Part I

Fundamentals of Professional Development

1

Introduction to Object-Oriented Programming

Object-oriented (OO) software development can be a confusing topic for developers who create primarily procedural code, but it doesn't need to be. In this chapter, you'll explore some of the basic theory behind OO, and learn its (sometimes daunting) multisyllabic terminology. You'll also learn why you should be interested in OO techniques, how they can greatly improve the speed with which you develop complex applications, and see the ease with which you can modify those applications.

The next few chapters expand on the ideas presented here and familiarize you with some slightly more advanced topics. If you have already had exposure to OO development outside of PHP 6, you can probably skip this chapter and the next. However, this material serves as a good review, so it is recommended that you read it through.

What Is Object-Oriented Programming?

Object-oriented programming (OOP) requires a different way of thinking about how you construct your applications. Objects enable you to more closely model in code the real-world tasks, processes, and ideas that your application is designed to handle. Instead of thinking about an application as a thread of control that passes chunks of data from one function to the next, an OOP approach enables you to model the application as a set of collaborating objects that independently handle certain activities.

As an analogy, when a house is being constructed, the plumbers deal with the pipes, and the electricians deal with the wires. The plumbers don't need to know whether the circuit in the bedroom is 10 amps or 20 amps. They must concern themselves only with their own activities. A general contractor ensures that each subcontractor is completing the work that needs to be accomplished, but isn't necessarily interested in the particulars of each task. An OOP approach is similar in that each object hides from the others the details of its implementation. How it does its

job is irrelevant to the other components of the system. All that matters is the service that the object is able to provide.

The concepts of classes and objects, and the ways in which you can leverage these ideas in the development of software, are the fundamental ideas behind OOP. This is, in a sense, the opposite of procedural programming, which is programming using functions and global data structures. As you'll see, an OO approach provides significant benefits over procedural programming and (with the new implementation of OO support that first appeared in PHP5, and was further improved in PHP6) some large performance boosts as well.

OOP Advantages

One of the main benefits of OOP is the ease with which you can translate individual business requirements into individual modules of code. Because an OOP approach enables you to model your application based on the idea of real-world objects, you can often identify a direct correlation between people, things, and concepts, and equivalent classes. These classes have the same properties and behaviors as the real-world concepts they represent, which helps you to quickly identify what code must be written and how different parts of the application must interact.

A second benefit of OOP is code reuse. You frequently need the same types of data in different places in the same application. For example, an application that enables a hospital to manage its patient records would definitely need a class called `Person`. A number of people are involved in patient care — the patient, the doctors, the nurses, hospital administrators, insurance claims people, and so on. At each step in the care of the patient, that patient's record requires a note about which person was performing a given action (such as prescribing medicine, cleaning wounds, or sending a bill to an insurance carrier) and verifying that the person is allowed to perform that action. By defining a generic class called `Person` that encompasses all the properties and methods common to all these people, you get an enormous amount of code reuse that isn't always possible in a procedural approach to programming.

What about other applications? How many applications can you think of that, at some point, handle information about individuals? Probably quite a few. A well-written `Person` class could easily be copied from one project to another with little or no change, instantly giving you all the rich functionality required for dealing with information about people that you developed previously. This is one of the biggest benefits of an OO approach — the opportunities for code reuse within a given application, as well as across different projects.

Another OOP advantage comes from the modularity of classes. If you discover a bug in your `Person` class, or you want to add to or change the way that class functions, you have only one place to go. All the functionality of that class is contained in a single file. Any processes of the application that rely on the `Person` class are immediately affected by changes to it. This can vastly simplify the search for bugs, and makes the addition of features a relatively painless task.

A Real-World Example

They might seem trivial in a smaller application, but in a more complex software architecture, the benefits of modularity can be enormous. One of the authors of this book worked on a project involving more than 200,000 lines of procedural PHP code. Easily, 65 percent of the time spent fixing bugs was devoted to uncovering where certain functions were located and determining which data interacted with

which functions. A subsequent rewrite of that software in an OO architecture resulted in dramatically less code. Had the application been implemented in such a manner in the first place, it would have resulted in not only less development time from the outset, but also the discovery of fewer bugs (the smaller the amount of code, the fewer the opportunities for problems) and a faster turnaround time on bug fixes.

Because an OO approach easily lends itself to a system of clearly documenting the structure of an application (see Chapter 2), learning the structure of an existing application is much easier when you are new to the development team. In addition, you have a framework to aid you in determining the appropriate location for new functionality you might develop.

Larger projects often have a multimember software development team, usually composed of programmers with varying degrees of ability. Here, too, an OO approach has significant benefits over a procedural approach. Objects hide the details of their implementation from the users of those objects. Instead of needing to understand complex data structures and all the quirks of the business logic, junior members of the team can (with just a little documentation) begin using objects created by more experienced members. The objects themselves are responsible for triggering changes to data or the state of the system.

When the large application mentioned previously was still written using procedural code, new members of the software development team often took up to two months to learn enough about the application to be productive. After the software was rebuilt using objects, new members of the team usually took no more than a couple of days to begin making substantial additions to the code base. They were able to use even the most complex objects quickly because they did not need to fully understand the particulars of how the functionality contained within those objects was implemented.

Now that you have a good idea about why you should consider using an OO paradigm as your programming method of choice, you should read the next few sections to gain a better understanding of the fundamental concepts behind OO. If all goes well, through the course of the next two chapters you will begin to see the benefits of this approach for yourself.

Understanding OOP Concepts

This section introduces the primary concepts of object-oriented programming and explores how they interact; Chapter 3 looks at the specifics of implementing them in PHP6. This chapters covers the following topics:

- *Classes* — The "blueprints" for an object and the actual code that defines the properties and methods
- *Objects* — Running instances of a class that contain all the internal data and state information needed for your application to function
- *Inheritance* — The ability to define a class of one kind as being a subtype of a different kind of class (much the same way a square is a kind of rectangle)
- *Polymorphism* — Allows a class to be defined as being a member of more than one category of classes (just as a car is "a thing with an engine" and "a thing with wheels")

- *Interfaces* — A way of specifying that an object is capable of doing something without actually defining how it is to be done (e.g., a dog and a human are "things that walk," but they do it very differently)
- *Encapsulation* — The ability of an object to protect access to its internal data

If any of these terms seem difficult to understand, don't worry. The material that follows will clarify everything. Your newfound knowledge may even completely change the way you approach your software development projects.

Classes

In the real world, objects have characteristics and behaviors. A car has a color, a weight, a manufacturer, and a gas tank of a certain volume. Those are its characteristics. A car can accelerate, stop, signal for a turn, and sound the horn. Those are its behaviors. Those characteristics and behaviors are common to all cars. Although two particular cars in the same parking lot may have different colors, all cars have a color. Using a construct known as a *class*, OOP enables you to establish the idea of a car as being something with all those characteristics. A class is a unit of code (composed of variables and functions) that describes the characteristics and behaviors of all the members of a set. A class called `Car` would describe the properties and methods common to all cars.

In OO terminology, the characteristics of a class are known as its *properties*. Properties have a name and a value. Some allow their value to be changed; others do not. For example, in the `Car` class, you would probably have such properties as `color` and `weight`. Although the color of the car can be changed by giving it a new paint job, the tare weight of the car (without cargo or passengers) is a fixed value.

Some properties represent the *state* of the object. State refers to those characteristics that change because of certain events but are not necessarily directly modifiable on their own. In an application that simulates vehicle performance, the `Car` class might have a property called `velocity`. The velocity of the car is not a value that can be changed on its own, but rather is a by-product of the amount of fuel being sent to the engine, the performance characteristics of that engine, and the terrain over which the car is traveling.

The behaviors of a class are known as its *methods*. Methods of classes are syntactically equivalent to functions found in traditional procedural code. Just like functions, methods can accept any number of parameters, each of any valid data type. Some methods act on external data passed to them as parameters, but they can also act on the properties of their object, either using those properties to inform actions made by the method (such as when a method called `accelerate` examines the remaining amount of fuel to determine whether the car is capable of accelerating) or to change the state of the object by modifying values such as the velocity of the car.

Objects

To begin with, you can think of a class as a blueprint for constructing an object. In much the same way that many houses can be built from the same blueprint, you can build multiple instances of an object from its class; but the blueprint doesn't specify details such as the color of the walls or the type of flooring. It merely specifies that those things will exist. Classes work much the same way. The class specifies the behaviors and characteristics the object will have, but not necessarily the values of those characteristics. An object is a concrete entity constructed using the blueprint provided by a class. The

idea of *a* house is analogous to a class. *Your* house (a specific instance of the idea of a house) is analogous to an object.

With a blueprint in hand and some building materials, you can construct a house. In OOP, when you use the class to build an object, this process is known as *instantiation*. Instantiating an object requires two things:

- A memory location into which to load the object. This is automatically handled for you by PHP.
- The data that will populate the values of the properties. This data can come from a database, a flat text file, another object, or some other source.

A class can never have property values or state. Only objects can. You must use the blueprint to build the house before you can give it wallpaper or vinyl siding. Similarly, you must instantiate an object from the class before you can interact with its properties or invoke its methods. Classes are manipulated at design time when you make changes to the methods or properties. Objects are manipulated at run-time when values are assigned to their properties, and their methods are invoked. The problem of when to use the word *class* and when to use the word *object* is something that often confuses those new to OOP.

After an object is instantiated, it can be put to work implementing the business requirements of the application. Let's look at exactly how to do that in PHP.

Creating a Class

Starting with a simple example, save the following in a file called `class.Demo.php`:

```
<?php

  class Demo {

  }

?>
```

There you have it — the `Demo` class. Although not terribly exciting just yet, this is the basic syntax for declaring a new class in PHP. Use the keyword `class` to let PHP know you're about to define a new class. Follow that with the name of the class and braces to indicate the start and end of the code for that class.

> *It's important to have a clearly defined convention for organizing your source code files. A good rule to follow is to put each class into its own file and to name that file* `class.[ClassName].php`.

You can instantiate an object of type `Demo` like this:

```
<?php

  require_once('class.Demo.php');

  $objDemo = new Demo();

?>
```

To instantiate an object, first ensure that PHP knows where to find the class declaration by including the file containing your class (`class.Demo.php` in this example); then invoke the `new` operator and supply the name of the class, followed by opening and closing parentheses. The return value of this statement is assigned to a new variable, `objDemo` in this example. Now you can invoke the `$objDemo` object's methods and examine or set the value of its properties — if it actually has any.

Even though the class you've created doesn't do much of anything just yet, it's still a valid class definition.

Adding a Method

The `Demo` class isn't particularly useful if it isn't able to do anything, so let's look at how you can create a method. Remember, a method of a class is basically just a function. By coding a function inside the braces of your class, you're adding a method to that class. Here's an example:

```
<?php

  class Demo {
    function sayHello($name) {
      print "Hello $name!";
    }
  }

?>
```

An object derived from your class is now capable of printing a greeting to anyone who invokes the `sayHello` method. To invoke the method on your `$objDemo` object, you need to use the operator `->` to access the newly created function:

```
<?php

  require_once('class.Demo.php');

  $objDemo = new Demo();

  $objDemo->sayHello('Steve');

?>
```

The object is now capable of printing a friendly greeting. The `->` operator is used to access all methods and properties of your objects.

For readers who have had exposure to OOP in other programming languages, note that the `->` operator is always used to access the methods and properties of an object. PHP does not use the dot operator (`.`) in its OO syntax at all.

Adding a Property

Adding a property to your class is just as easy as adding a method. You simply declare a variable inside the class to hold the value of the property. In procedural code, when you want to store some value, you assign that value to a variable. In OOP, when you want to store the value of a property, you also use a variable. This variable is declared at the top of the class declaration, inside the braces that bracket the

class's code. The name of the variable is the name of the property. If the variable is called `$color`, you will have a property called `color`.

Open the `class.Demo.php` file and add the following highlighted code:

```
<?php

  class Demo {
    public $name;

    function sayHello() {
      print "Hello $this->name!";
    }
  }
?>
```

This new variable, called `$name`, is all you have to do to create a property of the `Demo` class called `name`. To access this property, you use the same `->` operator as that of the previous example, along with the name of the property. The rewritten `sayHello` method shows how to access the value of this property.

Create a new file called `testdemo.php` and add the following:

```
<?php

  require_once('class.Demo.php');

  $objDemo = new Demo();
  $objDemo->name = 'Steve';

  $objAnotherDemo = new Demo();
  $objAnotherDemo->name = 'Ed';

  $objDemo->sayHello();
  $objAnotherDemo->sayHello();

?>
```

Save the file and then open it in your Web browser. The strings "Hello Steve!" and "Hello Ed!" print to the screen.

The keyword `public` is used to let the class know that you want to have access to the following variable from outside the class. Some member variables of the class exist only for use by the class itself and should not be accessible to external code; these variables are declared as `private` or `protected` (more on that later). In this example, you want to be able to set and retrieve the value of the property `name`. Note that the way the `sayHello` method works has changed. Instead of taking a parameter, it now fetches the `name` value from the property.

You use the variable `$this` so that the object can get information about itself. You might have multiple objects of a class, for example, and because you don't know in advance what the name of an object variable will be, the `$this` variable enables you to refer to the current instance.

In the previous example, the first call to `sayHello` prints `"Steve"` and the second call prints `"Ed"`. This is because the `$this` variable allows each object to access its own properties and methods without having to know the name of the variable that represents it in the exterior application. Previously, you learned that some properties influence the action of certain methods, such as the example in which the `accelerate` method of the `Car` class needs to examine the amount of fuel remaining. The code inside `accelerate` would use code such as `$this->amountOfFuel` to access this property.

> *When accessing properties, you need only one $. The syntax is* `$obj->property`*, not* `$obj->$property`*. This fact often causes confusion for those new to PHP. The property variable is declared as public* `$property` *and accessed using* `$obj->property`*.*

In addition to the variables that store the values for the properties of the class, other variables may be declared for use by the internal operations of the class. Both kinds of data are collectively referred to as the class's *internal member variables*. Some of these are accessible to code outside the class in the form of properties. Others are not accessible and are strictly for internal housekeeping. For example, if the `Car` class needed to get information from a database for whatever reason, it might keep a database connection handle in an internal member variable. This database connection handle is obviously not a property of the car, but rather something the class needs to carry out certain operations.

Protecting Access to Member Variables

As the previous example shows, you can set the value of the `name` property to just about anything you want — including an object, an array of integers, a file handle, or any other nonsensical value. However, you don't get an opportunity to do any sort of data validation or update any other values when the `name` property is set.

To work around this problem, always implement your properties in the form of functions called `get[ property name ]` and `set[ property name ]`. Such functions are known as *accessor methods*, and are demonstrated in the following example.

Change `class.Demo.php` as shown here:

```
<?php
class Demo {
    private $_name;

    public function sayHello() {
      print "Hello " . $this->getName() . "!"
    }

    public function getName() {
      return $this->_name;
    }

    public function setName($name) {
      if(!is_string($name) || strlen($name) == 0) {
        throw new Exception("Invalid name value!");
      }

      $this->_name = $name;
    }
}
?>
```

Edit `testdemo.php` as shown here:

```
<?php

require_once('class.Demo.php');

$objDemo = new Demo();
$objDemo->setName('Steve');
$objDemo->sayHello();

$objDemo->setName(37); //would trigger an error

?>
```

As you can see, the member access level of `name` has changed from `public` to `private` and has been prefixed with an underscore. The underscore is a recommended naming convention to indicate private member variables and functions; however, it is merely a convention — PHP does not require it. The keyword `private` protects code outside the object from modifying this value. Private internal member variables are not accessible from outside the class. Because you can't access these variables directly, you're forced to use the `getName()` and `setName()` accessor methods to obtain this information, ensuring that your class can examine the value before allowing it to be set.

In this example, an exception is thrown if an invalid value is supplied for the `name` property. Additionally, the `public` access specifier for the functions has been added. Public is the default visibility level for any member variables or functions that do not explicitly set one, but it is good practice to always explicitly state the visibility of all the members of the class.

A member variable or method can have three different levels of visibility: public, private, and protected. *Public members* are accessible to any and all code. *Private members* are accessible only to the class itself. These are typically items used for internal housekeeping, and might include such things as a database connection handle or configuration information. *Protected members* are available to the class itself and to classes that inherit from it. (Inheritance is defined and discussed in detail later in this chapter.)

By creating accessor methods for all your properties, you make it much easier to add data validation or new business logic, or make other changes to your objects later. Even if the current business requirements for your application involve no data validation of a given property, you should still implement that property with `get` and `set` functions so that you can add validation or business logic functionality in the future.

> *Always use accessor methods for your properties. Changes to business logic and data validation requirements in the future will be much easier to implement.*

Initializing Objects

For many of the classes you will create, you will need to do some special setup when an object of that class is first instantiated. You might need to fetch some information from a database or initialize some property values, for example. By creating a special method called a *constructor*, implemented in PHP using a function called `_construct()`, you can perform any activities required to instantiate the object. PHP will automatically call this special function when instantiating the object.

For example, you could rewrite the `Demo` class in the following way:

```
<?php
class Demo {
  private $name;

  public function __construct($name) {
    $this->name = $name;
  }
  function sayHello() {
    print "Hello $this->name!";
  }
}
?>
```

The `_construct` function will be automatically invoked when you instantiate a new object of class `Demo`. Note that you will need to update `testdemo.php` to pass the name in the constructor, rather than in the setter method.

> *In PHP 4, object constructors were functions with the same name as the class. PHP version 5 changed this to use a unified constructor scheme. For backward compatibility, PHP first looks for a function called* `__ construct`*, but if none is found, it will still look for a function with the same name as the class (* `public function Demo()` *in the preceding example). While this backward compatibility has been maintained in PHP 6, there is no guarantee it will be preserved in future versions.*

If you have a class that does not require any special initialization code to be run, you don't need to create a constructor. As you saw in the first version of the `Demo` class, PHP automatically does what it needs to do to create that object. Create a constructor function only when you need one.

Destroying Objects

The object variables that you create are removed from system memory when the requested page has completed running, when the variable falls out of scope, or when it is explicitly set to null. In PHP 6, you can trap the destruction of the object and take actions when that happens. To do so, create a function called `__destruct` with no parameters. Before the object is destroyed, this function is called automatically, if it exists.

Calling this function gives you the opportunity to perform any last-minute clean-up (such as closing file handles or database connections that might have been opened by the class), or any other last-minute housekeeping that might be needed before the object is destroyed.

The following example fetches the properties of an object from a database. If any properties of the object are changed, they are automatically saved back to the database when the object is destroyed. This eliminates the need to explicitly call a save method. The destructor also closes the open database connection handle.

As with most of the database examples in this book, this one uses PostgreSQL as its platform. The authors firmly believe that the advanced features, transaction support, and robust stored procedure mechanism of PostgreSQL make it a superior alternative to MySQL and other open-source relational database management systems (RDBMSs) for large-scale enterprise software development. If you don't

have a PostgreSQL environment at your disposal, or if you prefer MySQL (as many do), feel free to make the appropriate modifications for the database platform you use.

Create a table called `"widget"` using the following SQL statement:

```
CREATE TABLE "widget" (
  "widgetid" SERIAL PRIMARY KEY NOT NULL,
  "name" varchar(255) NOT NULL,
  "description" text
);
```

Insert some data:

```
INSERT INTO "widget" ("name", "description")
VALUES('Foo', 'This is a footacular widget!');
```

Create a file called `class.Widget.php` and enter the following code:

```
<?php

class Widget {

    private $id;
    private $name;
    private $description; private $hDB;
    private $needsUpdating = false;

    public function __construct($widgetID) {
      //The widgetID parameter is the primary key of a
      //record in the database containing the information
      //for this object

      //Create a connection handle and store it in a private member variable
      //This code assumes the DB is called "parts"
      $this->hDB = pg_connect('dbname=parts user=postgres');
      if(! is_resource($this->hDB)) {
        throw new Exception('Unable to connect to the database.');
      }

      $sql = "SELECT \"name", \"description\" FROM widget WHERE widgetid =
              $widgetID";
      $rs = pg_query($this->hDB, $sql);
      if(! is_resource($rs)) {
        throw new Exception("An error occurred selecting from the database.");
      }

      if(! pg_num_rows($rs)) {
        throw new Exception('The specified widget does not exist!');
      }
      $data = pg_fetch_array($rs);
      $this->id = $widgetID;
```

```
        $this->name = $data['name'];

        $this->description = $data['description'];
      }

      public function getName() {
        return $this->name;
      }

      public function getDescription() {
        return $this->description;
      }

      public function setName($name) {
        $this->name = $name;
        $this->needsUpdating = true;
  }
      public function setDescription($description) {
        $this->description = $description;
        $this->needsUpdating = true;
      }

     public function __destruct() {
       if($this->needsUpdating) {

         $sql = 'UPDATE "widget" SET ';
         $sql .= "\"name\" = '" . pg_escape_string($this->name) . "', ";
         $sql .= "\"description\" = '" .
         pg_escape_string($this->description) . "'";
         $sql .= "WHERE widgetID = " . $this->id;

         $rs = pg_query($this->hDB, $sql);
       }

       //We're done with the database. Close the connection handle.
       pg_close($this->hDB);
    }
  }
  ?>
```

The constructor to this object opens a connection to a database called `parts` using the default super-user account `postgres`. This connection handle is preserved in a private member variable for use later. The ID value passed as a parameter to the constructor is used to construct a SQL statement that fetches the information for the widget with the specified primary key in the database. The data from the database is then assigned to private member variables for use with the `get` and `set` functions. Note that if anything should go wrong, the constructor throws exceptions, so be sure to wrap any attempts to construct a `Widget` object in `try... catch` blocks.

The two accessor methods, `getName()` and `getDescription()`, enable you to fetch the values of the private member variables. Similarly, the `setName()` and `setDescription()` methods enable you to assign a new value to those variables. Note that when a new value is assigned, the `needsUpdating` value is set to `true`. If nothing changes, then nothing needs to be updated.

To test this, create a file called `testWidget.php` with the following content:

```
<?php

require_once('class.Widget.php');

  try {
    $objWidget = new Widget(1);
    print "Widget Name: " . $objWidget->getName() . "<br>\n";

    print "Widget Description: " . $objWidget->getDescription() . "<br>\n";
    $objWidget->setName('Bar');
    $objWidget->setDescription('This is a bartacular widget!');

  } catch (Exception $e) {
     die("There was a problem: " . $e->getMessage());
  }

?>
```

Access this file in your Web browser. The first time it runs, the output should be something like the following:

```
Widget Name: Foo
Widget Description: This is a footacular widget!
```

Any subsequent call should display as follows:

```
Widget Name: Bar
Widget Description: This is a bartacular widget!
```

Look at how powerful this technique can be. You can fetch an object from the database, change a property of that object, and automagically write the changed information back to the database with just a few lines of code in `testWidget.php`. If nothing changes, you don't need to go back to the database, so you save load on the database server and improve the application's performance.

Users of the object do not necessarily need to understand its internals. If a senior member of the software development team wrote the `Widget` class, he or she could give this object to a junior member, who perhaps doesn't understand SQL as well, and the junior member of the team could put this object to use without any knowledge whatsoever of where the data comes from or how to save changes to it. In fact, you could change the data source from a PostgreSQL database to a MySQL database or even an XML file without the junior team member ever knowing or having to touch any of the code that uses this class.

Inheritance

If you were creating an application to handle inventory at a car dealership, you would probably need classes such as `Sedan`, `PickupTruck`, and `MiniVan` that would correspond to the same types of automobiles in the dealer's inventory. Your application would need not only to show how many of these items you have in stock, but also to report on the characteristics of these vehicles so that the salespeople could give the information to customers.

A sedan is a four-door car, and you would probably want to record the back-seat space and the trunk capacity. A pickup truck doesn't have a trunk, but does have a cargo bed with a certain capacity, and the truck itself has a towing capacity (the maximum weight of any cargo that can be safely carried). For a minivan, you would probably need to list the number of sliding doors (either one or two) and the number of seats inside.

However, each of these vehicles is really just a different type of automobile, and as such would share a number of characteristics in your application (such as color, manufacturer, model, year, vehicle identification number, and so on). To ensure that each of the classes has these same properties, you could just copy the code that creates those properties into each of the files containing your class definitions. As mentioned earlier in this chapter, one of the benefits of an OOP approach is code reuse. Therefore, of course, you don't need to copy code, but instead can reuse the properties and methods of these classes through a process called *inheritance*. Inheritance is the ability for one class to assume the methods and properties of a parent class.

Inheritance enables you to define a base class, in this case `Automobile`. You can say that other classes are also a type of `Automobile`, and as such have all the same properties and methods that all `Automobile`s have. Because a `Sedan` is an `Automobile`, it therefore automatically inherits everything defined by the `Automobile` class without your having to copy any code. That way, you need to write only the additional properties and methods of the `Sedan` class that are not shared by all automobiles. In other words, the only work left for you to do is define the differences; the similarities between the classes are inherited from the base class.

The ability to reuse code is one benefit, but there's a second major advantage to using inheritance. Suppose that you have a class called `Customer` with a method `buyAutomobile`. This method would take one parameter, an object of class `Automobile`, and its internal operations would print the paperwork needed to document the sale, and decrement the car in question from the inventory system. Because all `Sedan`s, `PickupTruck`s, and `MiniVan`s are `Automobile`s, you can pass objects of these classes to a function expecting an `Automobile`. Because the three specific types inherit from the more generic parent class, you know that they will all have the same base set of properties and methods. As long as you need only the methods and properties common to all `Automobile`s, you can accept objects of any class that inherits from `Automobile`.

Consider the example of cats. All cats share some properties. They eat, sleep, purr, and hunt. They also have shared properties — weight, fur color, whisker length, and running speed. However, lions have a mane of a certain length (at least, the male lions do), and they growl. Cheetahs have spots. Domesticated cats have neither of these things, yet all three animals are cats.

In PHP you specify that a class is a subset of another by using the keyword `extends`, which tells PHP that the class you are declaring should inherit all the properties and methods from its parent class, and that you are adding functionality or providing some additional specialization to that class.

If you had to design an application to handle zoo animals, you'd probably need to have classes `Cat`, `Lion`, and `Cheetah`. Before writing any code, plan your class hierarchy in Unified Markup Language (UML) diagrams so you have something to work from when you write the code and the documentation of those classes. (UML is examined in more detail in Chapter 2, so don't worry if you don't completely understand what's shown here.) Your class diagram should indicate a parent class `Cat`, with subclasses `Lion` and `Cheetah` inheriting from it. Figure 1-1 shows that diagram.

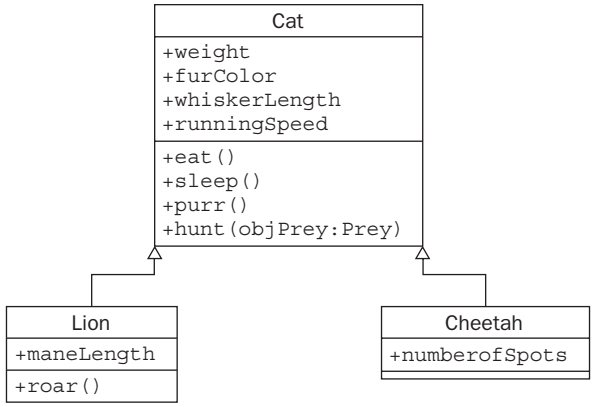

Figure 1-1: Class diagram for cats

Both the `Lion` and `Cheetah` classes inherit from `Cat`, but the `Lion` class also implements the property `maneLength` and the method `roar()`, whereas `Cheetah` adds the property `numberOfSpots`.

The `Cat` class, `class.Cat.php`, should be implemented as follows:

```
<?php

  class Cat {
    public $weight;         //in kg
    public $furColor;
    public $whiskerLength;
    public $maxSpeed;       //in km/hr
    public function eat() {
      //code for eating...
    }

    public function sleep() {
      //code for sleeping...
    }

    public function hunt(Prey $objPrey) {
      //code for hunting objects of type Prey
      //which we will not define...
    }

    public function purr() {
      print "purrrrrrr..." . "\n";
    }
  }
?>
```

This simple class sets up all the properties and methods common to all cats. To create the `Lion` and `Cheetah` classes, you could copy all the code from the `Cat` class to classes called `Lion` and `Cheetah`. However, this generates two problems.

First, if you find a bug in the `Cat` class, you must remember to fix it in the `Lion` and `Cheetah` classes as well. This creates more work for you, not less (and creating less work is supposed to be one of the primary advantages of an OO approach).

Second, imagine that you had a method of some other class (maybe `CatLover`) that looked like this:

```
public function petTheKitty(Cat $objCat) {
  $objCat->purr();
}
```

Although petting a lion or cheetah may not be a terribly safe idea, they will purr if they let you get close enough to do so. You should be able to pass an object of class `Lion` or `Cheetah` to the `petTheKitty()` function.

Therefore, you must take the other route to create the `Lion` and `Cheetah` classes, which is to use inheritance. By using the keyword `extends` and specifying the name of the class that is extended, you can easily create two new classes that have all the same properties as a regular cat but provide some additional features. Consider this example, which you can type into `class.Lion.php`:

```
<?php
  require_once('class.Cat.php');

  class Lion extends Cat {
    public $maneLength; //in cm

    public function roar() {
      print "Roarrrrrrrrr!";
    }
  }
?>
```

That's it! With the `Lion` class extending `Cat`, you can now do something like the following:

```
<?php
  include('class.Lion.php');

  $objLion = new Lion();
  $objLion->weight = 200;   //kg = \s450 lbs.
  $objLion->furColor = 'brown';
  $objLion->maneLength = 36; //cm = \s14 inches
  $objLion->eat();
  $objLion->roar();
  $objLion->sleep();
?>
```

As shown here, you can invoke the properties and methods of the parent class `Cat` without having to rewrite all that code. Remember that the `extends` keyword tells PHP to automatically include all the functionality of a `Cat`, along with any `Lion`-specific properties or methods. It also tells PHP that a `Lion`

object is also a `Cat` object, and you can now call the `petTheKitty()` function with an object of class `Lion` even though the function declaration uses `Cat` as the parameter hint:

```
<?php
  include('class.Lion.php');

  $objLion = new Lion();
  $objPetter = new CatLover();
  $objPetter->petTheKitty($objLion);
?>
```

In this way, any changes you make to the `Cat` class are automatically inherited by the `Lion` class. Bug fixes, changes to function internals, or new methods and properties are all passed along to the subclasses of a parent class. In a large, well-designed object hierarchy, this can make bug fixing and the addition of enhancements very easy. A small change to one parent class can have a large effect on the entire application.

In this next example, you'll see how a custom constructor can be used to extend and specialize a class. Create a new file called `class.Cheetah.php` and enter the following:

```
<?php
  require_once('class.Cat.php');

  class Cheetah extends Cat {
    public $numberOfSpots;

    public function __construct() {
      $this->maxSpeed = 100;
    }
  }
?>
```

Enter the following code into `testcats.php`:

```
<?php
require_once('class.Cheetah.php');

  function petTheKitty(Cat $objCat) {
    if($objCat->maxSpeed < 5) {
      $objCat->purr();
    } else {
       print "Can't pet the kitty - it's moving at " .
             $objCat->maxSpeed . " kilometers per hour!";
    }
  }
  $objCheetah = new Cheetah();
  petTheKitty($objCheetah);

  $objCat = new Cat();
  petTheKitty($objCat);
?>
```

The `Cheetah` class adds a new public member variable called `numberOfSpots` and a constructor that did not exist in the parent `Cat` class. Now, when you create a new `Cheetah`, the `maxSpeed` property (inherited from `Cat`) is initialized to 100 kilometers per hour (roughly 60 miles per hour), which is the approximate maximum speed of a cheetah over short distances. Note that because a default value for the `Cat` class isn't specified, the `maxSpeed` evaluates as `0` (actually, null) in the `petTheKitty()` function. As those who have ever had a house cat know, the amount of time they spend sleeping means that their maximum speed probably is approaching `0`!

By adding new functions, properties, or even constructors and destructors, the subclasses of a parent class can easily extend their functionality and, with a minimum amount of code, add new features and capabilities to your application.

> *When you can say that one class is a special type of some other class, use inheritance to maximize the potential for code reuse and increase the flexibility of your application.*

Overriding Methods

Just because a child class inherits from a parent class doesn't mean that the child class necessarily needs to use the parent class's implementation of a function. For example, if you were designing an application that needed to calculate the area of different geometric shapes, you might have classes called `Rectangle` and `Triangle`. Both of these shapes are polygons, and as such these classes will inherit from a parent class called `Polygon`.

The `Polygon` class will have a property called `numberOfSides` and a method called `getArea`. All polygons have a calculable area; however, the methods for calculating that area can be different for different types of polygons. (A generic equation exists for the area of any polygon, but it is often less efficient than the shape-specific equations for the simple polygons being used here.) The formula for the area of a rectangle is `w * h`, where `w` is the width of the rectangle and `h` is the height. The area of a triangle is calculated as `0.5 * h * b`, where `h` is the height of a triangle with base `b`. Figure 1-2 shows the area of two types of polygon.

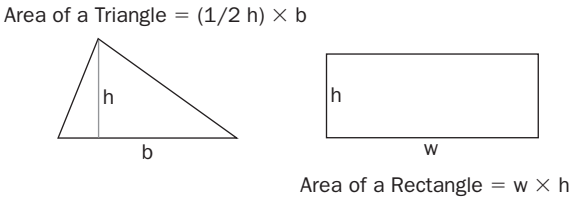

Figure 1-2: Area of two polygons

For each subclass of `Polygon` that you create, you will probably want to substitute an equation for the default implementation of the area method with one specific to the type of polygon class. By redefining that method for the class, you can provide your own implementation.

In the case of the `Rectangle` class, you would create two new properties, `height` and `width`, and override the `Polygon` class's implementation of the `getArea()` method. For the `Triangle` class, you would probably add properties to store information about the three angles, the height, and the length of the base segment, and override the `getArea()` method. By using inheritance and overriding methods of the parent class, you can allow the subclasses to specialize their implementations of those methods.

A function that takes a `Polygon` as a parameter and needs to print the area of that polygon will then automatically call the `getArea()` method of the subclass of `Polygon` that was passed to it (that is, `Rectangle` or `Triangle`). This capability for an OOP language to automatically determine at run-time which `getArea()` method to call is known as *polymorphism*. Polymorphism is the ability of an application to do different things based on the particular object it is acting on. In this case, that means invoking a different `getArea()` method.

> *Override a method in a subclass when the parent class's implementation is different from that required by the subclass. This allows you to specialize the activities of that subclass.*

Sometimes you want to retain the implementation provided by the parent class, but also perform some additional activities in the method of the subclass. For example, if you have an application that manages a nonprofit organization, you would probably have a class called `Volunteer` that would have a method called `signUp()`; this method would enable the volunteer to sign up for a community service project and add the user to the list of volunteers for that activity.

You might, however, have some users with restrictions (such as a criminal background) that should prevent them from signing up for certain projects. In this case, polymorphism enables you to create a class called `RestrictedUser` with an overridden `signUp()` method that first checks the restrictions on the user account against the properties of the project, preventing users from signing up if their restrictions do not allow them to volunteer for a particular activity. If their restrictions do not prohibit them from participating, then you should invoke the actions of the parent class to complete their registration.

When you override methods of the parent class, you do not necessarily need to completely rewrite the method. You can continue to use the implementation provided by the parent, but add additional specialization for your subclass. In this way, you can reuse code and also provide customizations as required by the business rules.

The capability of one class to inherit the methods and properties of another class is one of the most compelling features of an OO system, and one that enables you to gain an incredible level of efficiency and flexibility in your applications.

In the following example, you'll create two classes — `Rectangle` and `Square`. A square is a special kind of rectangle. Anything you can do with a rectangle you can do with a square; however, because a rectangle has two different side lengths and a square has only one, you need to do some things differently.

Create a file called `class.Rectangle.php` and add the following code:

```
<?php

  class Rectangle {
    public $height;
    public $width;

    public function __construct($width, $height) {
      $this->width = $width;
      $this->height = $height;
    }

    public function getArea() {
      return $this->height * $this->width;
    }

  }
?>
```

This is a fairly straightforward implementation of a class to model a rectangle. The constructor takes parameters for the width and height, and the area function calculates the area of the rectangle by multiplying them together.

Now, take a look at `class.Square.php`, shown here:

```
<?php
  require_once('class.Rectangle.php');

   class Square extends Rectangle {
    public function __construct($size) {
      $this->height = $size;
      $this->width = $size;
    }

    public function getArea() {
      return pow($this->height, 2);
    }

  }
?>
```

This code overrides both the constructor and the `getArea()` method. For a rectangle to be a square, all four sides must be of the same length. As a result, you need only one parameter for the constructor. If more than one parameter is passed to the function, any values after the first parameter are ignored.

> *PHP does not raise an error if the number of parameters passed to a user-defined function is greater than the number of parameters established in the function declaration. In a few cases, this is actually desired behavior. If you'd like to learn more, see the PHP documentation of the built-in* `func_get_args()` *function.*

The `getArea()` function also was overridden. The implementation in the `Rectangle` class would have returned a perfectly correct result for the `Square` objects. The method was overridden to improve application performance (although, in this case, the performance benefit is minuscule).

It is faster for PHP to fetch one property and compute its square than to fetch two properties and multiply them together.

By overriding constructors, destructors, and methods, you can alter aspects of how subclasses operate.

Preserving the Parent's Functionality

Sometimes you want to preserve the functionality provided by the parent. You don't need to completely override the function; you just need to add something to it. You could copy all the code from the parent method into the subclass's method, but as you've already seen, OOP offers you better ways of doing this than just copying lines of code.

To call the functionality provided by the parent, use the syntax `parent::[function name]`. When you just want to add additional behavior to a method, first you call `parent::[function name]` and then add your additional code. When extending a function in this way, always call the method on the parent before doing anything else. Doing so ensures that any changes to the operation of the parent won't break your code.

> *Because the parent class may be expecting the object to be in a certain state, or may alter the state of the object, overwrite property values, or manipulate the object's internal data, always invoke the parent method before adding your own code when extending an inherited method.*

The following example has two classes: `Customer` and `SweepstakesCustomer`. A supermarket has an application that, from time to time, switches which class is being used in the cash register application when certain promotions are run. Each customer who comes in has his or her own ID value (which comes from a database), as well as a customer number (which indicates how many customers have come to the supermarket before him or her). For these sweepstakes, the millionth customer wins a prize.

Create a file called `class.Customer.php` and add the following code:

```
<?php

    class Customer {
    public $id;
    public $customerNumber;
    public $name;

    public function __construct($customerID) {
      //fetch customer infomation from the database
      //
      //We're hard coding these values here, but in a real application
      //these values would come from a database
      $data = array();
      $data['customerNumber'] = 1000000;
      $data['name'] = 'Jane Johnson';

      //Assign the values from the database to this object
      $this->id = $customerID;
      $this->name = $data['name'];
      $this->customerNumber = $data['customerNumber'];
    }
  }
?>
```

Now create a file called `class.SweepstakesCustomer.php` and enter this code:

```
<?php
  require_once('class.Customer.php');

  class SweepstakesCustomer extends Customer {
    public function __construct($customerID) {
      parent::__construct($customerID);

      if($this->customerNumber == 1000000) {
        print "Congratulations $this->name! You're our " .
 "millionth customer!" .
               "You win a year's supply of frozen fish sticks! ";
       }

    }

  }
?>
```

How Inheritance Works

The `Customer` class initializes values from the database based on the customer ID. You would most likely retrieve the customer ID from a loyalty program swipe card such as the type available at most larger grocery store chains. With the customer ID, you can fetch the customer's personal data from the database (just hard-coded in this example), along with an integer value representing how many customers have entered the store before that customer. Store all this information in public member variables.

The `SweepstakesCustomer` class adds a bit of extra functionality to the constructor. You first invoke the parent class's constructor functionality by calling `parent::_construct` and passing to it the parameters it expects. You then look at the `customerNumber` property. If this number is the millionth, you inform this customer that he or she has won a prize.

To see how to use this class, create a file called `testCustomer.php` and enter the following code:

```
<?php

  require_once('class.SweepstakesCustomer.php');
  //since this file already includes class.Customer.php, there's
  //no need to pull that file in, as well.

  function greetCustomer(Customer $objCust) {
    print "Welcome back to the store $objCust->name!";
  }

  //Change this value to change the class used to create this customer object
  $promotionCurrentlyRunning = true;
  if ($promotionCurrentlyRunning) {
```

```
    $objCust = new SweepstakesCustomer(1000000);
  } else {
    $objCust = new Customer(1000000);
  }

  greetCustomer($objCust);

?>
```

Run `testCustomer.php` in your browser with the `$promotionCurrentlyRunning` variable set first to `false` and then to `true`. When the value is `true`, the prize message is displayed.

Interfaces

Sometimes you have a group of classes that are not necessarily related through an inheritance-type relationship. You may have totally different classes that just happen to share some behaviors in common. For example, both a jar and a door can be opened and closed, but they are in no other way related. No matter the kind of jar or the kind of door, they both can carry out these activities, but there is no other common thread between them.

What Interfaces Do

You see this same concept in OOP as well. An *interface* enables you to specify that an object is capable of performing a certain function, but it does not necessarily tell you how the object does so. An interface is a contract between unrelated objects to perform a common function. An object that implements this interface is guaranteeing to its users that it is capable of performing all the functions defined by the interface specification. Bicycles and footballs are totally different things, but objects representing those items in a sporting goods store inventory system must be capable of interacting with that system.

By declaring an interface and then implementing it in your objects, you can hand completely different classes to common functions. The following example demonstrates the rather prosaic door-and-jar analogy.

Create a file called `interface.Openable.php`:

```
<?php

  interface Openable {
    public function open();
    public function close();
  }

?>
```

Just as you name your class files `class.[class name].php`, you should use the same convention with interfaces and call them `interface.[interface name].php`.

You declare the interface `Openable` using a syntax similar to that of a class, except that you substitute the word `interface` for the word `class`. An interface does not have member variables, and it does not specify an implementation of any of its member functions.

Because no implementation is specified, you declare these functions to be `abstract`. Doing so tells PHP that any class implementing this interface is responsible for providing an implementation of the functions. If you fail to provide an implementation of *all* the abstract methods of an interface, PHP will raise a run-time error. You may not selectively choose some of the abstract methods to implement; you must provide implementations of them all.

How Interfaces Work

The `Openable` interface is a contract with other parts of the application that says any class implementing this interface will provide two methods, called `open()` and `close()`, that take no parameters. With this agreed-upon set of methods, you can allow very different objects to pass into the same function without the need for an inherited relationship to exist between them.

Create the following file, called `class.Door.php`:

```
<?php

require_once('interface.Openable.php');

class Door implements Openable {

    private $_locked = false;

    public function open() {
      if($this->_locked) {
        print "Can't open the door. It's locked.";
      } else {
        print "creak...<br>";
      }
    }

    public function close() {
      print "Slam!!<br>";
    }

    public function lockDoor() {
      $this->_locked = true;
    }
    public function unlockDoor() {
      $this->_locked = false;
    }

 }
?>
```

Now create a file called `class.Jar.php`:

```
<?
require_once('interface.Openable.php');

class Jar implements Openable {
  private $contents;
  public function __construct($contents) {
```

```
    $this->contents = $contents;
  }

  public function open() {
    print "the jar is now open<br>";
  }

  public function close() {
    print "the jar is now closed<br>";
  }
}
?>
```

To use these files, create a new file called `testOpenable.php` in the same directory:

```
<?php
  require_once('class.Door.php');
  require_once('class.Jar.php');

  function openSomething(Openable $obj) {
    $obj->open();
  }

  $objDoor = new Door();
  $objJar = new Jar("jelly");

  openSomething($objDoor);
  openSomething($objJar);
?>
```

Because both the `Door` class and the `Jar` class implement the `Openable` interface, you can pass both to the `openSomething()` function. Because that function accepts only something that implements the `Openable` interface, you know that you can call the functions `open()` and `close()` within it. However, you should not attempt to access the `contents` property of the `Jar` class or utilize the `lock()` or `unlock()` functions of the `Door` class within the `openSomething()` function, because that property and those methods are not part of the interface. The interface contract guarantees that you have `open()` and `close()` and nothing else.

By using interfaces in your application, you can allow completely different and unrelated objects to talk to each other with a guarantee that they will be able to interact on the terms specified in the interface. The interface is a contract to provide certain methods.

Encapsulation

As mentioned earlier in this chapter, objects enable you to hide the details of their implementation from users of the object. You do not need to know whether the `Volunteer` class mentioned earlier stores information in a database, a flat text file, an XML document, or another data-storage mechanism in order to be able to invoke the `signUp()` method. Similarly, you do not need to know whether the information about the volunteer contained within the object is implemented as single variables, an array, or even other objects.

This ability to hide the details of implementation is known as *encapsulation*. Generally speaking, encapsulation refers to two concepts: protecting a class's internal data from code outside that class, and hiding the details of implementation.

The word *encapsulate* literally means to place in a capsule, or outer container. A well-designed class provides a complete outer shell around its internals, and presents an interface to code outside the class that is wholly separated from the particulars of those internals. By doing so, you gain two advantages:

- You can change the implementation details at any time without affecting code that uses your class.
- Because you know that nothing outside your class can inadvertently modify the state or property values of an object built from your class without your knowledge, you can trust the state of the object and the value of its properties to be valid and to make sense.

The member variables of a class and its functions have a visibility. *Visibility* refers to what can be seen by code outside the class. As mentioned earlier in this chapter, *private* member variables and functions are not accessible to code outside the class and are used for the class's internal implementation. *Protected* member variables and functions are visible only to the subclasses of the class. *Public* member variables and functions are usable by any code, inside or outside the class.

Generally speaking, all internal member variables of a class should be declared private. Any access needed to those variables by code outside the class should be done through an accessor method. You don't let someone who wants you to try a new food blindfold and force-feed you; you need to be able to examine the item and determine whether you want to allow it into your body. Similarly, when an object wants to allow code outside it to change properties or in some other way affect its internal data, by encapsulating access to that data in a public function (and by keeping the internal data private), you have the opportunity to validate the changes and accept or reject them.

For example, if you are building an application for a bank that handles details of customer accounts, you might have an `Account` object with a property called `totalBalance` and methods called `makeDeposit()` and `makeWithdrawal()`. The total balance property should be read-only. The only way to affect the balance is to make a withdrawal or a deposit. If the `totalBalance` property were to be implemented as a public member variable, you could write code that would increase the value of that variable without having to actually make a deposit. This approach, obviously, would be bad for the bank.

Instead, you should implement this property as a private member variable and provide a public method called `getTotalBalance()`, which returns the value of that private member variable. Because the variable storing the value of the account balance is private, you can't manipulate it directly; and because the only public methods that affect the account balance are `makeWithdrawal()` and `makeDeposit()`, you have to actually make a deposit if you want to increase the value of your account.

By allowing you to hide the details of implementation and protect access to internal member variables, an object-oriented software development approach gives you a flexible, stable application.

> *Encapsulation of internal data and method implementations enables an object-oriented software system to protect and control access to data and hide the details of implementation.*

Changes to OO in PHP 6

Support for objects in PHP goes all the way back to PHP 3. There was never any intention of supporting the idea of classes or objects, but some limited support was added, almost as an afterthought, to provide "syntactic sugar" (to use Zeev Suraski's phrase) for associative arrays. Object support in PHP was originally designed as a convenient way of grouping data and functions, but only a small subset of the features traditionally associated with a full-blown object-oriented programming language was included. As PHP grew in popularity, the use of an OO approach became increasingly common in large applications. However, the poor internal implementation became limiting.

Most notably, there was no support for real encapsulation. You could not specify member variables or methods to be private or protected. Everything was public — which, as you've seen, can be problematic.

Additionally, there was no support for abstract interfaces or methods. Methods and member variables could not be declared static. There were no destructors. All these concepts are familiar to anyone with a background in another OOP language, and the lack of these features in PHP's object model could make the transition from a language such as Java (which does support all these ideas) to PHP difficult. For those who have previous experience with PHP 4, Table 1-1 lists some of the new features in the PHP

Table 1-1: Object Model Evolution from PHP 4 to PHP 5 and 6

New Feature	Benefit
Private, protected member variables and methods	Real encapsulation and data protection are now possible in PHP.
Improved dereferencing support	Statements such as `$obj->getSomething()->doSomething()` are now possible.
Static member variables and methods	Methods that can be called statically are now clearly identifiable. Class-level constants help control pollution of the global namespace.
Unified constructors	All class constructors are now called `__construct()`. This helps with encapsulation of overridden subclass constructors, and makes it easier to alter inheritance when multiple classes are involved in a tree of inheritance.
Destructor support	Through the `__destruct()` method, classes in PHP can now have destructors. This feature allows actions to be carried out when the object is destroyed.
Support for abstract classes and interfaces	You can define required methods in a parent class while deferring implementation to a subclass. Abstract classes can't be instantiated; only their non-abstract subclasses can.
Parameter type hints	You can specify the class for function parameters that are expecting an object. `function foo(Bar $objBar) {...` allows you to be sure that the data type of the parameter is what you expect.

object model that appeared starting in PHP 5. For those already familiar with OOP in PHP 5, little has changed in version 6.

Summary

In this chapter, you explored the concept of object-oriented programming (OOP). A class was shown as a blueprint for creating objects. Objects are run-time bundles of data and functions created from a class definition. They have characteristics, called *properties*, and behaviors, called *methods*. Properties can be thought of as variables, and methods as functions.

Some classes share a common parent type. For example, squares (children) are a type of rectangle (parent). When you declare a class to be a subtype of a parent class, it inherits the methods and properties of the parent. You have the option to override inherited methods. You can completely re-implement the method, if you so choose, or continue to use the parent's implementation but also add specializations particular to the subclass (or not override the method at all).

Encapsulation is an important concept to OOP. It refers to the capability of a class to protect access to its internal member variables and shield users of that class from the particulars of its implementation. Member methods and properties have three levels of visibility: private, protected, and public. Private members can be used only by the class's internal operations. Protected members are visible to subclasses. Public members can be used by code outside the class.

Object-oriented (OO) support in PHP received a major overhaul with the introduction of PHP 5 and the Zend Engine 2. New features and significant performance improvements since PHP version 5, and further enhanced in PHP 6, make PHP a real OOP language.

Chapter 2 discusses the Unified Modeling Language (UML), a system for diagramming OO software architecture that will provide you with the tools for describing and planning the architecture of even the most complex applications.