: " . $e->getMessage();
        }
    }
}$worker = new IdUpdate();
?>
```

The entire table now has a base of 1000 instead of 0 and has the following output:

- ID: 1001 Name: Jack Bequick Email: zoom@fast.com Purchase Amount: 200.00 Date and time: 2014-01-03 08:24:46
- ID: 1002 Name: Nancy Moss Email: mossy@nice.com Purchase Amount: 100.00 Date and time: 2014-01-03 08:26:43
- ID: 1003 Name: Benny Jet Email: bball@fld.com Purchase Amount: 200.00 Date and time: 2014-01-03 08:27:09
- ID: 1004 Name: Judy Galore Email: jg@tx.com Purchase Amount: 300.00 Date and time: 2014-01-03 08:27:44
The most important feature of this change is that when new data are entered, the id values are automatically updated on the basis of the last number in the sequence, 1007 instead of 7. So the next new record will be 1008 instead of 8.

**Targeted Updates**

An equally important type of update is when you need to target a specific field in a specific record. In these cases the WHERE clause may be used. The general format for such an update is where you specify a certain column with a certain value. For example, the following snippet changes only those columns that contain a certain value:

```
UPDATE $tableName SET col5='$newInfo' WHERE col5='$oldinfo'
```

For example, suppose that a person has to go into witness protection and his name must be changed from Smith to Jones. Your table shows you have a Joe Smith, and all you want to change it to Joe Jones. How would you do that? The basic steps include:

- Find the record you want to change
- Make the change

Since all IDs are unique, they are useful for making changes where only specific records are to be changed. The following snipped from an administrative HTML form shows how the change is entered:

```
<h2>&nbsp;Change Email Address</h2>
<form action="DataUpdate.php" method="post" target="feedback">
   &nbsp;New Email Address: <br />
   <input type="text" name="newEmail">
   <br />
   &nbsp;ID Number <br />
   <input type="text" name="id" size="6">
   <br />
   <input type="submit" name="addr" value="Change email">
</form>
```

The data are sent to the PHP program in the same way as other data that you may want to enter or give another SQL instruction. Listing 13-8 shows the PHP class that uses the change data:
Listing 13- 8 DataUpdate.php

```php
<?php
include_once("UniversalConnect.php");
class DataUpdate
{
    private $hookup;
    private $tableMaster;
    private $sql;
    //Fields
    private $id;
    private $email;
    public function __construct()
    {
        $this->id=intval($_POST['id']);
        $this->email=$_POST['newEmail'];
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        //Call update
        $this->changeEmail();
        //Close
        $this->hookup->close();
    }
    private function changeEmail()
    {
        $this->sql = "UPDATE $this->tableMaster SET
cusemail='$this->email' WHERE id='$this->id'";
        try
        {
            $result = $this->hookup->query($this->sql);
            echo "Email update complete.<br />
        } catch(Exception $e)
        {
            echo "Here's what went wrong: " . $e->getMessage();
        }
    }
}$worker = new DataUpdate();
?>
```

Many PHP programmers use the `isset()` function to check and make sure that the submitted data from the correct source has arrived. For example, the name of the submit button is `addr` if you want to be sure that the data is coming from the correct form, you can use the following:
if(isset($_POST['addr']))

looks to see whether the submit button named ‘change’ has sent the data. The other submit button is named ‘send’, and so you can reduce the possibility of working with the wrong data by having names for the submit buttons and using them with the `isset()` function.

**Deleting Records from a Table**

In addition to changing records or parts of records, you may have occasion to remove records. As you saw in Chapter 8, the TRUNCATE statement will remove all records from a table. For less Draconian record removal, the DELETE statement is very helpful. You may find that your database is storing thousands of records that you no longer want taking up space but at the same time you want to keep others. For even more fine-tuning, you may just wish to remove a selected record from a table.

**Deleting With the WHERE Clause**

When you want to clean up a table, usually you’re just going to remove certain records that meet criteria that indicate that they are no longer useful. For example, suppose that you maintain a mailing list, and once a year you use it to send out solicitations to raise money for the homeless. Since it costs money to mail each solicitation, you decide that if someone has not made a contribution greater than $3, you’ll eliminate them from the list.

```
DELETE FROM donateTable WHERE contribution < 3
```

As you can see, the syntax is similar to other MySQL statements that use the WHERE clause. However, with DELETE, the entire record is eliminated.

The following segment shows one way to delete only unique records:

```
<h2>Delete Record</h2>
<form action="DeleteRecord.php" method="post" target="feedback">
    ID Number of Record to Delete<br />
    <input type="text" name="idd" size="6">
    <br />
    <input type="submit" name="kill" value="Delete Record">
</form>
```

The labels and names are fairly deadly ones using the “kill” word throughout. The reason for that is to remind the user that the action will result in one or more deleted records. Listing 13-9 shows the accompanying PHP used to delete a record:

*Listing 13-9: DeleteRecord.php*
<?php
include_once("UniversalConnect.php");
class DeleteRecord
{
    //Variables for MySql connection
    private $hookup;
    private $sql;
    private $tableMaster;
    //From HTML
    private $deadman;

    public function __construct()
    {
        $this->deadman =intval($_POST['idd']);
        //Get table name and make connection
        $this->tableMaster="webbus";
        $this->hookup=UniversalConnect::doConnect();
        $this->recordKill();
        $this->hookup->close();
    }

    private function recordKill()
    {
        //Create Query Statement
        $this->sql ="Delete FROM $this->tableMaster WHERE id='$this->deadman'";

        try
        {
            $result = $this->hookup->query($this->sql);
            printf("Record with ID=%s: has been dropped.<br />", $this->deadman);
        }
        catch(Exception $e)
        {
            echo "Here's what went wrong: " . $e->getMessage();
        }
    }

$worker = new DeleteRecord();
?>

That particular use of DELETE will wipe out every record that has the last name sent from the HTML UI. So if you have ten names with “Smith” in your records, the program will delete all of them if the “kill name” is Smith. Only by using unique
names (such as an ID) can you be sure that what you want deleted and only that is affected by the DELETE action.

**The LIMIT Clause**

When you want to only delete a certain number of records, you can use the LIMIT clause. For example if you want to delete only one record, use the following format:

```sql
DELETE FROM $tableName WHERE lname = 'Smith' LIMIT 1;
```

If you have ten ‘Smiths’ in your table, it will only eliminate one. However, it eliminates the first one it encounters in a table. If you want to get rid of the first one that technique works quite well. The LIMIT clause is intended for very large databases and the developer does not want to tie up the database too long with a large number of DELETE operations. Also, you can do other things to manipulate your data to bring the intended recorders to delete to the top of the table order and then use the LIMIT to knock the top records off. However, you are far better off using DELETE targeting the most precise criteria in the WHERE clause.

**Do-It-Yourself Exercises**

This chapter has been about moving data between a UI and a table. The following exercises are designed to give you some simple practice applications.

- Create a Table with columns for an automatic ID and three string columns using `varchar()` types. You decide what the three string columns will be. Then, create an HTML5 page that you can use to enter data into the table.
- Make a PHP application using the `mysqli` class to display all records in the table. (Be sure to enter lots of records.)
- Create a PHP application that will allow you to change (UPDATE) any of the string fields in your table. Add HTML modules to the Web page used to enter data into the table.
Chapter 14: Employing Programming Principles in PHP

This chapter illustrates general OOP principles applied to PHP using both standard OOP and Design Patterns in PHP.

- SOLID by Design
  - Single Responsibility
  - Open/closed Principle
  - Liskov substitution principle
  - Interface segregation
  - Dependency Inversion (Program to the interface; not implementation)
    - Dependency injection

- Design Patterns
  - Creational Example:
    - Builder
    - One process: Different Objects
  - Structural Example
    - Bridge
    - Make a simple CMS using Bridge
  - Behavioral Example
    - Iterator
    - Creating a PHP Skip List and using the Iterator pattern to transverse it

- Program to an Interface with Type declarations
- Choose Composition over Inheritance.
Appendix A

Setting Up PHP on Your Computer: Using Localhost as a Server

You can access PHP in three ways.

- Sign up for a hosting service
- Download and install a server and PHP on your computer
- If you have Mac OS X, it's already on your computer—just set it up.

Two involve setting up a server on your computer, and the third depends on having a hosting service. The easiest thing to do is to sign up with a hosting service with PHP. Then you load your PHP files like you would a Web page. One that has been tested extensively and is reasonably priced is at http://www.jtl.net. The Linux minimum service is all you need.

If you want to install PHP on your computer do the following:

1. **Windows Only:** Go to http://windows.php.net/download and download the latest stable version of PHP5 (You'll find What version do I choose? in the left column to help you choose what you need for your system.) You'll also need to install an Apache server that you can get free from http://www.apache.org/.

2. **Macintosh Only:** Go to http://akrabat.com/computing/setting-up-php-mysql-on-os-x-mavericks/ and follow the instructions for accessing the PHP on your system. (It's important to be very careful because you're going to be using the built-in Terminal program in your Mac.) It shows how to set up both PHP and your built-in Apache server. If your have an older Mac, you may have to search around on the Web. If you go to http://www.php.net you will find a stable version that you can download free.

3. **Raspberry Pi & Linux:** Go to http://www.php5dp.com/easy-writer-setup-for-raspberry-pi-php/ for a simple set of steps to install PHP on your Raspberry Pi. It was done using the Raspbian version of Linux, and if you have a different version, you should check the Linux OS install instructions for the different implementations of Linux (e.g., Pidora Archlinux). See also, http://www.php.net/manual/en/install.unix.php.

4. **For Everything:** If you want to download and install everything at once (PHP, Apache Server, and a MySQL database) for your Mac go to: http://www.mamp.info/en/index.html and for Windows go to http://www.wampserver.com/en/ and you can take care of everything at once. This is the easiest way to set up an actual database on your computer.
It can be awkward setting up PHP and Apache, but once it’s set up, you don’t have to do it again. If you use the all-in-one method, you can get the MySQL server with which you can set up a database on your computer.

Testing PHP
Once you have your system set up, whether it is on your computer or a hosting service, entering the following program and test it:

```php
<?php
    print phpinfo();
?>
```

Save the program as First.php and place the file in your Apache root folder. For example, the following path is a typical one for Windows:

```
c:/Program Files/apache group/apache/htdocs/php
```

The added folder, php, is where to put your First.php. On a Mac, using the built-in PHP, the path is

```
Macintosh HD/Library/WebServer/Documents/php/First.php
```

Next, open a browser and enter type in, http://localhost/php/First.php and press Enter/return. Unlike a regular Web page you have to call the file from a browser. You cannot just double-click it on the desktop—localhost is the server’s name that it runs on. Figure 16-2 show what you will see if everything is installed correctly:
Your installed version may be different, but that information tells you that PHP is installed on your system and ready to go.