Creating Classes

Everything OOP starts with classes. Classes are the *type* of objects, in the same way that "integer" may be the type of a variable. You have to create the type before you can create specific objects of that type. That's important to realize—a class is an object's type. And because you can structure objects as you like, you have to specify their structure when you create classes. After you've created a class, you can then create objects of that class, as you're going to see.

The class is the type of the object, and the object is an instance of the class. You normally create objects using classes, and then use those objects in your code.

Here's an example, a class named Person. In a few pages, we're going to create people objects from the Person class; the class is the specification of what will be inside those objects. Here's how you create a class in PHP, with the class keyword:

```
class Person
{
    .
    .
    .
}
```

Okay, that creates a class named Person. In PHP, classes are usually given names that start with a capital letter, and objects are given names that start with a lowercase letter. As you know, classes can hold data items, called properties, so let's give the person a name, $name. In the class, you're declaring the properties and methods that will go into the objects of this class—and that means you use the var statement to declare properties in PHP. Here's how you add a property named $name to the Person class:

```
class Person
{
    var $name;
        .
        .
        .
}
```

You can also add methods (functions in non-OOP speak) to the Person class. For example, you'll need some way to set the person's name, so you might use a method named set_name to do that. Here's how you add that method to the Person class—note that it takes the person's name as an argument named $data:

```
class Person
{
    var $name;

    function set_name($data)
    {
        .
        .
        .
    }
}
```

Now you've got to store the name passed to the set_name function in the $name variable. You could do that, by accessing $name as a global variable:

```
class Person
{
    var $name;

    function set_name($data)
    {
        global $name;
        .
        .
        .
    }
}
```

Then you can assign the argument passed to set_name, $data, to the internally stored name, $name, like this:

```
class Person
{
    var $name;

    function set_name($data)
    {
        global $name;
        $name = $data;
    }
}
```

That stores the person's name. You'll also need some way to read the person's name, so you might add another method called get_name:

```
class Person
{
    var $name;

    function set_name($data)
    {
        global $name;
        $name = $data;
    }
```

```
    function get_name()
    {
        .
        .
        .
    }
}
```

In the get_name method, you can access the internal name, $name, and return it like this:

```
class Person
{
    var $name;

    function set_name($data)
    {
        global $name;
        $name = $data;
    }

    function get_name()
    {
        global $name;
        return $name;
    }
}
```

That looks good, and it will work, but there's one change to make. You won't normally see OOP done using the global keyword. Instead, you usually refer to the properties of the class using the $this keyword. The $this keyword points to the current object. In PHP terms, this:

```
global $name;
$name = $data;
```

can be replaced by this:

```
$this->name = $data;
```

Note the syntax here—you use $this, followed by the -> operator. In addition, you omit the $ in front of the property you're referring to, like this: $this->name.

Using this syntax, which is the normal syntax you'll see in PHP OOP programming, you can write the methods in the Person class like this:

```
class Person
{
    var $name;

    function set_name($data)
    {
        $this->name = $data;
    }
```

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

Here's one thing to know when talking about creating properties: you can't assign a property a computed value inside a class. For example, this is okay; it assigns the name "Ralph" to $name:

```
class Person
{
    var $name = "Ralph";

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

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

This means that when you create objects of the Person class, $name will be assigned the name "Ralph". However, you can't assign anything but a constant value to a property in a class. For example, you can't even assign the expression "Ralph " . "Kramden", which concatenates "Ralph " and "Kramden", to the $name property. This won't work:

```
class Person
{
    var $name = "Ralph " . "Kramden";  //NO GOOD.

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

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

That's one thing to be careful about when creating properties. Are there similar restrictions on methods? Not really, except that you shouldn't begin the name of a method with a double underscore (__)—there are a number of built-in methods that start that way, and you might conflict with them.

Okay, you've now created a PHP class, the Person class. Now it's time to put it to work, creating some objects.

Creating Objects

To create a new object, you use the new operator in PHP. You just need to define your class, and then you can use the new operator to create an object of that class. Objects are stored as variables, so the object-creation process using the new operator looks like this:

```
class Person
{
  var $name;
  function set_name($data)
  {
      $this->name = $data;
  }

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

$ralph = new Person;
```

Great, that's created a new Person object named $ralph. Easy.

All the methods and properties in the Person class are built into the $ralph object. So how can you call the set_name method, for example, to set the name stored in the object? You can call the $ralph object's set_name method like this, using the -> operator again:

```
$ralph = new Person;
$ralph->set_name("Ralph");
```

That's how it works in PHP; you create an object, and then you can call the methods built into that object using the -> operator. Executing the statement $ralph->set_name("Ralph"); sets the name stored internally in $ralph to "Ralph".

And you can read that name from the object using the object's get_name method. That looks like this in phpobject.php, where you can display the object's internal name:

```
<html>
  <head>
    <title>
      Creating an object
    </title>
  </head>

  <body>
    <h1>Creating an object</h1>
    <?php
      class Person
      {
        var $name;
```

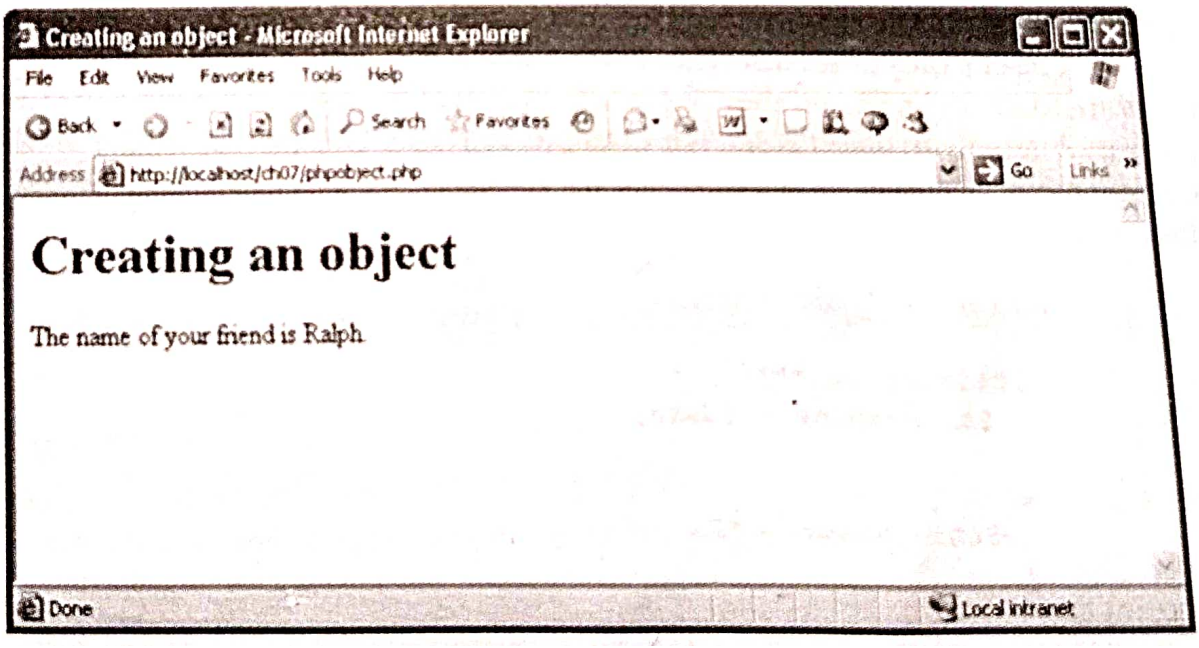
```
        function set_name($data)
        {
            $this->name = $data;
        }

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

    $ralph = new Person;
    $ralph->set_name("Ralph");

    echo "The name of your friend is ", $ralph->get_name(), ".";
  ?>
 </body>
</html>
```

You can see the results in Figure 7-1—this example created a class, then created an object of that class, configured that object by calling its set_name method—and then called get_name to get the stored name. Not bad.

You saw that, after you create an object, you can access the methods of that object like this:

```
$ralph = new Person;
$ralph->set_name("Ralph");
```

By default, you can also access the properties inside an object the same way. For example, you might want to read the name stored internally (as $name) in the $ralph



object, without having to call the get_name method. You can indeed access that property like this:

```
<?php
  class Person
  {
    var $name;
    function set_name($data)
    {
        $this->name = $data;
    }

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

  $ralph = new Person;
  $ralph->set_name("Ralph");

  echo "The name of your friend is ", $ralph->name, ".";
?>
```

So you can recover the name stored in the using the get_name method:

```
$ralph->get_name()
```

or this way, using just the property:

```
$ralph->name
```

In OOP programming, you usually call a method to get data from an object, you don't access the object's properties directly. Methods that return the values of properties are called *accessor* methods in OOP, and they're popular because they give you control over the way the internal data in an object is set. For example, someone may pass an empty string to the set_name method, and in such a case, you may want to set the object's internal name to a default value instead:

```
function set_name($data)
{
    if($data != ""){
      $this->name = $data;
    }
    else {
      $this->name = "Ralph";
    }
}
```

If you just let code outside the object set the value of the $name property directly, you wouldn't have this kind of control over the acceptable values of the $name property.

Okay, you might want to restrict access to an object's internal data and want code to have to call get_name instead of access the $name property directly. But how would you stop code from accessing that property directly? You can do it.

Setting Access to Properties and Methods

At this point, any code has access to the $name property in your object. That's because, by default, all the members (properties and methods) of a class or object are declared public. That means those members are accessible from everywhere in your code. That means that this code works, which accesses the $name property from outside the object:

```
<?php
  class Person
  {
    var $name;

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

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

  $ralph = new Person;
  $ralph->set_name("Ralph");

  echo "The name of your friend is ", $ralph->name, ".";
?>
```

You can restrict access to the members of a class or object with the PHP *access modifiers*, and here they are:

- **public** Means "Accessible to all"
- **private** Means "Accessible in the same class"
- **protected** Means "Accessible in the same class and classes derived from that class"

Public Access

Public access is the most unrestricted access of all, and it's the default. You can explicitly declare properties and methods to be public with the public keyword:

```
<?php
  class Person
  {
    public $name;
```

```
        public function set_name($data)
        {
            $this->name = $data;
        }

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

    $ralph = new Person;
    $ralph->set_name("Ralph");

    echo "The name of your friend is ", $ralph->name, ".";
?>
```

This code works, echoing the name stored internally in the object, because that name has been stored with public access. Now let's restrict access to that name.

Private Access

You can make a class or object member private with the private keyword. When you make a member private, you can't access it outside the class or object. Here's an example, phpprivate.php, where the $name property has been made private to the Person class, and we're trying to access it from outside the $ralph object:

```
<html>
  <head>
    <title>
      Creating an object
    </title>
  </head>

  <body>
    <h1>Creating an object</h1>
    <?php
      class Person
      {
        private $name;

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

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

```
    $ralph = new Person;
    $ralph->set_name("Ralph");

    echo "The name of your friend is ", $ralph->name, ".";
  ?>
 </body>
</html>
```

You can see this example at work in Figure 7-2—as you can see, you get an error because the property you're trying to access is private:

```
The name of your friend is PHP Fatal error: Cannot access private property
Person::$name in C:\Inetpub\wwwroot\ch07\phpprivate.php on line 29
```

That means you have forced code outside the class to access the $name variable.

We'll take a look at the
inheritance, coming up in this chapter.

Using Constructors to Initialize Objects

As you have seen, you can create objects with the new operator, and then use methods like set_name to set the internal data in that object:

```
function set_name($data)
{
  $this->name = $data;
}
```

In this way, you can initialize the data inside an object before you start working with that object.

Wouldn't it be convenient if you could both create and initialize an object at the same time? PHP allows you to do that with *constructors*, which, as in other languages that support OOP, are special methods automatically run when an object is created.

Constructors have a special name in PHP—__construct; that is, "construct" preceded by two underscores. Here's an example:

```
function __construct($data)
{
    .
    .
    .
}
```

As you see, this constructor takes an argument, $data. You can assign that data to the internal name stored in the object like this:

```
function __construct($data)
{
    $this->name = $data;
}
```

we can assign data to internal name stored in obj

How do you use this constructor? You pass data to the constructor when you use the new operator to create new objects, and you pass data to the constructor by enclosing that data in parentheses following the class name. Here's an example, phpconstructor, that initializes an object with the name "Dan":

```
<html>
  <head>
    <title>
      Using a constructor
    </title>
  </head>

  <body>
    <h1>Using a constructor</h1>
    <?php
      class Person
      {
        var $name;

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

        function set_name($data)
        {
            $this->name = $data;
        }
```

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

    $dan = new Person("Dan");

    echo "The name of your friend is ", $dan->get_name(), ".";
?>
</body>
</html>
```

You can see the results in Figure 7-4, where the constructor did indeed initialize the object.

You can pass as many arguments to constructors as you need—as long as the constructor is set up to take those arguments:

```
$dan = new Person("Dan", "brown hair", "blue eyes");
```

All PHP classes come with a default constructor that takes no arguments—it's the default constructor that gets called when you execute code like this:

```
$ralph = new Person;
$ralph->set_name("Ralph");
```

However, as soon as you create your own constructor, no matter how many arguments it takes, the default constructor is no longer accessible.

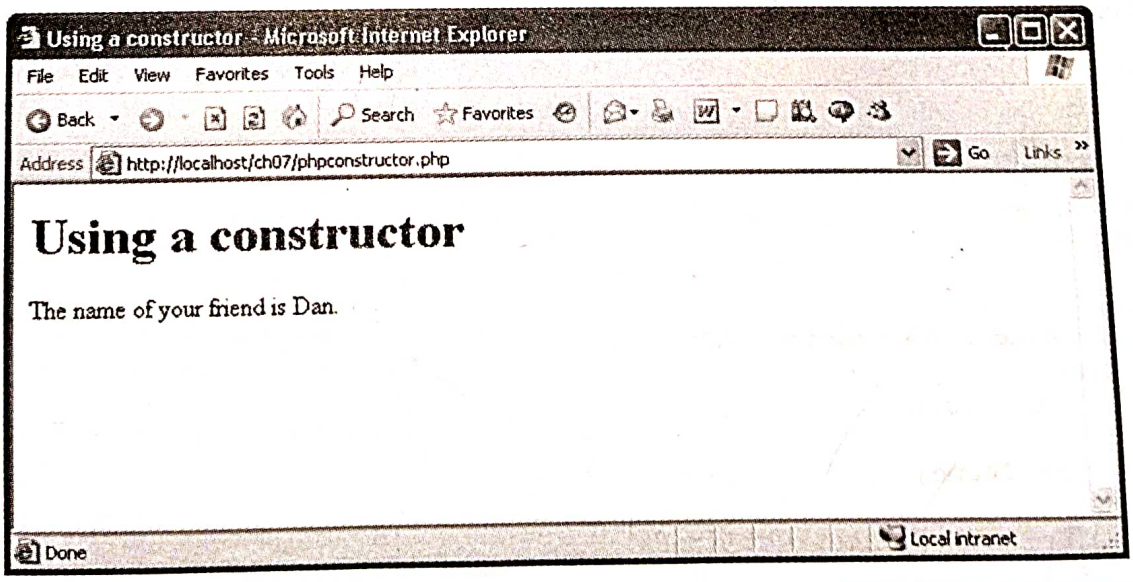


FIGURE 7-4 Using a constructor

Using Destructors to Clean Up after Objects

Besides constructors, PHP also supports *destructors*, which are called when you destroy an object. You use destructors to clean up after an object—terminating database or Internet connections, for example. In PHP, destructors are called when you explicitly destroy an object, or when all references to the object go out of scope.

Destructors are named __destruct in PHP, like this (you don't pass arguments to destructors):

```
function __destruct()
{
}
```

Here's an example, phpdestructor.php:

```
<html>
  <head>
    <title>
      Using a destructor
    </title>
  </head>

  <body>
    <h1>Using a destructor</h1>
    <?php
      class Person
      {
        var $name;

        function __construct($data)
        {
          echo "Constructing ", $data, "...<br>";
```

```
            $this->name = $data;
        }

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

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

        function __destruct()
        {
          echo "Destructing ", $this->name, "...<br>";
        }
      }

      $dan = new Person("Dan");

      echo "The name of your friend is ", $dan->get_name(), ".<br>";
    ?>
  </body>
</html>
```

You can see the results in Figure 7-5, where the destructor was run when the $dan object was destroyed when the script ended.

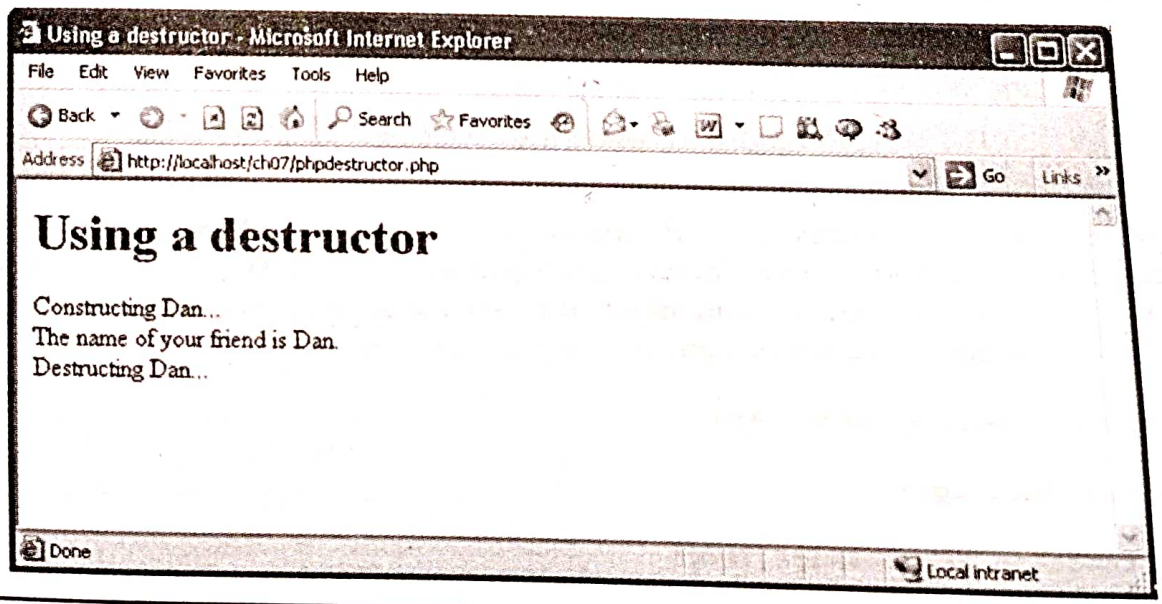


FIGURE 7-5 Using a destructor

Basing One Class on Another with Inheritance

The Person class is fine as far as it goes, but what if you wanted to customize it for your friends so that each friend could have their own saying? For example, Dan might say, "Hi, I'm Dan."; Ralph might say, "Ralph here.", and so on.

You can add functionality to your classes, like the Friend class, through *inheritance*. Inheritance in PHP works much the same as it does in other languages, like Java. Here's how it works; the Person class supports the set_name and get_name methods:

```
class Person
{
  var $name;

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

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

Now say that you want to create a new class, Friend, that has all the functionality of the Person class—that is, the set_name and get_name methods—but also supports some new methods. You can base the Friend class on the Person class with the *extends* keyword this way:

```
class Friend extends Person
{
  .
  .
  .
}
```

Now the Friend class supports the Person methods, set_name and get_name. You can also add your own methods to the Friend class. If you wanted each Friend object to have their own personalized messages, you might add a set_message method that lets you set each object's message, and a speak method that returns that message:

```
class Friend extends Person
{
  var $message;

  function set_message($msg)
  {
    $this->message = $msg;
  }
```

```
    function speak()
    {
      echo $this->message;
    }
  }
```

Now Friend objects will support the set_name, get_name, set_message, and speak methods. Here's an example, phpinheritcance.php, which shows this in action:

```
<html>
  <head>
    <title>
      Inheriting with constructors
    </title>
  </head>

  <body>
    <h1>Inheriting with constructors</h1>
    <?php
      class Person
      {
        var $name;

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

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

      class Friend extends Person
      {
        var $message;

        function set_message($msg)
        {
          $this->message = $msg;
        }

        function speak()
        {
          echo $this->message;
        }
      }

      $tony = new Friend;
      $tony->set_name("Tony");
      $tony->set_message("Hiya from Tony.");
```

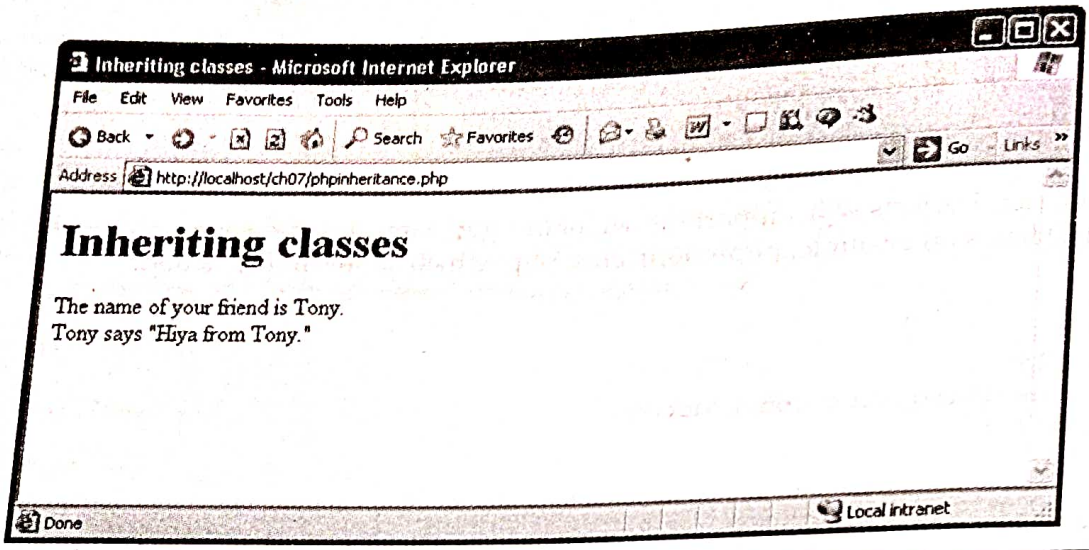


FIGURE 7-6 Inheriting classes

```
        echo "The name of your friend is ", $tony->get_name(), ".<br>";
        echo $tony->get_name(), " says \"", $tony->speak(), "\"<br>";
    ?>
  </body>
</html>
```

You can see the results in Figure 7-6, where the code has used all methods: set_name, get_name, set_message, and speak.

That's how inheritance works—you base one class on another, and the derived class inherits functionality from the base class. Inheriting properties and methods from the base class depends on the access modifiers you've used in the base class. Public members are visible in the derived class (and everywhere else), but members declared private in the base class won't be accessible in the derived class.

What if you want a member accessible in the base class—and in the derived class—but not to code outside the base class and derived classes? For that, you use protected access.

Overriding Methods

In PHP, as with OOP in many languages, you can *override* methods. That means that you can redefine a base class method in a derived class.

Here's an example, phpoverride.php. This example overrides the set_name method in the Person base class:

```
<?php
  class Person
  {
    var $name;

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

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

This method is overridden in the Friend class, simply by redefining it. The overriding version of this method capitalizes the name before storing it:

```
<html>
  <head>
    <title>
      Overriding methods
    </title>
  </head>

  <body>
    <h1>
      Overriding methods
    </h1>
    <?php
      class Person
      {
        var $name;

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

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

      class Friend extends Person
      {
        var $name;

        function speak()
        {
          echo $this->name, " is speaking<br>";
        }

        function set_name($data)
        {
          $this->name = strtoupper($data);
        }
      }

      echo "Creating your new friend...<BR>";
      $friend = new Friend;
      $friend->set_name("Susan");
      $friend->speak();
      ?>
  </body>
</html>
```

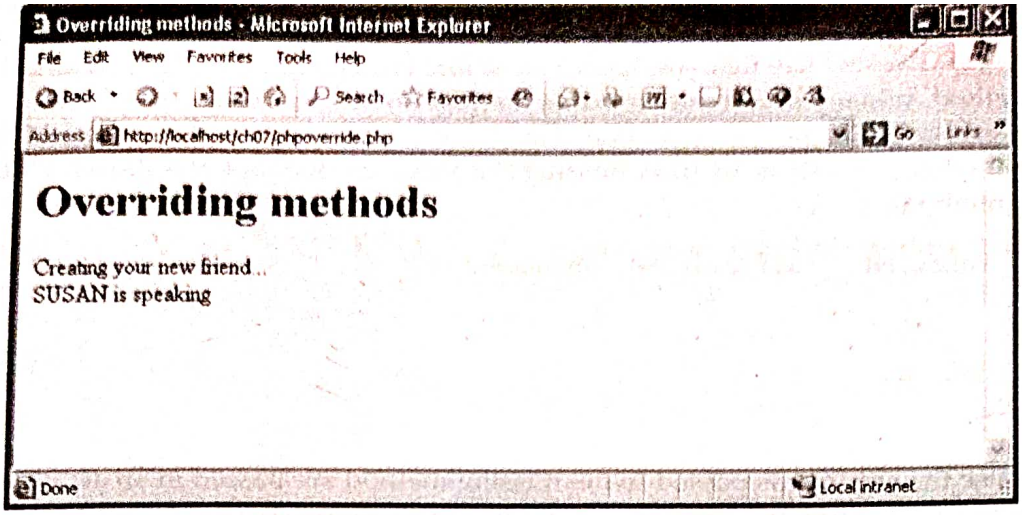


FIGURE 7-9 Overriding methods

That's it—now when you create a $susan object and give the object a name, that name is capitalized by the overridden version of the set_name method. You can see the results in Figure 7-9.

Overloading Methods

Besides overriding methods, you can also *overload* methods in PHP. Overriding a method means redefining it, but overloading it means creating an alternate version with a different argument list. In standard OOP languages, you overload a method by defining a new version of the method with a different argument list. Here's an example, where you define two versions of the set_name method, one that takes the person's name, and another that takes both the person's name and their message:

```
function set_name($data)
{
  $this->name = $data;
}

function set_name($data, $msg)
{
  $this->name = $data;
  $this->message = $msg;
}
```

Now you could call the set_name method with one or two arguments, and the OOP language would call the correct version of set_name, depending on how many arguments were passed:

```
$friend->set_name("Susan");
$friend->set_name("Ted", "Ted here");
```

That's how things work in standard OOP languages—but PHP is different.

Sessions, Cookies, and FTP

There's a lot of PHP power coming up in this chapter—sessions, cookies, FTP, e-mail, and more. These topics are of central interest of PHP programmers, so this chapter is going to be an interesting one.

By its very nature, the Internet is a stateless place—that is, Web pages don't store data. The next time you load most Web pages into a browser is the same as the first—the Web page is reinitialized and is displayed anew. However, the Web has become a more serious place in terms of programming, and that means writing Web applications that work in a multipage way: and that means storing data about the user that persists from page to page. You're going to see two ways of doing that in this chapter: sessions and cookies. And we're going to start with cookies.

Setting a Cookie

Cookies are those text segments that you can store on the user's computer, and PHP has good support for setting and reading cookies. Those segments of text can persist even when the user's computer is turned off, so you can store information about the user easily. That's great when you want to customize a Web application—the user might choose a color scheme, for example. Cookies are used for all kinds of purposes, from the benign to the more sinister—such as tracking what ads a user has already seen and responded to. Setting and reading cookies in PHP is not hard.

You set cookies on the user's machine with the PHP setcookie function:

```
setcookie(name [, value [, expire [, path [, domain [, secure]]]]])
```

Here is what the parameters mean:

- *name* The name of the cookie.
- *value* The value of the cookie. (This value is stored on the clients computer, so do not store sensitive information.)
- *expire* The time the cookie expires. This is the number of seconds since January 1, 1970. You'll most likely set this with the PHP time function plus the number of seconds before you want it to expire.
- *path* The path on the server in which the cookie will be available on.

- *domain* The domain for which the cookie is available.
- *secure* Indicates that the cookie should only be transmitted over a secure HTTPS connection. When set to 1, the cookie will only be set if a secure connection exists. The default is 0.

Want a real simple cookie example? Here it is, phpsetcookie.php. This example sets a cookie named message to the text "No worries.":

```
<?php
  setcookie("message", "No worries.");
?>
```

Here's what this PHP looks like in phpsetcookie.php:

```
<html>
  <head>
    <title>
      Setting a cookie
    </title>
    <?php
      setcookie("message", "No worries.");
    ?>
  </head>

  <body>
    <h1>
      Setting a cookie
    </h1>
    The cookie was set.

    Go to <a href="phpgetcookie.php">phpgetcookie.php</a> to read it.
  <body>
</html>
```

And you can see the results in Figure 11-1, where the cookie was set.

This PHP page includes a hyperlink to another page, phpgetcookie.php, where the cookie will be read:

```
<html>
        .
        .
        .
  <body>
    <h1>
      Setting a cookie
    </h1>
    The cookie was set.

    Go to <a href="phpgetcookie.php">phpgetcookie.php</a> to read it.
  <body>
</html>
```

And that page—phpgetcookie.php—is coming up next.

Here's what this looks like in phpgetcookie.php:

```
<html>
  <head>
    <title>
      Reading a cookie
    </title>
  </head>

  <body>
    <h1>
      Reading a cookie
    </h1>
      The cookie says:
      <?php
        if (isset($_COOKIE['message'])) {
          echo $_COOKIE['message'];
        }
      ?>
  <body>
</html>
```

You can see the results in Figure 11-2, where the cookie was read.

Okay, that lets you set cookies and read them. However, this cookie will be deleted as soon as the user closes their browser. How can you set a cookie's expiration time? That's coming up next.

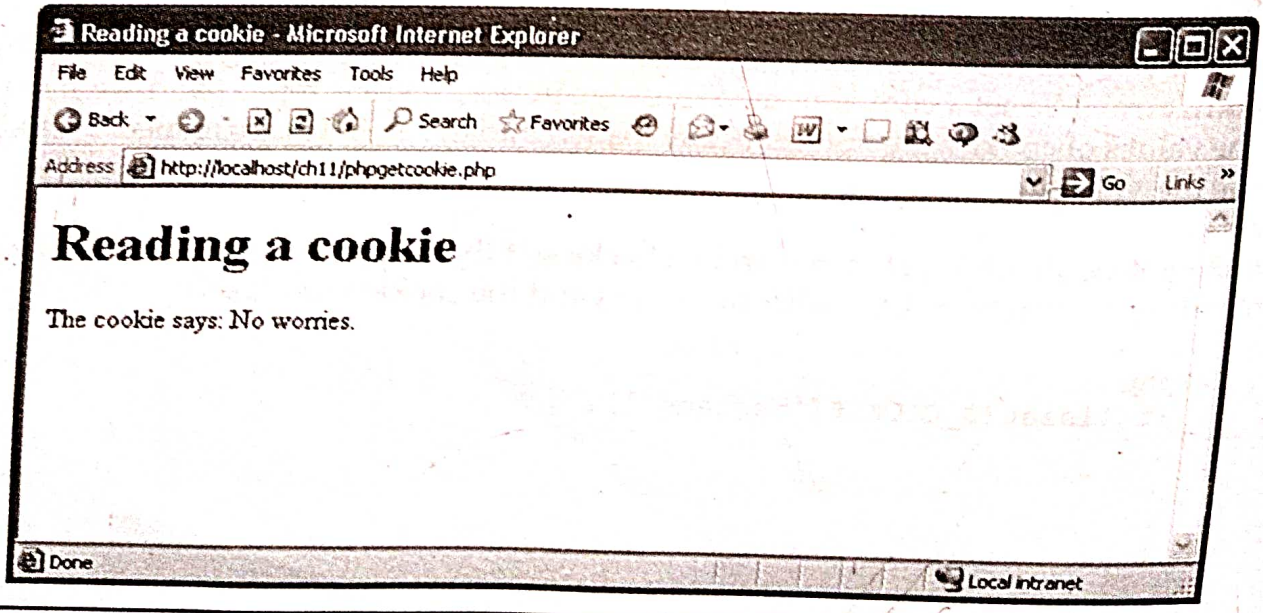


FIGURE 11-2 Reading a cookie

SERIALIZATION : Serializing an object means converting it to a bytestream representation that can b stored in a file. This is useful for persistent data; for example, PHP sessions automatically save and restor objects. Serialization in PHP is mostly automatic

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

Serialization is most commonly used with PHP's sessions, which handle the serialization for you. All yo need to do is tell PHP which variables to keep track of, and they're automatically preserved between vis to pages on your site. However, sessions are not the only use of serialization—if you want to implemer your own form of persistent objects, serialize() and unserialize() are a natural choice. An object's clas must be defined before unserialization can occur. Attempting to unserialize an object whose class is not y defined puts the object into stdClass, which renders it almost useless. One practical consequence of th is that if you use PHP sessions to automatically serialize and unserialize objects, you must include the fi containing the object's class definition in every page on your site. For example, your pages might start lik 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 unserialized. Objects can be serialized if they do not have these methods; however, they won't be notifi about the process.

The __sleep() method is called on an object just before serialization; it can perform any cleanup necessa to preserve the object's state, such as closing database connections, writing out unsaved persistent dat and so on. It should return an array containing the names of the data members that need to be written in 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 bytestream. The method can take any action it requires, such as reopening database connections ar other initialization tasks.

```
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");
}
}
```

WEB TECHNIQUES : PHP was designed as a web-scripting language and, although it is possible to it in purely command-line and GUI scripts, the Web accounts for the vast majority of PHP uses. A dyna website may have forms, sessions, and sometimes redirection, and this chapter explains how to implem those things in PHP. You'll learn how PHP provides access to form parameters and uploaded files, ho send cookies and redirect the browser, how to use PHP sessions, and more.

HTTP BASICS : The Web runs on HTTP, or HyperText Transfer Protocol. This protocol governs how browsers request files from web servers and how the servers send the files back.

To understand the various techniques we'll show you in this chapter, you need to have a basic understan of HTTP. When a web browser requests a web page, it sends an HTTP request message to a web se The request message always includes some header information, and it sometimes also includes a The web server responds with a reply message, which always includes header information and us contains a body. The first line of an HTTP request looks like this:This line specifies an HTTP comm called a method, followed by the address of a document and the version of the HTTP protocol being In this case, the request is using the GET method to ask for the index.html document using HTTP 1.1 this initial line, the request can contain optional header information that gives the server additiona about the request. For example:

```
User-Agent: Mozilla/5.0 (Windows 2000; U) Opera 6.0 [en]
Accept: image/gif, image/jpeg, text/*, */*
```

The User-Agent header provides information about the web browser, while the Accept header specifies the MIME types that the browser accepts. After any headers, the request contains a blank line to indicate the end of the header section. The request can also contain additional data, if that is appropriate for the method being used. If the request doesn't contain any data, it ends with a blank line. The web server receives the request, processes it, and sends a response. The first line of an HTTP response looks like this:

HTTP/1.1 200 OK

This line specifies the protocol version, a status code, and a description of that code.

In this case, the status code is "200", meaning that the request was successful (hence the description "OK"). After the status line, the response contains headers that give the client additional information about the response. For example:

Date: Thu, 31 May 2012 14:07:50 GMT
Server: Apache/2.2.14 (Ubuntu)
Content-Type: text/html
Content-Length: 1845

The Server header provides information about the web server software, while the Content-Type header specifies the MIME type of the data included in the response. After the headers, the response contains a blank line, followed by the requested data if the request was successful. The two most common HTTP methods are GET and POST. The GET method is designed for retrieving information, such as a document, an image, or the results of a database query, from the server. The POST method is meant for posting information, such as a credit card number or information that is to be stored in a database, to the server. The GET method is what a web browser uses when the user types in a URL or clicks on a link. When the user submits a form, either the GET or POST method can be used, as specified by the method attribute of the form tag. We'll discuss the GET and

VARIABLES : Server configuration and request information—including form parameters and cookies—are accessible in three different ways from your PHP scripts, as described in this section.

PHP creates six global arrays that contain the EGPCS information.

The global arrays are:

$_COOKIE:-Contains any cookie values passed as part of the request, where the keys of the array are the names of the cookies

$_GET:-Contains any parameters that are part of a GET request, where the keys of the array are the names of the form parameters

$_POST:-Contains any parameters that are part of a POST request, where the keys of the array are the names of the form parameters

$_FILES:-Contains information about any uploaded files

$_SERVER:-Contains useful information about the web server, as described in the next section

$_ENV:-Contains the values of any environment variables, where the keys of the array are the names of the environment variables

These variables are not only global, but are also visible from within function definitions.

The $_REQUEST array is also created by PHP automatically. The $_REQUEST array contains the elements of the $_GET, $_POST, and $_COOKIE arrays all in one array variable.

SERVER INFORMATION :

The $_SERVER array contains a lot of useful information from the web server. Much of this information comes from the environment variables required in the CGI specification.

Here is a complete list of the entries in $_SERVER that come from CGI: - common Gateway Interface.

PHP_SELF:-The name of the current script, relative to the document root (e.g., /store/cart.php). You should already have noted seeing this used in some of the sample code in earlier chapters. This variable is useful when creating self-referencing scripts, as we'll see later.

SERVER_SOFTWARE:-A string that identifies the server (e.g., "Apache/1.3.33 (Unix) mod_perl/1.26 PHP/5.0.4").

SERVER_NAME:-The hostname, DNS alias, or IP address for self-referencing URLs (e.g., www.example.com).

GATEWAY_INTERFACE:-The version of the CGI standard being followed (e.g., "CGI/1.1"). common gateway Interface

SERVER_PROTOCOL:-The name and revision of the request protocol (e.g., "HTTP/1.1").

SERVER_PORT:-The server port number to which the request was sent (e.g., "80").

REQUEST_METHOD:-The method the client used to fetch the document (e.g., "GET").

PATH_INFO:-Extra path elements given by the client (e.g., /list/users).

PATH_TRANSLATED:-The value of PATH_INFO, translated by the server into a filename (e.g., /home/httpd/htdocs/list/users).

SCRIPT_NAME:-The URL path to the current page, which is useful for self-referencing scripts (e.g., /~me/menu.php).

QUERY_STRING:-Everything after the ? in the URL (e.g., name=Fred+age=35).

REMOTE_HOST:-The hostname of the machine that requested this page (e.g., "dialup-192-168-0-1.example.com (http://dialup-192-168-0-1.example.com)"). If there's no DNS for the machine, this is blank and REMOTE_ADDR is the only information given.

REMOTE_ADDR:-A string containing the IP address of the machine that requested this page (e.g., "192.168.0.250").

AUTH_TYPE:-If the page is password-protected, this is the authentication method used to protect the page (e.g., "basic").

REMOTE_USER:-If the page is password-protected, this is the username with which the client authenticated (e.g., "fred"). Note that there's no way to find out what password was used.

REMOTE_IDENT:-If the server is configured to use identd (RFC 931) identification checks, this is the username fetched from the host that made the web request (e.g., "barney"). Do not use this string for authentication purposes, as it is easily spoofed.

CONTENT_TYPE:-The content type of the information attached to queries such as PUT and POST (e.g., "x-url-encoded").

CONTENT_LENGTH:-The length of the information attached to queries such as PUT and POST (e.g., "3,952").

The Apache server also creates entries in the $_SERVER array for each HTTP header in the request. For each key, the header name is converted to uppercase, hyphens (-) are turned into underscores (_), and the

string "HTTP_" is prepended. For example, the entry for the User-Agent header has the key "HTTP_USER_AGENT". The two most common and useful headers are:

HTTP_USER_AGENT:-The string the browser used to identify itself (e.g., "Mozilla/5.0 (Windows 2000;U) Opera 6.0 [en]")

HTTP_REFERER:-The page the browser said it came from to get to the current page (e.g., http://www.example.com/last_page.html)

PROCESSING FORMS :

It's easy to process forms with PHP, as the form parameters are available in the $_GET and $_POST arrays. There are many tricks and techniques for working with forms, though, which are described in this section.

Methods : As we already discussed, there are two HTTP methods that a client can use to pass form data to the server: GET and POST. The method that a particular form uses is specified with the method attribute to the form tag. In theory, methods are case-insensitive in the HTML, but in practice some broken browsers require the method name to be in all uppercase.

A GET request encodes the form parameters in the URL in what is called a query string; the text that follows the ? is the query string:

/path/to/chunkify.php?word=despicable&length=3

A POST request passes the form parameters in the body of the HTTP request, leaving the URL untouched. The most visible difference between GET and POST is the URL line. Because all of a form's parameters are encoded in the URL with a GET request, users can bookmark GET queries. They cannot do this with POST requests, however.

The biggest difference between GET and POST requests, however, is far subtler. The HTTP specification says that GET requests are idempotent—that is, one GET request for a particular URL, including form parameters, is the same as two or more requests for that URL. Thus, web browsers can cache the response pages for GET requests, because the response page doesn't change regardless of how many times the page is loaded. Because of idempotence, GET requests should be used only for queries such as splitting a word into smaller chunks or multiplying numbers, where the response page is never going to change.

POST requests are not idempotent. This means that they cannot be cached, and the server is re-contacted every time the page is displayed. You've probably seen your web browser prompt you with "Repost form data?" before displaying or reloading certain pages. This makes POST requests the appropriate choice for queries whose response pages may change over time—for example, displaying the contents of a shopping cart or the current messages in a bulletin board.

That said, idempotence is often ignored in the real world. Browser caches are generally so poorly implemented, and the Reload button is so easy to hit, that programmers tend to use GET and POST simply based on whether they want the query parameters shown in the URL or not. What you need to remember is that GET requests should not be used for any actions that cause a change in the server, such as placing an order or updating a database.

The type of method that was used to request a PHP page is available through $_SERVER['REQUEST_METHOD']. For example:

```
if ($_SERVER['REQUEST_METHOD'] == 'GET')
{
// handle a GET request
}
else
```

```
{
die("You may only GET this page.");
}
```

Parameters

Use the $_POST, $_GET, and $_FILES arrays to access form parameters from your PHP code. The key
are the parameter names, and the values are the values of those parameters. Because periods are legal
HTML field names but not in PHP variable names, periods in field names are converted to underscores (
in the array.

Example 7-1 shows an HTML form that chunkifies a string supplied by the user. The form contains t
fields: one for the string (parameter name word) and one for the size of chunks to produce (parame
name number).

to divide string into smaller parts

Example 7-1. The chunkify form (chunkify.html)

```
<html>
<head><title>Chunkify Form</title></head>
<body>
<form action="chunkify.php" method="POST">
Enter a word: <input type="text" name="word" /><br />
How long should the chunks be?
<input type="text" name="number" /><br />
<input type="submit" value="Chunkify!">
</form>
</body>
</html>
```

Example 7-2 lists the PHP script, chunkify.php, to which the form in Example 7-1 submits.

The script copies the parameter values into variables and uses them.

Example 7-2. The chunkify script (chunkify.php)

```
$word = $_POST['word'];
$number = $_POST['number'];
```

Self-Processing Pages

```
<html>
<head><title>Temperature Conversion</title></head>
<body>
<?php if ($_SERVER['REQUEST_METHOD'] == 'GET') { ?>
<form action="<?php echo $_SERVER['PHP_SELF'] ?>" method="POST">
Fahrenheit temperature:
<input type="text" name="fahrenheit" /><br />
<input type="submit" value="Convert to Celsius!" />
</form>
<?php }
else if ($_SERVER['REQUEST_METHOD'] == 'POST') {
$fahrenheit = $_POST['fahrenheit'];
$celsius = ($fahrenheit - 32) * 5 / 9;
printf("%.2fF is %.2fC", $ fahrenheit, $celsius);
}
else {
die("This script only works with GET and POST requests.");
} ?>
```

</body>
</html>

Form Validation : When you allow users to input data, you typically need to validate that data before using it or storing it for later use. There are several strategies available for validating data. The first is JavaScript on the client side. However, since the user can choose to turn JavaScript off, or may even be using a browser that doesn't support it, this cannot be the only validation you do.

A more secure choice is to use PHP to do the validation. Example shows a selfprocessing page with a form. The page allows the user to input a media item; three of the form elements—the name, media type, and filename—are required. If the user neglects to give a value to any of them, the page is presented anew with a message detailing what's wrong. Any form fields the user already filled out are set to the values she entered. Finally, as an additional clue to the user, the text of the submit button changes from "Create" to "Continue" when the user is correcting the form.

```
<?php
$name = $_POST['name'];
$mediaType = $_POST['media_type'];
$filename = $_POST['filename'];
$caption = $_POST['caption'];
$status = $_POST['status'];
```

sends raw http header

SETTING RESPONSE HEADERS : As we've already discussed, the HTTP response that a server sends back to a client contains headers that identify the type of content in the body of the response, the server that sent the response, how many bytes are in the body, when the response was sent, etc. PHP and Apache normally take care of the headers for you, identifying the document as HTML, calculating the length of the HTML page, and so on. Most web applications never need to set headers themselves. However, if you want to send back something that's not HTML, set the expiration time for a page, redirect the client's browser, or generate a specific HTTP error, you'll need to use the header() function.

The only catch to setting headers is that you must do so before any of the body is generated. This means that all calls to header() (or setcookie(), if you're setting cookies) must happen at the very top of your file, even before the <html> tag. For example:

```
<?php header("Content-Type: text/plain"); ?>
Date: today
From: fred
To: barney
Subject: hands off!
My lunchbox is mine and mine alone. Get your own,
you filthy scrounger!
```

Attempting to set headers after the document has started results in this warning:

Warning: Cannot add header information - headers already sent

You can instead use an output buffer; see ob_start(), ob_end_flush(), and related functions for more information on using output buffers.

Different Content Types : tells the type of file

The Content-Type header identifies the type of document being returned. Ordinarily this is "text/html", indicating an HTML document, but there are other useful document types. For example, "text/plain" forces the browser to treat the page as plain text. This type is like an automatic "view source," and it is useful when debugging.

Redirections:- To send the browser to a new URL, known as a redirection, you set the Location header. Generally, you'll also immediately exit afterwards, so the script doesn't bother generating and outputting the remainder of the code listing:

```
header("Location: http://www.example.com/elsewhere.html");
exit();
```

When you provide a partial URL (e.g., **/elsewhere.html**), the web server handles this redirection internally. This is only rarely useful, as the browser generally won't learn that it isn't getting the page it requested. If there are relative URLs in the new document, the browser interprets those URLs as being relative to the requested document, rather than to the document that was ultimately sent. In general, you'll want to redirect to an absolute URL.

Expiration :- A server can explicitly inform the browser, and any proxy caches that might be between the server and browser, of a specific date and time for the document to expire. Proxy and browser caches can hold the document until that time or expire it earlier. Repeated reloads of a cached document do not contact the server. However, an attempt to fetch an expired document does contact the server.

To set the expiration time of a document, use the Expires header:

```
header("Expires: Fri, 18 Jan 2006 05:30:00 GMT");
```

Authentication :- HTTP authentication works through request headers and response statuses. A browser can send a username and password (the credentials) in the request headers. If the credential aren't sent or aren't satisfactory, the server sends a "401 Unauthorized" response and identifies the realm of authentication (a string such as "Mary's Pictures" or "Your Shopping Cart") via the WWW-Authenticate header. This typically pops up an "Enter username and password for . . ." dialog box on the browser, and the page is then re-requested with the updated credentials in the header. To handle authentication in PHP, check the username and password (the PHP_AUTH_USER and PHP_AUTH_PW items of $_SERVER) and call header() to set the realm and send a "401 Unauthorized" response:

```
header("WWW-Authenticate: Basic realm="Top Secret Files"");
header("HTTP/1.0 401 Unauthorized");
```

You can do anything you want to authenticate the username and password; for example, you could consult a database, read a file of valid users, or consult a Microsoft domain server. This example checks to make sure that the password is the username reversed (not the most secure authentication method, to be sure!):

```
$authOK = false;

$user = $_SERVER['PHP_AUTH_USER'];
$password = $_SERVER['PHP_AUTH_PW'];
if (isset($user) && isset($password) && $user === strrev($password)) {
$authOK = true;
}
if (!$authOK) {
header('WWW-Authenticate: Basic realm="Top Secret Files"');
header('HTTP/1.0 401 Unauthorized');
// anything else printed here is only seen if the client hits "Cancel"
exit;
}
```

Maintaining State : HTTP is a stateless protocol, which means that once a web server completes a client's request for a web page, the connection between the two goes away. In other words, there is no way for a server to recognize that a sequence of requests all originate from the same client. State is useful though. You can't build a shopping-cart application, for example, if you can't keep track of a sequence of requests from a single user. You need to know when a user puts an item in his cart, when he adds items, when he removes them, and what's in the cart when he decides to check out. To get around the Web's lac

of state, programmers have come up with many tricks to keep track of state information between requests (also known as session tracking).One such technique is to use hidden form fields to pass around information. PHP treats hidden form fields just like normal form fields, so the values are available in the $_GET and $_POST arrays. Using hidden form fields, you can pass around the entire contents of a shopping cart. However, a more common technique is to assign each user a unique identifier and pass the ID around using a single hidden form field. While hidden form fields work in all browsers, they work only for a sequence of dynamically generated forms, so they aren't as generally useful as some other techniques.

Another technique is URL rewriting, where every local URL on which the user might click is dynamically modified to include extra information. This extra information is often specified as a parameter in the URL. For example, if you assign every user a unique ID, you might include that ID in all URLs, as follows:

http://www.example.com/catalog.php?userid=123

If you make sure to dynamically modify all local links to include a user ID, you can now keep track of individual users in your application. URL rewriting works for all dynamically generated documents, not just forms, but actually performing the rewriting can be tedious.

The third and most widespread technique for maintaining state is to use cookies. A cookie is a bit of information that the server can give to a client. On every subsequent request the client will give that information back to the server, thus identifying itself.

Cookies are useful for retaining information through repeated visits by a browser, but they're not without their own problems. The main problem is that most browsers allow users to disable cookies. So any application that uses cookies for state maintenance needs to use another technique as a fallback mechanism. We'll discuss cookies in more detail shortly.The best way to maintain state with PHP is to use the built-in session-tracking system.This system lets you create persistent variables that are accessible from different pages of your application, as well as in different visits to the site by the same user. Behind the scenes, PHP's session-tracking mechanism uses cookies (or URLs) to elegantly solve most problems that require state, taking care of all the details for you.

...several fields. A server can send one or more

File Handling in PHP

Opening and Closing Files : Files are opened in PHP using the **fopen** command. The command takes two parameters, the file to be opened, and the mode in which to open the file. The function returns a file pointer if successful, otherwise zero (false). Files are opened with **fopen** for reading or writing.

```
$fp = fopen("myfile.txt", "r");
```

If **fopen** is unable to open the file, it returns 0. This can be used to exit the function with an appropriate message.

```
if ( !($fp = fopen("myfile.txt", "r") ) )
exit("Unable to open the input file.");
```

```
<html>
<body>

<?php
$fp=fopen("welcome.txt","r");
?>

</body>
</html>
```

The following example generates a message if the fopen() function is unable to open the specified file:

```
<html>
<body>

<?php
$fp=fopen("welcome.txt","r") or exit("Unable to open file!");
?>

</body>
</html>
```

File Modes : The following table shows the different modes the file may be opened in.

Mode	Description
r	Read Only mode, with the file pointer at the start of the file.
r+	Read/Write mode, with the file pointer at the start of the file.
w	Write Only mode. Truncates the file (effectively overwriting it). If the file doesn't exist, fopen will attempt to create the file.
w+	Read/Write mode. Truncates the file (effectively overwriting it). If the file doesn't exist, fopen will attempt to create the file.
a	Append mode, with the file pointer at the end of the file. If the file doesn't exist, fopen will attempt to create the file.
a+	Read/Append, with the file pointer at the end of the file. If the file doesn't exist, fopen will attempt to create the file.
x	Write only. Creates a new file. Returns FALSE and an error if file already exists
x+	Read/Write. Creates a new file. Returns FALSE and an error if file already exists

Note : The mode may also contain the letter 'b'. This is useful only on systems which differentiate between binary and text files (i.e.Windows. It's useless on Unix/Linux). If not needed, it will be ignored.

Closing a File : The fclose function is used to close a file when you are finished with it.

```
fclose($fp);
<?php
$file = fopen("test.txt","r");

//some code to be executed

fclose($file);
?>
```

Check End-of-file

The feof() function checks if the "end-of-file" (EOF) has been reached.
The feof() function is useful for looping through data of unknown length.

Note: You cannot read from files opened in w, a, and x mode!

```
if (feof($file))
echo "End of file";
```

Reading from Files

You can read from files opened in r, r+, w+, and a+ mode. The **feof** function is used to determine if the end of file is true.

```
if( feof($fp) )
echo "End of file<br>";
```

The feof function can be used in a while loop, to read from the file until the end of file is encountered.

1) A line at a time can be read with the fgets function. Reads one line at a time, up to 254 characters. Each line ends with a newline.If the length argument is omitted, PHP defaults to a length of 1024.

```
<?php
$file = fopen("welcome.txt", "r");

while(!feof($file))
 {
 echo fgets($file). "<br>";
 }
```

```
fclose($file);
?>
```

2) You can read in a single character at a time from a file using the **fgetc** function:

```
<?php
$file=fopen("welcome.txt","r");
while (!feof($file))
 {
 echo fgetc($file);
 }
fclose($file);
?>
```

3) You can also read a single word at a time from a file using the **fscanf function.** The function takes a variable number of parameters, but the first two parameters are mandatory. The first parameter is the file handle, the second parameter is a C-style format string. Any parameters passed after this are optional, but if used will contain the values of the format string.

```
<?php
                                    // See next page for file contents
$fp = fopen("c:/welcome.txt", "r");

while (!feof($fp))
{
                                                    // Assign variables to optional arguments
$buffer = fscanf($fp, "%s %s %d", $name, $title, $age);
if ($buffer == 3)                   // $buffer contains the number of items it was able to assign
print "$name $title $age<br>\n";
}
?>
```

4) You can read in an entire file with the fread function. It reads a number of bytes from a file, up to the end of the file (whichever comes first). The filesize function returns the size of the file in bytes, and can be used with the fread function, as in the following example.

```
<?php
$lst ="c:/welcome.txt";
$fp = fopen($lst, "r");
                                    // See next page for file contents

                                                    // Assign variables to optional arguments
echo fread($fp,filesize($lst));
echo filesize($lst) ;
echo filetype($lst);
?>
```

5) You can also use the file function to read the entire contents of a file into an array instead of opening the file with **fopen:**

```
<?php
$array = file('c:/welcome.txt');
print_r($array);
?>
```

Each array element contains one line from the file, where each line is terminated by a newline.

Writing to Files

The fwrite function is used to write a string, or part of a string to a file. The function takes three parameters, the file handle, the string to write, and the number of bytes to write. If the number of bytes is omitted, the whole string is written to the file. If you want the lines to appear on separate lines in the file, use the \n character. **Note:**Windows requires a carriage return character as well as a new line character, so if you're using Windows, terminate the string with \r\n. The following example logs the visitor to a file, then displays all the entries in the file.

```
<?php
$fp=fopen("ABC.txt","w");
$str="Welcome to CPC";
fwrite($fp,$str);
fclose ($fp);
?>
```

We can write the file in append mode(a or a+).

```
<?php
$fp=fopen("ABC.txt","a+");
$str="Welcome to CPC";
fwrite($fp,$str);
fclose (Sfp);
?>
```

Sessions :

PHP has built-in support for sessions, handling all the cookie manipulation for you to provide persistent variables that are accessible from different pages and across multiple visits to the site. Sessions allow you to easily create multipage forms (such as shopping carts), save user authentication information from page to page, and store persistent user preferences on a site.Each first-time visitor is issued a unique session ID. By default, the session ID is stored in a cookie called PHPSESSID. If the user's browser does not support cookies or has cookies turned off, the session ID is propagated in URLs within the website.

Every session has a data store associated with it. You can register variables to be loaded from the data store when each page starts and saved back to the data store when the page ends. Registered variables persist between pages, and changes to variables made on one page are visible from others. For example, an "add this to your shopping cart" link can take the user to a page that adds an item to a registered array of items in the cart. This registered array can then be used on another page to display the contents of the cart.

Session Basics :

Sessions are started automatically when a script begins running. A new session ID is generated if necessary, possibly creating a cookie to be sent to the browser, and loads any persistent variables from the store.

You can register a variable with the session by passing the name of the variable to the $_SESSION[] array. For example, here is a basic hit counter:

```
session_start();
$_SESSION['hits'] = $_SESSION['hits'] + 1;
echo "This page has been viewed {$_SESSION['hits']} times.";
```

The session_start() function loads registered variables into the associative array $_SESSION. The keys are the variables' names (e.g., $_SESSION[hits']). If you're curious, the session_id(function returns the current session ID.

To end a session, call session_destroy(). This removes the data store for the current session, but it doesn't remove the cookie from the browser cache. This means that, on subsequent visits to sessions-enabled pages, the user will have the same session ID she had before the call to session_destroy().

```
<html>
<head><title>Preferences Set</title></head>
<body>
<?php
session_start();
$colors = array(
'black' => "#000000",
'white' => "#ffffff",
'red' => "#ff0000",
'blue' => "#0000ff"
);
$backgroundName = $_POST['background'];
$foregroundName = $_POST['foreground'];
$_SESSION['backgroundName'] = $backgroundName;
$_SESSION['foregroundName'] = $foregroundName;
?>
```

By default, PHP session ID cookies expire when the browser closes. That is, sessions don't persist after the browser ceases to exist. To change this, you'll need to set the session.cookie_lifetime option in php.ini to the lifetime of the cookie in seconds.

ALTERNATIVES TO COOKIES

By default, the session ID is passed from page to page in the PHPSESSID cookie. However, PHP's session system supports two alternatives: form fields and URLs. Passing the session ID via hidden fields is extremely awkward, as it forces you to make every link between pages to be a form's submit button. We will not discuss this method further here.

The URL system for passing around the session ID, however, is somewhat more elegant.

PHP can rewrite your HTML files, adding the session ID to every relative link. For this to work, though, PHP must be configured with the -enable-trans-id option when compiled. There is a performance penalty for this, as PHP must parse and rewrite every page. Busy sites may wish to stick with cookies, as they do not incur the slowdown caused by page rewriting. In addition, this exposes your session IDs, potentially allowing for man-in-the-middle attacks.

Custom storage

By default, PHP stores session information in files in your server's temporary directory.

Each session's variables are stored in a separate file. Every variable is serialized into the file in a proprietary format. You can change all of these values in the php.ini file.

You can change the location of the session files by setting the session.save_path value in php.ini. If you are on a shared server with your own installation of PHP, set the directory to somewhere in your own directory tree, so other users on the same machine cannot access your session files.

PHP can store session information in one of two formats in the current session store - either PHP's built-in format, or WDDX. You can change the format by setting the session.serialize_handler value in your php.ini file to either php for the default behavior, or wddx for WDDX format.

Combining Cookies and Sessions :

Using a combination of cookies and your own session handler, you can preserve state across visits. Any state that should be forgotten when a user leaves the site, such as which page the user is on, can be left up to PHP's built-in sessions. Any state that should persist between user visits, such as a unique user ID can be stored in a cookie. With Maintaining State | 199

the user's ID, you can retrieve the user's more permanent state, such as display preferences, mailing address, and so on, from a permanent store, such as a database.

Example 7-15 allows the user to select text and background colors and stores those values in a cookie. Any visits to the page within the next week send the color values in the cookie.

Example 7-15. Saving state across visits (save_state.php)

```
<?php
if($_POST['bgcolor']) {
setcookie('bgcolor', $_POST['bgcolor'], time() + (60 * 60 * 24 * 7));
}
if (isset($_COOKIE['bgcolor'])) {
$backgroundName = $_COOKIE['bgcolor'];
}
else if (isset($_POST['bgcolor'])) {
$backgroundName = $_POST['bgcolor'];
}
else {
$backgroundName = "gray";
} ?>
<html>
<head><title>Save It</title></head>
<body bgcolor="<?= $backgroundName; ?>">
<form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="POST">
<p>Background color:
<select name="bgcolor">
<option value="gray">Gray</option>
<option value="white">White</option>
<option value="black">Black</option>
<option value="blue">Blue</option>
<option value="green">Green</option>
<option value="red">Red</option>
</select></p>
<input type="submit" />
</form>
</body>
</html>
```

SSL

The Secure Sockets Layer (SSL) provides a secure channel over which regular HTTP requests and responses can flow. PHP doesn't specifically concern itself with SSL, so you cannot control the encryption in any way from PHP. An https:// URL indicates a secure connection for that document, unlike an http:// URL.

The HTTPS entry in the $_SERVER array is set to 'on' if the PHP page was generated in response to a request over an SSL connection. To prevent a page from being generated over a non-encrypted connection, simply use:

```
if ($_SERVER['HTTPS'] !== 'on')
{
die("Must be a secure connection.");
}
```

A common mistake is to send a form over a secure connection (e.g., https://www.example.com/form.html) but have the action of the form submit to an http:// URL. Any form parameters then entered by the user are sent over an insecure connection—a trivial packet sniffer can reveal them.