PHP Arrays and Superglobals

Arrays

PHP supports arrays. An array is a data structure that stores one or more similar type of values in a single value. In PHP an array is actually an ordered map, which associates each value in the array with a key.

Defining and Accessing an Array

In PHP, the array() function is used to create an array: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

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

or

```
$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 using the square bracket notation.

```
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.

In PHP, there are three types of arrays:

- **Indexed arrays** - Arrays with a numeric index
- **Associative arrays** - Arrays with named keys
- **Multidimensional arrays** - Arrays containing one or more arrays

PHP Indexed Arrays

There are two ways to create indexed arrays:
The index can be assigned automatically (index always starts at 0), like this:

$cars = array("Volvo", "BMW", "Toyota");

or the index can be assigned manually:

```
$cars[0]="Volvo";
$cars[1]="BMW";
$cars[2] = "Toyota";
```

Example:

```
<?php
$cars=                    array("Volvo",                   "BMW",                    "Toyota");
$arrlength=count($cars);
for($x=                                                             0;$x<$arrlength;$x++){
                                                        echo               $cars[$x];
                                                         echo                 "<br>";
}
?>
```

PHP Associative Arrays

Associative arrays are arrays that use named keys that you assign to them. There are two ways to create an associative array:

$age = array("Peter"=>"35", "Ben"=>"37", "Joe"=>"43");

or:

```
$age['Peter']="35";
$age['Ben']="37";
$age['Joe'] = "43";
```

Example:

```
<?php
$age=                array("Peter"=>"35",             "Ben"=>"37",             "Joe"=>"43");

foreach($age                          as                               $x=>$x_value){
           echo           "Key="          .$x.          ",Value="         .$x_value;
                                                           echo               "<br>";
}
?>
```

Multidimensional Arrays

PHP also supports multidimensional arrays. The values for an array can be any PHP object, which includes other arrays.

Example 1

```
$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
```

Example 2

```
    $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
```

Iterating through an Array

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

```
// 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>";
    }
```

For 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

```
// 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;
    }
```

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 **$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);
```

<u>output</u>

```
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.
- 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.
- We can delete array elements using the functions **unset()**

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

The above example 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.

```
$oddKeys = array (1 => "hello", 3 => "world", 5 => "!");
if (isset($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];
}
```

Array Sorting

In PHP, there are many built-in sort functions, which sort by key or by value.

- **sort()** - sort arrays in ascending order
- **rsort()** - sort arrays in descending order
- **asort()** - sort associative arrays in ascending order, according to the value
- **ksort()** - sort associative arrays in ascending order, according to the key
- **arsort()** - sort associative arrays in descending order, according to the value
- **krsort()** - sort associative arrays in descending order, according to the key

Example1 :
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)

Example1 :
    asort($days);
The resulting array in this case is:
Array ([4] => Fri [0] => Mon [5] => Sat [6] => Sun [3] => Thu [1] => Tue [2] => Wed)

[keeps keys and values associated together]

More Array Operations

- **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.

Superglobal Arrays

1. What are the superglobal arrays in PHP?

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. They are called superglobal because 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.

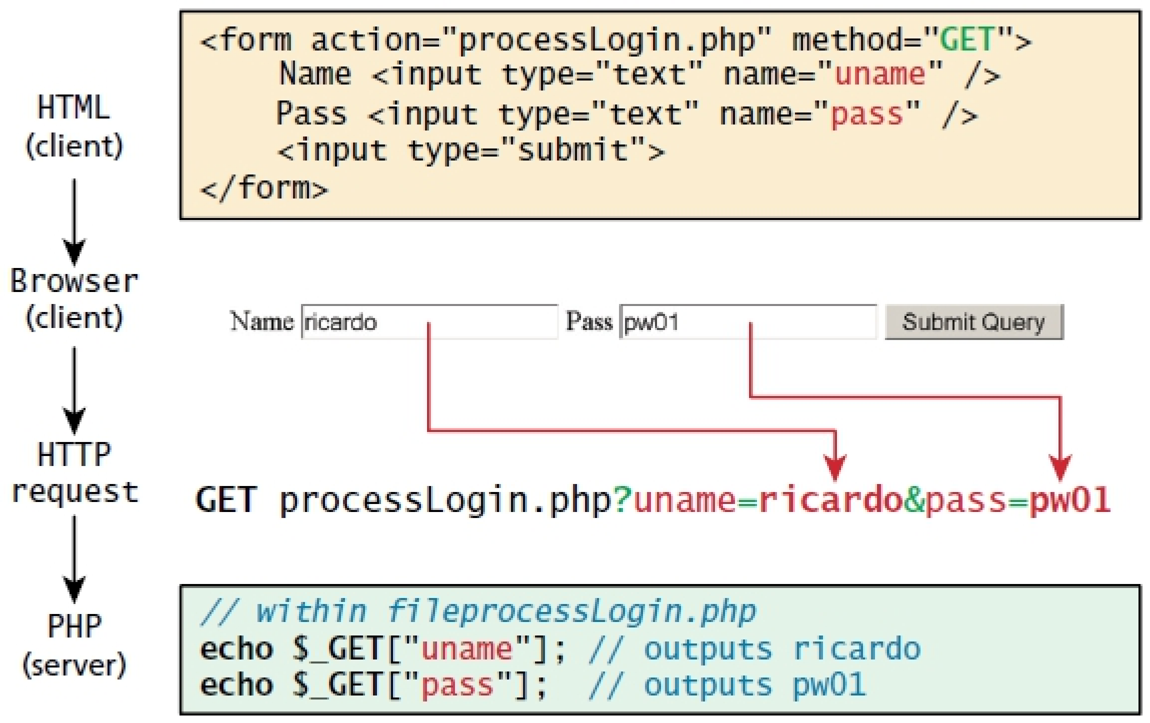
$GLOBALS Array for storing data that needs superglobal scope
$_COOKIES Array of cookie data passed to page via HTTP request
$_ENV Array of server environment data
$_FILES Array of file items uploaded to the server
$_GET Array of query string data passed to the server via the URL
$_POST Array of query string data passed to the server via the HTTP header
$_REQUEST Array containing the contents of $_GET, $_POST, and $_COOKIES
$_SESSION Array that contains session data
$_SERVER Array containing information about the request and the server.

$_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.

An HTML form 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 9.5 illustrates the relationship between an HTML form, the GET request, and the values in the $_GET array.



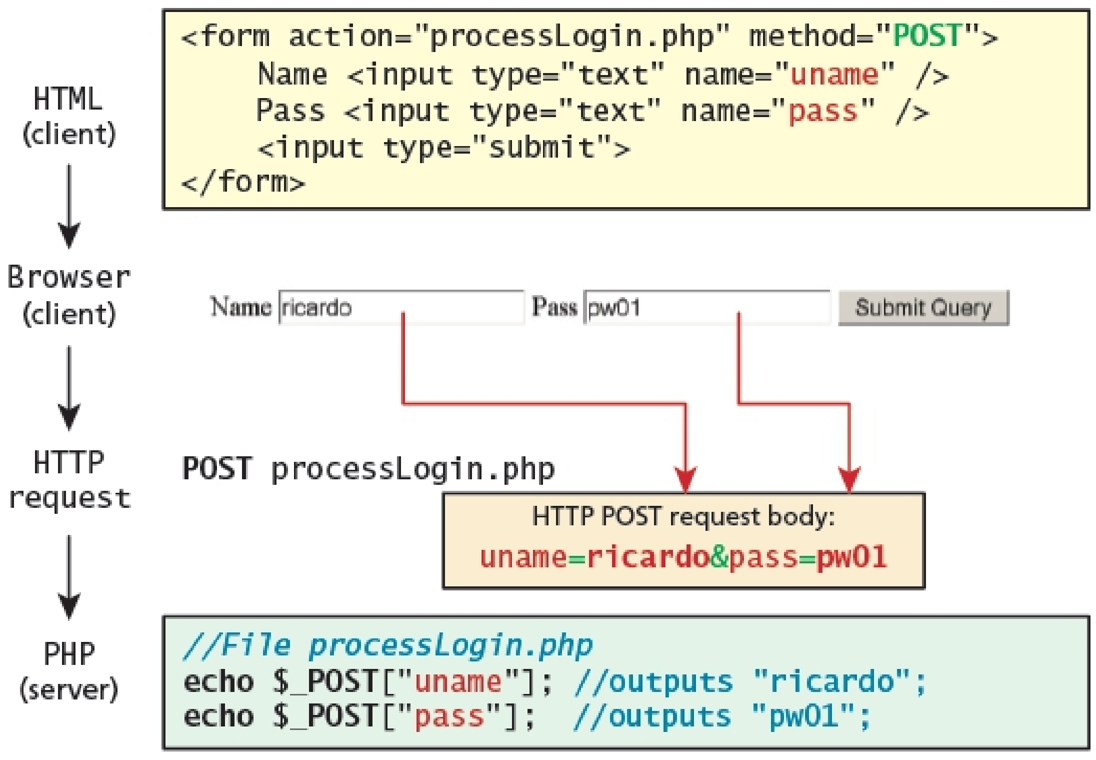
```
<form action="processLogin.php" method="GET">
    Name <input type="text" name="uname" />
    Pass <input type="text" name="pass" />
    <input type="submit">
</form>
```

```
GET processLogin.php?uname=ricardo&pass=pw01
```

```
// within fileprocessLogin.php
echo $_GET["uname"]; // outputs ricardo
echo $_GET["pass"];  // outputs pw01
```

FIGURE 9.5 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 9.6 illustrates how data from a HTML form using POST populates the $_POST array in PHP.



```
<form action="processLogin.php" method="POST">
    Name <input type="text" name="uname" />
    Pass <input type="text" name="pass" />
    <input type="submit">
</form>
```

```
//File processLogin.php
echo $_POST["uname"]; //outputs "ricardo";
echo $_POST["pass"];  //outputs "pw01";
```

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

Determining If Any Data Sent

To know whether any form data was submitted at all using either POST or GET, by checking the $_SERVER['REQUEST_METHOD'] variable. It contains as a string the type of HTTP request this script is responding to (GET, POST, HEAD, etc.).

```
<!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 .... here we could redirect or authenticate ";
echo " and hide login form or something else";
}
}
?>
<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>
```

Accessing Form Array Data

Sometimes in HTML forms you might have multiple values associated with a single name. **Example**

```
<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>
```

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.

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 . ", ";
}
?>
```

Using Query Strings in Hyperlinks

Sanitizing Query Strings

$_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, many configuration options for PHP itself, paths, and script locations, as shown in Fig

below.

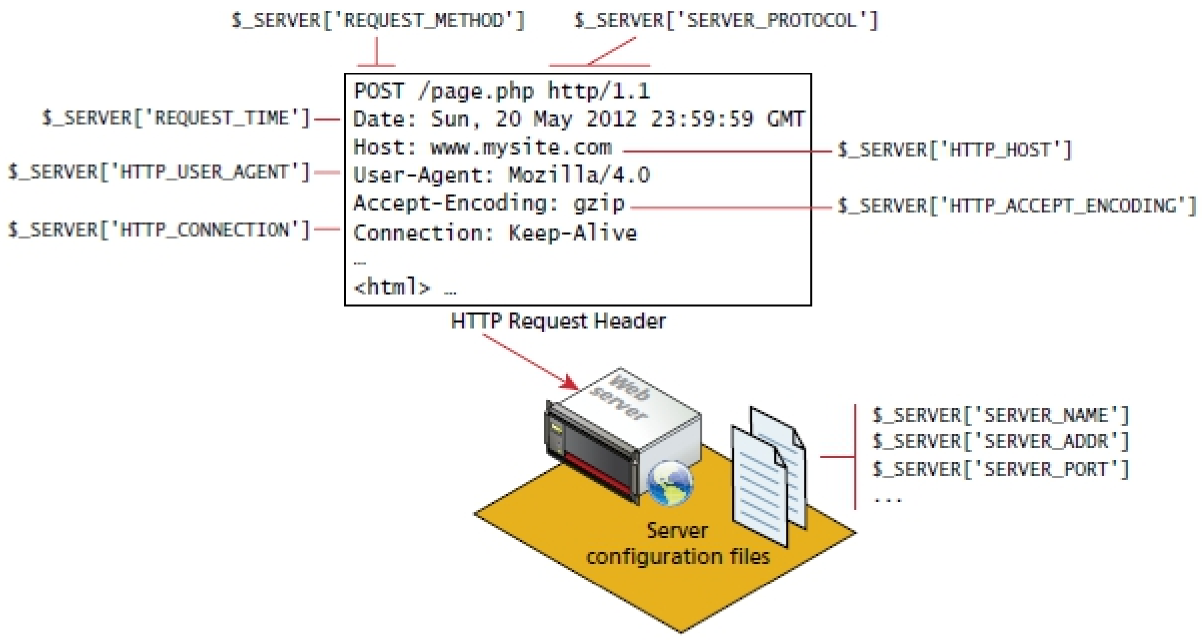


FIGURE 9.11 Relationship between request headers, the server, and the $_SERVER array

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

```
<?php
    echo $_SERVER["SERVER_NAME"] . "<br/>";
    echo $_SERVER["SERVER_SOFTWARE"] . "<br/>";
    echo $_SERVER["REMOTE_ADDR"] . "<br/>";
    echo                        $_SERVER['PHP_SELF'];.                    "<br>";
    echo                                          $_SERVER['HTTP_HOST'];."<br>";
    echo                                       $_SERVER['HTTP_REFERER'];."<br>";
    echo                                   $_SERVER['HTTP_USER_AGENT'];."<br>";
    echo                                               $_SERVER['SCRIPT_NAME'];
?>
```

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.
SCRIPT_NAME key that identifies the actual script being executed.

Request Header Information Keys

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.
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. Example 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);
?>
```

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.

$_FILES Array

The **$_FILES** associative array contains items that have been uploaded to the current script. 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.

HTML Required for File Uploads
To allow users to upload files, there are some specific things you must do:

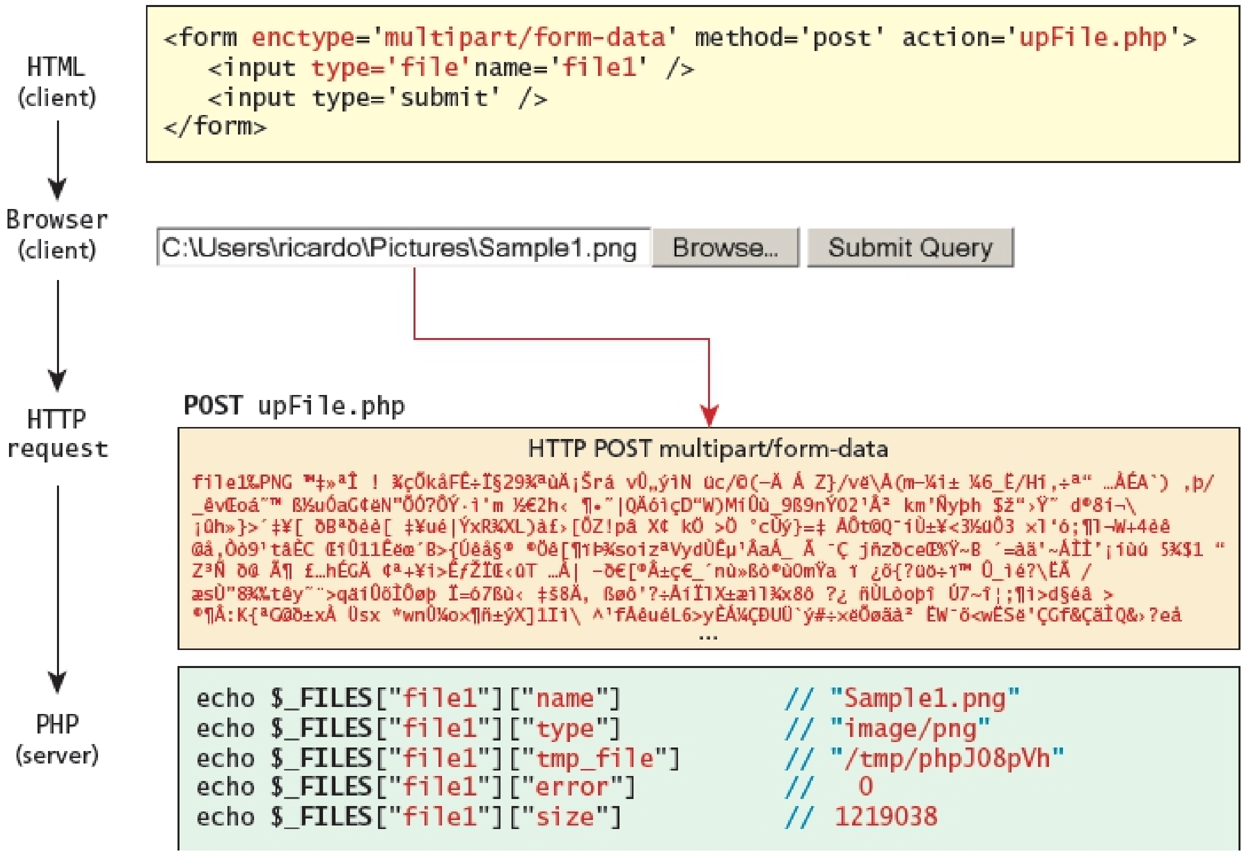
- First, you must ensure that the HTML form uses the HTTP POST method, since transmitting a file through the URL is not possible.
- Second, you must add the enctype="multipart/form-data" attribute to the HTML form that is performing the upload so that the HTTP request can submit multiple pieces of data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

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

<input type='submit' />
</form>

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.
- 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.
    - **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.



```
<form enctype='multipart/form-data' method='post' action='upFile.php'>
   <input type='file'name='file1' />
   <input type='submit' />
</form>
```

```
echo $_FILES["file1"]["name"]           // "Sample1.png"
echo $_FILES["file1"]["type"]           // "image/png"
echo $_FILES["file1"]["tmp_file"]       // "/tmp/phpJ08pVh"
echo $_FILES["file1"]["error"]          //   0
echo $_FILES["file1"]["size"]           // 1219038
```

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

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 9.2 and shows that there are many causes for bad file uploads.

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_DIR	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 9.2 Error Codes in PHP for File Upload Taken from php.net.[6]

File Size Restrictions 2. List and briefly describe the 3 ways you can limit the types and size of file uploaded.

- Some scripts limit the file size of each upload.
- 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.**

1. 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. 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>
```

2. 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 below

```
<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>
```

3. The third (and essential) mechanism for limiting the uploaded file size is to add a simple check on the server side .This technique checks the file size on the server by simply checking the size field in the $_FILES array. Limiting upload file size via PHP

```
$max_file_size = 10000000;
foreach($_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);
}
```

Limiting the Type of File Upload

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 9.17 PHP code to look for valid mime types and file extensions

Moving the File

To move the temporary file to a permanent location on your server, use 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.

```
$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 9.18 Using move_uploaded_file() function

5. Demonstrate reading and writing of file in PHP with suitable example

Reading/Writing Files

In web development, the ability to read and write to text files remains an important technical competency.
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.

Stream Access

fopen()

- 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.

fgets()

- 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.

fread()

- To read an arbitrary amount of data (typically for binary files), use fread() and for reading a single character use **fgetsc().** Finally, when finished processing the file you must close it using **fclose().**

fwrite()

- To write data to a file, the fwrite() function is used by passing the file handle and the string to write.

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

LISTING 9.19 Opening, reading lines, and closing a file

In-Memory File Access

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

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 $writeme to a file, you use

file_put_contents(FILENAME, $writeme);

PHP CLASSES AND OBJECTS

OBJECT-ORIENTED OVERVIEW

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.

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.

The Unified Modeling Language

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.

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 10.2 illustrates a UML class diagram, which shows an Artist class and multiple Artist objects, each object having its own properties.

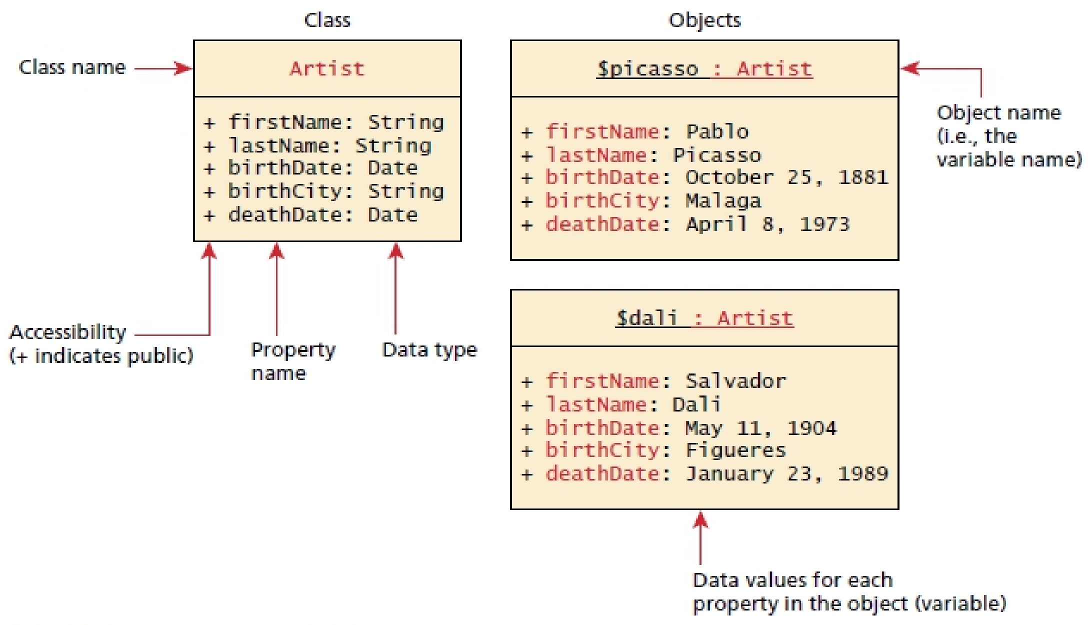


FIGURE 10.2 Relationship between a class and its objects in UML

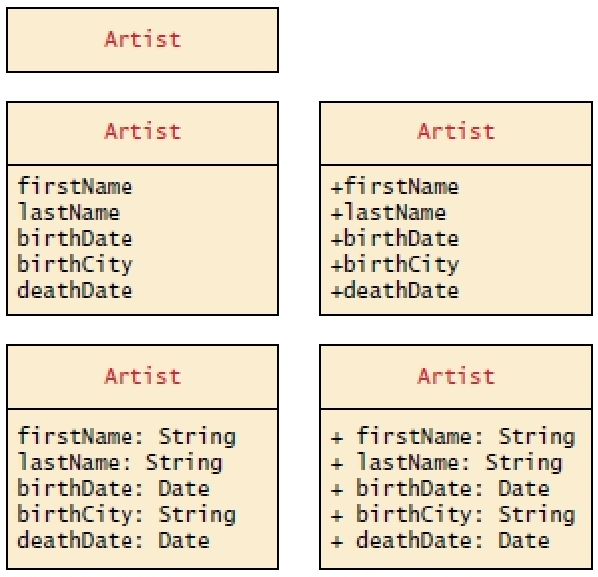


FIGURE 10.3 Different levels of UML detail

Differences between Server and Desktop Objects

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 10.4 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 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. However, serialization does not address the inherent inefficiency of recreating objects each time a new request comes in.

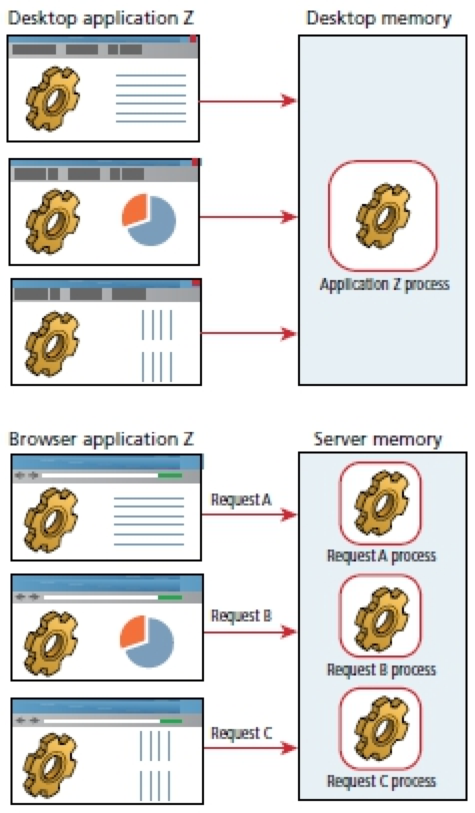


FIGURE 10.4 Lifetime of objects in memory in web versus desktop applications

CLASSES AND OBJECTS IN PHP

Classes should be defined in their own files so they can be imported into multiple scripts. We can 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 like include, include_once, require, or require_once.
Once a class has been defined, you can create as many instances of that object as memory will allow using the new keyword.

Defining Classes

The PHP syntax for defining a class uses the class keyword followed by the class name and { } braces. The properties and methods of the class are defined within the braces. Each property in the class is declared using one of the keywords public, protected, or private followed by the property or variable name.
Example: The Artist class with the properties illustrated below

```
class Artist {
        public $firstName;
```

```
            public $lastName;
            public $birthDate;
            public $birthCity;
            public $deathDate;
        }
```

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();
```

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.

```
$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";
```

Constructors

- Inside of a class definition, define constructors, which lets you specify parameters during instantiation to initialize the properties within a class.
- In PHP, constructors are defined as functions with the name __construct(). (Note: there are *two* underscores_ before the word construct.
- 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.

Example: an updated Artist class definition with a constructor.

```
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;
}
}
```

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 object

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

Methods

Objects only really become useful when you define behavior or operations that they can perform. In object-oriented operations are called methods. They define the tasks each instance of a class can perform and are useful since they associate behavior with objects.

Example: For artist example to write a method to convert the artist's details into a string of formatted HTML.

```
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;
        }
}
```

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

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

Class diagram updated Artist class

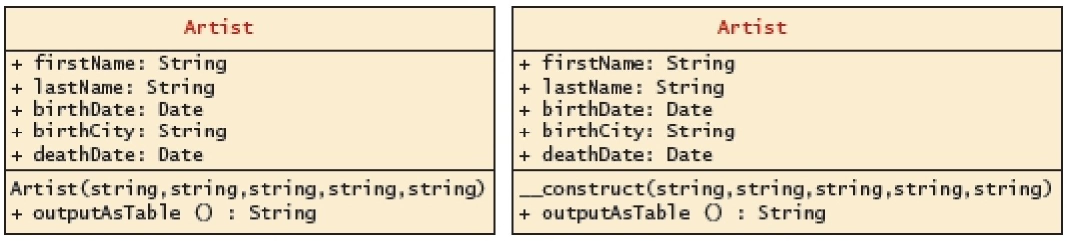


FIGURE 10.5 Updated class diagram

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.

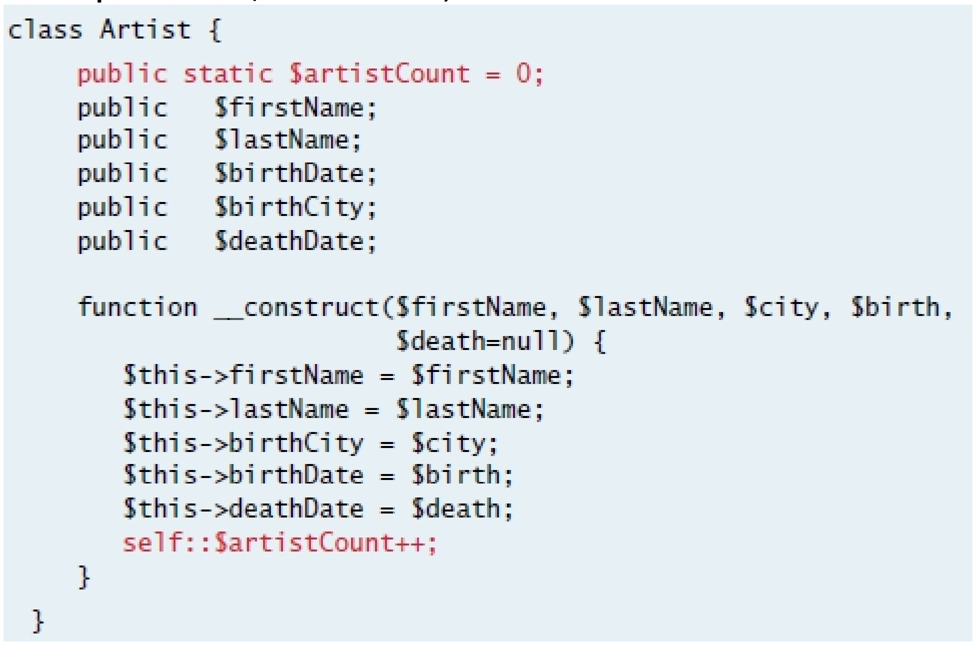
- The **public** keyword means that the property or method is accessible to any code that has a reference to the object.
- The private keyword sets a method or variable to only be accessible from within the class. This means that we cannot access or modify the property from outside of the class, even if we have a reference to it as shown in Figure 10.6.
- The protected keyword, member will be available to subclasses but not anywhere else.

- In UML, the "+" symbol is used to denote public properties and methods, the "–" symbol for private ones, and the "#" symbol for protected ones.

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, the static property artistCount to 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;



```php
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++;
    }
 }
```

We do not reference a static property using the $this-> syntax, but rather it has its own self:: syntax. This static variable can also be accessed without any instance of an Artist object by using the class name, that is, via

Artist::$artistCount.

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.

11. How do you achieve data encapsulation in PHP? Give example

OBJECT-ORIENTED DESIGN

Data Encapsulation

Encapsulation generally refers to restricting access to an object's internal components. That is, it is the hiding of an object's implementation details. The most important advantage to object-oriented design.

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.
The typical approach to access private properties 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).

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

```
public function setBirthDate($birthdate){
                $this->birthDate = $date;
                }
```

Using getters and setters functions we can have a fair bit of validation logic in them and thus class can handle the responsibility of ensuring its own data validation.
Example:

```
class Artist {
        …………
        function __construct($firstName, $lastName, $birthCity, $birthDate,$deathDate) {
                $this->setFirstName($firstName);
                $this->setLastName($lastName);
                $this->setBirthCity($birthCity);
                $this->setBirthDate($birthDate);
                $this->setDeathDate($deathDate);
                self::$artistCount++;
        }

        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 function setLastName($lastName)
```

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

To demonstrates how the Artist class could be used.

```
<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");
        // 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>';
?>
</body>
</html>
```

10. Define constructor and discuss the concepts of inheritence,polymorphism and object interface with respect to OOP.

Inheritance

- **Inheritance** is one of the three key concepts in object oriented design and programming.
- Inheritance enables you to create new PHP classes that reuse, extend, and modify the behavior that is defined in another PHP class.
- Although 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.
- A PHP class is defined as a subclass by using the extends keyword.

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

Referencing Base Class Members

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

- 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*

UML diagram

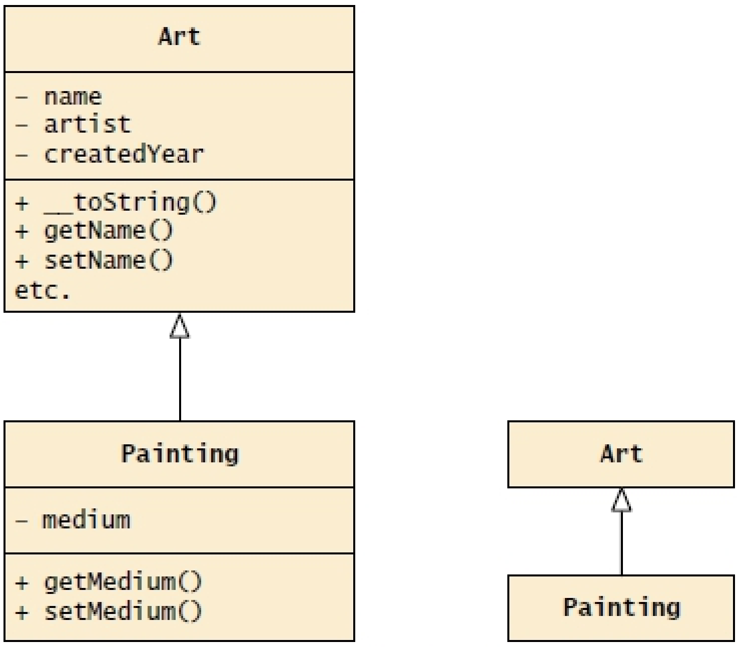


FIGURE 10.9 UML class diagrams showing inheritance

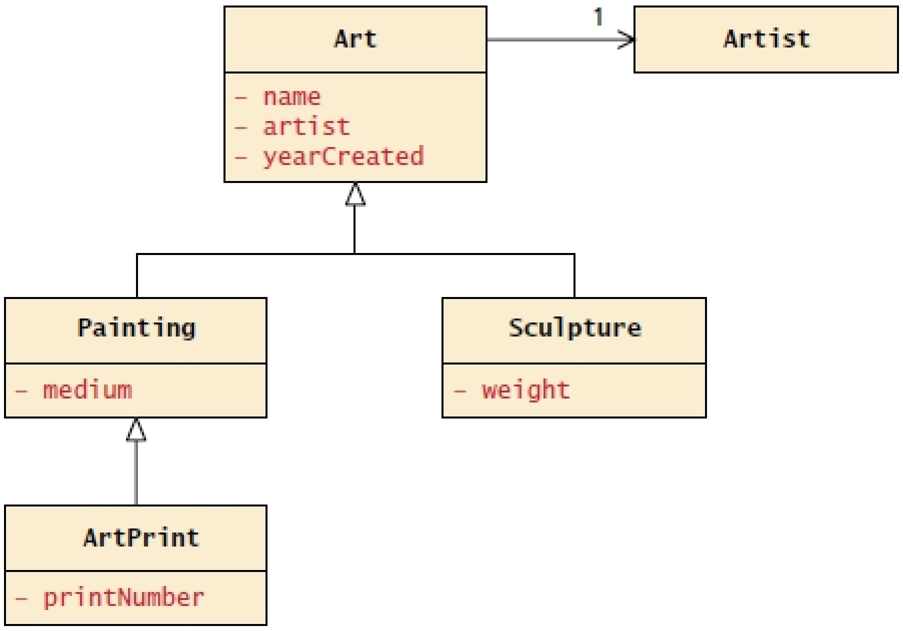


FIGURE 10.11 Class diagram for Art example

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 10.8 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.

Polymorphism

- Polymorphism is the third key object-oriented concept. In the inheritance example, 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.

```
$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);
// 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
}
```

- 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.
- In the above example 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.

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.
- 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.
  Example

```
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.
- 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.
- Example below 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
      ...
   }
}
```

```
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
      ...
   }
}
```

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 object implements.

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);
```

4. Illustrate how Error Handling and Validation is done in PHP with suitable example

ERROR HANDLING AND VALIDATION

What Are Errors and Exceptions?

Even the best-written web application can suffer from runtime errors. Most complex web applications must interact with external systems such as databases, web services, RSS feeds, email servers, file system, and other externalities that are beyond the developer's control. A failure in any one of these systems will mean that the web application will no longer run successfully. It is vitally important that web applications gracefully handle such problems.

Types of Errors

Not every problem is unexpected or catastrophic. One might say that there are three different types of website problems:

- **Expected errors**
- **Warnings**
- **Fatal errors**

Expected errors

An **expected error** is an error that routinely occurs during an application. Example of this type would be an error as a result of user inputs, for instance, entering letters when numbers were expected. Expect the user to not always enter expected values. Users will leave fields blank, enter text when numbers were expected (and vice versa), type in too much or too little text, forget to click certain things, and click things they should not. Your PHP code should always check user inputs for acceptable values. Not every expected error is the result of user input.

Web applications that rely on connections to externalities such as database management systems, legacy software systems, or web services should be expected to occasionally fail to connect.

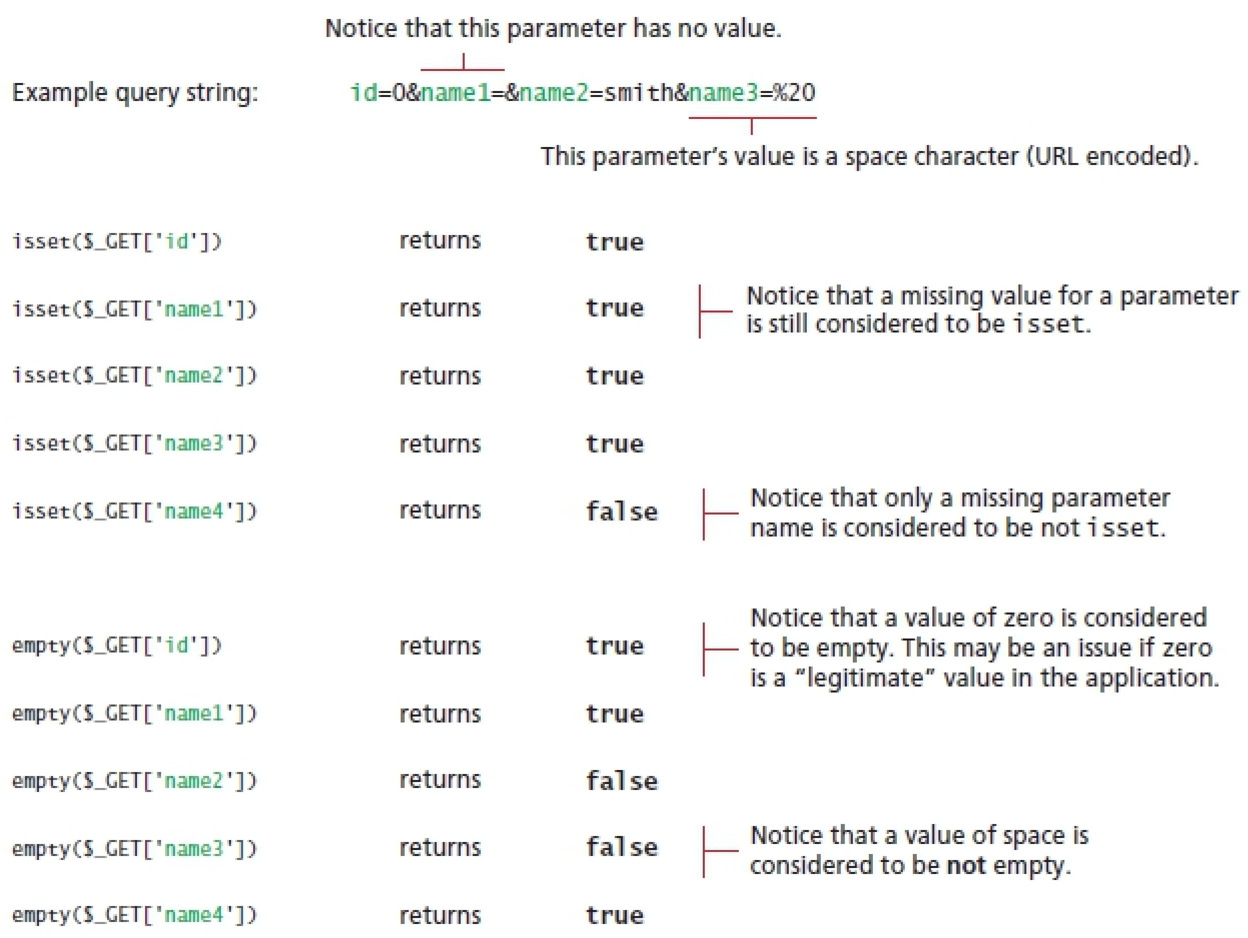
PHP provides two functions for testing the value of a variable.

isset(): which returns true if a variable is not null. However, isset() by itself does not provide enough error checking.

empty() : which returns true if a variable is null, false, zero, or an empty string.

is_numeric(): If you are expecting a query string parameter to be numeric, then you can use is_numeric() the function.

```
$id = $_GET['id'];
if (!empty($id) && is_numeric($id) ) {
// use the query string since it exists and is a numeric value
...
}
```



```
Example query string:     id=0&name1=&name2=smith&name3=%20

isset($_GET['id'])        returns     true
isset($_GET['name1'])     returns     true
isset($_GET['name2'])     returns     true
isset($_GET['name3'])     returns     true
isset($_GET['name4'])     returns     false

empty($_GET['id'])        returns     true
empty($_GET['name1'])     returns     true
empty($_GET['name2'])     returns     false
empty($_GET['name3'])     returns     false
empty($_GET['name4'])     returns     true
```

Notice that this parameter has no value.
This parameter's value is a space character (URL encoded).
Notice that a missing value for a parameter is still considered to be isset.
Notice that only a missing parameter name is considered to be not isset.
Notice that a value of zero is considered to be empty. This may be an issue if zero is a "legitimate" value in the application.
Notice that a value of space is considered to be not empty.

FIGURE 12.1 Comparing isset() and empty() with query string parameters

Warnings

Another type of error is **warnings**, which are problems that generate a PHP warning message (which may or may not be displayed) but will not halt the execution of the page. For instance, calling a function without a required parameter will generate a warning message but not stop execution. While not as serious as expected errors, these types of incidental errors should be eliminated by the programmer, since they harbor the potential for bugs. However, if warning messages are not being displayed (which is a common setup), then these warnings may escape notice, and hence require special strategies to ensure the developers are aware of them.

Fatal errors

The final type of error is **fatal errors**, which are serious in that the execution of the page will terminate unless handled in some way. These should truly be exceptional and unexpected, such as a required input file being missing or a database table or field disappearing. These types of errors not only need to be reported so that the developer can try to fix the problem, but also the page needs to recover gracefully from the error so that the user is not excessively puzzled or frustrated.

Exceptions

Developers sometimes treat the words "error" and "exception" as synonyms. In the context of PHP, they do have different meanings. An **error** is some type of problem that generates a nonfatal warning message or that generates an error message that terminates the program's execution. An **exception** refers to objects that are of type Exception and which are used in conjunction with the object-oriented try . . . catch language construct for dealing with runtime errors.

PHP Error Reporting

PHP has a flexible and customizable system for reporting warnings and errors that can be set programmatically at runtime or declaratively at design-time within the php.ini file. There are three main error reporting flags:

- **error_reporting**
- **display_errors**
- **log_errors**

The error_reporting Setting

The error_reporting setting specifies which type of errors are to be reported. It can be set programmatically inside any PHP file by using the error_reporting() function:

```
error_reporting(E_ALL);
```

It can also be set within the php.ini file:

```
error_reporting = E_ALL
```

The possible levels for error_reporting are defined by predefined constants; In some PHP environments, the default setting is zero, that is, no reporting.

Constant Name	Value	Description
E_ALL	8191	Report all errors and warnings
E_ERROR	1	Report all fatal runtime errors
E_WARNING	2	Report all nonfatal runtime errors (i.e., warnings)
	0	No reporting

TABLE 12.1 Some error_reporting Constants

The display_errors Setting

The display_error setting specifies whether error messages should or should not be displayed in the browser. It can be set programmatically via the ini_set() function:

```
ini_set('display_errors','0');
```

It can also be set within the php.ini file:

```
display_errors = Off
```

The log_error Setting

The log_error setting specifies whether error messages should or should not be sent to the server error log. It can be set programmatically via the ini_set() function:

```
ini_set('log_errors','1');
```

It can also be set within the php.ini file:

```
log_errors = On
```

When logging is turned on, error reporting will be sent to either the operating system's error log file or to a specified file in the site's directory. The server log file option will not normally be available in shared hosting environments.
If saving error messages to a log file in the site's directory, the file name and path can be set via the error_log setting (which is not to be confused with the log_error setting) programmatically:

```
ini_set('error_log', '/restricted/my-errors.log');
```

It can also be set within the php.ini file:

```
error_log = /restricted/my-errors.log
```

You can also programmatically send messages to the error log at any time via the error_log() function.

```
$msg = 'Some horrible error has occurred!';
// send message to system error log (default)
error_log($msg,0);
// email message
error_log($msg,1,'support@abc.com','From: somepage.php@abc.com');
// send message to file
error_log($msg,3, '/folder/somefile.log');
```

PHP Error and Exception Handling

When a fatal PHP error occurs, program execution will eventually terminate unless it is handled. The PHP documentation provides two mechanisms for handling runtime errors:

- procedural error handling
- object-oriented exception handling.

Procedural Error Handling

In the procedural approach to error handling, the programmer needs to explicitly test for error conditions after performing a task that might generate an error. For instance,to use procedural mysqli approach for accessing a database. In such a case you needed to test for and deal with errors after each operation that might generate an error state, as shown below.

```
$connection = mysqli_connect(DBHOST, DBUSER, DBPASS, DBNAME);
$error = mysqli_connect_error();
if ($error != null) {
// handle the error
...
}
```

Object-Oriented Exception Handling

When a runtime error occurs, PHP *throws* an *exception*. This exception can be *caught* and handled either by the function, class, or page that generated the exception or by the code that called the function or class. If an exception is not caught, then eventually the PHP environment will handle it by terminating execution with an "Uncaught Exception" message.

PHP uses the try . . . catch programming construct to programmatically deal with exceptions at runtime. Example below illustrates a sample example of a try . . . catch block. Notice that the catch construct expects some type of parameter of type Exception (or a subclass of Exception). The Exception class provides methods for accessing not only the exception message, but also the line number of the code that generated the exception and the stack trace, both of which can be helpful for understanding where and when the exception occurred.

```
// Exception throwing function
function throwException($message = null,$code = null) {
  throw new Exception($message,$code);
}

try {
  // PHP code here
  $connection = mysqli_connect(DBHOST, DBUSER, DBPASS, DBNAME)
    or throwException("error");
 //...
}
catch (Exception $e) {
  echo ' Caught exception: ' .  $e->getMessage();
  echo ' On Line : ' .  $e->getLine();
  echo ' Stack Trace: '; print_r($e->getTrace());
} finally {
  // PHP code here that will be executed after try or after catch
}
```

LISTING 12.3 Example of try . . . catch block

3. Describe why hidden form fields can easily be forged/changed by an end user

12. Write a PHP program to create a class STUDENT with the following specification.
Data members : Name, Roll number, Average marks
Member function : Read(getters) and write (setters)
1) Use the above specification to read and print the information of 2 students