CHAPTER 6
Objects

Object-oriented programming (OOP) opens the door to cleaner designs, easier maintenance, and greater code reuse. The proven value of OOP is such that few today would dare to introduce a language that wasn't object-oriented. PHP supports many useful features of OOP, and this chapter shows you how to use them.

OOP acknowledges the fundamental connection between data and the code that works on that data, and it lets you design and implement programs around that connection. For example, a bulletin-board system usually keeps track of many users. In a procedural programming language, each user would be a data structure, and there would probably be a set of functions that work with users' data structures (create the new users, get their information, etc.). In an object-oriented programming language, each user would be an *object*—a data structure with attached code. The data and the code are still there, but they're treated as an inseparable unit.

In this hypothetical bulletin-board design, objects can represent not just users, but also messages and threads. A user object has a username and password for that user, and code to identify all the messages by that author. A message object knows which thread it belongs to and has code to post a new message, reply to an existing message, and display messages. A thread object is a collection of message objects, and it has code to display a thread index. This is only one way of dividing the necessary functionality into objects, though. For instance, in an alternate design, the code to post a new message lives in the user object, not the message object. Designing object-oriented systems is a complex topic, and many books have been written on it. The good news is that however you design your system, you can implement it in PHP.

The object, as union of code and data, is the modular unit for application development and code reuse. This chapter shows you how to define, create, and use objects in PHP. It covers basic OOP concepts as well as advanced topics such as introspection and serialization.

Terminology

Every object-oriented language seems to have a different set of terms for the same old concepts. This section describes the terms that PHP uses, but be warned that in other languages these terms may have other meanings.

Let's return to the example of the users of a bulletin board. You need to keep track of the same information for each user, and the same functions can be called on each user's data structure. When you design the program, you decide the fields for each user and come up with the functions. In OOP terms, you're designing the user *class*. A class is a template for building objects.

An *object* is an instance (or occurrence) of a class. In this case, it's an actual user data structure with attached code. Objects and classes are a bit like values and data types. There's only one integer data type, but there are many possible integers. Similarly, your program defines only one user class but can create many different (or identical) users from it.

The data associated with an object are called its *properties*. The functions associated with an object are called its *methods*. When you define a class, you define the names of its properties and give the code for its methods.

Debugging and maintenance of programs is much easier if you use *encapsulation*. This is the idea that a class provides certain methods (the *interface*) to the code that uses its objects, so the outside code does not directly access the data structures of those objects. Debugging is thus easier because you know where to look for bugs—the only code that changes an object's data structures is within the class—and maintenance is easier because you can swap out implementations of a class without changing the code that uses the class, as long as you maintain the same interface.

Any nontrivial object-oriented design probably involves *inheritance*. This is a way of defining a new class by saying that it's like an existing class, but with certain new or changed properties and methods. The old class is called the *superclass* (or parent or base class), and the new class is called the *subclass* (or derived class). Inheritance is a form of code reuse—the base-class code is reused instead of being copied and pasted into the new class. Any improvements or modifications to the base class are automatically passed on to the derived class.

Creating an Object

It's much easier to create objects and use them than it is to define object classes, so before we discuss how to define classes, let's look at creating objects. To create an object of a given class, use the `new` keyword:

```
$object = new Class;
```

Assuming that a `Person` class has been defined, here's how to create a `Person` object:

```
$rasmus = new Person;
```

Do not quote the class name, or you'll get a compilation error:

```
$rasmus = new "Person"; // does not work
```

Some classes permit you to pass arguments to the `new` call. The class's documentation should say whether it accepts arguments. If it does, you'll create objects like this:

```
$object = new Person("Fred", 35);
```

The class name does not have to be hardcoded into your program. You can supply the class name through a variable:

```
$class = "Person";
$object = new $class;
// is equivalent to
$object = new Person;
```

Specifying a class that doesn't exist causes a runtime error.

Variables containing object references are just normal variables—they can be used in the same ways as other variables. Note that variable variables work with objects, as shown here:

```
$account = new Account;
$object = "account";
${$object}->init(50000, 1.10);  // same as $account->init
```

Accessing Properties and Methods

Once you have an object, you can use the `->` notation to access methods and properties of the object:

```
$object->propertyname $object->methodname([arg, ... ])
```

For example:

```
echo "Rasmus is {$rasmus->age} years old.\n";   // property access
$rasmus->birthday();                            // method call
$rasmus->setAge(21);                            // method call with arguments
```

Methods act the same as functions (only specifically to the object in question), so they can take arguments and return a value:

```
$clan = $rasmus->family("extended");
```

Within a class's definition, you can specify which methods and properties are publicly accessible and which are accessible only from within the class itself using the public and private access modifiers. You can use these to provide encapsulation.

You can use variable variables with property names:

```
$prop = 'age';
echo $rasmus->$prop;
```

A static method is one that is called on a class, not on an object. Such methods cannot access properties. The name of a static method is the class name followed by two colons and the function name. For instance, this calls the `p()` static method in the HTML class:

```
HTML::p("Hello, world");
```

When declaring a class, you define which properties and methods are static using the static access property.

Once created, objects are passed by reference—that is, instead of copying around the entire object itself (a time- and memory-consuming endeavor), a reference to the object is passed around instead. For example:

```
$f = new Person("Fred", 35);

$b = $f; // $b and $f point at same object
$b->setName("Barney");

printf("%s and %s are best friends.\n", $b->getName(), $f->getName());
```

```
Barney and Barney are best friends.
```

If you want to create a true copy of an object, you use the clone operator:

```
$f = new Person("Fred", 35);

$b = clone $f; // make a copy
$b->setName("Barney");// change the copy

printf("%s and %s are best friends.\n", $b->getName(), $f->getName());
```

```
Fred and Barney are best friends.
```

When you use the clone operator to create a copy of an object and that class declares the `__clone()` method, that method is called on the new object immediately after it's cloned. You might use this in cases where an object holds external resources (such as file handles) to create new resources, rather than copying the existing ones.

Declaring a Class

To design your program or code library in an object-oriented fashion, you'll need to define your own classes, using the `class` keyword. A class definition includes the class name and the properties and methods of the class. Class names are case-insensitive and must conform to the rules for PHP identifiers. The class name `stdClass` is reserved. Here's the syntax for a class definition:

```
class classname [ extends baseclass ] [ implements interfacename ,
    [interfacename, ... ] ]
{
  [ use traitname, [ traitname, ... ]; ]

  [ visibility $property [ = value ]; ... ]
```

```
[ function functionname (args) {
    // code
    }
    ...
  ]
}
```

Declaring Methods

A method is a function defined inside a class. Although PHP imposes no special restrictions, most methods act only on data within the object in which the method resides. Method names beginning with two underscores (__) may be used in the future by PHP (and are currently used for the object serialization methods `__sleep()` and `__wakeup()`, described later in this chapter, among others), so it's recommended that you do not begin your method names with this sequence.

Within a method, the `$this` variable contains a reference to the object on which the method was called. For instance, if you call `$rasmus->birthday()`, inside the `birthday()` method, `$this` holds the same value as `$rasmus`. Methods use the `$this` variable to access the properties of the current object and to call other methods on that object.

Here's a simple class definition of the `Person` class that shows the `$this` variable in action:

```
class Person
{
  public $name = '';

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

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

As you can see, the `getName()` and `setName()` methods use `$this` to access and set the `$name` property of the current object.

To declare a method as a static method, use the static keyword. Inside of static methods the variable `$this` is not defined. For example:

```
class HTMLStuff
{
  static function startTable()
  {
    echo "<table border=\"1\">\n";
  }
```

```
  static function endTable()
  {
    echo "</table>\n";
  }
}

HTMLStuff::startTable();
  // print HTML table rows and columns
HTMLStuff::endTable();
```

If you declare a method using the `final` keyword, subclasses cannot override that method. For example:

```
class Person
{
  public $name;

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

class Child extends Person
{
  // syntax error
  function getName()
  {
    // do something
  }
}
```

Using access modifiers, you can change the visibility of methods. Methods that are accessible outside methods on the object should be declared `public`; methods on an instance that can only be called by methods within the same class should be declared `private`. Finally, methods declared as `protected` can only be called from within the object's class methods and the class methods of classes inheriting from the class. Defining the visibility of class methods is optional; if a visibility is not specified, a method is public. For example, you might define:

```
class Person
{
  public $age;

  public function __construct()
  {
    $this->age = 0;
  }

  public function incrementAge()
  {
    $this->age += 1;
    $this->ageChanged();
  }
```

```
  protected function decrementAge()
  {
    $this->age -= 1;
    $this->ageChanged();
  }

  private function ageChanged()
  {
    echo "Age changed to {$this->age}";
  }
}

class SupernaturalPerson extends Person
{
  public function incrementAge()
  {
    // ages in reverse
    $this->decrementAge();
  }
}

$person = new Person;
$person->incrementAge();
$person->decrementAge();   // not allowed
$person->ageChanged();     // also not allowed

$person = new SupernaturalPerson;
$person->incrementAge();   // calls decrementAge under the hood
```

You can use type hinting (see Chapter 3 for more details on type hinting) when declaring a method on an object:

```
class Person
{
  function takeJob(Job $job)
  {
    echo "Now employed as a {$job->title}\n";
  }
}
```

Declaring Properties

In the previous definition of the `Person` class, we explicitly declared the `$name` property. Property declarations are optional and are simply a courtesy to whomever maintains your program. It's good PHP style to declare your properties, but you can add new properties at any time.

Here's a version of the `Person` class that has an undeclared `$name` property:

```
class Person
{
  function getName()
  {
```

```
    return $this->name;
  }

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

You can assign default values to properties, but those default values must be simple constants:

```
public $name = "J Doe";      // works
public $age  = 0;            // works
public $day  = 60 * 60 * 24; // doesn't work
```

Using access modifiers, you can change the visibility of properties. Properties that are accessible outside the object's scope should be declared `public`; properties on an instance that can only be accessed by methods within the same class should be declared `private`. Finally, properties declared as `protected` can only be accessed by the object's class methods and the class methods of classes inheriting from the class. For example, you might declare a user class:

```
class Person
{
  protected $rowId = 0;

  public $username = 'Anyone can see me';

  private $hidden = true;
}
```

In addition to properties on instances of objects, PHP allows you to define static properties, which are variables on an object class, and can be accessed by referencing the property with the class name. For example:

```
class Person
{
  static $global = 23;
}

$localCopy = Person::$global;
```

Inside an instance of the object class, you can also refer to the static property using the `self` keyword, like `echo self::$global;`.

If a property is accessed on an object that doesn't exist, and if the `__get()` or `__set()` method is defined for the object's class, that method is given an opportunity to either retrieve a value or set the value for that property.

For example, you might declare a class that represents data pulled from a database, but you might not want to pull in large data values—such as BLOBs—unless specifically requested. One way to implement that, of course, would be to create access methods

for the property that read and write the data whenever requested. Another method might be to use these overloading methods:

```
class Person
{
  public function __get($property)
  {
    if ($property === 'biography') {
      $biography = "long text here..."; // would retrieve from database

      return $biography;
    }
  }

  public function __set($property, $value)
  {
    if ($property === 'biography') {
      // set the value in the database
    }
  }
}
```

Declaring Constants

Like global constants, assigned through the `define()` function, PHP provides a way to assign constants within a class. Like static properties, constants can be accessed directly through the class or within object methods using the `self` notation. Once a constant is defined, its value cannot be changed:

```
class PaymentMethod
{
  const TYPE_CREDITCARD = 0;
  const TYPE_CASH = 1;
}

echo PaymentMethod::TYPE_CREDITCARD;

0
```

As with global constants, it is common practice to define class constants with uppercase identifiers.

Inheritance

To inherit the properties and methods from another class, use the `extends` keyword in the class definition, followed by the name of the base class:

```
class Person
{
  public $name, $address, $age;
}

class Employee extends Person
```

```
{
  public $position, $salary;
}
```

The `Employee` class contains the `$position` and `$salary` properties, as well as the `$name`, `$address`, and `$age` properties inherited from the `Person` class.

If a derived class has a property or method with the same name as one in its parent class, the property or method in the derived class takes precedence over the property or method in the parent class. Referencing the property returns the value of the property on the child, while referencing the method calls the method on the child.

To access an overridden method on an object's parent class, use the `parent::` *`method`*`()` notation:

```
parent::birthday(); // call parent class's birthday() method
```

A common mistake is to hardcode the name of the parent class into calls to overridden methods:

```
Creature::birthday(); // when Creature is the parent class
```

This is a mistake because it distributes knowledge of the parent class's name throughout the derived class. Using `parent::` centralizes the knowledge of the parent class in the `extends` clause.

If a method might be subclassed and you want to ensure that you're calling it on the current class, use the `self::`*`method`*`()` notation:

```
self::birthday(); // call this class's birthday() method
```

To check if an object is an instance of a particular class or if it implements a particular interface (see the section "Interfaces" on page 156), you can use the `instanceof` operator:

```
if ($object instanceof Animal) {
  // do something
}
```

Interfaces

Interfaces provide a way for defining contracts to which a class adheres; the interface provides method prototypes and constants, and any class that implements the interface must provide implementations for all methods in the interface. Here's the syntax for an interface definition:

```
interface interfacename
{
  [ function functionname();
  ...
  ]
}
```

To declare that a class implements an interface, include the `implements` keyword and any number of interfaces, separated by commas:

```
interface Printable
{
   function printOutput();
}

class ImageComponent implements Printable
{
  function printOutput()
  {
    echo "Printing an image...";
  }
}
```

An interface may inherit from other interfaces (including multiple interfaces) as long as none of the interfaces it inherits from declare methods with the same name as those declared in the child interface.

Traits

Traits provide a mechanism for reusing code outside of a class hierarchy. Traits allow you to share functionality across different classes that don't (and shouldn't) share a common ancestor in a class hierarchy. Here's the syntax for a trait definition:

```
trait traitname [ extends baseclass ]
{
  [ use traitname, [ traitname, ... ]; ]

  [ visibility $property [ = value ]; ... ]

  [ function functionname (args) {
    // code
    }
    ...
  ]
}
```

To declare that a class should include a trait's methods, include the `use` keyword and any number of traits, separated by commas:

```
trait Logger
{
  public function log($logString)
  {
    $className = __CLASS__;
    echo date("Y-m-d h:i:s", time()) . ": [{$className}] {$logString}";
  }
}

class User
{
  use Logger;
```

```
  public $name;

  function __construct($name = '')
  {
    $this->name = $name;
    $this->log("Created user '{$this->name}'");
  }

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

class UserGroup
{
  use Logger;

  public $users = array();

  public function addUser(User $user)
  {
    if (!$this->includesUser($user)) {
      $this->users[] = $user;
      $this->log("Added user '{$user}' to group");
    }
  }
}

$group = new UserGroup;
$group->addUser(new User("Franklin"));
```

```
2012-03-09 07:12:58: [User] Created user 'Franklin'
2012-03-09 07:12:58: [UserGroup] Added user 'Franklin' to group
```

The methods defined by the `Logger` trait are available to instances of the `UserGroup` class as if they were defined in that class.

Traits can be composed of other traits by including the `use` statement in the trait's declaration, followed by one or more trait names separated by commas, as shown here:

```
trait First
{
  public function doFirst(
  {
    echo "first\n";
  }
}

trait Second
{
  public function doSecond()
  {
    echo "second\n";
  }
```

```
}

trait Third
{
  use First, Second;

  public function doAll()
  {
    $this->doFirst();
    $this->doSecond();
  }
}

class Combined
{
  use Third;
}

$object = new Combined;
$object->doAll();
```

```
first
second
```

Traits can declare abstract methods.

If a class uses multiple traits defining the same method, PHP gives a fatal error. However, you can override this behavior by telling the compiler specifically which implementation of a given method you want to use. When defining which traits a class includes, use the `insteadof` keyword for each conflict:

```
trait Command
{
  function run()
  {
    echo "Executing a command\n";
  }
}

trait Marathon
{
  function run()
  {
    echo "Running a marathon\n";
  }
}

class Person
{
  use Command, Marathon {
    Marathon::run insteadof Command;
  }
}

$person = new Person;
```

```
$person->run();
```

```
Running a marathon
```

Instead of picking just one method to include, you can use the `as` keyword to alias a trait's method within a class including it to a different name. You must still explicitly resolve any conflicts in the included traits. For example:

```
trait Command
{
  function run()
  {
    echo "Executing a command";
  }
}

trait Marathon
{
  function run()
  {
    echo "Running a marathon";
  }
}

class Person
{
  use Command, Marathon {
    Command::run as runCommand;
    Marathon::run insteadof Command;
  }
}

$person = new Person;
$person->run();
$person->runCommand();
```

```
Running a marathon
Executing a command
```

Abstract Methods

PHP also provides a mechanism for declaring that certain methods on the class must be implemented by subclasses—the implementation of those methods is not defined in the parent class. In these cases, you provide an abstract method; in addition, if a class has any methods in it defined as abstract, you must also declare the class as an abstract class:

```
abstract class Component
{
  abstract function printOutput();
}

class ImageComponent extends Component
{
```

```
  function printOutput()
  {
    echo "Pretty picture";
  }
}
```

Abstract classes cannot be instantiated. Also note that unlike some languages, you cannot provide a default implementation for abstract methods.

Traits can also declare abstract methods. Classes that include a trait that defines an abstract method must implement that method:

```
trait Sortable
{
  abstract function uniqueId();

  function compareById($object)
  {
    return ($object->uniqueId() < $this->uniqueId()) ? -1 : 1;
  }
}

class Bird
{
  use Sortable;

  function uniqueId()
  {
    return __CLASS__ . ":{$this->id}";
  }
}

class Car
{
  use Sortable;
}

// this will fatal
$bird = new Bird;
$car = new Car;
$comparison = $bird->compareById($card);
```

When implementing an abstract method in a child class, the method signatures must match—that is, they must take in the same number of required parameters, and if any of the parameters have type hints, those type hints must match. In addition, the method must have the same or less-restricted visibility.

Constructors

You may also provide a list of arguments following the class name when instantiating an object:

```
$person = new Person("Fred", 35);
```

These arguments are passed to the class's *constructor*, a special function that initializes the properties of the class.

A constructor is a function in the class called `__construct()`. Here's a constructor for the `Person` class:

```
class Person
{
  function __construct($name, $age)
  {
    $this->name = $name;
    $this->age  = $age;
  }
}
```

PHP does not provide for an automatic chain of constructors; that is, if you instantiate an object of a derived class, only the constructor in the derived class is automatically called. For the constructor of the parent class to be called, the constructor in the derived class must explicitly call the constructor. In this example, the `Employee` class constructor calls the `Person` constructor:

```
class Person
{
  public $name, $address, $age;

  function __construct($name, $address, $age)
  {
    $this->name = $name;
    $this->address = $address;
    $this->age = $age;
  }
}

class Employee extends Person
{
  public $position, $salary;

  function __construct($name, $address, $age, $position, $salary)
  {
    parent::__construct($name, $address, $age);

    $this->position = $position;
    $this->salary = $salary;
  }
}
```

Destructors

When an object is destroyed, such as when the last reference to an object is removed or the end of the script is reached, its *destructor* is called. Because PHP automatically cleans up all resources when they fall out of scope and at the end of a script's execution, their application is limited. The destructor is a method called `__destruct()`:

```
class Building
{
  function __destruct()
  {
    echo "A Building is being destroyed!";
  }
}
```

Introspection

Introspection is the ability of a program to examine an object's characteristics, such as its name, parent class (if any), properties, and methods. With introspection, you can write code that operates on any class or object. You don't need to know which methods or properties are defined when you write your code; instead, you can discover that information at runtime, which makes it possible for you to write generic debuggers, serializers, profilers, etc. In this section, we look at the introspective functions provided by PHP.

Examining Classes

To determine whether a class exists, use the `class_exists()` function, which takes in a string and returns a Boolean value. Alternately, you can use the `get_declared_classes()` function, which returns an array of defined classes and checks if the class name is in the returned array:

```
$doesClassExist = class_exists(classname);

$classes = get_declared_classes();
$doesClassExist = in_array(classname, $classes);
```

You can get the methods and properties that exist in a class (including those that are inherited from superclasses) using the `get_class_methods()` and `get_class_vars()` functions. These functions take a class name and return an array:

```
$methods = get_class_methods(classname);
$properties = get_class_vars(classname);
```

The class name can be a bare word, a quoted string, or a variable containing the class name:

```
$class = "Person";
$methods = get_class_methods($class);
$methods = get_class_methods(Person);     // same
$methods = get_class_methods("Person");   // same
```

The array returned by `get_class_methods()` is a simple list of method names. The associative array returned by `get_class_vars()` maps property names to values and also includes inherited properties.

One quirk of `get_class_vars()` is that it returns only properties that have default values and are visible in the current scope; there's no way to discover uninitialized properties.

Use `get_parent_class()` to find a class's parent class:

```
$superclass = get_parent_class(classname);
```

Example 6-1 lists the `display_classes()` function, which displays all currently declared classes and the methods and properties for each.

Example 6-1. Displaying all declared classes

```
function displayClasses()
{
  $classes = get_declared_classes();

  foreach ($classes as $class) {
    echo "Showing information about {$class}<br />";
    echo "Class methods:<br />";

    $methods = get_class_methods($class);

    if (!count($methods)) {
      echo "<i>None</i><br />";
    }
    else {
      foreach ($methods as $method) {
        echo "<b>{$method}</b>()<br />";
      }
    }

    echo "Class properties:<br />";

    $properties = get_class_vars($class);

    if (!count($properties)) {
      echo "<i>None</i><br />";
    }
    else {
      foreach(array_keys($properties) as $property) {
        echo "<b>\${$property}</b><br />";
      }
    }

    echo "<hr />";
  }
}
```

Examining an Object

To get the class to which an object belongs, first make sure it is an object using the `is_object()` function, and then get the class with the `get_class()` function:

```
$isObject = is_object(var);
$classname = get_class(object);
```

Before calling a method on an object, you can ensure that it exists using the `method_exists()` function:

```
$methodExists = method_exists(object, method);
```

Calling an undefined method triggers a runtime exception.

Just as `get_class_vars()` returns an array of properties for a class, `get_object_vars()` returns an array of properties set in an object:

```
$array = get_object_vars(object);
```

And just as `get_class_vars()` returns only those properties with default values, `get_object_vars()` returns only those properties that are set:

```
class Person
{
  public $name;
  public $age;
}

$fred = new Person;
$fred->name = "Fred";
$props = get_object_vars($fred); // array('name' => "Fred", 'age' => NULL);
```

The `get_parent_class()` function accepts either an object or a class name. It returns the name of the parent class, or `FALSE` if there is no parent class:

```
class A {}
class B extends A {}

$obj = new B;
echo get_parent_class($obj);
echo get_parent_class(B);
```

```
A
A
```

Sample Introspection Program

Example 6-2 shows a collection of functions that display a reference page of information about an object's properties, methods, and inheritance tree.

Example 6-2. Object introspection functions

```
// return an array of callable methods (include inherited methods)
function getCallableMethods($object)
{
  $methods = get_class_methods(get_class($object));

  if (get_parent_class($object)) {
    $parent_methods = get_class_methods(get_parent_class($object));
    $methods = array_diff($methods, $parent_methods);
  }

  return $methods;
}
```

```
// return an array of inherited methods
function getInheritedMethods($object)
{
  $methods = get_class_methods(get_class($object));

  if (get_parent_class($object)) {
    $parentMethods = get_class_methods(get_parent_class($object));
    $methods = array_intersect($methods, $parentMethods);
  }

  return $methods;
}

// return an array of superclasses
function getLineage($object)
{
  if (get_parent_class($object)) {
    $parent = get_parent_class($object);
    $parentObject = new $parent;

    $lineage = getLineage($parentObject);
    $lineage[] = get_class($object);
  }
  else {
    $lineage = array(get_class($object));
  }

  return $lineage;
}

// return an array of subclasses
function getChildClasses($object)
{
  $classes = get_declared_classes();

  $children = array();

  foreach ($classes as $class) {
    if (substr($class, 0, 2) == '__') {
      continue;
    }

    $child = new $class;

    if (get_parent_class($child) == get_class($object)) {
      $children[] = $class;
    }
  }

  return $children;
}

// display information on an object
function printObjectInfo($object)
{
```

```
$class = get_class($object);
echo "<h2>Class</h2>";
echo "<p>{$class}</p>";

echo "<h2>Inheritance</h2>";

echo "<h3>Parents</h3>";
$lineage = getLineage($object);
array_pop($lineage);

if (count($lineage) > 0) {
  echo "<p>" . join(" -&gt; ", $lineage) . "</p>";
}
else {
  echo "<i>None</i>";
}

echo "<h3>Children</h3>";
$children = getChildClasses($object);
echo "<p>";

if (count($children) > 0) {
  echo join(', ', $children);
}
else {
  echo "<i>None</i>";
}

echo "</p>";

echo "<h2>Methods</h2>";
$methods = getCallableMethods($class);
$object_methods = get_methods($object);

if (!count($methods)) {
  echo "<i>None</i><br />";
}
else {
  echo '<p>Inherited methods are in <i>italics</i>.</p>';

  foreach($methods as $method) {
    if (in_array($method, $object_methods)) {
      echo "<b>{$method}</b>();<br />";
    }
    else {
      echo "<i>{$method}</i>();<br />";
    }
  }
}

echo "<h2>Properties</h2>";
$properties = get_class_vars($class);

if (!count($properties)) {
  echo "<i>None</i><br />";
```

```
  }
  else {
    foreach(array_keys($properties) as $property) {
      echo "<b>\${$property}</b> = " . $object->$property . "<br />";
    }
  }

  echo "<hr />";
}
```

Here are some sample classes and objects that exercise the introspection functions from Example 6-2:

```
class A
{
  public $foo = "foo";
  public $bar = "bar";
  public $baz = 17.0;

  function firstFunction()
  {
  }

  function secondFunction()
  {
  }
}

class B extends A
{
  public $quux = false;

  function thirdFunction()
  {
  }
}

class C extends B
{
}

$a = new A;
$a->foo = "sylvie";
$a->bar = 23;

$b = new B;
$b->foo = "bruno";
$b->quux = true;

$c = new C;

printObjectInfo($a);
printObjectInfo($b);
printObjectInfo($c);
```

Serialization

Serializing an object means converting it to a bytestream representation that can be stored in a file. This is useful for persistent data; for example, PHP sessions automatically save and restore objects. Serialization in PHP is mostly automatic—it requires little extra work from you, beyond calling the `serialize()` and `unserialize()` functions:

```
$encoded = serialize(something);
$something = unserialize(encoded);
```

Serialization is most commonly used with PHP's sessions, which handle the serialization for you. All you need to do is tell PHP which variables to keep track of, and they're automatically preserved between visits to pages on your site. However, sessions are not the only use of serialization—if you want to implement your own form of persistent objects, `serialize()` and `unserialize()` are a natural choice.

An object's class must be defined before unserialization can occur. Attempting to unserialize an object whose class is not yet defined puts the object into `stdClass`, which renders it almost useless. One practical consequence of this is that if you use PHP sessions to automatically serialize and unserialize objects, you must include the file containing the object's class definition in every page on your site. For example, your pages might start like this:

```
include "object_definitions.php";   // load object definitions
session_start();                    // load persistent variables
?>
<html>...
```

PHP has two hooks for objects during the serialization and unserialization process: `__sleep()` and `__wakeup()`. These methods are used to notify objects that they're being serialized or unserialized. Objects can be serialized if they do not have these methods; however, they won't be notified about the process.

The `__sleep()` method is called on an object just before serialization; it can perform any cleanup necessary to preserve the object's state, such as closing database connections, writing out unsaved persistent data, and so on. It should return an array containing the names of the data members that need to be written into the bytestream. If you return an empty array, no data is written.

Conversely, the `__wakeup()` method is called on an object immediately after an object is created from a bytestream. The method can take any action it requires, such as reopening database connections and other initialization tasks.

Example 6-3 is an object class, `Log`, that provides two useful methods: `write()` to append a message to the logfile, and `read()` to fetch the current contents of the logfile. It uses `__wakeup()` to reopen the logfile and `__sleep()` to close the logfile.

Example 6-3. The Log.php file

```
class Log
{
  private $filename;
  private $fh;

  function __construct($filename)
  {
    $this->filename = $filename;
    $this->open();
  }

  function open()
  {
    $this->fh = fopen($this->filename, 'a') or die("Can't open {$this->filename}");
  }

  function write($note)
  {
    fwrite($this->fh, "{$note}\n");
  }

  function read()
  {
    return join('', file($this->filename));
  }

  function __wakeup()
  {
    $this->open();
  }

  function __sleep()
  {
    // write information to the account file
    fclose($this->fh);

    return array("filename");
  }
}
```

Store the `Log` class definition in a file called *Log.php*. The HTML page in Example 6-4 uses the `Log` class and PHP sessions to create a persistent log variable, `$logger`.

Example 6-4. front.php

```
<?php
include_once "Log.php";
session_start();
?>

<html><head><title>Front Page</title></head>
<body>
```

```
<?php
$now = strftime("%c");

if (!isset($_SESSION['logger'])) {
  $logger = new Log("/tmp/persistent_log");
  $_SESSION['logger'] = $logger;
  $logger->write("Created $now");

  echo("<p>Created session and persistent log object.</p>");
}

$logger->write("Viewed first page {$now}");

echo "<p>The log contains:</p>";
echo nl2br($logger->read());
?>

<a href="next.php">Move to the next page</a>

</body></html>
```

Example 6-5 shows the file *next.php*, an HTML page. Following the link from the front page to this page triggers the loading of the persistent object `$logger`. The `__wakeup()` call reopens the logfile so the object is ready to be used.

Example 6-5. next.php

```
<?php
include_once "Log.php";
session_start();
?>

<html><head><title>Next Page</title></head>
<body>

<?php
$now = strftime("%c");
$logger->write("Viewed page 2 at {$now}");

echo "<p>The log contains:";
echo nl2br($logger->read());
echo "</p>";
?>

</body></html>
```