Apress™
Books for Professionals by Professionals

Chapter Six: “Object-Oriented PHP”

A Programmer’s Introduction to PHP 4.0

by William Jason Gilmore
ISBN # 1-893115-85-2

info@apress.com

CHAPTER 6

Object-Oriented PHP

If you are familiar with programming strategy, object-oriented programming (OOP) is most likely a part of your daily developmental strategy. If you are new to OOP, after reading this chapter and implementing a few of the examples, you will look at coding in a whole new light. This chapter focuses on OOP and PHP's particular implementation of the OOP strategy, introducing the necessary syntax and providing examples that will allow you to begin building your own OO applications.

The OOP strategy can be best summed up as a shift of developmental focus from an application's functional operations to its data structures. This enables programmers to model real world objects and scenarios in the applications that they write. In particular, the OOP strategy offers three advantages:

- Easier to understand: OOP makes it possible to think of programs in terms of everyday objects.

- Improved reliability and maintenance: Designed properly, OO programs are easily expanded and modified. The modular property makes it possible to independently edit various parts of the program, effectively minimizing the risk of programming errors.

- Faster development cycles: Modularity again plays an important role, as various parts of OO programs can be easily reused, eliminating code redundancy and ultimately resulting in the reduction of unnecessarily repeated coding errors.

These intrinsic advantages of OOP have resulted in major efficiency gains for developers, enabling programmers to develop more powerful, scalable, and efficient applications. Many of these advantages are due to one of OOP's foundational concepts known as *encapsulation*, or *information hiding*. Encapsulation is the concept of hiding various elements in a larger entity, causing the programmer to concentrate on the larger object. This results in the overall reduction of program complexity due to the diminution of unnecessary details.

The concept of encapsulation can be correlated to the typical driver's operation of a car. Most drivers are oblivious to the actual mechanical operations of the automobile, yet are capable of operating the vehicle exactly in the way it was intended to be operated. Knowledge of the inner-workings of the engine, brakes,

and steering is unnecessary, because proper interfacing has been provided to the driver that makes these otherwise highly complex operations automated and easy. The same idea holds true with encapsulation and OOP, as many of these "inner workings" are hidden from the user, allowing the user to focus on the task at hand. OOP makes this possible through the use of classes, objects, and various means of expressing hierarchical relationships between data entities. (Classes and objects are discussed shortly.)

PHP and OOP

Although PHP offers general object-oriented features, it is not yet a full-featured OO language, like C++ or Java, for example. Unlike OOP, PHP does not explicitly offer the following object-oriented characteristics:

- Multiple inheritance
- Constructor chaining (you must call a parent class constructor explicitly if you would like it to execute on construction of a derived class object)
- Class abstraction
- Method overloading
- Operator overloading (because PHP is a loosely typed language; see Chapter 2, "Variables and Data Types," for more info)
- Concepts of private, virtual, and public
- Destructors
- Polymorphism

However, even without these important OO features, you can still benefit from using those OO features that PHP does support. PHP's OO implementation can aid tremendously in packaging your programming functionality. Read on to learn more.

Classes, Objects, and Method Declarations

The *class* is the syntactical foundation of object-oriented programming and can be considered a container of sorts that holds an interrelated set of data and the

data's corresponding functions, better known as *methods* (discussed shortly). A class is a template from which specific instances of the class may be created and used in a program. These instances are also known as *objects*.

One way to grasp the relationship between classes and objects is to consider the class as a general blueprint for a structure. From this blueprint, several structures (or objects) can be built, each sharing the same set of core characteristics (for example, one door, two windows, and a wall thickness). However, each structure is independent of the others in the sense that it is free to change the characteristic values without affecting the values of the others. (For example, one structure might have a wall thickness of five inches, while another has a wall thickness of ten inches.) The important thing to keep in mind is that they all share this characteristic of wall thickness.

A class can also be thought of as a data type (discussed in Chapter 2), much as one would consider a variable entitled `$counter` to be of type int, or a variable entitled `$last_name` to be of type string. One could simultaneously manipulate several objects of type class just as one manipulates several variables of type int. The general format of a PHP class is shown in Listing 6-1.

Listing 6-1: PHP class declaration structure

```
class Class_name {
     var $attribute_1;
     . . .
     var $attribute_N;

     function function1() {
     . . .
     }
     . . .
     function functionN() {
     . . .
     }

} // end Class_name
```

To summarize Listing 6-1, a class declaration must begin with the keyword class, much like a function declaration begins with the keyword function. Each attribute declaration contained in a class must be preceded by the keyword var. An attribute can be of any PHP-supported data type and should be thought of as a variable with minor differences, as you will learn throughout the remainder of this chapter. Following the attributes are the method declarations, which bear a close resemblance to typical function declarations.

NOTE *It is a general convention that OO classes begin with a capital letter, while methods start in lowercase with uppercase separating each word from a multiword function name. Of course, you can use whatever nomenclature you feel most comfortable with; just be sure to choose a standard and stick with it.*

One main use of methods is to manipulate the various attributes constituting the class. However, these attributes are referenced in the methods using a special variable called `$this`. Consider the following example demonstrating the use of this syntax:

```
<?
class Webpage {
     var $bgcolor;

     function setBgColor($color) {
          $this->bgcolor = $color;
     }

     function getBgColor() {
          return $this->bgcolor;
     }

}
?>
```

The `$this` variable is referring to the particular object making use of the method. Because there can be many object instances of a particular class, `$this` is a means of referring to the attribute belonging to the calling (or 'this') object. Furthermore, there are two points regarding this newly introduced syntax worth mentioning:

- The attribute being referenced in the method does *not* have to be passed in as would a functional input parameter.

- A dollar sign ($) precedes only the `$this` variable and *not* the attribute itself, as would be the case with a normal variable.

Creating and Working with Objects

An object is created using the new operator. An object based on the class Webpage can be instantiated as follows:

```
$some_page = new Webpage;
```

The new object `$some_page` now has its own set of attributes and methods specified in the class Webpage. The attribute `$bgcolor` corresponding to this specific object can then be assigned or changed via the predefined method `setBgColor()`:

```
$some_page->setBgColor("black");
```

- Keep in mind that PHP also allows you to retrieve the value by explicitly calling the attribute along with the object name:

```
$some_page->bgcolor;
```

However, this second method of retrieval defeats the purpose of encapsulation, and you should never retrieve a value this way when working with OOP. To better understand why this is the case, take a moment to read the next section.

Why Insufficient Encapsulation Practice Is BAD!

Consider a scenario in which you assign an array as an attribute in a given class. However, instead of calling intermediary methods to control the array (for example, add, delete, modify elements, and so on), you directly call the array whenever needed. Over the period of a month, you confidently design and code a massive "object-oriented" application and revel in the glory of the praise provided to you by your fellow programmers. Ahhhh, a pension plan, paid vacation, and maybe your own office are just around the corner.

But wait, one month after the successful launch of your Web application, your boss decides that arrays aren't the way to go and instead wants all data controlled via a database.

Uh-oh. Because you decided to explicitly manipulate the attributes, you now must go through the code, changing every instance in which you did so to fit the new requirements of a database interface. A time-consuming task to say the least, but also one that could result in the introduction of many new coding errors.

However, consider the result if you had used methods to interface with this data. The only thing you would have to do to switch from an array to a database storage protocol would be to modify the attribute itself and the code contained in

the methods. This modification would result in the automatic propagation of these changes to every part of the code in which the relevant methods are called.

Constructors

Often, just creating a new object is a bit inefficient, as you may need to assign several attributes along with each object. Thankfully, the designers of the OOP strategy took this into consideration, introducing the concept of a *constructor*. A constructor is nothing more than a method that sets particular attributes (and can also trigger methods), simultaneously called when a new object is created. For this concurrent process to occur, the constructor method must be given the same name as the class in which it is contained. Listing 6-2 shows how you might use a constructor method.

Listing 6-2: Using a constructor method

```
<?
class Webpage {
     var $bgcolor;

     function Webpage($color) {
          $this->bgcolor = $color;
     }
}

// call the Webpage constructor
$page = new Webpage("brown");
?>
```

Previously, two steps were required for the class creation and initial attribute assignment, one step for each task. Using constructors, this process is trimmed down to just one step.

Interestingly, different constructors can be called depending on the number of parameters passed to them. Referring to Listing 6-2, an object based on the Webpage class can be created in two ways: You can use the class as a constructor, which will simply create the object, but not assign any attributes, as shown here:

```
$page = new Webpage;
```

Or you can create the object using the predefined constructor, creating an object of class Webpage and setting its bgcolor attribute, as you see here:

```
$page = new Webpage("brown");
```

Destructors

As I've already stated, PHP does not explicitly support destructors. However, you can easily build your own destructor by calling the PHP function `unset()`. This function acts to erase the contents of a variable, thereby returning its resources back to memory. Quite conveniently, `unset()` works with objects in the same way that it does with variables. For example, assume that you are working with the object *$Webpage*. You've finished working with this particular object, so you call:

```
unset($Webpage);
```

This will remove all of the contents of *$Webpage* from memory. Keeping with the spirit of encapsulation, you could place this command within a method called `destroy()` and then call:

```
$Website->destroy();
```

Keep in mind that there really isn't a need to use destructors, unless you are using objects that are taking up considerable resources; all variables and objects are automatically destroyed once the script finishes execution.

Inheritance and Multilevel Inheritance

As you are already aware, a class is a template for a real world object that acts as a representation of its characteristics and functions. However, you probably know of instances in which a particular object could be a subset of another. For example, an automobile could be considered a subset of the category vehicle because airplanes are also considered vehicles. Although each vehicle type is easily distinguishable from the other, assume that there exists a core set of characteristics that all share, including number of wheels, horsepower, current speed, and model. Of course, the values assigned to the attributes of each may differ substantially, but nonetheless these characteristics do exist. Consequently, it could be said that the subclasses automobile and airplane both inherit this core set of characteristics from a superclass known as vehicle. The concept of a class inheriting the characteristics of another class is known as *inheritance*.

Inheritance is a particularly powerful programming mechanism because it can eliminate an otherwise substantial need to repeat code that could be shared between data structures, such as the shared characteristics of the various vehicle types mentioned in the previous paragraph. The general PHP syntax used to inherit the characteristics of another class follows:

```
class Class_name2 extends Class_name1 {

     attribute declarations;

     method declarations;

}
```

The notion of a class extending another class is just another way of stating that `Class_name2` inherits all of the characteristics contained in `Class_name1` and in turn possibly extends the use and depth of the `Class_name1` characteristics with those contained in `Class_name2`.

Other than for reason of code reusability, inheritance provides a second important programming advantage: it reduces the possibility of error when a program is modified. Considering the class inheritance hierarchy shown in Figure 6-1, realize that a modification to the code contained in auto will have no effect on the code (and data) contained in airplane, and vice versa.

CAUTION *A call to the constructor of a derived class does not imply that the constructor of the parent class is also called.*

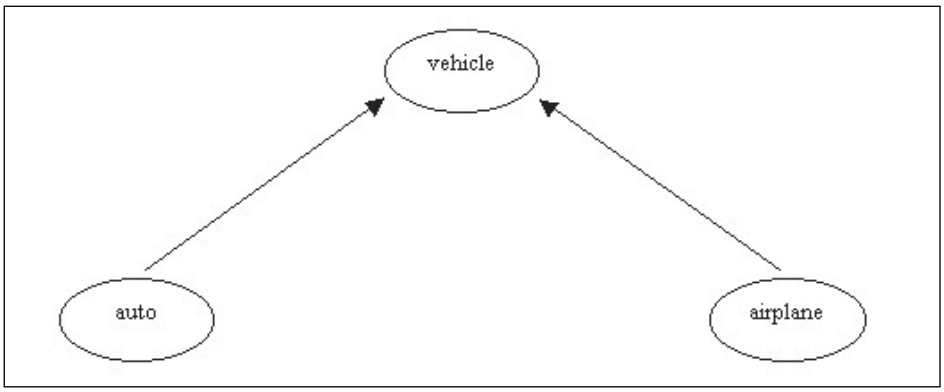

Figure 6-1. Relationship diagram of the various vehicle types

Let's use Listing 6-3 to build the code needed to accurately represent Figure 6-1.

Listing 6-3: Using inheritance to efficiently represent various vehicle types

```
<?
class Vehicle {
     var $model;
     var $current_speed;

     function setSpeed($mph) {
          $this->current_speed = $mph;
     }

     function getSpeed() {
          return $this->current_speed;
     }
} // end class Vehicle

class Auto extends Vehicle {
    var $fuel_type;

    function setFuelType($fuel) {
          $this->fuel_type = $fuel;
    }

     function getFuelType() {
          return $this->fuel_type;
     }

} // end Auto extends Vehicle

class Airplane extends Vehicle {
     var $wingspan;

     function setWingSpan($wingspan) {
          $this->wingspan = $wingspan;
     }

     function getWingSpan() {
          return $this->wingspan;
     }
} // end Airplane extends Vehicle
```

We could then instantiate various objects as follows:

```
$tractor = new Vehicle;
$gulfstream = new Airplane;
?>
```

Two objects have been created. The first, `$tractor`, is a member of the Vehicle class. The second, `$gulfstream`, is a member of the Airplane class, possessing the characteristics of the Airplane and the Vehicle class.

CAUTION *The idea of a class inheriting the properties of more than one parent class is known as* multiple inheritance. *Unfortunately, multiple inheritance is not possible in PHP. For example, you cannot do this in PHP:*

```
Class Airplane extends Vehicle extends Building . . .
```

Multilevel Inheritance

As programs increase in size and complexity, you may need several levels of inheritance, or classes that inherit from other classes, which in turn inherit properties from other classes, and so on. Multilevel inheritance further modularizes the program, resulting in an increasingly maintainable and detailed program structure. Continuing along with the Vehicle example, a larger program may demand that an additional class be introduced between the Vehicle superclass to further categorize the class structure. For example, the class Vehicle may be divided into the classes land, sea, and air, and then specific instances of each of those subclasses can be based on the medium in which the vehicle in question travels. This is illustrated in Figure 6-2.

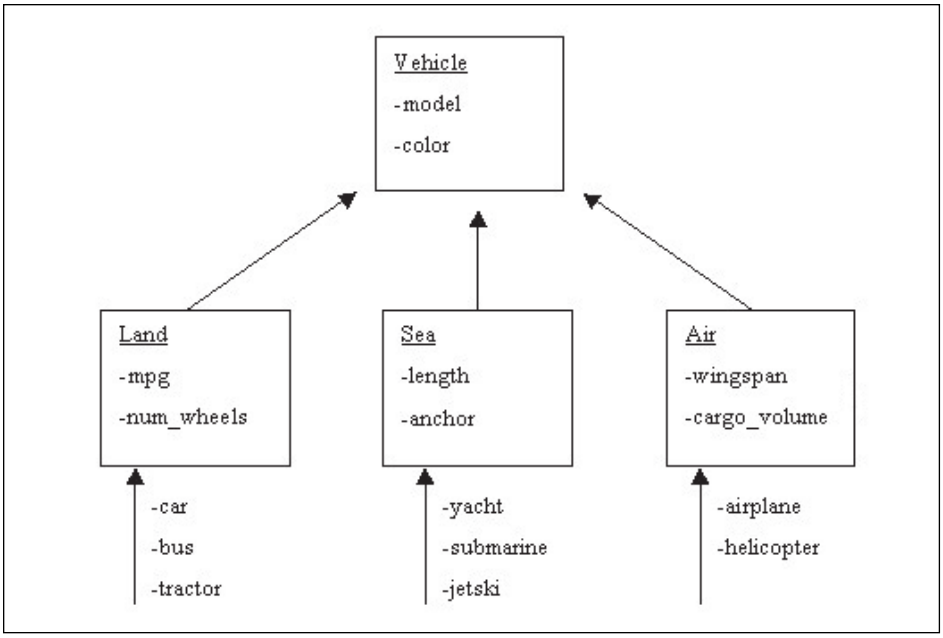

Figure 6-2. Multilevel inheritance model of the Vehicle superclass

Consider the brief example in Listing 6-4, which serves to highlight a few important aspects of multilevel inheritance in regard to PHP.

Listing 6-4: Making use of multilevel inheritance

```
<?
class Vehicle {
   Attribute declarations. . .
   Method declarations. . .
}

class Land extends Vehicle {
   Attribute declarations. . .
   Method declarations. . .
}

class Car extends Land {
   Attribute declarations. . .
   Method declarations. . .
}
$nissan = new Car;
?>
```

Once instantiated, the object `$nissan` has at its disposal all of the attributes methods available in Car, Land, and Vehicle. As you can see, this is an extremely modular structure. For example, sometime throughout the lifecycle of the program, you may wish to add a new attribute to Land. No problem: just modify the Land class accordingly, and that attribute becomes immediately available to itself and Car, without affecting the functionality of any other class. This idea of code modularity and flexibility is indeed one of the great advantages of OOP.

NOTE *Keep in mind that although a class can inherit characteristics from a chain of parents, the parents' constructors are not called automatically when you instantiate an object from the inheriting class. These constructors become methods for the child class.*

Class Abstraction

Sometimes it is useful to create a class that will never be instantiated and instead will just act as the base for a derived class. This kind of class is known as an *abstract class*. An abstract class is useful when a program designer wants to ensure that certain functionality is available in any subsequently derived classes based on that abstract class.

PHP does not offer explicit class abstraction, but there is an easy workaround. Just create a default constructor and place a call to `die()` in it. Referring to the classes in Listing 6-4, chances are you will never wish to instantiate the Land or Vehicle classes, because neither could represent a single entity. Instead, you would extend these classes into a real world object, such as the car class. Therefore, to ensure that Land or Vehicle is never directly instantiated, place the `die()` call in each, as seen in Listing 6-5.

Listing 6-5: Building abstract classes

```
<?
class Vehicle {
               Attribute declarations. . .
               function Vehicle() {
                              die("Cannot create Abstract Vehicle class!");
               }
               Other Method declarations. . .
}

class Land extends Vehicle {
               Attribute declarations. . .
               function Land() {
                              die("Cannot create Abstract Land class!");
               }
               Other Method declarations. . .
}

class car extends Land {
               Attribute declarations. . .
               Method declarations. . .
}
?>
```

Therefore, any attempt to instantiate these abstract classes results in an appropriate error message and program termination.

Method Overloading

Method overloading is the practice of defining multiple methods with the same name, but each having a different number or type of parameters. This too is not a feature supported by PHP, but an easy workaround exists, as shown in Listing 6-6.

Listing 6-6: Method overloading

```
<?
class Page {
     var $bgcolor;
     var $textcolor;

     function Page() {
          // Determine the number of arguments
          // passed in, and create correct method name
          $name = "Page".func_num_args();
          // Call $name with correct number of arguments passed in
          if ( func_num_args() == 0 ) :
               $this->$name();
          else :
               $this->$name(func_get_arg(0));
          endif;
     }

     function Page0() {
          $this->bgcolor = "white";
          $this->textcolor = "black";
          print "Created default page";
     }

     function Page1($bgcolor) {
          $this->bgcolor = $bgcolor;
          $this->textcolor = "black";
          print "Created custom page";
     }
}
$html_page = new Page("red");
?>
```

In this example, a new object entitled `$html_page` is created, with one argument passed in. Since a default constructor has been created (`Page()`), the instantiation begins there. However, this default constructor is simply used to determine exactly which of the other constructor methods (`Page0()` or `Page1()`) is called. This is determined by making use of the `func_num_args()` and `func_get_arg()` functions, which count the number of arguments and retrieve the arguments, respectively.

Obviously, this is not method overloading as it was intended to be implemented, but it does the job for those of you who cannot live without this important OOP feature.

Class and Object Functions

PHP offers a number of predefined class and object functions, which are discussed in the following sections. All can be useful, particularly for interface development, code administration, and error checking.

get_class_methods()

The `get_class_methods()` function returns an array of methods defined by the class specified by class_name. The syntax is:

```
array get_class_methods (string class_name)
```

A simple example of how `get_class_methods()` is used is in Listing 6-7.

Listing 6-7: Retrieving the set of methods available to a particular class

```
<?
. . .
class Airplane extends Vehicle {
     var $wingspan;

     function setWingSpan($wingspan) {
          $this->wingspan = $wingspan;
     }

     function getWingSpan() {
          return $this->wingspan;
     }
}

$cls_methods = get_class_methods(Airplane);
// $cls_methods will contain an array of all methods
// declared in the classes "Airplane" and "Vehicle".
?>
```

As you can see by following the code in Listing 6-7, `get_class_methods()` is an easy way to obtain a listing of all supported methods of a particular class.

get_class_vars()

The get_class_vars() function returns an array of attributes defined in the class specified by class_name. Its syntax is:

```
array get_class_vars (string class_name)
```

An example of how get_class_vars() is used is in Listing 6-8.

Listing 6-8: Using get_class_vars() to create $attribs

```
<?
class Vehicle {
     var $model;
     var $current_speed;
}
class Airplane extends Vehicle {
     var $wingspan;
}

$a_class = "Airplane";

$attribs = get_class_vars($a_class);
// $attribs = array ( "wingspan", "model", "current_speed")
?>
```

Therefore, the variable $attribs is created and becomes an array containing all available attributes of the class Airplane.

get_object_vars()

The get_object_vars() function returns an array containing the properties of the attributes assigned to the object specified by obj_name. Its syntax is:

```
array get_object_vars (object obj_name)
```

An example of how get_object_vars() is used is in Listing 6-9.

Listing 6-9: Obtaining object variables

```
<?
class Vehicle {
     var $wheels;
}

class Land extends Vehicle {
     var $engine;
}

class car extends Land {
     var $doors;

     function car($doors, $eng, $wheels) {
          $this->doors = $doors;
          $this->engine = $eng;
          $this->wheels = $wheels;
     }

     function get_wheels() {
          return $this->wheels;
     }

}

$toyota = new car(2,400,4);

$vars = get_object_vars($toyota);

while (list($key, $value) = each($vars)) :

   print "$key ==> $value <br>";

endwhile;
// displays:
// doors ==> 2
// engine ==> 400
// wheels ==> 2
?>
```

Using `get_object_vars()` is a convenient way to quickly obtain all of the attribute/value mappings of a particular object.

method_exists()

The method_exists() function checks to see if a particular method (denoted by method_name), exists in the object specified by obj_name, returning true if it exists, or false if it does not. Its syntax is:

```
bool method_exists (object obj_name, string method_name)
```

An example of the usage of method_exists() is in Listing 6-10.

Listing 6-10: Using method_exists() to verify an object/method mapping.

```
    <?
 class Vehicle {
     . . .
     }

     class Land extends Vehicle {
          var $fourWheel;
          function setFourWheelDrive() {
               $this->fourWeel = 1;
          }
     }

     //  create object named $car
     $car = new Land;

     // if method "fourWheelDrive" is a part of classes "Land" or "Vehicle",
     // then the call to method_exists() will return true;
     // Otherwise false will be returned.
     // Therefore, in this case, method_exists() will return true.

     if (method_exists($car, "setfourWheelDrive")) :
          print "This car is equipped with 4-wheel drive";
     else :
          print "This car is not equipped with 4-wheel drive";
     endif;
?>
```

In Listing 6-10, the function method_exists() is used to verify whether or not the object $car has access to the method setFourWheelDrive(). If it does, true is returned, and the appropriate message is displayed. Otherwise, false is returned, and a message stating that four-wheel drive is not available with that particular object.

get_class()

The get_class() function returns the name of the class from which the object specified by obj_name is instantiated. The syntax is:

```
string get_class(object obj_name);
```

An example of how get_class() is implemented is in Listing 6-11.

Listing 6-11: Using get_class()to return the name of an instantiation class

```
<?
class Vehicle {
     . . .
}

class Land extends Vehicle {
. . .
}

// create object named $car
$car = new Land;

// $class_a is assigned "Land"
$class_a = get_class($car);
?>
```

Simply enough, the variable $class_a is assigned the name of the class from which the object $car was derived.

get_parent_class()

The get_parent_class() function returns the name, if any, of the parent class of the object specified by objname. The syntax is:

```
string get_parent_class(object objname);
```

Listing 6-12 illustrates usage of get_parent_class().

Listing 6-12: Name of the class parent returned using get_parent_class()

```
<?
class Vehicle {
. . .
}

class Land extends Vehicle {
. . .
}

// create object named $car
$car = new Land;

// $parent is assigned "Vehicle"
$parent = get_parent_class($car);
?>
```

As you would expect, the call to `get_parent_class()` assigns the value "Vehicle" to the variable $parent.

is_subclass_of()

The `is_subclass_of()` function ensures whether or not an object was created from a class whose parent is specified by class_name, returning true if it was, and false otherwise. Its syntax is:

```
bool is_subclass_of (object obj, string class_name)
```

Listing 6-13 illustrates proper usage of `is_subclass_of()`.

Listing 6-13: Using is_subclass_of() to determine whether an object was created from a class derived from a specific parent class

```
<?
class Vehicle {
. . .
}

class Land extends Vehicle {
. . .
}
$auto = new Land;
// $is_subclass receives the value "true"
$is_subclass = is_subclass_of($auto, "Vehicle");
?>
```

In Listing 6-13, the variable `$is_subclass` is used to determine whether the object `$auto` is derived from a subclass of the parent class Vehicle. In fact, `$auto` is derived from Land; therefore, `$is_subclass` will receive the boolean value true.

get_declared_classes()

The `get_declared_classes()` function returns an array of all defined classes, as shown in Listing 6-14. Its syntax is:

```
array get_declared_classes()
```

Listing 6-14: Retrieving all defined classes with get_declared_classes()

```
<?
class Vehicle {
. . .
}
class Land extends Vehicle {
. . .
}
// $declared_classes = array("Vehicle", "Land")
$declared_classes = get_declared_classes();
?>
```

What's Next?

This chapter introduced you to several of object-oriented programming's basic concepts, concentrating on how these concepts are applied to the PHP programming language. In particular, the following subjects were discussed in detail:

- Introduction to object-oriented programming
- Classes, objects, and methods
- Inheritance and multilevel inheritance
- Class abstraction
- Method overloading
- PHP's class and object functions

Although not overly complicated, the object-oriented programming strategy usually requires of the programmer an initial exploration period before all of the concepts are really understood. However, I guarantee that the extra time you take to understand these notions will add an entirely new level of efficiency and creativity to your programming repertoire.