

MODULE-4 PHP Arrays and Superglobals

4.1 arrays

Like most other programming languages, PHP supports arrays. In general, an array is a data structure that allows the programmer to collect a number of related elements together in a single variable. Unlike most other programming languages, in PHP an array is actually an **ordered map**, which associates each value in the array with a key. The description of the map data structure is beyond the scope of this chapter, but if you are familiar with other programming languages and their collection classes, a PHP array is not only like other languages' arrays, but it is also like their vector, hash table, dictionary, and list collections. This flexibility allows you to use arrays in PHP in a manner similar to other languages' arrays, but you can also use them like other languages' collection classes.

For some PHP developers, arrays are easy to understand, but for others they are a challenge. To help visualize what is happening, one should become familiar with the concept of keys and associated values. Figure 4.1 illustrates a PHP array with five strings containing day abbreviations.

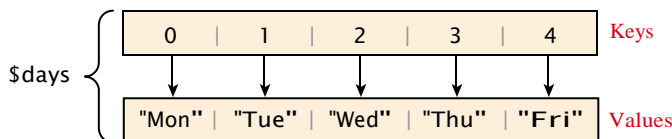


FIGURE 4.1 Visualization of a key-value array

Array keys in most programming languages are limited to integers, start at 0, and go up by 1. In PHP, keys *must* be either integers or strings and need not be sequential. This means you cannot use an array or object as a key (doing so will generate an error).

One should be especially careful about mixing the types of the keys for an array since PHP performs cast operations on the keys that are not integers or strings. You cannot have key "1" distinct from key 1 or 1.5, since all three will be cast to the integer key 1.

Array values, unlike keys, are not restricted to integers and strings. They can be any object, type, or primitive supported in PHP. You can even have objects of your own types, so long as the keys in the array are integers and strings.

4.1.1 Defining and accessing an array

Let us begin by considering the simplest array, which associates each value inside of it with an integer index (starting at 0). The following declares an empty array named days:

```
$days = array();
```

To define the contents of an array as strings for the days of the week as shown in Figure 4.1, you declare it with a comma-delimited list of values inside the () braces using either of two following syntaxes:

```
$days = array("Mon","Tue","Wed","Thu","Fri");
$days = ["Mon","Tue","Wed","Thu","Fri"]; // alternate syntax
```

In these examples, because no keys are explicitly defined for the array, the default key values are 0, 1, 2, . . . , n. Notice that you do not have to provide a size for the array: arrays are dynamically sized as elements are added to them.

Elements within a PHP array are accessed in a manner similar to other programming languages, that is, using the familiar square bracket notation. The code example below echoes the value of our \$days array for the key=1, which results in output of Tue.

```
echo "Value at index 1 is ". $days[1]; // index starts at zero
```

You could also define the array elements individually using this same square bracket notation:

```
$days = array();
$days[0] = "Mon";
$days[1] = "Tue";
$days[2] = "Wed";

// also alternate approach
$daysB = array();
$daysB[] = "Mon";
$daysB[] = "Tue";
$daysB[] = "Wed";
```

In PHP, you are also able to explicitly define the keys in addition to the values. This allows you to use keys other than the classic 0, 1, 2, . . . , n to define the indexes of an array. For clarity, the exact same array defined above and shown in Figure 4.1 can also be defined more explicitly by specifying the keys and values as shown in Figure 4.2.

```

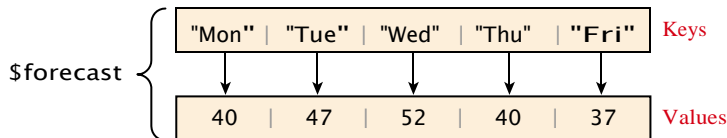
      key
      |
$days = array(0 => "Mon", 1 => "Tue", 2 => "Wed", 3 => "Thu", 4 => "Fri");
                  |
                  value

```

Figure 4.2 Explicitly assigning keys to array elements

```
$forecast = array("Mon" => 40, "Tue" => 47, "Wed" => 52, "Thu" => 40, "Fri" => 37);
```

key
value



```
echo $forecast["Tue"]; // outputs 47
echo $forecast["Thu"]; // outputs 40
```

Figure 4.3 Array with strings as keys and integers as values

Explicit control of the keys and values opens the door to keys that do not start at 0, are not sequential, and that are not even integers (but rather strings). This is why you can also consider an array to be a dictionary or hash map. These types of arrays in PHP are generally referred to as **associative arrays**. You can see in Figure 4.3 an example of an associative array and its visual representation. Keys must be either integer or string values, but the values can be any type of PHP data type, including other arrays. In the example in Figure 4.3, the keys are strings (for the weekdays) and the values are weather forecasts for the specified day in integer degrees.

As can be seen in Figure 4.3, to access an element in an associative array, you simply use the key value rather than an index:

```
echo $forecast["Wed"]; // this will output 52
```

4.1.2 Multidimensional arrays

PHP also supports multidimensional arrays. Recall that the values for an array can be any PHP object, which includes other arrays. Listing 4.1 illustrates the creation of two different multidimensional arrays (each one contains two dimensions).

```
$month = array (
    array("Mon","Tue","Wed","Thu","Fri"),
    array("Mon","Tue","Wed","Thu","Fri"),
    array("Mon","Tue","Wed","Thu","Fri"),
    array("Mon","Tue","Wed","Thu","Fri")
);

echo $month[0][3]; // outputs Thu
```

(continued)

```
$cart = array();
$cart[] = array("id" => 37, "title" => "Burial at Ornans",
               "quantity" => 1);
$cart[] = array("id" => 345, "title" => "The Death of Marat",
               "quantity" => 1);
$cart[] = array("id" => 63, "title" => "Starry Night", "quantity" => 1);

echo $cart[2]["title"]; // outputs Starry Night
```

Listing 4.1 Multidimensional arrays

Figure 4.4 illustrates the structure of these two multidimensional arrays.

4.1.3 iterating through an array

One of the most common programming tasks that you will perform with an array is to iterate through its contents. Listing 4.2 illustrates how to iterate and output the

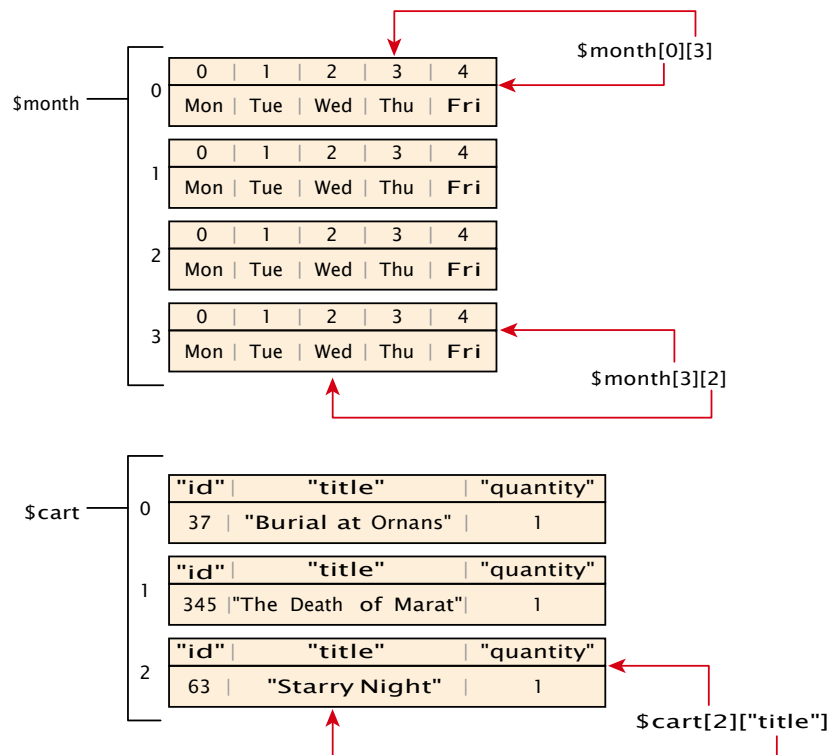


Figure 4.4 Visualizing multidimensional arrays

```
// while loop
$i=0;
while ($i < count($days)) {
    echo $days[$i] . "<br>";
    $i++;
}

// do while loop
$i=0;
do {
    echo $days[$i] . "<br>";
    $i++;
} while ($i < count($days));

// for loop
for ($i=0; $i<count($days); $i++) {
    echo $days[$i] . "<br>";
}
```

Listing 4.2 Iterating through an array using while, do while, and for loops

content of the `$days` array using the built-in function `count()` along with examples using `while`, `do while`, and `for` loops.

The challenge of using the classic loop structures is that when you have non-sequential integer keys (i.e., an associative array), you can't write a simple loop that uses the `$i++` construct. To address the dynamic nature of such arrays, you have to use iterators to move through such an array. This iterator concept has been woven into the `foreach` loop and illustrated for the `$forecast` array in Listing 4.3.

```
// foreach: iterating through the values
foreach ($forecast as $value) {
    echo $value . "<br>";
}

// foreach: iterating through the values AND the keys
foreach ($forecast as $key => $value) {
    echo "day" . $key . "=" . $value;
}
```

Listing 4.3 Iterating through an associative array using a `foreach` loop

4.1.4 adding and Deleting elements

In PHP, arrays are dynamic, that is, they can grow or shrink in size. An element can be added to an array simply by using a key/index that hasn't been used, as shown below:

```
$days[5] = "Sat";
```

Since there is no current value for key 5, the array grows by one, with the new key/value pair added to the end of our array. If the key had a value already, the same style of assignment replaces the value at that key. As an alternative to specifying the index, a new element can be added to the end of any array using the following technique:

```
$days[] = "Sun";
```

The advantage to this approach is that we don't have to worry about skipping an index key. PHP is more than happy to let you "skip" an index, as shown in the following example.

```
$days = array("Mon","Tue","Wed","Thu","Fri");  
$days[7] = "Sat";  
print_r($days);
```

What will be the output of the `print_r()`? It will show that our array now contains the following:

```
Array ([0] => Mon [1] => Tue [2] => Wed [3] => Thu [4] => Fri [7] => Sat)
```

That is, there is now a "gap" in our array that will cause problems if we try iterating through it using the techniques shown in Listing 4.2. If we try referencing `$days[6]`, for instance, it will return a **NULL** value, which is a special PHP value that represents a variable with no value.

You can also create "gaps" by explicitly deleting array elements using the `unset()` function, as shown in Listing 4.4.

```
$days = array("Mon","Tue","Wed","Thu","Fri");

unset($days[2]);
unset($days[3]);

print_r($days); // outputs: Array ( [0] => Mon [1] => Tue [4] => Fri )

$days = array_values($days);
print_r($days); // outputs: Array ( [0] => Mon [1] => Tue [2] => Fri )
```

Listing 4.4 Deleting elements

Listing 4.4 also demonstrates that you can remove “gaps” in arrays (which really are just gaps in the index keys) using the `array_values()` function, which reindexes the array numerically.

Checking if a value exists

Since array keys need not be sequential, and need not be integers, you may run into a scenario where you want to check if a value has been set for a particular key. As with undefined null variables, values for keys that do not exist are also undefined. To check if a value exists for a key, you can therefore use the `isset()` function, which returns true if a value has been set, and false otherwise. Listing 4.5 defines an array with noninteger indexes, and shows the result of asking `isset()` on several indexes.

```
$oddKeys = array(1 => "hello", 3 => "world", 5 => "!");
if (isset($oddKeys[0])) {
    // The code below will never be reached since $oddKeys[0] is not set!
    echo "there is something set for key 0";
}
if (isset($oddKeys[1])) {
    // This code will run since a key/value pair was defined for key 1
    echo "there is something set for key 1, namely ". $oddKeys[1];
}
```

Listing 4.5 Illustrating nonsequential keys and usage of `isset()`

4.1.5 array sorting

One of the major advantages of using a mature language like PHP is its built-in functions. There are many built-in sort functions, which sort by key or by value. To sort the `$days` array by its values you would simply use:

```
sort($days);
```

As the values are all strings, the resulting array would be:

```
Array([0] => Fri [1] => Mon [2] => Sat [3] => Sun [4] => Thu
      [5] => Tue [6] => Wed)
```

However, such a sort loses the association between the values and the keys! A better sort, one that would have kept keys and values associated together, is:

```
asort($days);
```

The resulting array in this case is:

```
Array ([4] => Fri [0] => Mon [5] => Sat [6] => Sun [3] => Thu  
[1] => Tue [2] => Wed)
```

After this last sort, you really see how an array can exist with nonsequential keys! There are even more complex functions available that let you sort by your own comparator, sort by keys, and more. You can read more about sorting functions in the official PHP documentation.¹

4.1.6 More array Operations

In addition to the powerful sort functions, there are other convenient functions you can use on arrays. It does not make sense to reinvent the wheel when valid, efficient functions have already been written for you. While we will not go into detail about each one, here is a brief description of some key array functions:

- **array_keys(\$someArray):** This method returns an indexed array with the values being the *keys* of \$someArray.

For example, `print_r(array_keys($days))` outputs

```
Array ( [0] => 0 [1] => 1 [2] => 2 [3] => 3 [4] => 4 )
```

- **array_values(\$someArray):** Complementing the above `array_keys()` function, this function returns an indexed array with the values being the *values* of \$someArray.

For example, `print_r(array_values($days))` outputs

```
Array ( [0] => Mon [1] => Tue [2] => Wed [3] => Thu [4] => Fri )
```

- **array_rand(\$someArray, \$num=1):** Often in games or widgets you want to select a random element in an array. This function returns as many random keys as are requested. If you only want one, the key itself is returned; otherwise, an array of keys is returned.

For example, `print_r(array_rand($days,2))` might output:

```
Array (3, 0)
```

- **array_reverse(\$someArray):** This method returns \$someArray in reverse order. The passed \$someArray is left untouched.

For example, `print_r(array_reverse($days))` outputs:

```
Array ( [0] => Fri [1] => Thu [2] => Wed [3] => Tue [4] => Mon )
```

- **array_walk(\$someArray, \$callback, \$optionalParam):** This method is extremely powerful. It allows you to call a method (\$callback), for each value in \$someArray. The \$callback function typically takes two parameters, the value first, and the key second. An example that simply prints the value of each element in the array is shown below.

```
$someA = array("hello", "world");
array_walk($someA, "doPrint");
function doPrint($value,$key){
    echo $key . ": " . $value;
}
```

- **in_array(\$needle, \$haystack):** This method lets you search array \$haystack for a value (\$needle). It returns `true` if it is found, and `false` otherwise.
- **shuffle(\$someArray):** This method shuffles \$someArray. Any existing keys are removed and \$someArray is now an indexed array if it wasn't already.

For a complete list, visit the Array class documentation [php.net](https://www.php.net).²

4.1.7 superglobal arrays

PHP uses special predefined associative arrays called **superglobal variables** that allow the programmer to easily access HTTP headers, query string parameters, and other commonly needed information (see Table 4.1). They are called superglobal because

name	Description
\$GLOBALS	Array for storing data that needs superglobal scope
\$_COOKIES	Array of cookie data passed to page via HTTP request
\$_ENV	Array of server environment data
\$_FILES	Array of file items uploaded to the server
\$_GET	Array of query string data passed to the server via the URL
\$_POST	Array of query string data passed to the server via the HTTP header

(continued)

name	Description
<code>\$_REQUEST</code>	Array containing the contents of <code>\$_GET</code> , <code>\$_POST</code> , and <code>\$_COOKIES</code>
<code>\$_SESSION</code>	Array that contains session data
<code>\$_SERVER</code>	Array containing information about the request and the server

table 4.1 Superglobal Variables

these arrays are always in scope and always exist, ready for the programmer to access or modify them without having to use the `global` keyword as in Chapter 8.

The following sections examine the `$_GET`, `$_POST`, `$_SERVER`, and the `$_FILE` superglobals. Chapter 13 on State Management uses `$_COOKIES`, `$_GLOBALS`, and `$_STATE`.

4.2 `$_Get` and `$_pOst` superglobal arrays

The `$_GET` and `$_POST` arrays are the most important superglobal variables in PHP since they allow the programmer to access data sent by the client in a query string. As you will recall from Chapter 4, an HTML form (or an HTML link) allows a client to send data to the server. That data is formatted such that each value is associated with a name defined in the form. If the form was submitted using an HTTP GET request, then the resulting URL will contain the data in the query string. PHP will populate the superglobal `$_GET` array using the contents of this query string in the URL. Figure 4.5 illustrates the relationship between an HTML form, the GET request, and the values in the `$_GET` array.

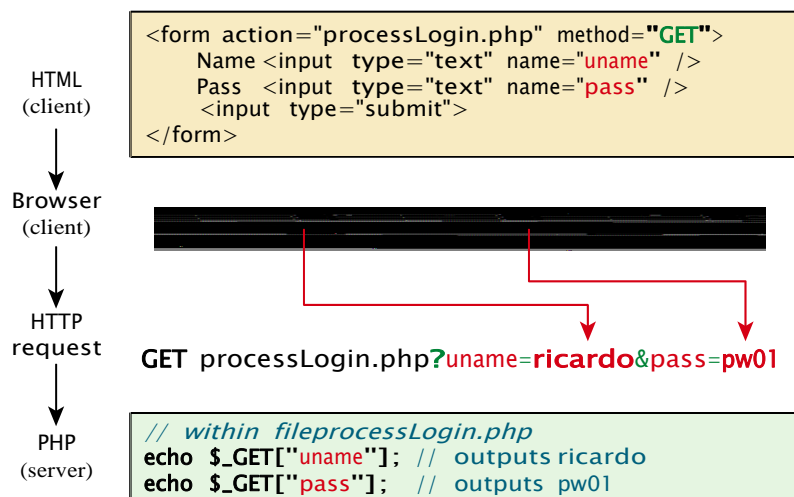


Figure 4.5 Illustration of flow from HTML, to request, to PHP's `$_GET` array

If the form was sent using HTTP POST, then the values would not be visible in the URL, but will be sent through HTTP POST request body. From the PHP programmer's perspective, almost nothing changes from a GET data post except that those values and keys are now stored in the `$_POST` array. This mechanism greatly simplifies accessing the data posted by the user, since you need not parse the query string or the POST request headers. Figure 4.6 illustrates how data from a HTML form using POST populates the `$_POST` array in PHP.

4.2.1 Determining if any Data sent

There will be times as you develop in PHP that you will use the same file to handle both the display of a form as well as the form input. For example, a single file is often used to display a login form to the user, and that same file also handles the processing

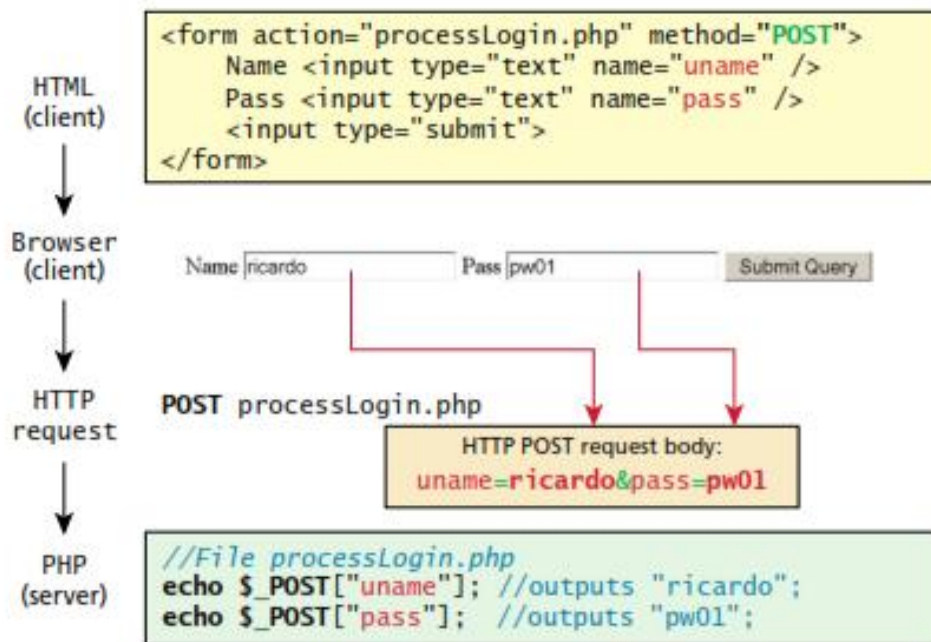


FIGURE 9.6 Data flow from HTML form through HTTP request to PHP's `$_POST` array

of the submitted form data, as shown in Figure 4.8. In such cases you may want to know whether any form data was submitted at all using either POST or GET.

In PHP, there are several techniques to accomplish this task. First, you can determine if you are responding to a POST or GET by checking the `$_SERVER['REQUEST_METHOD']` variable (we will cover the `$_SERVER` superglobal in more detail in Section 4.3). It contains as a string the type of HTTP request this script is responding to (GET, POST, HEAD, etc.). Even though you may know that, for

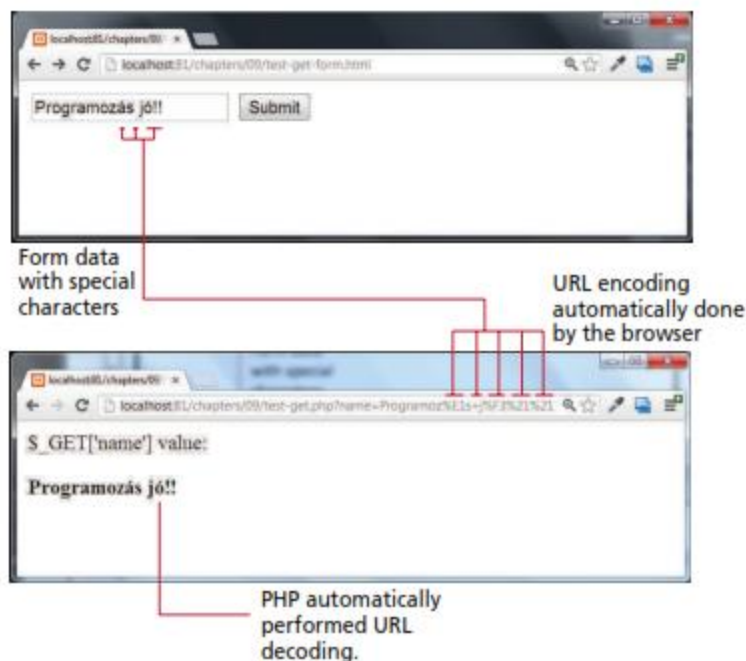


FIGURE 9.7 URL encoding and decoding

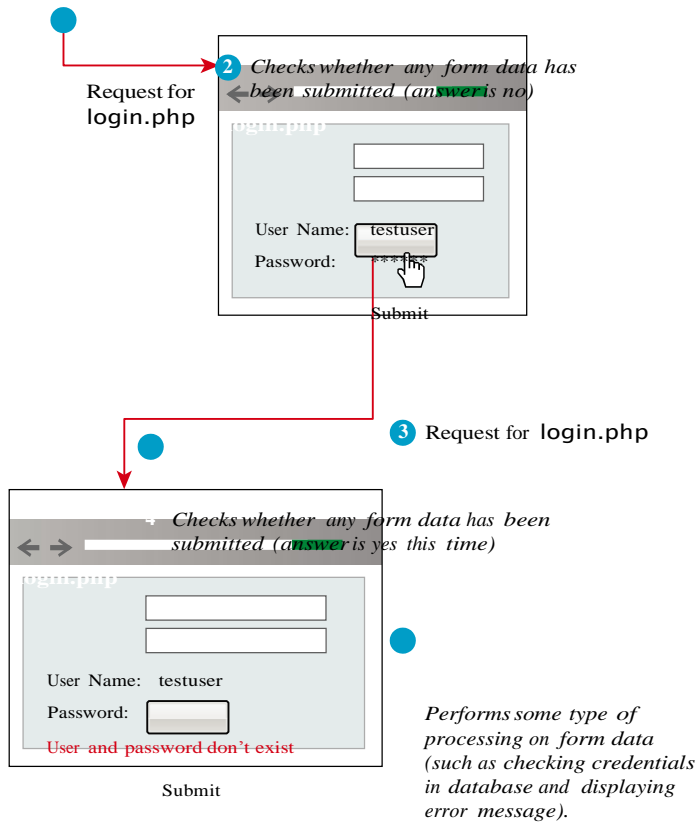


Figure 4.8 Form display and processing by the same PHP page

instance, a POST request was performed, you may want to check if any of the fields are set. To do this you can use the `isset()` function in PHP to see if there is anything set for a particular query string parameter, as shown in Listing 4.6.

```
<!DOCTYPE html>
<html>
<body>
<?php
if ($_SERVER["REQUEST_METHOD"] == "POST") {
    if ( isset($_POST["uname"]) && isset($_POST["pass"]) ) {
        // handle the posted data.
        echo "handling user login now ...";
        echo "... here we could redirect or authenticate ";
        echo " and hide login form or something else";
    }
}
```

(continued)

```
?>
<h1>Some page that has a login form</h1>
<form action="samplePage.php" method="POST">
    Name <input type="text" name="uname"/><br/>
    Pass <input type="password" name="pass"/><br/>
    <input type="submit">
</form>
</body>
</html>
```

Listing 4.6 Using `isset()` to check query string data

4.2.2 accessing Form array Data

Sometimes in HTML forms you might have multiple values associated with a single name; back in Chapter 4, there was an example in Section 4.4.2 on checkboxes. Listing 4.7 provides another example. Notice that each checkbox has the same name value (`name="day"`).

```
<form method="get">
    Please select days of the week you are free.<br />
    Monday <input type="checkbox" name="day" value="Monday" /> <br />
    Tuesday <input type="checkbox" name="day" value="Tuesday" /> <br />
    Wednesday <input type="checkbox" name="day" value="Wednesday" /> <br />
    Thursday <input type="checkbox" name="day" value="Thursday" /> <br />
    Friday <input type="checkbox" name="day" value="Friday" /> <br />
    <input type="submit" value="Submit">
</form>
```

Listing 4.7 HTML that enables multiple values for one name

Unfortunately, if the user selects more than one day and submits the form, the `$_GET['day']` value in the superglobal array *will only contain the last value from the list* that was selected.

To overcome this limitation, you must change the HTML in the form. In particular, you will have to change the name attribute for each checkbox from `day` to `day[]`.

```
Monday <input type="checkbox" name="day[]" value="Monday" />
Tuesday <input type="checkbox" name="day[]" value="Tuesday" />
...
```

After making this change in the HTML, the corresponding variable `$_GET['day']` will now have a value that is of type array. Knowing how to use arrays, you can process the output as shown in Listing 4.8 to echo the number of days selected and their values.

```
<?php

echo "You submitted " . count($_GET['day']) . " values";
foreach ($_GET['day'] as $d) {
    echo $d . ", ";
}

?>
```

Listing 4.8 PHP code to display an array of checkbox variables

4.2.3 Using Query strings in hyperlinks

As mentioned several times now, form information packaged in a query string is transported to the server in one of two locations depending on whether the form method is GET or POST. It is important to also realize that making use of query strings is not limited to only data entry forms.

You may wonder if it is possible to combine query strings with anchor tags . . . the answer is YES! Anchor tags (i.e., hyperlinks) also use the HTTP GET method. Indeed it is extraordinarily common in web development to programmatically construct the URLs for a series of links from, for instance, database data. Imagine a web page in which we are displaying a list of book links. One approach would be to have a separate page for each book (as shown in Figure 4.4). This is not a very sensible approach. Our database may have hundreds or thousands of books in it: surely it would be too much work to create a separate page for each book!

It would make a lot more sense to have a single Display Book page that receives as input a query string that specifies which book to display, as shown in Figure 4.10. Notice that we typically pass some type of unique identifier in the query string (in this case, using the book's ISBN).



FIGURE 9.9 Inefficient approach to displaying individual items

4.2.4 sanitizing Query strings

One of the most important things to remember about web development is that you should actively distrust all user input. That is, just because you are expecting a proper query string, it doesn't mean that you are going to get a properly constructed query string. What will happen if the user edits the value of the query string parameter? Depending on whether the user removes the parameter or changes its type, either an empty screen or even an error page will be displayed. More worrisome is the threat of SQL injection, where the user actively tries to gain access to the underlying database server (we will examine SQL injection attacks in detail in Chapter 16).

Clearly this is an unacceptable result! At the very least, your program must be able to handle the following cases for *every* query string or form value (and, after we learn about them in Chapter 13, every cookie value as well):

- If query string parameter doesn't exist.
- If query string parameter doesn't contain a value.
- If query string parameter value isn't the correct type.
- If value is required for a database lookup, but provided value doesn't exist in the database table.

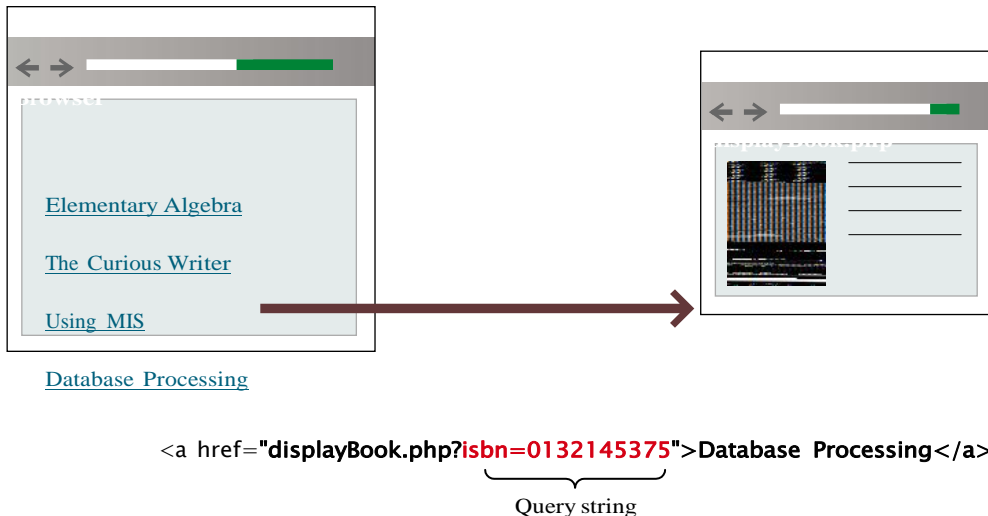


Figure 4.10 Sensible approach to displaying individual items using query strings

The process of checking user input for incorrect or missing information is sometimes referred to as the process of **sanitizing user inputs**. How can we do these types of validation checks? It will require programming similar to that shown in Listing 4.4.

```
// This uses a database API . . . we will learn about it in Chapter 11
$pid = mysqli_real_escape_string($link, $_GET['id']);

if ( is_int($pid) ) {
    // Continue processing as normal
}
else {
    // Error detected. Possibly a malicious user
}
```

Listing 4.4 Simple sanitization of query string values

What should we do when an error occurs in Listing 4.4? There are a variety of possibilities; Chapter 12 will examine the issue of exception and error handling in more detail. For now, we might simply redirect to a generic error handling page using the header directive, for instance:

```
header("Location: error.php"); exit();
```

4.3 `$_SERVER` array

The `$_SERVER` associative array contains a variety of information. It contains some of the information contained within HTTP request headers sent by the client. It also contains many configuration options for PHP itself, as shown in Figure 4.11.

To use the `$_SERVER` array, you simply refer to the relevant case-sensitive key name:

```
echo $_SERVER["SERVER_NAME"] . "<br/>";  
echo $_SERVER["SERVER_SOFTWARE"] . "<br/>";  
echo $_SERVER["REMOTE_ADDR"] . "<br/>";
```

It is worth noting that because the entries in this array are created by the web server, not every key listed in the PHP documentation will necessarily be available. A complete list of keys contained within this array is listed in the online PHP documentation, but we will cover some of the critical ones here. They can be classified into keys containing request header information and keys with information about the server settings (which is often configured in the `php.ini` file).

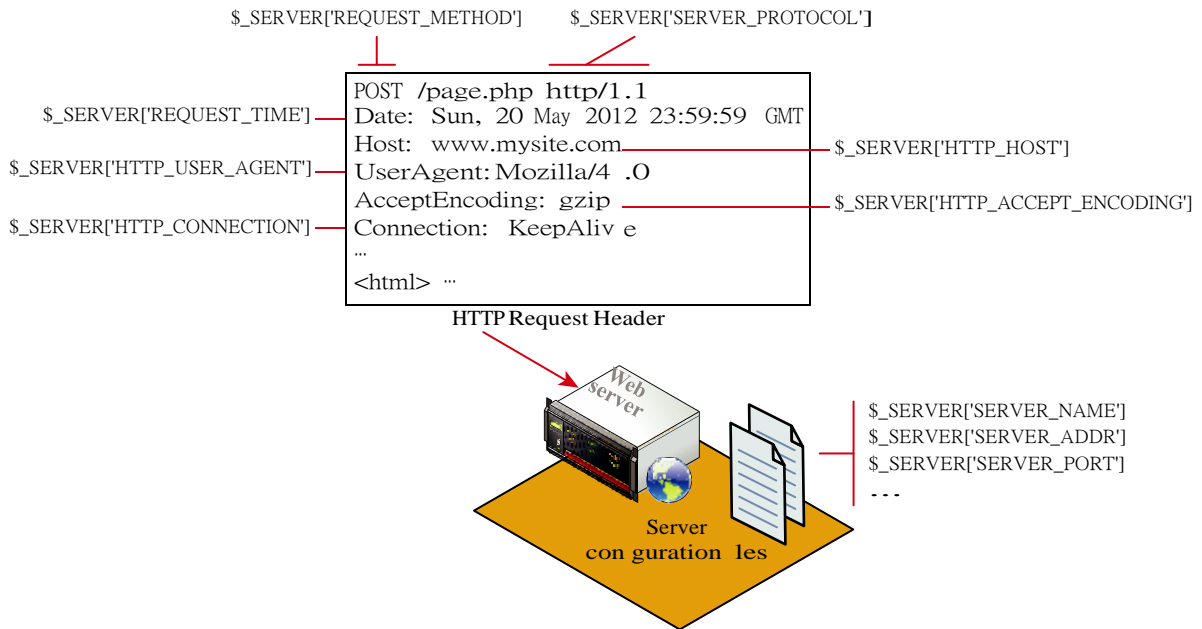


Figure 4.11 Relationship between request headers, the server, and the `$_SERVER` array

4.3.1 server information Keys

`SERVER_NAME` is a key in the `$_SERVER` array that contains the name of the site that was requested. If you are running multiple hosts on the same code base, this can be a useful piece of information. `SERVER_ADDR` is a complementary key telling us the IP of the server. Either of these keys can be used in a conditional to output extra HTML to identify a development server, for example.

`DOCUMENT_ROOT` tells us the file location from which you are currently running your script. Since you are often moving code from development to production, this key can be used to great effect to create scripts that do not rely on a particular location to run correctly. This key complements the `SCRIPT_NAME` key that identifies the actual script being executed.

4.3.2 request header information Keys

Recall that the web server responds to HTTP requests, and that each request contains a request header. These keys provide programmatic access to the data in the request header.

The `REQUEST_METHOD` key returns the request method that was used to access the page: that is, GET, HEAD, POST, PUT.

The `REMOTE_ADDR` key returns the IP address of the requestor, which can be a useful value to use in your web applications. In real-world sites these IP addresses are often stored to provide an audit trail of which IP made which requests, especially on sensitive matters like finance and personal information. In an online poll, for example, you might limit each IP address to a single vote. Although these can be forged, the technical competence required is high, thus in practice one can usually assume that this field is accurate.

One of the most commonly used request headers is the **user-agent** header, which contains the operating system and browser that the client is using. This header value can be accessed using the key `HTTP_USER_AGENT`. The user-agent string as posted in the header is cryptic, containing information that is semicolon-delimited and may be hard to decipher. PHP has included a comprehensive (but slow) method to help you debug these headers into useful information. Listing 4.10 illustrates a script that accesses and echoes the user-agent header information.

```
<?php
echo $_SERVER['HTTP_USER_AGENT'];

$browser = get_browser($_SERVER['HTTP_USER_AGENT'], true);
print_r($browser);
?>
```

Listing 4.10 Accessing the user-agent string in the HTTP headers

One can use user-agent information to redirect to an alternative site, or to include a particular style sheet. User-agent strings are also almost always used for analytic purposes to allow us to track which types of users are visiting our site, but this technique is captured in later chapters.

`HTTP_REFERER` is an especially useful header. Its value contains the address of the page that referred us to this one (if any) through a link. Like `HTTP_USER_AGENT`, it is commonly used in analytics to determine which pages are linking to our site.

Listing 4.11 shows an example of context-dependent output that outputs a message to clients that came to this page from the search page, a message that is not shown to clients that came from any other link. This allows us to output a link back to the search page, but only when the user arrived from the search page.

```
$previousPage = $_SERVER['HTTP_REFERER'];  
// Check to see if referer was our search page  
if (strpos("search.php",$previousPage) != 0) {  
    echo "<a href='search.php'>Back to search</a>";  
}  
// Rest of HTML output
```

Listing 4.11 Using the HTTP_REFERER header to provide context-dependent output

4.4 \$_FILES array

The \$_FILES associative array contains items that have been uploaded to the current script. Recall from Chapter 4 that the <input type="file"> element is used to create the user interface for uploading a file from the client to the server. The user interface is only one part of the uploading process. A server script must process the upload file(s) in some way; the \$_FILES array helps in this process.

4.4.1 HTML required for File Uploads

To allow users to upload files, there are some specific things you must do:

- First, you must ensure that the HTML form uses the HTTP POST method, since transmitting a file through the URL is not possible.
- Second, you must add the enctype="multipart/form-data" attribute to the HTML form that is performing the upload so that the HTTP request can

submit multiple pieces of data (namely, the HTTP post body, and the HTTP file attachment itself).

- Finally you must include an input type of **file** in your form. This will show up with a browse button beside it so the user can select a file from their computer to be uploaded. A simple form demonstrating a very straightforward file upload to the server is shown in Listing 4.12.

```
<form enctype='multipart/form-data' method='post'>
  <input type='file' name='file1' id='file1' />
  <input type='submit' />
</form>
```

Listing 4.12 HTML for a form that allows an upload

4.4.2 handling the File Upload in php

The corresponding PHP file responsible for handling the upload (as specified in the HTML form's action attribute) will utilize the superglobal `$_FILES` array.⁴ This array will contain a key=value pair for each file uploaded in the post. The key for each element will be the name attribute from the HTML form, while the value will be an array containing information about the file as well as the file itself. The keys in that array are the name, type, tmp_name, error, and size.

Figure 4.12 illustrates the process of uploading a file to the server and how the corresponding upload information is contained in the `$_FILES` array. The values for each of the keys, in general, are described below.

- **name** is a string containing the full file name used on the client machine, including any file extension. It does not include the file path on the client's machine.
- **type** defines the MIME type of the file. This value is provided by the client browser and is therefore not a reliable field.
- **tmp_name** is the full path to the location on your server where the file is being temporarily stored. The file will cease to exist upon termination of the script, so it should be copied to another location if storage is required.
- **error** is an integer that encodes many possible errors and is set to `UPLOAD_ERR_OK` (integer value 0) if the file was uploaded successfully.
- **size** is an integer representing the size in bytes of the uploaded file.

Just having the data in a temporary file, and the reference to it in `$_FILES` is not enough. You must also write a script to handle the uploaded files. If you want to store the file, you will have to move it to a location on the server to which Apache has write access. You must also decide what to name the file, and whether to make it accessible to the world. Alternatively, you might decide to save the



FIGURE 9.12 Data flow from HTML form through POST to PHP `$_FILES` array

uploaded information within a database (you will learn how to do this at the end of the next chapter). Regardless of which approach you take, before “saving” the file, you should also perform a variety of checks. This might include looking for transmission errors, setting file size limits and type restrictions, or handling previous uploads.

4.4.3 Checking for errors

For every uploaded file, there is an error value associated with it in the `$_FILES` array. The error values are specified using constant values, which resolve to integers. The value for a successful upload is `UPLOAD_ERR_OK`, and should be looked for before proceeding any further. The full list of errors is provided in Table 4.2 and shows that there are many causes for bad file uploads.

A proper file upload script will therefore check each uploaded file by checking the various error codes as shown in Listing 4.13.

4.4.4 File size restrictions

Some scripts limit the file size of each upload. There are many reasons to do so, and ideally you would prevent the file from even being transmitted in the first place if it is too large. There are three main mechanisms for maintaining uploaded file size restrictions: via HTML in the input form, via JavaScript in the input form, and via PHP coding.

error Code	Integer	Meaning
upload_err_ok	0	Upload was successful.
upload_err_ini_size	1	The uploaded file exceeds the upload_max_filesize directive in php.ini .
upload_err_Form_size	2	The uploaded file exceeds the max_file_size directive that was specified in the HTML form.
upload_err_partial	3	The file was only partially uploaded.
upload_err_no_File	4	No file was uploaded. Not always an error, since the user may have simply not chosen a file for this field.
upload_err_no_tMp_Dlr	6	Missing the temporary folder.
upload_err_Cant_Write	7	Failed to write to disk.
upload_err_eXtension	8	A PHP extension stopped the upload.

table 4.2 Error Codes in PHP for File Upload Taken from php.net.⁶

```
foreach ($_FILES as $fileKey => $fileArray) {
    if ($fileArray["error"] != UPLOAD_ERR_OK) { // error
        echo "Error: " . $fileKey . " has error" . $fileArray["error"]
            . "<br>";
    }
    else { // no error
        echo $fileKey . "Uploaded successfully ";
    }
}
```

Listing 4.13 Checking each file uploaded for errors

The first of these mechanisms is to add a hidden input field before any other input fields in your HTML form with a name of `MAX_FILE_SIZE`. This technique allows your `php.ini` maximum file size to be large, while letting some forms override that large limit with a smaller one. Listing 4.14 shows how the HTML from Listing 4.12 must be modified to add such a check. It should be noted that though this mechanism is set up in the HTML form, it is only available to use when your server-side environment is using PHP.

```
<form enctype='multipart/form-data' method='post'>
  <input type="hidden" name="MAX_FILE_SIZE" value="1000000" />
  <input type='file' name='file1' />
  <input type='submit' />
</form>
```

Listing 4.14 Limiting upload file size via HTML

As intuitive as it is, this hidden field can easily be overridden by the client, and is therefore unacceptable as the only means of limiting size. Moreover, since it is a server-side check and not a client-side one, this means that the file uploading must be complete before an error message can be received. This could be quite frustrating for the user to wait for a large upload to finish only to get an error that the uploaded file was too large!

The more complete client-side mechanism to prevent a file from uploading if it is too big is to prevalidate the form using JavaScript. Such a script, to be added to a handler for the form, is shown in Listing 4.15.

```
<script>
var file = document.getElementById('file1');
var max_size = document.getElementById("max_file_size").value;
if (file.files && file.files.length == 1){
    if (file.files[0].size > max_size) {
        alert("The file must be less than " + (max_size/1024) + "KB");
        e.preventDefault();
    }
}
</script>
```

Listing 4.15 Limiting upload file size via JavaScript

The third (and essential) mechanism for limiting the uploaded file size is to add a simple check on the server side (just in case JavaScript was turned off or the user modified the MAX_FILE_SIZE hidden field). This technique checks the file size on the server by simply checking the size field in the \$_FILES array. Listing 4.16 shows an example of such a check.

```
$max_file_size = 10000000;
foreach($_FILES as $fileKey => $fileArray) {
    if ($fileArray["size"] > $max_file_size) {
        echo "Error: " . $fileKey . " is too big";
    }
    printf("%s is %.2f KB", $fileKey, $fileArray["size"]/1024);
}
```

Listing 4.16 Limiting upload file size via PHP

4.4.5 Limiting the type of File Upload

Even if the upload was successful and the size was within the appropriate limits, you may still have a problem. What if you wanted the user to upload an image and they uploaded a Microsoft Word document? You might also want to limit the uploaded image to certain image types, such as jpg and png, while disallowing bmp and others. To accomplish this type of checking you typically examine the file extension and the type field. Listing 4.17 shows sample code to check the file extension of a file, and also to compare the type to valid image types.

```
$validExt = array("jpg", "png");
$validMime = array("image/jpeg","image/png");
foreach($_FILES as $fileKey => $fileArray ){
    $extension = end(explode(".", $fileArray["name"]));
    if (in_array($fileArray["type"],$validMime) &&
        in_array($extension, $validExt)) {
        echo "all is well. Extension and mime types valid";
    }
    else {
        echo $fileKey." Has an invalid mime type or extension";
    }
}
```

Listing 4.17 PHP code to look for valid mime types and file extensions

4.4.6 Moving the File

With all of our checking completed, you may now finally want to move the temporary file to a permanent location on your server. Typically, you make use of the PHP function `move_uploaded_file()`, which takes in the temporary file location and the file's final destination. This function will only work if the source file exists and if the destination location is writable by the web server (Apache). If there is a problem the function will return false, and a warning may be output. Listing 4.18 illustrates

```
$fileToMove = $_FILES['file1']['tmp_name'];
$destination = "./upload/" . $_FILES["file1"]["name"];
if (move_uploaded_file($fileToMove,$destination)) {
    echo "The file was uploaded and moved successfully!";
}
else {
    echo "there was a problem moving the file";
}
```

Listing 4.18 Using `move_uploaded_file()` function

a simple use of the function. Note that the upload location uses `./upload/`, which means the file will be uploaded to a subdirectory named **upload** under the current directory.

4.5 reading/Writing Files

Before the age of the ubiquitous database, software relied on storing and accessing data in files. In web development, the ability to read and write to text files remains an important technical competency. Even if your site uses a database for storing its information, the fact that the PHP file functions can read/write from a file or from an external website (i.e., from a URL) means that file system functions still have relevance even in the age of database-driven websites.

There are two basic techniques for read/writing files in PHP:

- **Stream access.** In this technique, our code will read just a small portion of the file at a time. While this does require more careful programming, it is the most memory-efficient approach when reading very large files.
- **All-In-Memory access.** In this technique, we can read the entire file into memory (i.e., into a PHP variable). While not appropriate for large files, it does make processing of the file extremely easy.

4.5.1 stream access

To those of you familiar with functions like `fopen()`, `fclose()`, and `fgets()` from the C programming language, this first technique will be second nature to you. In the C-style file access you separate the acts of opening, reading, and closing a file.

The function `fopen()` takes a file location or URL and access mode as parameters. The returned value is a **stream resource**, which you can then read sequentially. Some of the common modes are “r” for read, “rw” for read and write, and “c,” which creates a new file for writing.

Once the file is opened, you can read from it in several ways. To read a single line, use the `fgets()` function, which will return false if there is no more data, and if it reads a line it will advance the stream forward to the next one so you can use the `===` check to see if you have reached the end of the file. To read an arbitrary amount of data (typically for binary files), use `fread()` and for reading a single character use `fgetc()`. Finally, when finished processing the file you must close it using `fclose()`. Listing 4.14 illustrates a script using `fopen()`, `fgets()`, and `fclose()` to read a file and echo it out (replacing new lines with `
` tags).

```
$f = fopen("sample.txt", "r");
$ln = 0;
while ($line = fgets($f)) {
    $ln++;
    printf("%2d: ", $ln);
    echo $line . "<br>";
}
fclose($f);
```

Listing 4.14 Opening, reading lines, and closing a file

To write data to a file, you can employ the `fwrite()` function in much the same way as `fgets()`, passing the file handle and the string to write. However, as you do more and more processing in PHP, you may find yourself wanting to read or write entire files at once. In support of these situations there are simpler techniques, which we will now explore.

4.5.2 in-Memory File access

While the previous approach to reading/writing files gives you complete control, the programming requires more care in dealing with the streams, file handles, and other low-level issues. The alternative simpler approach is much easier to use, at the cost of relinquishing fine-grained control. The functions shown in Table 4.3 provide a simpler alternative to the processing of a file in PHP.

Function	Description
file()	Reads the entire file into an array, with each array element corresponding to one line in the file
file_get_contents	Reads the entire file into a string variable
file_put_contents	Writes the contents of a string variable out to a file

table 4.3 In-Memory File Functions

The `file_get_contents()` and `file_put_contents()` functions allow you to read or write an entire file in one function call. To read an entire file into a variable you can simply use:

```
$fileAsString = file_get_contents(FILENAME);
```

To write the contents of a string `$writeeme` to a file, you use

```
file_put_contents(FILENAME, $writeeme);
```

These functions are especially convenient when used in conjunction with PHP's many powerful string-processing functions. For instance, let us imagine we have a comma-delimited text file that contains information about paintings, where each line in the file corresponds to a different painting:

```
01070,Picasso,The Actor,1404
01080,Picasso,Family of Saltimbanques,1405
02070,Matisse,The Red Madras Headdress,1407
05010,David,The Oath of the Horatii,1784
```

To read and then parse this text file is quite straightforward, as shown in Listing 4.20.

```
// read the file into memory; if there is an error then stop processing
$paintings = file($filename) or die('ERROR: Cannot find file');

// our data is comma-delimited
$delimiter = ',';

// loop through each line of the file
foreach ($paintings as $painting) {

    // returns an array of strings where each element in the array
    // corresponds to each substring between the delimiters
```

```
$paintingFields = explode($delimiter, $painting);
```

```
$id= $paintingFields[0];
$artist = $paintingFields[1];
$title = $paintingFields[2];
$year = $paintingFields[3];
```

```
// do something with this data
```

```
- - -
```

```
}
```

Listing 4.20 Processing a comma-delimited file

MODULE-4 PHP Classes and Objects

4.1.1 Object-Oriented Over view

Unlike JavaScript, PHP is a full-fledged object-oriented language with many of the syntactic constructs popularized in languages like Java and C++. Although earlier versions of PHP did not support all of these object-oriented features, PHP versions after 5.0 do. There are only a handful of classes included in PHP, some of which will be demonstrated in detail. The usage of objects will be illustrated alongside their definition for increased clarity.

4.1.1.1 terminology

The notion of programming with objects allows the developer to think about an item with particular **properties** (also called attributes or **data members**) and methods (functions). The structure of these **objects** is defined by **classes**, which outline the properties and methods like a blueprint. Each variable created from a class is called an object or **instance**, and each object maintains its own set of variables, and behaves (largely) independently from the class once created.

Figure 4.1.1 illustrates the differences between a class, which defines an object's properties and methods, and the objects or instances of that class.



FIGURE 10.1 Relationship between a class and its objects

4.1.1.2 the Unified Modeling Language

When discussing classes and objects, it helps to have a quick way to visually represent them. The standard diagramming notation for object-oriented design is **UML (Unified Modeling Language)**. UML is a succinct set of graphical techniques to describe software design. Some integrated development environments (IDEs) will even generate code from UML diagrams.

Several types of UML diagram are defined. Class diagrams and object diagrams, in particular, are useful to us when describing the properties, methods, and relationships between classes and objects. Throughout this and subsequent chapters, we will be illustrating concepts with UML diagrams when appropriate. For a complete definition of UML modeling syntax, look at the Object Modeling Group's living specification.¹

To illustrate classes and objects in UML, consider the artist we have looked at in the Art Case Study. Every artist has a first name, last name, birth date, birth city, and death date. Using objects we can encapsulate those properties together into a class definition for an Artist. Figure 4.1.2 illustrates a UML class diagram, which shows an **Artist** class and multiple **Artist** objects, each object having its own properties.

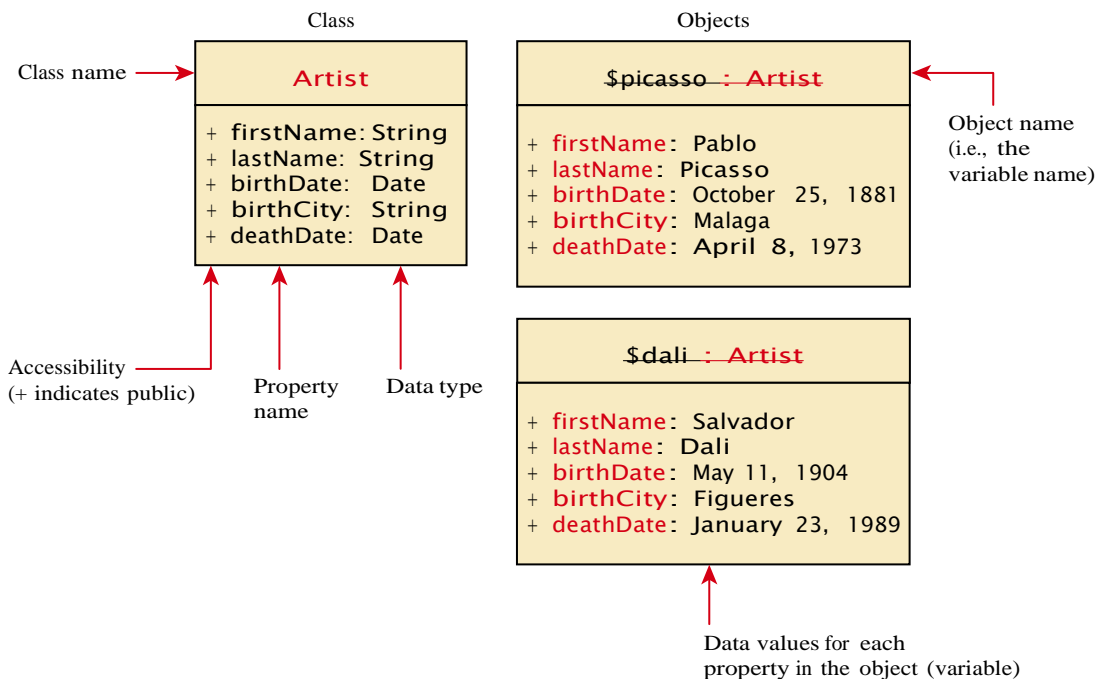


Figure 4.1.2 Relationship between a class and its objects in UML

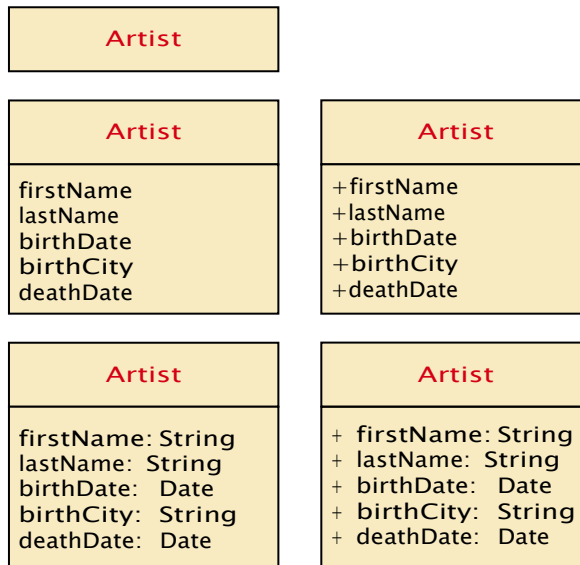


Figure 4.1.3 Different levels of UML detail

In general, when diagramming we are almost always interested in the classes and not so much in the objects. Depending on whether one is interested in showing the big picture, with many classes and their relationships, or showing instead exact details of a class, there is a wide variety of flexibility in how much detail you want to show in your class diagrams, as shown in Figure 4.1.3.

4.1.1.3 Differences between server and Desktop Objects

If you have programmed desktop software using object-oriented methods before, you will need to familiarize yourself with the key differences between desktop and client-server object-oriented analysis and design (OOAD). One important distinction between web programming and desktop application programming is that the objects you create (normally) only exist until a web script is terminated. While desktop software can load an object into memory and make use of it for several user interactions, a PHP object is loaded into memory only for the life of that HTTP request. Figure 4.1.4 shows an illustration of the lifetimes of objects in memory between a desktop and a browser application.

For this reason, we must use classes differently than in the desktop world, since the object must be recreated and loaded into memory for each request that requires it. Object-oriented web applications can see significant performance degradation

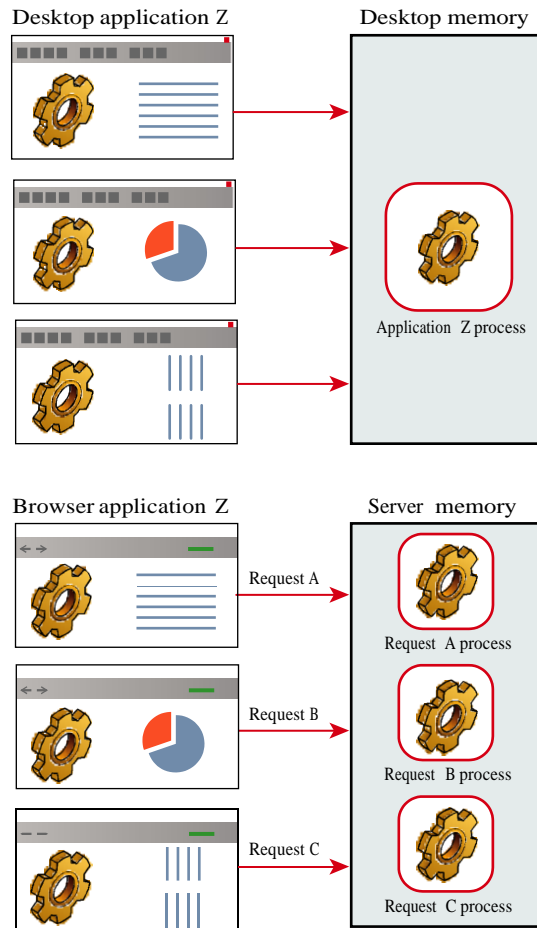


Figure 4.1.4 Lifetime of objects in memory in web versus desktop applications

compared to their functional counterparts if objects are not utilized correctly. Remember, unlike a desktop, there are potentially many thousands of users making requests at once, so not only are objects destroyed upon responding to each request, but memory must be shared between many simultaneous requests, each of which may load objects into memory.

It is possible to have objects persist between multiple requests using serialization, which is the rapid storage and retrieval of an object (and which is covered in Chapter 13). However, serialization does not address the inherent inefficiency of recreating objects each time a new request comes in.

4.1.2 Classes and Objects in php

In order to utilize objects, one must understand the classes that define them. Although a few classes are built into PHP, you will likely be working primarily with your own classes.

Classes should be defined in their own files so they can be imported into multiple scripts. In this book we denote a class file by using the naming convention **classname.class.php**. Any PHP script can make use of an external class by using one of the include statements or functions that you encountered in Chapter 8, that is, `include`, `include_once`, `require`, or `require_once`; in Chapter 14, you will learn how to use the `spl_autoload_register()` function to automatically load class files without explicitly including them. Once a class has been defined, you can create as many instances of that object as memory will allow using the new keyword.

4.1.2.1 Defining Classes

The PHP syntax for defining a class uses the `class` keyword followed by the class name and `{ }` braces.² The properties and methods of the class are defined within the braces. The **Artist** class with the properties illustrated in Figure 4.1.2 is defined using PHP in Listing 4.1.1.

```
class Artist {  
    public    $firstName;  
    public    $lastName;  
    public    $birthDate;  
    public    $birthCity;  
    public    $deathDate;  
}
```

Listing 4.1.1 A simple Artist class

Each property in the class is declared using one of the keywords `public`, `protected`, or `private` followed by the property or variable name. The differences between these keywords will be covered in Section 4.1.2.6.

4.1.2.2 instantiating Objects

It's important to note that defining a class is not the same as using it. To make use of a class, one must **instantiate** (create) objects from its definition using the new keyword. To create two new instances of the `Artist` class called `$picasso` and `$dali`, you instantiate two new objects using the new keyword as follows:

```
$picasso = new Artist();
$dali = new Artist();
```

Notice that assignment is right to left as with all other assignments in PHP. Shortly you will see how to enhance the initialization of objects through the use of constructors.

4.1.2.3 properties

Once you have instances of an object, you can access and modify the properties of each one separately using the variable name and an arrow (`->`), which is constructed from the dash and greater than symbols. Listing 4.1.2 shows code that defines the two `Artist` objects and then sets all the properties for the `$picasso` object.

```
$picasso = new Artist();
$dali = new Artist();
$picasso->firstName = "Pablo";
$picasso->lastName = "Picasso";
$picasso->birthCity = "Malaga";
$picasso->birthDate = "October 25 1881";
$picasso->deathDate = "April 8 1973";
```

Listing 4.1.2 Instantiating two `Artist` objects and setting one of those object's properties

4.1.2.4 Constructors

While the code in Listing 4.1.2 works, it takes multiple lines and every line of code introduces potential maintainability problems, especially when we define more artists. Inside of a class definition, you should therefore define **constructors**, which lets you specify parameters during instantiation to initialize the properties within a class right away.

In PHP, constructors are defined as functions (as you shall see, all methods use the `function` keyword) with the name `__construct()`. (Note: there are *two* underscores `_` before the word `construct`.) Listing 4.1.3 shows an updated `Artist` class definition that now includes a constructor. Notice that in the constructor each parameter is assigned to an internal class variable using the `$this->` syntax. Inside of a class you **must** always use the `$this` syntax to reference all properties and methods associated with this particular instance of a class.

```
class Artist {
    // variables from previous listing still go here
    ...

    function __construct($firstName, $lastName, $city, $birth,
                        $death=null) {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
        $this->birthCity = $city;
        $this->birthDate = $birth;
        $this->deathDate = $death;
    }
}
```

Listing 4.1.3 A constructor added to the class definition

Notice as well that the `$death` parameter in the constructor is initialized to `null`; the rationale for this is that this parameter might be omitted in situations where the specified artist is still alive.

This new constructor can then be used when instantiating so that the long code in Listing 4.1.2 becomes the simpler:

```
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25,1881",
                    "Apr 8,1973");
$dali = new Artist("Salvador","Dali","Figures","May 11 1904",
                  "Jan 23 1989");
```

4.1.2.5 Methods

Objects only really become useful when you define behavior or operations that they can perform. In object-oriented lingo these operations are called **methods** and are like functions, except they are associated with a class. They define the tasks each instance of a class can perform and are useful since they associate behavior

with objects. For our artist example one could write a method to convert the artist's details into a string of formatted HTML. Such a method is defined in Listing 4.1.4.

```
class Artist {
    ...
    public function outputAsTable() {
        $table = "<table>";
        $table .= "<tr><th colspan='2'>";
        $table .= $this->firstName . " " . $this->lastName;
        $table .= "</th></tr>";
        $table .= "<tr><td>Birth:</td>";
        $table .= "<td>" . $this->birthDate;
        $table .= "(" . $this->birthCity . "</td></tr>";
        $table .= "<tr><td>Death:</td>";
        $table .= "<td>" . $this->deathDate . "</td></tr>";
        $table .= "</table>";
        return $table;
    }
}
```

Listing 4.1.4 Method definition

To output the artist, you can use the reference and method name as follows:

```
$picasso = new Artist( ... )
echo $picasso->outputAsTable();
```

The UML class diagram in Figure 4.1.2 can now be modified to include the newly defined `outputAsTable()` method as well as the constructor and is shown in Figure 4.1.5. Notice that two versions of the class are shown in Figure 4.1.5, to illustrate that there are different ways to indicate a PHP constructor in UML.

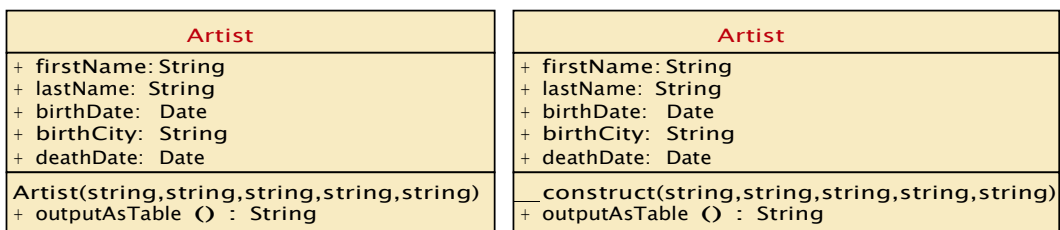


Figure 4.1.5 Updated class diagram

4.1.2.6 visibility

The **visibility** of a property or method determines the accessibility of a **class member** (i.e., a property or method) and can be set to **public**, **private**, or **protected**. Figure 4.1.6 illustrates how visibility works in PHP.

As can be seen in Figure 4.1.6, the **public** keyword means that the property or

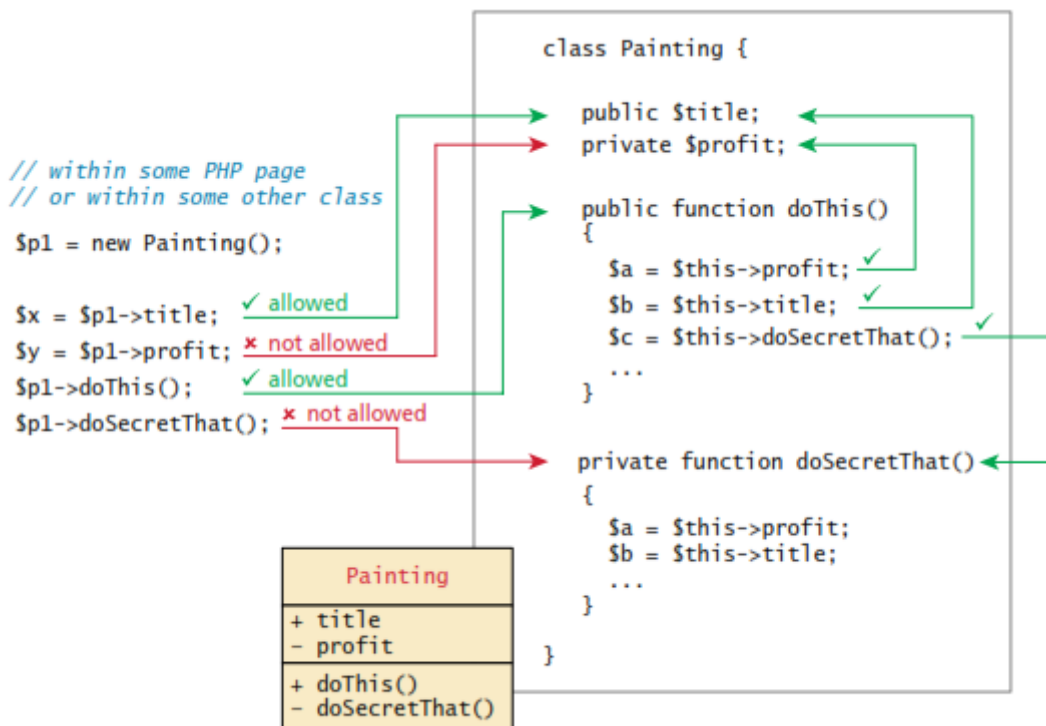


FIGURE 10.6 Visibility of class members

method is accessible to any code that has a reference to the object. The **private** keyword sets a method or variable to only be accessible from within the class. This means that we cannot access or modify the property from outside of the class,

even if we have a reference to it as shown in Figure 4.1.6. The `protected` keyword will be discussed later after we cover inheritance. For now consider a protected property or method to be private. In UML, the "+" symbol is used to denote public properties and methods, the "-" symbol for private ones, and the "#" symbol for protected ones.

4.1.2.7 static Members

A **static** member is a property or method that all instances of a class share. Unlike an instance property, where each object gets its own value for that property, there is only one value for a class's static property.

To illustrate how a static member is shared between instances of a class, we will add the static property `artistCount` to our **Artist** class, and use it to keep a count of how many **Artist** objects are currently instantiated. This variable is declared static by including the **static** keyword in the declaration:

```
public static $artistCount = 0;
```

For illustrative purposes we will also modify our constructor, so that it increments this value, as shown in Listing 4.1.5.

```
class Artist {  
    public static $artistCount = 0;  
    public $firstName;  
    public $lastName;  
    public $birthDate;  
    public $birthCity;  
    public $deathDate;  
  
    function __construct($firstName, $lastName, $city, $birth,  
                        $death=null) {  
        $this->firstName = $firstName;  
        $this->lastName = $lastName;  
        $this->birthCity = $city;  
        $this->birthDate = $birth;  
        $this->deathDate = $death;  
        self::$artistCount++;  
    }  
}
```

Listing 4.1.5 Class definition modified with static members

Notice that you do not reference a static property using the `$this->` syntax, but rather it has its own **self::** syntax. The rationale behind this change is to force the programmer to understand that the variable is static and not associated with an instance (`$this`). This static variable can also be accessed without any instance of an **Artist** object by using the class name, that is, via `Artist::$artistCount`.

To illustrate the impact of these changes look at Figure 4.1.7, where the shared property is underlined (UML notation) to indicate its static nature and the shared reference between multiple instances is illustrated with arrows, including one reference without any instance.

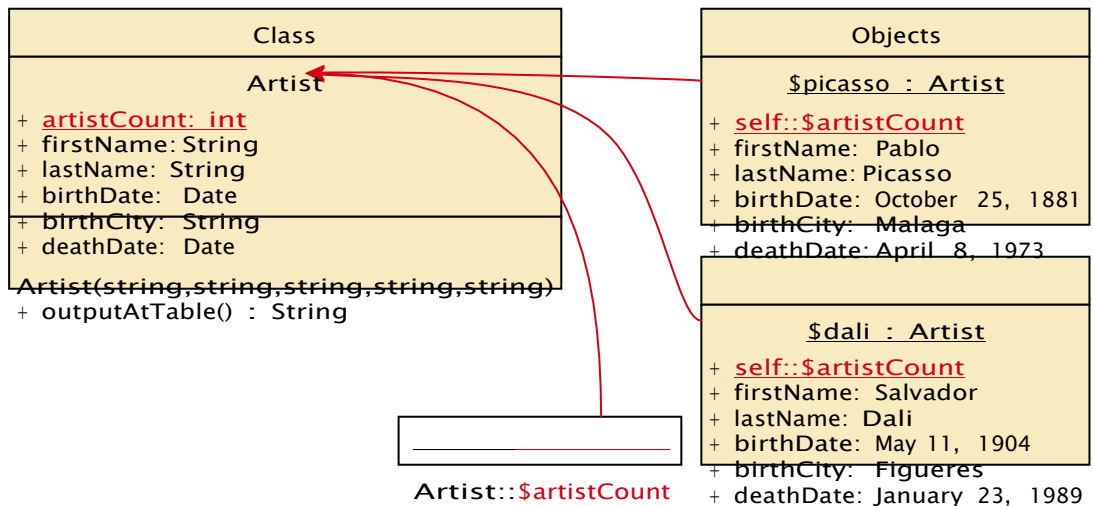


Figure 4.1.7 A static property

Static methods are similar to static properties in that they are globally accessible (if public) and are not associated with particular objects. It should be noted that static methods cannot access instance members. Static methods are called using the same double colon syntax as static properties.

Why would you need a static member? Static members tend to be used relatively infrequently. However, classes sometimes have data or operations that are independent of the instances of the class. We will find them helpful when we create a more sophisticated class hierarchy in Chapter 14 on Web Application Design.

4.1.2.8 Class Constants

If you want to add a property to a class that is constant, you could do it with static properties as shown above. However, constant values can be stored more efficiently as class constants so long as they are not calculated or updated. Example constants might include strings to define a commonly used literal. They are added to a class using the `const` keyword.

```
const EARLIEST_DATE = 'January 1, 1200';
```

Unlike all other variables, constants don't use the `$` symbol when declaring or using them. They can be accessed both inside and outside the class using `self::EARLIEST_DATE` in the class and `classReference::EARLIEST_DATE` outside.

4.1.3 Object-Oriented Design

Now that you have a basic understanding of how to define and use classes and objects, you can start to get the benefits of software engineering patterns, which encourage understandable and less error-prone code. The object-oriented design of software offers many benefits in terms of modularity, testability, and reusability.

4.1.3.1 Data encapsulation

Perhaps the most important advantage to object-oriented design is the possibility of **encapsulation**, which generally refers to restricting access to an object's internal components. Another way of understanding encapsulation is: it is the hiding of an object's implementation details.

A properly encapsulated class will define an interface to the world in the form of its public methods, and leave its data, that is, its properties, hidden (that is, private). This allows the class to control exactly how its data will be used.

If a properly encapsulated class makes its properties private, then how do you access them? The typical approach is to write methods for accessing and modifying properties rather than allowing them to be accessed directly. These methods are commonly called **getters and setters** (or accessors and mutators). Some development environments can even generate getters and setters automatically.

A getter to return a variable's value is often very straightforward and should not modify the property. It is normally called without parameters, and returns the property from within the class. For instance:

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

Setter methods modify properties, and allow extra logic to be added to prevent properties from being set to strange values. For example, we might only set a date property if the setter was passed an acceptable date:

```
public function setBirthDate($birthdate){  
    // set variable only if passed a valid date string  
    $date = date_create($birthdate);  
  
    if ( ! $date ) {  
        $this->birthDate = $this->getEarliestAllowedDate();  
    }  
    else {  
        // if very early date then change it to  
        // the earliest allowed date  
        if ( $date < $this->getEarliestAllowedDate() ) {  
            $date = $this->getEarliestAllowedDate();  
        }  
        $this->birthDate = $date;  
    }  
}
```

}

Listing 4.1.6 shows the modified **Artist** class with getters and setters. Notice that the properties are now private. As a result, the code from Listing 4.1.2 will no longer work for our class since it tries to reference and modify private properties. Instead we would have to use the corresponding getters and setters. Notice as well that two of the setter functions have a fair bit of validation logic in them; this illustrates one of the key advantages to using getters and setters: that the class can handle the responsibility of ensuring its own data validation. And since the setter functions are performing validation, the constructor for the class should use the setter functions to set the values, as shown in this example.

```
class Artist {
    const EARLIEST_DATE = 'January 1, 1200';

    private static $artistCount = 0;
    private $firstName;
    private $lastName;
    private $birthDate;
    private $deathDate;
    private $birthCity;

    // notice constructor is using setters instead
    // of accessing properties
    function __construct($firstName, $lastName, $birthCity, $birthDate,
                        $deathDate) {
        $this->setFirstName($firstName);
        $this->setLastName($lastName);
        $this->setBirthCity($birthCity);
        $this->setBirthDate($birthDate);
        $this->setDeathDate($deathDate);
        self::$artistCount++;
    }
    // saving book space by putting each getter on single line
    public function getFirstName() { return $this->firstName; }
    public function getLastName() { return $this->lastName; }
    public function getBirthCity() { return $this->birthCity; }
    public function getBirthDate() { return $this->birthDate; }
    public function getDeathDate() { return $this->deathDate; }
    public static function getArtistCount() { return self::$artistCount; }
    public function getEarliestAllowedDate () {
        return date_create(self::EARLIEST_DATE);
    }

    public function setLastName($lastName)
    { $this->lastName = $lastName; }
    public function setFirstName($firstName)
    { $this->firstName = $firstName; }
```

```

public function setBirthCity($birthCity)
{ $this->birthCity = $birthCity; }

public function setBirthDate($birthdate) {
    // set variable only if passed a valid date string
    $date = date_create($birthdate);
    if ( ! $date ) {
        $this->birthDate = $this->getEarliestAllowedDate();
    }
    else {

        // if very early date then change it to earliest allowed date
        if ( $date < $this->getEarliestAllowedDate() ) {
            $date = $this->getEarliestAllowedDate();
        }
        $this->birthDate = $date;
    }
}

public function setDeathDate($deathdate){
    // set variable only if passed a valid date string
    $date = date_create($deathdate);

    if ( ! $date ) {
        $this->deathDate = $this->getEarliestAllowedDate();
    }
    else {
        // set variable only if later than birth date
        if ( $date > $this->getBirthDate() ) {
            $this->deathDate = $date;
        }
        else {
            $this->deathDate = $this->getBirthDate();
        }
    }
}
}

```

Listing 4.1.6 Artist class with better encapsulation

Two forms of the updated UML class diagram for our data encapsulated class are shown in Figure 4.1.8. The longer one includes all the getter and setter methods. It is quite common, however, to exclude the getter and setter methods from a class

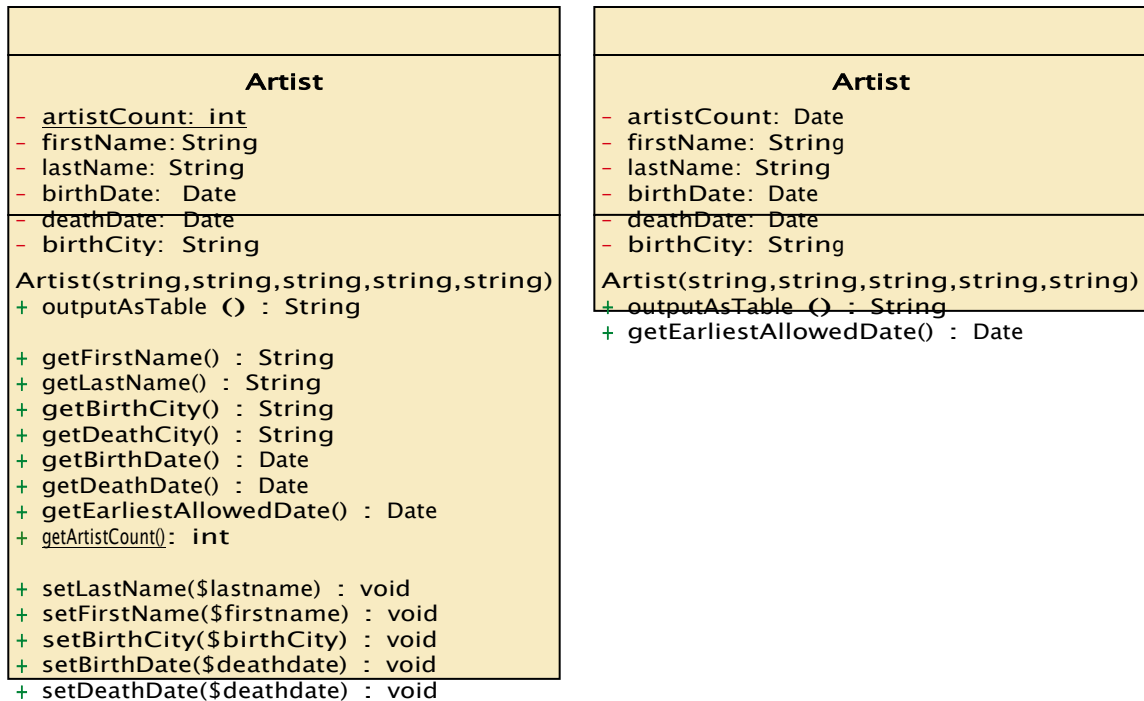


Figure 4.1.8 Class diagrams for fully encapsulated Artist class

diagram; we can just assume they exist due to the private properties in the property compartment of the class diagram.

Now that the encapsulated **Artist** class is defined, how can one use it? Listing 4.1.7 demonstrates how the **Artist** class could be used and tested.

```
<html>
<body>
<h2>Tester for Artist class</h2>

<?php
// first must include the class definition
include 'Artist.class.php';

// now create one instance of the Artist class
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25,1881",
                    "Apr 8,1973");
```

(continued)

```
// output some of its fields to test the getters
echo $picasso->getLastName() . ' : ' ;
echo date_format($picasso->getBirthDate(),'d M Y') . ' to ' ;
echo date_format($picasso->getDeathDate(),'d M Y') . '<hr>' ;

// create another instance and test it
$dali = new Artist("Salvador","Dali","Figures","May 11,1904",
                  "January 23,1989");

echo $dali->getLastName() . ' : ' ;
echo date_format($dali->getBirthDate(),'d M Y') . ' to ' ;
echo date_format($dali->getDeathDate(),'d M Y') . '<hr>' ;

// test the output method
echo $picasso->outputAsTable();

// finally test the static method: notice its syntax
echo '<hr>' ;
echo 'Number of Instantiated artists: ' . Artist::getArtistCount();

?>
</body>

</html>
```

Listing 4.1.7 Using the encapsulated class

4.1.3.2 inheritance

Along with encapsulation, **inheritance** is one of the three key concepts in object-oriented design and programming (we will cover the third, polymorphism, next). Inheritance enables you to create new PHP classes that reuse, extend, and modify the behavior that is defined in another PHP class. Although some languages allow it, PHP only allows you to inherit from one class at a time.

A class that is inheriting from another class is said to be a **subclass** or a **derived class**. The class that is being inherited from is typically called a **superclass** or a **base class**. When a class inherits from another class, it inherits all of its public and protected methods and properties. Figure 4.1.9 illustrates how inheritance is shown in a UML class diagram.

Just as in Java, a PHP class is defined as a subclass by using the **extends** keyword.

```
class Painting extends Art { . . . }
```

referencing base Class Members

As mentioned above, a subclass inherits the public and protected members of the base class. Thus in the following code based on Figure 4.1.9, both of the references

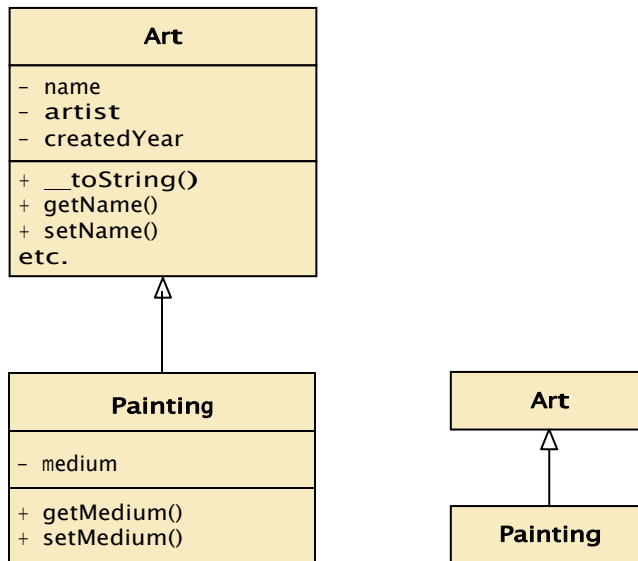


Figure 4.1.9 UML class diagrams showing inheritance

will work because it is *as if* the base class public members are defined within the subclass.

```

$p = new Painting();
...
// these references are ok
echo $p->getName();    // defined in base class
echo $p->getMedium();  // defined in subclass
  
```

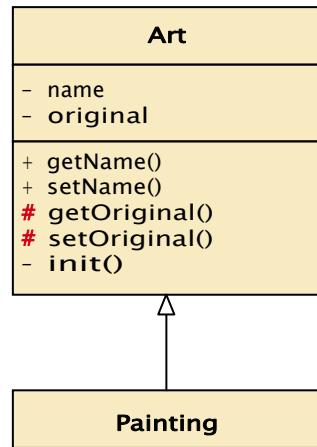
Unlike in languages like Java or C#, in PHP any reference to a member in the base class requires the addition of the `parent::` prefix instead of the `$this->` prefix. So within the `Painting` class, a reference to the `getName()` method would be:

```
parent::getName()
```

It is important to note that **private** members in the base class are **not** available to its subclasses. Thus, within the `Painting` class, a reference like the following would **not** work.

```
$abc = parent::name; // would not work within the Painting class
```

If you want a member to be available to subclasses but not anywhere else, you can use the **protected** access modifier, which is shown in Figure 4.1.4.1.



```

class Painting extends Art {
    ...
    private function foo() {
        ...
        // these are allowed
        ✓ $w = parent::getName();
        ✓ $x = parent::getOriginal();

        // this is not allowed
        ✗ $y = parent::init();
    }
}
    
```

// in some page or other class

```

$p = new Painting();
$a = new Art();
    
```

// neither of these references are allowed

```

✗ $w = $p >getOriginal();
✗ $y = $a >getOriginal();
    
```

Figure 4.1.4.1 Protected access modifier

To best see the potential benefits of inheritance, let us look at a slightly *extended* example involving different types of art. For our previously defined **Artist** class, imagine we include a list of works of art for each artist. We might manage that list inside the class with an array of objects of type **Art**. Such a list must allow objects of many types, for what is art after all? We can have music works, paintings, writings, sculptures, prints, inventions, and more, all considered **Art**. We will therefore use the idea of art as the basis for demonstrating inheritance in PHP. Figure 4.1.11 shows the relationship of the classes in our example.

In this example, paintings, sculptures, and art prints are all types of **Art**, but they each have unique attributes (a **Sculpture** has weight, while a **Painting** has a medium, such as oil or acrylic, while an **ArtPrint** is a special type of **Painting**). In the art world, a print is like a certified copy of the original painting. A print is typically signed by the artist and given a print run number, which we will record in the `printNumber` property. Finally, notice that the **Art** class has an association with **Artist**, meaning that the `artist` property will contain an object of type **Artist**.

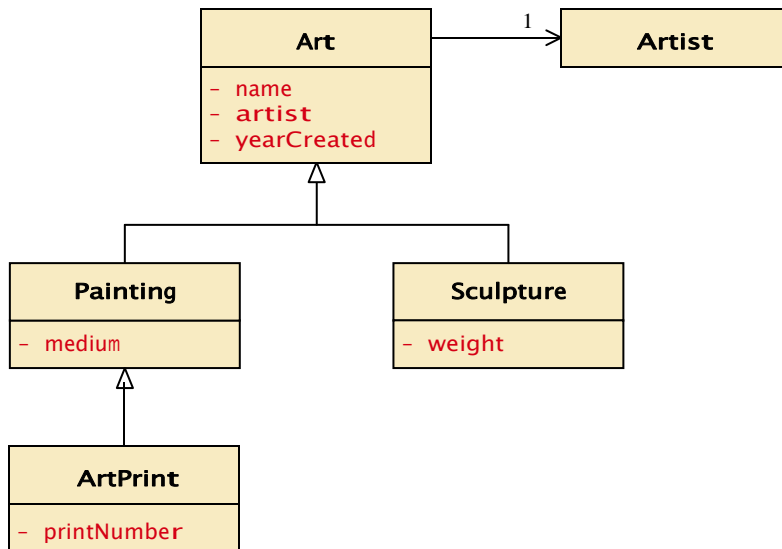


Figure 4.1.11 Class diagram for Art example

Listing 4.1.8 lists the implementation of these four classes. Notice how the sub-class constructors invoke the constructors of their base class and that many of the setter methods are performing some type of validation. Notice as well the use of the **abstract** keyword in the first line of the definition of the **Art** class. An abstract class is one that cannot be instantiated. In the context of art, there can be concrete types of art, such as paintings, sculpture, or prints, but not “art” in general, so it makes sense to programmatically model this limitation via the **abstract** keyword.

```

/* The abstract class that contains functionality required by all
types of Art */

abstract class Art {
    private $name;
    private $artist;
    private $yearCreated;

    function __construct($year, $artist, $name) {
        $this->setYear($year);
        $this->setArtist($artist);
        $this->setName($name);
    }
    public function getYear() { return $this->yearCreated; }
    public function getArtist() { return $this->artist; }
    public function getName() { return $this->name; }
}

```

(continued)

```

        public function setYear($year) {
            if (is_numeric($year))
                $this->yearCreated = $year;
        }
        public function setArtist($artist) {
            if ((is_object($artist)) && ($artist instanceof Artist))
                $this->artist = $artist;
        }
        public function setName($name) {
            $this->name = $name;
        }
    }

    public function __toString() {
        $line = "Year:" . $this->getYear();
        $line .= ", Name: " . $this->getName();
        $line .= ", Artist: " . $this->getArtist()->getFirstName() . ' ' ;
        $line .= $this->getArtist()->getLastName();
        return $line;
    }
}

class Painting extends Art {
    private $medium;

    function __construct($year, $artist, $name, $medium) {
        parent::__construct($year, $artist, $name);
        $this->setMedium($medium);
    }
    public function getMedium(){ return $this->medium; }
    public function setMedium($medium) {
        $this->medium = $medium;
    }
    public function __toString() {
        return parent::__toString() . ", Medium: " . $this->getMedium();
    }
}

class Sculpture extends Art {
    private $weight;

    function __construct($year, $artist, $name, $weight) {
        parent::__construct($year, $artist, $name);
        $this->setWeight($weight);
    }
    public function getWeight() { return $this->weight; }
    public function setWeight($weight) {

```

```
        if (is_numeric($weight))
            $this->weight = $weight;
    }
    public function __toString() {
        return parent::__toString() . ", Weight: " . $this->getWeight()
            . "kg";
    }
}

class ArtPrint extends Painting {
    private $printNumber;

    function __construct($year, $artist, $name, $medium, $printNumber) {
        parent::__construct($year, $artist, $name, $medium);
        $this->setPrintNumber($printNumber);
    }
    public function getPrintNumber() { return $this->printNumber; }
    public function setPrintNumber($printNumber) {
        if (is_numeric($printNumber))
            $this->printNumber = $printNumber;
    }
    public function __toString() {
        return parent::__toString() . ", Print Number: "
            . $this->getPrintNumber();
    }
}
```

Listing 4.1.8 Class implementations for Listing 4.1.11

Whenever you create classes, you will eventually need to use them. The authors often find it useful to create tester pages that verify a class works as expected. Listing 4.1.9 illustrates a typical tester. Notice that since the **Art** class has a data member of type **Artist**, it is possible to also access the **Artist** properties through the **Art** object.

```
<?php
// include the classes
include 'Artist.class.php';
include 'Art.class.php';
include 'Painting.class.php';
include 'Sculpture.class.php';
include 'ArtPrint.class.php';
```

(continued)

```
// instantiate some sample objects
$picasso = new Artist("Pablo","Picasso","Malaga","May 11,904",
    "Apr 8, 1973");
$guernica = new Painting("1937",$picasso,"Guernica","Oil on
    canvas");
$stein = new Painting("1907",$picasso,"Portrait of Gertrude Stein",
    "Oil on canvas");
$woman = new Sculpture("1909",$picasso,"Head of a Woman", 30.5);
$bowl = new ArtPrint("1912",$picasso,"Still Life with Bowl and Fruit",
    "Charcoal on paper", 25);

?>
<html>
<body>
<h1>Tester for Art Classes</h1>

<h2>Paintings</h2>
<p><em>Use the __toString() methods </em></p>
<p><?php echo $guernica; ?></p>
<p><?php echo $stein; ?></p>

<p><em>Use the getter methods </em></p>
<?php
echo $guernica->getName() . ' by '
    . $guernica->getArtist()->getLastName();
?>

<h2>Sculptures</h2>
<p><?php echo $woman; ?></p>

<h2>Art Prints</h2>
<?php
echo 'Year: ' . $bowl->getYear() . '<br/>';
echo 'Artist: ';
echo $bowl->getArtist()->getFirstName() . ' ';
echo $bowl->getArtist()->getLastName() . ' (';
echo date_format( $bowl->getArtist()->getBirthDate() , 'd M Y') . ' - ' .
echo date_format( $bowl->getArtist()->getDeathDate() , 'd M Y');
echo ')<br/>';
echo 'Name: ' . $bowl->getName() . '<br/>';
echo 'Medium: ' . $bowl->getMedium() . '<br/>';
echo 'Print Number: ' . $bowl->getPrintNumber() . '<br/>';
?>
</body>
</html>
```

Listing 4.1.9 Using the classes

inheriting Methods

Every method defined in the base/parent class can be overridden when extending a class, by declaring a function with the same name. A simple example of overriding can be found in Listing 4.1.8 in which each subclass overrides the `__toString()` method.

To access a public or protected method or property defined within a base class from within a subclass, you do so by prefixing the member name with `parent::`. So to access the parent's `__toString()` method you would simply use `parent::__toString()`.

parent Constructors

If you want to invoke a parent constructor in the derived class's constructor, you can use the `parent::` syntax and call the constructor on the first line `parent::__construct()`. This is similar to calling other parent methods, except that to use it we *must* call it at the beginning of our constructor.

4.1.3.3 polymorphism

Polymorphism is the third key object-oriented concept (along with encapsulation and inheritance). In the inheritance example in Listing 4.1.8, the classes `Sculpture` and `Painting` inherited from `Art`. Conceptually, a sculpture *is a* work of art and a painting *is a* work of art. **Polymorphism** is the notion that an object can in fact be multiple things at the same time. Let us begin with an instance of a `Painting` object named `$guernica` created as follows:

```
$guernica = new Painting("1937",$picasso,"Guernica","Oil on canvas");
```

The variable `$guernica` is both a `Painting` object and an `Art` object due to its inheritance. The advantage of polymorphism is that we can manage a list of `Art` objects, and call the same overridden method on each. Listing 4.1.4.1 illustrates polymorphism at work.

```
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25, 1881",  
                    "Apr 8, 1973");  
  
// create the paintings  
$guernica = new Painting("1937",$picasso,"Guernica","Oil on canvas");  
$chicago = new Sculpture("1967",$picasso,"Chicago", 454);
```

(continued)

```
// create an array of art
$works = array();
$works[0] = $guernica;
$works[1] = $chicago;
// to test polymorphism, loop through art array
foreach ($works as $art)
{
    // the beauty of polymorphism:
    // the appropriate __toString() method will be called!
    echo $art;
}

// add works to artist... any type of art class will work
$picasso->addWork($guernica);
$picasso->addWork($chicago);
// do the same type of loop
foreach ($picasso->getWorks() as $art) {
    echo $art; // again polymorphism at work
}
```

Listing 4.1.4.1 Using polymorphism

Due to overriding methods in child classes, the actual method called will depend on the type of the object! Using `__toString()` as an example, a `Painting` will output its name, date, and medium and a `Sculpture` will output its name, date, and weight. The code in Listing 4.1.4.1 calls `echo` on both a `Painting` and a `Sculpture` with different output for each shown below:

```
Date:1937, Name:Guernica, Medium: Oil on canvas
Date:1967, Name:Chicago, Weight: 454kg
```

The interesting part is that the correct `__toString()` method was called for both `Art` objects, based on their type. The formal notion of having a different method for a different class, all of which is determined at run time, is called **dynamic dispatching**. Just as each object can maintain its own properties, each object also manages its own table of methods. This means that two objects of the same type can have different implementations with the same name as in our `Painting/Sculpture` example. The point is that at *compile time*, we may not know what type each of the `Art` objects will be. Only at *run time* are the objects' types known, and the appropriate method selected.

4.1.3.4 Object interfaces

An object **interface** is a way of defining a formal list of methods that a class **must** implement without specifying their implementation. Interfaces provide a mechanism for defining what a class can do without specifying how it does it, which is often a very useful design technique. The class infrastructure that will be defined in Chapter 14 makes use of interfaces.

Interfaces are defined using the **interface** keyword, and look similar to standard PHP classes, except an interface contains no properties and its methods do not have method bodies defined. For instance, an example interface might look like the following:

```
interface Viewable {  
    public function getSize();  
    public function getPNG();  
}
```

Notice that an interface contains only public methods, and instead of having a method body, each method is terminated with a semicolon.

In PHP, a class can be said to *implement* an interface, using the implements keyword:

```
class Painting extends Art implements Viewable { ... }
```

This means then that the class **Painting** must provide implementations (i.e., normal method bodies) for the **getSize()** and **getPNG()** methods.

When learning object-oriented development, it is not usually clear at first why interfaces are useful, so let us work through a quick example extending the art example further. So far, we have looked at paintings, sculptures, and prints as types of art. They are examples of art that is viewed (or in the lingo of interfaces, *viewable*). But one could imagine other types of art that are not viewed, such as music. In the case of music, it is not viewable, but *playable*. Other types of art, such as movies, are *both* viewable and playable.

With interfaces we can define these multiple ways of enjoying the art, and then classes derived from **Art** can declare what interfaces they implement. This allows us to define a more formal structure apart from the derived classes themselves. Listing 4.1.11 defines a **Viewable** interface, which defines methods to return a **png** image to represent the viewable piece of art and get its size. Since our existing **Painting** class is no doubt viewable, it should implement this interface by modifying our class definition and add an implementation for the methods in the interface not yet defined. We then declare that the **Painting** class implements the **Viewable** interface.

```

interface Viewable {
    public function getSize();
    public function getPNG();
}

class Painting extends Art implements Viewable {
    ...
    public function getPNG() {
        //return image data would go here
        ...
    }
    public function getSize() {
        //return image size would go here
        ...
    }
}

```

Listing 4.1.11 Painting class implementing an interface

Listing 4.1.12 defines another interface (Playable), and then two classes that use it.

```

interface Playable {
    public function getLength();
    public function getMedia();
}

class Music extends Art implements Playable {
    ...
    public function getMedia() {
        //returns the music
        ...
    }
    public function getLength() {
        //return the length of the music
    }
}

class Movie extends Painting implements Playable, Viewable {
    ...
    public function getMedia() {
        //return the movie
        ...
    }
}

```

```

public function getLength() {
    //return the length of the movie
    ...
}
public function getPNG() {
    //return image data
    ...
}
public function getSize() {
    //return image size would go here
    ...
}
}

```

Listing 4.1.12 Playable interface and multiple interface implementations

While PHP prevents us from inheriting from two classes, it does not prevent us from implementing two or more interfaces. The `Movie` class therefore extends from `Painting` but also implements the two interfaces `Viewable` and `Playable`. The diagram illustrating this relationship in UML is shown in Figure 4.1.12. In UML, interfaces are denoted through the `<<interface>>` stereotype. Classes that implement an interface are shown to implement using the same hollow triangles as inheritance but with dotted lines.

runtime Class and interface Determination

One of the things you may want to do in code as you are iterating polymorphically through a list of objects is ask what type of class this is, or what interfaces this

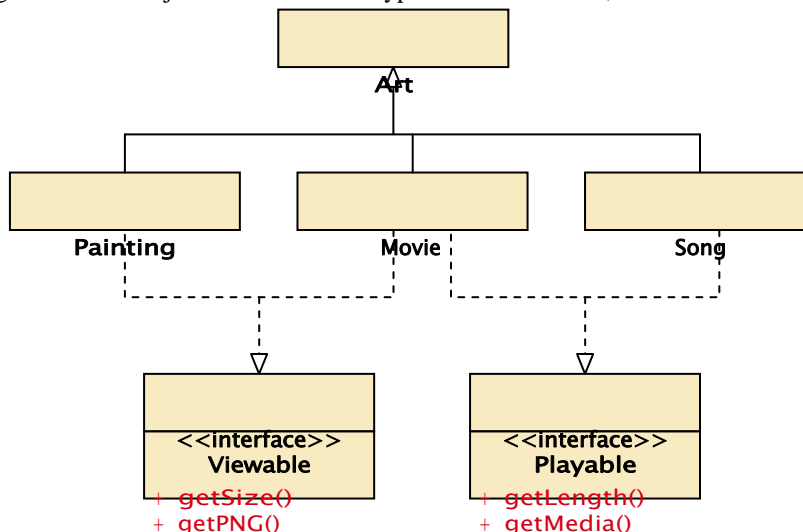


Figure 4.1.12 Indicating interfaces in a class diagram

object implements. Usually if you find yourself having to ask this too often, you are not using inheritance and interfaces in a correct object-oriented manner, since it is better to define logic inside the classes rather than put logic in your loops to determine what type of object this is. Nonetheless we can echo the class name of an object `$x` by using the `get_class()` function:

```
echo get_class($x);
```

Similarly we can access the parent class with:

```
echo get_parent_class($x);
```

To determine what interfaces this class has implemented, use the function `class_implements()`, which returns an array of all the interfaces implemented by this class or its parents.

```
$allInterfaces = class_implements($x);
```