CHAPTER 2

Overview of Classes, Objects, and Interfaces

If you have worked with PHP in its earlier incarnations, then much of what you will find in PHP 5 is quite familiar, as you no doubt noticed from Lee Babin's survey of the major language features in Chapter 1. Like its predecessor, PHP 5 supports object-oriented programming, but the way in which this is implemented has changed; in fact, it has been expanded significantly from what was formerly available. In this chapter, we will acquaint you with the basics of classes, objects, and interfaces in PHP 5. In doing so, we will address the needs of several groups of readers to make sure that we all (forgive the expression) wind up on the same page.

This book assumes you have prior experience with PHP 4 or some other programming or scripting language. Of course, it is possible to write plenty of useful PHP 4 code (or PHP 5 code, for that matter) without using classes or objects, but we will start with a brief primer on those two concepts so you start to see the advantages of using them. If you are already familiar with the basics of object-oriented programming, then you will still find this chapter to be a useful time-saver; however, it might not be a revelation. If you are not familiar with object-oriented programming, prepare to be taken to the next level as a developer; once you begin to see the advantages of using it, you will likely not want to go back to a purely procedural way of coding.

If you are accustomed to writing object-oriented code in PHP 4, you will find that things have changed quite a bit, and mastering the differences might take some time. By the way, you will likely be happy to learn that the classes and objects you may have written for PHP 4 will usually work in PHP 5, although you may have to do a bit of tweaking here and there. We will point out the differences as we go and highlight any "gotchas" that might make things difficult for you.

If you are new to PHP but you have had some experience programming in a heavily object-oriented language such as Java or C++, you will find that most of the features of PHP 5 classes and objects will look familiar. In fact, the new object-oriented paradigm in PHP 5 is modeled largely on Java and includes interfaces, abstract classes and methods, and exceptions. We will discuss how these are handled in PHP 5 in due course.

Other topics we will cover in this chapter include the following:

- Creating, copying, and destroying objects
- Creating class diagrams
- Finding the methods and properties of an object
- Overriding methods and using polymorphism
- Creating classes and objects dynamically

By the time you have finished reading this chapter, you will have a good grasp of how classes and objects work in PHP 5, how they might prove useful in applications, how to obtain information about them, and the key differences between PHP 4 and PHP 5 in this regard. We will start with a brief look at what classes and objects are and explain some concepts you will need to understand in order to discuss and work with them.

Tip In this chapter, we have barely scratched the surface when it comes to object-oriented design, which is a huge field of its own and largely independent of any particular programming language. For more about object-oriented design principles as applied to PHP 5, refer to Matt Zandstra's excellent *PHP 5 Objects, Patterns, and Practice* (Apress, 2004). If you are serious about digging deep into PHP 5 objects and getting the most from them in your own applications, we strongly recommend you read this book. Since much object-oriented design literature is written with the assumption that the reader is fluent in C++ and/or Java, you will find *PHP 5 Objects, Patterns, and Practice* especially helpful if you do not have a great deal of experience in one of those two languages.

Understanding Basic Concepts

If you have never done any object-oriented programming before, then the terms *class* and *object* may sound mysterious and even a bit scary. (If you have, then you can safely gloss over or even skip the next few paragraphs and proceed to recipe 2-1.) However, classes and objects are really neither mysterious nor scary. After you have used them a few times, you will start to wonder how you managed to get along without them. Once you understand how they work, you should not have any trouble creating your own.

An *object* is basically a type of data that allows you to group data and functions under a single variable. In PHP, the -> operator denotes a member of a specific object, that is, a property (piece of data) or method (function) belonging to an object. You can think of it as meaning "has a," with the arrow pointing from the thing doing the having to the thing being had. In PHP 5, as in previous versions of the language, it is possible to create simple objects by doing nothing more than assigning object properties to an otherwise unused variable. For example:

```
<?php
  $ball->color = "green";
  $ball->weight = 100;
```

```
  printf("The ball's color is %s, and its weight is %d kilos.",
         $ball->color, $ball->weight);
?>
```

The output of the previous code is as follows:

```
The ball's color is green, and its weight is 100 kilos.
```

If you are familiar with associative arrays, then you will understand what we mean when we say that you can think of an object as being like an associative array with alternative notation. They are quite similar, both being just unordered lists with named elements (rather than indexed elements). Whether you use this

```
$ball['weight'] = 100;
```

or this

```
$ball->weight = 100;
```

each gives you a way of saying "the `$ball` has something called `weight`, whose value is 100."

Where objects really become useful is when you have a way to create them on demand from a single pattern. A *class* defines a reusable template from which you can create as many similar objects (*instances* of the class) as you need. For example, if you are writing an inventory system for a pet shop, you are likely to be working with information about lots of birds, and you can write a `Bird` class to represent a generic bird; each instance of this class then represents an individual bird. You can think of the class as a collection of variables and functions common to all birds. Variables attached to an object in this way are *properties*, and functions manipulating these variables are *methods*. Together, the methods and properties of an object are its *members*.

We have always thought of a programming object as being like a noun, with its properties being adjectives (describing aspects of the objects) and its methods being verbs (representing the object's actions or changes that can be made to its properties). In the case of the `Bird` class, you would likely want to have properties to account for a bird's name, breed, and price. In this context, you probably are not going to be too concerned about the bird's flight speed or direction of travel, even though most birds are capable of flight—if you are not likely to use some aspect or capability of the thing you are modeling when you define a class, then you should not bother creating a corresponding class member.

> **Note** A class is a template or prototype object. An instance of this class represents a particular case of this class. The word *object* can apply to classes and instances of classes alike.

Of course, programming objects do not have to model concrete objects; you can use them to represent abstractions as well. In subsequent chapters of this book, you will use classes to model abstract concepts such as dates and times (Chapter 5), files and directories (Lee Babin's Chapter 7), and Hypertext Markup Language (HTML) and Extensible Markup Language (XML) tags (Frank M. Kromann's Chapter 14).

2-1. Creating Instances Using Constructors

To create a new instance of a class (also referred to as *instantiating* a class), you can use the `new` operator in conjunction with the class name called as though it were a function. When used in this way, it acts as what is known as the *class constructor* and serves to initialize the instance. This instance is represented by a variable and is subject to the usual rules governing variables and identifier names in PHP. For example, consider the following:

```
$tweety = new Bird('Tweety', 'canary');
```

This creates a specific instance of `Bird` and assigns it to the variable named `$tweety`. In other words, you have defined `$tweety` as a `Bird`. So far you have not actually defined any members for the `Bird` class, and we have not discussed exactly how you define classes in PHP 5, so let's do those tasks now.

The Code

```
<?php
  class Bird
  {
    function __construct($name, $breed)
    {
      $this->name = $name;
      $this->breed = $breed;
    }
  }
?>
```

How It Works/Variations

This is about as simple a class definition as you can write in PHP 5. As is the case in PHP 4, a PHP 5 class is defined in a block of code that begins with the `class` keyword and the name of the class. In most cases, a class is not going to be useful unless you can create instances of it (this rule has exceptions, which you will learn about in recipe 2-11). To accomplish this task, you need a class constructor. In PHP 5, you do this by defining a method with the name `__construct()`; this method is called whenever you create a new instance of the class.

■ **Note** It is customary in most programming languages, including PHP, to begin class names with a capital letter and to write names of class instances beginning with a lowercase letter. This is a convention we will observe throughout this book.

■ **Note** In PHP 4, a class constructor was written as a method with the same name as the class (for example, using a `Bird()` method as the constructor for the `Bird` class). While PHP 5 still supports this way of creating instances of classes for backward compatibility, it is not recommended for new code. You will see why when we talk about extending classes in recipe 2-7.

The `$this` keyword has a special purpose: it allows you to refer to the instance from within the class definition. It works as a placeholder and means, "the current instance of this class." The `Bird` class constructor assigns the string `'Tweety'` to the `name` property of the instance you are creating and the string `'canary'` to its `breed` property. You can put this to the test like so:

```
<?php
  class Bird
  {
    function __construct($name, $breed)
    {
      $this->name = $name;
      $this->breed = $breed;
    }
  }

  $tweety = new Bird('Tweety', 'canary');

  printf("<p>%s is a %s.</p>\n", $tweety->name, $tweety->breed);
?>
```

The resulting output is as follows:

```
Tweety is a canary.
```

To determine the price you want to charge for Tweety, you could set the `price` property and output it like so:

```
<?php
  //  ...class defined and constructor called as previously shown...

  $tweety->price = 24.95;

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $tweety->name, $tweety->breed, $tweety->price);
?>
```

The output from this is as follows:

```
Tweety is a canary and costs $24.95.
```

Notice that no `$price` variable is defined within the class itself; you have created one arbitrarily. While this is not a terribly bad thing, it is also not a terribly good one: it means you can easily create `Bird` objects that are structurally inconsistent with others. If your application depends on all birds having prices, then you will run into trouble the first time you forget to assign a price to a bird. It is much better if you make sure every `Bird` has a `price` property by including it in the constructor. We will return to this topic shortly, but first we will finish discussing class instance creation and initialization.

Note It is possible to write a class without a `__construct()` method and even to instantiate it, but most of the time this is not very useful.

2-2. Using Default Constructors

Suppose also that most—say, 80 percent—of your birds are priced at $15. Wouldn't it be more convenient if all your `Bird` instances came with prices already set to that amount and you were required to set the prices of only the remaining 20 percent? PHP lets you set default values for function parameters and for class constructors. The following example shows a slightly revised `Bird` class.

The Code

```
<?php
  class Bird
  {
    function __construct($name='No-name', $breed='breed unknown', $price = 15)
    {
      $this->name = $name;
      $this->breed = $breed;
      $this->price = $price;
    }
  }

  $aBird = new Bird();
  $tweety = new Bird('Tweety', 'canary');

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $aBird->name, $aBird->breed, $aBird->price);

  $tweety->price = 24.95;

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $tweety->name, $tweety->breed, $tweety->price);
?>
```

Here is the output:

```
No-name is a breed unknown and costs $15.00.

Tweety is a canary and costs $24.95.
```

How It Works

You have created a default constructor for the `Bird` class. If you forget to set one or more properties when creating a new instance of the class, you will not get caught short later by a division-by-zero error, for example.

2-3. Setting Object Properties

A page or two back, we said it is better to include all properties of an object in its class definition rather than creating them dynamically. This is for two reasons. First, as we mentioned, you want to be sure all instances of a class have the same properties; otherwise, what happens when you forget to set a price for a `Bird` when some other part of your code expects there to be one? Second, when you assign a value to an object property, PHP does not check to see whether the property already exists. This means that it is all too easy to make a mistake that can be difficult to detect later, such as this one:

```
<?php
  class Bird
  {
      function __construct($name='No-name', $breed='unknown', $price = 15)
    {
      $this->name = $name;
      $this->breed = $breed;
      $this->price = $price;
    }
  }

  $polly = new Bird('Polynesia', 'parrot');

  $polly->rice = 54.95;  //  ooooops...!

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $polly->name, $polly->breed, $polly->price);
?>
```

The output from this script is as follows:

```
Polynesia is a parrot and costs $15.00.
```

Just in case you have not spotted the error, you can add the following line of debugging code to this script to see all of Polynesia's properties at a glance:

```
printf("<pre>%s</pre>\n", print_r(get_object_vars($polly), TRUE));
```

The function `get_object_vars()` makes a handy addition to your object-oriented programming toolkit. It takes any object as a parameter and returns an array whose keys are the names of the object's properties and whose values are the values of the properties. The output in this case is as follows:

```
Array
(
    [name] => Polynesia
    [breed] => parrot
    [price] => 15
    [rice] => 54.95
)
```

The typographical error has resulted in the addition of a new `rice` property to the `$polly` object, which is not what you wanted at all. You can avoid this sort of stuff by using methods to get and set properties rather than setting them directly. Let's rewrite the class, except this time we will include a `setPrice()` method.

The Code

```
<?php
  class Bird
  {
    function __construct($name='No-name', $breed='unknown', $price = 15)
    {
      $this->name = $name;
      $this->breed = $breed;
      $this->price = $price;
    }

    function setPrice($price)
    {
      $this->price = $price;
    }
  }

  $polly = new Bird('Polynesia', 'parrot');

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $polly->name, $polly->breed, $polly->price);

  $polly->setPrice(54.95);

  printf("<p>%s is a %s and costs \$%.2f.</p>\n",
          $polly->name, $polly->breed, $polly->price);
?>
```

The output from this example is as follows:

```
Polynesia is a parrot and costs $15.00.

Polynesia is a parrot and costs $54.95.
```

Variations

What happens if you change the line containing the call to the setPrice() method to something like the following?

```
$polly->setPice(54.95);
```

Because you are attempting to call a method that has not been defined, the result is an error message:

```
Fatal error: Call to undefined method Bird::setPice()
in /home/www/php5/bird-5.php on line 22
```

You will probably agree that this makes it much easier to find the source of the problem. The same situation exists with regard to getting values of object properties: if you ask PHP for the value of an undeclared variable, the chances are good that you will obtain zero, an empty string, NULL, or boolean FALSE. If this is the same as the property's default value (or if the property has no default value), then finding the source of the error can be particularly difficult. On the other hand, defining and using a getPrice() method minimizes the likelihood of such problems occurring. A construct such as this

```
printf("<p>%s is a %s and costs \$%.2f.</p>\n",
        $polly->getName(), $polly->getBreed(), $polly->getPrice());
```

may require a few extra keystrokes, but you will find that the time saved in tracking down problems that do not give rise to any error messages is worth the effort.

Note It is customary to name class members beginning with a lowercase letter. (A possible exception to this is static members, which we will talk about in recipe 2-5.) As for what to do when a name contains more than one word, two major schools of thought exist. Some programmers prefer to separate the words using underscores, for example, `my_long_method_name()`. Others use what is known as intercap notation, which consists of running the words together and capitalizing the first letter of each word after the first: `myLongMethodName()`. We prefer the latter, so that is what we use. If you do not have to work to someone else's coding conventions, then it is really just a matter of personal taste, as PHP does not care which one you use. However, you will find it easier in the long run to adopt one style or the other and stick with it.

2-4. Controlling Access to Class Members

We will start the discussion of this topic with a modified version of the previous example. The following shows the new `Bird` class, including a complete collection of get and set methods.

```
The Code<?php
  //  file bird-get-set.php
  class Bird
  {
    function __construct($name='No-name', $breed='unknown', $price = 15)
    {
      $this->name = $name;
      $this->breed = $breed;
      $this->price = $price;
    }

    function setName($name)
    {
      $this->name = $name;
    }

    function setBreed($breed)
    {
      $this->breed = $breed;
    }
```

Notice that we have written the `setPrice()` method in such a way that the price cannot be set to a negative value; if a negative value is passed to this method, the price will be set to zero.

```
function setPrice($price)
{
  $this->price = $price < 0 ? 0 : $price;
}

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

function getBreed()
{
  return $this->breed;
}

function getPrice()
{
  return $this->price;
}
```

To save some repetitive typing of the printf() statement that you have been using to output all the information you have about a given Bird object, you can add a new method named display() that takes care of this task:

```
  function display()
  {
    printf("<p>%s is a %s and costs \$%.2f.</p>\n",
            $this->name, $this->breed, $this->price);
  }
}
```

Variations

Now let's create a new instance of Bird. Let's say that before you have the chance to write this example, the shop sells Polynesia the parrot; so, you will use a magpie this time. First, call the constructor with some plausible values:

```
  $magpie = new Bird('Malaysia', 'magpie', 7.5);

  $magpie->display();
?>
```

You can verify that the class is working as expected by viewing the output in a browser:

```
Malaysia is a magpie and costs $7.50.
```

Because the neighborhood cats are begging you to get rid of the magpie—even if it means paying someone to take it off your hands—try using setPrice() to set the magpie's asking price to a negative number:

```
$magpie->setPrice(-14.95);

$magpie->display();
```

The setPrice() method prevents you from setting the price to a value less than zero:

```
Malaysia is a magpie and costs $0.00.
```

However, it is still possible to circumvent this restriction, whether you do so by accident or the culprit is some particularly crafty, magpie-hating feline hacker:

```
  $magpie->price = -14.95;

  $magpie->display();
?>
```

As you can see here, this is the output:

```
Malaysia is a magpie and costs $-14.95.
```

How can you stop this sort of thing from happening? The solution lies in a feature that will be familiar to anyone who has studied Java, but it is new in PHP 5: *visibility*. This allows you to control how class members can be accessed through three keywords:

- `public`: The property or method can be accessed by any other code. This is the default visibility for all class members in PHP 5. (Note: In PHP 4, all class members are `public`.)
- `private`: A `private` class member can be accessed only from within the same class. Attempting to do so from outside the class will raise an error.
- `protected`: A class member that is declared as `protected` may be accessed from within the class and from within any class that extends that class. (We will discuss how to extend classes in recipe 2-7.)

Now that you know about visibility, fixing the problem you encountered is simple. Just insert the following into the `Bird` class before the definition of the constructor:

```
private $name;
private $breed;
private $price;
```

When you reload the example in your browser, you will see something like this:

```
Malaysia is a magpie and costs $7.50.

Malaysia is a magpie and costs $0.00.

Fatal error: Cannot access private property Bird::$price in
 /home/www/php5/bird-7.php on line 60
```

Making the instance variables `private` forces you (or anyone else using the `Bird` class) to set its properties via the set methods you have defined, which ensures that any restrictions you have made on the values of those properties are followed.

■ **Tip** You can also declare methods as `public`, `private`, or `protected`, which has the same effect as for class variables. You will see some more examples of `private` methods from recipe 2-5 onward and examples of `protected` methods in recipe 2-11.

While it is true that the visibility of all class members defaults to public and that (unlike the case with Java or C++) you are not required to declare public variables, it is still a good idea to declare the visibility for all your variables. For one thing, it is good from an organizational viewpoint; for example, if you are in the habit of declaring all variables in advance, you will not surprise yourself later by accidentally reusing one of them. For another, the only way you can use private and protected variables is to declare them explicitly.

2-5. Using Static Members and the self Keyword

Sometimes you will want to access a variable or method in the context of a class rather than an object (class instance). You can do this using the static keyword, which is new in PHP 5. As an example, let's add a static property and a static method to the Bird class as it was in the previous example (in the file bird-get-set.php). The ordering does not matter a great deal, but our preference is to list all static members of a class first, so let's insert the new code immediately following the opening bracket in the class declaration.

The Code

```
public static $type = "bird";

public static function fly($direction = 'around')
{
  printf("<p>The bird is flying %s.</p>\n", $direction);
}
```

Note that static members have visibility just like any other class members, and if you do not declare them, they default to public. You can place the static keyword before or after the visibility keyword, but by convention, the visibility is declared first. Static methods are the same as any other method in that they take arguments, can return values, and can have default arguments. However, static methods and static properties are not linked to any particular instance of the class but rather to the class itself. You can reference them in your calling code using the name of the class and the :: operator. For example:

```
printf("<p>The Bird class represents a %s.</p>\n", Bird::$type);

Bird::fly();
Bird::fly('south');
```

The output from this snippet of code is as follows:

```
The Bird class represents a bird.

The bird is flying around.

The bird is flying south.
```

To access a static member from within an *instance* of the class, you have to do things a bit differently. Let's modify the `display()` method a bit to illustrate this:

```
public function display()
{
  printf("<p>The %s named '%s' is a %s and costs \$%.2f.</p>\n",
          self::$type, $this->name, $this->breed, $this->price);
}
```

Now you will create a new instance of `Bird` and see what this change accomplishes. Here is the code:

```
$sam = new Bird('Toucan Sam', 'toucan');
$sam->display();
```

Here is the output of the altered `display()` method:

```
The bird named 'Toucan Sam' is a toucan and costs $15.00.
```

If you look at the new version of the `display()` method, you will likely notice a new keyword, `self`. This keyword refers to the class. It is important not to confuse `self` with `this`: `this` means, "the current object" or "the current instance of a class." `self` means, "the current class" or "the class to which the current object belongs." The differences between them are as follows.

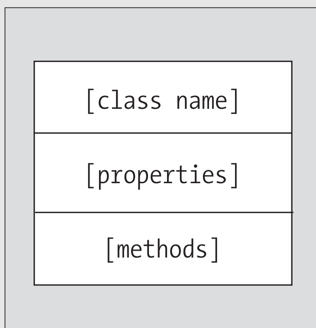
- The `self` keyword does the following:
  - Represents a class.
  - Is *never* preceded by a dollar sign (`$`).
  - Is followed by the `::` operator.
  - A variable name following the operator always takes a dollar sign (`$`). (Note that we said this about names of variables, not names of constants. Keep this in mind when you read the next section.) For example: `self::$type`.
- The `this` keyword does the following:
  - Represents an object or an instance of a class.
  - Is always preceded by a dollar sign (`$`).
  - Is followed by the `->` operator.
  - A variable name following the operator never takes a dollar sign (`$`). For example: `$this->name`.

Tip You will *never* see `$this` followed by `::` in working PHP 5 code.

CLASS DIAGRAMS

For short and simple classes, it is pretty easy to visualize the class and its members as a whole. However, as your classes grow longer and more complex—and particularly as you begin to use and write class libraries—you will probably want to use class diagrams both for designing new classes and for helping you understand classes written by others that you need to use. Fortunately, there's already a way to model classes in a language-neutral fashion. Universal Modeling Language (UML) is a standard for representing classes, their members, and the relationships between classes. UML actually does much more than model classes; it is a fairly lengthy and complex specification, and it would be impossible to cover all of it here. To find out more, visit the UML website at `http://www.uml.org/`, where you can obtain specifications, read tutorials, and get information about UML tools.

We will show you a limited subset of UML here, just enough to let you do some basic diagramming. A class is represented by a box divided into three regions or compartments, with the class name at the top, the class properties (also referred to as *attributes*) listed in the middle, and methods (known as *operations*) at the bottom, as shown in the following illustration. The only required section is the one containing the class name; the other two are optional.



You list properties like this:

```
<visibility> <property-name> : <data type> [= default-value]
```

You list the property's visibility first and then the name of the property. This is followed by a colon (`:`) and the property's data type. Optionally, you can include an equals sign followed by the property's default value, if it has one.

You list methods like this:

```
<visibility> <method-name>([<parameter-list>]) : <return-type>
```

As with properties, you list a method's visibility first and then the name of the method. Next comes a set of parentheses containing an optional list of parameters. The parentheses are followed by a colon and a return type. If the method returns no value, you use the keyword `void` to indicate the absence of one.

You write input parameters in this form:

```
[in] <parameter-name> : <data type> [= <default-value>]
```

Continued

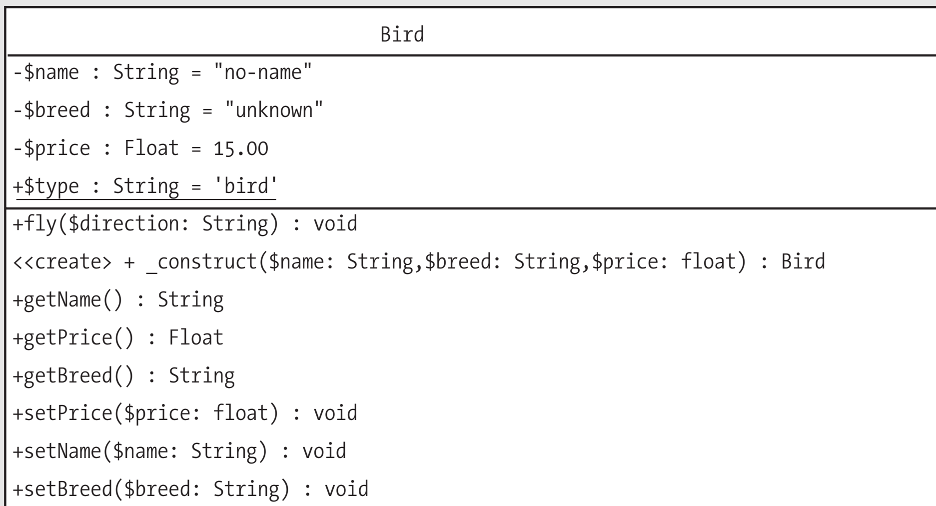
List each parameter name with a colon and then the parameter's data type. Some languages have both input and output parameters, and for this reason, you can precede parameter names with in, out, or inout. Because PHP has only input parameters, you will sometimes omit the in keyword, although some class diagramming tools may include it regardless. You can optionally follow with an equals sign and the parameter's default value, if it has one.

You indicate visibility with these symbols:

- **public**: + (plus sign)
- **private**: - (minus sign)
- **protected**: # (hash sign)

Static members are underlined or preceded by the modifier <<static>>. Other specifics are also represented by keywords enclosed in doubled angle brackets (also known as *stereotypes*). For instance, class constructors (which appear in recipes 2-8, 2-12, and others) and destructors (which are discussed extensively in recipe 2-10) are often indicated using, respectively, <<create>> and <<destroy>>.

For example, here's a UML representation of the Bird class:



You can use several tools to create UML class diagrams, including Microsoft Visio (Windows platforms only) and Borland Together Designer (Windows, Linux, Mac OS X, Solaris). Many of the more sophisticated tools include code-generation and reverse-engineering capabilities. For most of the diagrams in this book, we used something a bit simpler and less expensive: the open-source Umbrello UML Modeller, which is already included in some Linux distributions as part of the K Desktop Environment (KDE). You can also get the Umbrello source code for Linux from http://uml.sourceforge.net/ and compile it yourself. It is also possible to compile and run Umbrello on Windows platforms using Cygwin, a Unix emulator available from http://www.cygwin.com/. Version 1.4 is included with KDE 3.4. We had no problems compiling or using this release, or the more recent version 1.4.1, with KDE 3.3 and 3.4.

A cross-platform application called ArgoUML is available for free under a Berkeley Software Distribution (BSD) license from `http://argouml.tigris.org/`. Because ArgoUML is written in Java, it should run identically on all common platforms (which is important to us, as we use Linux, Windows, and occasionally FreeBSD and Solaris). It is also easy to install and run:

1. Download the archive for the latest release.
2. Unpack the archive into a convenient directory.
3. Open a shell or DOS prompt.
4. `cd` to the directory in which you unpacked the archive, and run the following command: `java -jar argouml.jar` (it should not be difficult to create a shortcut to handle this for you).

The only other requirement for ArgoUML is that you have the Java 2 Virtual Machine installed on your computer. If you run into problems, you can obtain documentation from the project website. While ArgoUML remains under development, the latest version (0.18.1) is sufficiently complete and stable for basic day-to-day use and makes a good learning tool.

In both the open-source modeling applications, the interface is fairly intuitive, and you can generate and save your class diagrams in PNG, JPG, SVG, PostScript, and other formats, as well as store data in the portable XML format. Each will also allow you to generate skeleton class code from your diagrams.

2-6. Using Class Constants

It is also useful sometimes to employ *class constants*. To declare a constant in a class, all you have to do is precede an identifier with the `const` keyword. A class constant is always public and static, and for this reason you cannot use the keywords `public`, `private`, `protected`, or `static` when declaring one. The following is an example of an `Employee` class that uses constants to enumerate classifications of employees. Let's walk through the class listing and some code to test this class. We will explain what is happening along the way.

The Code

```
<?php
  class Employee
  {
```

Let's say you need to allow for three categories of workers: regular workers, supervisors, and managers. You can define three constants, one per category:

```
const CATEGORY_WORKER = 0;
const CATEGORY_SUPERVISOR = 1;
const CATEGORY_MANAGER = 2;
```

Each employee classification has an associated job title and rate of pay. With this in mind, it seems reasonable to store those items of information in one or more arrays. Like other constants in PHP, a class constant must be a scalar type such as an integer or a string; you cannot use arrays or objects as constants. Since you might want to access information relating to employee categories independent of any given employee, create a couple of static arrays to hold job titles and rates of pay:

```
public static $jobTitles = array('regular worker', 'supervisor', 'manager');

public static $payRates = array(5, 8.25, 17.5);
```

Next, define a couple of static methods with which you can use the constants defined previously. They are both pretty simple: `getCategoryInfo()` takes a category number and returns the corresponding job title and rate of pay; `calcGrossPay()` takes two arguments (a number of hours and a category number) and returns the gross pay due an employee in that category working that many hours. Notice that when referring to static variables from within a method of that class—whether it is a static method or an instance method—you need to prefix the variable name with `self::`.

Note It is sometimes customary to use the `::` operator when discussing an instance method in relation to a class as a whole. For example, you might use `Employee::getFirstName()` as shorthand for "the `getFirstName()` method of the `Employee` class," even though `getFirstName()` is an instance method and not a static method. This should usually be clear from the context.

```
public static function getCategoryInfo($cat)
{
  printf("<p>A %s makes \$%.2f per hour.</p>\n",
          self::$jobTitles[$cat],
          self::$payRates[$cat]);
}

public static function calcGrossPay($hours, $cat)
{
  return $hours * self::$payRates[$cat];
}
```

Now let's define some instance variables. Each employee has a first name, a last name, an ID number, and a job category code. These are all private variables; but we will define public methods for manipulating them.

```
private $firstName;
private $lastName;
private $id;
private $category;
```

The `Employee` constructor is pretty simple. It just assigns its parameters to the corresponding instance variables. For convenience, give `$cat` (the job category identifier) a default value, as shown here:

```
public function __construct($fname, $lname, $id, $cat=self::CATEGORY_WORKER)
{
  $this->firstName = $fname;
  $this->lastName = $lname;
  $this->id = $id;
  $this->category = $cat;
}
```

Next, define some (unremarkable) get and set methods:

```
public function getFirstName()
{
  return $this->firstName;
}

public function getLastName()
{
  return $this->lastName;
}

public function getId()
{
  return $this->id;
}

public function getCategory()
{
  return $this->category;
}

public function setFirstName($fname)
{
  $this->firstName = $fname;
}

public function setLastName($lname)
{
  $this->lastName = $lname;
}

public function setId($id)
{
  $this->id = $id;
}
```

Instead of a `setCategory()` method, you define two methods—`promote()` and `demote()`—to update the employee's job category. The first of these increments the category property, but only if it is less than the maximum (`Employee::CATEGORY_MANAGER`); the second decrements it, but only if it is greater than the minimum (`Employee::CATEGORY_WORKER`).

Notice that these values are prefixed with `self`. If you do not do this, you will make PHP think you are trying to use global constants with these names rather than class constants, which is not what you want to do here.

```
public function promote()
{
  if($this->category < self::CATEGORY_MANAGER)
    $this->category++;
}

public function demote()
{
  if($this->category > self::CATEGORY_WORKER)
    $this->category--;
}
```

Finally, define a `display()` method that outputs the current values of all the properties:

```
 public function display()
 {
   printf(
     "<p>%s %s is Employee #%d, and is a %s making \$%.2f per hour.</p>\n",
     $this->getFirstName(),
     $this->getLastName(),
     $this->getId(),
     self::$jobTitles[ $this->getCategory() ],
     self::$payRates[ $this->getCategory() ]
         );
 }
}  //  end class Employee
```

Figure 2-1 shows a UML diagram of the `Employee` class.

Let's put the `Employee` class through a few paces. First, test the static `getCategoryInfo()` method:

```
Employee::getCategoryInfo(Employee::CATEGORY_SUPERVISOR);
```

Next, create an instance of `Employee`; Bob Smith is employee number 102 and is a supervisor. You can `display()` Bob and verify that his attributes are what you would expect them to be:

```
$bob = new Employee('Bob', 'Smith', 102, Employee::CATEGORY_SUPERVISOR);
$bob->display();
```

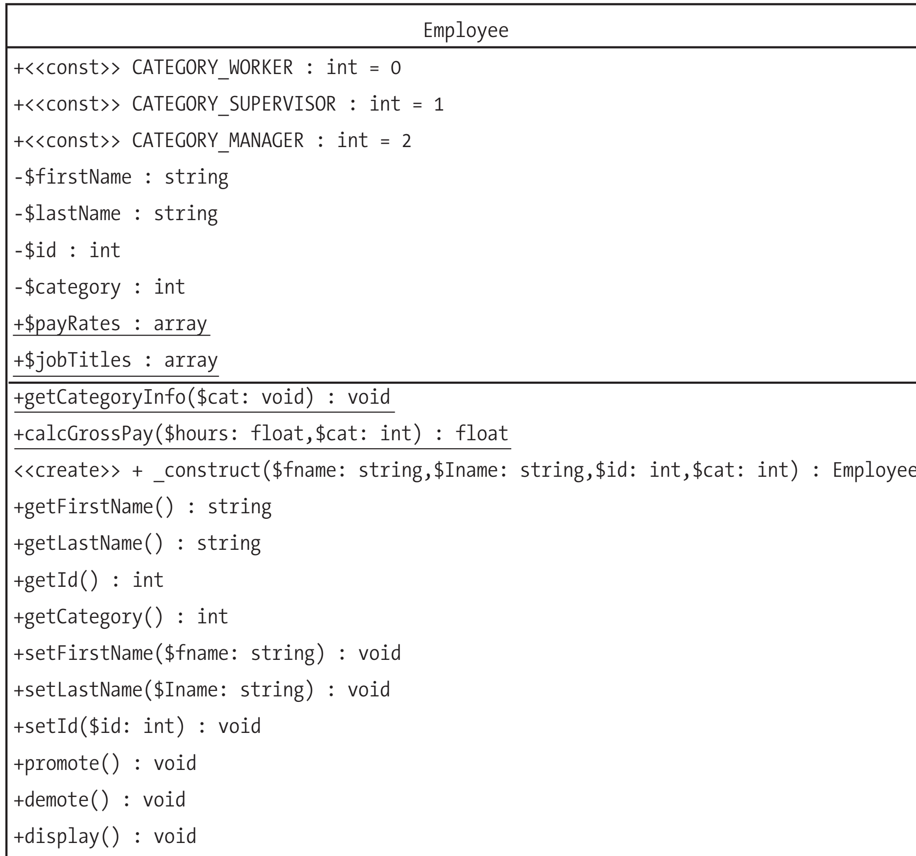


Figure 2-1. *UML representation of the* `Employee` *class*

You can promote Bob and then call the `display()` method once again to show that the change was made:

```
$bob->promote();
$bob->display();
```

If you try to promote Bob a second time, nothing about him should change; the previous call to the `promote()` method has already made him a manager, and there's no higher employee category.

```
$bob->promote();
$bob->display();
```

Now you will demote Bob. He should be returned to his original supervisor role:

```
$bob->demote();
$bob->display();
```

Finally, test the static `calcGrossPay()` method:

```
  $hours_worked = 35.5;
  printf("<p>If %s %s works %.2f hours, he will gross \$%.2f.</p>\n",
          $bob->getFirstName(),
          $bob->getLastName(),
          $hours_worked,
          Employee::calcGrossPay($hours_worked, $bob->getCategory())
        );
?>
```

Tip The `::`, or scope resolution operator, is sometimes referred to as the *paamayim nekudotayim*, which is Hebrew for "double colon." If you see this term as part of a PHP error message (for example, `Parse error: Unexpected T_PAAMAYIM_NEKUDOTAYIM...`), it is often an indicator that you are using the `::` operator where PHP is expecting `->` or the reverse.

You can see this output:

```
A supervisor makes $8.25 per hour.

Bob Smith is Employee #102 and is a supervisor making $8.25 per hour.

Bob Smith is Employee #102 and is a manager making $17.50 per hour.

Bob Smith is Employee #102 and is a manager making $17.50 per hour.

Bob Smith is Employee #102 and is a supervisor making $8.25 per hour.

If Bob Smith works 35.50 hours, he will gross $292.88.
```

In the call to `getCategoryInfo()` (and to `calcGrossPay()`, by inference), you can see the advantage to using named class constants; you do not have to remember that a supervisor has a job category ID of 1. Instead, you just write `Employee::CATEGORY_SUPERVISOR`. In addition, if you add a new job category—say, assistant manager—you do not have to hunt through your code and change a bunch of numbers. You can merely update the appropriate section of the class to read something like this:

```
const CATEGORY_WORKER = 0;
const CATEGORY_SUPERVISOR = 1;
const CATGORY_ASST_MANAGER = 2;
const CATEGORY_MANAGER = 3;

public static $jobTitles
  = array('regular worker', 'supervisor', 'assistant manager', 'manager');

public static $payRates = array(5, 8.25, 12.45, 17.5);
```

Try making this modification to `Employee`, and you will find that the example code still works (although the output will be slightly different). Obviously, you can make further improvements in this class; for instance, the set methods (including `promote()` and `demote()`) could return boolean values to indicate success or failure. (However, you will look at a feature new in PHP 5 that actually gives you a better strategy when it comes to handling errors in recipe 2-11). We have quite a bit left to cover in this introduction to classes and objects, so we will now show how you can build sets of classes that relate to one another.

2-7. Extending Classes

If you were not already familiar with classes and objects, then by now perhaps you are starting to see just how useful and economical they can be in PHP 5. However, we have not touched on one of their most powerful features, which lies in the ability to reuse an existing class when creating one or more new ones. This technique is known as *extending* a class.

Extending classes is useful when you have multiple objects that have some but not all properties or methods in common. Rather than write a separate class for each object that duplicates the members that are common to all, you can write a generic class that contains these common elements, extend it with subclasses that inherit the common members, and then add those that are specific to each subclass.

■ **Note** Unlike some object-orienting programming languages, PHP 5 does not support multiple inheritance. In other words, a derived class can have only one parent. However, a class *can* have multiple child classes. In addition, a PHP 5 class can implement multiple interfaces (see recipe 2-9 later in this chapter).

Figure 2-2 shows an example in which we have reworked the `Bird` class from earlier in this chapter and split it up into three classes. The new `Parrot` and `Canary` classes are subclasses of `Bird`. The fact that they each inherit the methods and properties of the `Bird` class is indicated by the arrows, whose heads point to the parent class.

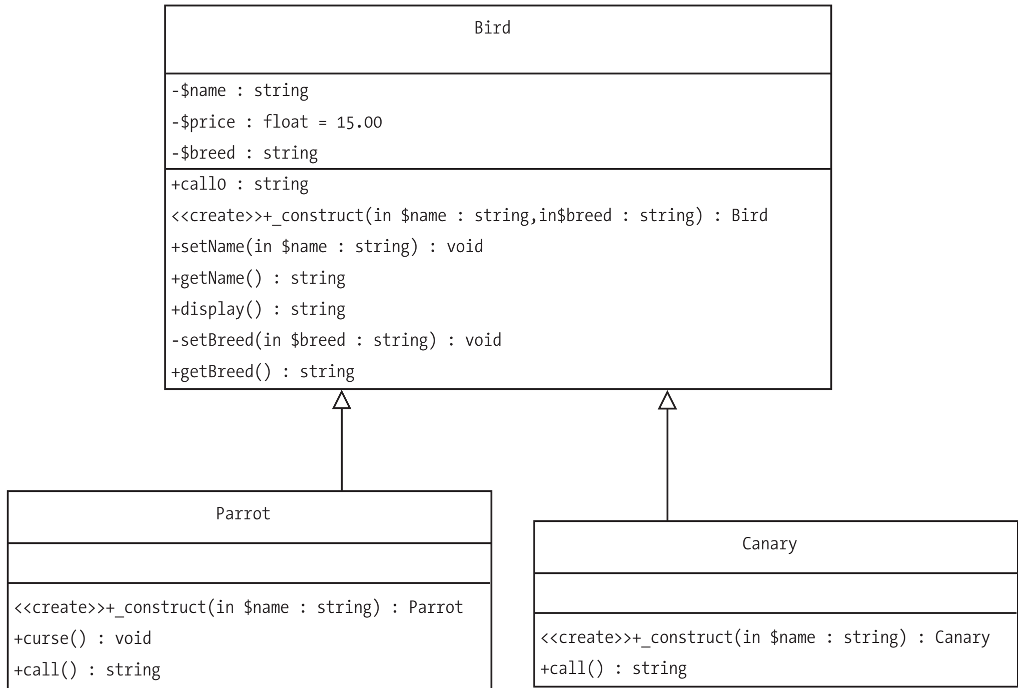


Figure 2-2. *UML diagram showing class inheritance*

The following is some PHP 5 code that implements these three classes. `Bird` has three properties (`$name`, `$price`, and `$breed`), all of which are private. You can set the first two of these with the public methods `setName()` and `setPrice()`, respectively, or in the class constructor. You can set the breed *only* from the `Bird` class constructor; because the `setBreed()` method is private, it can be called only from within `Bird`, not from any other code. Since `$breed` has no default value, you will receive a warning if you do not set it in the constructor. This seems reasonable—you could rename a bird or change its price easily enough in real life, but you will not often be transforming a pigeon into a bird of paradise unless you are a magician. Notice that you have changed this from the earlier incarnations of this class where you had a default value for this property; here you are saying, "I do not want anyone adding a bird to my inventory unless they say exactly what sort of bird it is." You also force the programmer to name the bird when it is created; however, the price does have a default value.

The Code

```
<?php
  //  file: bird-multi.php
  //  example classes for inheritance example
  class Bird
  {
    private $name;
    private $breed;
    private $price;
```

```
    public function __construct($name, $breed, $price=15)
    {
      $this->setName($name);
      $this->setBreed($breed);
      $this->setPrice($price);
    }

    public function setName($name)
    {
      $this->name = $name;
    }

    private function setBreed($breed)
    {
      $this->breed = $breed;
    }

    public function setPrice($price)
    {
      $this->price = $price;
    }
```

All the get methods of this class are public, which means you can call them at any time from within the `Bird` class, from within any subclasses of `Bird` that you might create, and from any instance of `Bird` or a `Bird` subclass. The same is true for the `display()` and `birdCall()` methods.

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

public function getBreed()
{
  return $this->breed;
}

public function getPrice()
{
  return $this->price;
}
```

Each bird makes some sort of sound. Unless you override the `birdCall()` method in a subclass, you assume that the bird chirps. We will discuss overriding class methods in the "Variations" section. (We have named this method `birdCall()` rather than just `call()` to avoid writing any confusing bits such as "make a call to `call()`" in the course of this discussion. Do not let this lead you to think that there's some requirement we are not telling you about to make class names part of the names of their members or anything of that sort.)

```
  public function birdCall()
  {
    printf("<p>%s says: *chirp*</p>\n", $this->getName());
  }

  public function display()
  {
    printf("<p>%s is a %s and costs \$%.2f.</p>",
            $this->getName(),
            $this->getBreed(),
            $this->getPrice());
  }
}  //  end class Bird
```

Variations

Now let's extend `Bird` to create a `Parrot` class. You indicate that `Parrot` extends `Bird` by using the `extends` keyword as follows. What this means is that `Parrot` inherits all the properties and methods of `Bird`. For example, each instance of `Parrot` has a `birdCall()` method. Because `birdCall()` is a public method, you can redefine it in `Parrot` without it affecting the `birdCall()` method when called by an instance of `Bird` or another subclass. This is what we mean by *overriding* a method of a parent class.

```
class Parrot extends Bird
{
  public function birdCall()
  {
    printf("<p>%s says: *squawk*</p>\n", $this->getName());
  }
```

You can also override the `Bird` class constructor. In this case, what you do is call the parent's constructor using the `parent` keyword. This keyword means "the class from which the current class is derived," and when employing it, the double-colon operator is always used to indicate its members.

Caution When extending a class in PHP 5, you should always call the parent constructor in the constructor of the derived class; this is *not* done automatically. If you do not call `parent::__construct()` at some point in the constructor of the subclass, the derived class will not inherit the properties and methods of the parent. Also note that when you do so, you must make sure the parent constructor receives any parameters it is expecting. For this reason, it is often advantageous to write the parent class constructor in a way such that all parameters have default values; however, sometimes you do not want this to happen, and you must judge this for yourself on a case-by-case basis.

The $name is passed to the Parrot constructor; you supply the values parrot and 25 for the $breed and $price parameters. Thus, every Parrot has parrot as its breed and $25 as its price, and while the price can later be updated, the breed cannot be changed once the Parrot has been instantiated.

```
public function __construct($name)
{
  parent::__construct($name, 'parrot', 25);
}
```

Notice that while you cannot call the setBreed() method of Bird directly from within Parrot, you can call the Bird constructor, which does call setBreed(). The difference is that setBreed() gets called *from within Bird*.

Note Is it possible to override a method of a parent class where that method was declared as private? Yes and no. If you try to call the parent class method directly—for example, if you write parent::setBreed() at some point in the Parrot class—you will get a fatal error. If you do some experimenting, you will find that nothing is preventing you from defining a new setBreed() method in Parrot, but you must keep in mind that this method has nothing to do with the method of the same name found in Bird. In any case, you cannot set the $breed property in the Parrot class, because it was defined as private in Bird. The moral of the story is this: if you need to override a parent method in a subclass in any meaningful way, declare the method as either public or protected in the parent class.

Now define a new method that is specific to Parrot, reflecting that parrots are often graced with a vocabulary that is not available to other birds.

```
  public function curse()
  {
    printf("<p>%s curses like a sailor.</p>\n", $this->getName());
  }
}  //  end class Parrot
```

The curse() method is defined only for Parrot, and attempting to use it with Bird or Canary will give rise to a fatal error.

The Canary class also extends Bird. You override the birdCall() method, but with a bit of a twist: you provide the option to use either the parent's birdCall() method or a different one. To invoke the canary-specific functionality, all that is required is to invoke birdCall() with the value TRUE.

```
class Canary extends Bird
{
  public function birdCall($singing=FALSE)
  {
    if($singing)
      printf("<p>%s says: *twitter*</p>\n", $this->getName());
    else
      parent::birdCall();
  }
```

The Canary constructor overrides the parent's constructor in the same way that the Parrot constructor does, except of course it passes canary as the value for $breed and uses the default value for $price.

```
    public function __construct($name)
    {
      parent::__construct($name, 'canary');
    }
  }
?>
```

Let's test these classes:

```
<?php
  //  file: bird-multi-test.php
  //  test Bird class and its Parrot and Canary subclasses
  //  depends on classes defined in the file bird-multi.php
```

Of course, you cannot use the classes defined previously unless they are available to the current script either by including the class code itself or by including the file in which the classes are defined. You use the require_once() function so that the script will fail if the file containing the classes is not found.

```
require_once('./bird-multi.php');
```

The tests themselves are pretty simple. First, create a new Parrot and call those methods that produce output, including the curse() method defined specifically for Parrot. (Because display() is a public method of Bird, you can use it as an instance method of any class deriving from Bird without redefining it.)

```
$polly = new Parrot('Polynesia');
$polly->birdCall();
$polly->curse();
$polly->display();
```

Next, instantiate the Canary class, and call its output methods. In the case of the Bird::birdCall() method, the Parrot object $polly always shows the overridden behavior; $tweety uses the parent's birdCall() method unless you pass boolean TRUE to it, in which case this Canary object's birdCall() method acts in the alternative manner that you defined for it. You can invoke Canary::birdCall() in both ways (with and without TRUE as a parameter) to demonstrate that this is so:

```
$tweety = new Canary('Tweety');
$tweety->birdCall();
$tweety->birdCall(TRUE);
$tweety->display();
```

Now use the setName() method to give the canary a different name, once again invoking its display() method to verify that the name has changed:

```
$tweety->setName('Carla');
$tweety->display();
```

Finally, you can still use the `Bird` constructor directly in order to create a bird of some type other than a parrot or canary. Invoke its `birdCall()` and `display()` methods to illustrate that the object was created and has the attributes and behavior you would expect:

```
  $keet = new Bird('Lenny', 'lorakeet', 9.5);
  $keet->birdCall();
  $keet->display();
?>
```

Here is the output from the test script:

```
Polynesia is a parrot and costs $25.00.

Polynesia says: *squawk*

Polynesia curses like a sailor.

Tweety is a canary and costs $15.00.

Tweety says: *chirp*

Tweety says: *twitter*

Carla is a canary and costs $15.00.

Lenny is a lorakeet and costs $9.50.

Lenny says: *chirp*
```

Tip PHP 5 introduces a feature that makes it easier to include classes in files by allowing you to define an `__autoload()` function, which automatically tries to include a class file for any class that is not found when you attempt to use it. To take advantage of this, you need to save each class in its own file and follow a strict naming convention for these files, such as saving a class named `ClassName` in a file named `ClassName.inc.php`. For example, define the following:

```
function __autoload($classname)
{
  require_once("/includes/classes/$classname.inc.php");
}
```

In this case, if you try to use the class `MyClass` and if it was not already defined in your script, then PHP automatically attempts to load the class named `MyClass` from the file `/includes/classes/MyClass.inc.php`. However, you must be careful to follow the naming convention implied by your `__autoload()` function, because PHP will raise a fatal (unrecoverable!) error if it cannot find the class file. The `__autoload()` function also works with regard to interfaces not already defined in your scripts (see recipe 2-9).

2-8. Using Abstract Classes and Methods

The `Bird::birdCall()` method you used in the previous example has a fallback in case a derived class does not override it. Now let's suppose you are not interested in providing a default behavior for this method; instead, you want to *force* all `Bird` subclasses to provide `birdCall()` methods of their own. You can accomplish this using another feature that is new to PHP in version 5—*abstract* classes and methods.

Note When it is necessary to emphasize that a class or method is *not* abstract (for instance, when a class completely implements an abstract class), it is often referred to as being *concrete*.

An abstract method is one that is declared by name only, with the details of the implementation left up to a derived class. You should remember three important facts when working with class abstraction:

- Any class that contains one or more abstract methods must itself be declared as abstract.
- An abstract class cannot be instantiated; you must extend it in another class and then create instances of the derived class. Put another way, only concrete classes can be instantiated.
- A class that extends the abstract class must implement the abstract methods of the parent class or itself be declared as abstract.

Let's update the `Bird` class so that its `birdCall()` method is abstract. We will not repeat the entire class listing here—only two steps are necessary to modify `Bird`. The first step is to replace the method declaration for `Bird::birdCall()` with the following:

```
abstract public function birdCall();
```

An abstract method has no method body; it consists solely of the `abstract` keyword followed by the visibility and name of the function, the `function` keyword, a pair of parentheses, and a semicolon. What this line of code says in plain English is, "Any class derived from this one must include a `birdCall()` method, and this method must be declared as `public`."

The second step is to modify the class declaration by prefacing the name of the class with the `abstract` keyword, as shown here:

```
abstract class Bird
```

Figure 2-3 shows a UML diagram of the modified three-class package. Abstract classes and methods are usually indicated with their names in italics; alternatively, you can use the stereotype `<<abstract>>` for this purpose.

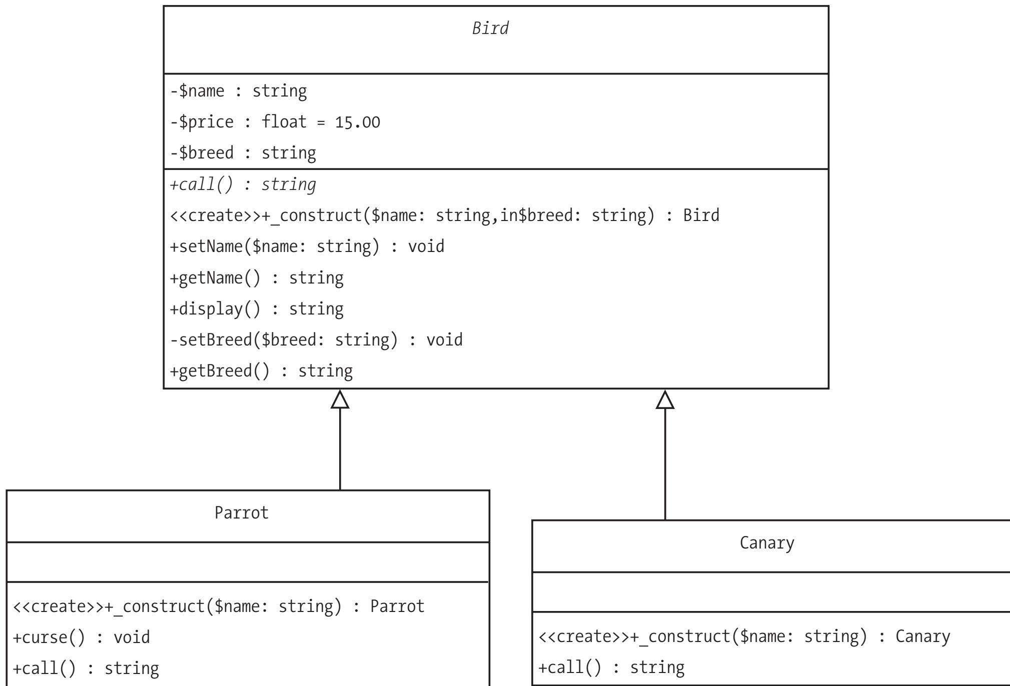


Figure 2-3. *Modified (abstract)* `Bird` *and derived (concrete) classes*

Now you need to consider how `birdCall()` is implemented in `Parrot` and `Canary`. `Parrot::birdCall()` is fine the way it is; it is not abstract, and it does not refer to the `birdCall()` method of the parent class. With `Canary`'s `birdCall()` method, however, you have a problem: you cannot invoke the parent's version of the method because it is abstract. However, it is not much work to reimplement `birdCall()` so that this does not happen.

The Code

```
public function birdCall($singing=FALSE)
{
  $sound = $singing ? "twitter" : "chirp";

  printf("<p>%s says: *%s*</p>\n", $this->getName(), $sound);
}
```

Let's see what happens when you rerun the test code in `bird-multi-test.php`:

```
Polynesia is a parrot and costs $25.00.

Polynesia says: *squawk*

Polynesia curses like a sailor.

Tweety is a canary and costs $15.00.

Tweety says: *chirp*

Carla is a canary and costs $15.00.

Carla says: *chirp*

Carla says: *twitter*

Fatal error: Cannot instantiate abstract class Bird in
 /home/www/php5/bird-multi-test-2.php on line 18
```

Extension

You run into trouble at the point where you try to create an object representation of Lenny the lorakeet. You cannot create an instance of `Bird` because it is now an abstract class. You can solve this problem in two ways (unless you want to pretend that Lenny is actually a parrot), and they both involve creating another concrete class that extends `Bird`. You can write either a `Lorakeet` class just for use with lorakeets or a generic bird class (which you can call `GenericBird` or whatever you like) that provides a catchall for species of birds for which you do not want to write separate classes. We will leave the choice up to you; as an exercise, spend a bit of time thinking about this sort of problem and the ramifications of both solutions.

Tip If you do not want a method to be overridden in a subclass, you can keep this from happening by declaring it with the `final` keyword, which functions more or less as the opposite to `abstract`. For example, you could have declared `Bird::display()` as `final` without affecting either the `Parrot` class or the `Canary` class as written, since neither subclass tries to override `display()`. You can also declare an entire class as `final`, which means it cannot be subclassed at all. Since this is not difficult to prove, we will leave that task as an exercise for you to do. Note that the `final` keyword comes before `public`, `protected`, or `static`. We should also point out that it makes no sense to use `final` with `private`, since you cannot override a private member of a class in any case, and a class declared as both `final` (no subclassing) and `private` (no direct access) could not be used at all.

2-9. Using Interfaces

As you saw in the previous section, abstract classes and methods allow you to declare some of the methods of a class but defer their implementation to subclasses. So...what happens if you write a class that has *all* abstract methods? We will offer you a somewhat indirect answer to this question: what you end up with is just one step removed from an *interface*. You can think of an interface as a template that tells you what methods a class should expose but leaves the details up to you. Interfaces are useful in that they can help you plan your classes without immediately getting bogged down in the details. You can also use them to distill the essential functionality from existing classes when it comes time to update and extend an application.

To declare an interface, simply use the `interface` keyword, followed by the name of the interface. Within the body of the interface, list declarations (delimited, as with classes, by braces, {...}) for any methods to be defined by classes that implement the interface.

Note In PHP 5 you can provide type hints for parameters of functions and methods, but only for types you define. In other words, if you have defined a class named `MyClass` and then define a method `MyMethod` (of `MyClass` or any other class) that takes an instance of `MyClass` as a parameter, you can declare it as (for instance) `public function myMethod(MyClass $myParam)`. This will cause PHP to issue a warning if you try to use a value of some other type with `myMethod`. However, you cannot use type hints for predefined data types such as `int` or `string`; attempting to do so will raise a syntax error.

The Code

Looking at the `Bird` class, you might deduce that you are really representing two different sorts of functional units: a type of animal (which has a name and a breed) and a type of product (which has a price). Let's generalize these into two interfaces, like so:

```
interface Pet
{
  public function getName();
  public function getBreed();
}

interface Product
{
  public function getPrice();
}
```

How It Works

To show that a class implements an interface, you add the `implements` keyword plus the name of the interface to the class declaration. One advantage that interfaces have over abstract classes is that a class can implement more than one interface, so if you wanted to show that `Bird` implements both `Pet` and `Product`, you would simply rewrite the class declaration for `Bird`, as shown here:

```
abstract class Bird implements Pet, Product
```

In fact, if you do this to the existing example that uses the `Bird`, `Parrot`, and `Canary` classes, you will find that the example still runs. This is probably a good place to point out that when you use (for example) `implements anInterface` in a class declaration, you are basically saying, "I promise to implement in this class any methods listed in `anInterface`." You cannot defer the implementation of an interface method by declaring it as `abstract` in the implementing class.

Caution In PHP 5, interfaces may declare only methods. An interface cannot declare any variables.

An interface is represented in UML diagrams by a box with two compartments, the top one containing the stereotype `<<Interface>>` followed by the name of the interface and the bottom one containing the signatures of the interface's methods. Figure 2-4 shows the updated `Bird` class diagram with the `Pet` and `Product` interfaces and their relationship with the `Bird` class. Note that the implementation by a class of an interface is indicated by a dashed arrow that points from the class to the interface that it implements.

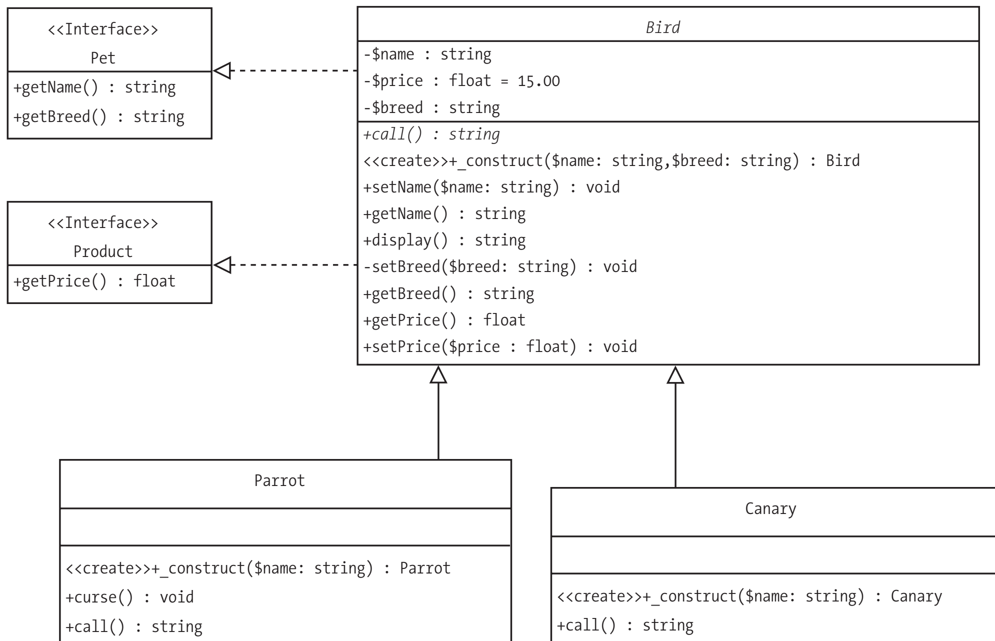


Figure 2-4. *Updated class diagram showing implementation of interfaces by the* `Bird` *class*

Using interfaces can help you keep your classes consistent with one another. For example, if you need to write classes to represent additional pets for sale by the pet store, you can, by implementing `Pet` and `Product` in those classes, guarantee that they will have the same methods that `Bird` and its subclasses do.

2-10. Using Class Destructors

In PHP 5, classes can have *destructors* as well as constructors. A destructor is simply a method that is guaranteed to be invoked whenever an instance of the class is removed from memory, either as the result of a script ending or because of a call to the `unset()` function. For example, suppose that when a user of your e-commerce website—represented by an instance of a `SiteUser` class—leaves the site, you want to make sure that all the user's preference data is saved to the site's user database. Suppose further that `SiteUser` already has a `savePrefs()` method that accomplishes this task; you just need to make sure it is called when the user logs out. In that case, the class listing might include something like the following.

The Code

```
class SiteUser
{
  //  class variables...

  public function __construct()
  {
    //  constructor method code...
  }

  //  other methods...

  public function savePref()
  {
    //  code for saving user preferences...
  }

  //  Here's the class destructor:

  public function __destruct()
  {
    $this->savePrefs();
  }
}
```

How It Works

As you can see from this listing, all you need to do is to add a `__destruct()` method to the class containing whatever code you want to be executed when an instance of the class ceases to exist.

"MAGIC" METHODS

Method and function names beginning with a double underscore—such as `__construct()`, `__destruct()`, and `__autoload()`—are reserved in PHP and are often referred to as *magic*. Several others, such as those you already looked at in this chapter, are invoked automatically in response to certain events. (For this reason, you should *never* name your own functions or methods with two leading underscores.)

Here is a listing of most of these magic methods, along with a brief description of each:

- `__construct()`: Called when a new instance of the class is created.
- `__destroy()`: Called when an instance of the class passes out of memory; this happens when you either `unset()` the instance or a script finishes running.
- `__autoload()`: Called when you refer to a class for the first time (for example, call its constructor, call one of its static methods, and so on).
- `__clone()`: Called when you create a copy of an object using the `clone` keyword.
- `__get()` and `__set()`: Called when you attempt to get or set an object property that is not defined for that object. `__get()` takes a single parameter, which represents the name of the property; `__set()` takes two parameters: the name of the property you tried to set and the value you tried to assign to it.
- `__call()`: Called when you try to call an undefined method. It takes two arguments: the method name that was used and an array containing any values that were passed to the method.
- `__sleep()` and `__wakeup`: `__sleep()` is called when you try to `serialize()` an object. This is useful when (for example) you need to close a database connection used by an object before saving it or when you want to save only some of an object's properties. This method should return an array containing the names of the variables you want to be serialized. `__wakeup()` is called when you `unserialize()` an object; you can use it to re-establish database connections or reinitialize the object in whatever other ways you require.
- `__toString()`: Called when a string representation of the object is required.

Of course, any of these magic methods comes into play only if you have defined it for a given class. You should also note that they cannot be called directly, only via the event they are supposed to intercept. For more information on magic methods and their uses, see the PHP manual or *PHP 5 Objects, Patterns, and Practice* by Matt Zandstra (Apress, 2004).

2-11. Using Exceptions

PHP 5 introduces a much-improved mechanism for handling errors. Like many of PHP 5's new features, *exceptions* may already be familiar to you if you are a Java programmer. If you are not, here is your chance to become acquainted with them.

The purpose of exceptions is to help segregate error-handling code from the parts of your application that are actually doing the work. A typical situation is working with a database. The following is a bit of code showing how you might do this in PHP 4 or how you might do this in PHP 5 without using exceptions:

```
<?php
  $connection = mysql_connect($host, $user, $password)
    or die("Error #". mysql_errno() .": " . mysql_error() . ".");

  mysql_select_db($database, $connection)
    or die("Error: could not select database $database on host $hostname.");

  $query = "SELECT page_id,link_text,parent_id
            FROM menus
            WHERE page_id='$pid'";

  $result = mysql_query($query)
    or die("Query failed: Error #". mysql_errno() .": " . mysql_error() . ".");

  if(mysql_num_rows($result) == 0)
    echo "<h2>Invalid page request -- click <a href=\""
            . $_SERVER["PHP_SELF"] . "?pid=1\">here</a> to continue.</h2>\n";
  else
  {
    $value = mysql_fetch_object($result)
      or die("Fetch operation failed: Error #". mysql_errno()
              . ": " . mysql_error() . ".");
    // ...
  }

  //  etc. ...
?>
```

Notice that every time you interact with the database, you include an explicit error check. This is good in that you are practicing defensive programming and not leaving the user with a blank page or half-completed page in the event of an error. However, it is not so good in that the error checking is mixed up with the rest of the code. Using PHP 5 exception handling, the same block might look something like the following example.

The Code

```
<?php
  function errors_to_exceptions($code, $message)
  {
    throw new Exception($code, $message);
  }

  set_error_handler('errors_to_exceptions');

  try
  {
    $connection = mysql_connect($host, $user, $password);
```

```
    mysql_select_db($database, $connection);

    $query = "SELECT page_id,link_text,parent_id
              FROM menus
              WHERE page_id='$pid'";

    $result = mysql_query($query);

    if(mysql_num_rows($result) == 0)
      echo "<h2>Invalid page request -- click <a href=\""
          . $_SERVER["PHP_SELF"] . "?pid=1\">here</a> to continue.</h2>\n";
    else
    {
      $value = mysql_fetch_object($result);
      // ...
    }

    //  etc. ...
  }
  catch Exception $e
  {
    printf("<p>Caught exception: %s.</p>\n", $e->getMessage());
  }
?>
```

How It Works

The basic structure for exception handling looks like this:

```
try
{
  perform_some_action();
  if($some_action_results_in_error)
    throw new Exception("Houston, we've got a problem...");

  perform_another_action();
  if($other_action_results_in_error)
    throw new Exception("Houston, we've got a different problem...");
}
catch Exception $e
{
  handle_exception($e);
}
```

The try block contains any code that is to be tested for exceptions. When an exception is thrown, either automatically or by using the throw keyword, script execution immediately passes to the next catch block. (If PHP cannot find any catch block following the try block, then it will issue an Uncaught Exception error.)

Having to use throw to signal an exception manually each time an error condition is encountered really is not more efficient or cleaner than using repeated if or or die() constructs. In some programming languages, such as Java, the most common exceptions are thrown automatically, and all you have to worry about is supplying the code that goes inside the try and catch blocks. However, because PHP 5 has to maintain backward compatibility with older code, the traditional error-production mechanism takes precedence. To override this behavior, you can use the set_error_handler() function to call a function to be executed whenever an error is generated, in place of PHP's default behavior. This function takes the name of an error-handling function as its sole argument and causes the function with this name to be executed whenever PHP raises an error. (Note that the name of the function to be executed is passed to set_error_handler() as a string.) In the second version of the database code snippet, you have defined a function named errors_to_exceptions, which simply throws an exception.

You may have noticed that when you throw an exception, you actually use the throw keyword followed by an Exception object. The definition of the Exception class is as follows, and Figure 2-5 shows a UML representation of this class:

```
<?php
  class Exception
  {
    protected $message = 'Unknown exception';   // Exception message
    protected $code = 0;                        // Exception code (user-defined)
    protected $file;                            // Filename
    protected $line;                            // Line number

    function __construct($message = null, $code = 0);

    final function getMessage();                // Message
    final function getCode();                   // Code
    final function getFile();                   // Filename
    final function getLine();                   // Line number
    final function getTrace();                  // Backtrace (array)
    final function getTraceAsString();          // Backtrace (string)

    function __toString();                      // Formatted string for display
  }
?>
```

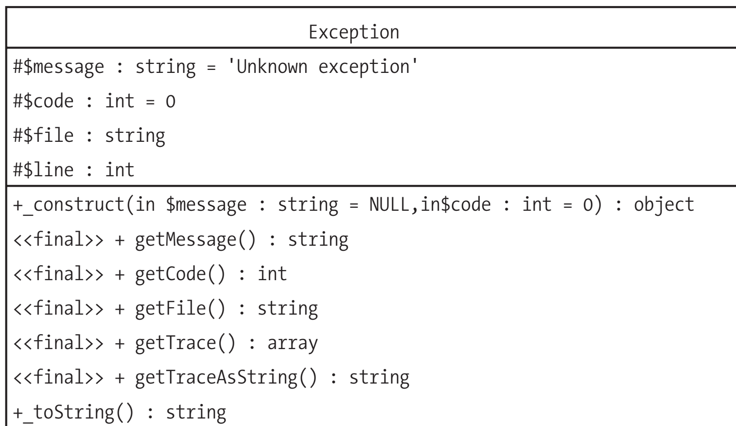


Figure 2-5. *The PHP 5* `Exception` *class (UML representation)*

■ **Tip** You can define a `__toString()` method for any class. It is generally *not* a good idea to declare this method as `final`. (The only time you might want this to happen is in a class that is itself declared final, in which case there's no need.)

Like any well-designed class, the properties of an `Exception` are not directly accessible to calling code, and their values must be obtained using the get methods shown. Only the `$message` and `$code` can be set by the user, and this must be done via the class constructor. You can extend `Exception`, which can be a good idea when you are dealing with several classes or different sets of classes with dissimilar functionality. Note that all the get methods are `final` and thus cannot be overridden in any subclasses. The `__toString()` method is a "magic" method (as discussed earlier in this chapter) that is called whenever you try to output an instance of `Exception` directly using `echo` or `print`. You can override this method in an `Exception` subclass.

■ **Tip** For some code examples using multiple `Exception` subclasses, see *MySQL Database Design and Optimization* (Apress, 2004).

Some PHP object-oriented libraries and extensions supply their own exception classes. For example, the Document Object Model (DOM) extension implements a `DOMException` class and raises an instance of this class whenever an illegal DOM operation is attempted. When using a new PHP 5 class library for the first time, be sure to check whether it includes its own `Exception` subclasses.

Getting Information About Classes and Objects

What do you do when you need to use one or more classes for which no documentation is available? Since PHP is an interpreted rather than a compiled language, you will usually be able to turn to the source code; however, sometimes this is not possible:

- You may not have access to all the files making up an application.
- Source code can be encrypted using tools such as IonCube.
- You may need to work with an extension that was compiled from C or Java and for which neither complete documentation nor the original sources is available.
- The sources may be available, but you might not be a C or Java programmer or simply not have time to study them in depth.
- You may be writing some highly abstracted code and not know ahead of time whether a given class is available or which of two or more classes might be.

To help you in these situations, PHP 5 provides two mechanisms for obtaining information about classes, class members, and objects. The *class and object functions*, which we will discuss first, are mostly the same as those found in PHP 4, with a few additions and enhancements. These can provide you with basic information about the availability of classes, interfaces, and their public members. For more serious reverse-engineering, PHP 5 has introduced a set of interfaces and classes known collectively as the *Reflection API*. Using this application programming interface (API), it is possible to find out just about everything you might want to know about an interface or a class and its members, including private and protected members and the arguments expected by class methods. In fact, using the Reflection API classes, it is possible to reverse-engineer complete extensions.

Using Class and Object Functions

PHP's class and object functions are fairly straightforward and simple to use. You will find most of them in Table 2-1.

Table 2-1. *PHP 5 Class and Object Functions (Partial Listing)*

Function	Arguments/Types	Description/Purpose
`class_exists()` is	string `$class`, bool `$autoload=TRUE`	Returns `TRUE` if the class named `$class` has been defined. Attempts to call `__autoload()` if the class is not defined unless `$autoload` set to `FALSE`.[1]
`class_implements()`	object `$object`	Returns an array containing the names of all interfaces implemented by the class of which `$object` is an instance.[2]
`class_parents()`	object `$object`	Returns an array containing the names of all classes from which `$object` descends (does not include the name of the class of which `$object` is an instance).[2]
`get_class_methods()`	string `$class` or object `$object`	Returns an array of class public method names. Can take either the name of a class or an instance of the class as an argument.
`get_class_vars()`	string `$class`	Returns an array of default public properties of the class named `$class`.
`get_class()`	object `$object`	Returns the name of the class of an object.
`get_declared_classes()`	`void`	Returns an array containing the names of all classes defined in the current script.
`get_declared_interfaces()`	`void`	Returns an array of the names of all interfaces defined in the current script.[2]
`get_object_vars()`	object `$object`	Returns an associative array whose keys are the names of the properties of `$object` and whose values are the values of those properties.
`get_parent_class()`	string `$class` or object `$object`	Returns the name of the parent class of the `$class` or `$object`.
`interface_exists()`	string `$interface`, bool `$autoload=TRUE`	Returns `TRUE` if `$interface` is defined in the current script. Unless `$autoload` is set to `FALSE`, this function will attempt to invoke `__autoload()` (if defined).[2]

Function	Arguments/Types	Description/Purpose
is_a()	object $object, string $class	Returns TRUE if $object is an instance of $class or one of its subclasses.
is_subclass_of()	object $object, string $class	Returns TRUE if $object is a descendant of $class. As of PHP 5.0.3, the first argument may also be the name of a class (as a string).[1]
method_exists()	object $object, string $method	Returns TRUE if $object has a method named $method.

Notes: (1) changed in PHP 5 (2) added in PHP 5

The next few recipes assume you have defined the set of classes (Bird, Parrot, Canary) and interfaces (Pet and Product) shown earlier in Figure 2-4.

2-12. Checking for the Existence of Classes and Interfaces Using class_exists() and interface_exists()

The following defines an additional class Shape that has some static and public variables (the reason for this will become apparent shortly):

```
class Shape
{
  const NUM_SIDES_TRIANGLE = 3;
  const NUM_SIDES_SQUARE = 4;
  const NUM_SIDES_PENTAGON = 5;
  const NUM_SIDES_HEXAGON = 6;

  static $shapeNames = array('triangle', 'quadrilateral', 'pentagon', 'hexagon');

  public $numberOfSides;
  public $perimeter;

  private $name;

  function __construct($numberOfSides = 3, $sideLength = 10)
  {
    if($numberOfSides < 3)
      $this->numberOfSides = 3;
    elseif($numberOfSides > 6)
      $this->numberOfSides = 6;
    else
      $this->numberOfSides = $numberOfSides;

    $this->setName( Shape::$shapeNames[$this->numberOfSides - 3] );
```

```
    $this->perimeter = ($sideLength < 1 ? 1 : $sideLength) * $this->numberOfSides;
  }

  protected function setName($name)
  {
    $this->name = $name;
  }

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

Let's also create some class instances to use in the tests:

```
$polly = new Parrot('Polynesia');
$tweety = new Canary('Tweety');
$square = new Shape(Shape::NUM_SIDES_SQUARE);
```

Next you will look at the `class_exists()` and `interface_exists()` functions, which do pretty much what their names sound like; they tell you whether a given class or interface is defined. Each takes a string and returns a `TRUE` or `FALSE` value.

The Code

```
$classes = array('Parrot', 'Canary', 'Bird', 'Monkey', 'Pet');
$interfaces = array('Pet', 'Product', 'Customer', 'Bird');

print "<p>";
foreach($classes as $class)
  printf("The class '%s' is %sdefined.<br />\n",
         $class,
         class_exists($class, FALSE) ? '' : 'un');

print "</p>\n<p>";

foreach($interfaces as $interface)
  printf("The interface '%s' is %sdefined.<br />\n",
         $interface,
         interface_exists($interface, FALSE) ? '' : 'un');

print "</p>\n";
```

Here is the output:

```
The class 'Parrot' is defined.
The class 'Canary' is defined.
The class 'Bird' is defined.
The class 'Monkey' is undefined.
The class 'Pet' is undefined.

The interface 'Pet' is defined.
The interface 'Product' is defined.
The interface 'Customer' is undefined.
The interface 'Bird' is undefined.
```

2-13. Listing Methods and Interfaces Using get_class_methods()

You can use `get_class_methods()` to obtain a list of the public methods exposed by either a class or a class instance; you can also use this function with interfaces, as demonstrated in this example:

The Code

```
printf("<p>Parrot class methods: %s</p>\n",
        implode(', ', get_class_methods('Parrot')));
printf("<p>\$polly instance methods: %s</p>\n",
        implode(', ', get_class_methods($polly)));
printf("<p>Shape class methods: %s</p>\n",
        implode(', ', get_class_methods('Shape')));
printf("<p>Pet interface methods: %s</p>\n",
        implode(', ', get_class_methods('Pet')));
```

Here is the output:

```
Parrot class methods: call, __construct, curse, setBreed, setName, ➥
 setPrice, getName, getBreed, getPrice, display

$polly instance methods: call, __construct, curse, setBreed, setName, ➥
 setPrice, getName, getBreed, getPrice, display

Shape class methods: __construct, getName

Pet interface methods: getName, getBreed
```

Notice that the array returned by `get_class_methods()` contains names of public methods either defined in the class or inherited from a parent class. Private and protected methods are not listed.

2-14. Obtaining Variable Names

PHP has two functions for obtaining the names of public variables; you can use `get_class_variables()` with classes (it takes the name of the class as its argument), and you can use `get_object_variables()` with objects (it acts on an instance of a class). You might wonder why two functions exist instead of one function that can act on either a class or a class instance, as there is for class and object methods. Let's compare the results using these with the class `Shape` and the `Shape` instance `$square` and see what happens.

The Code

```
printf("<pre>Shape class variables: %s</pre>",
        print_r(get_class_vars('Shape'), TRUE));
printf("<pre>\$square object variables: %s</pre>",
        print_r(get_object_vars($square), TRUE));
```

As you can see here, the output of these two functions can be markedly different, even when comparing the variables of a class with those of an instance of the same class:

```
Shape class variables: Array
(
    [numberOfSides] =>
    [perimeter] =>
    [shapeNames] => Array
        (
            [0] => triangle
            [1] => quadrilateral
            [2] => pentagon
            [3] => hexagon
        )

)

$square object variables: Array
(
    [numberOfSides] => 4
    [perimeter] => 40
)
```

How It Works

Static variables are shown in the output of `get_class_variables()` but not in that of `get_object_variables()`. In the case of `Shape`, `$numberOfSides` and `$perimeter` have no default value, so no variable is shown; however, when you call `get_object_variables()` on an instance of `Shape`, the values set by the class constructor for these variables are reported. Variables that are declared as `private` or `protected` are not reported by either of these functions.

Tip When you call `get_class_variables()` using a class that has no public variables, or `get_object_variables()` on an instance of that class, the value returned is an empty array. However, it is possible to view an object's `private` and `protected` variables using `print_r()` or `var_dump()`.

2-15. Determining Whether an Object Is an Instance of a Particular Class

You can use the `is_a()` function to determine whether an object is an instance of a given class. This function takes two parameters, an object (`$object`) and the name of a class or an interface (`'name'`), and returns `TRUE` if any of the following conditions is true:

- `$object` is an instance of a class named `name`.
- `$object` is an instance of a class that descends from a class named `name`.
- `$object` is an instance of a class implementing an interface named `name`.
- `$object` is an instance of a class that descends from a class implementing an interface named `name`.

This is one of those things that sounds more complicated than it really is, so perhaps the following code will help make it clearer.

The Code

```
print "<p>";
printf("\$polly is %sa Parrot.<br />\n",
        is_a($polly, 'Parrot') ? '' : 'not ');
printf("\$polly is %sa Canary.<br />\n",
        is_a($polly, 'Canary') ? '' : 'not ');
printf("\$polly is %sa Bird.<br />\n",
        is_a($polly, 'Bird') ? '' : 'not ');
printf("\$polly is %sa Pet.<br />\n",
        is_a($polly, 'Pet') ? '' : 'not ');
print "</p>\n";
```

Here is the output:

```
$polly is a Parrot.
$polly is not a Canary.
$polly is a Bird.
$polly is a Pet.
```

How It Works

Since `$polly` is an instance of `Parrot`, the first `is_a()` test is true. It is not an instance of the `Canary` class, and it does not descend from `Canary`, so the second test is false. `$polly` is an instance of `Parrot`, which extends the `Bird` class, so the third test is true. `Bird` implements the `Pet` interface, so the fourth test using `is_a()` also returns `TRUE`.

Variations

`is_a()` answers the question, Does object A descend from class B? A closely related question is, What is the parent class of class C? You can answer this with the help of `get_parent_class()`. This function takes the name of a class and returns the name of its parent, if it has one; otherwise, it returns an empty string. It is not difficult to write a bit of recursive code that takes care of tracing the complete inheritance trail of a class:

```
function write_parents($class)
{
  $parent = get_parent_class($class);

  if($parent != '')
  {
    printf("<p>%s is a child of %s.</p>\n", $class, $parent);
    write_parents($parent);
  }
  else
    printf("<p>%s has no parent class.</p>\n", $class);
}

write_parents('Canary');
```

This yields the following result:

```
Canary is a child of Bird.
Bird has no parent class.
```

However, PHP 5 introduces a simpler way to accomplish this using the `class_parents()` function. As a bonus, an analogous `class_interfaces()` function returns a list of all interfaces implemented by a class. Notice that both of these functions take an instance of the class as a parameter:

```
printf("<p>Canary class parents: %s</p>\n",
        implode(', ', class_parents($tweety)));
printf("<p>Canary class implements: %s</p>\n",
        implode(', ', class_implements($tweety)));
```

Here is the output:

```
Canary class parents: Bird

Canary class implements: Product, Pet
```

2-16. Listing Currently Loaded Interfaces and Classes

You can obtain lists of the interfaces and classes currently loaded with the functions `get_declared_interfaces()` and `get_declared_classes()`, respectively.

The Code

```
printf("<p>Interfaces currently available: %s</p>",
        implode(', ', get_declared_interfaces()));
printf("<p>Classes currentlyavailable: %s</p>",
        implode(', ', get_declared_classes()));
```

The following is the output generated by this code on one of our test systems running PHP 5.0.4 under Apache 1.3.33 on Windows 2000. Your results are likely to be different, depending on which operating system and web server software you are using, as well as which PHP extensions you have loaded at the time.

```
Interfaces currently available: Traversable, IteratorAggregate,
 Iterator, ArrayAccess, Reflector, RecursiveIterator, SeekableIterator,
 Pet, Product


Classes currently available: stdClass, Exception, ReflectionException,
 Reflection, ReflectionFunction, ReflectionParameter, ReflectionMethod,
 ReflectionClass, ReflectionObject, ReflectionProperty,
ReflectionExtension, COMPersistHelper, com_exception,
 com_safearray_proxy, variant, com, dotnet, RecursiveIteratorIterator,
 FilterIterator, ParentIterator, LimitIterator, CachingIterator,
CachingRecursiveIterator, ArrayObject, ArrayIterator,
DirectoryIterator, RecursiveDirectoryIterator, SQLiteDatabase,
SQLiteResult, SQLiteUnbuffered, SQLiteException, __PHP_Incomplete_Class,
 php_user_filter, Directory, DOMException, DOMStringList, DOMNameList,
 DOMImplementationList, DOMImplementationSource, DOMImplementation,
 DOMNode, DOMNameSpaceNode, DOMDocumentFragment, DOMDocument,
 DOMNodeList, DOMNamedNodeMap, DOMCharacterData, DOMAttr, DOMElement,
 DOMText, DOMComment, DOMTypeinfo, DOMUserDataHandler, DOMDomError,
 DOMErrorHandler, DOMLocator, DOMConfiguration, DOMCdataSection,
 DOMDocumentType, DOMNotation, DOMEntity, DOMEntityReference,
 DOMProcessingInstruction, DOMStringExtend, DOMXPath, SimpleXMLElement,
SimpleXMLIterator, SWFShape, SWFFill, SWFGradient, SWFBitmap, SWFText,
```

```
SWFTextField, SWFFont, SWFDisplayItem, SWFMovie, SWFButton, SWFAction,
SWFMorph, SWFSprite, SWFSound, mysqli, mysqli_result, mysqli_stmt,
PDFlibException, PDFlib, tidy, tidyNode, XSLTProcessor, Shape, Bird,
Parrot, Canary
```

Variations

The `get_declared_classes()` function can be handy when writing scripts to run in places where you do not know ahead of time which extensions or programming classes might be available. For example, suppose you need to process an XML file, but you are not sure which XML APIs might be available:

```
$xmlfile = '/xmlfiles/myfile.xml';
$classes = get_declared_classes();

if( in_array('SimpleXMLElement', $classes) )
{
  $xmldoc = simplexml_load_file($xmlfile);
  //  process XML using SimpleXML API...
}
elseif( in_array('DOMDocument', $classes) )
{
  $xmldoc = new DOMDocument();
  $xmldoc->load($xmlfile);
  //  process XML using DOM API...
}
else
{
  //  process XML using Expat or other means...
}
```

You can use these functions with predefined classes as well as those you have written or included yourself. Here is an example showing what you obtain by using `class_parents()`, `class_implements()`, `get_class_methods()`, and `get_class_variables()` in order to obtain information about the built-in `ArrayIterator` class:

```
$class = 'ArrayIterator';
eval("@\$object = new \$class();");

printf("<p>%s class parents: %s</p>\n",
        $class,
        print_r(class_parents($object), TRUE));
printf("<p>%s class implements: %s</p>\n",
        $class,
        implode(', ', class_implements($object)));
printf("<p>%s class methods: %s</p>\n",
        $class,
        implode(', ', get_class_methods($class)));
```

```
printf("<p>%s class variables: %s</p>",
        $class,
        print_r(get_class_vars($class), TRUE));
```

We have "cheated" here and used `print_r()` with the arrays that we knew would be empty so you could see that these are in fact empty arrays and not empty strings or NULLs:

```
ArrayIterator class parents: Array()

ArrayIterator class implements: Iterator, Traversable, ArrayAccess, SeekableIterator

ArrayIterator class methods: __construct, offsetExists, offsetGet,➥
 offsetSet, offsetUnset, append, getArrayCopy, count, rewind, current,➥
 key, next, valid, seek

ArrayIterator class variables: Array()
```

Using the Class Reflection API

PHP 5's class and object functions do not tell you anything about the classes' internals, only about their public members. To make the most of a class, you really need to know about its private and protected members, as well as about the parameters expected by its methods. The Reflection API comes in handy here; it allows you to perform thorough reverse-engineering of any class or interface.

It has been said that he who understands a thing by breaking it apart loses the thing he understands, but in some programming situations you want or need to do exactly that. While the class and object functions you looked at in the previous set of recipes can be useful in this regard, the Reflection API has a number of advantages over those functions:

- It is completely object-oriented.
- You can obtain detailed information about extensions.
- It allows you to access private and protected variables more easily and to obtain default values of properties of classes (without having to instantiate an object using the default constructor).
- It is possible to obtain complete method signatures (except for return types).
- You can examine private and protected methods, which is not possible using the class and object functions.

The Reflection API (as illustrated in Figure 2-6) consists of eight classes, all of which except for the `Reflection` class implement the `Reflector` interface, which is defined as shown here:

```
Interface Reflector
{
  public static function export();
  public function __toString();
}
```

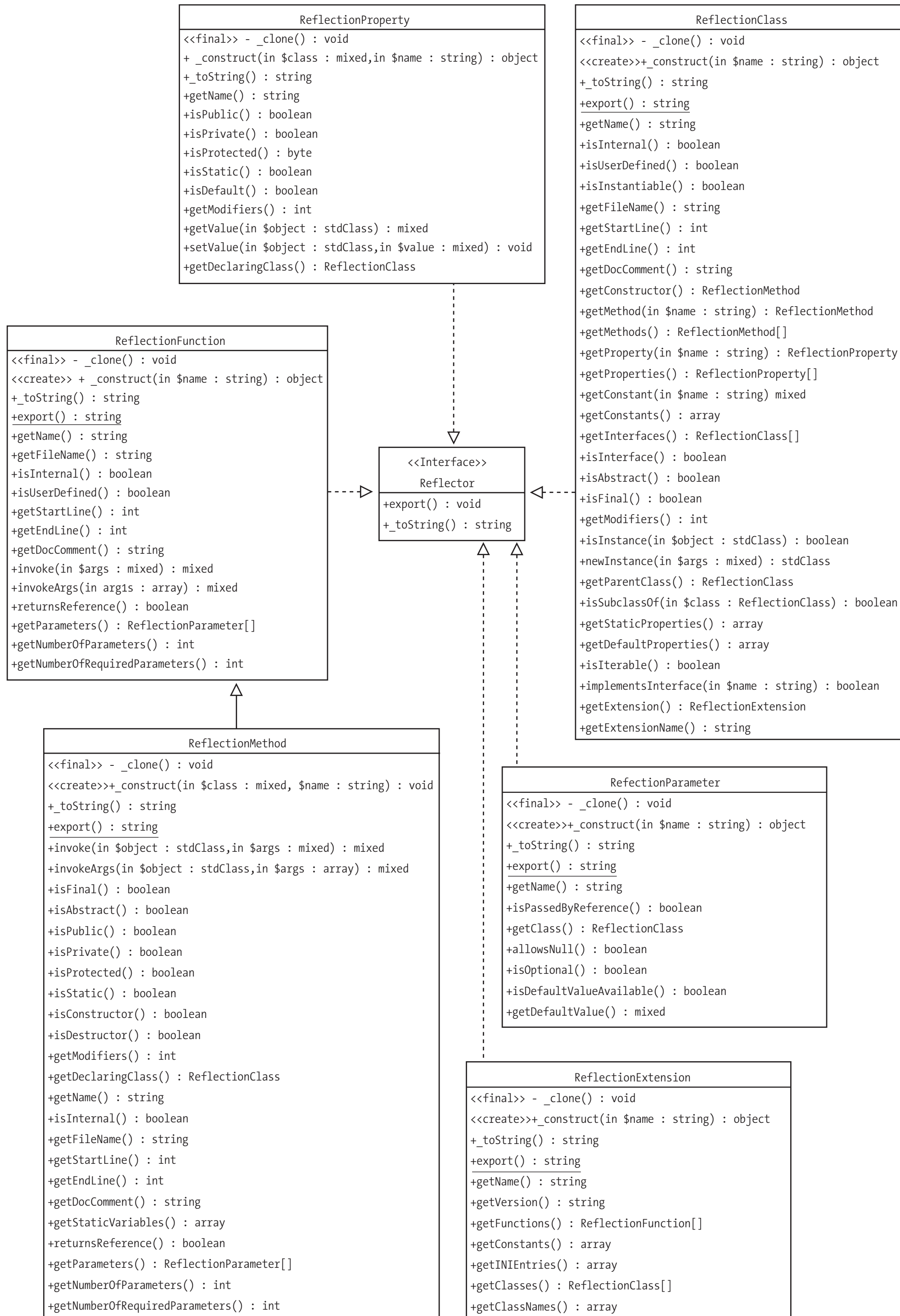


Figure 2-6. *PHP 5 Reflection API classes (UML diagram)*

The Reflection API classes are as follows:

- `Reflection`: This class implements as a static method the `export()` method defined by `Reflector`, although it does not actually implement the `Reflector` interface. You can use this method to dump all the methods and/or properties of a class, extension, property, method, or parameter.
- `ReflectionClass`: This class models a PHP 5 class and exposes methods for accessing nearly all aspects of the class, including its properties, methods, any parent classes that it extends or interfaces it implements, whether it is abstract or final, and so on. Note that you can also use the `ReflectionClass` to represent an interface.
- `ReflectionFunction`: Represents a function.
- `ReflectionMethod`: Extends the `ReflectionFunction` class and is used to model a class method.
- `ReflectionParameter`: Represents a parameter of a function or method.
- `ReflectionProperty`: Models a class property.
- `ReflectionExtension`: Represents a PHP extension.
- `ReflectionException`: Represents an exception thrown by one of the Reflection API classes. This class actually extends the `Exception` class.

We will not list all the methods of these classes here; you will get to see some of them used in later recipes, and for the rest you can consult Figure 2-6 or refer to the PHP manual (`http://docs.php.net/en/language.oop5.reflection.html`).

2-17. Obtaining a Dump of the Reflection API

If you are feeling adventurous, you can use `export()` to get a dump of the Reflection API. This example shows how to do this, using the `Shape` class defined earlier, which we have saved to a file named `Shape.class.php`.

The Code

```
<?php
  //  file: reflection-export-1.php
  //  simple Reflection::export() example

  //  include class file
  require_once('./Shape.class.php');

  //  create new instance of ReflectionClass
  $rc = new ReflectionClass('Shape');

?><pre><?php

  //  dump class info
```

```
  Reflection::export($rc);

?></pre>
```

How It Works

The `ReflectionClass` constructor takes the name of the class you want to examine (as a string value). To get a dump of all class members, simply pass the `ReflectionClass` object you have just created to the static `Reflection::export()` method. The result contains all properties and methods of the class along with all parameters of those methods, and it even includes line numbers and comments from the source code for the class.

Here is the output:

```
/**
  * An example class for class/object functions
  * and Reflection examples - contains a mix of
  * public/private/protected/static members,
  * constants, etc.
  */
Class [  class Shape ] {
  @@ /home/www/php5/ch2/Shape.class.php 11-66

  - Constants [4] {
    Constant [ integer NUM_SIDES_TRIANGLE ] { }
    Constant [ integer NUM_SIDES_SQUARE ] { }
    Constant [ integer NUM_SIDES_PENTAGON ] { }
    Constant [ integer NUM_SIDES_HEXAGON ] { }
  }

  - Static properties [1] {
    Property [ public static $shapeNames ]
  }

  - Static methods [0] {
  }

  - Properties [3] {
    Property [  public $numberOfSides ]
    Property [  public $perimeter ]
    Property [  private $name ]
  }

  - Methods [3] {
    /**
    * Class constructor
    * input params:
    * int $numberOfSides, int $sideLength
```

```
    */
    Method [   public method __construct ] {
      @@ /home/www/php5/ch2/Shape.class.php 32 - 44

      - Parameters [2] {
        Parameter #0 [  $numberOfSides = 3 ]
        Parameter #1 [  $sideLength = 10 ]
      }
    }

    /**
    * Sets the name value
    * Input param:
    * string $name
    */
    Method [  protected method setName ] {
      @@ /home/www/php5/ch2/Shape.class.php 52 - 55

      - Parameters [1] {
        Parameter #0 [  $name ]
      }
    }

    /**
    * Retrieves the name value
    * returns string
    */
    Method [  public method getName ] {
      @@ /home/www/php5/ch2/Shape.class.php 62 - 65
    }
  }
}
```

Variations

That wasn't so difficult, was it? You can accomplish the same thing for any PHP 5 class by replacing `Shape` with the class name. This includes built-in classes and those made available by extensions. However, while dumping a pile of data about a class can occasionally be useful during development, this does not use the real power of the Reflection API, which is that it exposes fully object-oriented interfaces for almost any aspect of a class or an object that you might need to use. For example, harking back to the pet shop scenario, suppose you have defined a number of classes implementing the `Pet` interface. Rather than having a single `petCall()` method in `Pet` that you have extended for each class, let's also suppose you have defined a method performing this function but that it is different for each subclass. See Figure 2-7 for an abbreviated UML representation.

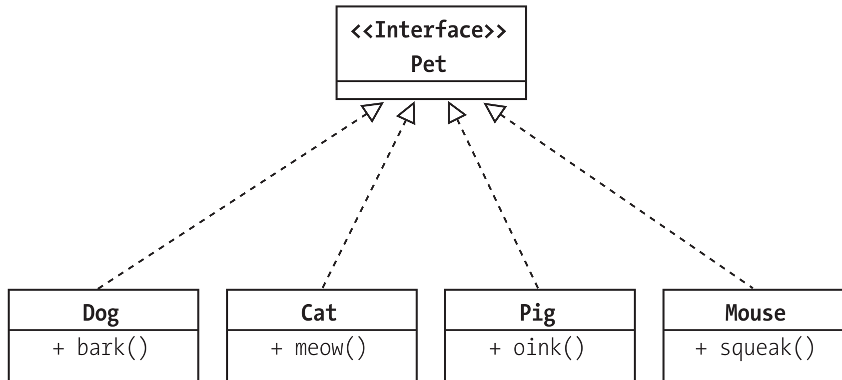


Figure 2-7. *Modified set of* `Pets`

2-18. Performing Dynamic Class Instantiation

Now imagine that you have to write a module that works with some existing code that uses a `Pet` class but you have no way of knowing ahead of time which one it might be. Using the Reflection API's `ReflectionClass` and `ReflectionMethod` classes, the following code shows one way you could solve this problem.

The Code

```
<?php
  //  Existing code has created an instance of a class
  //  implementing the Pet interface as the variable $pet...
  //  This array uses the class names as keys and the
  //  'make a noise' method names as values
  $noises = array('Dog' => 'bark', 'Cat' => 'meow',
                  'Pig' => 'oink', 'Mouse' => 'squeak');

  //  Now you need for the pet to make a noise...

  //  First you need to get the name of the class, and then
  //  create a corresponding instance of ReflectionClass
  $pet_name = get_class($pet);
  $pet_rc = new ReflectionClass($pet_name);

  //  Get the name of the correct method based on the name of the class
  //  (if no match is found, set the method name to NULL):


  //  If there was a match on the class name, create
  //  an instance of ReflectionMethod using the class name
  //  and corresponding method name, and then invoke this method
  //  by calling ReflectionMethod's invoke() method
  if( array_key_exists($pet_name, $noises) )
```

```
  {
    $pc_method = new ReflectionMethod($pet_rc, $noises[$pet_name]);
    $pc_method->invoke($pet);
  }
  else  //  Otherwise, indicate that there's no sound for this Pet
    print "This pet does not make a sound";
?>
```

The `ReflectionMethod` class constructor takes two arguments: the name or an instance of the class and the name of the method. You can use `ReflectionMethod::invoke()` to simulate a call to the corresponding method; this method requires at least one argument, either NULL (if the referenced object method is static) or an instance of the class. In other words, using the `Parrot` class defined earlier, you can invoke its `birdCall()` method dynamically like so:

```
<?php
  require_once('./bird-interface.php');

  $class = 'Parrot';
  $method = 'birdCall';
  $rm = new ReflectionMethod($class, $method);
  $rm->invoke( new $class('Polly') );

  //  Output is: Polly says *squawk*.
?>
```

The output is identical to that produced by the following code:

```
<?php
  require_once('./bird-interface.php');

  $polly = new Parrot('Polly');
  $polly->birdCall();
?>
```

Note that you can also invoke a class constructor dynamically simply by using `new $class()`, where `$class` is the name of the class.

2-19. Using the Reflection API to Deconstruct the Shape Class

We will finish this chapter with a more involved example of using the Reflection API to deconstruct the `Shape` class in an object-oriented fashion by using the `ReflectionClass`, `ReflectionMethod`, and `ReflectionParameter` classes. Let’s dive right into the code, and we will explain what is happening along the way.

The Code

```
<?php
  //  file: reflection-example-2.php
  //  more involved Reflection API example
```

```
//  Include the proper class file
$class = 'Shape';

require_once("./$class.class.php");

//  Create a new ReflectionClass instance
$rc = new ReflectionClass($class);

//  Display the name of the class
printf("<p>Name: *%s*<br />\n", $rc->getName());

//  Display the file the class is defined in, and
//  the beginning and ending line numbers
printf("Defined in file '%s', lines %d - %d<br />\n",
        $rc->getFileName(),
        $rc->getStartLine(),
        $rc->getEndLine());
```

It is possible to get documentation comments from the class source; both `ReflectionClass` and `ReflectionMethod` provide a `getDocComment()` method. (Note that these are both instance methods.) `ReflectionClass::getDocComment()` returns a string containing the multiline comment immediately preceding the class definition; `ReflectionMethod::getDocComment()` returns a string containing the multiline comment immediately preceding the method declaration in the source. In both cases, the string includes the opening and closing `/*` and `*/` comment delimiters. If there is no matching comment, then `getDocComment()` returns an empty string.

```
printf("<p>Contains the comments:<pre>%s</pre></p>",
       $rc->getDocComment());
```

`ReflectionClass` has a number of boolean methods that tell you whether the class is public, private, static, abstract, and so on. In addition, because this class can also model an interface, the `isInterface()` method returns `TRUE` if you are examining an interface and `FALSE` if you are introspecting a class.

```
printf("%s is %san interface.<br />\n",
       $rc->getName(),
       $rc->isInterface() ? '' : 'not ');

printf("%s is %sinstantiable.<br />\n",
       $rc->getName(),
       $rc->isInstantiable() ? '' : 'not ');

printf("%s is %sabstract.<br />\n",
       $rc->getName(),
       $rc->isAbstract() ? '' : 'not ');

printf("%s is %sfinal.</p>\n",
       $rc->getName(),
       $rc->isFinal() ? '' : 'not ');
```

The getConstants() method returns an associative array of all class constants. This array's keys are the names of the constants, and their values are those of the corresponding constants.

```
$constants = $rc->getConstants();
$num_constants = count($constants);

printf("%s defines %d constant%s",
        $rc->getName(),
        $num_constants == 0 ? 'no' : $num_constants,
        $num_constants != 1 ? 's' : '');

if($num_constants > 0)
  printf(":<pre>%s</pre>", print_r($constants, TRUE));
```

The instance method ReflectionClass::getProperties() returns an array of class properties (actually an array of ReflectionProperty objects). Here you will just supply each of these in turn as a parameter to the static Reflection::export() method to obtain a dump of the property's attributes, but you can also employ a number of ReflectionProperty methods to determine the property's name, access, whether it is static, and how to get or set the value currently stored by the property represented.

Note Each of the elements in the array returned by getConstants() is an instance of ReflectionParameter.

```
$props = $rc->getProperties();
$num_props = count($props);

printf("%s defines %d propert%s",
        $rc->getName(),
        $num_props == 0 ? 'no' : $num_props,
        $num_props == 1 ? 'y' : 'ies');

if($num_props > 0)
{
  print ':';
  foreach($props as $prop)
  {
    print "<pre>";
    Reflection::export($prop);
    print "</pre>";
  }
}
```

The ReflectionClass method getMethods() returns an array of ReflectionMethod objects, each corresponding to a class method.

```
$methods = $rc->getMethods();
$num_methods = count($methods);

printf("%s defines %d method%s<br />\n",
        $rc->getName(),
        $num_methods == 0 ? 'no' : $num_methods,
        $num_methods != 1 ? 's' : '');

if($num_methods > 0)
{
  print '<p>';

  foreach($methods as $method)
  {
    printf("%s%s%s%s%s%s() ",
            $method->isFinal() ? 'final ' : '',
            $method->isAbstract() ? 'abstract ' : '',
            $method->isPublic() ? 'public ' : '',
            $method->isPrivate() ? 'private ' : '',
            $method->isProtected() ? 'protected ' : '',
            $method->getName());

    $params = $method->getParameters();
    $num_params = count($params);
```

In addition to the methods shown previously for determining access and other method attributes (as well as some others that you can see listed in Figure 2-6 or at http://docs.php.net/en/language.oop5.reflection.html). Each instance of ReflectionMethod has a getParameters() method that returns an array of ReflectionParameter objects. Each of these models a method parameter. In this example script, you list only the names of any parameters using the getName() method; however, this class has several additional methods that are well worth investigating, and we strongly urge you to take a bit of time to do so. You can use this script as a starting point, adding more methods calls and trying it on different classes, and observe the results.

```
      printf("has %s parameter%s%s",
              $num_params == 0 ? 'no' : $num_params,
              $num_params != 1 ? 's' : '',
              $num_params > 0 ? ': ' : '');

      if($num_params > 0)
      {
        $names = array();

        foreach($params as $param)
          $names[] = '$' . $param->getName();
```

```
        print implode(', ', $names);
      }

      print '<br />';
    }
  }
?>
```

Here you can see the output from the preceding script:

```
Name: *Shape*
Defined in file '/home/www/php5/ch2/Shape.class.php', lines 11 - 66

Contains the comments:

/**
  * An example class for class/object functions
  * and Reflection examples - contains a mix of
  * public/private/protected/static members,
  * constants, etc.
  */

Shape is not an interface.
Shape is instantiable.
Shape is not abstract.
Shape is not final.

Shape defines 4 constants:

Array
(
    [NUM_SIDES_TRIANGLE] => 3
    [NUM_SIDES_SQUARE] => 4
    [NUM_SIDES_PENTAGON] => 5
    [NUM_SIDES_HEXAGON] => 6
)

Shape defines 4 properties:

Property [ public static $shapeNames ]


Property [  public $numberOfSides ]


Property [  public $perimeter ]
```

```
Property [  private $name ]


Shape defines 3 methods

public __construct() has 2 parameters: $numberOfSides, $sideLength
protected setName() has 1 parameter: $name
public getName() has no parameters
```

Using the Reflection API, it is possible to find out virtually anything you would ever need to know about an extension, interface, class, object, method, or property—whether you need to get information about a library for which you cannot obtain documentation or to be able to work with dynamic classes and objects in your PHP 5 applications. Although we have barely scratched the surface in this chapter, it is well worth your time to read more about this API and experiment with it.

Summary

If we were asked to name the biggest difference between PHP 4 and PHP 5, we would say without hesitation that it is the introduction of the Zend II language engine. The resulting changes in PHP's handling of classes and objects are little short of revolutionary. PHP 4 was a procedural scripting language with some capacity to work with basic objects and classes. PHP 5 is a different animal: it has the capability for being used as a fully fledged object-oriented language with polymorphism, inheritance, and encapsulation. The fact that it has managed to achieve these objectives while maintaining almost complete backward compatibility with PHP 4–style classes and objects is amazing.

In this chapter, we covered the most important and useful of these new capabilities, starting with an overview of basic object-oriented concepts. You looked at what classes and objects are, their major parts (members), and how to create them. One major change in PHP 5 from its predecessors is that true encapsulation is now supported; that is, you can control access to class members by declaring them to be `public`, `private`, or `protected`. In addition, you can now force subclasses to implement methods (using abstract classes and methods) as well as prevent subclasses from modifying class members (by declaring methods or classes as `final`). PHP 5 also allows for a higher level of abstraction by introducing interfaces; just as a class serves as a template for an object, you can think of an interface as a template for a class—or perhaps it is better to think of a class as implementing one or more interfaces.

Another object-oriented feature making its first appearance in PHP 5 is a new way of handling errors. Exceptions, implemented using an `Exception` class, make it possible to streamline error handling by reducing the number of checks required. They also make it possible to separate error checking from the functional portions of your code. Because the PHP developers wanted to maintain backward compatibility, it is necessary to do a bit of extra preparation to bypass the default error-handling mechanism if you want to use exceptions. However, as you have now seen, doing so is not terribly difficult to accomplish and makes it possible to write much cleaner code than before.

Object-oriented programming is not really complete without a way to obtain information about classes, class instances, and class members, and PHP 4 provided a number of functions to accomplish this. In this chapter, you looked at how PHP 5 retains these functions and adds a few new ones. PHP 5 also introduces a set of classes whose main purpose is to model classes. These classes, known collectively as the Reflection API, make it possible to examine extensions, interfaces, classes, functions, class methods, and properties and their relationships to one another. In addition to introspecting classes, the Reflection API helps facilitate the dynamic generation and manipulation of classes and objects. Both of these capabilities can prove extremely useful when writing generic routines to handle classes and objects whose identity and composition are not known before runtime.

Looking Ahead

In Chapter 3, Frank M. Kromann shows how to perform math calculations in PHP 5. He will cover a number of useful topics in this area, including a survey of the types of numbers supported in PHP 5, ways to identify them, how they are expressed, and techniques enabling the programmer to format numbers in many different ways for output. PHP 5 has a wealth of mathematical functions and operators, and you will get the opportunity to see how to use some of them to help you solve problems you are likely to encounter in your work with PHP 5; these mathematical functions include all common trigonometric functions and functions for working with exponents and logarithms. Chapter 3 will also cover how to generate random numbers for arbitrary ranges of numbers and intervals within those ranges. Finally, you will examine a couple of “bonus” math libraries: BCMath, used for performing calculations requiring a high degree of precision, and GMP, which allows you to work with large integers. Of course, the chapter will provide heaps of examples and useful bits of code that you can easily adapt and build on for your own PHP 5 projects.