PHP Arrays and Superglobals

9.1 Arrays

- Figure 9.1 illustrates a PHP array with five strings containing day abbreviations.

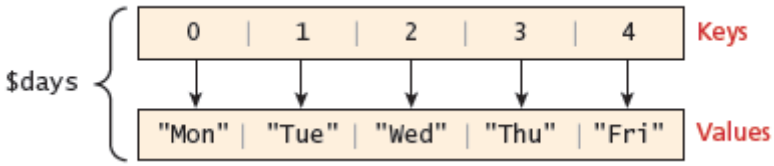


Figure 9.1 Visualization of a key-value array

- **Array keys** start at 0, and go up by 1.
- In PHP, keys *must* be either integers or strings and need not be sequential.
- **Array values**, unlike keys, are not restricted to integers and strings. They can be any object, type, or primitive supported in PHP.

9.1.1 Defining and Accessing 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 as shown in Figure 9.1, we 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
```

- 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
```

- We 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";
```

- we can also define explicitly by specifying the keys and values as shown in Figure 9.2.

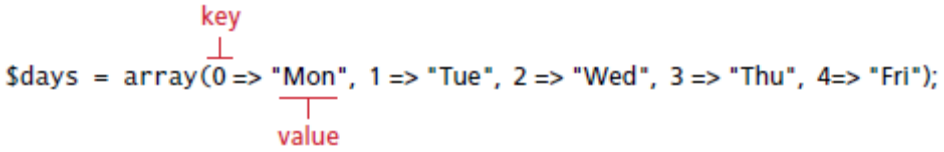
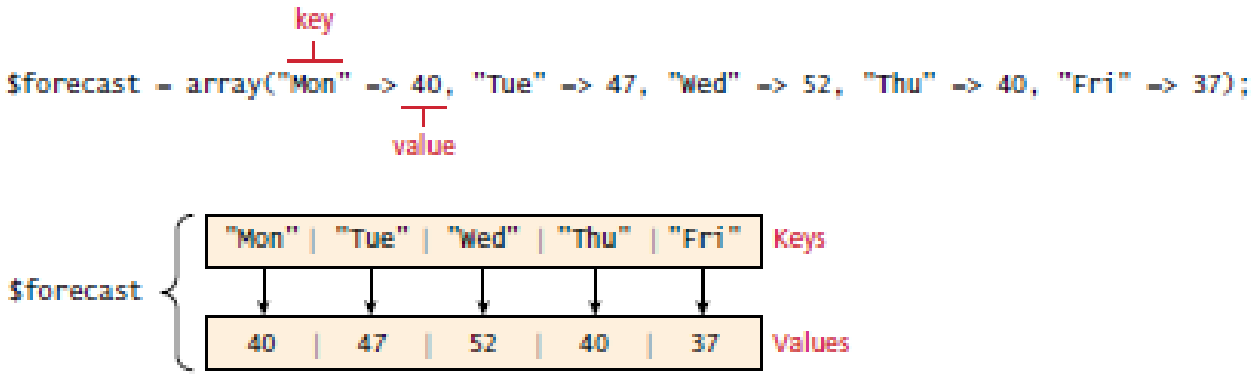


Figure 9.2 Explicitly assigning keys to array elements

- we can consider an array to be a dictionary or hash map. These types of arrays in PHP are referred to as **associative arrays**.
- We can see in Figure 9.3 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.




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

Figure 9.3 Array with strings as keys and integers as values

- As can be seen in Figure 9.3, to access an element in an associative array, we simply use the key value rather than an index:

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

9.1.2 Multidimensional Arrays

- Listing 9.1 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
```

```
$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 9.1 Multidimensional arrays

- Figure 9.4 illustrates the structure of these two multidimensional arrays.

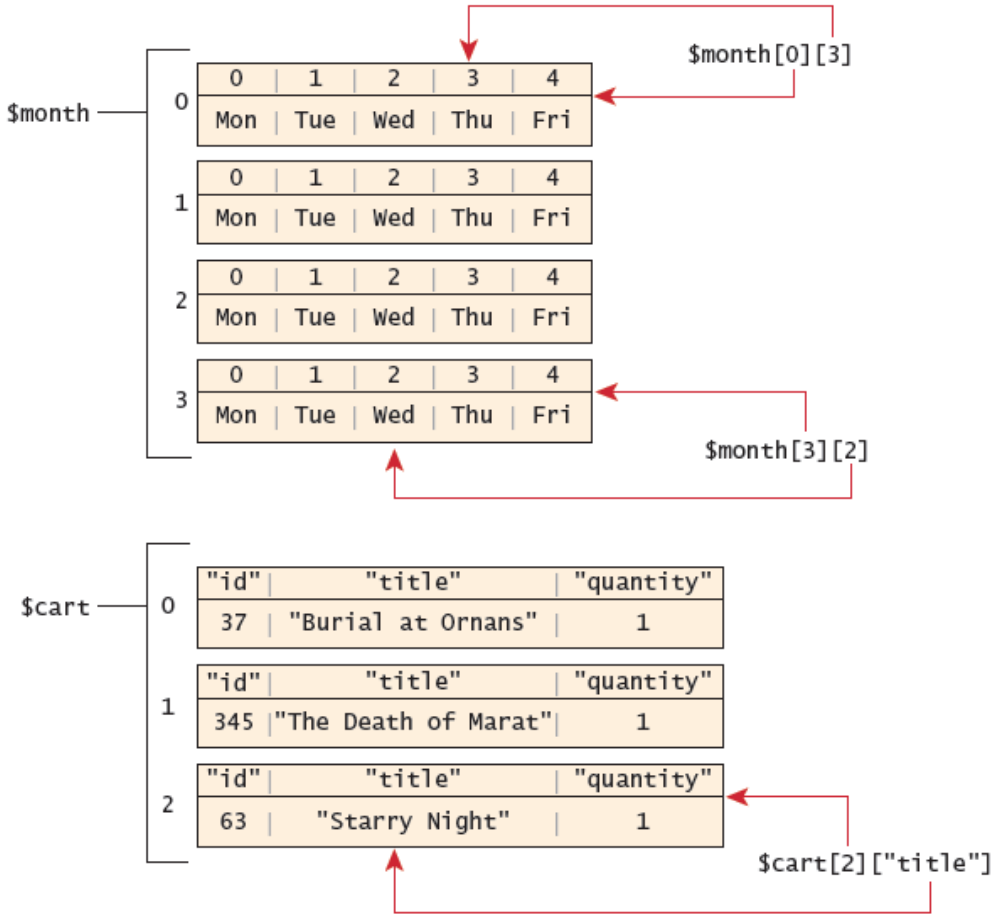


Figure 9.4 Visualizing multidimensional arrays

9.1.3 Iterating through an Array

- Listing 9.2 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>";
}
```

listing 9.2 Iterating through an array using while, do while, and for loops

- Iterating using foreach loop and illustrated for the $forecast array in Listing 9.3.

```
// 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 9.3 Iterating through an associative array using a foreach loop

9.1.4 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";
```

- 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 we "skip" an index, as shown in the following example.

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

- It will show that our array now contains the following:

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

Checking If a Value Exists

- To check if a value exists for a key, we can therefore use the isset() function, which returns true if a value has been set, and false otherwise.
- Listing 9.5 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 9.5 Illustrating nonsequential keys and usage of isset( )

9.1.5 Array Sorting

- There are many built-in sort functions, which sort by key or by value.
- To sort the $days array by its values we 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)
```

- 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)
```

9.1.6 More Array Operations

- a brief description of some key array functions:

1. **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)
2. **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)
3. **array_rand($someArray, $num=1)**: Often in games or widgets we want to select a random element in an array. This function returns as many random keys as are requested. If we only want one, the key itself is returned; otherwise, an array of keys is returned.
   For example, `print_r(array_rand($days,2))` might output:
   Array (3, 0)
4. **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)

5. **array_walk($someArray, $callback, $optionalParam)**: This method is extremely powerful. It allows we to call a method ($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;
}
```

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

9.1.7 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 9.1).

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
$_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

table 9.1 Sugerglobal Variables

9.2 $_GET and $_POST Superglobal Arrays

- 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.

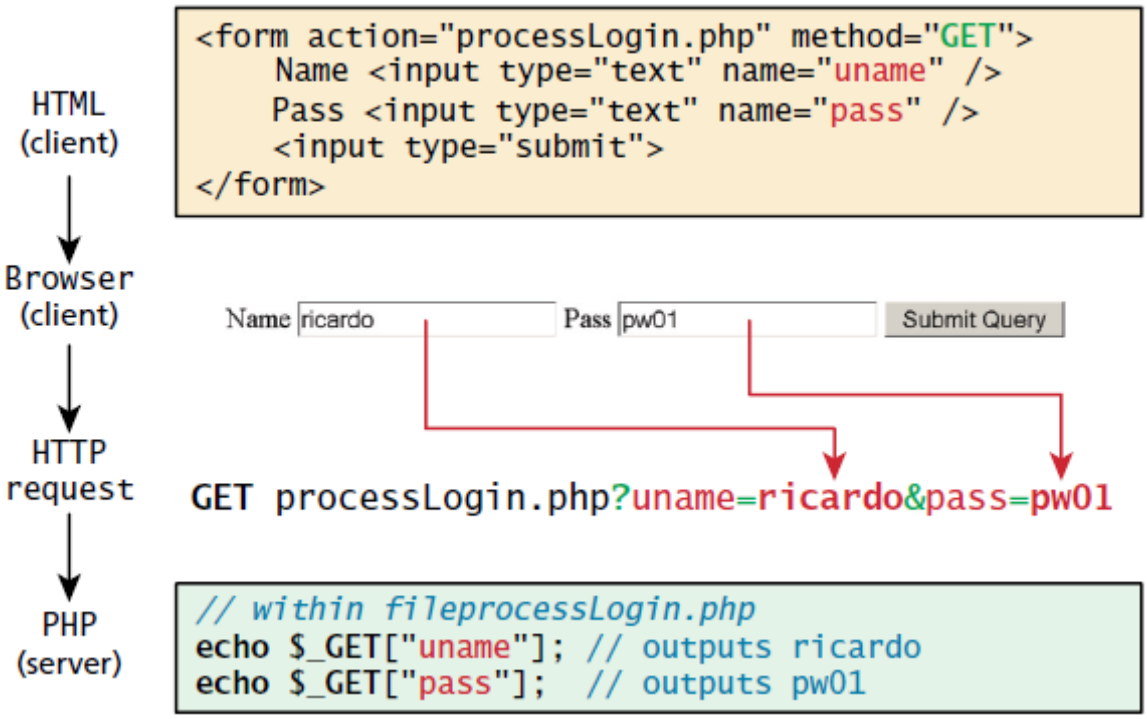


Figure 9.5 Illustration of flow from HTML, to request, to PHP's $_GET array

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

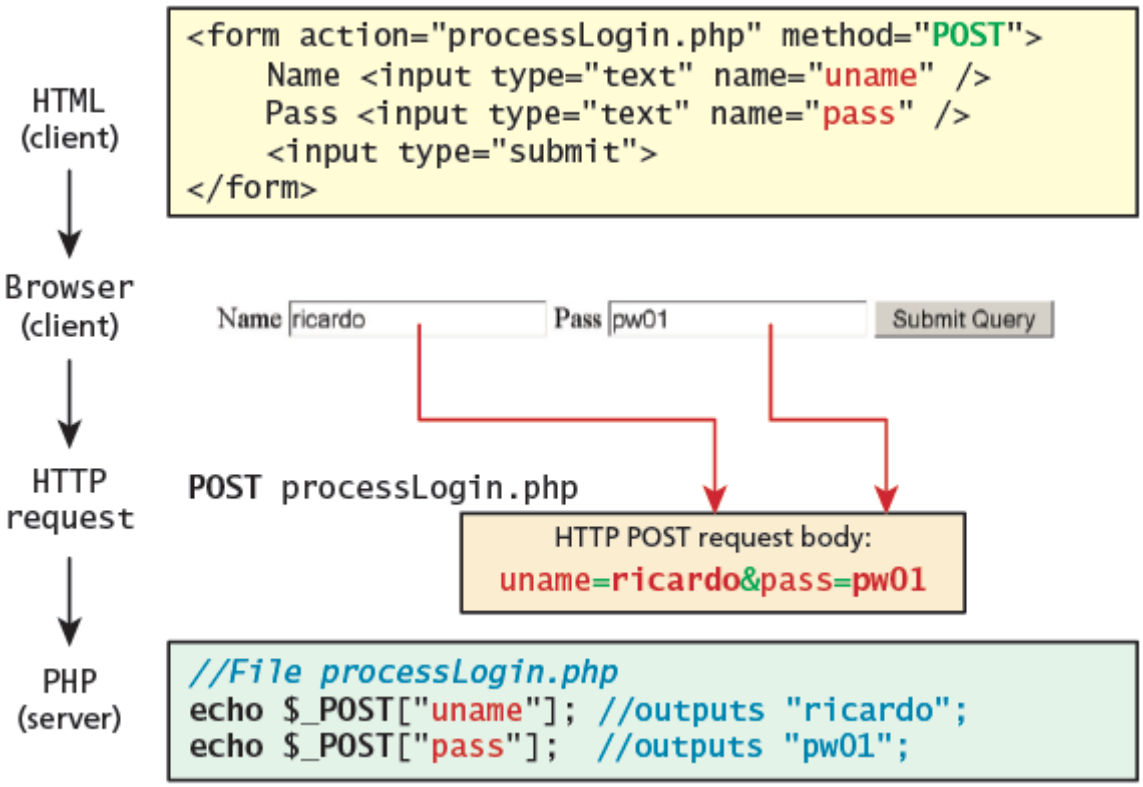


Figure 9.6 Data flow from HTML form through HTTP request to PHP's $_POST array

9.2.1 Determining If Any Data Sent

- a single file is often used to display a login form to the user, and that same file also handles the processing of the submitted form data, as shown in Figure 9.8.

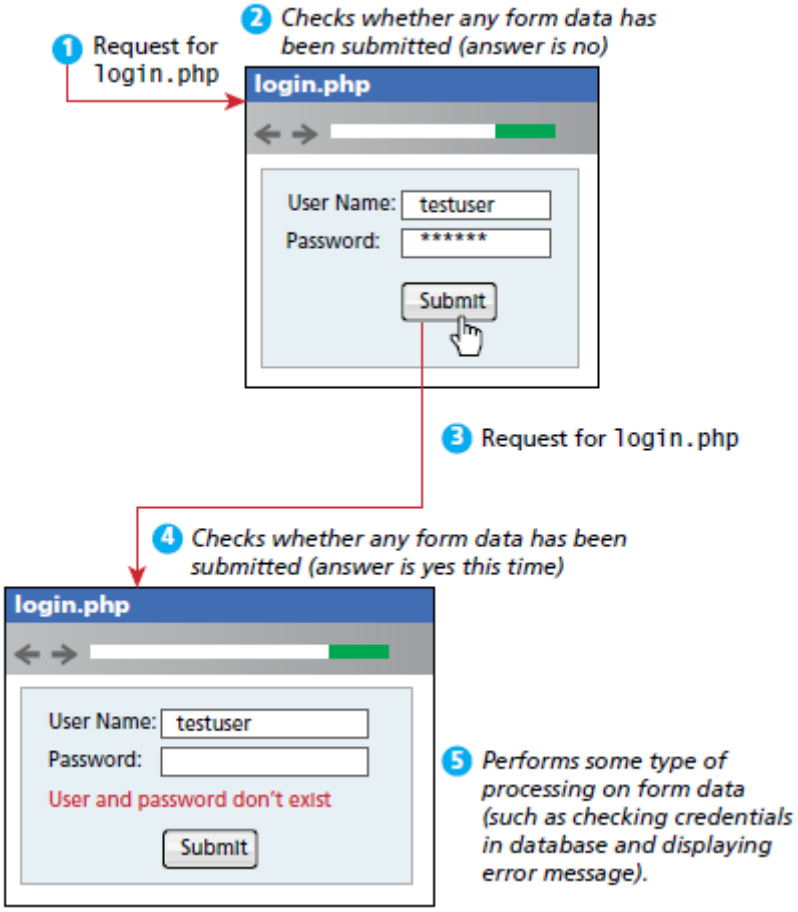


Figure 9.8 Form display and processing by the same PHP page

- In such cases we may want to know whether any form data was submitted at all using either POST or GET.
- First, we can determine if we are responding to a POST or GET by checking the $_SERVER['REQUEST_METHOD']. It contains as a string the type of HTTP request this script is responding to (GET, POST, HEAD, etc.).
- we can use the isset() function in PHP to see if there is anything set for a particular query string parameter, as shown in Listing 9.6.

```
<!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";
   }
}
```

```
?>
<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 9.6 Using isset() to check query string data

9.2.2 Accessing Form Array Data

- Listing 9.7 provides example for form data. In this 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.

```
<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 9.7 HTML that enables multiple values for one name

- To overcome this limitation, we 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, we can process the output as shown in Listing 9.8 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 9.8 PHP code to display an array of checkbox variables

9.2.3 Using Query Strings in Hyperlinks

- 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 possible to combine query strings with anchor tags (i.e., hyperlinks also use the HTTP GET method).

9.2.4 Sanitizing Query Strings

- The process of checking user input for incorrect or missing information is sometimes referred to as the process of **sanitizing user inputs**.
- we do these types of validation checks? It will require programming similar to that shown in Listing 9.9.

```
// 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 9.9 Simple sanitization of query string values

9.3 $_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 9.11.

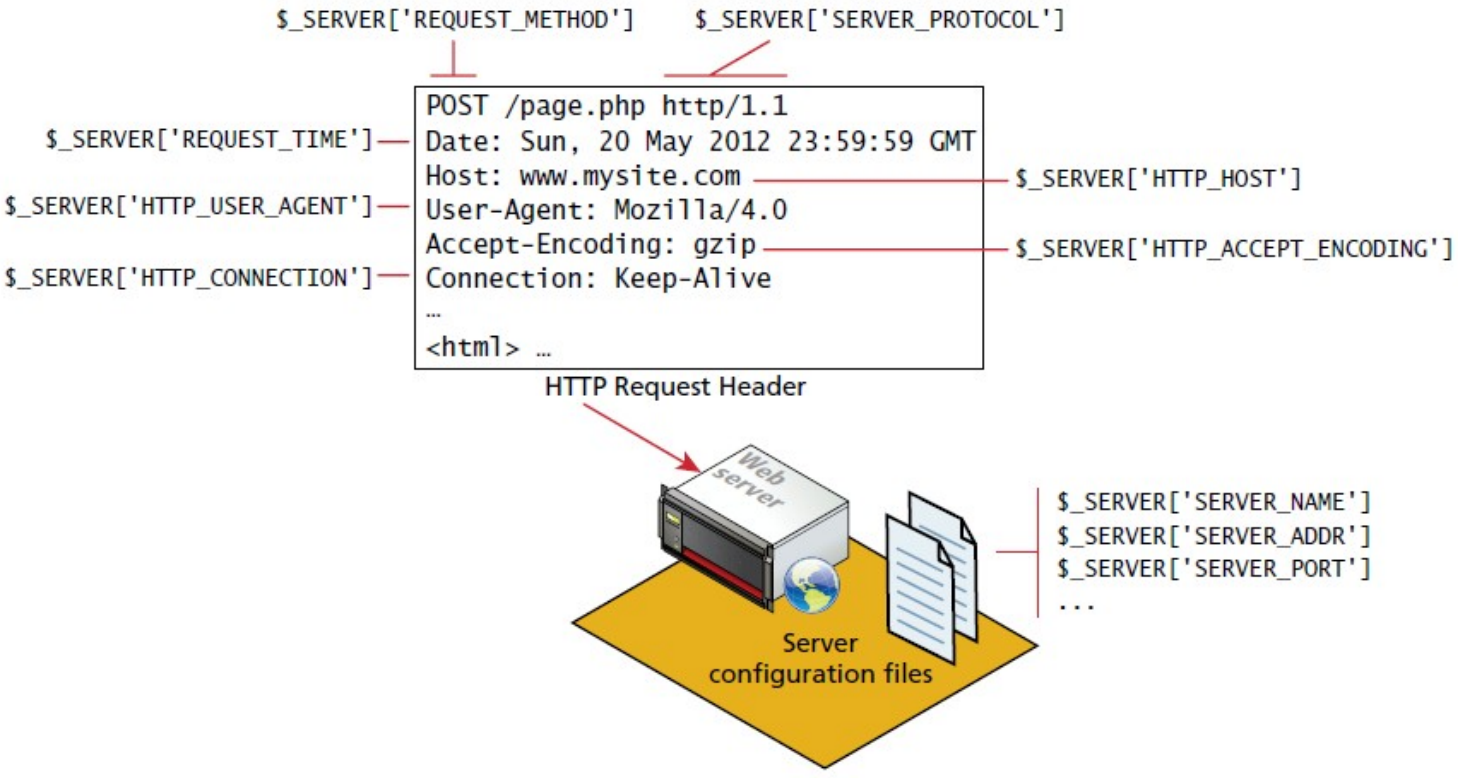


Figure 9.11 Relationship between request headers, the server, and the $_SERVER array

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

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

9.3.1 Server Information Keys

- SERVER_NAME is a key in the $_SERVER array that contains the name of the site that was requested.
- SERVER_ADDR is a complementary key telling us the IP of the server.

- DOCUMENT_ROOT tells us the file location from which we are currently running our script.

9.3.2 Request Header Information Keys

- 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 our web applications.
- The HTTP_USER_AGENT key returns the operating system and browser that the client is using.
- Listing 9.10 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 9.10 Accessing the user-agent string in the HTTP headers

- HTTP_REFERER contains the address of the page that referred us to this one (if any) through a link.
- Listing 9.11 shows an example of context-dependent output.

```
$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 9.11 Using the HTTP_REFERER header to provide context-dependent output.

9.4 $_FILES Array

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

9.4.1 HTML Required for File Uploads

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

1. First, we must ensure that the HTML form uses the HTTP POST method, since transmitting a file through the URL is not possible.
2. Second, we 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).
3. Finally we must include an input type of file in our 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 9.12.

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

listing 9.12 HTML for a form that allows an upload

9.4.2 Handling the File Upload in PHP

- PHP file responsible for handling the upload will utilize the superglobal $_FILES array.
- Figure 9.12 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.

1. **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.
2. **type** defines the MIME type of the file. This value is provided by the client browser and is therefore not a reliable field.
3. **tmp_name** is the full path to the location on our 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.
4. **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.
5. **size** is an integer representing the size in bytes of the uploaded file.

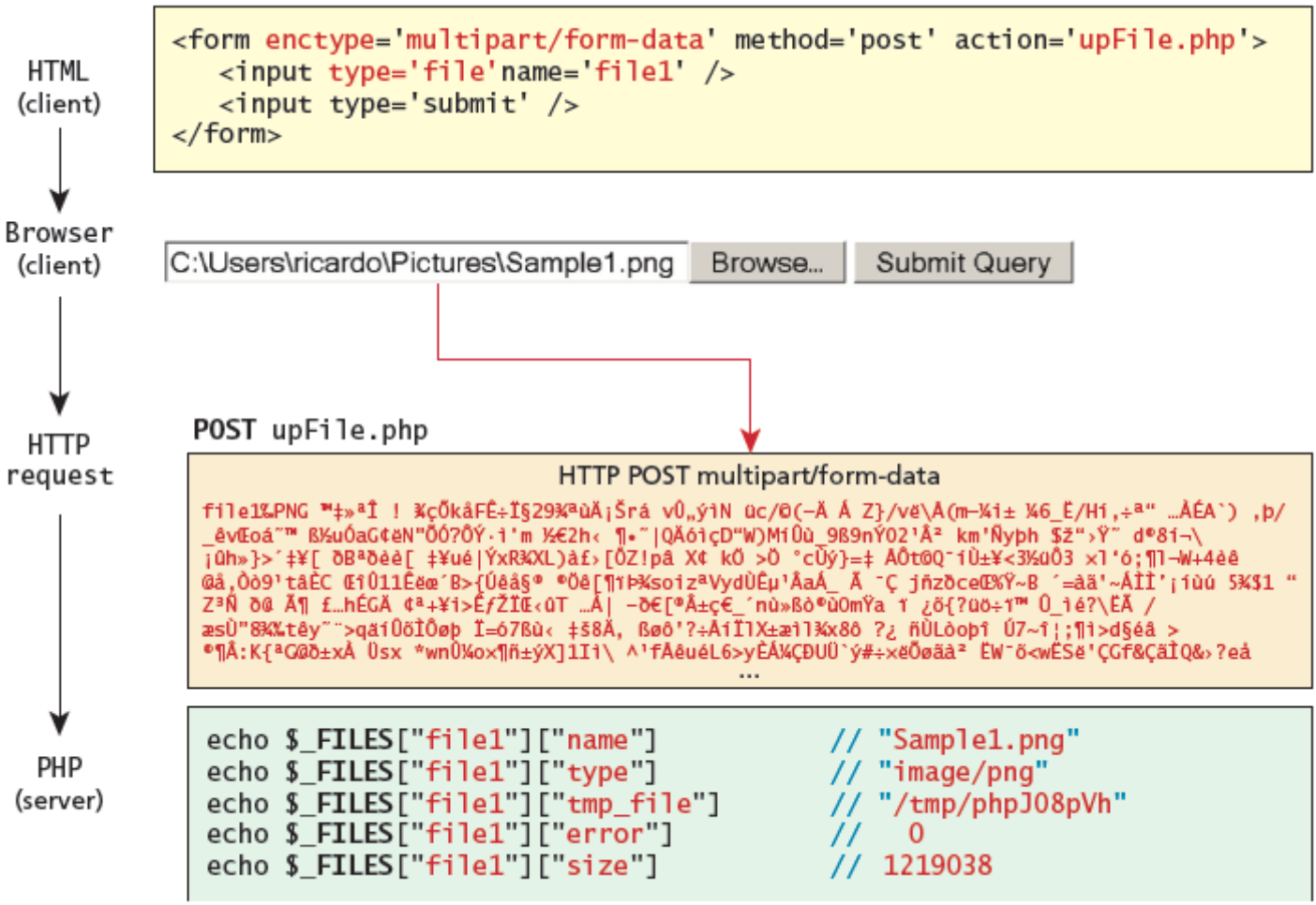


Figure 9.12 Data flow from HTML form through POST to PHP $_FILES array

9.4.3 Checking for Errors

- For every uploaded file, there is an error value associated with it in the $_FILES array.
- The full list of errors is provided in Table 9.2.

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.

- A proper file upload script will therefore check each uploaded file by checking the various error codes as shown in Listing 9.13.

```
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 9.13 Checking each file uploaded for errors

9.4.4 File Size Restrictions

- 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.
- The first of these mechanisms is to add a hidden input field before any other input fields in our HTML form with a name of MAX_FILE_SIZE.
- Listing 9.14 shows how the HTML Limiting upload file.

```
<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 9.14 Limiting upload file size via HTML

- 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 9.15.

```
<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 9.15 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 9.16 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 9.16 Limiting upload file size via PHP

9.4.5 Limiting the Type of File Upload

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

9.4.6 Moving the File

- we 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 9.18 illustrates a simple use of the function.

```
$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

9.5 Reading/Writing Files

- There are two basic techniques for read/writing files in PHP:
  1. **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.
  2. **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.

9.5.1 Stream Access

- The function fopen() takes a file location or URL and access mode as parameters.
- The returned value is a **stream resource**, which we 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, we 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 we can use the === check to see if we 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 fgetsc(). Finally, when finished processing the file we must close it using fclose().
- Listing 9.19 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 9.19 Opening, reading lines, and closing a file

- To write data to a file, we can employ the fwrite() function.

9.5.2 In-Memory File Access

- The functions shown in Table 9.3 provide a functions 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 9.3 In-Memory File Functions

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

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

- To write the contents of a string $writeme to a file, we use

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

9.6.2 Review Questions

1. What are the superglobal arrays in PHP?
2. What function is used to determine if a value was sent via query string?
3. How do we handle arrays of values being posted to the server?
4. Describe the relationship between keys and indexes in arrays.
5. How does one iterate through all keys and values of an array?
6. Are arrays sorted by key or by value, or not at all?
7. How would we get a random element from an array?
8. What does urlencode() do? How is it “undone”?
9. What information is uploaded along with a file?
10. How do we read or write a file on the server from PHP?
11. List and briefly describe the ways we can limit the types and size of file uploaded.
12. What classes of information are available via the $_SERVER superglobal array?
13. Describe why hidden form fields can easily be forged/changed by an end user.

PHP Classes and Objects

In this chapter we will learn . . .

- The principles of object-oriented development using PHP
- How to use built-in and custom PHP classes
- How to articulate we designs using UML class diagrams
- Some basic object-oriented design patterns

10.1 Object-Oriented Overview

10.1.1 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 10.1 illustrates the differences between a class, which defines an object's properties and methods, and the objects or instances of that class.

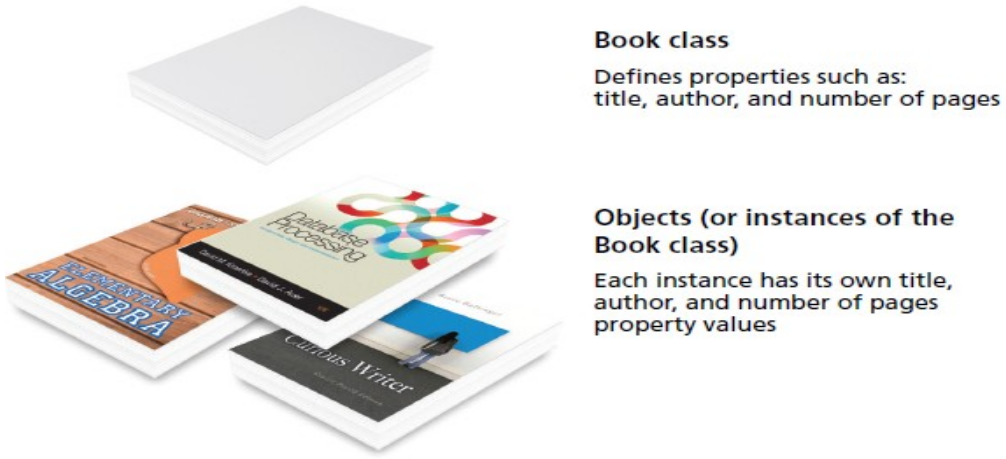


Figure 10.1 Relationship between a class and its objects

10.1.2 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.
- To illustrate classes and objects in UML, consider the artist.
- Figure 10.2 illustrates a UML class diagram, which shows an Artist class and multiple Artist objects, each object having its own properties.

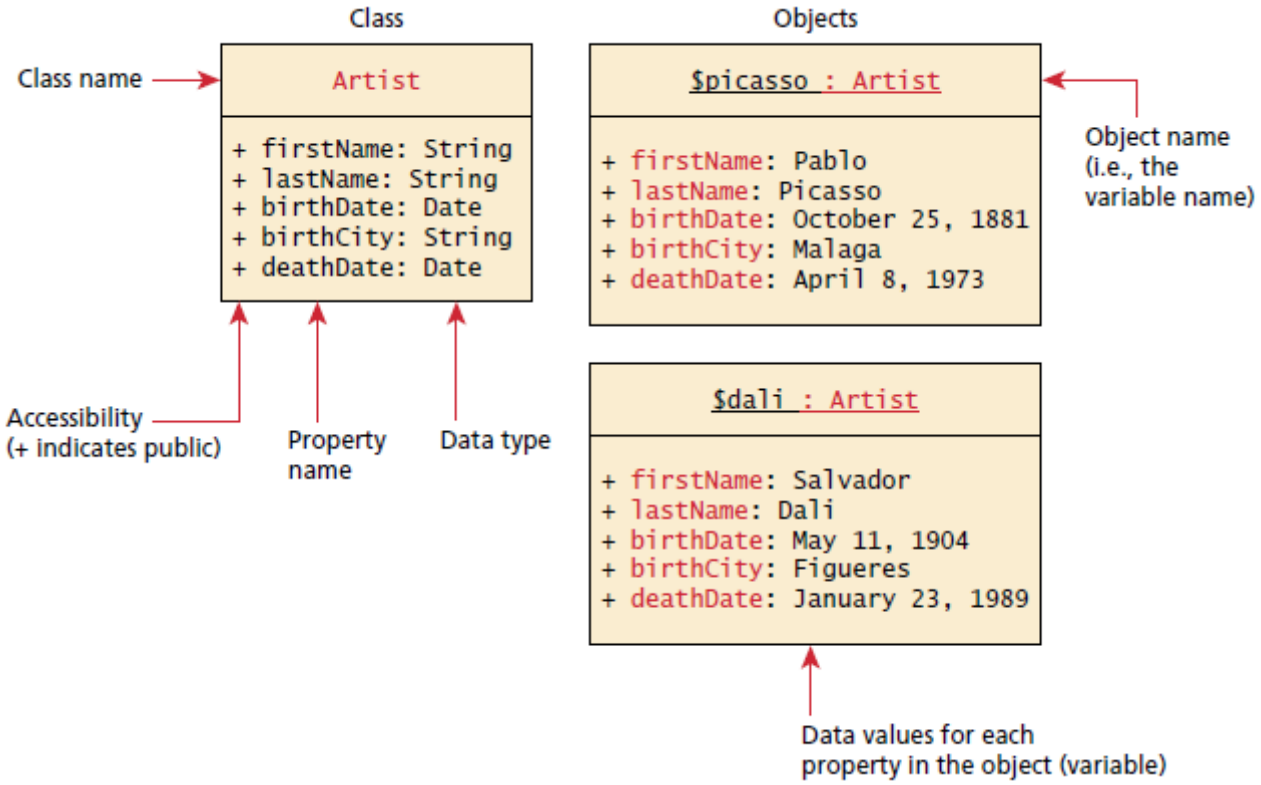


Figure 10.2 Relationship between a class and its objects in UML

10.1.3 Differences between Server and Desktop Objects

Figure 10.4 shows an illustration of the lifetimes of objects in memory between a desktop and a browser application.

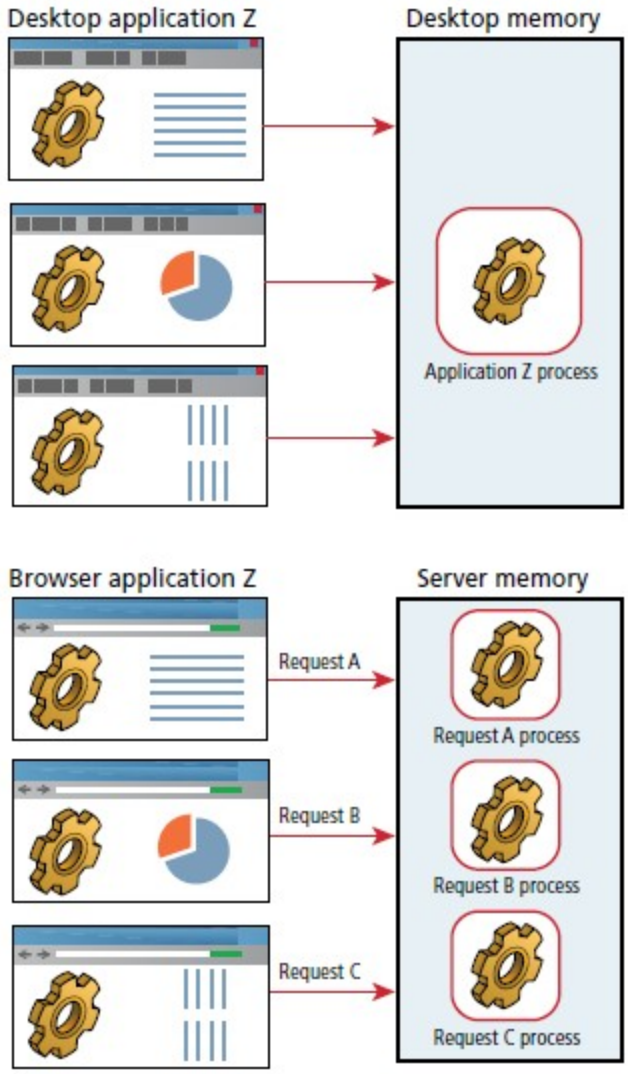


Figure 10.4 Lifetime of objects in memory in web versus desktop applications

10.2 Classes and Objects in PHP

- Classes should be defined in their own files so they can be imported into multiple scripts.
- Here we denote a class file by using the naming convention **classname.class.php**.

10.2.1 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.
- The Artist class with the properties illustrated in Figure 10.2 is defined using PHP in Listing 10.1.

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

Listing 10.1 A simple Artist class

10.2.2 Instantiating Objects

- 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, we instantiate two new objects using the new keyword as follows:

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

10.2.3 Properties

- we 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 10.2 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 10.2 Instantiating two Artist objects and setting one of those object's properties

10.2.4 Constructors

- In PHP, constructors are defined as functions (as we shall see, all methods use the function keyword) with the name __construct(). (Note: there are *two* underscores _ before the word construct.)
- Listing 10.3 shows an updated Artist class definition that now includes a constructor.
- In the constructor each parameter is assigned to an internal class variable using the $this-> syntax.

```
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 10.3 A constructor added to the class definition

10.2.5 Methods

- 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.

10.2.6 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 10.6 illustrates how visibility works in PHP.

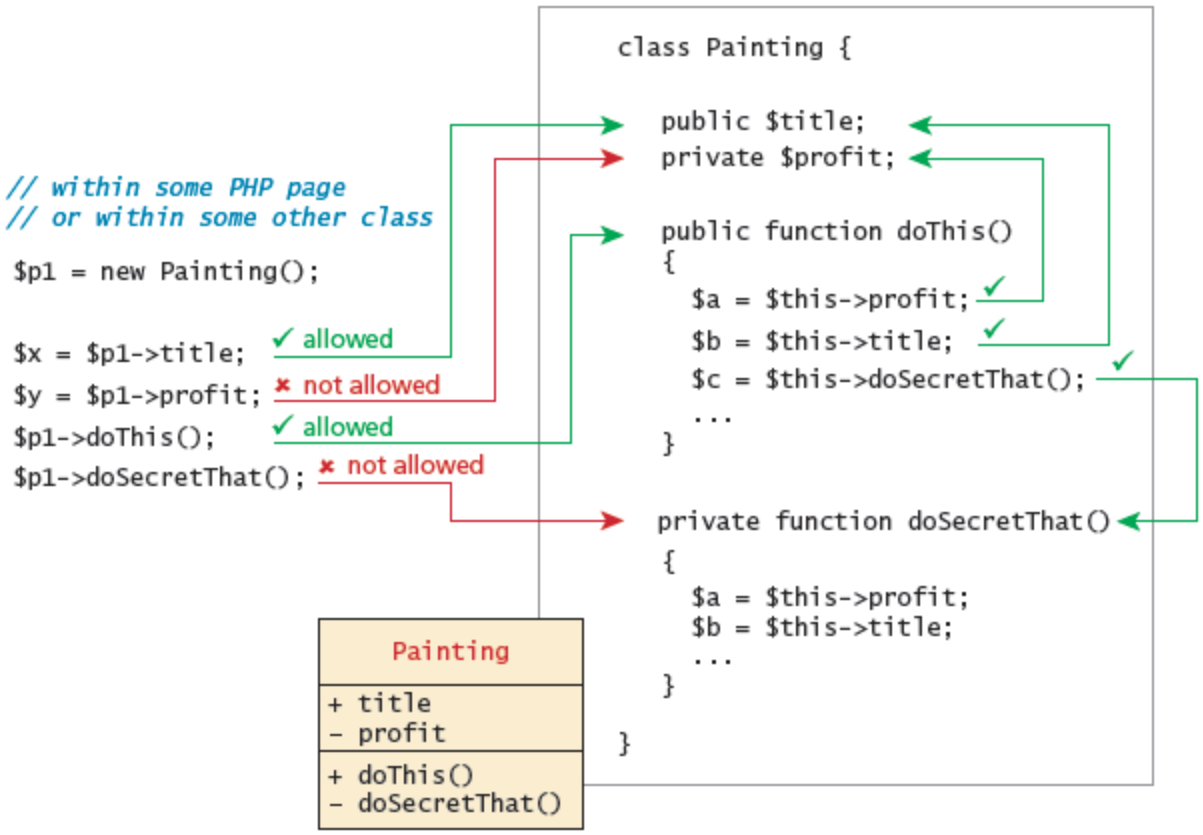


Figure 10.6 Visibility of class members

- As can be seen in Figure 10.6, 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.
- In UML, the "+" symbol is used to denote public properties and methods, the "–" symbol for private ones, and the "#" symbol for protected ones.

10.2.7 S tatic Members

- A **static** member is a property or method that all instances of a class share.
- 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.
- The 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 10.5.

```
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 10.5 Class definition modified with static members

- Notice that we do not reference a static property using the $this-> syntax, but rather it has its own self:: syntax.

10.2.8 Class Constants

- 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.

10.3 Object-Oriented Design

- The object-oriented design of software offers many benefits in terms of modularity, testability, and reusability

10.3.1 Data Encapsulation

- Encapsulation means hiding of an object's implementation details.
- 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).
- Encapsulated class makes its properties private, to access them we 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 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;
     }
}
```

10.3.2 Inheritance

- Inheritance enables we to create new PHP classes that reuse, extend, and modify the behavior that is defined in another PHP class.
- 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 10.9 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 { . . . }

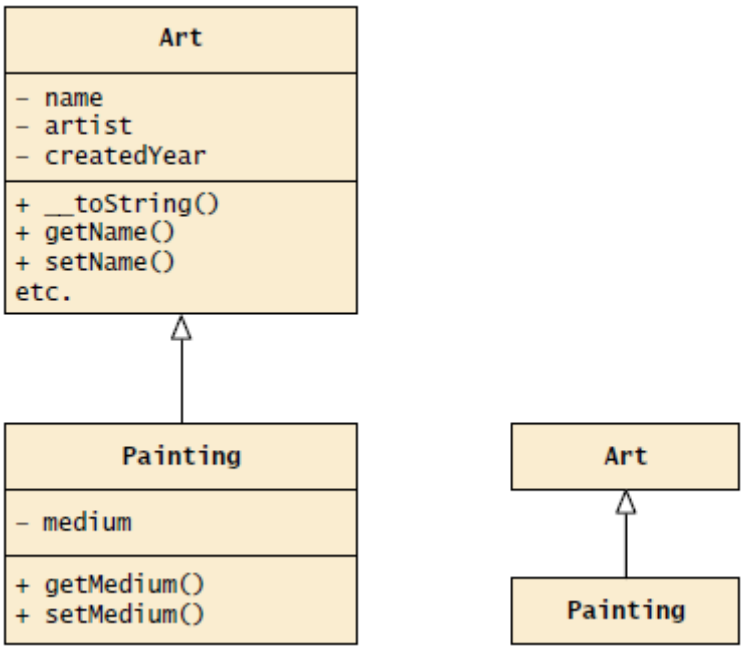


Figure 10.9 UML class diagrams showing inheritance

Referencing Base Class Members

- A subclass inherits the public and protected members of the base class.

```
$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()
```

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.

Parent Constructors

- If we want to invoke a parent constructor in the derived class's constructor, we can use the parent:: syntax and call the constructor on the first line parent::__construct().

10.3.3 Polymorphism

- **Polymorphism** is the notion that an object can in fact be multiple things at the same time.
- Listing 10.10 illustrates polymorphism at work.

```php
// 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 10.10 Using polymorphism

10.3.4 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.
- 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();
}
```

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

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

- Listing 10.11 defines a Viewable interface, which defines methods to return a **png** image to represent the viewable piece of art and get its size.

```php
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 10.11 Painting class implementing an interface

- In UML, interfaces are denoted through the <<interface>> stereotype.

Runtime Class and Interface Determination

- 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);
```

Review Questions

1. What is a static variable and how does it differ from a regular one?
2. What are the three access modifiers?
3. What is a constructor?
4. Explain the role of an interface in object-oriented programming.
5. What are the principles of data encapsulation?
6. What is the advantage of polymorphism?
7. When is the determination made as to which version of a method to call?Compile time or run time.

Error Handling and Validation

In this chapter we will learn . . .

- What the different types of errors are and how they differ fromexceptions
- The different forms of error reporting in PHP
- How to handle errors and exceptions
- What regular expressions are and how to use them in JavaScript and PHP
- Some best practices in design of user input validation
- How to validate user input in HTML5, JavaScript, and PHP

12.1 What Are Errors and Exceptions?

12.1.1 Types of Errors

There are three different types of website problems:

1. Expected errors
2. Warnings
3. Fatal errors

- An **expected error** is an error that routinely occurs during an application. Perhaps the most common example of this type would be an error as a result of user inputs, for instance, entering letters when numbers were expected.
- PHP provides two functions for testing the value of a variable. Figure 12.1 illustrates how these functions differ.

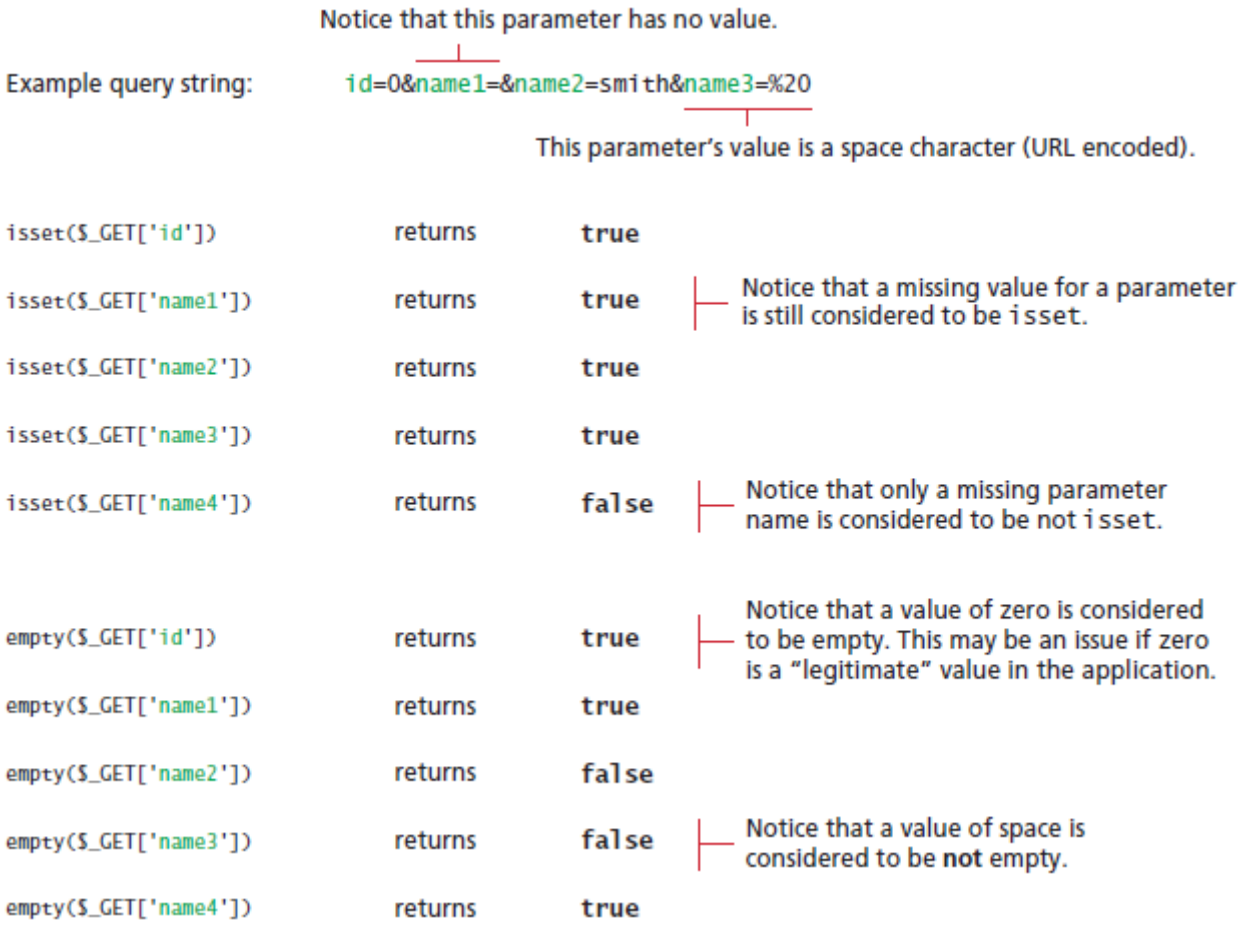


Figure 12.1 Comparing isset( ) and empty( ) with query string parameters

- If we are expecting a query string parameter to be numeric, then we can use the is_numeric() function, as shown in Listing 12.1.

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

Listing 12.1 Testing a query string to see if it exists and is numeric

- 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.
- 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.

12.1.2 Exceptions

- 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.

12.2 PHP Error Reporting

- There are three main error reporting flags:
  1. error_reporting
  2. display_errors

3. log_errors

12.2.1 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; Table 12.1 lists some of the most common values.

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

12.2.2 The display_errors Setting

- The display_error setting specifies whether error messages should or should not be displayed in the browser.2 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
```

12.2.3 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
```

- We can also programmatically send messages to the error log at any time via the error_log() function.Some examples of its use are as follows:

```
$msg = 'Some horrible error has occurred!';
// send message to system error log (default)
error_log($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');
```

12.3 PHP Error and Exception Handling

12.3.1 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.
- We needed to test for and deal with errors after each operation that might generate an error state, as shown in Listing 12.2.

```
$connection = mysqli_connect(DBHOST, DBUSER, DBPASS, DBNAME);

$error = mysqli_connect_error();
if ($error != null) {
   // handle the error
   ...
}
```

Listing 12.2 Procedural approach to error handling

12.3.2 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.
- Listing 12.3 illustrates a sample example of a try . . . catch block.

```
// 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

12.3.3 Custom Error and Exception Handlers

- If using the procedural approach (i.e., *not* using try . . . catch), we can define a custom *error*-handling function and then register it with the set_error_handler() function.
- If we are using the object-oriented exception approach with try . . . catch blocks, we can define a custom *exception*-handling function and then register it with the set_exception_handler() function.
- Listing 12.6 illustrates a sample custom exception handler function.

```
function my_exception_handler($exception) {

  // put together a detailed exception message
  $msg = "<p>Exception Number " . $exception->getCode();
  $msg .= $exception->getMessage() . " occurred on line ";
  $msg .= "<strong>" . $exception->getLine() . "</strong>";
  $msg .= " and in the file: ";
  $msg .= "<strong>" . $exception->getFile() . "</strong> </p>";

  // email error message to someone who cares about such things
  error_log($msg, 1, 'support@domain.com',
            'From: reporting@domain.com');
```

```
  // if exception serious then stop execution and tell maintenance fib
  if ($exception->getCode() !== E_NOTICE) {
     die("Sorry the system is down for maintenance. Please try
          again soon");
  }
}
```

Listing 12.6 Custom exception handler

- Once the handler function is defined, it must be registered, presumably at the beginning of the page, using the following code:

```
set_exception_handler('my_exception_handler');
```

12.4 Regular Expressions

- A **regular expression** is a set of special characters that define a pattern. They are a type of language that is intended for the matching and manipulation of text.
- In web development they are commonly used to test whether a user's input matches a predictable sequence of characters, such as those in a phone number, postal or zip code, or email address.

12.4.1 Regular Expression Syntax

- A regular expression consists of two types of characters: literals and metacharacters.
- A **literal** is just a character we wish to match in the target .
- A **metacharacter** is a special symbol that acts as a command to the regular expression parser. There are 14 metacharacters (Table 12.2). To use a metacharacter as a literal, we will need to *escape* it by prefacing it with a backslash (\).

.	[	]	\	(	)	^	$	|	*	?	{	}	+

Table 12.2 Regular Expression Metacharacters (i.e., Characters with Special Meaning)

- Table 12.3 lists examples of typical metacharacter usage to create patterns a typical regular expression is made up of several patterns.

Pattern	Description
^ qwerty $	If used at the very start and end of the regular expression, it means that the entire string (and not just a substring) must match the rest of the regular expression contained between the ^ and the $ symbols.
\t	Matches a tab character.
\n	Matches a new-line character.
.	Matches any character other than \n.
[qwerty]	Matches any single character of the set contained within the brackets.
[^qwerty]	Matches any single character not contained within the brackets.
[a-z]	Matches any single character within range of characters.
\w	Matches any word character. Equivalent to [a-zA-Z0-9].
\W	Matches any nonword character.
\s	Matches any white-space character.
\S	Matches any nonwhite-space character.
\d	Matches any digit.
\D	Matches any nondigit.
*	Indicates zero or more matches.
+	Indicates one or more matches.
?	Indicates zero or one match.
{n}	Indicates exactly n matches.
{n,}	Indicates n or more matches.
{n, m}	Indicates at least n but no more than m matches.
|	Matches any one of the terms separated by the | character. Equivalent to Boolean OR.
()	Groups a subexpression. Grouping can make a regular expression easier to understand.

Table 12.3 Common Regular Expression Patterns

- In PHP, regular expressions are contained within forward slashes. So, for instance, to define a regular expression, we would use the following:

```
$pattern = '/ran/';
```

12.4.2 Extended Example

- Table 12.4 contains several common regular expressions that we might use within a web application.

Regular Expression	Description
^\S{0,8}$	Matches 0 to 8 nonspace characters.
^[a-zA-Z]\w{8,16}$	Simple password expression. The password must be at least 8 characters but no more than 16 characters long.
^[a-zA-Z]+\w*\d+\w*$	Another password expression. This one requires at least one letter, followed by any number of characters, followed by at least one number, followed by any number of characters.
^\d{5}(-\d{4})?$	American zip code.
^((0[1-9])|(1[0-2]))\/(\d{4})$	Month and years in format mm/yyyy.
^(.+)@([^\.].*)\.([a-z]{2,})$	Email validation based on current standard naming rules.
^((http|https)://)?([\w-]+\.)+[\w]+(/[\w- ./?]*)?$	URL validation. After either http:// or https://, it matches word characters or hyphens, followed by a period followed by either a forward slash, word characters, or a period.
^4\d{3}[\s\-]d{4}[\s\-] d{4}[\s\-]d{4}$	Visa credit card number (four sets of four digits beginning with the number 4), separated by a space or hyphen.
^5[1-5]\d{2}[\s\-]d{4}[\s\-]d{4}[\s\-]d{4}$	MasterCard credit card number (four sets of four digits beginning with the numbers 51-55), separated by a space or hyphen.

Table 12.4 Some Common Web-Related Regular Expressions

12.5 Validating User Input

12.5.1 Types of Input Validation

- The following list indicates most of the common types of user input validation.
  1. **Required information**. Some data fields just cannot be left empty. For instance, the principal name of things or people is usually a required field.
  2. **Correct data type**. While some input fields can contain any type of data, other fields, such as numbers or dates, must follow the rules for its data type in order to be considered valid.
  3. **Correct format**. Some information, such as postal codes, credit card numbers, and social security numbers have to follow certain pattern rules.
  4. **Comparison**. Some user-entered fields are considered correct or not in relation to an already-inputted value.
  5. **Range check**. Information such as numbers and dates have infinite possible values.
  6. **Custom**. Some validations are more complex and are unique to a particular application. Some custom validations can be performed on the client side.

12.5.2 Notifying the User

- Most user validation problems need to answer the following questions:
  1. **What is the problem?** Users do not want to read lengthy messages to determine what needs to be changed. They need to receive a visually clear and textually concise message.
  2. **Where is the problem?** Some type of error indication should be located near the field that generated the problem.
  3. **If appropriate, how do I fix it?** For instance, don't just tell the user that a date is in the wrong format, tell him or her what format we are expecting, such as "The date should be in yy/mm/dd format."

12.5.3 How to Reduce Validation Errors

- Some of the most common ways of doing so include:

1. Using pop-up JavaScript alert (or other popup) messages.
2. Provide textual hints to the user on the form itself.
3. Using tool tips or pop-overs to display context-sensitive help about the expected input.
4. Another technique for helping the user understand the correct format for an input field is to provide a JavaScript-based mask.
5. Providing sensible default values for text fields can reduce validation errors.
6. Finally, many user input errors can be eliminated by choosing a better data entry type than the standard <input type="text">.

12.6 Where to Perform Validation

- The advantage of validation using JavaScript is that it reduces server load and provides immediate feedback to the user.
- Figure 12.7 illustrates the interaction of the different levels of validation.

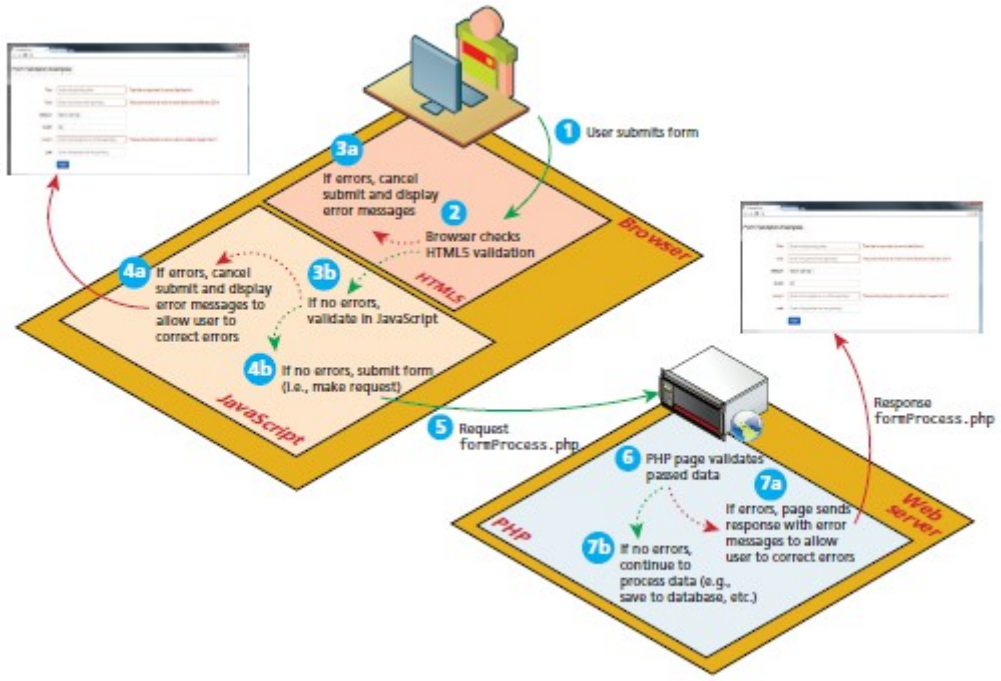


Figure 12.7 Visualizing levels of validation

- Consider the form and validations shown in Figure 12.8. Listing 12.8 shows the markup to which we will add validation.

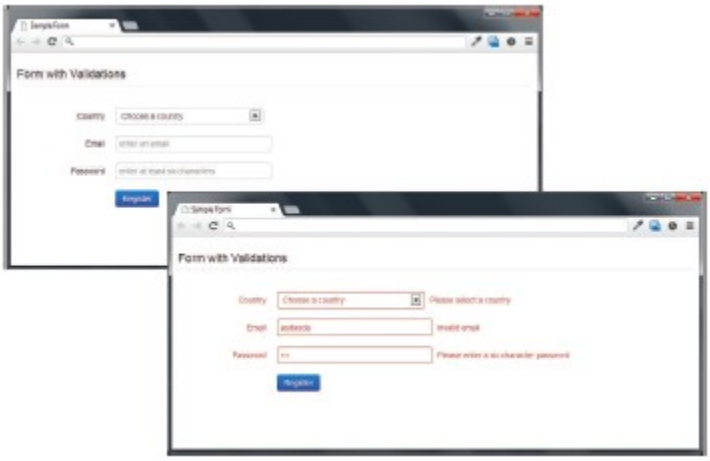


Figure 12.8 Example form to be validated

```
<form method="POST" action="validationform.php"
   class="form-horizontal" id="sampleForm" >
<fieldset>
<legend>Form with Validations</legend>

<div class="control-group" id="controlCountry">
  <label class="control-label" for="country">Country</label>
  <div class="controls">
    <select id="country" name="country" class="input-xlarge">
      <option value="0">Choose a country</option>
      <option value="1">Canada</option>
      <option value="2">France</option>
      <option value="3">Germany</option>
      <option value="4">United States</option>
    </select>
    <span class="help-inline" id="errorCountry"></span>
  </div>
</div>

<div class="control-group" id="controlEmail">
  <label class="control-label" for="email">Email</label>
  <div class="controls">
    <input id="email" name="email" type="text"
           placeholder="enter an email"
           class="input-xlarge" required>
    <span class="help-inline" id="errorEmail"></span>
  </div>
</div>

<div class="control-group" id="controlPassword">
  <label class="control-label" for="password">Password</label>
  <div class="controls">
    <input id="password" name="password" type="password"
          placeholder="enter at least six characters"
          class="input-xlarge" required>
    <span class="help-inline" id="errorPassword"></span>
  </div>
</div>

<div class="control-group">
  <label class="control-label" for="singlebutton"></label>
  <div class="controls">
    <button id="singlebutton" name="singlebutton"
            class="btn btn-primary">
       Register
    </button>
  </div>
</div>

</fieldset>
</form>
```

Listing 12.8 Example form (validationform.php) to be validated

- The required attribute can be added to an input element, and browsers that support it will perform their own validation and message as shown in Figure 12.9.

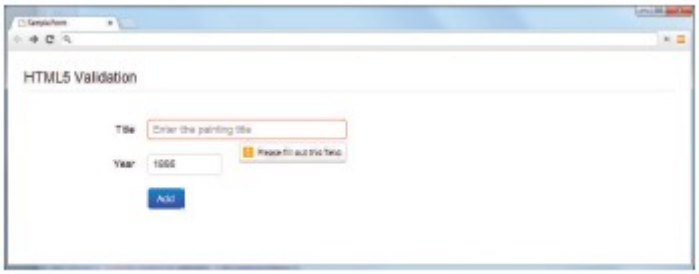


Figure 12.9 HTML5 browser validation

12.6.1 Validation at the JavaScript Level

- The second element in our validation strategy will be implemented within JavaScript.
- We can perform validation on an element once it loses its focus and when the user submits the form.

- Listing 12.9 lists the complete JavaScript validation solution.

```
<script>
// we will reference these repeatedly
var country = document.getElementById('country');
var email = document.getElementById('email');
var password = document.getElementById('password');

/*
  Add passed message to the specified element
*/
function addErrorMessage(id, msg) {
    // get relevant span and div elements
    var spanId = 'error' + id;
    var span = document.getElementById(spanId);
    var divId = 'control' + id;
    var div = document.getElementById(divId);

    // add error message to error <span> element
    if (span) span.innerHTML = msg;
    // add error class to surrounding <div>
    if (div) div.className = div.className + " error";
}

/*
  Clear the error messages for the specified element
*/
function clearErrorMessage(id) {
    // get relevant span and div elements
    var spanId = 'error' + id;
    var span = document.getElementById(spanId);
    var divId = 'control' + id;
    var div = document.getElementById(divId);
   // clear error message and class to error span and div elements
   if (span) span.innerHTML = "";
   if (div) div.className = "control-group";
}

/*
  Clears error states if content changes
*/
function resetMessages() {
   if (country.selectedIndex > 0)  clearErrorMessage('Country');
   if (email.value.length > 0)     clearErrorMessage('Email');
   if (password.value.length > 0)  clearErrorMessage('Password');
}
/*
  sets up event handlers
*/
function init() {
   var sampleForm = document.getElementById('sampleForm');
   sampleForm.onsubmit = validateForm;

   country.onchange = resetMessages;
   email.onchange = resetMessages;
   password.onchange = resetMessages;
}

/*
  perform the validation checks
*/
function validateForm() {
   var errorFlag = false;

   // check email
   var emailReg = /(.+)@([^\.].*)\.([a-z]{2,})/;
   if (! emailReg.test(email.value)) {
      addErrorMessage('Email', 'Enter a valid email');
      errorFlag = true;
   }

   // check password
   var passReg = /^[a-zA-Z]\w{8,16}$/;
   if (! passReg.test(password.value)) {
      addErrorMessage('Password', 'Enter a password between 9-16
                      characters');
      errorFlag = true;
   }
```

```
    // check country
    if ( country.selectedIndex <= 0 ) {
       addErrorMessage('Country', 'Select a country');
       errorFlag = true;
    }

    // if any error occurs then cancel submit; due to browser
    // irregularities this has to be done in a variety of ways
    if (! errorFlag)
       return true;
    else {
       if (e.preventDefault) {
          e.preventDefault();
       } else {
          e.returnValue = false;
       }
       return false;
    }
}

// set up validation handlers when page is downloaded and ready
window.onload = init;
</script>
```

Listing 12.9 Complete JavaScript validation

12.6.2 Validation at the PHP Level

- Validation on the server side using PHP is the most important form of validation and the only one that is absolutely essential.
- In this case, we will be validating the query string parameters rather than the form elements directly as with JavaScript. (Note:check out **Listing 12.11** PHP form validation in text book)
- Listing 12.12 illustrates Form with PHP validation messages

```
<form method="POST" action="<?php echo $_SERVER["PHP_SELF"];?>"
   class="form-horizontal" id="sampleForm" >
<fieldset>
<legend>Form with Validations</legend>

<!- Country select list -->
<div class="control-group <?php echo
     $countryValid->getCssClassName(); ?>" id="controlCountry">
  <label class="control-label" for="country">Country</label>
  <div class="controls">
    <select id="country" name="country" class="input-xlarge"
        value="<?php echo $countryValid->getValue(); ?>" >
      <option value="0">Choose a country</option>
      <option value="1">Canada</option>
      <option value="2">France</option>
      <option value="3">Germany</option>
      <option value="4">United States</option>
    </select>
    <span class="help-inline" id="errorCountry">
       <?php echo $countryValid->getErrorMessage(); ?>
    </span>
  </div>
</div>
```

```
<!- Email text box -->
<div class="control-group <?php echo
            $emailValid-> getCssClassName(); ?>" id="controlEmail">
  <label class="control-label" for="email">Email</label>
  <div class="controls">
    <input id="email" name="email" type="text"
       value="<?php echo $emailValid->getValue(); ?>"
          placeholder="enter an email" class="input-xlarge"
               required>
    <span class="help-inline" id="errorEmail">
       <?php echo $emailValid->getErrorMessage(); ?>
    </span>
  </div>
</div>

<!-- Password text box -->
<div class="control-group <?php echo $passValid->
            getCssClassName(); ?>" id="controlPassword">
  <label class="control-label" for="password">Password</label>
  <div class="controls">
    <input id="password" name="password" type="password"
       placeholder="enter at least six characters"
         class="input-xlarge" required>
    <span class="help-inline" id="errorPassword">
     <?php echo $passValid->getErrorMessage(); ?>
    </span>
  </div>
</div>

<!-- Submit button -->
<div class="control-group">
  <label class="control-label" for="singlebutton"></label>
  <div class="controls">
    <button id="singlebutton" name="singlebutton"
            class="btn btn-primary">
    Register</button>
  </div>
</div></fieldset>
</form>
```

Listing 12.12 Form with PHP validation messages

Review Questions

1. What are the three types of errors? How are errors different from exceptions?
2. What is the role of error reporting in PHP? How should it differ for development sites compared to production sites?
3. Discuss the trade-offs between procedural and object-oriented exception handling.
4. Discuss the role that regular expressions have in error and exception handling.
5. What are the most common types of user input validation?
6. Discuss strategies for handling validation errors. That is, what should our page do (from a user experience perspective) when an error occurs?
7. What strategies can one adopt when designing a form that will help reduce validation errors?
8. What problem does CAPTCHA address?
9. Validation checks should occur at multiple levels. What are the levels and why is it important to do so?