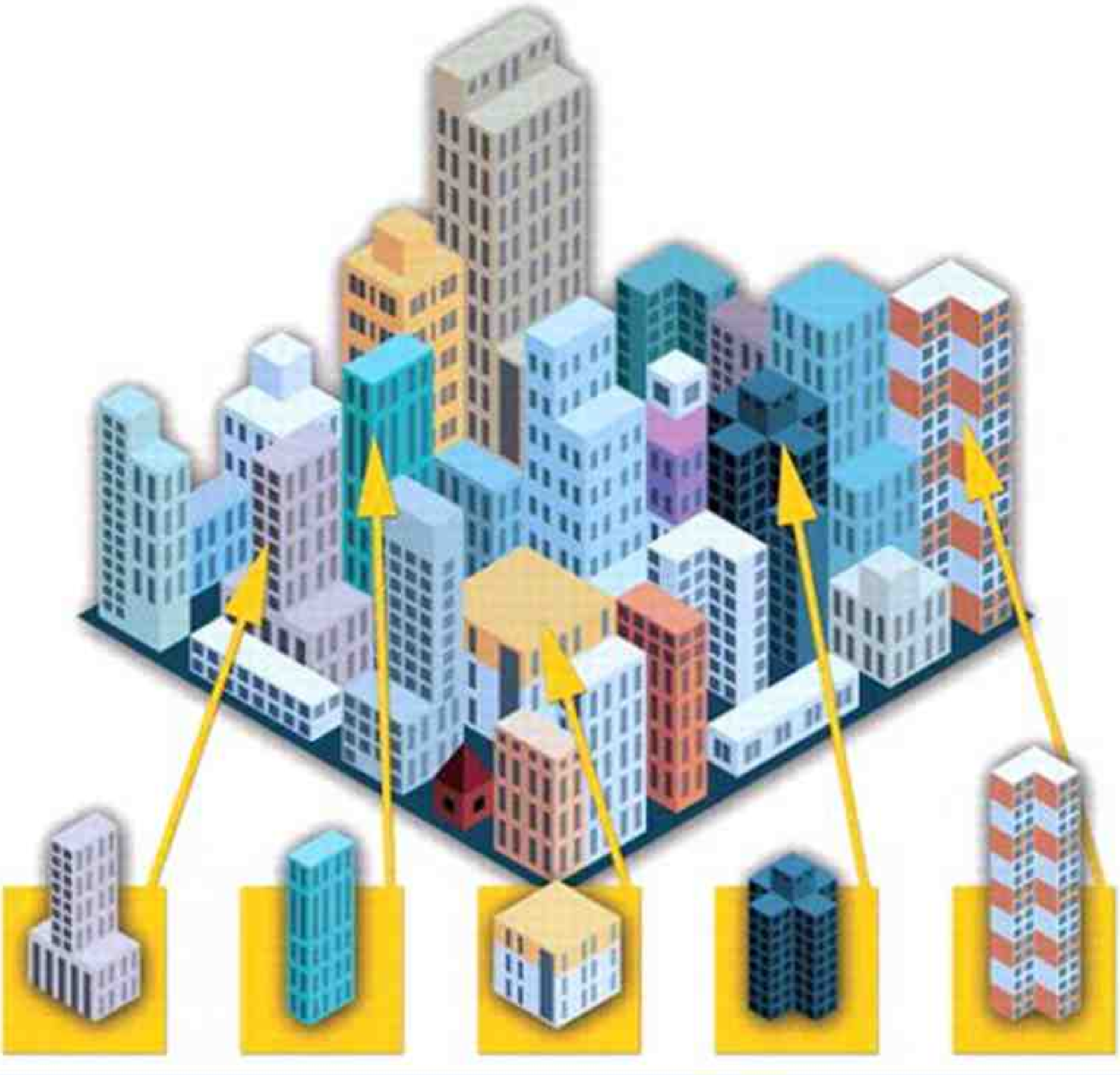
FUNDAMENTALS of WEB DEVELOPMENT



RANDY CONNOLLY • RICARDO HOAR

Variables, Data Types, and Constants 339
Writing to Output 342

8.4 Program Control 346
if . . . else 346
switch . . . case 347
while and do . . . while 348
for 349
Alternate Syntax for Control Structures 349
Include Files 350

8.5 Functions 351
Function Syntax 352
Calling a Function 353
Parameters 353
Variable Scope within Functions 356

8.6 Chapter Summary 358
Key Terms 358
Review Questions 358
Hands-On Practice 359
References 363

Chapter 9 PHP Arrays and Superglobals 364

9.1 Arrays 365
Defining and Accessing an Array 365
Multidimensional Arrays 367
Iterating through an Array 367
Adding and Deleting Elements 369
Array Sorting 371
More Array Operations 372
Superglobal Arrays 373

9.2 $_GET and $_POST Superglobal Arrays 374
Determining If Any Data Sent 375

Accessing Form Array Data 378
Using Query Strings in Hyperlinks 379
Sanitizing Query Strings 380

9.3 $_SERVER Array 382
Server Information Keys 383
Request Header Information Keys 383

9.4 $_FILES Array 385
HTML Required for File Uploads 385
Handling the File Upload in PHP 386
Checking for Errors 388
File Size Restrictions 388
Limiting the Type of File Upload 390
Moving the File 391

9.5 Reading/Writing Files 392
Stream Access 392
In-Memory File Access 393

9.6 Chapter Summary 395
Key Terms 395
Review Questions 395
Hands-On Practice 396
References 401

Chapter 10 PHP Classes and Objects 402

10.1 Object-Oriented Overview 403
Terminology 403
The Unified Modeling Language 403
Differences between Server and Desktop Objects 404

10.2 Classes and Objects in PHP 407
Defining Classes 407
Instantiating Objects 408
Properties 408

Constructors 409
Methods 410
Visibility 412
Static Members 412
Class Constants 414
10.3 Object-Oriented Design 415
Data Encapsulation 415
Inheritance 420
Polymorphism 427
Object Interfaces 429
10.4 Chapter Summary 432
Key Terms 432
Review Questions 433
Hands-On Practice 433
References 435

Chapter 11 Working with Databases 436

11.1 Databases and Web Development 437
The Role of Databases in Web Development 437
Database Design 437
Database Options 443
11.2 SQL 445
SELECT Statement 445
INSERT, UPDATE, and DELETE Statements 447
Transactions 448
Data Definition Statements 453
Database Indexes and Efficiency 453
11.3 Database APIs 454
PHP MySQL APIs 455
Deciding on a Database API 455

11.4 Managing a MySQL Database 456
Command-Line Interface 456
phpMyAdmin 457
MySQL Workbench 459

11.5 Accessing MySQL in PHP 460
Connecting to a Database 460
Handling Connection Errors 462
Executing the Query 464
Processing the Query Results 470
Freeing Resources and Closing Connection 474
Using Transactions 475

11.6 Case Study Schemas 476
Art Database 477
Book CRM Database 477
Travel Photo Sharing Database 478

11.7 Sample Database Techniques 479
Display a List of Links 479
Search and Results Page 480
Editing a Record 484
Saving and Displaying Raw Files in the Database 492

11.8 Chapter Summary 495
Key Terms 496
Review Questions 496
Hands-On Practice 496
References 502

Chapter 12 Error Handling and Validation 503

12.1 What Are Errors and Exceptions? 504
Types of Errors 504
Exceptions 506

12.2 PHP Error Reporting 506

The error_reporting Setting 507

The display_errors Setting 507

The log_error Setting 508

12.3 PHP Error and Exception Handling 509

Procedural Error Handling 509

Object-Oriented Exception Handling 509

Custom Error and Exception Handlers 512

12.4 Regular Expressions 513

Regular Expression Syntax 513

Extended Example 516

12.5 Validating User Input 519

Types of Input Validation 519

Notifying the User 520

How to Reduce Validation Errors 521

12.6 Where to Perform Validation 524

Validation at the JavaScript Level 528

Validation at the PHP Level 531

12.7 Chapter Summary 536

Key Terms 536

Review Questions 537

Hands-On Practice 537

References 540

Chapter 13 Managing State 541

13.1 The Problem of State in Web Applications 542

13.2 Passing Information via Query Strings 544

13.3 Passing Information via the URL Path 546

URL Rewriting in Apache and Linux 546

13.4 Cookies 547

How Do Cookies Work? 548

9 PHP Arrays and Superglobals

CHAPTER OBJECTIVES

In this chapter you will learn . . .

- How to create and use arrays in PHP
- How superglobal PHP variables simplify access to HTTP resources
- How to upload files to the server
- How to read and write text files

This chapter covers a variety of important PHP topics that build upon the PHP foundations introduced in Chapter 8. It covers PHP arrays, from the most basic all the way through to superglobal arrays, which are essential for almost any PHP web application. The chapter ends with a look at file processing in PHP, where you will learn to handle file uploads, as well as read and write text files directly.

9.1 Arrays

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

For some PHP developers, arrays are easy to understand, but for others they are a challenge. To help visualize what is happening, one should become familiar with the concept of keys and associated values. Figure 9.1 illustrates a PHP array with five strings containing day abbreviations.

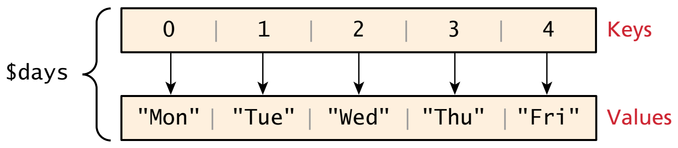


FIGURE 9.1 Visualization of a key-value array

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

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

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Use PHP Arrays

9.1.1 Defining and Accessing an Array

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

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

To define the contents of an array as strings for the days of the week as shown in Figure 9.1, you declare it with a comma-delimited list of values inside the ( ) braces using either of two following syntaxes:

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

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

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

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

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

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

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

In PHP, you are also able to explicitly define the keys in addition to the values. This allows you to use keys other than the classic 0, 1, 2, . . . , n to define the indexes of an array. For clarity, the exact same array defined above and shown in Figure 9.1 can also be defined more explicitly by specifying the keys and values as shown in Figure 9.2.

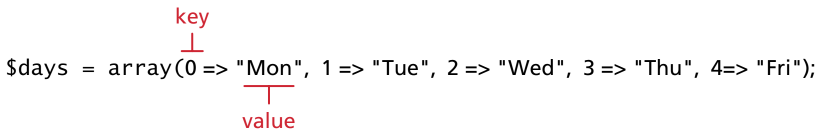
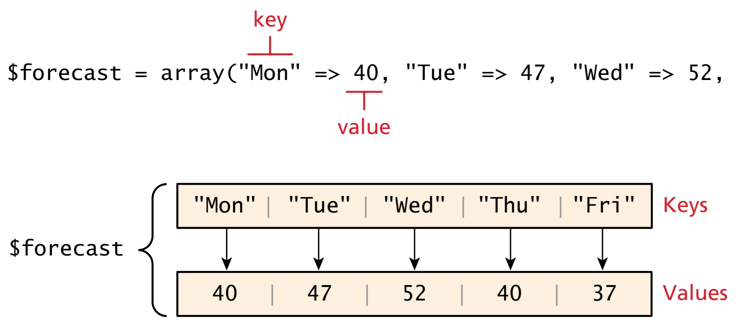

key
$days = array(0 => "Mon", 1 => "Tue", 2 => "Wed", 3 => "Thu", 4=> "Fri");
value

FIGURE 9.2 Explicitly assigning keys to array elements


key
$forecast = array("Mon" => 40, "Tue" => 47, "Wed" => 52, "Thu" => 40, "Fri" => 37);
value
$forecast
"Mon" | "Tue" | "Wed" | "Thu" | "Fri"
Keys
40 | 47 | 52 | 40 | 37
Values

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

FIGURE 9.3 Array with strings as keys and integers as values

Explicit control of the keys and values opens the door to keys that do not start at 0, are not sequential, and that are not even integers (but rather strings). This is why you can also consider an array to be a dictionary or hash map. These types of arrays in PHP are generally referred to as associative arrays. You can see in Figure 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. In the example in Figure 9.3, the keys are strings (for the weekdays) and the values are weather forecasts for the specified day in integer degrees.

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

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

9.1.2 Multidimensional Arrays

PHP also supports multidimensional arrays. Recall that the values for an array can be any PHP object, which includes other arrays. Listing 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
```

(continued)

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

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

LISTING 9.1 Multidimensional arrays

Figure 9.4 illustrates the structure of these two multidimensional arrays.

9.1.3 Iterating through an Array

HANDS-ON EXERCISES

LAB 9 EXERCISE
Iterating through a 2D Array

One of the most common programming tasks that you will perform with an array is to iterate through its contents. Listing 9.2 illustrates how to iterate and output the

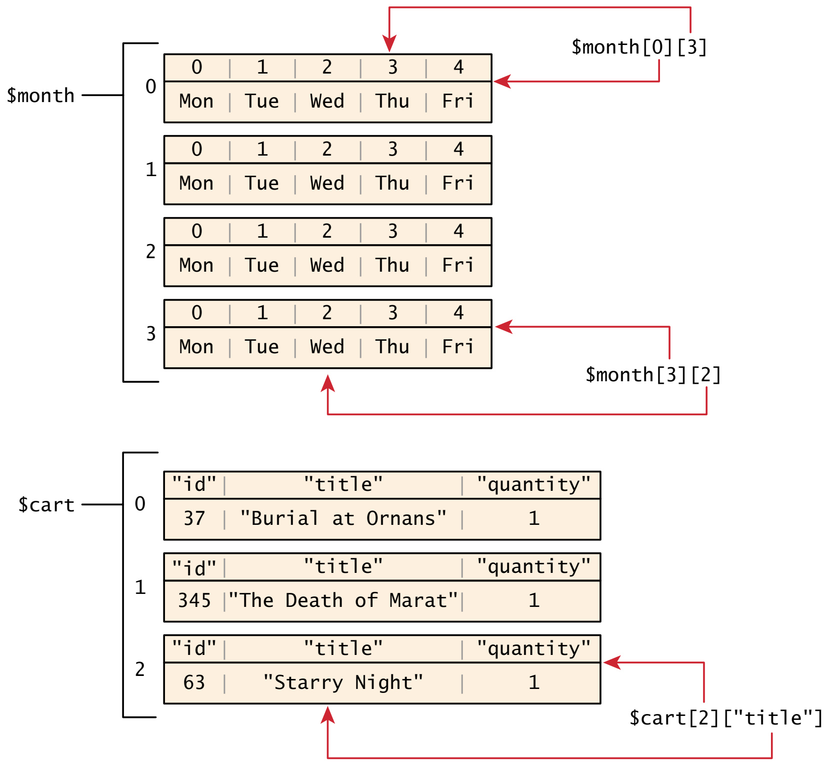


FIGURE 9.4 Visualizing multidimensional arrays

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


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


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

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

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

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

PRO TIP

In practice, arrays are printed in web apps using a loop as shown in Listing 9.2 and Listing 9.3. However, for debugging purposes, you can quickly output the content of an array using the `print_r()` function, which prints out the array and shows you the keys and values stored within. For example,

```
print_r($days);
```

Will output the following:

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

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";
```

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

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

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

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

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

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

That is, there is now a "gap" in our array that will cause problems if we try iterating through it using the techniques shown in Listing 9.2. If we try referencing `$days[6]`, for instance, it will return a NULL value, which is a special PHP value that represents a variable with no value.

You can also create "gaps" by explicitly deleting array elements using the `unset()` function, as shown in Listing 9.4.

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

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

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

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

LISTING 9.4 Deleting elements

Listing 9.4 also demonstrates that you can remove "gaps" in arrays (which really are just gaps in the index keys) using the `array_values()` function, which reindexes the array numerically.

Checking If a Value Exists

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

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

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

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

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Array Sorting

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

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

The resulting array in this case is:

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

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

9.1.6 More Array Operations

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

For a complete list, visit the `Array` class documentation php.net.[2]

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). They are called superglobal because

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

(continued)

Name	Description
`$_REQUEST`	Array containing the contents of `$_GET`, `$_POST`, and `$_COOKIES`
`$_SESSION`	Array that contains session data
`$_SERVER`	Array containing information about the request and the server

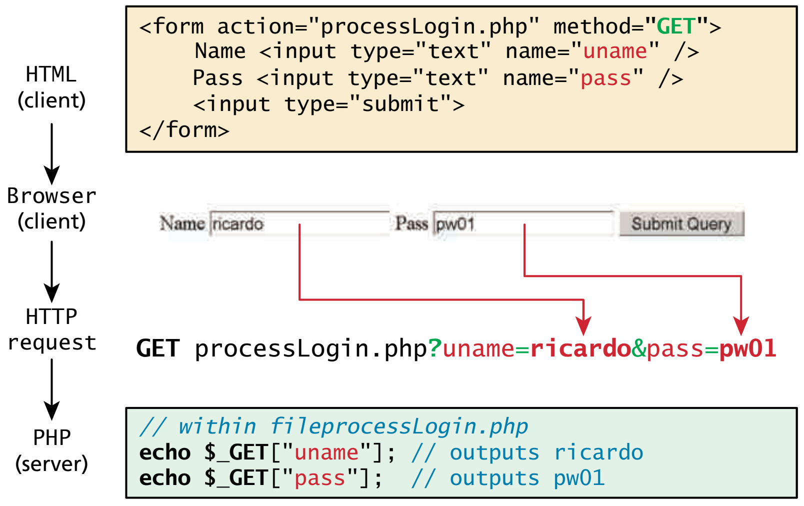
TABLE 9.1 Sugerglobal Variables

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

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

9.2 $_GET and $_POST Superglobal Arrays

The `$_GET` and `$_POST` arrays are the most important superglobal variables in PHP since they allow the programmer to access data sent by the client in a query string. As you will recall from Chapter 4, an HTML form (or an HTML link) allows a client to send data to the server. That data is formatted such that each value is associated with a name defined in the form. If the form was submitted using an HTTP `GET` request, then the resulting URL will contain the data in the query string. PHP will populate the superglobal `$_GET` array using the contents of this query string in the URL. Figure 9.5 illustrates the relationship between an HTML form, the `GET` request, and the values in the `$_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

NOTE

Although in our examples we are transmitting login data, including a password, we are only doing so to illustrate how sensitive information must at some point be transmitted. You should always use POST to transmit login credentials, on a secured SSL site, and moreover, you should hide the password using a password form field.

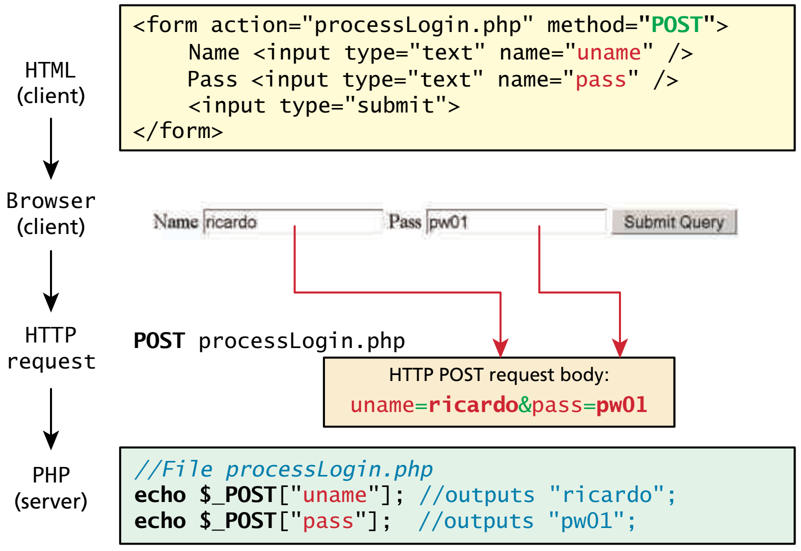
If the form was sent using HTTP POST, then the values would not be visible in the URL, but will be sent through HTTP POST request body. From the PHP programmer's perspective, almost nothing changes from a GET data post except that those values and keys are now stored in the $_POST array. This mechanism 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.

9.2.1 Determining If Any Data Sent

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Checking for POST



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

```
POST processLogin.php
```

HTTP POST request body:
`uname=ricardo&pass=pw01`

```
//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

NOTE

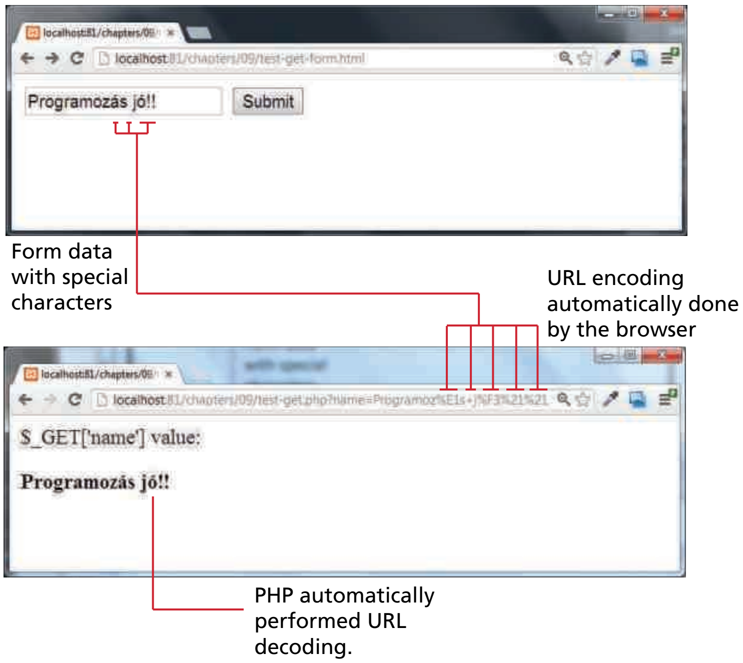
Recall from Chapter 4 that within query strings, characters such as spaces, punctuation, symbols, and accented characters cannot be part of a query string and are instead URL encoded.

One of the nice features of the `$_GET` and `$_POST` arrays is that the query string values are already URL decoded, as shown in Figure 9.7.

If you do need to manually perform URL encoding/decoding (say, for database storage), you can use the `urlencode()` and `urldecode()` functions. This should not be confused with HTML entities (symbols like >, <) for which there exists the `htmlentities()` function.

of the submitted form data, as shown in Figure 9.8. In such cases you may want to know whether any form data was submitted at all using either POST or GET.

In PHP, there are several techniques to accomplish this task. First, you can determine if you are responding to a `POST` or `GET` by checking the `$_SERVER['REQUEST_METHOD']` variable (we will cover the `$_SERVER` superglobal in more detail in Section 9.3). It contains as a string the type of HTTP request this script is responding to (`GET`, `POST`, `HEAD`, etc.). Even though you may know that, for



localhost:81/chapters/09/test-get-form.html

Programozás jó!!

localhost:81/chapters/09/test-get.php?name=Programoz%E1s+j%F3%21%21

$_GET['name'] value:

Programozás jó!!

localhost:81/chapters/09
Submit
Form data with special characters
URL encoding automatically done by the browser
PHP automatically performed URL decoding.

FIGURE 9.7 URL encoding and decoding

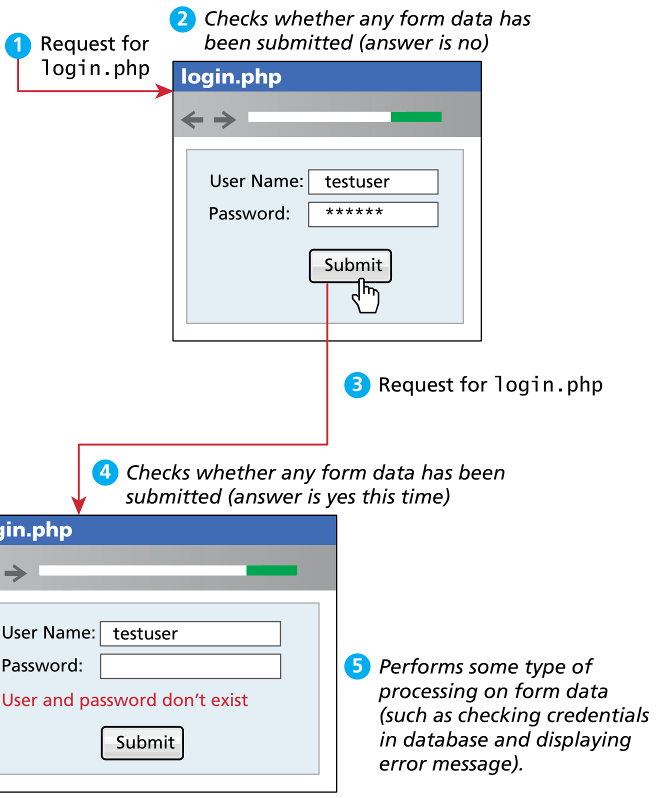

1 Request for login.php
2 Checks whether any form data has been submitted (answer is no)
login.php
User Name: testuser
Password: ******
Submit
3 Request for login.php
4 Checks whether any form data has been submitted (answer is yes this time)
login.php
User Name: testuser
Password:
User and password don't exist
Submit
5 Performs some type of processing on form data (such as checking credentials in database and displaying error message).

FIGURE 9.8 Form display and processing by the same PHP page

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

(continued)

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

LISTING 9.6 Using isset() to check query string data

NOTE

The PHP function `isset()` only returns `false` if a parameter name is missing. It still returns `true` if the parameter name exists but not the value. For instance, let us imagine that the query string looks like the following:

```
uname=&pass=
```

In such a case the condition `if(isset($_POST['uname']) && isset($_POST['pass']))` will evaluate to `true`. Thus, more coding will be necessary to further test the values of the parameters. Alternately, these two checks can be combined using the `empty()` function. However, the `empty()` function has its own limitations. To learn more about checking query strings, see Section 12.1.1.

9.2.2 Accessing Form Array Data

Sometimes in HTML forms you might have multiple values associated with a single name; back in Chapter 4, there was an example in Section 4.4.2 on checkboxes. Listing 9.7 provides another example. Notice that each checkbox has the same `name` value (`name="day"`).

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

LISTING 9.7 HTML that enables multiple values for one name

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

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

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

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

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Using Query String Values

You may wonder if it is possible to combine query strings with anchor tags . . . the answer is YES! Anchor tags (i.e., hyperlinks) also use the HTTP `GET` method. Indeed it is extraordinarily common in web development to programmatically construct the URLs for a series of links from, for instance, database data. Imagine a web page in which we are displaying a list of book links. One approach would be to have a separate page for each book (as shown in Figure 9.9). This is not a very sensible approach. Our database may have hundreds or thousands of books in it: surely it would be too much work to create a separate page for each book!

It would make a lot more sense to have a single Display Book page that receives as input a query string that specifies which book to display, as shown in Figure 9.10. Notice that we typically pass some type of unique identifier in the query string (in this case, using the book's ISBN).

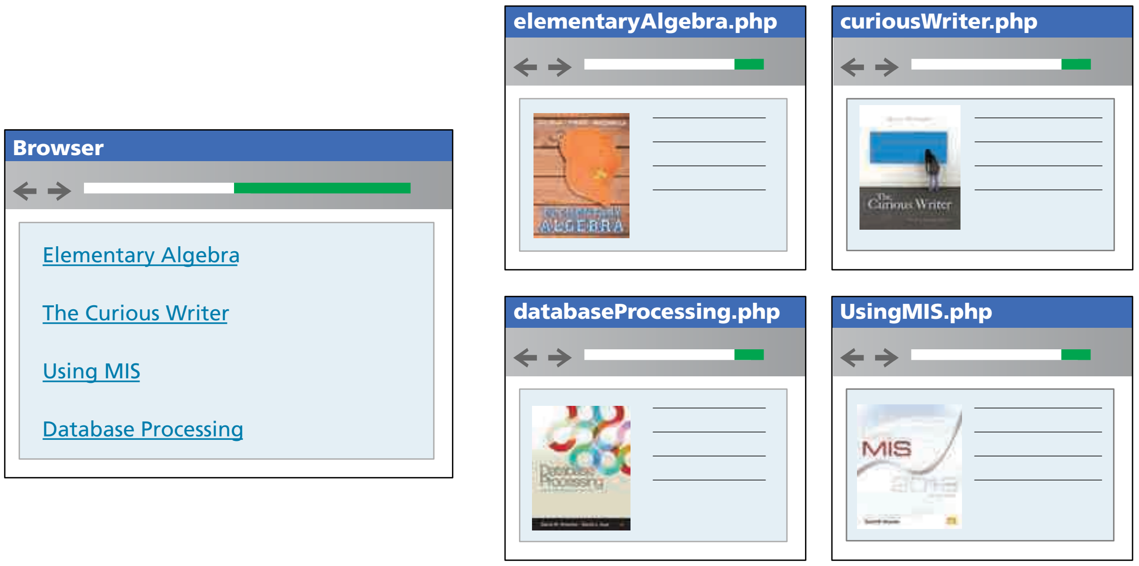


FIGURE 9.9 Inefficient approach to displaying individual items

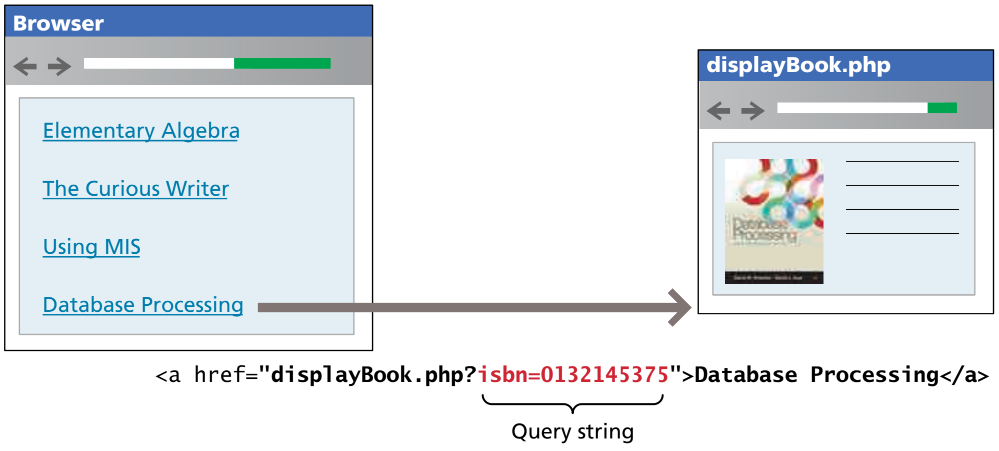
We will learn more about how to implement such pages making use of database information in Chapter 11.

9.2.4 Sanitizing Query Strings

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

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

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



```
<a href="displayBook.php?isbn=0132145375">Database Processing</a>
```

FIGURE 9.10 Sensible approach to displaying individual items using query strings

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

SECURITY

All data values that are potentially modifiable by the user, such as query strings, form values, or cookie values, must be sanitized before use. We will come back to this vital topic in Chapters 11, 12, and 16.

What should we do when an error occurs in Listing 9.9? There are a variety of possibilities; Chapter 12 will examine the issue of exception and error handling in more detail. For now, we might simply redirect to a generic error handling page using the header directive, for instance:

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

PRO TIP

In some situations, a more secure approach to query strings is needed, one that detects any user tampering of query string parameter values. One of the most common ways of implementing this detection is to encode the query string value with a one-way hash, which is a mathematical algorithm that takes a variable-length input string and turns it into fixed-length binary sequence. It is called one-way because it is designed to be difficult to reverse the process (that is, go from the binary sequence to the input string) without knowing the secret text (or salt in encryption lingo) used to generate the original hash. In such a case, our query string would change from id=16 to id=53e5e07397f7f01c2b276af813901c29.

9.3 $_SERVER Array

HANDS-ON EXERCISES

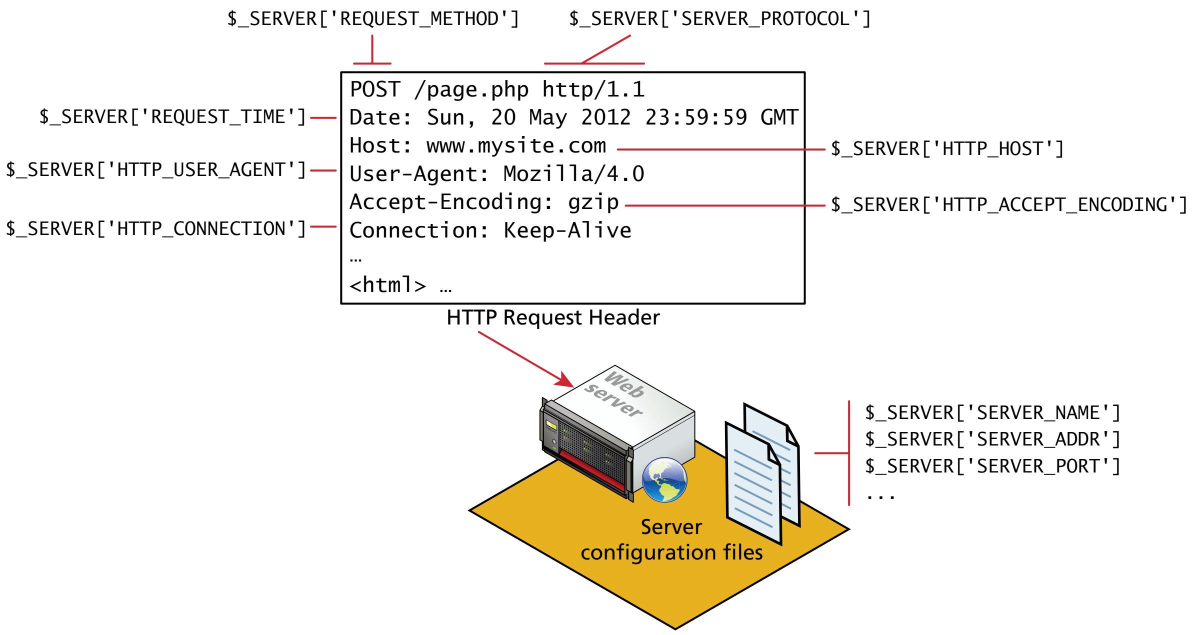
LAB 9 EXERCISE
Using the $_SERVER Superglobal

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.

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

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

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


$_SERVER['REQUEST_METHOD']
$_SERVER['SERVER_PROTOCOL']
$_SERVER['REQUEST_TIME']
$_SERVER['HTTP_USER_AGENT']
$_SERVER['HTTP_CONNECTION']
$_SERVER['HTTP_HOST']
$_SERVER['HTTP_ACCEPT_ENCODING']
HTTP Request Header
Web server
Server configuration files
$_SERVER['SERVER_NAME']
$_SERVER['SERVER_ADDR']
$_SERVER['SERVER_PORT']
...

```
POST /page.php http/1.1
Date: Sun, 20 May 2012 23:59:59 GMT
Host: www.mysite.com
User-Agent: Mozilla/4.0
Accept-Encoding: gzip
Connection: Keep-Alive
…
<html> …
```

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

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. If you are running multiple hosts on the same code base, this can be a useful piece of information. SERVER_ADDR is a complementary key telling us the IP of the server. Either of these keys can be used in a conditional to output extra HTML to identify a development server, for example.

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

9.3.2 Request Header Information Keys

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

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

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

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

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

PRO TIP

In order for get_browser() to work, your php.ini file must point the browscap setting to the correct location of the browscap.ini file on your system. A current browscap.ini file can be downloaded from php.net.[3] Also, this function is very complete, but slow. More simplistic string comparisons are often used when only one or two aspects of the user-agent string are important.

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

Listing 9.11 shows an example of context-dependent output that outputs a message to clients that came to this page from the search page, a message that is not shown to clients that came from any other link. This allows us to output a link back to the search page, but only when the user arrived from the search page.

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

LISTING 9.11 Using the HTTP_REFERER header to provide context-dependent output

SECURITY

All headers can be forged! The HTTP_REFERER header need not be honest about its contents, just as the USER_AGENT need not actually summarize the operating system and browser the client is using. Plug-ins exist in Firefox to allow the developer to in fact modify these headers. None of these headers can be trusted for security purposes, although they can be used to enhance the user experience since most users are not forging them.

9.4 $_FILES Array

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Processing File Uploads

9.4.1 HTML Required for File Uploads

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

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

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

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

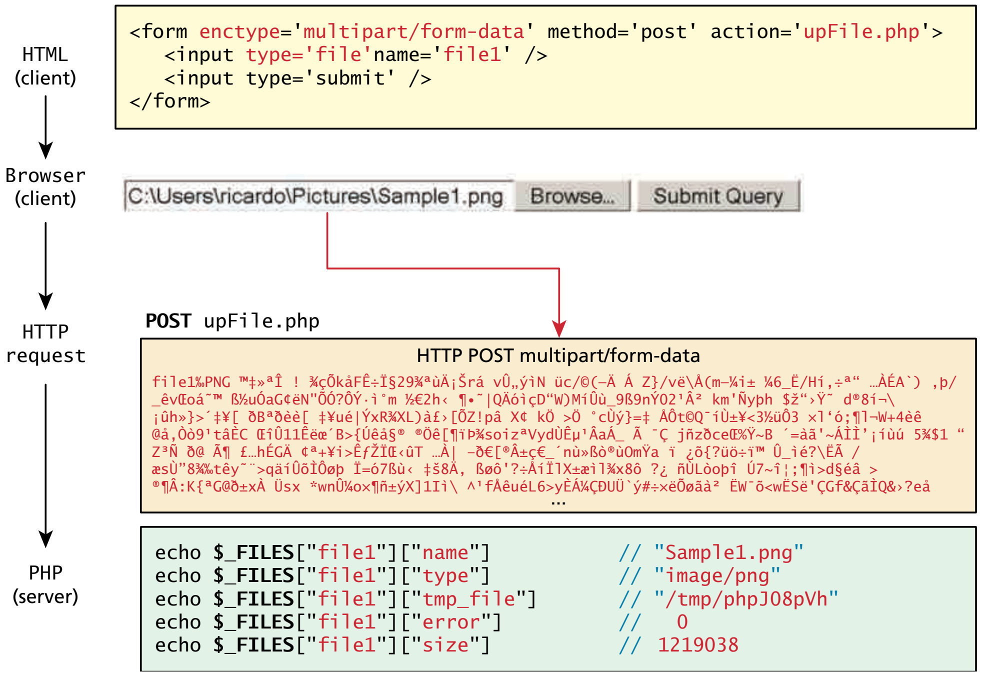
9.4.2 Handling the File Upload in PHP

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

Figure 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.

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

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



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

POST upFile.php

```
file1‰PNG ™‡»ªÎ ! ¾çÕkåFÊ÷Ï§29¾ªùÄ¡Šrá vÛ„ýìN üc/©(–Ä Á Z}/vë\Å(m–¼i± ¼6_Ë/Hí,÷ª“ …ÀÉA`) ,þ/
_êvŒoá˜™ ß½uÓaG¢ëN"ÕÓ?ÔÝ·ì°m ½€2h‹ ¶•˜|QÄóìçD“W)MíÛù_9ß9nÝ02¹Â² km'Ñyþh $ž“›Ÿ˜ d®8í¬\
¡ûh»}>´‡¥[ ðBªðèè[ ‡¥ué|ÝxR¾XL)à£›[ÕZ!pâ X¢ kÖ >Ö °cÙý}=‡ ÅÔt©Q¯íÙ±¥<3½üÔ3 ×l‘ó;¶l¬W+4èê
@å,Òò9¹tâÈC ŒîÛ11Êëæ´B>{Úêå§® ®Öê[¶ïÞ¾soizªVydÙÊµ¹ÂaÁ_ Ã ¯Ç jñzðceŒ%Ÿ~B ´=àã'~ÁÌÌ’¡íùú 5¾$1 “
Z³Ñ ð@ Ã¶ £…hÉGÄ ¢ª+¥i>ÊƒŽÏŒ‹ûT …À| –ð€[®Â±ç€_´nù»ßò®ùOmŸa ï ¿õ{?üö÷ï™ Û_ìé?\ËÃ /
æsÙ”8¾‰têy˜¨>qäíÛõÌÔøþ Ï=ó7ßù‹ ‡š8Ä, ßøô'?÷ÅíÏlX±æìl¾x8ô ?¿ ñÙLòoþî Ú7~î¦;¶ì>d§éâ >
®¶Â:K{ªG@ð±xÀ Üsx *wnÛ¼o×¶ñ±ýX]1Iì\ ^¹fÅêuéL6>yÈÁ¼ÇÐUÜ`ý#÷xëÕøãà² ËW¯õ<wËSë'ÇGf&ÇãÌQ&›?eå
…
```

```
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

NOTE

When PHP scripts are written to accept user uploads, they often run into errors since PHP is by default configured very conservatively. First and foremost, you must ensure your destination folder can be written to by the Apache web server. Check out Section 19.3.6 for more details.

In addition, you will want to be aware of several php.ini configuration directives including: `file_uploads`, `upload_file_maxsize`, `post_max_size`, `memory_limit`, `max_execution_time`, and `max_input_time`.

Some shared web hosts will not allow you to override these settings since they can negatively impact server performance. The setting `max_input_time`, for example, allows Apache to terminate scripts that run too long. Increasing this value too high would allow a badly written script with an infinite loop to run for as long as specified, slowing down the server for everyone else.

The location for storage of temporary files is also controlled in php.ini. It can be changed by modifying the path associated with the `upload_tmp_dir` attribute. Be aware that on some shared hosting packages your temporary files are accessible to others!

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

9.4.3 Checking for Errors

For every uploaded file, there is an error value associated with it in the `$_FILES` array. The error values are specified using constant values, which resolve to integers. The value for a successful upload is `UPLOAD_ERR_OK`, and should be looked for before proceeding any further. The full list of errors is provided in Table 9.2 and shows that there are many causes for bad file uploads.

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

9.4.4 File Size Restrictions

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

Error Code	Integer	Meaning
UPLOAD_ERR_OK	0	Upload was successful.
UPLOAD_ERR_INI_SIZE	1	The uploaded file exceeds the `upload_max_filesize` directive in php.ini.
UPLOAD_ERR_FORM_SIZE	2	The uploaded file exceeds the `max_file_size` directive that was specified in the HTML form.
UPLOAD_ERR_PARTIAL	3	The file was only partially uploaded.
UPLOAD_ERR_NO_FILE	4	No file was uploaded. Not always an error, since the user may have simply not chosen a file for this field.
UPLOAD_ERR_NO_TMP_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]

```
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

The first of these mechanisms is to add a hidden input field before any other input fields in your HTML form with a name of MAX_FILE_SIZE. This technique allows your php.ini maximum file size to be large, while letting some forms override that large limit with a smaller one. Listing 9.14 shows how the HTML from Listing 9.12 must be modified to add such a check. It should be noted that though this mechanism is set up in the HTML form, it is only available to use when your server-side environment is using PHP.

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

LISTING 9.14 Limiting upload file size via HTML

NOTE

This MAX_FILE_SIZE hidden field **must** precede the file input field. As well, its value must be within the maximum file size accepted by PHP.

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

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

HANDS-ON EXERCISES

LAB 9 EXERCISE
Managing Uploaded Files

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

SECURITY

The file extension and type field are transmitted by the client, and could be forged. You have likely yourself encountered how easy it is to change a file extension. Changing the type transmitted is also possible. Therefore when uploading data that will be publicly accessible, a more robust check should be done. For images this might include exploring the Exif data (Exchangeable image file format, which contains a wide range of metadata about an image), embedded inside the image file, which we will show you later in Chapter 17.

9.4.6 Moving the File

With all of our checking completed, you may now finally want to move the temporary file to a permanent location on your server. Typically, you make use of the PHP function `move_uploaded_file()`, which takes in the temporary file location and the file's final destination. This function will only work if the source file exists and if the destination location is writable by the web server (Apache). If there is a problem the function will return false, and a warning may be output. Listing 9.18 illustrates

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

LISTING 9.18 Using move_uploaded_file() function

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

9.5 Reading/Writing Files

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

NOTE

When reading a file from an external site, you should be aware that your script will not proceed until the remote website responds to the request. If you do not control the other website, you should be cautious about relevant intellectual property restrictions on the data you are retrieving.

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.

9.5.1 Stream Access

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

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

Once the file is opened, you can read from it in several ways. To read a single line, use the `fgets()` function, which will return false if there is no more data, and if it reads a line it will advance the stream forward to the next one so you can use the === check to see if you have reached the end of the file. To read an arbitrary amount of data (typically for binary files), use `fread()` and for reading a single character use `fgetsc()`. Finally, when finished processing the file you 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 `<br>` 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

NOTE

When processing text files, the operating system on which they were created will define how a new line is encoded. Unix and Linux systems use a `\n` while Windows systems use `\r\n` and many Macs use `\r`. Common errors can arise when the developer relies on the system they were using at the time, which might not work across platforms.

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

9.5.2 In-Memory File Access

While the previous approach to reading/writing files gives you complete control, the programming requires more care in dealing with the streams, file handles, and other low-level issues. The alternative simpler approach is much easier to use, at the cost of relinquishing fine-grained control. The functions shown in Table 9.3 provide a simpler alternative to the processing of a file in PHP.

HANDS-ON EXERCISES

LAB 9 EXERCISE
PHP File Access

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 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);
```

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

```
01070,Picasso,The Actor,1904
01080,Picasso,Family of Saltimbanques,1905
02070,Matisse,The Red Madras Headdress,1907
05010,David,The Oath of the Horatii,1784
```

To read and then parse this text file is quite straightforward, as shown in Listing 9.20.

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

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

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

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

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

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

    // do something with this data
    . . .
  }
```

LISTING 9.20 Processing a comma-delimited file

9.6 Chapter Summary

This chapter covered some important features of PHP. It began by exploring how to create, use, iterate, and sort arrays. Superglobal arrays were then introduced, which provide easy access to server and request variables, along with GET and POST query string data. Finally, file upload and processing in PHP was covered including some validation techniques to manage the type and size of uploaded assets.

9.6.1 Key Terms

All-in-memory access	NULL	stream access
array keys	one-way hash	stream resource
array values	ordered map	superglobal variables
associative arrays	sanitizing user input	user-agent

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 you 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 you read or write a file on the server from PHP?
11. List and briefly describe the ways you 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.

9.6.3 Hands-On Practice

PROJECT 1: Art Store

DIFFICULTY LEVEL: Beginner

HANDS-ON EXERCISES

PROJECT 9.1

Overview

Demonstrate your ability to work with arrays and superglobals in PHP.

Instructions

1. You have been provided with two files: the data entry form (Chapter09-project01.php) and the page that will process the form data (art-form-process.php). Examine both in the browser.
2. Modify Chapter09-project01.php so that it uses the `POST` method and art-form-process.php as the form `action`.
3. Modify art-form-process.php so that it displays the email, first name, last name, and privacy values that were entered into the form, as shown in Figure 9.13. This will require using the appropriate superglobal array. Also display the first name and last name in the welcome greeting.
4. In art-form-process.php, define an array that contains the labels for the account menu (see Figure 9.13). Replace the hard-coded list in the file with a loop that displays the equivalent list using the contents of your just-defined array. Notice that some conditional logic will be required to add the `class="active"` attribute to the correct `<li>` element.
5. Modify art-footer.inc.php so that it includes the array defined within art-data.php. Replace the hard-coded markup in the file with a loop that outputs the equivalent markup but uses the data defined in the array.

Test

1. Test the page. Remember that you cannot simply open a local PHP page in the browser using its open command. Instead you must have the browser request the page from a server. If you are using a local server such as XAMMP, the file must exist within the htdocs folder of the server, and then the request will be localhost/some-path/Chapter09-project01.php.

PROJECT 2: Share Your Travel Photos

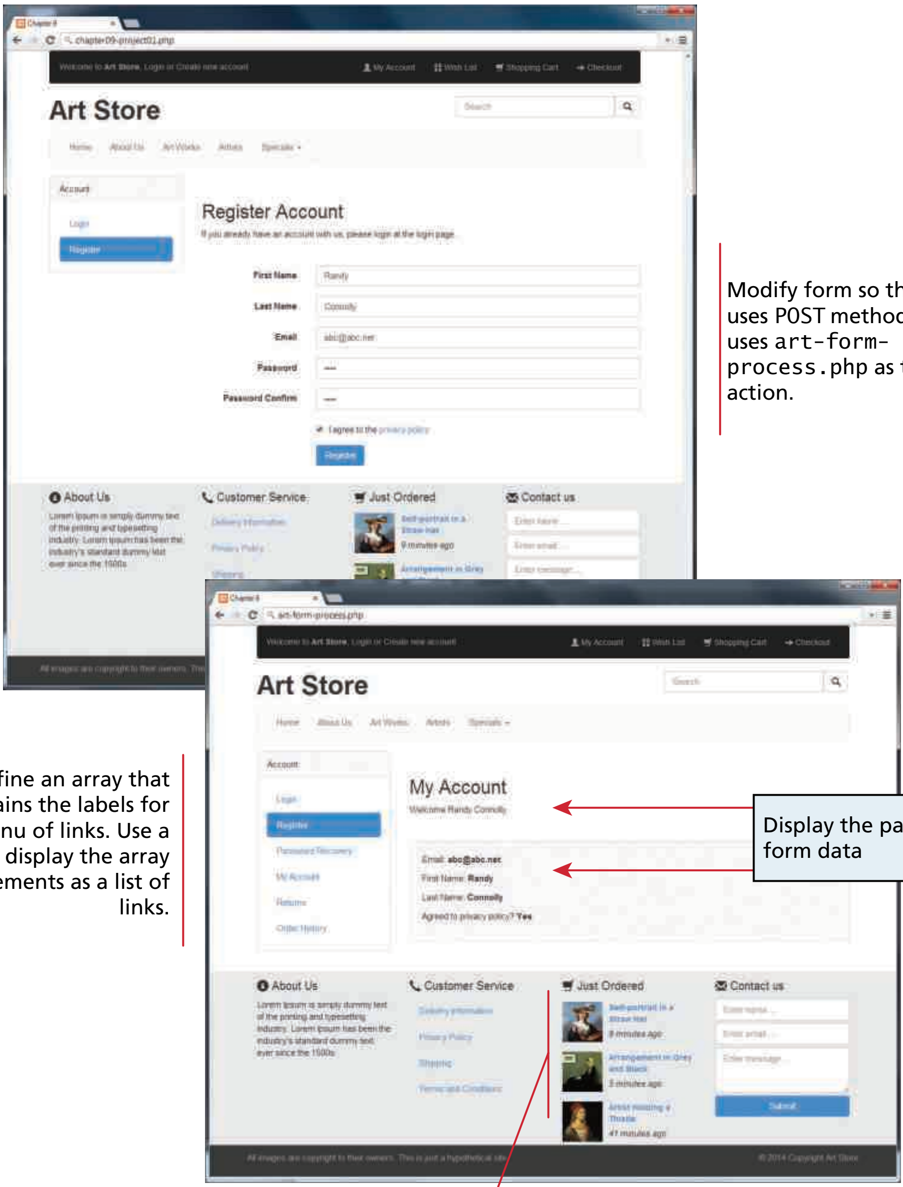
DIFFICULTY LEVEL: Intermediate

HANDS-ON EXERCISES

PROJECT 9.2

Overview

You have been provided with two files: a page that will eventually contain thumbnails for a variety of travel images (Chapter09-project02.php) and a page that will eventually display the details of a single travel image (travel-image.php). Clicking a thumbnail in the first file will take you to the second page where you will be able to see details for that image, as shown in Figure 9.14.



First Name: Randy
Last Name: Connolly
Email: abc@abc.net

Email: **abc@abc.net**
First Name: **Randy**
Last Name: **Connolly**
Agreed to privacy policy? **Yes**

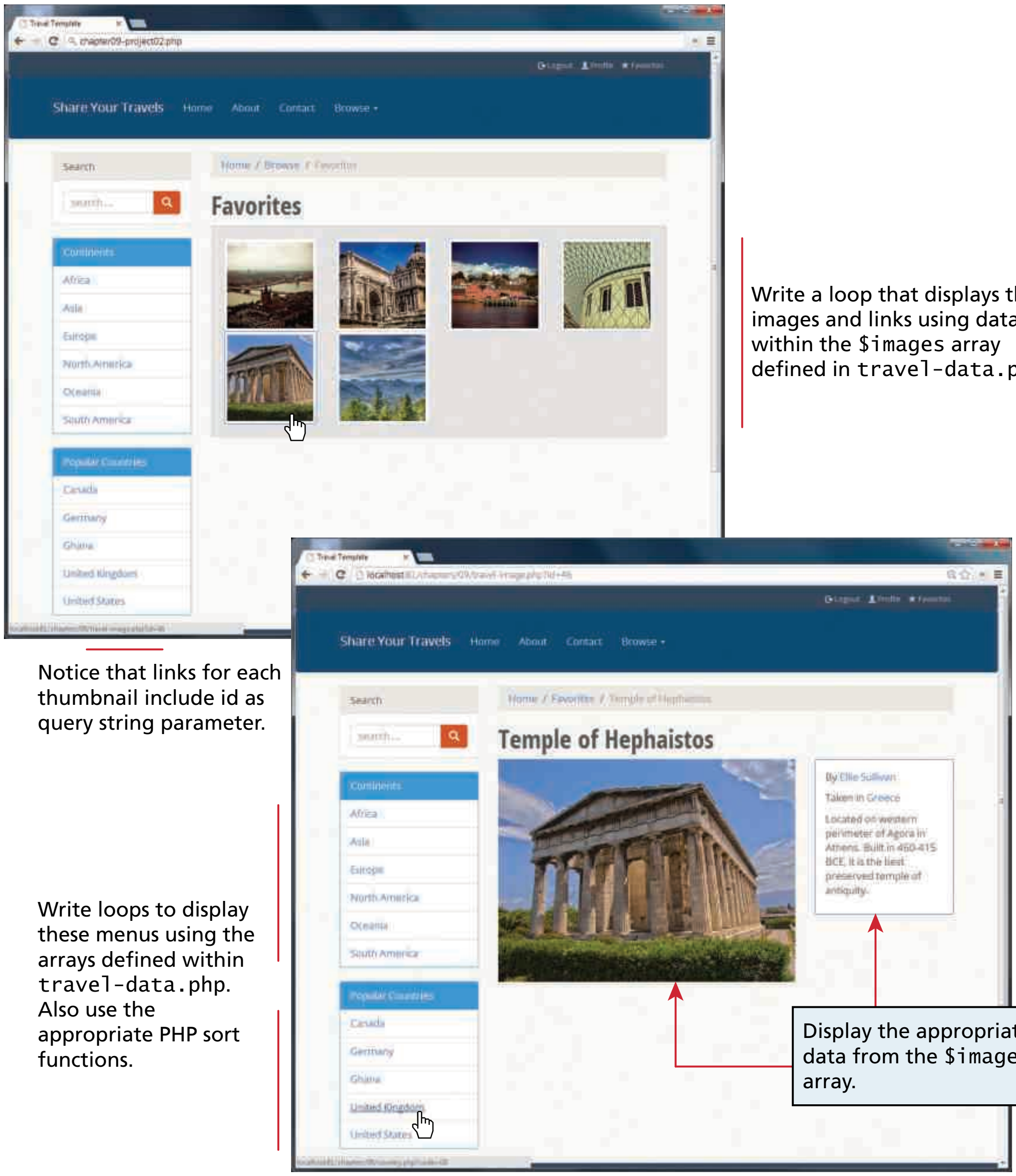
Modify form so that it uses POST method and uses `art-form-process.php` as the action.

Define an array that contains the labels for this menu of links. Use a loop to display the array elements as a list of links.

Display the passed form data

Modify footer so that this list is displayed via a loop using the array defined within `art-data.php`.

FIGURE 9.13 Completed Project 1



By Ellie Sullivan

Taken in Greece

Located on western perimeter of Agora in Athens. Built in 460-415 BCE, it is the best preserved temple of antiquity.

Write a loop that displays these images and links using data within the `$images` array defined in `travel-data.php`.

Notice that links for each thumbnail include id as query string parameter.

Write loops to display these menus using the arrays defined within `travel-data.php`. Also use the appropriate PHP sort functions.

Display the appropriate data from the `$images` array.

Notice that links for countries need to include the country code as a query string parameter.

FIGURE 9.14 Completed Project 2

Instructions

1. Both pages will make use of arrays that are contained within the include file travel-data.php. Include this file in both pages.
2. Both pages make use of an include file called travel-left-rail.php, which contains the markup for the navigation rail on the left side of both pages. Replace the hard-coded continents and countries lists in the file with loops that output the equivalent markup but use the data defined in the arrays. After verifying that it works, use the appropriate sort functions to sort the continents and countries arrays alphabetically. For the country list, notice that the links include the code in the query string.
3. Within Chapter09-project02.php in the `<div>` below the Favorites heading, replace the existing markup with a loop that displays the thumbnail image and link for each of the elements within the `$images` array (which is provided within travel-data.php). Notice that the links are to travel-image.php and that they pass the `id` element as a query string parameter.
4. In travel-data.php, retrieve the passed `id` in the query string, and use it as an index into the `$images` array. With that index, you can output the relevant title, image (in the images/travel/medium folder), user name, country, and description.

Test

1. Test the pages in the browser (see the test section of the previous section to remind yourself about how to do this).

PROJECT 3: Book Rep Customer Relations Management

DIFFICULTY LEVEL: Advanced

HANDS-ON EXERCISES

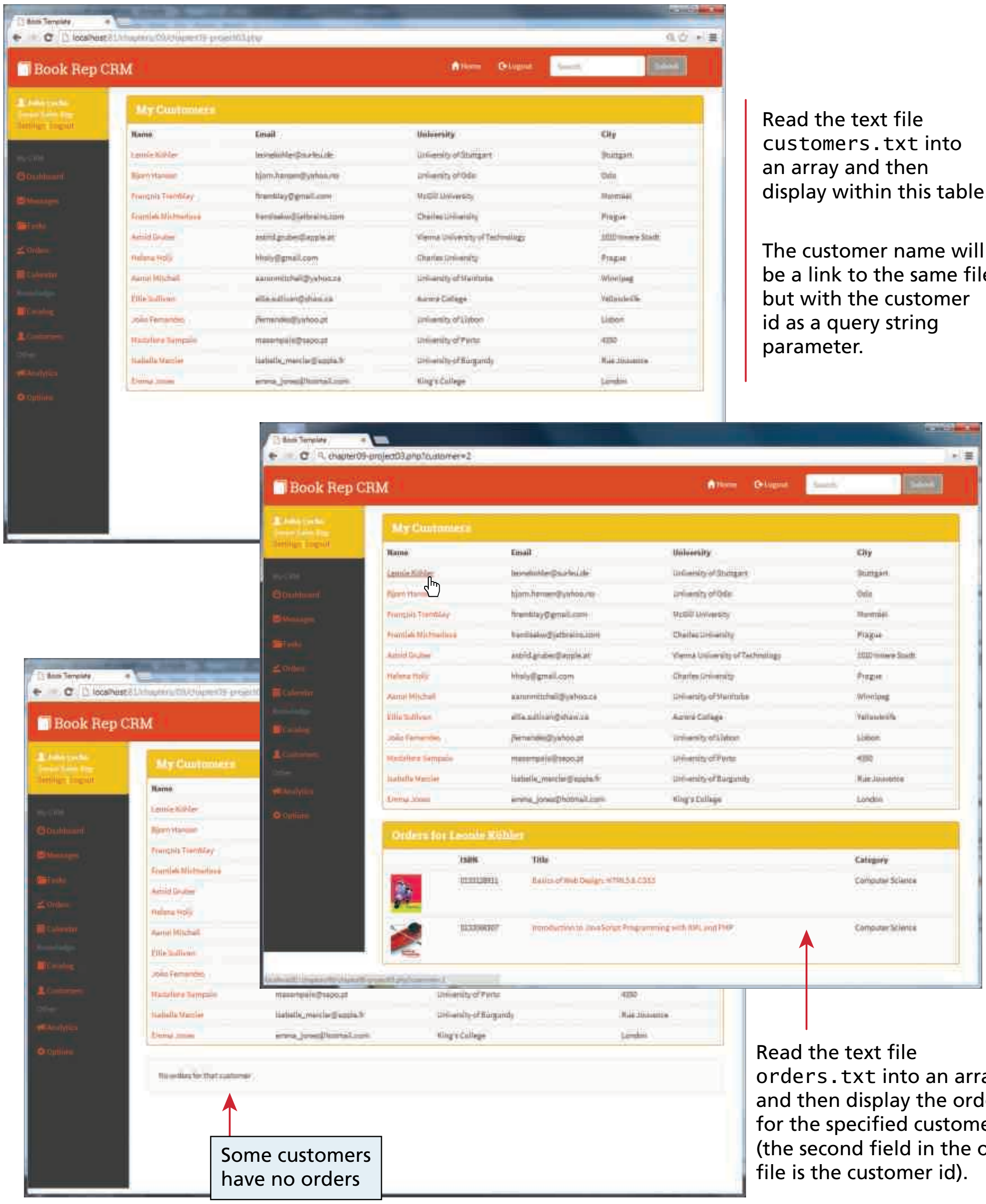
PROJECT 9.3

Overview

Demonstrate your ability to fill arrays from text files and then display the content.

Instructions

1. You have been provided with a PHP file (Chapter09-project03.php) that includes all the necessary markup. You have also been provided with two text files: customers.txt and orders.txt that contain information on customers and on customer's orders.
2. Read the data in customers.txt into an array, and then display the customer data in a table. Each line in the file contains the following information: customer id, first name, last name, email, university, address, city, state, country, zip/postal, phone. You will notice that you are only displaying some of that data.
3. Each customer name must be a link back to Chapter09-project03.php, but with the customer id data as a query string (see Figure 9.15).



My Customers

Name	Email	University	City
Leonie Köhler	leonekohler@surfeu.de	University of Stuttgart	Stuttgart
Bjørn Hansen	bjorn.hansen@yahoo.no	University of Oslo	Oslo
François Tremblay	ftremblay@gmail.com	McGill University	Montréal
František Wichterlová	frantisekw@jetbrains.com	Charles University	Prague
Astrid Gruber	astrid.gruber@apple.at	Vienna University of Technology	1010 Innere Stadt
Helena Holý	hholy@gmail.com	Charles University	Prague
Aaron Mitchell	aaronmitchell@yahoo.ca	University of Manitoba	Winnipeg
Ellie Sullivan	ellie.sullivan@shaw.ca	Aurora College	Yellowknife
João Fernandes	jfernandes@yahoo.pt	University of Lisbon	Lisbon
Madalena Sampaio	masampaio@sapo.pt	University of Porto	4350
Isabelle Mercier	isabelle_mercier@apple.fr	University of Burgundy	Rue Jouvence
Emma Jones	emma_jones@hotmail.com	King's College	London

Orders for Leonie Köhler

	ISBN	Title	Category
	0133128911	Basics of Web Design: HTML5 & CSS3	Computer Science
	0133068307	Introduction to JavaScript Programming with XML and PHP	Computer Science

No orders for that customer

Read the text file `customers.txt` into an array and then display within this table.

The customer name will be a link to the same file but with the customer id as a query string parameter.

Read the text file `orders.txt` into an array, and then display the orders for the specified customer (the second field in the order file is the customer id).

Some customers have no orders

FIGURE 9.15 Completed Project 3

4. When the user clicks on the customer name (that is, makes a request to the same page but with the customer id passed as a query string), then read the data in orders.txt into an array, and then display any matching order data for that customer (see Figure 9.15). Each line in the orders file contains the following data: order id, customer id, book ISBN, book title, and book category. Be sure to display a message when there is no order information for the requested customer.

Test

1. Test the page in the browser. Verify the correct orders are displayed for different customers. Also note that the customer name is displayed in the panel heading for the orders.

9.6.4 References

1. PHP. [Online]. http://ca2.php.net/manual/en/array.sorting.php.
2. PHP. [Online]. http://php.net/manual/en/ref.array.php.
3. PHP. [Online]. http://php.net/manual/en/function.get-browser.php.
4. PHP. [Online]. http://php.net/manual/en/features.file-upload.php.
5. PHP. [Online]. http://php.net/manual/en/features.file-upload.errors.php.

10 PHP Classes and Objects

CHAPTER OBJECTIVES

In this chapter you will learn . . .

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

This chapter begins by introducing object-oriented design principles and practices as applied to server-side development in PHP. You will learn how to create your own classes and how to use them in your pages. The chapter also covers more advanced object-oriented principles, such as derivation, abstraction, and polymorphism, and others will be covered using the Unified Modeling Language (UML), all with the aim of helping you design and develop modular and reusable code.

10.1 Object-Oriented Overview

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

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.

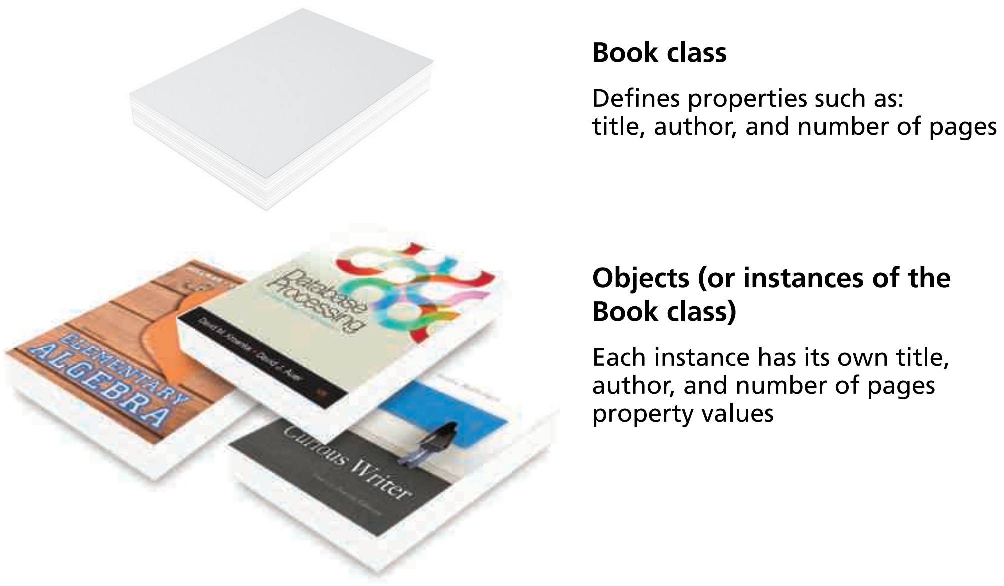

Book class
Defines properties such as: title, author, and number of pages
Objects (or instances of the Book class)
Each instance has its own title, author, and number of pages property values
Database Processing
Elementary Algebra
Curious Writer

FIGURE 10.1 Relationship between a class and its objects

10.1.2 The Unified Modeling Language

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

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

To illustrate classes and objects in UML, consider the artist we have looked at in the Art Case Study. Every artist has a first name, last name, birth date, birth city, and death date. Using objects we can encapsulate those properties together into a class definition for an Artist. Figure 10.2 illustrates a UML class diagram, which shows an `Artist` class and multiple `Artist` objects, each object having its own properties.

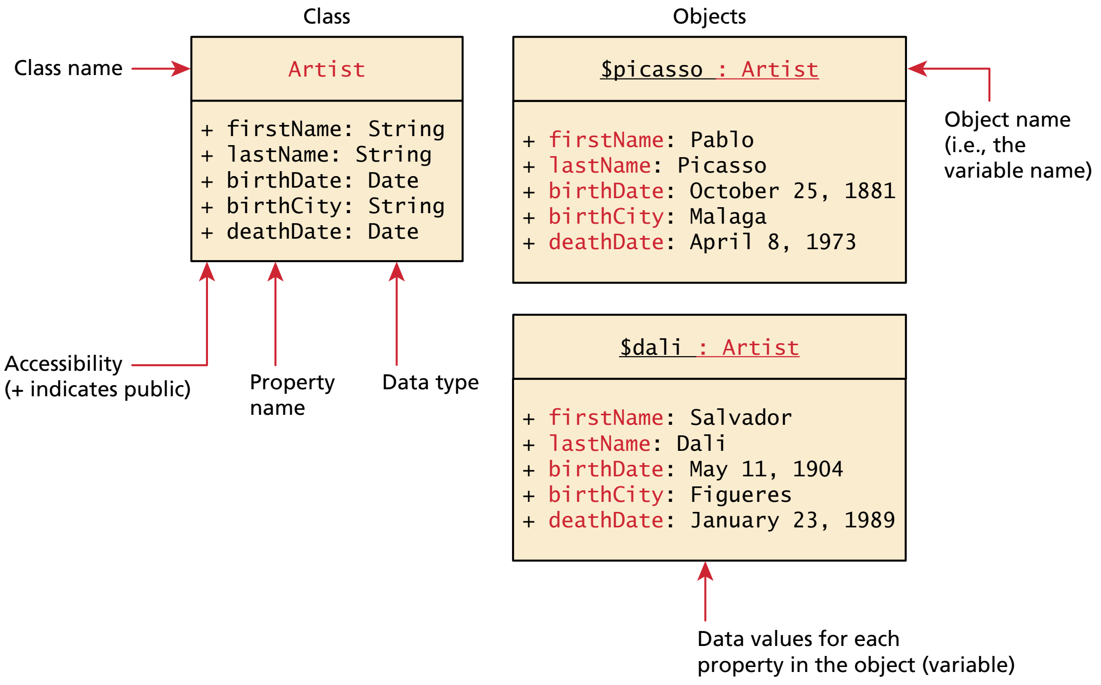


FIGURE 10.2 Relationship between a class and its objects in UML

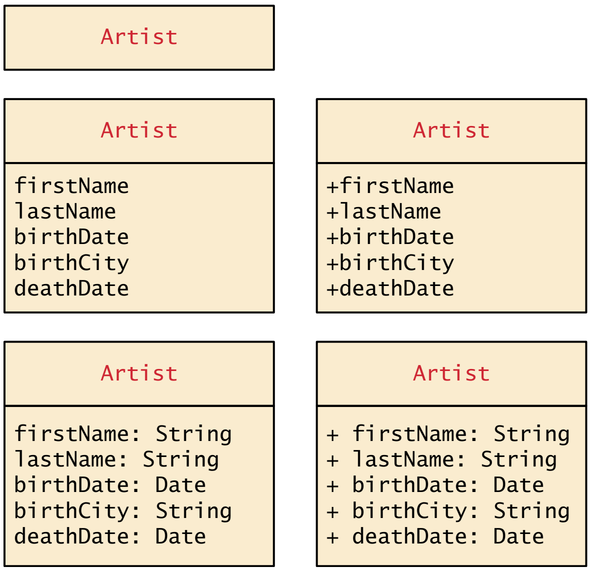


FIGURE 10.3 Different levels of UML detail

In general, when diagramming we are almost always interested in the classes and not so much in the objects. Depending on whether one is interested in showing the big picture, with many classes and their relationships, or showing instead exact details of a class, there is a wide variety of flexibility in how much detail you want to show in your class diagrams, as shown in Figure 10.3.

10.1.3 Differences between Server and Desktop Objects

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

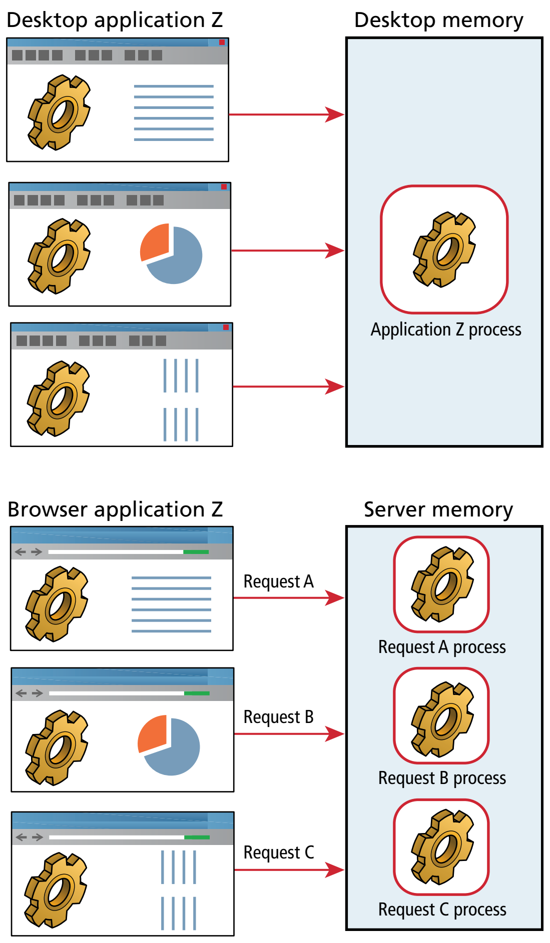


FIGURE 10.4 Lifetime of objects in memory in web versus desktop applications

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

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

10.2 Classes and Objects in PHP

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

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

10.2.1 Defining Classes

The PHP syntax for defining a class uses the class keyword followed by the class name and `{ }` braces.[2] The properties and methods of the class are defined within the braces. The `Artist` class with the properties illustrated in Figure 10.2 is defined using PHP in Listing 10.1.

HANDS-ON EXERCISES

LAB 10 EXERCISE
Define a Class

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

LISTING 10.1 A simple Artist class

NOTE

Prior to version 5 of PHP, the keyword `var` was used to declare a property. From PHP 5.0 to 5.1.3, the use of `var` was considered deprecated and would issue a warning. Since version 5.1.3, it is no longer deprecated and does not issue the warning. If you declare a property using `var`, then PHP 5 will treat the property as if it had been declared as `public`.

Each property in the class is declared using one of the keywords `public`, `protected`, or `private` followed by the property or variable name. The differences between these keywords will be covered in Section 10.2.6.

10.2.2 Instantiating Objects

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

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

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

10.2.3 Properties

Once you have instances of an object, you can access and modify the properties of each one separately using the variable name and an arrow (->), which is constructed from the dash and greater than symbols. Listing 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

HANDS-ON EXERCISES

LAB 10 EXERCISE
Instantiate Objects

While the code in Listing 10.2 works, it takes multiple lines and every line of code introduces potential maintainability problems, especially when we define more artists. Inside of a class definition, you should therefore define constructors, which lets you specify parameters during instantiation to initialize the properties within a class right away.

In PHP, constructors are defined as functions (as you shall see, all methods use the `function` keyword) with the name `__construct()`. (Note: there are *two* underscores _ before the word `construct`.) Listing 10.3 shows an updated `Artist` class definition that now includes a constructor. Notice that in the constructor each parameter is assigned to an internal class variable using the `$this->` syntax. Inside of a class you **must** always use the `$this` syntax to reference all properties and methods associated with this particular instance of a class.

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

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

LISTING 10.3 A constructor added to the class definition

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

This new constructor can then be used when instantiating so that the long code in Listing 10.2 becomes the simpler:

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

10.2.5 Methods

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

PRO TIP

The special function `__construct()` is one of several **magic methods** or magic functions in PHP. This term refers to a variety of reserved method names that begin with two underscores.

These are functions whose interface (but not implementation) is always defined in a class, even if you don't implement them yourself. That is, PHP does not provide the definitions of these magic method; you the programmer must write the code that defines what the magic function will do. They are called by the PHP engine at run time.

The magic methods are: `__construct()`, `__destruct()`, `__call()`, `__callStatic()`, `__get()`, `__set()`, `__isset()`, `__unset()`, `__sleep()`, `__wakeup()`, `__toString()`, `__invoke()`, `__set_state()`, `__clone()`, and `__autoload()`.

with objects. For our artist example one could write a method to convert the artist's details into a string of formatted HTML. Such a method is defined in Listing 10.4.

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

LISTING 10.4 Method definition

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

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

The UML class diagram in Figure 10.2 can now be modified to include the newly defined `outputAsTable()` method as well as the constructor and is shown in Figure 10.5. Notice that two versions of the class are shown in Figure 10.5, to illustrate that there are different ways to indicate a PHP constructor in UML.

NOTE

If a class implements the `__toString()` magic method so that it returns a string, then wherever the object is echoed, it will automatically call `__toString()`. If you renamed your `outputAsTable()` method to `__toString()`, then you could print the HTML table simply by calling:

```
echo $picasso;
```

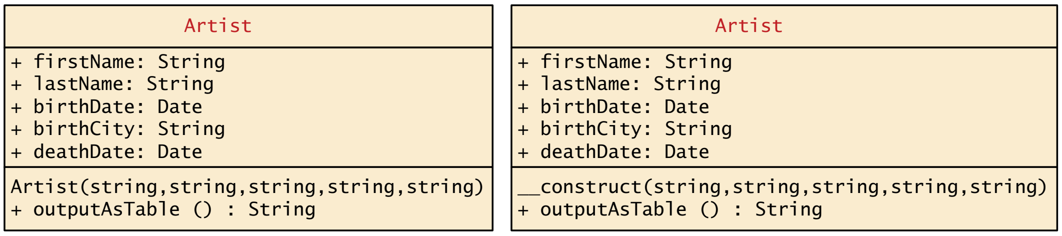


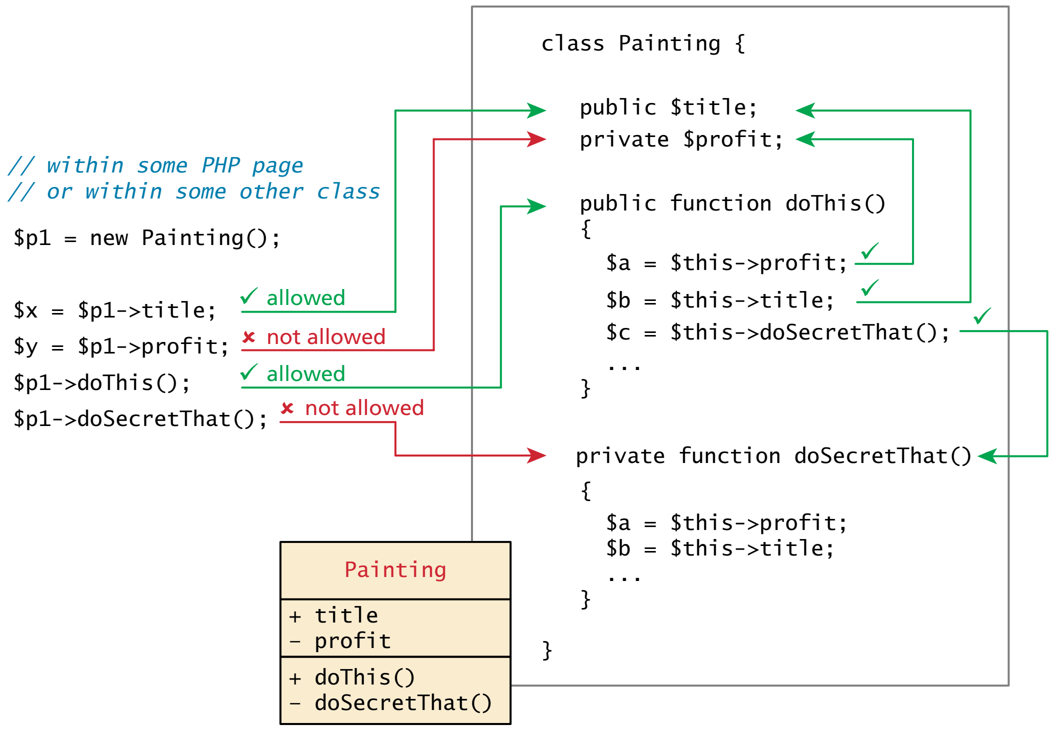
FIGURE 10.5 Updated class diagram

NOTE

Many languages support the concept of overloading a method so that two methods can share the same name, but have different parameters. While PHP has the ability to define default parameters, no method, including the constructor, can be overloaded!

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.



```
// within some PHP page
// or within some other class

$p1 = new Painting();

$x = $p1->title;
$y = $p1->profit;
$p1->doThis();
$p1->doSecretThat();
```

```
class Painting {

   public $title;
   private $profit;

   public function doThis()
   {
      $a = $this->profit;
      $b = $this->title;
      $c = $this->doSecretThat();
      ...
   }

   private function doSecretThat()
   {
      $a = $this->profit;
      $b = $this->title;
      ...
   }
}
```

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. 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 will be discussed later after we cover inheritance. For now consider a protected property or method to be private. In UML, the "+" symbol is used to denote public properties and methods, the "–" symbol for private ones, and the "#" symbol for protected ones.

HANDS-ON EXERCISES

LAB 10 EXERCISE
Add Static Variables

10.2.7 Static Members

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

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

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

For illustrative purposes we will also modify our constructor, so that it increments this value, as shown in Listing 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 you do not reference a static property using the `$this->` syntax, but rather it has its own `self::` syntax. The rationale behind this change is to force the programmer to understand that the variable is static and not associated with an instance (`$this`). This static variable can also be accessed without any instance of an `Artist` object by using the class name, that is, via `Artist::$artistCount`.

To illustrate the impact of these changes look at Figure 10.7, where the shared property is underlined (UML notation) to indicate its static nature and the shared reference between multiple instances is illustrated with arrows, including one reference without any instance.

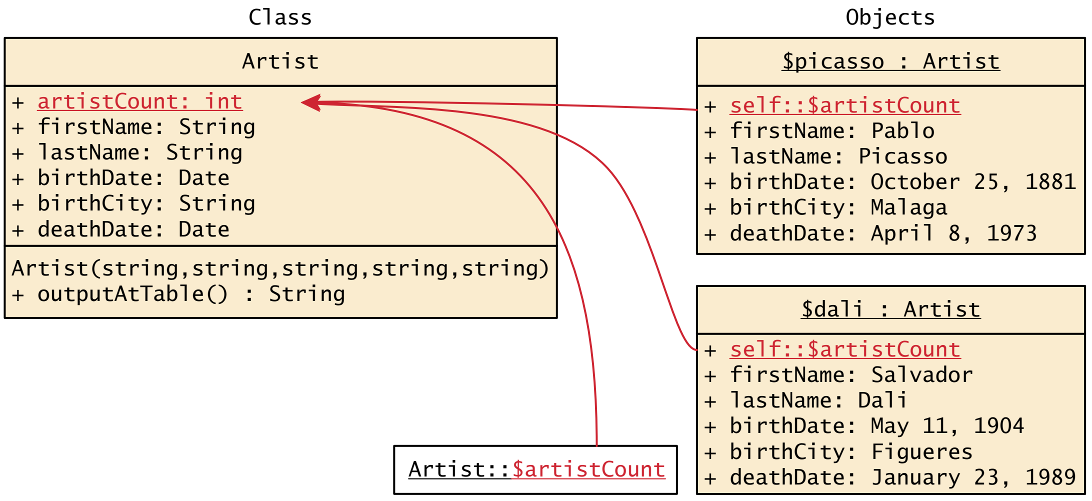


FIGURE 10.7 A static property

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

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

10.2.8 Class Constants

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

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

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

NOTE

Naming conventions can help make your code more understandable to other programmers. They typically involve a set of rules for naming variables, functions, classes, and so on. So far, we have followed the naming convention of beginning PHP variables with a lowercase letter, and using the so-called "camelCase" (that is, begin lowercase, and any new words start with uppercase letter) for functions. You might wonder what conventions to follow with classes.

PHP is an open-source project without an authority providing strong coding convention recommendations as with Microsoft and ASP.NET or Oracle and Java. Nonetheless, if we look at examples within the PHP documentation, and examples in large PHP projects such as PEAR and Zend, we will see four main conventions.

- Class names begin with an uppercase letter and use underscores to separate words (e.g., `Painting_Controller`).
- Public and protected members (properties and methods) use camelCase (e.g., `getSize()`, `$firstName`).
- Constants are all capitals (e.g., `DBNAME`).
- Names should be as descriptive as possible.

In the PEAR documentation and the older Zend documentation, there is an additional convention: namely, that private members begin with an underscore (e.g., `_calculateProfit()`, `$_firstName`). The rationale for doing so is to make it clear when looking for the member name whether the reference is to a public or private member. With the spread of more sophisticated IDE this practice may seem less necessary. Nonetheless, it is a common practice and you may encounter it when working with existing code or examining code examples online.

10.3 Object-Oriented Design

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

10.3.1 Data Encapsulation

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

HANDS-ON EXERCISES

LAB 10 EXERCISE
Data Encapsulation

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

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

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

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

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

```
public function setBirthDate($birthdate){
   // set variable only if passed a valid date string
   $date = date_create($birthdate);

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

Listing 10.6 shows the modified `Artist` class with getters and setters. Notice that the properties are now private. As a result, the code from Listing 10.2 will no longer work for our class since it tries to reference and modify private properties. Instead we would have to use the corresponding getters and setters. Notice as well that two of the setter functions have a fair bit of validation logic in them; this illustrates one of the key advantages to using getters and setters: that the class can handle the responsibility of ensuring its own data validation. And since the setter functions are performing validation, the constructor for the class should use the setter functions to set the values, as shown in this example.

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

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

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

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

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

(*continued*)

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

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

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

LISTING 10.6 Artist class with better encapsulation

PRO TIP

Listing 10.6 uses the more complicated `DateTime` class or its alias method (that is a method, `date_create()`), rather than the simpler and more commonly used `strtotime()` function for converting a string containing a free format date into a Unix timestamp. The drawback to the `strtotime()` function is that it only supports a very constrained year range. On some systems, this means only years between 1970 and 2038, or on some systems between 1900 and 2038. Because the birth and death years of artists can fall before 1900, the example class must make use of the more complicated `DateTime` class.

Two forms of the updated UML class diagram for our data encapsulated class are shown in Figure 10.8. The longer one includes all the getter and setter methods. It is quite common, however, to exclude the getter and setter methods from a class

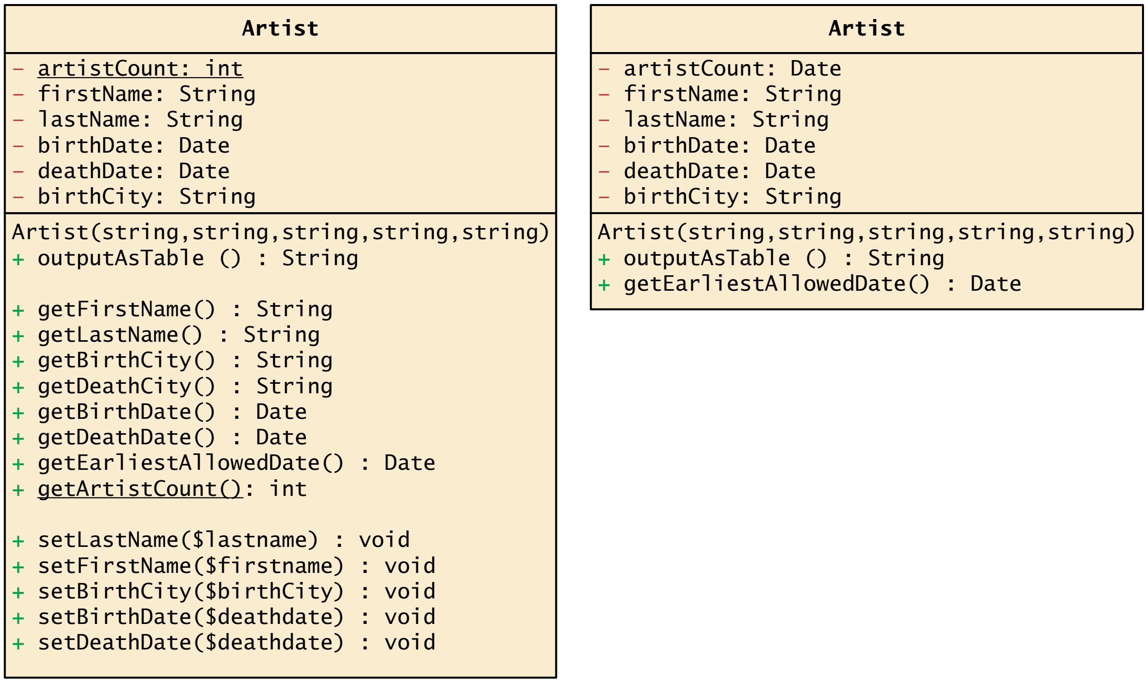

Artist
- artistCount: int
- firstName: String
- lastName: String
- birthDate: Date
- deathDate: Date
- birthCity: String
Artist(string,string,string,string,string)
+ outputAsTable () : String
+ getFirstName() : String
+ getLastName() : String
+ getBirthCity() : String
+ getDeathCity() : String
+ getBirthDate() : Date
+ getDeathDate() : Date
+ getEarliestAllowedDate() : Date
+ getArtistCount(): int
+ setLastName($lastname) : void
+ setFirstName($firstname) : void
+ setBirthCity($birthCity) : void
+ setBirthDate($deathdate) : void
+ setDeathDate($deathdate) : void
Artist
- artistCount: Date
- firstName: String
- lastName: String
- birthDate: Date
- deathDate: Date
- birthCity: String
Artist(string,string,string,string,string)
+ outputAsTable () : String
+ getEarliestAllowedDate() : Date

FIGURE 10.8 Class diagrams for fully encapsulated Artist class

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

Now that the encapsulated `Artist` class is defined, how can one use it? Listing 10.7 demonstrates how the `Artist` class could be used and tested.

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

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

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

(continued)

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

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

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

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

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

?>
</body>
</html>
```

LISTING 10.7 Using the encapsulated class

10.3.2 Inheritance

HANDS-ON EXERCISES

LAB 10 EXERCISE
Inheritance

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

A class that is inheriting from another class is said to be a subclass or a derived class. The class that is being inherited from is typically called a superclass or a base class. When a class inherits from another class, it inherits all of its public and protected methods and properties. Figure 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 { . . . }
```

Referencing Base Class Members

As mentioned above, a subclass inherits the public and protected members of the base class. Thus in the following code based on Figure 10.9, both of the references

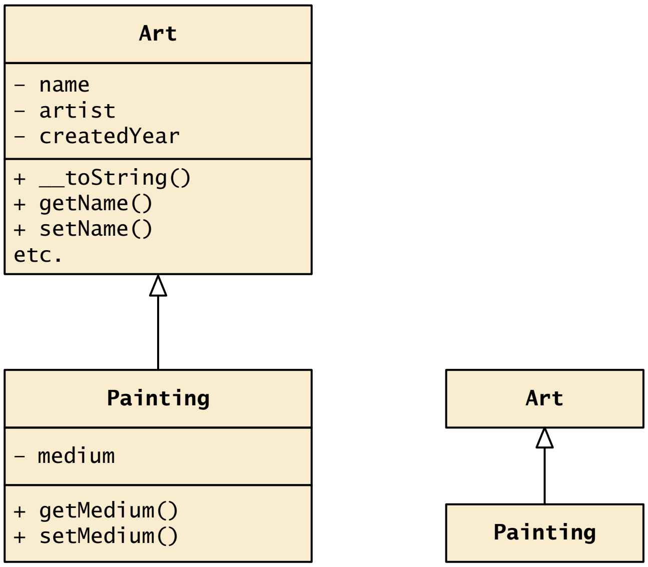


FIGURE 10.9 UML class diagrams showing inheritance

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

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

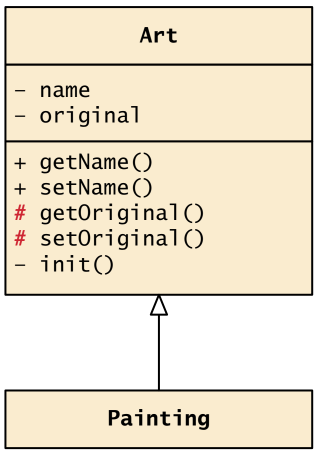
Unlike in languages like Java or C#, in PHP any reference to a member in the base class requires the addition of the `parent::` prefix instead of the `$this->` prefix. So within the `Painting` class, a reference to the `getName()` method would be:

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

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

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

If you want a member to be available to subclasses but not anywhere else, you can use the `protected` access modifier, which is shown in Figure 10.10.



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

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

```
  // in some page or other class
  $p = new Painting();
  $a = new Art();

  // neither of these references are allowed
✗ $w = $p->getOriginal();
✗ $y = $a->getOriginal();
```

FIGURE 10.10 Protected access modifier

To best see the potential benefits of inheritance, let us look at a slightly *extended* example involving different types of art. For our previously defined `Artist` class, imagine we include a list of works of art for each artist. We might manage that list inside the class with an array of objects of type `Art`. Such a list must allow objects of many types, for what is art after all? We can have music works, paintings, writings, sculptures, prints, inventions, and more, all considered `Art`. We will therefore use the idea of art as the basis for demonstrating inheritance in PHP. Figure 10.11 shows the relationship of the classes in our example.

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

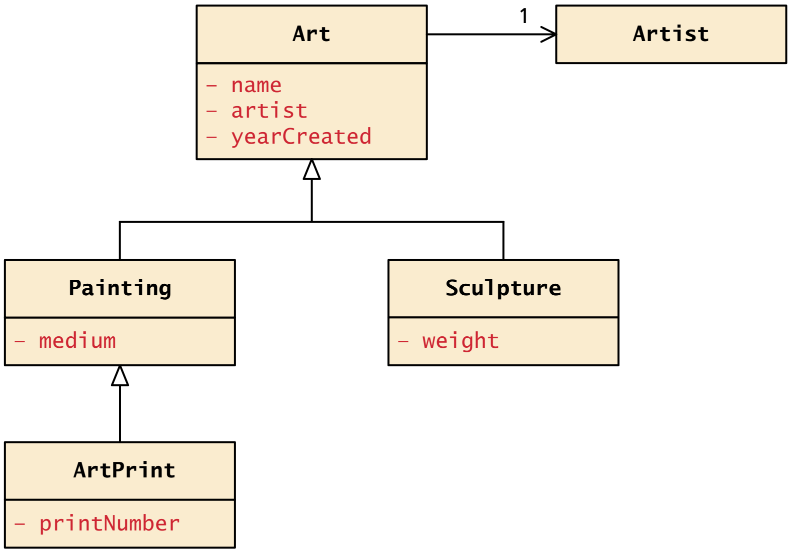


FIGURE 10.11 Class diagram for Art example

Listing 10.8 lists the implementation of these four classes. Notice how the subclass constructors invoke the constructors of their base class and that many of the setter methods are performing some type of validation. Notice as well the use of the `abstract` keyword in the first line of the definition of the `Art` class. An abstract class is one that cannot be instantiated. In the context of art, there can be concrete types of art, such as paintings, sculpture, or prints, but not "art" in general, so it makes sense to programmatically model this limitation via the `abstract` keyword.

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

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

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

(continued)

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

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

class Painting extends Art {
   private $medium;

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

class Sculpture extends Art {
   private $weight;

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

class ArtPrint extends Painting {
   private $printNumber;

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

LISTING 10.8 Class implementations for Listing 10.11

Whenever you create classes, you will eventually need to use them. The authors often find it useful to create tester pages that verify a class works as expected. Listing 10.9 illustrates a typical tester. Notice that since the Art class has a data member of type Artist, it is possible to also access the Artist properties through the Art object.

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

(continued)

```
// instantiate some sample objects
$picasso = new Artist("Pablo","Picasso","Malaga","May 11,904",
                      "Apr 8, 1973");
$guernica = new Painting("1937",$picasso,"Guernica","Oil on
                         canvas");
$stein = new Painting("1907",$picasso,"Portrait of Gertrude Stein",
                      "Oil on canvas");
$woman = new Sculpture("1909",$picasso,"Head of a Woman", 30.5);
$bowl = new ArtPrint("1912",$picasso,"Still Life with Bowl and Fruit",
                    "Charcoal on paper", 25);
?>
<html>
<body>
<h1>Tester for Art Classes</h1>

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

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

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

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

LISTING 10.9 Using the classes

Inheriting Methods

Every method defined in the base/parent class can be overridden when extending a class, by declaring a function with the same name. A simple example of overriding can be found in Listing 10.8 in which each subclass overrides the `__toString()` method.

To access a public or protected method or property defined within a base classfrom 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.

10.3.3 Polymorphism

HANDS-ON EXERCISES

LAB 10 EXERCISE
Iterating Polymorphic Objects

Polymorphism is the third key object-oriented concept (along with encapsulation and inheritance). In the inheritance example in Listing 10.8, the classes `Sculpture` and `Painting` inherited from `Art`. Conceptually, a sculpture *is a* work of art and a painting *is a* work of art. Polymorphism is the notion that an object can in fact be multiple things at the same time. Let us begin with an instance of a `Painting` object named `$guernica` created as follows:

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

The variable `$guernica` is both a `Painting` object and an `Art` object due to its inheritance. The advantage of polymorphism is that we can manage a list of `Art` objects, and call the same overridden method on each. Listing 10.10 illustrates polymorphism at work.

```
$picasso = new Artist("Pablo","Picasso","Malaga","Oct 25, 1881",
                     "Apr 8, 1973");

// create the paintings
$guernica = new Painting("1937",$picasso,"Guernica","Oil on canvas");
$chicago = new Sculpture("1967",$picasso,"Chicago", 454);
```

(continued)

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

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

LISTING 10.10 Using polymorphism

Due to overriding methods in child classes, the actual method called will depend on the type of the object! Using `__toString()` as an example, a `Painting` will output its name, date, and medium and a `Sculpture` will output its name, date, and weight. The code in Listing 10.10 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.

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. The class infrastructure that will be defined in Chapter 14 makes use of interfaces.

HANDS-ON EXERCISES

LAB 10 EXERCISE
Using Interfaces

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

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

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

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

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

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

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

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

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

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

LISTING 10.11 Painting class implementing an interface

Listing 10.12 defines another interface (`Playable`), and then two classes that use it.

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

class Music extends Art implements Playable {
   ...
   public function getMedia() {
     //returns the music
      ...
   }
   public function getLength() {
      //return the length of the music
   }
}
class Movie extends Painting implements Playable, Viewable {
   ...
   public function getMedia() {
      //return the movie
      ...
   }
```

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

LISTING 10.12 Playable interface and multiple interface implementations

While PHP prevents us from inheriting from two classes, it does not prevent us from implementing two or more interfaces. The `Movie` class therefore extends from `Painting` but also implements the two interfaces `Viewable` and `Playable`. The diagram illustrating this relationship in UML is shown in Figure 10.12. In UML, interfaces are denoted through the `<<interface>>` stereotype. Classes that implement an interface are shown to implement using the same hollow triangles as inheritance but with dotted lines.

Runtime Class and Interface Determination

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

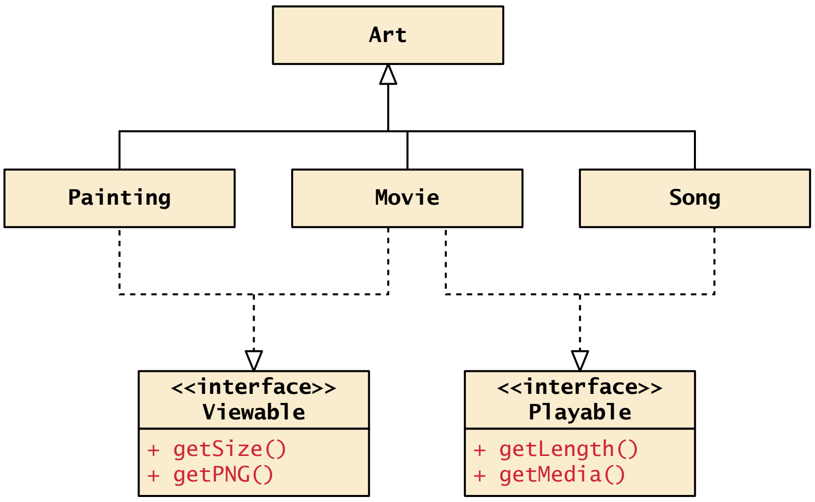


FIGURE 10.12 Indicating interfaces in a class diagram

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

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

Similarly we can access the parent class with:

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

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

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

PRO TIP

As of PHP 5.3.2 there is a new mechanism called traits, which can be thought of as interfaces with code (rather than just signatures). These traits can be added to any class like a block of code pasted in, but do not affect the class relationship like inheritance or interface implementation does.[3] In this book we will not use traits, because of their odd behavior when used with other mechanisms.

10.4 Chapter Summary

In this chapter, we have covered what is a vital topic in modern-day programming, namely, how to do object-oriented programming in PHP. While it is possible to work with PHP without using classes and objects, their use industry-wide is evidence of their ability to generate more modular, reusable, and maintainable code. PHP programmers can benefit from these experiences by also using these object-oriented techniques, thereby improving the maintainability and portability of their web applications.

10.4.1 Key Terms

base class	class member	data members
class	constructor	derived class

dynamic dispatching	magic method	subclass
encapsulation	methods	superclass
getters and setters	naming conventions	UML (Unified Modeling Language)
inheritance	object	visibility
instance	polymorphism	
instantiate	properties	
interface	static	

10.4.2 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.

10.4.3 Hands-On Practice

PROJECT 1: Share Your Travel Photos

DIFFICULTY LEVEL: Intermediate

HANDS-ON EXERCISES

PROJECT 10.1

Overview

This exercise walks you through the usage of a static class variable, and simple data encapsulation. It builds on the structure you have from Chapter 9 Project 2, but replaces arrays of arrays with a single array of objects of type `TravelImage`.

Instructions

1. Create a file named TravelPhoto.class.php and within it define a class named `TravelPhoto`, which has private properties: `date`, `fileName`, `description`, `title`, `latitude`, `longitude`, and `ID`.
2. Define a static member variable named `photoID`, which will be used to set each instance's ID value and then be incremented, all inside the class constructor.
3. Create a constructor that takes in `fileName`, `title`, `description`, `latitude`, and `longitude`.
4. Implement the `__toString()` method that should return the HTML markup for an `<img>` element for the member data within this object. This `<img>` element should also have `alt` and `title` attributes set to the value of the object's `title` property.

5. Open travel-data-classes.php. Notice that it contains instantiations of `TravelPhoto` objects inside an array.
6. Modify your Chapter09-project02.php to use the array of objects within travel-data-classes.php rather than the data in travel-data.php. Hint: Use your new `__toString()` method.

Testing

1. Open your script in a browser to see the output. You should see output identical to that in Figure 9.14.
2. Hover over the image to ensure the title attribute of each image is set.
3. Clicking the link will still take you to travel-image.php with the `id` element passed as a query string parameter.

PROJECT 2: Share Your Travel Photos

DIFFICULTY LEVEL: Intermediate

HANDS-ON EXERCISES

PROJECT 10.2

Overview

This exercise builds on the last one by improving the design to be more modular and less coupled. In particular we will guide you on separating the `Location` out from the `TravelPhoto` class. The files from Project 1 will be used as a starting point for this project.

Instructions

1. Define a new class, `Location`, inside of a new file named Location.class.php. Make the constructor take three parameters: a `latitude`, `longitude`, and a `city code`.
2. Modify the `TravelPhoto` class to store an instance of a `Location`, rather than the latitude and longitude. You may need to modify small pieces of code throughout to account for the change. Hint: Create the new `Location` object in the constructor of `TravelPhoto`.
3. Write a function that given one instance of TravelPhoto, finds the nearest travel photo in the array of `TravelPhoto` objects. Hint: Compare the latitude and longitude values.
4. Modify the travel-image.php detail page to output a link to the nearest image underneath the main photo.

Testing

1. Ensure the site still looks the same, despite making better use of objects.
2. To confirm that your location proximity function works correctly, input several proposed "nearest" locations into a map to visually confirm that the photos are in fact close to one another.

PROJECT 3: Book Rep Customer Relations Management

DIFFICULTY LEVEL: Intermediate

HANDS-ON EXERCISES

PROJECT 10.3

Overview

Demonstrate your ability to instantiate classes from text files and then display the content. This project has output identical to Chapter 9 Project 3.

Instructions

1. You have been provided with a PHP file (Chapter10-project03.php) that includes all the necessary markup. You have also been provided with two text files, customers.txt and orders.txt, that contain information on customers and their orders. (These files are the same as files from Chapter 9 Project 3.)
2. Define classes to encapsulate the data of a `Customer` and an `Order`. Each line in the file contains the following information: customer id, first name, last name, email, university, address, city, state, country, zip/postal, phone. Each line in the orders file contains the following data: order id, customer id, book ISBN, book title, book category.
3. Read the data in customers.txt and for each line in that file create a new instance of `Customer` in an array, and then display the customer data in a table.
4. Each customer name must be a link back to Chapter10-project03.php but with the customer id data as a query string.
5. When the user clicks on the customer name (i.e., makes a request to the same page but with the customer id passed as a query string), then read the data in orders.txt into an array of `Order` objects, and then display any matching order data for that customer. Be sure to display a message when there is no order information for the requested customer.

Test

1. Test the page in the browser. Verify the correct orders are displayed for different customers. Also note that the customer name is displayed in the panel heading for the orders.
2. Try writing a `print_r()` statement to output the structure of all `Customer` and `Order` objects and verify they match the data in the files.

10.4.4 References

1. Open Modelling Group, “OMG® Specifications.” [Online]. http://www.omg.org/spec/.
2. PHP, “Classes and Objects.” [Online]. http://php.net/manual/en/language.oop5.php.
3. PHP, “Traits.” [Online]. http://php.net/manual/en/language.oop5.traits.php.

Error Handling and Validation

12

CHAPTER OBJECTIVES

In this chapter you will learn . . .

- What the different types of errors are and how they differ from exceptions
- 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

This chapter covers one of the most vital topics in web application development: how to prevent and deal with unexpected errors. Even the best-written application may fail. Whether it is due to strange user input, the failure of a remote service, or simple programmer oversight, errors and exceptions happen. Constructing a web application that can handle exceptions gracefully and meaningfully requires some additional approaches to those used in desktop applications. PHP provides both language-level and function-level mechanisms for helping the developer to construct useful exception handling.

12.1 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.

12.1.1 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

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. If you plan on remembering only one thing from this chapter, it should be this: 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.

NOTE

Remember that user input is not limited to data entry forms: query strings attached to hyperlinks (as well as cookies, which are covered in Chapter 13) are also a type of user input, and your application should be able to handle the user modifying and messing with query string parameter names and values. Your PHP code should *always* check query string parameters for acceptable values.

So how should you deal with expected errors with user inputs? You will need some type of logic that verifies that first, the user input exists and second, it contains the expected values.

Notice that this parameter has no value.

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

This parameter's value is a space character (URL encoded).

`isset($_GET['id'])`	returns	**`true`**	
`isset($_GET['name1'])`	returns	**`true`**	Notice that a missing value for a parameter is still considered to be `isset`.
`isset($_GET['name2'])`	returns	**`true`**	
`isset($_GET['name3'])`	returns	**`true`**	
`isset($_GET['name4'])`	returns	**`false`**	Notice that only a missing parameter name is considered to be not `isset`.
`empty($_GET['id'])`	returns	**`true`**	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.
`empty($_GET['name1'])`	returns	**`true`**	
`empty($_GET['name2'])`	returns	**`false`**	
`empty($_GET['name3'])`	returns	**`false`**	Notice that a value of space is considered to be **not** empty.
`empty($_GET['name4'])`	returns	**`true`**	

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

PHP provides two functions for testing the value of a variable. You have already encountered `isset()`, which returns `true` if a variable is not null. However, `isset()` by itself does not provide enough error checking. Generally a better choice for checking query string values is the `empty()` function, which returns `true` if a variable is null, `false`, zero, or an empty string. Figure 12.1 illustrates how these functions differ.

If you are expecting a query string parameter to be numeric, then you 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

There are many other checks that a page might make to test that a user's input is in the correct format. We will explore several of these in depth after you have learned more about regular expressions in Section 12.4.

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.

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.

12.1.2 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. Section 12.3 covers exception handling in more detail.

12.2 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.[1] There are three main error reporting flags:

- `error_reporting`
- `display_errors`
- `log_errors`

The meaning of each of these is important and should be learned by PHP developers.

12.2.1 The error_reporting Setting

The `error_reporting` setting specifies which type of errors are to be reported.[1] It can be set programmatically inside any PHP file by using the `error_reporting()` function:

```
error_reporting(E_ALL);
```

HANDS-ON EXERCISES

LAB 12 EXERCISE
Turn on Reporting

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. It is worth noting that in some PHP environments, the default setting is zero, that is, no reporting.

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');
```

HANDS-ON EXERCISES

LAB 12 EXERCISE
Display Errors

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

```
display_errors = Off
```

NOTE

Error and warning messages are quite helpful for programmers trying to debug problems. However, they should **never** be displayed to the end user. Not only are they unhelpful for end users, but these messages can be a security risk as they may provide information that can be useful to someone trying to find attack vectors into a system.

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

12.2.3 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
```

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
```

NOTE

It is **strongly advised** to turn on error logging for production sites. In fact, because warning messages might not always be visible in the browser, it is recommended to turn on error logging also while an application is in development mode as well.

HANDS-ON EXERCISES

LAB 12 EXERCISE
Tail Your Logs

You can also programmatically send messages to the error log at any time via the `error_log()` function.[3] 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');
```

As you can see, this function has the added advantage of being able to email error messages.

12.3 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 and the more object-oriented 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. For instance, in Chapter 11 you learned how to use the 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 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

While this approach might seem more straightforward, it does require the programmer to know ahead of time what code is going to generate an error condition. As well, it might result in a great deal of code duplication. The advantage of the `try...catch` mechanism is that it allows the developer to handle a wider variety of exceptions in a single catch block.

Yet, even with explicit testing for error conditions, there will still be situations when an unforeseen error occurs. In such a case, unless a custom error handler has been defined, PHP will terminate the execution of the application. Custom error handlers are covered below in Section 12.3.3.

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.[4]

Like other object-oriented programming languages, 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 similar to that you have already seen in Chapter 11. 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

The `finally` block is optional. Any code within it will always be executed *after* the code in the `try` or in the `catch` blocks, even if that code contains a `return` statement. It is typically used if the developer wants certain things done regardless of whether an exception occurred, such as closing a connection or removing temporary files. However, the `finally` block is only available in PHP 5.5 and later, which was released in June 2013.

It is also possible in PHP to programmatically throw an exception via the `throw` keyword, as shown in Listing 12.4.

Why would you throw an exception? If you are, for instance, creating functions that are general purpose and to be used in a variety of contexts that you have no control over, it might make sense to throw an exception when an expected programming assumption is not met. Listing 12.4 is an example of this use.

```
function processArray($array)
{
   // make sure the passed parameter is an array with values
   if ( empty($array) ) {
      throw new Exception('Array with values expected');
   }
   // process the array code
   ...
}
```

LISTING 12.4 Throwing an exception

Do you remember the brief discussion in Chapter 10 on what to do in a class setter method in which the input parameter was invalid (e.g., `setBirthDate()` in Section 10.3.1)? One possible strategy for such a scenario is to throw an exception:

```
public function setBirthDate($birthdate){
   // set variable only if passed a valid date string
   if ( $timestamp = strtotime($birthdate) ) {
      $this->birthDate=$timestamp;
   }
   else {
     throw new Exception("Invalid Date in Artist->setBirthDate()");
   }
}
```

It might also make sense to rethrow an exception within a `catch` block. For instance, you may want to do some application-specific handling of the exception and then pass it on to the PHP environment (or some other intermediary). Listing 12.5 illustrates an example of rethrowing. Notice that it does not create a new exception as in Listing 12.4 but throws the original exception.

```
try {
  // PHP code here
}
catch (Exception $e) {
    // do some application-specific exception handling here
    ...
    // now rethrow exception
    throw $e;
}
```

LISTING 12.5 Rethrowing an exception

NOTE

Warnings in PHP do **not** generate a runtime exception and hence cannot be caught.

12.3.3 Custom Error and Exception Handlers

HANDS-ON EXERCISES

LAB 12 EXERCISE
Custom Error Handlers

When a web application is in development, one can generally be content with displaying and/or logging error messages and then terminating the script. But for production applications, you will likely want to handle significant errors in a better way. It is possible to define your own handler for uncaught errors and exceptions; the mechanism for doing so varies depending upon whether you are using the procedural or object-oriented mechanism for responding to errors.

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

What should a custom error or exception handler do? It should provide the *developer* with detailed information about the state of the application when the exception occurred, information about the exception, and when it happened. It should hide any of those details from the regular end user, and instead provide the user with a generic message such as "Sorry but there was a problem," or even better perhaps from a security standpoint, "Sorry but the system is down for maintenance." Why might the latter, less descriptive message be better? Because it doesn't let a potential malicious user know that he or she did something that caused a problem. 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. Their history usage goes back further, including the formal specification defined by the IEEE POSIX standard.[5]

HANDS-ON EXERCISES

LAB 12 EXERCISE
Getting Started with Regex

Regular expressions are a concise way to eliminate the conditional logic that would be necessary to ensure that input data follows a specific format. Consider a postal code: in Canada a postal code is a letter, followed by a digit, followed by a letter, followed by an optional space or dash, followed by number, letter, and number. Using `if` statements, this would require many nested conditionals (or a single `if` with a very complex expression). But using regular expressions, this pattern check can be done using a single concise function call.

PHP, JavaScript, Java, the .NET environment, and most other modern languages support regular expressions. They do use different regular expression engines which operate in different ways, so not all regular expressions will work the same in all environments.

12.4.1 Regular Expression Syntax

A regular expression consists of two types of characters: literals and metacharacters. A literal is just a character you wish to match in the target (i.e., the text that you

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

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

are searching within). 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, you will need to *escape* it by prefacing it with a backslash (\). Table 12.3 lists examples of typical metacharacter usage to create patterns a typical regular expression is made up of several patterns.

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

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

It should be noted that regular expression pattern checks are case sensitive.

Regular expressions can be complicated to visually decode; to help, this section will use the convention of alternating between red and blue to indicate distinct sub-patterns in an expression and black text for literals.

This regular expression will find matches in all three of the following strings.

```
'randy connolly'
'Sue ran to the store'
'I would like a cranberry'
```

To perform the pattern check in PHP, you would write something similar to the following:

```
$pattern = '/ran/';
$check = 'Sue ran to the store';
if ( preg_match($pattern, $check) ) {
  echo 'Match found!';
}
```

To perform the same pattern check in JavaScript, you would write something similar to the following:

```
var pattern = /ran/;
if ( pattern.test('Sue ran to the store') ) {
  document.write('Match found!');
}
```

In JavaScript a regular expression is its own data type. Just as a string literal begins and ends with quote characters, in JavaScript, a regular expression literal begins and ends with forward slashes.

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

12.4.2 Extended Example

HANDS-ON EXERCISES

LAB 12 EXERCISE
Advanced Regular Expressions

Perhaps the best way to understand regular expressions is to work through the creation of one. For instance, if we wished to define a regular expression that would match a North American phone number without the area code, we would need one that matches any string that contains three numbers, followed by a dash, followed by four numbers without any other character. The regular expression for this would be:

```
^\d{3}-\d{4}$
```

While this may look quite intimidating at first, it is in reality a fairly straightforward regular expression. In this example, the dash is a literal character; the rest are all metacharacters. The `^` and `$` symbol indicate the beginning and end of the string, respectively; they indicate that the entire string (and not a substring) can only contain that specified by the rest of the metacharacters. The metacharacter `\d` indicates a digit, while the metacharacters `{3}` and `{4}` indicate three and four repetitions of the previous match (i.e., a digit), respectively.

A more sophisticated regular expression for a phone number would not allow the first digit in the phone number to be a zero ("0") or a one ("1"). The modified regular expression for this would be:

```
^[2-9]\d{2}-\d{4}$
```

The `[2-9]` metacharacter indicates that the first character must be a digit within the range 2 through 9.

We can make our regular expression a bit more flexible by allowing either a single space (440 6061), a period (440.6061), or a dash (440-6061) between the two sets of numbers. We can do this via the `[]` metacharacter:

```
^[2-9]\d{2}[-\s\.]\d{4}$
```

This expression indicates that the fourth character in the input must match one of the three characters contained within the square brackets (- matches a dash, `\s` matches a white space, and `\.` matches a period). We must use the escape character for the dash and period, since they have a metacharacter meaning when used within the square brackets.

If we want to allow multiple spaces (but only a single dash or period) in our phone, we can modify the regular expression as follows.

```
^[2-9]\d{2}[-\s\.]\s*\d{4}$
```

The metacharacter sequence `\s*` matches zero or more white spaces. We can further extend the regular expression by adding an area code. This will be a bit more complicated, since we will also allow the area code to be surrounded by brackets (e.g., (403) 440-6061), or separated by spaces (e.g., 403 440 6061), a dash (e.g., 403-440-6061), or a period (e.g., 403.440.6061). The regular expression for this would be:

```
^\(?\s*\d{3}\s*[\)-\.]?\s*[2-9]\d{2}\s*[-\.]\s*\d{4}$
```

The modified expression now matches zero or one "(" characters (\(?), followed by zero or more spaces (\s*), followed by three digits (\d{3}), followed by zero or more spaces (\s*), followed by either a ")" a "-", or a "." character ([\)-\.]?), finally followed by zero or more spaces (\s*).

Finally, we may want to make the area code optional. To do this, we will group the area code by surrounding the area code subexpression within grouping metacharacters—which are "(" and ")"—and then make the group optional using the ? metacharacter. The resulting regular expression would now be:

```
^(\(?\s*\d{3}\s*[\)-\.]?\s*)?[2-9]\d{2}\s*[-\.]\s*\d{4}$
```

While this regular expression does look frightening, when you compare the efficiency of making this check via a single line of code in comparison to the many lines of code via conditionals, you quickly see the benefit of regular expressions. To illustrate, consider the lengthy JavaScript code in Listing 12.7, which validates a phone number using only conditional logic. Needless to say, the regular expression is far more succinct!

Hopefully by now you are able to see that many web applications could potentially benefit from regular expressions. Table 12.4 contains several common regular expressions that you might use within a web application. Many more common regular expressions can easily be found on the web.

```
var phone=document.getElementById("phone").value;
var parts = phone.split(".");                // split on .
if (parts.length !=3){
   parts = phone.split("-");                 // split on -
}
if (parts.length == 3) {
   var valid=true;                     // use a flag to track validity
   for (var i=0; i < parts.length; i++) {
      // check that each component is a number
      if (!isNumeric(parts[i])) {
         alert( "you have a non-numeric component");
         valid=false;
      } else { // depending on which component make sure it's in range
         if (i<2) {
            if (parts[i]<100 || parts[i]>999) {
               valid=false;
            }
         }
```

(*continued*)

```
         else {
            if (parts[i]<1000 || parts[i]>9999) {
               valid=false;
            }
         }
      } // end if isNumeric
  } // end for loop
  if (valid) {
     alert(phone + "is a valid phone number");
  }
}
alert ("not a valid phone number");
```

LISTING 12.7 A phone number validation script without regular expressions

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

PRO TIP

MySQL also supports regular expressions through the REGEXP operator (or the alternative RLIKE operator, which has the identical functionality). This operator provides a more powerful alternative to the regular SQL LIKE operator (though it doesn't support all the normal regular expression metacharacters). For instance, the following SQL statement matches all art works whose title contains one or more numeric digits:

```
SELECT * FROM ArtWorks WHERE Title REGEXP '[0-9]+'
```

While MySQL regular expressions provide opportunities for powerful text-matching queries, it should be remembered that these queries do not make use of indices so the use of regular expressions can be relatively slow when querying large tables.

12.5 Validating User Input

As mentioned several times already, user input must always be tested for validity. But what types of validity checks should a form be making? How should we notify the user?

12.5.1 Types of Input Validation

The following list indicates most of the common types of user input validation.

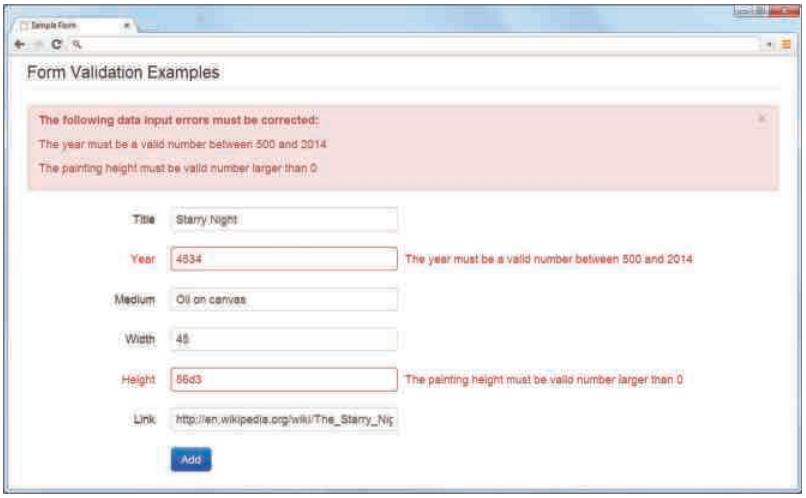
- **Required information.** Some data fields just cannot be left empty. For instance, the principal name of things or people is usually a required field. Other fields such as emails, phones, or passwords are typically required values.
- **Correct data type.** While some input fields can contain any type of data, other fields, such as numbers or dates, must follow the rules for its data type in order to be considered valid.
- **Correct format.** Some information, such as postal codes, credit card numbers, and social security numbers have to follow certain pattern rules. It is possible, however, to go overboard with these types of checks. Try to make life easier for the user by making user input forgiving. For instance, it is an easy matter for your program to strip out any spaces that users entered in their credit card numbers, which is a better alternative to displaying an error message when the user enters spaces into the credit card number.

- **Comparison**. Some user-entered fields are considered correct or not in relation to an already-inputted value. Perhaps the most common example of this type of validation is entering passwords: most sites require the user to enter the password twice and then a comparison is made to ensure the two entered values are identical. Other forms might require a value to be larger or smaller than some other value (this is common with date fields).
- **Range check**. Information such as numbers and dates have infinite possible values. However, most systems need numbers and dates to fall within realistic ranges. For instance, if you are asking a user to input her birthday, it is likely you do not want to accept January 1, 214 as a value; it is quite unlikely she is 1800 years old! As a result, almost every number or date should have some type of range check performed.
- **Custom**. Some validations are more complex and are unique to a particular application. Some custom validations can be performed on the client side. For instance, the author once worked on a project in which the user had to enter an email (i.e., it was required), unless the user entered both a phone number and a last name. This required multiple conditional validation logic. Other custom validations require information on the server. Perhaps the most common example is user registration forms that will ensure that the user doesn't enter a login name or email that already exists in the system.

12.5.2 Notifying the User

What should your pages do when a validation check fails? Clearly the user needs to be notified . . . but how? Most user validation problems need to answer the following questions:

- **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. These messages can be gathered together in one group and presented near the top of a page and/or beside the fields that generated the errors. Figure 12.2 illustrates both approaches.
- **Where is the problem?** Some type of error indication should be located near the field that generated the problem. Some sites will do this by changing the background color of the input field, or by placing an asterisk or even the error message itself next to the problem field. Figure 12.3 illustrates the latter approach.
- **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 you are expecting, such as "The date should be in yy/mm/dd format."


Form Validation Examples

The following data input errors must be corrected:
The year must be a valid number between 500 and 2014
The painting height must be valid number larger than 0

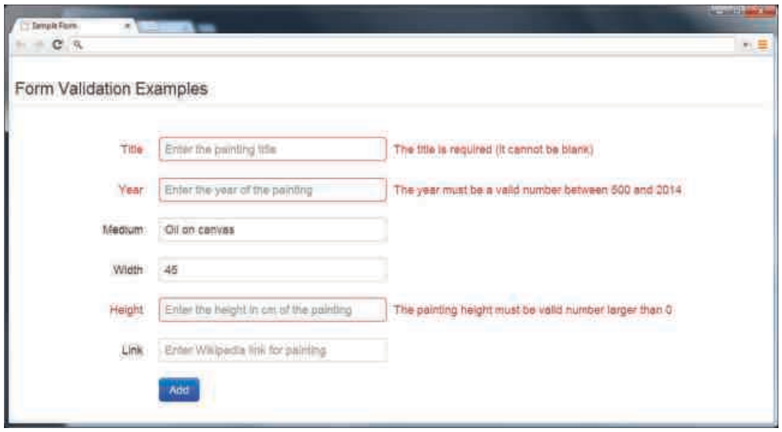
Title	Starry Night	
Year	4634	The year must be a valid number between 500 and 2014
Medium	Oil on canvas	
Width	45	
Height	56d3	The painting height must be valid number larger than 0
Link	http://en.wikipedia.org/wiki/The_Starry_Nig	

FIGURE 12.2 Displaying error messages

12.5.3 How to Reduce Validation Errors

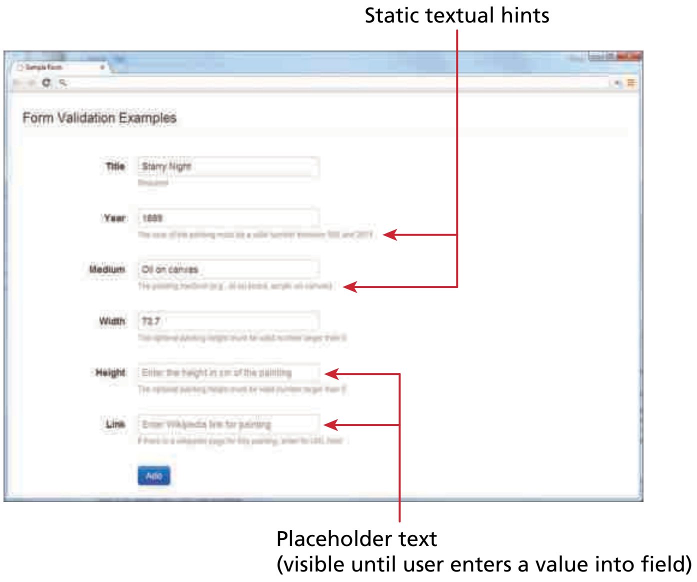
Users dislike having to do things again, so if possible, we should construct user input forms in a way that minimizes user validation errors. The basic technique for doing so is to provide the user with helpful information about the expected data before he or she enters it. Some of the most common ways of doing so include:

- Using pop-up JavaScript alert (or other popup) messages. This approach is fine if you are debugging a site still in development mode or you are trying to re-create the web experience of 1998, but it is an approach that you should generally avoid for almost any other production site. Probably the


Form Validation Examples

Title	Enter the painting title	The title is required (it cannot be blank)
Year	Enter the year of the painting	The year must be a valid number between 500 and 2014
Medium	Oil on canvas	
Width	45	
Height	Enter the height in cm of the painting	The painting height must be valid number larger than 0
Link	Enter Wikipedia link for painting	

FIGURE 12.3 Indicating where an error is located



Form Validation Examples

Title	Starry Night
Year	1889
Medium	Oil on canvas
Width	72.7
Height	Enter the height in cm of the painting
Link	Enter Wikipedia link for painting

```
<input type="text" … placeholder="Enter the height ...">
```

FIGURE 12.4 Providing textual hints

only usability justification for pop-up error messages is for situations where it is absolutely essential that the user see the message. Destructive and/or consequential actions such as deleting or purchasing something might be an example of a situation requiring pop-up messages or confirmations.

- Provide textual hints to the user on the form itself, as shown in Figure 12.4. These could be static or dynamic (that is, only displayed when the field is active). The `placeholder` attribute in text fields is an easy way to add this type of textual hint (though it disappears once the user enters text into the field).
- Using tool tips or pop-overs to display context-sensitive help about the expected input, as shown in Figure 12.5. These are usually triggered when the user hovers over an icon or perhaps the field itself. These pop-up tips are especially helpful for situations in which there is not enough screen space to display static textual hints. However, hover-based behaviors will generally not work in environments without a mouse (e.g., mobile or tablet-based browsers). HTML does not provide support for tool tips or pop-ups, so you will have to use a JavaScript-based library or jQuery plug-in to add this behavior to your pages. The examples shown in Figure 12.5 were added via the Bootstrap framework introduced in Chapter 5.
- Another technique for helping the user understand the correct format for an input field is to provide a JavaScript-based mask, as shown in Figure 12.6. The advantage of a mask is that it provides immediate feedback about the nature

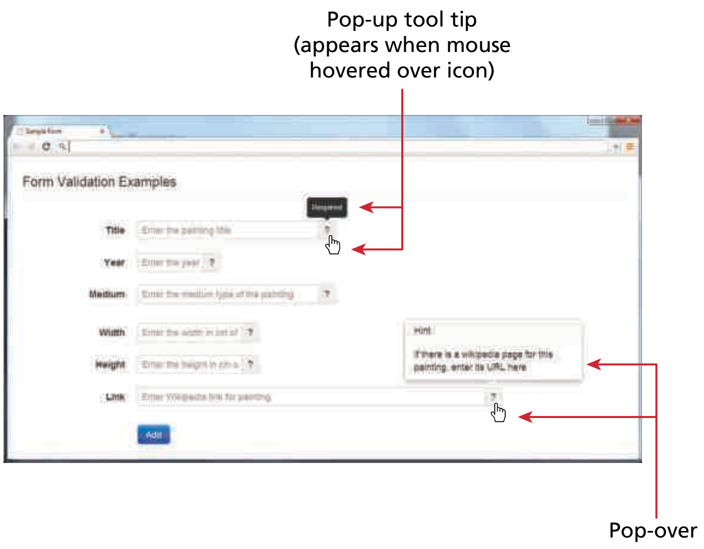

Pop-up tool tip (appears when mouse hovered over icon)
Form Validation Examples
Required
Title
Enter the painting title
Year
Enter the year
Medium
Enter the medium type of the painting
Width
Enter the width in cm of
Height
Enter the height in cm o
Link
Enter Wikipedia link for painting
Hint
If there is a wikipedia page for this painting, enter its URL here
Add
Pop-over

FIGURE 12.5 Using tool tips

of the input and typically will force the user to enter the data in a correct form. While HTML5 does provide support for regular expression checks via the `pattern` attribute, if you want visible masking, you will have to use a JavaScript-based library or jQuery plug-in to add masking to your input fields.

- Providing sensible default values for text fields can reduce validation errors (as well as make life easier for your user). For instance, if your site is in the .uk top-level domain, make the default country for new user registrations the United Kingdom.

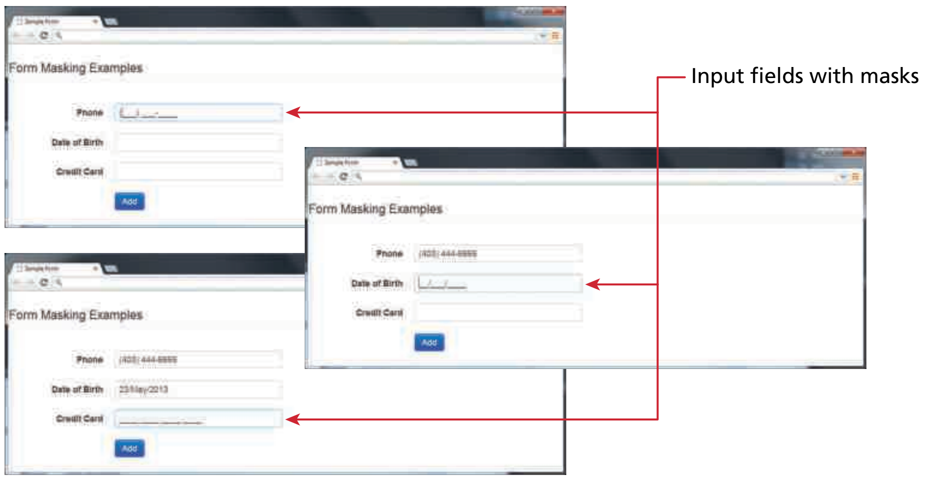


FIGURE 12.6 Using input masks

- Finally, many user input errors can be eliminated by choosing a better data entry type than the standard `<input type="text">`. For instance, if you need the user to enter one of a small number of correct answers, use a select list or radio buttons instead. If you need to get a date from the user, then use either the HTML5 `<input type="date">` type (or one of the many freely available jQuery versions). If you need a number, use the HTML5 `<input type="number">` input type.

PRO TIP

One of the most common problems facing the developers of real-world web forms is how to ensure that the user submitting the form is actually a human and not a bot (that is, a piece of software). The reason for this is that automated form bots (often called spam bots) can flood a web application form with hundreds or thousands of bogus requests.

This problem is generally solved by a test commonly referred to as a CAPTCHA (which stands for Completely Automated Public Turing test to tell Computers and Humans Apart) test. Most forms of CAPTCHA ask the user to enter a string of numbers and letters that are displayed in an obscured image that is difficult for a software bot to understand. Other CAPTCHAs ask the user to solve a simple mathematical question or trivia question.

We think it is safe to state that most human users dislike filling in CAPTCHA fields, as quite often the text is unreadable for humans as well as for bots. They also present a usability challenge for users with visual disabilities. As such, in general one should only add CAPTCHA capabilities to a form if your site is providing some type of free service or if the site is providing a mechanism for users to post content that will appear on the site. Both of these scenarios are especially vulnerable to spam bots.

If you do need CAPTCHA capability, there are a variety of third-party solutions. Perhaps the most common is reCAPTCHA, which is a free open-source component available from Google. It comes with a JavaScript component and PHP libraries that make it quite easy to add to any form.

12.6 Where to Perform Validation

Validation can be performed at three different levels. With HTML5, the browser can perform basic validation with no need for any JavaScript. However, since the validation that can be achieved in HTML5 is quite basic, most web applications also perform validation in the browser using JavaScript. The advantage of validation using JavaScript is that it reduces server load and provides immediate feedback to

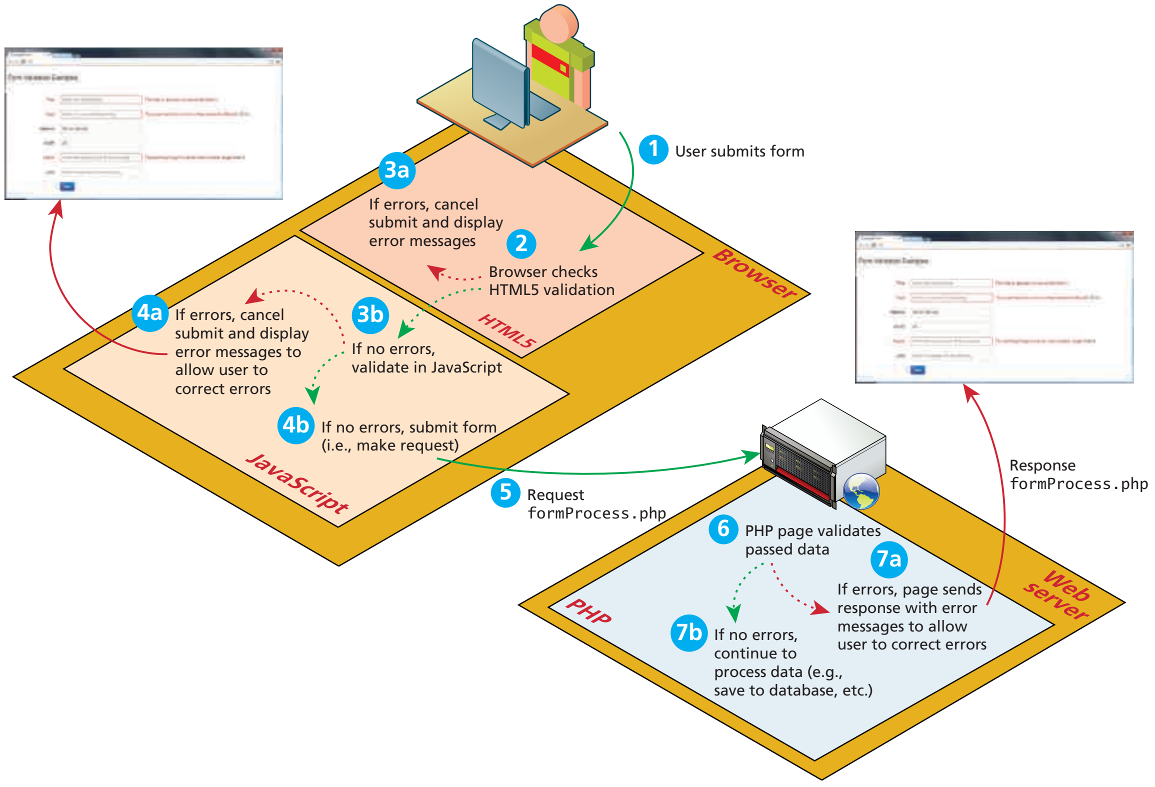
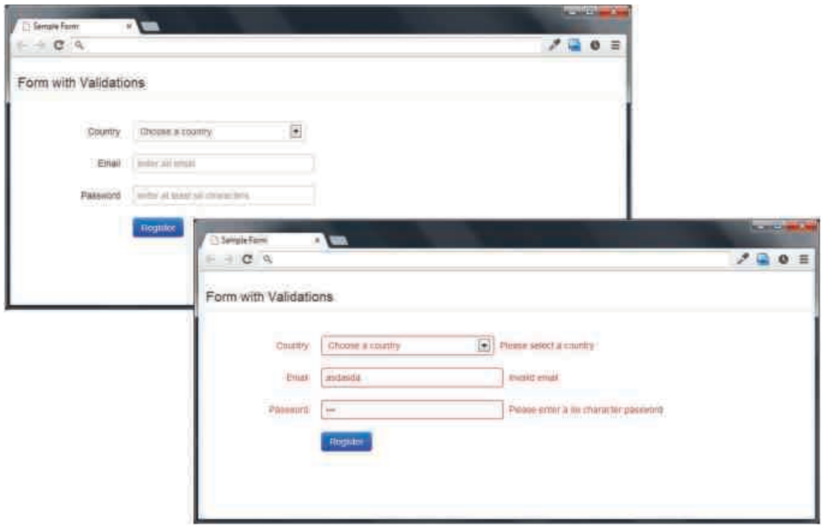

1 User submits form
2 Browser checks HTML5 validation
3a If errors, cancel submit and display error messages
3b If no errors, validate in JavaScript
4a If errors, cancel submit and display error messages to allow user to correct errors
4b If no errors, submit form (i.e., make request)
5 Request formProcess.php
6 PHP page validates passed data
7a If errors, page sends response with error messages to allow user to correct errors
7b If no errors, continue to process data (e.g., save to database, etc.)
Response formProcess.php
Browser
HTML5
JavaScript
Web server
PHP

FIGURE 12.7 Visualizing levels of validation

the user. Unfortunately, JavaScript validation cannot be relied on: for instance, it might be turned off on the user's browser. For these reasons, validation must always be done on the server side. Indeed, you should always perform the same validity checks on *both* the client in JavaScript and on the server in PHP, but server-side validation is the most important since it is the only validation that is guaranteed to run. Figure 12.7 illustrates the interaction of the different levels of validation.

To illustrate this strategy, let us take a look at a simple validation example. We will be creating the form and validations shown in Figure 12.8. The markup makes use of a variety of CSS classes defined in the Bootstrap framework, which was examined back in Chapter 5. Listing 12.8 shows the markup to which we will add validation.

Notice that each form element is contained within a `<div>` element with the `control-group` class. We will later programmatically add a CSS class to this element to visually indicate that an input element has a validation error. Notice as well the `<span>` element with the class `help-inline`. We will programmatically insert error messages into this span when a validation error occurs.



Please select a country

Invalid email

Please enter a six character password

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

Notice as well the use of the `required` attributes on the input elements, which is the first step in the validation strategy shown in Figure 12.7. You may recall from Chapter 4 that HTML5 also includes its own validation checks. 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.

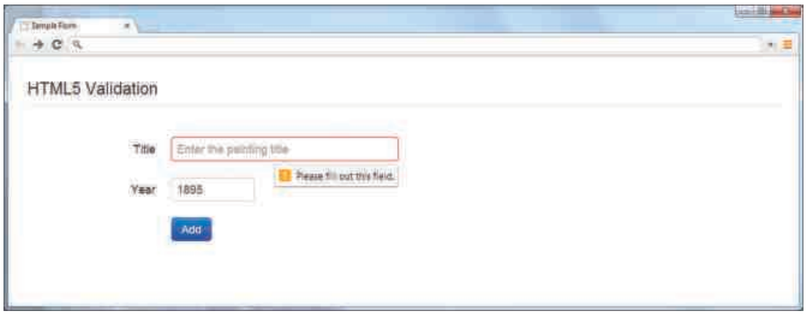


FIGURE 12.9 HTML5 browser validation

If you wish to disable the browser validation (perhaps because you want a unified visual appearance to all validations), you can do so by adding the novalidate attribute to the form attribute:

```
<form id="sampleForm" method="..." action="..." novalidate>
```

NOTE

It cannot be stressed enough that all user input **should** be validated if possible on both the client side and on the server side. But **all user input must be validated on the server side.**

To reinforce this principle, JavaScript validation is sometimes referred to as *prevalidation*, to allude to the server-side validation that must always occur no matter what happens in JavaScript.

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. To simplify our example, we will only validate on a form submit.

```
function init() {
   var sampleForm = document.getElementById('sampleForm');
   sampleForm.onsubmit = validateForm;
}
// call the init function once all the html has been loaded
window.onload = init;
```

The basic validation is quite straightforward since we will be using regular expressions. For instance, to check if the value in the form's password input element is between 8 and 16 characters, the JavaScript would be:

```
var passReg = /^[a-zA-Z]\w{8,16}$/;
if (! passReg.test(password.value)) {
   // provide some type of error message
}
```

What do we want to do when the JavaScript finds a validation error? In this example, we will insert error message text into the relevant <span> element and add the error class to the parent <div id="control-group"> elements. For instance, to display the appropriate changes for the password element, we would do something similar to the following:

```
    var span = document.getElementById('errorPassword');
    var div = document.getElementById('controlPassword');

    // add error message to error span element
    if (span) span.innerHTML = "Enter a password between 8-16 characters";
    // add error class to surrounding <div>
    if (div) div.className = div.className + " error";
```

Our form would also need to clear these error messages once the user fixes them. To simplify for clarity's sake, we will clear the error state once the user makes some change to the element. 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);
```

(*continued*)

```
   // 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

No matter how good the HTML5 and JavaScript validation, client-side prevalidation can always be circumvented by hackers, or turned off by savvy users. 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. Since we will be doing reasonably similar checks on all three of the parameters, we will encapsulate the code into the class shown in Listing 12.10. Notice that the `checkParameter()` method is static.

Since most of the validation work is being done by the regular expressions and the `ValidationResult` class, the PHP needed in the form is minimal, as shown in Listing 12.11. To help us differentiate the JavaScript error messages from the PHP error messages, this example has the text "[PHP]" appended to the end of the error message strings.

```
<?php
/*
  Represents the results of a validation
*/
class ValidationResult
{
   private $value;            // user input value to be validated
   private $cssClassName;     // css class name for display
   private $errorMessage;     // error message to be displayed
   private $isValid = true;   // was the value valid

   // constructor
   public function __construct($cssClassName, $value, $errorMessage,
                               $isValid) {
      $this->cssClassName = $cssClassName;
      $this->value = $value;
      $this->errorMessage = $errorMessage;
      $this->isValid = $isValid;
   }

   // accessors
   public function getCssClassName() { return $this->cssClassName; }
   public function getValue() { return $this->value; }
   public function getErrorMessage() { return $this->errorMessage; }
   public function isValid() { return $this->isValid; }

   /*
     Static method used to check a querystring parameter
     and return a ValidationResult
   */
   static public function checkParameter($queryName, $pattern,
                                          $errMsg) {
      $error = "";
      $errClass = "";
      $value = "";
      $isValid = true;

      // first check if the parameter doesn't exist or is empty
      if (empty($_POST[$queryName])) {
         $error = $errMsg;
         $errClass = "error";
         $isValid = false;
      }
      else {
         // now compare it against a regular expression
```

```
            $value = $_POST[$queryName];
            if ( ! preg_match($pattern, $value) ) {
               $error = $errMsg;
               $errClass = "error";
               $isValid = false;
            }
         }
         return new ValidationResult($errClass, $value, $error, $isValid);
      }
   }
   ?>
```

LISTING 12.10 ValidationResult class

```
<?php
// turn on error reporting to help potential debugging
error_reporting(E_ALL);
ini_set('display_errors','1');

include_once('ValidationResult.class.php');

// create default validation results
$emailValid = new ValidationResult("", "", "", true);
$passValid = new ValidationResult("", "", "", true);
$countryValid = new ValidationResult("", "", "", true);

// if GET then just display form
//
// if POST then user has submitted data, we need to validate it
if ($_SERVER["REQUEST_METHOD"] == "POST") {
   $emailValid = ValidationResult::checkParameter("email",
                 '/(.+)@([^\.].*)\.([a-z]{2,})/',
                 'Enter a valid email [PHP]');
   $passValid = ValidationResult::checkParameter("password",
                '/^[a-zA-Z]\w{8,16}$/',
                'Enter a password between 8-16 characters [PHP]');
   $countryValid = ValidationResult::checkParameter("country",
                   '/[1-4]/', 'Choose a country [PHP]');

   // if no validation errors redirect to another page
   if ($emailValid->isValid() && $passValid->isValid() &&
                           $countryValid->isValid() ) {
      header( 'Location: success.php' );
   }
```

(*continued*)

```
}

?>
<!DOCTYPE html>
<html>
...
```

LISTING 12.11 PHP form validation

NOTE

The PHP `header()` function will only redirect if there has been no other output to the response stream (i.e., before any HTML or PHP echo-type statements).

Finally, we need to display error messages and error CSS classes (or display empty strings if no errors) if the PHP encounters any errors, as shown in Listing 12.12. Notice the revised `action` attribute in the listing. If a form is posting back to itself, it

```
<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 Revised form with PHP validation messages

is preferable to use $_SERVER["PHP_SELF"] instead of hard-coding a location since you won't have to update the code if you change the script's name.

Since this example has validation at both the JavaScript and PHP levels, you will need a way to test whether the PHP validation is working. You can do this by turning off JavaScript in the browser, or by *temporarily* commenting out the following line in the JavaScript (which loads the event handler that sets up the JavaScript validators):

```
window.onload = init;
```

PRO TIP

In Chapter 15, you will be learning about the popular JavaScript-based jQuery framework. There are many jQuery-based validation plug-ins that can not only simplify client-side validation and message display, but can perform the validations during the focus and change events and even perform validations that require server-based information asynchronously using AJAX techniques.

12.7 Chapter Summary

This chapter covers perhaps the least exciting topic in software development: that of exception and error handling. But what the topic lacks in excitement, it makes up for in importance. The improper handling of exceptions and errors is one of the main reasons sites can get into trouble, and requires careful attention by developers. This chapter began by examining the different types of errors and how errors are different from exceptions. It also briefly examined how to customize the way PHP reports warnings and errors. It covered how to handle both errors and exceptions in PHP. The vital topic of regular expressions was introduced along with a more involved example. A variety of validation best practices were then enumerated. Finally, the chapter demonstrated how a multilevel approach to user input validation can be constructed that integrates validation at the HTML, JavaScript, and PHP levels.

12.7.1 Key Terms

CAPTCHA	fatal error	spam bots
error	literal	warning
exception	metacharacter	
expected error	regular expression	

12.7.2 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 your 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?

12.7.3 Hands-On Practice

PROJECT 1: Photo Sharing Site

DIFFICULTY LEVEL: Easy

HANDS-ON EXERCISES

PROJECT 12.1

Overview

This project simply walks you through various logging techniques and settings but illustrates how error message management is important.

Instructions

1. Open lab12-project01.html in the editor of your choice, so you can start making changes. This file is riddled with various levels of errors and warnings, but otherwise is just a poorly done implementation from Chapter 9.
2. Add a line to display errors inside the browser. Refresh the page to see a wealth of errors output as shown in Figure 12.10.
3. Since the errors being displayed reveal quite a lot about your application, we will have to log the errors to somewhere safer. Create an entry in the script to output the errors to a log file (or locate the log file on your server).
4. Turn off error display inside the browser.

Testing

1. Run the page, and you should see no error output.
2. Fix the errors if you want to. Knowing how to see them, and knowing that you could fix them may well suffice for this project.
3. Remember how easy it is to look up and configure error logging outside of the browser. Try to apply this principle throughout your development to avoid creating security holes that leak information out through error messages.

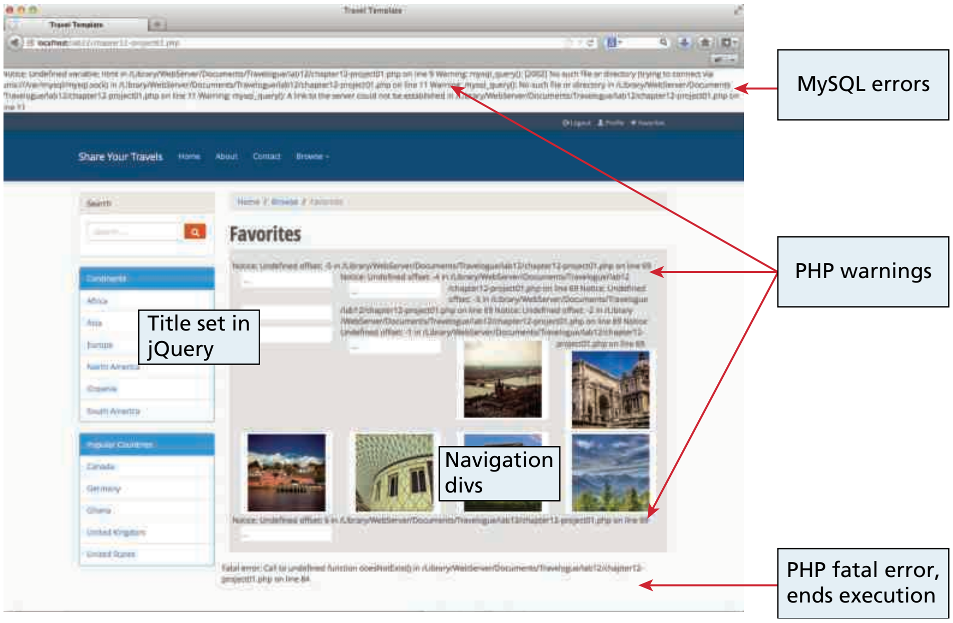


FIGURE 12.10 Illustration of the errors being displayed inside the browser

PROJECT 2: Art Store

DIFFICULTY LEVEL: Basic

HANDS-ON EXERCISES

PROJECT 12.2

Overview

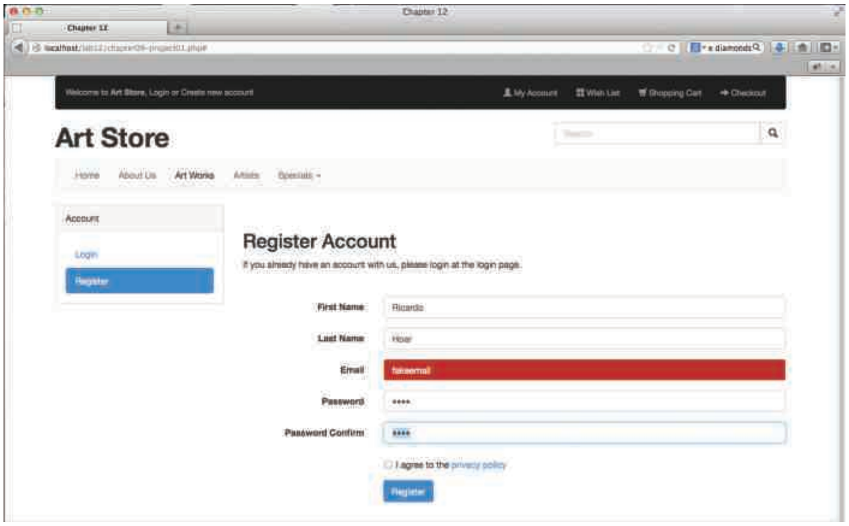
To get started with validation techniques, let us build a client-side form validation script that will check for a valid email and phone number, in addition to nonblank fields all using regular expressions.

Instructions

1. Take the PHP file you worked on in Chapter 9 Project 1 and open it in the editor of your choice, so you can start making changes. Remember, this file is used to simply echo the information posted back to the user. You may also want to make use of some code from Chapter 6 where we checked for empty fields.
2. Attach a listener to the submit event of the form that will trigger your validation script. Hint: Your validation script should use regular expressions to test for valid field values.
3. In the event there is an error, prevent the form from submitting and highlight the field in red using JavaScript as shown in Figure 12.11. Hint: See if you can add a useful error message in addition to the field highlighting.

Testing

1. Try registering with an invalid email and verify that the form identifies the error, prevents the form from submitting, and highlights the field.



Register Account

If you already have an account with us, please login at the login page.

First Name	Ricardo
Last Name	Hoar
Email	fakeemail
Password	••••
Password Confirm	••••

FIGURE 12.11 Completed Project 2

2. Now try an invalid phone number, and expect the same result.
3. Finally, submit the form with correct information, and ensure that it actually posts to the desired destination.

PROJECT 3: Art Store

DIFFICULTY LEVEL: Medium

Overview

To properly implement validation, server-side code must perform validation, even if you performed JavaScript prevalidation. This project adds server-side validation to check for a valid email and phone number.

HANDS-ON EXERCISES

PROJECT 12.3

Instructions

1. Continue working on the file from Project 2.
2. Modify your registration form so that it posts new users into the database (and mails them a confirmation link . . . more on that in Chapter 16).
3. Write validation regular expressions to ensure the submitted form had good data. Perform the same tests you did in JavaScript.
4. Reflect on why you are writing similar code on the server, as you did on the client. What is the purpose of having validation on client and server?
5. Modify the PHP file to return the login page if there is any error in registration with the errors highlighted using the same CSS styles as in the last project. Note: There should be no difference between the CSS styling used in client-side validation and server-side validation.

Testing

1. Disable JavaScript to purposefully circumvent the client-side validation you just created for Project 1.
2. Try registering with an invalid email and verify that the server-side script catches the error and returns the HTML with highlighting around the invalid fields.
3. Now try an invalid phone number, and expect the server catches that error also, and similarly highlights the bad field.
4. Finally, submit the form with valid information, and ensure that it posts to the desired destination, and returns a success message.

12.7.4 References

1. PHP, "error_reporting." [Online]. http://php.net/manual/en/function.error-reporting.php.
2. PHP, "Runtime Configuration." [Online]. http://php.net/manual/en/errorfunc.configuration.php.
3. PHP, "error_log." [Online]. http://php.net/manual/en/function.error-log.php.
4. PHP, "Exceptions." [Online]. http://php.net/manual/en/language.exceptions.php.
5. IEEE Standards Association, "IEEE Standards." POSIX: Austin Joint Working Group. [Online]. http://standards.ieee.org/develop/wg/POSIX.html.