# PHP and MySQL

Marc Wandschneider


Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

*Library of Congress Control Number:* 2008933546

Visit us on the Web: informit.com/ph

**Corporate and Government Sales**

The publisher offers excellent discounts on this LiveLesson when ordered in quantity for bulk purchases or special sales, which may include custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact: U.S. Corporate and Government Sales, (800) 382-3419, corpsales@pearsontechgroup.com.

For sales outside the United States, please contact: International Sales, international@pearsoned.com.

**Warning and Disclaimer**

This book and video product is designed to provide information about PHP and MySQL programming. Every effort has been made to make it as complete and as accurate as possible, but no warranty or fitness is implied. The information is provided on an "as is" basis. The author and Pearson shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book or from the use of the disc or programs that may accompany it. The opinions expressed in this LiveLesson belong to the author and are not necessarily those of Pearson.

**Feedback Information**

At Pearson, our goal is to create in-depth technical products of the highest quality and value. Each product is crafted with care and precision, undergoing rigorous development that involves the unique expertise of members from the professional technical community. Readers' feedback is a natural continuation of this process. If you have any comments regarding how we could improve the quality of this LiveLesson, or otherwise alter it to better suit your needs, you can contact us through e-mail at mylivelessons@pearsoned.com. Please make sure to include the title and ISBN in your message.

We greatly appreciate your assistance.

**Trademark Acknowledgments**



ISBN-13: 978-0-13-715575-0
ISBN-10: 0-13-715575-1

Text printed in the United States at RR Donnelley in Crawfordsville, Indiana.
First printing, September 2008

**Publisher**
Paul Boger

**Editor-in-Chief**
Mark L. Taub

**Video Project Manager**
John A. Herrin

**Editorial Assistant**
Kim Boedigheimer

**Managing Editor**
John Fuller

**Project Editor**
Julie B. Nahil

**Copy Editor**
Ruth Davis

**Proofreader**
Julie Lonergan

**Multimedia Developer**
Eric Strom, Pearson Video Production Services

**Designer**
Gary Adair

# Contents

lesson 1

# Installation of MySQL, Apache, and PHP

## Windows

To install MySQL, Apache, and PHP on Microsoft Windows, we will largely use installers that we can download from the appropriate Web sites.

### MySQL

We will visit http://mysql.com, where we click "Downloads" to download the Community Server Edition of MySQL, which is a free and open-source package. Select the appropriate version of Windows and choose the Windows ZIP/Setup.EXE option.

After downloading, extract the setup program from the ZIP file and then run it. We will use the default values for all of the steps in the dialog, except that we will choose a custom install and select C:\MySQL as our install path instead of the recommended location. When you enter the administrator password for this new installation, don't forget to write it down somewhere. You will need this in later lessons (and indeed any other time you want to administer your MySQL installation).

After the installation is complete, we will configure the server right away, opting to use default values for almost everything except for character-set support, where we will use UTF-8 as the default. Set a password, and then MySQL will be ready to go!

### Apache HTTPD Server

Visit http://httpd.apache.org to download the latest version of the Apache Web server software, and download the MSI installer package. After downloading, run this program, and follow the steps for installation. Choose a custom installation and install the software to C:\Apache\HttpServer2.2 instead of the default location.

By default, this Web server software will use C:\Apache\HttpServer2.2\htdocs as its *DocumentRoot* for the *localhost* host. The DocumentRoot is where Apache looks for any requested documents or scripts for a given host.

So, if the user wants to view http://localhost/test.php, Apache looks in C:\Apache\HttpServer2.2\htdocs for a (script) file called test.php, and executes it if it finds it.

### PHP

Finally, visit http://php.net and download the PHP binary installer for Windows. You can run this, and change the default installation location to C:\PHP. Then, in addition to the default installation, install the Multi-byte Strings and MySQLi extensions, and also choose to install PEAR and the PHP manual under the final section of installation.

The PHP installer will ask you where the Apache configuration files are. They are located in C:\Apache\HttpServer2.2\conf, assuming you chose that as your Apache HTTPD installation location. With that, PHP configures almost everything itself.

You have to do one more configuration step yourself, however, and that is to tell the Apache Server that files with the extension *.php* should go to the PHP interpreter. To do this, open up notepad.exe, edit C:\Apache\HttpServer2.2\conf\httpd.conf, and in the section `<IfModule mime_module>`, add the following:

```
AddType application/x-httpd-php .php
```

PHP places its configuration file in C:\PHP, and the file is named *php.ini*. It is a simple text file that can be edited with a simple program such as notepad.exe on Windows.

With this, you'll be ready to go.

## Mac / Unix

We'll cover the installation steps for the Mac here. Since it's a reasonably standard Unix operating system these days, the instructions for operating systems such as Linux or FreeBSD are very similar.

### Xcode

To compile programs on the Mac, you need to download and install Apple's Xcode tools from http://developer.apple.com/tools/download. Version 3.0 is for Leopard users and 2.5 is for Tiger users. The Leopard install DVD comes with this in the section of extras, saving you a huge download.

Install this and just choose all the default options.

### MySQL

We will actually use the package installer that MySQL provides for its software, since it's quite convenient and handles all the user account setup and permissions for us.

Visit http://mysql.com and download the Mac package format appropriate for your version of Mac OS X and your processor architecture. Mount the Disk Image (*.dmg*) file and run through the installer, after which MySQL will be installed on your machine (but not running yet).

### Apache HTTP Server

We visit http://httpd.apache.org to download the Apache Web server; the 2.2.x series was current as we wrote this document. Download the *.tar.bz2* version of it, then open up */Applications/Utilities/Terminal.app/*. We'll create a new temporary directory to build it in:

```
# mkdir tmp
# cd tmp
```

```
# tar xfj ../Downloads/httpd-2.2.9.tar.bz2
# cd http2.2.9
# ./configure --prefix=/usr/local/apache2 --enable-modules=so
# make
# sudo make install
```

After the server software is installed, you must then do some configuration. We will create a virtual host called *livelessons* with which to do all of our development. This is a two-step process:

1. Add a virtual host for *livelessons* to the *httpd.conf* file.
2. Add a hostname entry to */etc/hosts* to let the operating system know about the host.

Our virtual host will have a *DocumentRoot* in the *Sites* subdirectory of our home directory. On my Mac computer this is */Users/marcw/Sites*—substitute the path for your own home directory in the Virtual Host section below.

To edit the Apache configuration files, we will have to use the *sudo* command, which lets us execute commands as the super user. We will also need to be familiar with a Unix command line text editor, such as *emacs*, *vim*, or *jove*.

```
sudo emacs /usr/local/apache2/conf/httpd.conf
```

We will first make sure the following line near the bottom is uncommented:

```
# Virtual hosts
Include conf/extra/httpd-vhosts.conf
```

Next, we will open up */usr/local/apache2/conf/extra/httpd-vhosts.conf*, and replace the two <VirtualHost *:80> sections with the following:

```
<VirtualHost *:80>
    DocumentRoot "/Users/marcw/Sites"
    ServerName livelessons
</VirtualHost>
```

We also will need to make sure the server thinks it has permission to serve content from the DocumentRoot, so we must add the following as well:

```
<Directory "/Users/marcw/Sites">
    #
    # Possible values for the Options directive are "None", "All",
    # or any combination of:
    #   Indexes Includes FollowSymLinks SymLinksifOwnerMatch #
    # Note that "MultiViews" must be named *explicitly* ---
    # doesn't give it to you.
    #
    Options Indexes FollowSymLinks

    #
    # AllowOverride controls what directives may be placed in .htaccess
```

```
    # It can be "All", "None", or any combination of the keywords:
    #   Options FileInfo AuthConfig Limit
    #
    AllowOverride None

    #
    # Controls who can get stuff from this server.
    #
    Order allow,deny
    Allow from all

</Directory>
```

This actually completes the basic configuration of the Apache server.

The second step is to create the hostname entry in */etc/hosts* so that the local computer knows how to find the *livelessons* host.

```
sudo emacs /etc/hosts
```

We will then add the following entry near the top:

```
127.0.0.1     livelessons
```

This tells the computer that the host *livelessons* has an IP address of 127.0.0.1, which is actually our local computer!

## PHP

We will visit http://php.net/download to download the *.tar.bz2* for the latest version of PHP source code. We will then compile and install it from Terminal.app, much like for the Web server:

```
# cd ~
# cd tmp
# tar xfj ../Downloads/php-5.2.6.tar.bz2
# cd php-5.2.6
# /configure  --prefix=/usr/local/php5 \
--with-apxs2=/usr/local/apache2/bin/apxs \
--with-mysqli=/usr/local/mysql/bin/mysql_config \
--enable-mbstring --enable-mbregex
# make
# sudo make install
```

Once the software is compiled and installed, the only thing left to do is to tell the Apache httpd.conf about the *.php* file extension:

```
sudo emacs /usr/local/apache/conf/httpd.conf
```

In the section `<IfModule mime_module>`, add the following:

```
AddType application/x-httpd-php .php
```

Finally, we'll copy over a suggested *php.ini* file to */usr/local/php5/lib*, so we have something to work with in later lessons when we want to change the default behavior of PHP:

```
sudo cp ~/tmp/php-5.2.6/php.ini-dist /usr/local/php5/lib/php.ini
```

## Starting and Stopping the Servers

To start and stop the MySQL server on your Mac machine:

```
sudo /usr/local/mysql/bin/safe_mysqld --user=mysql
```

Press CTRL+Z to pause it, then type `bg` to put the process into the background.

To stop MySQL:

```
sudo /usr/local/mysql/bin/mysqladmin shutdown
```

To start the Apache Web server:

```
sudo /usr/local/apache2/bin/apachectl start
```

To stop it:

```
sudo /usr/local/apache2/bin/apachectl stop
```

To restart in case of configuration file change:

```
sudo /usr/local/apache2/bin/apachectl restart
```

## Troubleshooting

The three most common errors you will encounter when trying to run PHP scripts are as follows:

- **Permission denied—**This is almost always because the <Directory> entry for your DocumentRoot in *httpd.conf* or *http-vhosts.conf* doesn't match the actual DocumentRoot directory in your <VirtualHost> section.
- **Couldn't connect to host,** or the browser tries to connect to *livelessons.com*—Your Web server is probably not running. Start the Apache server software again.
- Source code prints on the screen instead of being executed—You forgot to add the AddType section to <IfModule mime_module> in *httpd.conf*. Apache does not know that the script is PHP, so it assumes it's something like HTML and sends the file's contents down to the user.

lesson 2

# Your First Web Application

## Getting Started

The first part of writing our sample *Hello World* application will be to identify the directory where the Apache HTTPD server software loads documents. This location is called the *DocumentRoot*, and we will refer to this frequently throughout the LiveLessons and documentation.

- On Apple Macintosh and Unix machines, when we installed the server software, we created a new virtual host in */etc/hosts* called "livelessons". We set the *DocumentRoot* for this virtual server to be *~/Sites*, which is the Sites subdirectory of our home directory.
- On Windows machines, we decided to use the default "localhost" virtual server, and also used the default *DocumentRoot*, which is in C:\Apache\HttpServer2.2\htdocs.

Thus, whenever we say we're going to add a document to our DocumentRoot, you will just be creating a file in this directory.

To create files, you will want to select any program that will let you enter and save plain text files. Those that handle UTF-8 are slightly more useful if you ever plan to enter accented or foreign letters—fortunately, most modern editors support this. Some common editors that people use are:

- emacs—available on almost any platform that PHP is
- vi (and its open-source cousin vim)—also available on both Windows and Mac/Unix platforms
- BBEdit—a Macintosh-only editor that's quite popular with PHP programmers
- Programmer's Notepad—a Windows-only editor
- TextEdit.app—The default Macintosh text editor works fine for editing plain text files, as long as you go to the Format menu and select "Make Plain Text".
- Notepad.exe—The default Windows text editor also works fine for editing plain text.

Our first little Web application is roughly as follows:

```
<html>
  <head>
    <title>My first PHP Web Application</title>
  </head>
```

```
<body>
<?php
     echo "Hello World!";
?>
</body>
</html>
```

When you view the source in the client Web browser, you see that none of the PHP code is there, and there's only HTML markup. This is because the PHP code is executed on the server. When the client browser makes a request, the server loads PHP and tells it to process the script. The PHP language engine, in turn, processes the script, outputting whatever text it sees, and only starts processing PHP language when it sees the `<?php` marker. It then executes the code we have, which tells it to print out a single line. After the `?>`, it just continues dumping the text it sees as it works through the file.

# Entering PHP Scripts

## Marking Sections of PHP Code

There are a few ways to indicate that a section of the input file contains PHP script. The most common way, which we have seen before, is as follows:

```
<?php
  echo "Hello Everybody!";
?>
```

Another similar way to demarcate PHP script is as follows:

```
<?
  echo "Bonjour tout le monde!";
?>
```

This is called using *short tags*, and is available only if the `short_open_tag` setting is enabled in your *php.ini* configuration file. We have turned this setting on in all of our configuration files.

A third way of entering script is:

```
<script language="php">
  echo "Ciao a tutti!";
</script>
```

One final style of inputting script exists to support some older graphical HTML editing programs that did not (do not) understand script directives very well. To let users continue to use these editors, PHP script can be marked using these ASP tags:

```
<%
   echo "Guten Tag alle!";
%>
```

ASP tags can only be used if the `asp_tags` setting is enabled in the *php.ini* configuration file.

## Mixing PHP and HTML

There is nothing that compels or requires large blocks of PHP code when writing HTML and PHP. You are completely free to mix the markup and script as much as you wish:

```
<?php
   $userName = "Chippy the Chipmunk";
?>

<p align='left'>
  Hello there, <b><?php echo $userName; ?></b>
</p>
```

There is a shortcut that exists for this particular usage, involving the short tags discussed in the previous section along with an equals sign ( = ):

```
<?= $userName ?>
```

This is the same as typing in full:

```
<?php echo expression; ?>
```

The flexibility available when mixing PHP and HTML lets us get creative when we get into more advanced language constructs. You are perfectly welcome to mix them with various control structures, which we will introduce in the next lesson:

```
<?php

   if ($file_received_successfully === TRUE)
   {
?>
   <p align='center'> Thank for your contribution </p>
<?php
   }
   else
   {
?>
   <p align='left'>
     <font color='red'>
     <b>Error:  The file was not correctly received.</b>
     </font>
   </p>
<?php
   }

?>
```

## Statements and Comments

Statements in PHP, like many other languages, are separated by a semi-colon (;) character. Statements can be grouped together by wrapping them in brackets ( { and } ); this is sometimes called a *block* of code. Any number of statements can be placed on one line, individual items (also known as tokens) within a statement can be separated by arbitrary amounts of white-space (space, newline characters, or tabs), and statements can span more than one line:

```
<?php

    $x = 123; $y = 456; $z = "hello there"; $a = "moo";

  {
       echo "This is a group of statements";
       $m = "oink";
  }

    $userName
                         =
  "Chippy the Chipmunk"
                               ;
?>
```

There are three basic styles for entering comments in PHP.

```
<?php
     /*
      * This is our first style of comments. They can span multiple
      * lines.
      */
     echo "Style 1";

     //
     // This is our second style of comments.  It is "single line", but
     // you can use as many of them as you'd like.
     //
     echo "Style 2";

     #
     # This third style is also "single line"
     #
     echo "Style 3";
?>
```

The two types of comments that are single line cause the PHP language processor to ignore all code until the end of the current line or current PHP script section.

```
<?php
    // all of this line is ignored.
    echo "But this line prints just fine.";
?>

<?php #Comment!! ?><b>This prints</b><?php echo "this prints" ?>
```

# Variables

To declare a variable in PHP, you simply assign it a value. Variable names in PHP are represented by a dollar sign (`$`) followed by an identifier that begins with either a letter or underscore, which in turn can be followed by any number of underscores, numbers, or letters.

```
<?php
  $varname = "moo";               // ok
  $var_____Name = "oink";         // ok
  $__12345var = 12345;            // ok
  $12345__var = 12345;            // NOT ok - starts w/ number
?>
```

In versions of PHP prior to version 4, variables would be declared at their first use (instead of first *assignment*), which often proved tricky when debugging problems with your code.

```
<?php
   $cieling = "roof";           // whoops misspelled it!
   echo "$ceiling";             // prints an empty string.
?>
```

Fortunately, PHP 5 will now print a warning saying that "`$ceiling`" has not been assigned a value.

## Numbers

There are two basic types of numbers in the language, *integer* and *float*, associated with the keywords `int` and `float` (or `double`).

Integers are specified in code either in *octal* (base-8 notation), *decimal* (base-10 notation), or *hexadecimal* (base-16 notation)

```
<?php

   $abc = 123;      // decimal
   $def = -123;
   $ghi = 0173;     // octal, value is 123 in decimal
   $jkl = -0173;    // octal, value is -123 in decimal
   $mno = 0x7b;     // hexadecimal, 123
   $pqr = -0x7B;    // hexadecimal, -123

?>
```

Integer precision varies largely by the underlying operating system, but 32 bits is common. There are no unsigned integers in PHP, so the maximum value for an integer is typically larger than 2 billion. Unlike some other languages that will overflow large positive integers into large negative integers, however, PHP will actually overflow integers to floating-point numbers.

```
<?php
   $large = 2147483647;
   var_dump($large);
   $large = $large + 1;
   var_dump($large)
?>
```

The output of this script will be:

```
int(2147483647)  float(2147483648)
```

Floating-point variables can be input in a few different ways, as follows:

```
<?php
    $floatvar1 = 7.555;
    $floatvar2 = 6.43e2;         // same as 643.0
    $floatvar3 = 1.3e+4;         // same as 13000.0;
    $floatvar4 = 5.555e-4;       // same as 0.0005555;
    $floatvar5 = 1000000000000;  // too big for int ==> float
?>
```

## Strings

A string is a sequence of characters. In PHP, these characters are 8-bit values. There are three ways to specify a string value.

### Single-Quoted

Single-quoted strings are sequences of characters that begin and end with a single quote (`'`) character.

```
<?php echo 'This is a single-quoted string.'; ?>
```

In order to include a single quote within one, you simply put a backslash in front of it, which is called *escaping* the character.

```
<?php echo 'This is a single-quoted (\') string.'; ?>
```

### Double-Quoted

Double-quoted strings are similar to single-quoted strings, except that the PHP language processor will actually dissect them looking for special escape sequences and variables, which will be replaced.

```
<?php echo "This is a double-quoted string."; ?>
```

Both single- and double-quoted strings can span multiple lines. The newline characters are simply interpreted as part of the input string.

### Heredoc Notation

The third way to input strings in PHP script is to use the *heredoc* syntax. In this, a string begins with <<< and an identifier, and continues until PHP sees an input line of text consisting *only* of the left-aligned (same) identifier and a semicolon character ( ; ). For example:

```
<?php

   echo <<<HTML

   <p align='center'>
     This is an example of text being input using the heredoc
     notation in PHP.
   </p>

HTML;

?>
```

*Heredoc* strings behave much like double-quoted strings as above, although you do not need to escape as many of the characters—newline characters and tabs can be freely entered.

## Booleans

Booleans are the simplest type in the PHP type system, and express a binary value: true or false, yes or no, one or zero. The value of a Boolean variable can be either `TRUE` or `FALSE`. These two keywords are not at all case sensitive.

# Working with Multiple Pages

The simple calculator application that we start working on in this lesson is our first serious, multipage Web application. It demonstrates one way of sending data from one page to another— using the HTML `<form>` element.

```
<form method='post' action='results.php' name='calc_form'>
      <input type='text' name='input1' size='15'>
      <input type='submit' value='Go!'>
</form>
```

The key elements to the form are the `action` attribute, which tells the client browser where to send the form contents, and the `method` attribute, which tells the browser how to send these form contents. We will cover the `POST` method and how it works in a later lesson. The value the user enters in the client browser is in the `input` element with the name `input1`. The name is important, because it is how we will refer to the data in the results page of the application.

To access the form data on the second page, we use the $_POST variable provided by PHP. It is an array, and the value of the `input` element named `input1` is at the key or index “input1”.

```
<?php
$input1 = $_POST['input1'];
?>
```

We assign the data from the form to a local variable, which we have also named *input1*, and then we can print it out later on the page.

# lesson 3

# Language Basics

## Arrays

We have already learned about the `$_POST` array. Arrays in PHP are a very powerful and flexible tool, and we will use this data type very frequently in our Web applications. In this lesson, we'll look at some of the basics of using them.

Arrays can be created in two ways: by using the `array` function or by using the `[]` operators to add values to a variable:

```
$arr1 = array(234, 32.234, 'fish fish', true);

// or

$arr2[] = 100;
$arr2[] = 200;
$arr2[] = 'cat';
$arr2[] = true;
```

Once you've created the array, you use the [] operators to add more elements to the array:

```
$arr1[] = 'a new value!';
```

By default, PHP assigns numeric indexes for each of the values in the array, starting at zero.

## Testing Variables

If you are not sure if a variable is set or not, and want your code to be robust for those cases where it is not, you can use the `isset` function in PHP to tell if you a variable or array index has been defined previously in the script.

```
if (!isset($x))
   echo "\$x is not set yet";

if (!isset($_POST['input1']))
   echo "You did not provide the correct input, sorry!";
```

# Simple Variable Substitution

PHP has a very handy feature with strings by which it can substitute values from variables directly into the strings. This is called *variable substitution*, and we will look at the simple kind now:

```
$number_of_fish = 10;

echo "There are $number_of_fish in the bowl";

$arr = array(235, 436262436, 4.55e10, 'hat');

echo "The 2nd element of the array is: $arr[1]";
```

Simple variable substitution works in double-quote strings and *heredoc* strings, but not in single-quote strings. If you want to print a $ symbol in these two types, you can escape it with a backslash character ( \ ).

# Arithmetic Operators

PHP comes with a full set of arithmetic operators, which operate on numbers. These are:

- Addition: +
- Subtraction: –
- Multiplication: *
- Division: /
- Modulus: % (This returns the integer remainder when dividing the second operand into the first.)

You can use these to build up expressions by simply providing two operands for each:

```
$x = 10 + 20;
$a = $b * $c;
$z = $f - 20;
```

All of these operators have what are called *self-assigning versions*, as follows:

```
$x += 10;
// this is the same as:
$x = $x + 10;
$x /= 10;  // divide $x by 10 and assign result back to $x
```

There are two other operators in PHP: increment, represented by ++, and decrement, represented by -- (minus minus). They can be placed before or after a variable, and behave slightly differently in each case:

`$x++` takes the current value of `$x`, and then increments its value by one behind the scenes. `++$x` immediately increments `$x` by one, and then uses that value in whatever context we are operating. The decrement operator similarly decrements the value of a variable before or after evaluating its value:

```
$x = 10;
echo $x++;  // prints 10, AFTER, $x is 11

echo ++$x;  // prints 12, AFTER $x is 12

echo $x--;  // prints 12, AFTER $x is 11
echo --$x;  // prints 10, AFTER $x is 10.
```

# Type Conversions

Everything that comes to us via the `$_POST` array is a string. However, our calculator wants to perform arithmetic operations on numbers. So, how do we convert from strings to numbers? The answer is that PHP implicitly does a lot of these type conversions for us.

```
$x = '10';
$y = '20';
$z = $x + $y;  // PHP first converts $x and $y to numbers, then adds them!

echo $z;       // prints 30!
```

Similarly, if we perform an operation that creates a result that doesn't fit into an integer, PHP will automatically convert it to a float:

```
echo 5 / 2;  // prints 2.5, a float!
```

For those cases where the implicit type conversions aren't what we want or we want to force something else to happen, we can use explicit type conversions, or *type casting*. We do this by using `(type)` in front of the value (constant or variable) to convert:

```
$x = 5.55;
$y = (int)$x;    // $y is now integer 5
$z = 2;
$a = (float)$z;  // $z is now 2.0 float.
```

When converting strings to numbers, PHP parses as much of a number as it can until a non-numeric character appears, and then stops. If the string does not start with any numeric characters, the conversion results in 0.

```
$x = (int)"123 Happy Lane";     // $x = 123 (integer)
$y = (int)"Three little pigs";  // $y = 0.
$z = (float)"123.45 is a number"; // $z = 123.45 (float)
```

When converting strings to Booleans, the rule is as follows:

- The string `"0"` or the empty string `""` evaluates to `FALSE`.
- All other strings evaluate to `TRUE`.

This means that both `'true'` and `'false'` convert to `TRUE` because neither is empty nor `"0"`.

# NULL

NULL (case insensitive) is a special value you can assign to variables in PHP. It means the variable is declared, but has no real value.

# The if Statement

The first control statement we'll see in PHP is the *if* statement, which allows for conditional evaluation of code depending on an expression. The basic syntax is:

```
if (expression)
   statement or code block
else
   statement or code block
```

If the expression evaluates to `true`, then the first statement or block of code (wrapped in `{` and `}`) is executed, otherwise the second block is executed. The *else* and second block are optional. You can chain together any number of these by using *else if* or *elseif*.

For example:

```
if (!isset($x))
   echo "\$x is not set yet!"
else if ($x != null)
   echo "\$x is not null.  Its value is: $x";
else
   echo "\$x is null.  No value, sorry!";
```

# Comparison Operators: Equality and Identity

PHP has a number of comparison operators. In this lesson, we will look at testing for equality. The `==` (equality) operator evaluates to `TRUE` if the two operands have the same basic value. This can, however, use PHP's implicit conversion, and will return `TRUE` even if the values are of different types:

```
// All of these evaluate to true:
10 == 10
10 == '10'
0 == false
0 == null
1 == true
```

The `===` (identity) operator, however, only evaluates to `TRUE` if the two operands have the same value *and* are of the same type:

```
10 === '10'      // false!
'10' === '10'    // true
0 === false      // false - different types
0 === null       // false - different types
```

Both have versions with the `!` character in them which evaluate to `TRUE` if the operands are not equal (identical):

```
// These all evaluate to true.
0 != 1
0 !== false
null !== false
11 != '10'
'11' !== '11'
```

lesson 4

# More Language Features

## Script Lifetime

We have mentioned in the past that each time the browser requests a page from the server, Apache starts up the PHP language engine to process the request. We've also mentioned that variables in PHP are valid for the duration of the currently executing script. Let's then look at the following script:

```
if (!isset($x))
{
      echo "initializing \$x";
      $x = 10;
}

echo $x;
$x++;
```

One might expect that the first time we run this script, `$x` would be initialized to 10, and then each subsequent time we run it, we'd see it incremented by one. But, in fact, this script prints out 10 every single time.

This is because PHP is completely stateless. Each time you start executing a new script, the language engine starts with a "clean sheet of paper," and everything initializes to empty. We will learn in later lessons how to remember state between different page executions.

## Strings, Newlines, and the Web Browser

We can insert newline characters into our double-quoted strings with "\n", and in *heredoc* strings by simply pressing the Enter key. However, when we display these strings in the browser, the newlines are not displayed. The problem is that our output medium, HTML, is a markup language that formats the output based on specially tagged instructions in the input stream.

If you look at the source for the Web page, you'll see the newlines rendered correctly, but if you want to see them in HTML, you'll have to do one of three things:

- Wrap the text to print in a *block-level* HTML markup element, such as `<p>` or `<div>`. These elements have implied newlines at the end of them.

- Use the preformatted output markup element, `<pre>`. This says that any content in between the `<pre>` and `</pre>` should be displayed exactly as it is shown. This is frequently used for showing code on a Web page.
- Use the line-break markup element, `<br>`, after the line of text you want to break. This is used less and less, as more people move to using proper block-level elements. If you have a string with lots of newlines in it, you can easily convert these to `<br>` tags with the `nl2br` function.

# Constants

For those cases where we want to use the same value throughout our code, PHP provides a convenient feature called *constants*. This lets us predefine values and give them a simple name to use throughout our script. When executing the script, PHP substitutes in the defined value before continuing.

Constants are declared with the `define` function. Once they have been defined, they can never be changed in the same script. When using constants, you do not need a `$` character as you do with variables.

```
define('ERROR', "Error: Something bad happened!");

if (!isset($x))
      echo ERROR;
```

Constants can be any scalar value, such as strings, numbers, Booleans, or null.

# More Comparison Operators

We've already seen the == and === operators, along with their negative counterparts. PHP has a few more comparisons:

- Greater than: >
- Greater than or equals to: >=
- Less than: <
- Less than or equals to: <=
- Not equals: <> (equivalent to !=)
- Inequality: !

All of these operators take two operands except for the last, which only evaluates to `TRUE` if its operand evaluates to `FALSE`.

# Logical Operators

We can combine expressions using comparison or arithmetic operators by using the logical operators in PHP. There are three of these:

- `and` (also `&&`)—This evaluates to `TRUE` if both its operands evaluate to `TRUE`.
- `or` (also `¦¦`)—This evaluates to `TRUE` if either of its operands evaluates to `TRUE`. If the first does, the second is not even evaluated.
- `xor`—This evaluates to `TRUE` if, and only if, *one* of the operands evaluates to `TRUE`.

You can combine expressions in PHP to arbitrary degrees of complexity. You can wrap expressions in parentheses to force a certain order of evaluation or help with readability. Otherwise, PHP has built-in rules describing in which order complicated expressions should be evaluated:

```
$x + 20 > 50 or $y - 430 < 1000
// same as
(($x + 20) > 50) or (($y - 430) < 1000)
```

You are encouraged to use parentheses as much as possible to help maintain some degree of readability in your code.

# The switch Statement

At some point, complex *if/elseif/elseif/else* statements can become cumbersome to read and maintain. For these situations, another control structure exists in PHP, called the *switch* statement. It basically takes an expression and evaluates it over possible values (called *cases*), then executes a block of code when a case match is found. You can optionally provide a *default* case for when no other one matches:

```
switch ($day)
{
      case 0:
            echo "Monday";
            break;
      case 1:
            echo "Tuesday";
            break;
      // etc
      case 7:
            echo "Sunday";
            break;
      default:
            echo "Unknown day of the week";
            break;
}
```

Once a case matches, PHP starts executing all statements inside the *switch* statement after the *case* match, ignoring any other *case* declarations. To have it stop, use the keyword *break*.

lesson 5

# Functions and Loops

## Loops

PHP provides several ways to iterate over the same code, called *loop* statements. We'll take a look at the most common types here.

### *while* loop

The basic syntax of the *while* loop is as follows:

```
while (expression)
    statement or block
```

The body of a *while* loop can either be a single statement or a block inside of brackets. The basic functionality of the loop is as follows:

- PHP evaluates the expression inside the parentheses. If it evaluates to `TRUE`, PHP executes the body of the loop.
- Repeat.

For example, to calculate one number to the power of n (that is, xy):

```
$base = 10;
$exponent = 3;

$result = $base;

while ($exponent > 1)
{
    $result *= $base;
    $exponent--;
}
```

Another type of loop is the *do…while* loop, which works like the *while* loop, except that it evaluates the expression *after* executing the statement(s) in the body. Thus, the body must execute at least once.

```
do
{
   // statements
}
while (expression);
```

Another type of loop is the *for* loop, which at first seems more complex than the others, but is actually quite useful and powerful:

```
for (expr1; expr2; expr3)
    statement or block
```

It works as follows:

- PHP executes or evaluates `expr1` once before doing anything with the loop.
- PHP evaluates `expr2`. If it evaluates to `TRUE`, it executes the body of the loop.
- After executing the body of the loop, PHP always executes `expr3`.
- PHP then goes back and evaluates `expr2` a further time. If `TRUE`, it continues executing the body and evaluating `expr3`. Otherwise, it breaks.

So, to do our exponent calculation using a *for* loop:

```
$base = 10;
$exponent = 3;

$result = $base;

for ($x = 1; $x < $exponent; $x++)
   $result *= $base;
```

To break out of a loop, regardless of what the expression would evaluate to, you can use the `break` keyword in PHP. To have the loop stop executing the body and go back to the expression evaluation, use the `continue` keyword.

# Functions

To declare a function in PHP, you use the `function` keyword. Afterward comes the name of the function, which must start with a letter or an underscore character ( _ ). The rest of the name can be any combination of letters, numbers, or underscores.

You can pass data to functions by passing arguments, or parameters to the function declaration. These look like regular variable declarations in the language.

```
function power_of($base, $exponent)
{
    $result = $base;
    while ($exponent --> 1)
        $result *= $base;

    return $result;
}
```

By default, all functions in PHP evaluate to `null`. To have a function return a value other than null, you use the `return` keyword, as shown above.

To call a function, simply name the function and give any parameter values in parentheses, separated by commas:

```
echo power_of(10, 3);  // prints 1000
```

You can provide default values for parameters in function declarations as follows:

```
function power_of($base, $exponent = 1)
{
    // etc.
}

echo power_of(10);    // prints 10^1, or just 10.
echo power_of(10, 2); // prints 100
```

To use a default parameter, you don't specify its value when calling the function.

# Byval and Byref Parameters

By default, PHP passes parameters to functions using a convention called by-value, or *byval*. This means that PHP passes a copy of the parameter to the function (this is true for numbers, strings, arrays, Booleans, and null).

```
$x = 10;

function f($y)
{
  $y++;
}

f($x);
echo $x; // prints 10 because $y is a COPY of $x
```

If you want a function to actually modify a parameter when using it, you want to use what are called by-reference, or *byref,* parameters. These are declared in the function header by prefixing them with the ampersand character ( `&` ). Byref parameters cause PHP to actually work with the outer variable in your script:

```
$x = 10;

function f(&$y)
{
  $y++;
}

f($x);
echo $x; // prints 11 because $y is a reference to $x
```

# Scoping

You can declare variables inside of functions in PHP. These variables are not accessible to code executing outside of the context of that function. Furthermore, you cannot use variables declared outside of a function from within that function's body. If you declare a variable inside of a function using the same name as one outside of the function, nothing unusual or bad happens: It's a local copy strictly for that function.

If you do want to use a variable from your outer script from within your function body, you make the function aware of that variable by using the `global` keyword.

```
$x = 10;

function f()
{
    global $x;
    $x++;
}

f();
echo $x;  // prints 11
```

There are some variables in PHP, such as the `$_POST` variable we've seen earlier, that are called *superglobals*. These are always accessible from anywhere within our script, and the *global* declaration is not necessary.

As for functions, if you declare a function inside of a script, you can use it from anywhere within that script, either before or after it is declared. To see if a function exists or not, you can use the `function_exists` function.

lesson 6

# Text and Strings

## Review of Strings in PHP

We've seen that there are three ways to enter strings in PHP: single-quoted, double-quoted, and *heredoc* strings:

```
$str1 = 'single quoted';
$str2 = "double quoted";
$str3 = <<<EODOC
this is a heredoc string.
EODOC;
$str4 = 'strings
can
span
multiple
lines';
```

To use single quotes inside a single-quoted string, we escape them with the backslash character. We do the same to insert double quotes inside of double-quoted strings. To print a backslash character, we escape it with a backslash:

```
$str1 = 'Marc\'s car';
$str2 = "Hello \"Marc\", if that IS your real name";
$str3 = 'I \\ love \\ backslashes';
```

Double-quoted strings can contain other escape characters, such as newlines, tabs, and linefeeds (`\n`, `\t`, and `\r`, respectively).

```
$str4 = "One\nword\n\per\line";
```

Any newlines or TAB characters in *heredoc* strings are preserved.

# More on Variable Expansion

We've already seen *simple variable expansion*, by which we can have PHP insert variable values into strings for us. This works for double-quoted strings and *heredoc* strings, but not single-quoted.

```
$thing = 'jar';
$contents = 'cookies';
$arr = array('fish', 'dogs', 'cats', 'birds');

echo "The $thing is full of $contents.";

echo <<<EOM

My favorite animal is:  $arr[2]

EOM;
```

There are two key limitations to the simple variable expansion. The first comes if we want to use string indexes for arrays, such as `$_POST['input1']`. The second comes if we want to embed a variable expansion right next to other letters or words. To solve both of these, we'll use complex variable expansion, which just involves us wrapping the variable expansion in brackets (`{` and `}`).

```
$thing = 'jar';
$contents = 'cookie';

echo "The $thing is full of {$contents}s";

echo <<<EOM

The "input1" textbox has the value: {$_POST['input1']}
EOM;
```

# String Operators

The key language operator on strings in PHP is the *concatenation operator*, or ".". There is also a self-assigning version of this operator, which concatenates the operand to the end of the string:

```
$str1 = 'happy';
$str2 = 'the cat is ' . $str1;
$str2 .= ".  Really happy.";
```

Unlike some other languages that use the PLUS sign ( + ) for string concatenation, in PHP, if you use this operator on strings, they will first be converted to numbers before the operation is evaluated as a numeric arithmetic operation.

If you use the == operator on strings in PHP, it will return `TRUE` if the two strings have the same text content. This comparison is case sensitive:

```
'fish' == 'fish'             TRUE
'Chupacabra' == 'chupacabra' FALSE
```

# Common Operations and Functions for Strings

While PHP contains a dizzying array of string functions, here are some that you'll use most frequently in your Web applications.

## strlen

The `strlen` function tells you how many characters there are in a string. If your string is holding binary data, then this function tells you how many bytes are in the data—strings in PHP are simply 8-bit byte sequences.

```
$str = 'happy happy 123';
echo strlen($str);    // prints 15
```

## strcmp

The `strcmp` function compares two strings in the same way that the == operator works on strings. It, however, does not evaluate to a Boolean. Instead, it returns:

- 0 if the two strings are the same
- 1 if the first string is considered to be greater than the second string ("F" > "B")
- -1 if the first string is considered to be less than the second string. ("m" < "z")

## strcasecmp

This function is similar to the `strcmp` function, except that it operates in a case-insensitive manner:

```
Strcasecmp('Fish', "FISH") == 0    // TRUE
```

## strncmp

This function lets you compare the first *n* characters of two strings, ignoring any others.

```
strncmp("I like motorcycles", "I like puppies", 6) == 0  // TRUE
```

## strtolower / strtoupper

These two functions are used to convert strings to and from upper and lower case. They are straightforward to use:

```
echo strtolower("I AM YELLING");     // prints i am yelling
echo strtoupper("feeling sluggish"); // prints FEELING SLUGGISH
```

## trim / ltrim / rtrim

These functions remove white space at both ends of a string (`trim`), or at one end (left – `ltrim`, right – `rtrim`). Any white space in the middle of the string is left untouched.

```
echo trim('       blah
');                       // prints 'blah'
```

```
echo rtrim("  fish  ");    // prints "  fish"
echo ltrim("  fish  ");   // prints "fish  "
```

## ord / chr

The `ord` function returns the ASCII character code for a given character in a string, while the `chr` function takes a numeric ASCII code and returns a string with that single character in it:

```
$num = ord('a');           // $num is 97
$str = chr($num);          // $str = "a"
```

## substr

This function gives you a way to extract a substring from another string. You provide it with the string to use, the starting index (zero-based) of the substring to extract, and how many characters long that substring should be:

```
echo substr('Wankel-Rotary Engine', 7, 6);    // prints "Rotary"
```

## strpos

If you want to find a string inside of another, this is the function to use. The first argument is the haystack in which to search, while the second is the needle you are looking for.

```
echo strpos("The house that I built", "house");   // prints 4
```

The return value of the function is the zero-based index of the first character of the string you are looking for, or `FALSE` if it's not found.

---

**Complex return values**

Many functions in PHP return an integer sometimes and FALSE in other situations (typically a failure situation). Since 0 would then be a valid success return value, you cannot use the following code to test failure:

```
$ret = substr("Cats are cute", "Cats");
if ($ret == false)
    echo "FAIL!";
```

This doesn't work because using this simple comparison, 0 would evaluate to false, and thus you'd see an unexpected "FAIL!".

To get around this problem, use the === or !== comparison operators:

```
$ret = substr("Cats are cute", "Cats");
if ($ret === false)
    echo "FAIL";
```

---

# Blowing Up Stuff (and Fixing It)

Two incredibly useful functions in PHP are the `explode` and `implode` functions. The former takes a string and explodes it around a specifier you provide, returning an array, while the latter takes an array of strings and implodes it, joining the strings with the provided glue:

```
$string = "It is hot today";
$arr = explode(" ", $string);
// arr = Array([0] => It, [1] => is, [2] => hot, [3] => today)

$string2 = implode(":", $arr);
// string2 is now: "It:is:hot:today";
```

# Sending Data via GET Parameters

We've thus far been using POST parameters to send data from one page to another in our Web application. Another way to send information is to add it to the URL you request for any given page. You can add parameters (name/value pairs) to this URL by using `?`, `&`, and `=` characters. These name/value pairs are called GET parameters, and work as follows:

```
http://site/page.php?first=value1&subsequent1=value2&sub2&sub3=value3
```

The `=` sign and value are optional.

These values are accessed from within your PHP scripts by using the `$_GET` superglobal, which is much like its `$_POST` counterpart.

```
echo $_GET['first'];               // prints value1
echo $_GET['subsequent1'];         // prints value2
var_export(isset($_GET['sub2']));  // prints "true"
```

# PHP and Unicode

Most modern Web pages are written in UTF-8, especially if they want to support more than simple ASCII characters. PHP was written to be internally 8-bit, which at first would seem quite incompatible with UTF-8 (in which letters can span multiple bytes).

The good news, however, is that PHP works quite well with UTF-8 if you use the multibyte string (*mbstring*) extension in the language and its string manipulation functions. Please consult the *PHP Online Manual* for more information on this extension and how to use it.

lesson 7

# Arrays, File Organization

## Review of Array Basics

Arrays in PHP are associative maps of key/value pairs, much like a dictionary or hash table data structure seen in other languages. Keys can be either strings or numbers, and there is no required or implied ordering to the contents of an array.

Reviewing what we saw in Lesson 2, arrays are created in two ways: with the `array` function or by assigning values to a variable with the array access operator `[]`:

```
$arr = array(234, 6823.32523, 'zebra', false);

$arr2[] = 'oink';
$arr2[] = 'wiggles';
$arr2[] = 9.5321e+11;
```

By default, PHP assigns numeric indexes to array contents. You can also provide string indexes for values. These can be specified in the `array` function call by separating names and values with the `=>` operator.

```
$user = array('name' => 'Marc', 'address' => '123 Happy Street',
              'favorite number' => 75, 'location' => 'Beijing');

$user['email'] = 'marcwan';
```

In general, it is not a great idea to mix numeric and string indexes within a given array.

To access values in an array, you use the array access operator ( `[]` ) with the index you want:

```
echo $user['email'];    // prints "marcwan"
echo $arr[0];           // prints 234
$field = "address";
echo $user[$address];   // prints "123 Happy Street"
```

# Multidimensional Arrays

Arrays in PHP can actually contain other arrays as contents. This array nesting can be arbitrarily deep. The array that results is referred to as a *multidimensional* array.

```
$the_matrix = array(array(1, 2, 3), array(4, 5, 6), array(7, 8, 9));
```

PHP provides some syntactic help for working with multidimensional arrays by letting you use multiple groups of `[]` operators on an array:

```
echo $the_matrix[0][2];       // prints 3
echo $the_matrix[2][0];       // prints 7
echo $the_matrix[5][43];      // invalid offset error message
```

# Counting Array Elements

To find out how many elements an array has, use the `count` function. This counts the number of top-level items in the array, and does not count the items in subarrays:

```
$arr = array(234, 5, true, array(3, 5, 1, 5, 6));
echo count($arr); // prints
```

If you provide an argument that is not an array, count returns 0.

# Removing and Deleting

To remove an element from an array, use the PHP `unset` function with that index:

```
$arr = array(1, 2, 3, 4);
unset($arr[0]);
var_dump($arr);
/* prints:  array(3) {[1]=> int(2) [2]=> int(3) [3]=> int(4)} */
```

To delete an entire array, simply use the `unset` function on the array variable:

```
unset($arr);    // Array $arr is no longer defined.
```

Indeed, you can use the `unset` function on any variable in PHP, after which it will no longer be declared or available to your scripts. If you attempt to reference that variable, you will get an error saying it has not been declared yet.

# Iterating over Array Contents – foreach Loops

PHP provides a very powerful way to iterate over array contents in your scripts, called the *foreach* loop. In its most basic form:

```
$arr = array(1, 2, 3, 4);
foreach ($arr as $value)
    echo $value;                // prints 1 2 3 4
```

PHP executes the body of the function once for each value in your array, putting the value each time in the variable whose name you provide (here we use `$value`).

For those cases where you want to actually see the keys as well, you can tell PHP to give you these in the *foreach* loop as well by providing a keyname variable and separating it from the value variable by a `=>` operator:

```
$arr = array(1, 2, 3, 4);
foreach ($arr as $key => $value)
    echo "$key: $value";          // prints 0:1 1:2 2:3 3:4
```

# Common Array Operations

There are a few functions that are very useful on arrays, beside the `count` function seen earlier.

## sort

There are actually a number of functions that can sort your arrays in PHP. They differ depending on whether you want to sort values or keys, sort forward or backward, or want to use your own sorting algorithm.

All take the array to sort and modify it directly in place.

```
$arr = array('fish', 'cat', 'zebra', 'tiger');
sort($arr);
// arr order is now ('cat', 'fish', 'tiger', 'zebra').
```

## array_merge

This function takes two arrays and merges them into one, returning the resulting array.

```
$arr1 = array(318, 68, 38, 23, 36.26, true);
$arr2 = array(342, 'cat', 325, 1968, 381.23861);

$arrm = array_merge($arr1, $arr2));
/* $arrm now has contents:
array(11) {
  [0]=>  int(318)  [1]=>  int(68)  [2]=>  int(38)  [3]=>  int(23)
  [4]=>  float(36.26)  [5]=>  bool(true)  [6]=>  int(342)
  [7]=>  string(3) 'cat'  [8]=>  int(325)  [9]=>  int(1968)
  [10]=>  float(381.23861)
}
```

## array_slice

This function extracts a section of an array and returns this as a new array, much like the `substr` function works on a string:

```
$arr1 = array(318, 68, 38, 23, 36.26, true);
$arr2 = array_slice($arr1, 2, 3);   // This has 3 values: 38, 23, 36.26
```

# File Inclusion

As our files and code libraries expand, we want to be able to manage them better, and maybe even reuse them in other Web applications on which we work. A common way to do this is to split them into different files, and then *include* or *require* those files in our other scripts.

This is done by using one of the following functions:

```
include
include_once
require
require_once
```

The difference between `include` and `require` is that the former only prints out a warning if the requested file cannot be found, and continues processing your script. On the other hand, `require`, will abort execution if the file cannot be loaded.

The `_once` versions of the function make sure that no matter how many times you ask to *include* or *require* the given file, it is only included once in your actual executing script. This prevents errors from redefining variables or functions.

PHP looks for requested files by first looking in the current directory (that of the currently executing script), and then looking through its *include_path*. This is defined in *php.ini*, and can be modified there, or at run-time in your scripts by calling the `set_include_path` function. You can inspect the current value by calling the `get_include_path` function. Any changes that are made to the *include_path* with the `set_include_path` function are only valid for the duration of the currently executing script. Each time PHP starts a new script, it goes back to the default values specified in *php.ini*.

You are free to use relative paths when including other files, and often this will help you avoid changing the *include_path* altogether.

lesson 8

# Object-Oriented Programming I

## Declaring New Types

Arrays are a powerful and interesting data type in PHP, but they have some limitations if you want to represent complex objects, such as users. You have to remember to use the same key names on each instance of an array; you must be sure not to forget to include the same key/value pairs each time you create one; and you must be sure to pass the arrays to functions that might modify them as *byref* parameters, because by default parameters in PHP are *byval*.

At some point, we would like to just be able to declare new types in PHP, somehow encompassing all the properties and operations that we would like to have associated with any given thing. To do this, we use Object-Oriented Programming (OOP), which lets us declare these new types.

Using objects, you declare *classes*, which contain *properties* ("member" variables describing the object) and *methods* (operations on the object and its property data). By combining these all into one place, we make our code easier to work with and less error prone.

To declare a class in PHP, we use the `class` keyword:

```
class User
{
}
```

To add properties to a class, you just declare variables inside the class, and we'll prefix those with `public` (more on this later):

```
class User
{
    public $Username;
    public $FullName;
    public $EmailAddress;
}
```

You can even set default values for member variables:

```
class A
{
  public $Variable = 123;
}
```

To create one of these objects, called an *instance* of that class, you use the *new* keyword in PHP:

```
$user = new User();
```

To access member variables and set their values, use the `->` operator:

```
$user = new User();
$user->Username = 'marcwan';
$user->FullName = 'Marc Wandschneider';
```

While we've been saying that everything in PHP is *byval*, objects in PHP are the exception to this: You always work with a reference to an object. When you pass an object to a function, it's not a copy of the object, just a reference to the same one. Similarly, if you do the following:

```
$a = new User();
$b = $a;
```

The variables `$a` and `$b` refer to the same underlying object instance.

To add an operation or a method to an object class, you simply declare the function inside the class:

```
class User
{
    public $Username;
    public $FullName;
    public $EmailAddress;
    public $Password;

    public function changePassword($new_pw)
    {
        $this->Password = md5($new_pw);
    }
}
```

To call a member function on an object instance, you also use the `->` operator.

```
$user->changePassword('secret key');
```

To refer to a member variable or function within our object class, we use the object pointer `$this`:

```
class User
{
    public $Username;
    public $FullName;
    public $EmailAddress;
    public $Password;
    public $LoggedIn = FALSE;

    public login($pw)
    {
```

```
        if ($this->checkPasword($pw) == false)
            return FALSE;

        $this->LoggedIn = TRUE;
    }

}
```

# Constructors

One of the problems with our object thus far, however, is that we can still forget to set member variables after instantiating it with the `new` operator. We'd like for there to be a way to force the user to provide all the object information necessary when creating the object instance. This is done with *constructors*. This is a function that is called as PHP creates a new instance of your class. To implement one, you create a function called `__construct` in your class:

```
class User
{
    // etc

    public function __construct($un, $fn, $pw, $em)
    {
        $this->Username = $un;
        $this->FullName = $fn;
        $this->Password = $pw;
        $this->EmailAddress = $em;
     }
}
```

Now, in order to instantiate a `User` object, we have to provide parameters to the *new* operator:

```
$user = new User('marcwan', 'Marc Wands...', 'secret', 'marcwan@');
```

Now, if we change the class, we can change the constructor, and PHP will immediately complain in all those places where we're not calling the constructor correctly—we've made it so that we can be sure to fix our code in all the right places!

# Access Levels

We've been declaring all of our member variables and functions with the *public* keyword thus far. Public means that anybody inside or outside of the class can access the variable or the function. Here are all three possible values for the access modifier:

- *public*—Anybody can access the member variable or function from within or outside of the class.
- *protected*—Only code inside of this class or one of its descendants can use the member variable or function. (We'll learn about descendants and inheritance in Lesson 9.)

- *private*—Only code inside of this class can access the variable or function. Nobody may view or modify it.

If you try to call a member function or access a member variable in a context where you do not have permissions, PHP will print an error.

lesson 9

# Object-Oriented Programming II

## Better Code Reuse with Inheritance

With our ability to declare new types, we have solved one major problem previously facing us in PHP. We still do not have a solution, however, for the cases where we'd like to have two classes that are basically the same, and share much of the same code, or we'd like to declare a new type that is an extension or slight modification of an existing type.

To solve this problem, we use another feature of object-oriented languages called *inheritance*. With it, we can declare a class as *extending* from another class, or *inheriting* from it. It would thus also inherit all the basic properties and operations from the *base* or *parent* class, and we could then add our own new features or modify any existing behavior as necessary.

To extend a class in PHP, we use the `extends` keyword:

```
class SuperUser extends User
{
    protected $Permissions;

    public function deleteExistingUser($username)
    { // etc }
}
```

We do have one problem to solve with these inheriting classes: What if we want to override some functionality in our parent class, but still use that functionality in addition to our new code? We solve this by using the parent keyword and the `:: scoping` operator, which tells PHP to execute the same function on our parent class:

```
class SuperUser extends User
{
     public $Permissions;

     public function __construct($un, $fn, $pw, $em, $prms)
     {
          parent::__construct($en, fn, $pw, $em);
          $this->Permissions = $prms;
     }
}
```

To reference a member variable defined in our parent class, just use the `$this` pointer as before:

```
class SuperUser extends User
{
    // etc

    public function printUser()
    {
        echo "HAHAHA! I am a SuperUser: " . $this->Username;
    }
}
```

With object inheritance, we are now free to create hierarchies with objects that share as much code as we want.

# Further Refining Our Objects

For those cases where we declare a base class that we actually only *ever* want to use as a base class, and not have anybody instantiate, we can tell PHP to enforce this by declaring the class as *abstract*:

```
abstract class A
{
    public $Var1;
}

class B extends A
{
    public function printVar() { echo $this->Var1; }
}

$a = new A();  // Error! It's abstract!
$b = new B();  // OK!
```

We can actually declare member functions in an abstract class as *abstract* themselves. This means that any inheriting class must implement that function or else PHP will consider it abstract and flag an error if you create an instance of it:

```
abstract class A
{
    // Inheriting classes MUST implement:
    public abstract function method1();
}

class B extends A
{
    public function method1()
    {
```

```
        echo "This is method 1";
    }
}
```

If you declare a method on a class as abstract, you must also declare the class as abstract.

If we want to declare a method in our class that no inheriting class can override, we can use the `final` keyword to ensure this:

```
class A
{
    public final function method1()
    {
        echo "You can't override me, MUAHAHAHHA!";
    }
}

class B extends A
{
    public function method1() // PHP will generate an error here.
    {
        // etc
    }
}
```

lesson 10

# Object-Oriented Programming III

## Class Constants

We can assign constants to our classes in PHP by using the `const` keyword.

```
class SuperUser extends User
{
    const PERMS_DELETE_USERS = 0x1;
    const PERMS_CREATE_USERS = 0x2;
    const PERMS_MODIFY_USERS = 0x4;

    // etc.
}
```

These are much like regular constants in PHP, but they are now closely coupled to our new data type. To refer to these constants in regular code, you scope the reference with the class name and `::` characters:

```
if ($su->Permissions == User::PERM_DELETE_USERS) ...
```

To refer to these class constants from within the class, we can either use the same `class_name::` syntax, or instead we can use `self::` as follows:

```
class SuperUser extends User
{
    const PERMS_DELETE_USERS = 0x1;
    const PERMS_CREATE_USERS = 0x2;
    const PERMS_MODIFY_USERS = 0x4;

    public function canDeleteUser()
    {
        if ($this->Permissions == self::PERMS_DELETE_USERS)
            // etc
    }
}
```

# Static Class Data

You can actually scope global variables inside of a class in PHP by declaring a member variable as static. These variables are not associated with any particular instance of that class, but are a way to associate data with that class "namespace."

```
class User
{
     public static $s_num_instances;

     public function __construct(...)
     {
         self::$s_num_instances++;
         // etc
     }
}
```

To reference these static class variables, you again can use the class name or `self` keywords and the `::` scoping operator.

# Static Methods

You can also associate functions with a class's namespace without making it an instance method by declaring it as *static*:

```
class User
{
     public static $s_num_instances;

     // etc

     public static function instCount()
     {
        echo "There have been " . User::$s_num_instances
              . " instances created while running this script";
     }
}

User::instCount();
```

Referencing these static functions is the same as for member variables or constants.

# Operations on Objects

PHP lets us use a number of the operators available for other types on objects.

## Comparison

If we use the == operator on two objects, PHP indicates `TRUE` if the two objects are of the same class and have the same values for all of their instance variables (even if they're separate instances).

If we use the `===` operator on two objects, PHP indicates `TRUE` if, and only if, the two objects are the exact same object instance.

```
$user1 = new User('marcwan', 'Marc', 'secret', 'emailaddr');
$user2 = new User('marcwan', 'Marc', 'secret', 'emailaddr');

$user1 == $user2          TRUE!
$user1 === $user2         FALSE - not the same instance
```

## Converting to String

We might eventually want to convert objects to strings in PHP, either with the `(string)` type-cast operator, the `tostring` function, or simply by typing something like:

```
echo "The current user is: " . $user1;
```

Unfortunately, by default, PHP just prints out an error saying it doesn't know how to perform this conversion.

We can, however, provide the code so that PHP *can* do this conversion. This is done by implementing a function in our class called `__toString`:

```
class User
{
    // etc

    public function __toString()
    {
        return <<< EOM
UserName: {$this->Username}
FullName: {$this->FullName"}
EOM;
     }
}
```

## Iterating over Object Properties

PHP will let us use the *foreach* loop on objects. When we do this, each of the member properties become the values over which we iterate:

```
$user = new User('marcwan', 'Marc', 'secret', 'emailaddr');

foreach ($user as $prop => $value)
    echo "$prop:    $value";
```

## Copying Objects

As we explained, when assigning an object variable to another, PHP just copies a reference to the object. For those cases where we truly want to copy the object, we have to do something else. For this, PHP provides the `clone` keyword, which causes it to create an instance of the same class and copy over all the same values for the member variables.

```
$user1 = new User('marcwan', "Marc", 'secret', 'emailaddr');

$user2 = clone $user1;      // $user2 is separate instance
```

If the default variable copying behavior is not sufficient, then you can implement a `__clone` function in your class, which PHP will call when cloning to let you perform other operations.

# Structured Exception Handling

One problem with our objects so far is that there is no way to report errors in the constructor: Constructor functions cannot return values since they are invoked only by the `new` operator. Indeed, even in other places in our application, we would like a better way to handle errors.

The solution to this in PHP is to use what is called *structured exception handling*. With this, you declare classes of errors, all of which inherit from the `Exception` class and each of which signals a specific error condition.

Once we have determined that an error condition exists, we create an instance of the appropriate exception class, and then we *throw* this exception. Other parts of our code can then *catch* this exception, and depending on exactly what type it was, either process it, ignore it, or *rethrow* it for somebody else to process.

A basic example would be as follows:

```
class MyInvalidArgumentException extends Exception { }
class User
{
    // etc

    public function __construct($un, $fn, $pw, $em)
    {
        if ($un == '' or $fn == '' or $pw == '' or $em == '')
            throw new MyInalidArgumentException();
        // etc
    }
}
```

To work with exceptions, we use a *try/catch* block in our code. We effectively try an operation or set of operations, and then catch any errors that this code throws:

```
try
{
    $user = new User('', '', '', '');
}
catch (MyInvalidArgumentException $e)
{
     echo "Invalid Arguments provided to User constructor!!!";
}
catch (Exception $e)
{
     echo "Augh!  Some other exception was thrown!  Re-throwing!!!";
     throw $e;
}
```

PHP processes *try/catch* blocks as follows:

- PHP executes the body of the *try* block.
- If an exception is thrown, PHP starts working through the various *catch* blocks (there can be any number of them, provided each has a different specific class of exception). For each block, PHP checks to see if the thrown exception is of that type or one of its parent types.
- If the *catch* block matches, PHP executes that body.
- If no *catch* block matches, PHP assumes the exception is *uncaught*, and looks elsewhere for somebody to catch it.

If no block of code anywhere catches an exception, PHP has a default exception handler that just aborts script execution and prints out a message about the exception.

When we say "PHP looks elsewhere" for somebody to catch the exception, we are referring to what is called the *call stack* in PHP. As we call functions, that in turn call other functions, that in turn call other functions, and so on, we are building up a "stack" of function calls. When we throw an exception, PHP aborts all code execution and starts walking back up the stack. If the current function body doesn't catch the exception, then PHP stops executing code in it, and sees if the current function call is in a *try/catch* block in the parent function on the stack. If not, it keeps unrolling the stack until a try/catch block catches the exception, or there are no functions left on the stack, and PHP resorts to the global exception handler. If a *try/catch* block does catch the exception, then that function can resume execution of code after the *catch* block.

We will generally try to avoid using the global exception handler, and instead be aware of all the different ways our pages can fail as we write the code for them.

# lesson 11

# Learning about the Web Server

## More FORM Elements

We have seen a wide variety of elements we can put in forms, including text boxes, select/option drop-downs, and buttons. We even saw radio buttons in Lesson 6 with our simple calculator. We'll introduce two more now—checkboxes and text areas—just to round out our knowledge of forms.

### Checkboxes

You can add checkboxes to your forms by using the `INPUT` markup element, but with the type `checkbox`. The input element has no place for the text of the checkbox, so you put this in markup next to the element:

```
<input type='checkbox' name='remember'> Remember Login
```

Checkboxes are, by default, not checked. To indicate in markup that you would like a checkbox to be checked, you use the `checked` keyword, much as we did for the radio buttons previously:

```
<input type='checkbox' name='remember' checked> Remember Login
```




To find out, in our server scripts, whether a checkbox element was checked when the form was sent to us, we look to see if the appropriate entry in `$_POST` is set to `"on"`.

```
if ($_POST['remember'] == "on")
    remember_login();
```

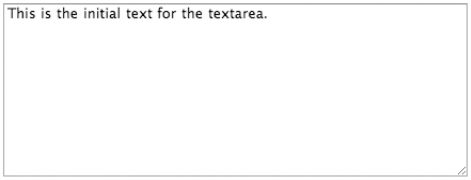
### Text Areas

The text input boxes we have seen thus far have all been limited to a single line and don't typically provide a lot of space. For those situations where you'd like to let your users write more, such as a user bio or a letter, you can use the HTML element TEXTAREA. This element also has the name attribute, as well as the number of rows and columns you would like the text area to have.

```
<textarea name='user_bio' rows='10' cols='50'></textarea>
```

To place some initial text in the text area, you put it between the opening and closing tags:

```
<textarea name='user_bio' rows='10' cols='50'>
This is the initial text for the text area.
</textarea>
```




Data from text areas in our forms are sent to the server in the same way a regular textbox's data are sent: by just accessing the appropriate slot in the `$_POST` superglobal:

```
echo $_POST['user_bio'];
```

# POST vs. GET

All of the forms we have been submitting thus far in our Web applications have used the "POST" method. We also saw once in our simple calculator application that we used GET params. Let's finally solve the mystery of what those two terms mean now.

When your Web browser requests a page from the server, it makes a network connection and then proceeds to talk to the server using the HyperText Transfer Protocol, or HTTP. This is a simple request/response protocol, in which the browser or client application makes a request of the server, and the server responds.

These requests are made up of a number of *headers*, and then optionally followed by a *body*. The response also is made up of a number of headers and then the body of the response contains the content for the browser to render.

When your browser makes a request of the server, it can choose among a number of types of requests, two of which are `POST` and `GET`. `GET` requests simply tell the server that we want the contents of the given URL (which might have some `GET` parameters tagged on to the end of it). `POST` requests, while also requesting the contents of a given URL, sends a request body to the server with some data for the server to process.

Your computer comes with a little program called *telnet* (*telnet.exe* on Windows) that lets you connect to your Web server and simulate HTTP conversations. By using this, we can also figure out why we might want to check input parameters so carefully in our server scripts, despite the fact that forms clearly specify all possible values:

```
Murasaki:Sites marcw$ telnet livelessons http
Trying 127.0.0.1...
Connected to livelessons
Escape character is '^]'.
POST /Lesson06/results.php HTTP/1.1
Host: livelessons
```

```
Content-type: application/x-www-form-urlencoded
Content-length: 42

radix=dec&input1=10&input2=3&operator=FISH
```

After entering all of these headers, the server then sends us the response:

```
HTTP/1.1 200 OK
Date: Mon, 26 May 2008 05:26:50 GMT
Server: Apache/2.2.6 (Unix)
Content-Length: 205
Content-Type: text/html

<html>
<head>
  <title>Results</title>
</head>
<body>
  <h3>Your Results!</h3>
  <p> The final result is: ERROR: Invalid Operator</p>
  <p> <a href='calc.php/'>start again</a></p>
</body>
</html>
```

If you ever want to simulate a simple page request that your browser does, you can just use the `GET` request:

```
GET /Lesson06/calc.php HTTP/1.1
Host: livelessons
```

The PHP software, once it gets the request from the Apache server, will help us out and convert the `POST` and `GET` data into the `$_POST` and `$_GET` superglobals, respectively.

One downside to using `POST` parameters is that many programmers believe that, since users can't see them, they're inherently more secure than `GET` parameters (which are easily visible in the address bar of the Web browser). This is, however, a false sense of security, as HTTP operates over the network in simple plain-text format, and anybody with a program like *tcpdump* who is watching your network traffic can see what's going on.

## GET vs. POST in Our Applications

To decide when to use GET parameters and when to use POST parameters, the following guidelines are typically used:

- For data that only modify the output or display of your page, such as sorting on a particular column, refining the contents displayed with a filter or search string, or otherwise effecting a change of an ephemeral nature, use GET parameters.
- For other data, use POST parameters.

These are not concrete rules, but rough guidelines that many users try to follow.

# The $_SERVER superglobal

In addition to the `$_GET` and `$_POST` superglobals we've seen already, another one we will use often in our Web applications is the `$_SERVER` superglobal, which contains all sorts of information about the current server configuration and the incoming request that caused this script to start executing.

You can see the full contents of it by using the *var_dump* function. We'll look at a couple of interesting members here:

## PHP_SELF

This key in the `$_SERVER` array tells us the URI of the currently executing script, relative to the root of the Web site being accessed. For example, if the user asked to see:

```
http://www.cutefluffybunnies.com/scripts/showbunnies.php
```

a request to see `$_SERVER["PHP_SELF"]` would return `/scripts/showbunnies.php`. Please note that if we ask for the value of this from within a script that is included in another script, the outermost executing script (the one that performed the inclusion) will be the value returned here.

## SERVER_NAME

This is the name of the server to which the request was sent. It will not be prefixed with the `http://` that one might expect, but will simply be the name of the server, such as `www.cutefluffybunnies.com`. This will correctly return the name of the requested *virtual server* for those situations where the current Web server is actually serving up the content for more than one named Web site (most modern Web servers support this feature).

## SERVER_SOFTWARE

This value will tell you what software the server is running. While there are few situations in which we will actually care about which server on which we are running, we can test the value in a manner similar to the following:

```
<?php

  if (strcmp(substr($_SERVER['SERVER_SOFTWARE'], 0, 6),
             'Apache') == 0)
  {
    // call some apache specific function
  }

?>
```

## SERVER_PROTOCOL

This value tells us via which protocol the client requested this page. The value will nearly always be "*HTTP/1.1,*" although it is possible that some clients will send us an older version (such as *HTTP/1.0*),

implying that some functionality will not be available or understood. We will learn more about the HTTP protocol in Lesson 13.

## REQUEST_METHOD

This is the data submission method used by the HTTP request sent to us. In addition to the `GET` and `POST` methods, which we learned about earlier in this lesson, this value could alternately contain `PUT` or `HEAD`, which we will not have much occasion to use. Although we can use this to learn whether a form was sent to us via `GET` or `POST`, we will generally know how our scripts are interacting, and will not often query this.

## REQUEST_TIME

This variable is not available under all servers, but for those that support it, it serves as a way to learn when a request was received by the server. For those who really need this information and are on a server where it is not provided, the `date` and `time` functions will serve as a reasonable compromise. The best place to learn more about these functions is via the PHP Online Manual, found at http://php.net/manual.

## DOCUMENT_ROOT

To find out in which directory we are executing code, we can query the `DOCUMENT_ROOT` field .

## HTTP_USER_AGENT

If you want to see via which agent (browser, program, etc.) the client has made the request for the page, you can look at this field. Values for a very old version of Mozilla.org's Firefox Browser on Windows will print the following:

```
Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.7.5)
   Gecko/20041107 Firefox/1.0
```

Some Web applications insist on being sure that the connecting program is a valid Web browser to try and prevent "bots" (automated programs that crawl the Web without requiring user interaction) from accessing their content. This is only a marginally effective tactic, however, as the user agent is an easily included field in the HTTP request.

We will take this opportunity to strongly discourage Web application authors from using this field to require users to visit their site with a particular browser. This is only likely to annoy prospective customers or users of your site, and will not save you significant amounts of work.

## REMOTE_ADDR

If you want to know (and perhaps even log in your database) from which IP address the client is connecting, this is the field to query. Although not foolproof, as advanced users can modify ("spoof") this on incoming packets, it can still be a useful tool for identifying people for some applications, such as public forums or discussion areas.

It should be noted that individual requests from the same user in the same "session" can, in fact, come from different IP addresses—depending on the Internet Service Provider via which the user is connecting to the Internet, data might be routed by multiple machines in a short span of time.

# FORM Security

What happens if somebody enters the following for his user name?

```
<script>alert('owned');</script>
```

If we have any code in our Web application that accepts user input and then prints it back out to the user, we will have discovered that users can cause arbitrary JavaScript to be executed on anybody who views that page's browser. While the example we created was a simple annoying dialog, there are much more mischievous things that can be done.

To prevent this, we absolutely must screen all user input in our applications. Two functions will help us with this.

## strip_tags

The PHP `strip_tags` function will attempt to remove any markup tags from a given string. For example:

```
strip_tags("<script>alert('owned');</script>");
```

will print:

```
alert('owned');
```

which is quite harmless.

This function is, unfortunately, pretty limited in its functionality, and pretty easy to get around. Thus, even if you use the `strip_tags` function, you will still want to use the next function.

## htmlspecialchars

This function takes a string and escapes any characters in it that could be part of HTML markup, rendering them completely harmless for display. For example:

```
echo htmlspecialchars("<script>alert('owned');</script>");
```

prints out:

```
&lt;script&gt;alert('owned');&lt;/script&gt;
```

With any tag markers such as < or > converted to HTML entities, anything the user has entered is now completely harmless (even if it's still uninteresting or unattractive).

# More on Processing Forms

If a user of our application fills in a form on a page and then clicks Submit, he will be taken to the results page. If the user, for some reason, tries to refresh the page, he might see the following:


This message is alarming and nonintuitive to most users, and it borders on too long. What we'd like is some way to avoid this for users altogether.

We do this by creating an intermediate *processing* page for our forms before *redirecting* users to the final results page in the application.

If we want to do some processing and then send the user to a new page from within PHP script, we will use a new function called `header`, which allows us to manipulate what HTTP headers are sent back to the client as we process a page. This function can be used to generate any HTTP headers, but we will use the `Location:` header for now. For example, our *process_new_user.php* might look similar to the following:

```
<?php

  // create new user account from $_POST information.
  // etc ...
  //
  $processing_error = create_new_user_account(
         $_POST['username'], $_POST['fullname'],
         $_POST['password']);

  //
  // Now go and redirect to welcome page if no error.
  //
  if ($processing_error === FALSE)
  {
    header('Location: http://' . $_SERVER['HTTP_HOST']
                               . dirname($_SERVER['PHP_SELF'])
                               . '/welcome_new_user.php');
  }
  else
  {
    // Send them back to the form entry page.
    header('Location: http://' . $_SERVER['HTTP_HOST']
                               . dirname($_SERVER['PHP_SELF'])
```

```
                                . '/create_account.php?err='
                                . $processing_error);
  }

?>
```

One critical point to note with the `header` function is that there can be *absolutely no text output* before the header function. Any white-space or other characters will cause PHP to start sending an output stream and will cause your `header` function call to generate an error. The following code, for example, will generate an error because of the spaces sent before the opening `<?php` tag:

```
     <?php

  header('Location: http://' . $_SERVER['HTTP_HOST']
                             . '/welcome2.php');
?>
```

Any white-space characters occurring outside of PHP section markers in any files included by our script will also cause the same problem!

# Serializing Objects

If we ever want to get a string representation of an object (or array) in PHP that we can either pass to another page or store in a database, we can use the `serialize` function:

```
$user = new User('marcwan', 'Marc W', 'secret', 'emailaddr');
$save_me = serialize($user);
```

This creates a string that contains all the information needed to save and later reload the object of type `User`.

To then recreate the object from the serialized string, you use the `unserialize` function:

```
$user = unserialize($save_me);
```

Note that if the class that is referenced in the serialized string is not currently declared in PHP when `unserialize` is called, the language will print an error message saying so.

To make a serialized object safe for passing between pages in `GET` parameters, you can use the `urlencode` function, which makes sure that any characters that are not allowed or have special meaning in URLs are processed properly:

```
 $url = 'ViewUser.php?user=' . urlencode(serialize($user));
```

Note that PHP automatically urldecodes `POST` and `GET` parameters for us when building up the superglobals.

# lesson 12

# Getting Started with the Database

## Getting to MySQL

One of the more common ways to access the MySQL server, with which we will be working through the next few lessons, is the MySQL command-line client. On Windows, we put this in C:\MySQL\bin\mysql.exe; on Mac and Unix machines, we put it in /usr/local/mysql/bin/mysql. Once the database server is running, we can then run this program with the name of the user to connect to and the `-p` flag, telling it to ask for a password:

```
/usr/local/mysql/bin/mysql -u root -p
```

Once we have entered our password (you *did* remember to write it down after installing in Lesson 1, didn't you?) we are in the client and connected to the server.

```
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 37
Server version: 5.0.45-log MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

Now you can try the following three commands and see what output you get:

```
SHOW DATABASES;
SHOW CHARACTER SET;
SHOW COLLATION;
```

The first of these just shows us which databases the server is managing, while the latter two tell us about characters and collations (sort orders) of which the server is aware. A vast majority of the Web applications we develop will use the UTF-8 character set in the database, and will tell the server to use the `utf8_general_ci` *collation* (UTF-8 case-insensitive sorting/ordering).

# Creating Our Database

For the PhotoShare application we're going to develop over the rest of these LiveLessons, we will create a database for our system, which we'll call `PhotoShareApp`. This is done with the MySQL query `CREATE DATABASE`:

```
CREATE DATABASE PhotoShareApp
    DEFAULT CHARACTER SET utf8
    DEFAULT COLLATE utf8_general_ci;
```

Now, if we run the `SHOW DATABASES` command again, we'll see our new database listed.

# Creating a Database User

While we play around with the MySQL command-line client, we will use the *root* user account, but within our Web apps, we will want to use more restricted accounts. To create one of these, you use the `GRANT` query, which takes a set of permissions to grant a user, the user to whom they should be granted, and the password via which they should be identified.

So, for our PhotoShareApp database, we'll create the following user:

```
GRANT SELECT,UPDATE,INSERT,DELETE
   ON PhotoShareApp.*
   TO 'psa_user'@'localhost'
   IDENTIFIED BY 'secret_password';
```

We have granted this new user the ability to fetch, change, insert, and delete data from all the tables (`.*`) in our PhotoShareApp database. The user's name is *psa_user* and the password is "secret_password," as shown above.

# Creating the First Table

We'll start by creating a table to represent users in our database. This is done with the `CREATE TABLE` command, and you specify each of the columns you want to have in your table, separating them by commas. A column specification is roughly as follows:

```
name TYPE extra_flags,
```

So, for our first version of the *Users* table, we'll have a user ID, user name, full name, password, email address, last login time, and privilege level.

Before we call `CREATE TABLE`, we must first tell the MySQL server which database we're using right now, and we do that with the `USE` query:

```
USE PhotoShareApp;
```

Now we're ready to create our table:

```
CREATE TABLE Users
(
  id INTEGER AUTO_INCREMENT PRIMARY KEY,
  user_name VARCHAR(100) NOT NULL UNIQUE,
  full_name VARCHAR(255) NOT NULL,
  password VARCHAR(32) NOT NULL,
  email_address VARCHAR(255) NOT NULL,
  last_login DATETIME,
  priv_level INTEGER NOT NULL DEFAULT 0,

  INDEX(user_name)
)
ENGINE = InnoDB;
```

We've given our table the name *Users*. Whenever we refer to this in queries or code, this is the name we will use.

- The first field is the user ID, and we've said it is of type `INTEGER`. We've also told MySQL to fill in the values for us by specifying `AUTO_INCREMENT`. Thus, we will never actually select an ID for a user—the server will always do this for us and will be sure to use a value that was not used previously.
- The *user_name* field we declared as a `VARCHAR` of length 100, which means the field will be a string of somewhere from 1 to 100 characters long. We have indicated that a value *must* be specified by indicating `NOT NULL`, and we have also said that no duplicate values are allowed by specifying `UNIQUE`.
- Our *full_name, email_address,* and *password* fields are simply variable character strings and must both contain a value that is non-null.
- The *last_login* field uses a type called `DATETIME`, which is how MySQL represents a date and time value in a single field. There are also `DATE` and `TIME` types for when both are not needed.
- We declare a field called *priv_level* to represent the user's privilege level. This is also of type `INTEGER`, must have a value, and will be given the `DEFAULT` value 0 if you don't specify a value when creating the row.
- We create this table using the language engine InnoDB. The two primary language engines you will run into initially when working with MySQL are InnoDB and MyISAM. The former is more robust and supports more features that make it better suited to multi-user transactional environments, while the latter is quite lightweight and fast, and has some nice text-searching functionality available.

There are two other things in our table creation code that we must look at now, and they relate to how MySQL finds data in a table. By default, when looking for something (a user ID or user name for example), it searches through all the rows in the table until it finds what it is looking for. This is extremely inefficient.

Thus, you have the option to build what are called *indexes*. These are special data structures that allow MySQL to find very quickly any data you're looking for in a given table. By declaring the user ID field as a `PRIMARY KEY`, we've made that an index for searches on user IDs (and a few properties we won't use for now).

However, we might also want to search for users giving only their user name (a login form, for example). Thus, we'll create an index for the *user_name* field, which we did by adding the `INDEX(user_name)` to the `CREATE TABLE` query. Now, whether we search for user IDs or user names, MySQL will be able to do so quite efficiently.

# Inserting Data into Tables

Now that we have a table, we will want to begin putting data in it. The primary way we do this is via the `INSERT INTO` query. The basic syntax is:

```
INSERT INTO TableName (fields to insert) VALUES (values for fields);
```

So, to insert a new user into our table we will want to do the following:

```
INSERT INTO Users
    (user_name, full_name, password, email_address, priv_level)
    VALUES ('marcwan', 'Marc W', 'secret', 'emailaddr', 1);
```

When inserting strings, we'll wrap those in single or double quotes. Also note that we didn't bother inserting a value for the user ID field, since MySQL will create one for us.

# Wrecking Things

To delete a table in MySQL, you use the `DROP TABLE` command:

```
DROP TABLE Users;
```

Please be aware that MySQL is extremely obedient and doesn't ask questions. If you tell it to remove the table, it will immediately and quietly do so. If you didn't mean to and don't have a backup, you are out of luck.

To delete an entire database, you similarly use the `DROP DATABASE` command:

```
DROP DATABASE PhotoShareApp;
```

Again, if you tell it to do so, MySQL will immediately do so. Make sure you really want to do this.

# Working with MySQL from within PHP

The primary means via which we'll interact with MySQL in our PHP scripts is the Improved MySQL extension, or *mysqli*. This system provides a nice object-oriented interface to the database server, and lets us manage connections and queries quite well.

To create a connection to the server, you just instantiate a `mysqli` object with the host, user name, password, and database to use.

```
$conn = new mysqli('localhost',
                   'psa_user',
                   'secret_password',
                   'PhotoShareApp');
```

Once we have the connection established, we will quickly execute the following query to make sure that all text data being sent to the database are interpreted as UTF-8:

```
$conn->query('SET NAMES "utf8"');
```

Now we're ready to do our `INSERT INTO` query. We do this by using the `query` function on the `mysqli` object:

```
$query = <<<EOQ
INSERT INTO Users
  (user_name, full_name, password, email_address, priv_level)
  VALUES ('bobotheclown', 'Bobo the Clown',
          'weak password', 'bobo@theinvalidclowndomainblah.com', 0)
EOQ;

$conn->query($query);
```

Once a query has executed, we test its success or failure by looking at the `errno` member variable on the `mysqli` object. If the value is 0, then all is well; but if not, then we can look at the `error` member of the `mysqli` object to see what the error message was.

```
if ($conn->errno != 0)
    echo "Something bad happened: " . $conn->error;
else
    echo 'It worked!';
```

# Securing User Data

Just as we had to worry about users sending us mischievous data via forms in HTML, we also need to worry about this security problem when we send user-form information to the database.

The problem arises when a user enters the following user name:

```
'; DELETE FROM Users;
```

As we will see in a later lesson, this will delete all of the rows from a given table, irrevocably and immediately. Such user input could wreak havoc on our databases and tables. If we weren't careful with user permissions and granted too many permissions, bad input could cause worse to happen (`DROP DATABASE`).

The solution to this problem is to use the `mysqli` function `real_escape_string` on any string data before sending them to the server:

```
$safe_un = $conn->real_escape_string($_POST['user_name']);
$safe_fn = $conn->real_escape_string($_POST['full_name']);

$query = <<<EOQ
INSERT INTO TableName (user_name, full_name)
               VALUES ('$safe_un', '$safe_fn');
EOQ;
```

lesson 13

# Fetching Data from the Database

## Retrieving Data

The primary query we will use to retrieve data from our MySQL tables is the `SELECT` query. The basic syntax is as follows:

```
SELECT what FROM where
```

To fetch all of the information our database contains for all of the users in our system, we would use:

```
SELECT * FROM Users;
```

The asterisk tells MySQL to fetch all columns for the table. If you only want user names and IDs, you might do the following instead:

```
SELECT id, user_name FROM Users;
```

We will not always want every single row in our table. In such cases, we can qualify the `SELECT` query by telling it how to restrict the *result set*. This is done with the `WHERE` clause, which can contain many kinds of expressions, comparisons, and functions.

```
SELECT * FROM Users WHERE id = 3;
SELECT * FROM Users WHERE id < 3;
SELECT * FROM Users WHERE id > 1000 OR id = 1;
SELECT * FROM Users WHERE last_login > '2008-08-01';
```

To test if a column has a `NULL` value in MySQL, you use some slightly special syntax:

```
SELECT * FROM Users WHERE last_login IS NULL;
SELECT * FROM Users WHERE last_login IS NOT NULL;
```

If we want to do partial matches on string fields, we can use the keyword `LIKE` along with the string *wildcard* character, `%`.

```
SELECT * FROM Users WHERE user_name LIKE '%ma%';
SELECT * FROM Users WHERE user_name LIKE '%ma';
SELECT * FROM Users WHERE user_name LIKE 'ma%';
```

The first query returns those users whose user name contains the string "ma" anywhere in it. The second returns only those users whose name starts with "ma," while the last returns only those whose name ends with the two characters.

It should be noted that `LIKE` matching is not terribly efficient, so we should try to add as many other qualifiers as possible to our query to restrict the amount of searching MySQL has to do. LIKE should definitely not be used as a means for implementing text searches.

# More on Query Expressions: Functions

You can use a number of MySQL functions in your queries to the database server.

Functions come in two formats: aggregate and scalar. *Aggregate* functions operate on a result set and look at all the values for a particular column in that set, while a *scalar* function operates on individual values and can be used to either effect the query or modify the values from the returned result set.

The most common aggregate function is `COUNT`, which returns the number of rows in the complete result set. For example, to count the number of users who have logged in since January 1, 2008:

```
SELECT COUNT(*) FROM Users WHERE last_login >= '2008-01-01';
```

Two other common functions are `MIN` and `MAX`, which return the minimum and maximum value, respectively, for a column in the result set. To find the user who logged in most recently:

```
SELECT MAX(login_date) FROM Users;
```

An aggregate function to compute the average value of a column, named `AVG`, is also available, among many others.

We'll look at a few of the more common other functions now.

- `CURRENT_DATE`—Returns the current date, for example, `'2008-09-19'`.
- `NOW`—Returns the current date and time, for example, `'2008-09-19 09:32:16'`.
- `YEAR`, `MONTH`, `DAY`—Returns the respective part of the given date or date/time field.
- `SUBSTRING`—Lets you extract part of a string, for example, `SUBSTRING(user_name, 1, 4)`. Note that indexes are offset from 1 instead of 0 as with the PHP `substr` function.
- `TRIM`—Returns the value of the given field value, trimming any white space from the beginning or end.

Let's look at some examples:

```
SELECT SUBSTRING(user_name, 1, 5) FROM Users
```

This returns the first five characters of the user name for all rows.

```
SELECT YEAR(last_login), MONTH(last_login) FROM Users;
```

This returns the year and month of the last login for all users.

```
SELECT * FROM Users WHERE YEAR(last_login) = '2008'
```

This returns all users whose last login date was sometime in the year 2008.

```
SELECT * FROM Users WHERE last_login = CURRENT_DATE();
```

This returns all users whose last login was today.

# Sorting Result Sets

To sort the data returned by a MySQL query, you use the `ORDER BY` clause in the `SELECT` query. You can specify one or more columns on which to sort, and for each column, whether it should be sorted ascending (`ASC`—the default), or descending (`DESC`). This clause always goes after the `WHERE` clause.

```
SELECT * FROM Users WHERE YEAR(last_login) > '2008' ORDER BY id;
SELECT * FROM Users ORDER BY DATE(last_login) DESC, user_name ASC;
```

The first query above just sorts the users by their IDs, while the second query first sorts the user by the date of the last login, and within those sorted dates, sorts the user names in an ascending order.

Please note that if you use the `ORDER BY` clause on a particular field, you should have an index for it, as MySQL will frequently be able to use that to help perform the sorting.

# Fetching Subsets of the Result Sets

If our system has a large number of users, we might want to present them to the viewer one page a time, perhaps only 25 or 50 per page. Having MySQL fetch all of the rows each time and then discarding a vast majority of them is horrendously inefficient. We'd like to be able to just specify that subset of rows right when doing the query.

This is done with the `LIMIT` clause in MySQL, which comes at the end of the `SELECT` query. You specify the row to start fetching at (0-based offset), and the number of rows to fetch:

```
SELECT * FROM Users LIMIT 101, 25
```

This fetches 25 rows, starting at the 101st row of the result set.

# Modifying a Table

On occasion, we will want to modify the structure in the database. We will use the `ALTER TABLE` query for this.

To change the type of a column—for example, to make our password accept 100 characters instead of just 32—we would enter:

```
ALTER TABLE Users MODIFY password VARCHAR(100) NOT NULL;
```

To add a new column to the table, we enter:

```
ALTER TABLE Users ADD COLUMN join_date DATETIME AFTER last_login;
```

This tells MySQL to add the new column with type date/time right after the *last_login* field.

lesson 14

# Modifying Data in the Database

## Modifying Rows in Our Tables

We have thus far been able to insert and retrieve rows from our database tables. Next, we would like to learn how to modify existing data in the database. To do this, we will use the `UPDATE` query in MySQL. Its basic syntax is as follows:

```
UPDATE table SET field1=value1, field2=value,... WHERE ...
```

So, to change the password for the user with ID 2:

```
UPDATE Users SET password = 'super awesome password' WHERE id = 2;
```

You can change as many fields as you want. If somebody wanted to change her user name and full name at the same time:

```
UPDATE Users
   SET user_name = 'maryjones',
       Full_name = 'Mary Jones'
 WHERE user_name = 'marysmith';
```

Note: You *must* use a `WHERE` clause with an `UPDATE` query, otherwise MySQL will simply change all of the specified values for all rows in the table. So be careful when using `UPDATE` and don't forget the `WHERE` clause!

## Deleting Rows from a Table

To delete a row from a MySQL table, use the `DELETE` query with a `WHERE` clause specifying which rows to remove.

```
DELETE FROM Users WHERE id = 2;
DELETE FROM Users WHERE YEAR(last_login) < '2006';
```

Note: Be very careful to use the `WHERE` clause! If you simply type:

```
DELETE FROM Users;
```

MySQL will remove all rows from the table, with no way to get them back.

# FULLTEXT Indexes

We mentioned in Lesson 13 that using `LIKE` in `WHERE` clauses was not a very efficient way to search through tables and that it should definitely not be used with advanced searches. Fortunately, there exists a way to get good search results from MySQL— by using a MyISAM table with a `FULLTEXT` index. (We talked about table engine types like MyISAM and InnoDB in Lesson 12.)

Our problem is that we would like to use InnoDB for most of our tables, since it scales a bit better for our application needs. So, in order to store a fully searchable user bio for our *Users* table, what we will do is create a second "helper" table to mirror our *Users* table. This table will contain only the user ID (to help match rows to the first table), and then a field of type `TEXT`, which allows for much bigger values than `VARCHAR` permits.

We will create an index for the user ID, so that MySQL can quickly find the matching row in the helper table, and then we will create a `FULLTEXT` index for the user bios, which will cause MySQL to set up robust searching on that column as well.

```
CREATE TABLE UserBios
(
  userid INTEGER NOT NULL,
  userbio TEXT,

  INDEX(userid),
  FULLTEXT(userbio)
)
ENGINE = MyISAM;
```

We will see in Lesson 15 exactly how to make proper use of these `FULLTEXT` indexes.

# Joining Fetches

The first new problem we have created for ourselves now is that we will have to fetch data from two tables to fetch all of the user data. To solve this problem, we will use a special type of `SELECT` query, called a *join*. By using these joins, we will be able to fetch data from multiple tables at the same time.

The basic technique is to specify from which tables you are fetching the data, and then specify a *join condition* to tell MySQL how to match up the rows from the two tables. We will include this as part of the `WHERE` clause.

```
SELECT Users.*, UserBios.*
  FROM Users, UserBios
 WHERE UserBios.userid = Users.id
       AND Users.id = 1;
```

The first part of the `WHERE` clause in the previous query tells MySQL that if you match up the *userid* in the *UserBios* table with the *id* of the *Users* table, the rows should be considered as one. Thus, our above query has retrieved all of the information we have for the user with ID 1.

You can create joins that are much more complicated than this, involving multiple tables, join clauses, and all sorts of extra conditions, but the performance degrades as the complexity increases, so we will endeavor to keep our joins reasonably simple.

## Transacting Queries

The second problem we have created for ourselves is that in order to insert a new user into our database, we now have to update two tables. We, however, want to make sure that exactly both of these `INSERT INTO` queries succeeds. If one fails, we don't want our database to be in an inconsistent state where there is half a user's information but the rest is missing.

The way we solve this is by using *transactions*. This lets us group a bunch of operations together and tell the database server "either all of these or none of these succeed." Once we have *begun* or *started* the transaction, we will either then *commit* the results, or *rollback* and abort the entire transaction. The MySQL commands for these, respectively, are:

```
START TRANSACTION;
COMMIT;
ROLLBACK;
```

Thus, to update a user, we would:

```
START TRANSACTION

INSERT INTO Users (fields ...) VALUES (...);
INSERT INTO UserBios (fields ...) VALUES (...);

COMMIT;   (or ROLLBACK if something bad happened)
```

You can see one problem we'd have with our new user insertion code: When we create the new row in the *UserBios* table, we need to know the user's ID from the *Users* table. We would like to avoid doing another query to get this ID, so how do we fetch it?

The good news is that the `mysqli` object comes to our rescue here with the `insert_id` property, which gives the value of the auto-generated ID from the `AUTO_INCREMENT` column of the last query. Thus, our code in PHP would become:

```
$conn->query('START TRANSACTION');
try
{
  $query = 'INSERT INTO Users ...';

  $conn->query($query);
  if ($conn-errno != 0)
    throw new DatabaseErrorException($conn->error);

  $query = 'INSERT INTO UserBios ...';
```

```
  $conn->query($query);
  if ($conn->errno != 0)
    throw new DatabaseErrorException($conn->error);

  $conn->query('COMMIT');
}
catch (Exception $e)
{
  $conn->query('ROLLBACK');  // abort
  throw $e;                  // re-throw to let somebody else process
}
```

# Using Hidden Fields on Forms

When we implement a modify user form in our Web application, we face an interesting problem: When we submit the form, the processing page needs to know exactly which user's data are being edited. One possible solution would be:

```
<form action='process_user.php?userid=234' method='post'
      name='modify_user'>
  <!-- etc -->
</form>
```

This feels a little odd for some reason, as the rest of the user's data are in the form, but we have to put their user ID in a `GET` parameter when calling the processing script. We'd like for there to be a way to put the user ID in the form along with all the other form data, but without it showing up visually on the page.

Enter the `input` element of type `hidden`. When you add one of these to the form, you can add a name and a value. This value is sent along with the rest of the form data to the server but is never shown visually on the page:

```
<form method='post' action='SubmitModifyUser.php'
      name='modify_user_form'>
     <input type='hidden' name='userid' value='{$user->Userid}'>
     <div>
       <label>Full Name:</label>
       <input type='text' name='full_name' value='$sfn' size='30'>
     </div>
     <div>
       <label>Email Address:</label>
       <input type='text' name='email_address' value='$sem' size='30'>
     </div>
     <div>
       <label>User Bio:</label>
       <textarea name='user_bio' rows='10' cols='40'>$sbi</textarea>
     </div>
     <p><input type='submit' value='Modify User'></p>
  </form>
```

Using these hidden input elements, we can embed as much data in our forms as we want and have them sent back to our server scripts for processing.

We should again note that just because these input elements are hidden does not make them secure or immune to mischief. Viewing the source of the page will show the hidden values—and, as always, anyone with the `TELNET` command can send arbitrary form data to your server.

lesson 15

# Remembering Things: Cookies and Sessions

## Carrying Information across Page Requests: Cookies

We have previously mentioned that HTTP is a simple request/response protocol, which maintains no state between requests. The solution to this problem, introduced in HTTP/1.1, is known as *cookies;* it's a mechanism enabling the server to send a small tidbit of information back to the client along with the server response for a given request. This tidbit consists of a name and a value, both sent as text strings. When the client subsequently sends another request to the server, it sends back any cookies it has for that server along with the new request, and the server can process them as appropriate.

### Setting Cookies

Setting cookies in PHP is easy, requiring the use of but one function: `setcookie`.

```
setcookie(cookie_name, cookie_value);
```

The `setcookie` function will make sure to escape the cookie value you give it so that it can be safely sent as part of an HTTP response message. The cookie name parameter should consist of alphanumeric characters only, although some special characters, such as underscores ( _ ) and dashes ( - ) are permitted. The resulting cookie will last as long as a client browser is open, and will be deleted when they are all closed.

The one trick to calling `setcookie` is that, like the `header` function seen previously, absolutely *no* output can come before this function is called. That is because cookies are sent as part of the response *header*, and if you send any output to the client, PHP will quietly send the headers before beginning the output stream for you. This can be somewhat tricky to always prevent, as even a single blank line or space character (that is not part of a PHP code block) in any part of the script—or indeed in any of the *include* files included from within the script—will cause output to be sent.

### Accessing Cookie Values

When the client browser sends a cookie back to the server with a request, you can access its value by using the `$_COOKIE` superglobal array. PHP only populates the `$_COOKIE` array, however, with cookies sent along *with the request to the page*. Thus, if we executed the following code before any other script had called `setcookie('loaded_okay', '...')`:

```
<?php

  setcookie('loaded_okay', 'TRUE');
  echo "{$_COOKIE['loaded_okay']}";   // not OK: not set yet!!

?>
```

we would receive a warning that the `loaded_okay` key in the `$_COOKIE` array had not been set. Only after a new page is loaded (and hence we have received a new request) will this array be populated correctly.

## How Cookies Work

The server sets a cookie in the client by sending a `Set-Cookie` header along with the response. Multiple cookies require multiple headers. As an example, consider a simple page to remember a color the user has selected. When the user clicks the Submit button after selecting a color, the browser sends an HTTP `GET` request to the server. That script, as part of its output, will send the following response headers (we will omit the body for brevity):

```
HTTP/1.x 200 OK
Server: Microsoft-IIS/5.1
Date: Fri, 07 Oct 2005 18:38:55 GMT
X-Powered-By: ASP.NET, PHP/5.0.2
Connection: close
Content-Type: text/html
Set-Cookie: user_color=green; path=/
```

The next time the user clicks Submit, the request will now look something such as:

```
GET /setcookie.php?colour=blue HTTP/1.1
Host: localhost
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; rv:1.7.3)
    Gecko/20040913 Firefox/0.10.1
Keep-Alive: 300
Connection: keep-alive
Referer: http://phpsrvr/setcookie.php?color=green
Cookie: user_color=green
```

Although multiple cookies are sent with individual `Set-Cookie` headers in the response, they are all sent back to the server in one single `Cookie` header in requests, each cookie name/value pair being separated by semicolons:

```
Cookie: user_color=orange; animal=moo+cow
```

## Controlling Cookie Validity

The `setcookie` function actually accepts up to six parameters, not only the two shown above.

```
  setcookie(name, value, expires, path, domain, secure)
```

We have already seen the *cookie name* and *cookie value* parameters. The remaining parameters are useful for controlling and restricting the availability of the cookie you set.

The *expire* parameter lets you control how long the cookie will be valid. It specifies the expiration date for the cookie, specified in seconds since midnight, January 1, 1970 (you find the current time in seconds since that date by calling the `time` function). For example, to specify that a cookie is to expire in one hour (there are *3,600* seconds in one hour), you might write:

```
setcookie('one_hour', 'still good', time() + 3600);
```

After a cookie has expired, the Web browser will no longer send it back to the server as part of requests. A value of *0* for this parameter (the default value if it is omitted) indicates that this should be considered what is called a *session cookie*. This means that it is stored in memory instead of on disk and is valid only as long as the user is browsing. Once the user closes down the Web browser, the value is lost.

The *path* parameter lets you restrict the set of pages within the site for which the cookie is valid. For example, if you set the path parameter to "`/`", then the cookie is valid for all pages in the site, but if you set it to "`/admin`", then it is only valid for those URLs on that site beginning with */admin*. If no path is specified, the default value is the directory in which the cookie is being set.

The *domain* parameter lets you restrict or expand the set of machines for which the cookie is valid. By default, it is valid for the server from which it was sent. But, if you have a large series of servers with names like *www1.example.com*, *www2.example.com*, *www3.example.com*, and so on, which are sharing the Web application load, then you might want to set this to "`.example.com`", which says that the cookie is valid for any machine within the domain.

Finally, the *secure* parameter merely lets you state that a cookie should be valid only over HTTPS connections by specifying a value of *1*. It defaults to allowing them over both secure and insecure connections, represented by the value *0*. This does not change the fact that the cookies are still stored as plain text on the user's computer.

## Deleting Cookies

There come times when you no longer need a cookie and would like to get rid of it. One option is to just leave it on the client machine, waiting for it to expire and not worrying about it at all. This, unfortunately, is probably not a good idea. Most cookies are stored on client machines in an easily-read text file. If the user was on a public machine at an Internet café or library, then any information in this cookie could be read by the next user.

Much better is to explicitly delete the cookie. Doing this in PHP is trivial: You merely set a new value into the same cookie, but with an expiration date in the past:

```
setcookie('loaded_okay', '', time() - 3600);
```

This causes the client to realize it has expired and clean it up.

# Sessions

Along with the ability to send small tidbits of information back and forth between client and server, PHP provides us with the ability to actually follow the progress of a particular client (user) as he or she works

through our Web application. This is done via the feature called *sessions*, which is integrated directly into PHP, requiring no extra compilation or installation efforts.

Sessions work by using *session cookies* (cookies with an expiration of *0*) and associate with the user or client a unique identifier called a *session id*, typically a long series of alphanumeric characters. The Web application can store data along with these sessions, and have these data move from page to page along with the user.

## Basic Usage

A session is initiated by calling the `session_start` function. This is called near the top *of each and every script* for which you wish the session to be active or valid. Calling this function makes sure that a session cookie has been assigned to the client, and initializes what is called the *session storage* for the session. If this is the first time the session is started, then new storage is created for the session, and the `$_SESSION` superglobal array representing this session storage is empty. If, on the other hand, there is already a session associated with this visitor when `session_start` is called, the `$_SESSION` array will have any session data as members.

The most basic example showing how to use sessions is one in which we create a session and store a single variable—such as one counting how many visits the user has made to a particular page. The code is as follows:

```
<?php

session_start();

// Create session variable if it doesn't exist yet.
if (isset($_SESSION['counter']))
  $_SESSION['counter'] ++;
else
  $_SESSION['counter'] = 1;

var_dump($_SESSION); echo "<br/>\n";
var_dump(session_id()); echo "<br/>\n";
var_dump(session_name()); echo "<br/>\n";
var_dump(session_get_cookie_params()); echo "<br/>\n";

?>
```

Each time you reload this page in the client Web browser, you will see that the `counter` variable inside of the `$_SESSION` array is increased by one. The output will look roughly as follows:

```
array(1) { ["counter"]=> int(7) }
string(32) "2412e9b2ac02dac44eae25d06b8601c7"
string(9) "PHPSESSID"
array(4) { ["lifetime"]=> int(0) ["path"]=> string(1) "/"
           ["domain"]=> string(0) "" ["secure"]=> bool(false)}
```

You access the session ID by calling the `session_id` function, while the `session_name` function shows you the *name* of the session. This session name is simply the name of the cookie that PHP returns to the client

to hold the session ID. It can be set in *.php*, or can be individually set by calling this function with a new name. Finally, the `session_get_cookie_params` shows you the details of the session cookie associated with the session.

### Configuring PHP for Sessions

There are a number of configuration options in *php.ini* to control the way in which PHP sessions operate. We will show the more interesting of these options now, and refer the reader to the PHP manual for other options.

- `session.auto_start`—We fibbed slightly above when we said that you must use `session_start` to initiate sessions in PHP. If this configuration option is set to *1* (it defaults to *0*), a session will be initiated every time a new page is requested in PHP.
- `session.name`—This is the name of the cookie sent to the client browser to hold the session ID. All users connected to our page(s) using sessions will get this same session name, but different session ids. The default value is `"PHPSESSID"`.
- `session.save_path`—If you are using the PHP session data storage mechanism (that is, when `session.save_handler` is set to "`files`"), this specifies the directory into which session data files are stored. It defaults on Unix machines to */tmp* and on Windows machines to C:\php\sessiondata.

## FULLTEXT Searching in MySQL

In Lesson 14, we created a MyISAM database table with a `TEXT` field and `FULLTEXT` index on that field. When you do this, MySQL can do quite powerful text searches on the field, provided there are enough rows in the table with which to build an index (usually, three to five rows is enough).

The syntax in MySQL to perform a full-text search is to use the `MATCH . . . AGAINST` expression in your query.

`MATCH(`*field_name*`) AGAINST(`*text to search for*`)`

So, to search through our user bios for those bios that contain the words *computer* and *programmer*, we could write:

```
SELECT Users.*, UserBios.*
  FROM Users, UserBios
 WHERE UserBios.userid = Users.id
       AND MATCH(UserBios.userbio) AGAINST('computer programmer');
```

The MySQL full-text searching functionality is highly configurable and tunable, and you should definitely consult the database documentation for this feature to learn more about it.

lesson 16

# Files and File Uploads

## Uploading User Files

Allowing file uploads in your PHP Web applications requires a few key steps:

- Configuring PHP to allow file uploads.
- Modifying your forms in HTML to have the browser send files to the server.
- Adding the server code to handle the recently uploaded content.

### Configuring PHP for Uploading

While PHP includes built-in support for file uploads, it typically requires some configuration before it can be used without any problems. There are a few key directives that you'll want to check in *php.ini*.

- `file_uploads`—Indicates whether or not file uploads are permitted at all. Defaults to *yes* (*1*).
- `upload_tmp_dir`—Must be set to a valid directory into which uploaded files can be temporarily placed to await processing.
- `max_input_time`—Specifies the amount of time (in seconds) a `POST` request may take to submit all of its data, after which it is cut off.

The `max_input_time` directive effectively limits the amount of time a client may stay connected to a particular server uploading the contents of a request (including any attached files). Thus, if our Web application is designed to allow users to attach 15MB files on a regular basis, and we expect them to be using normal Internet connections such as DSL or cable modems, then we will definitely need to increase the value beyond 60 seconds. For sites that are going to want to limit their data to maybe 500KB, then 60 seconds would be an entirely acceptable value.

Many installations of PHP come without the `upload_tmp_dir` configured at all. You will absolutely need to set this to some directory to which the user that the PHP server operates as has write permissions. If not, no uploads will succeed, and you may spend some time puzzling why that is. To try to be a bit more secure, we will use some empty and unimportant file system where no problems will be caused if it completely fills up.

```
upload_tmp_dir = z:/webapp_uploads     ; Windows
upload_tmp_dir = /export/uploads       ; Unix
```

## The Client Form

Modifying a form to allow file uploads in HTML requires two changes:

1. You add a new `<input>` markup tag with the `"file"` type.
2. You next add the `enctype` attribute to the form to show that you will use the new *multipart/form-data* MIME type.

To create a simple file upload form:

```
<form enctype="multipart/form-data"
      action="processfile.php" method="POST">

     Image File:
      <input name="userfile" type="file"/><br/>

    <input type="submit" value="Upload! " /><br/>
</form>
```

Once the user selects a file and clicks the Upload! button, the client browser sends a new request to the server.

## The Server Code

Once our request is on the way to the server with any files attached to it, we then must look at how to actually access these files on the server. This is primarily done through the superglobal array called `$_FILES`. This contains one element, with the key being the same name as the `<input>` field from the HTML file (in our previous example, this was *userfile*). The value of this is itself an array containing information about the uploaded file.

```
array(1) {
  ["userfile"]=> array(5) {
           ["name"]=> string(8) "fair.jpg"
           ["type"]=> string(10) "image/jpeg"
           ["tmp_name"]=> string(28)
                         "/export/uploads/phpC9.tmp"
           ["error"]=> int(0)
           ["size"]=> int(48823)
  }
}
```

One feature of file uploads in PHP is that they are not immediately placed for all to see in a visible place on the file system. When they are first received by the server they are placed in the location specified by the `upload_tmp_dir` directive in *php.ini*. From here, you must perform any validation on them (if necessary) and then move them to some other location. You find the location of the temporary file location by querying the `tmp_name` key in the `$_FILES` array for the appropriate uploaded file. Any uploaded files still in the temporary upload directory after script execution ends will be deleted by PHP in the name of security.

To process any uploaded file, we must perform the following actions:

1. Check the error code associated with that file to see if the upload was successful (more in a moment).
2. If the error code indicates the file was uploaded properly, perform any validation or antivirus scanning we wish to do on this file.
3. Once we are comfortable with the file, move it to whatever location we wish it to reside in. This should be done with the `move_uploaded_file` function.

The error field for our file in the `$_FILES` array will have one of the values shown in Table 16-1.

Table 16-1 Error Codes for File Uploads

| Code | Integer Value | Description |
|---|---|---|
| `UPLOAD_ERR_OK` | `0` | The file uploaded successfully. |
| `UPLOAD_ERR_INI_SIZE` | `1` | The file was larger than the value in `upload_max_filesize` in *php.ini*. |
| `UPLOAD_ERR_FORM_SIZE` | `2` | The file was larger than the value specified in the `MAX_FILE_SIZE` field of the form. This functionality is rarely used. |
| `UPLOAD_ERR_PARTIAL` | `3` | The file was not completely uploaded (usually because the request took too long to complete and was cut off). |
| `UPLOAD_ERR_NO_FILE` | `4` | No file was uploaded with the request. |
| `UPLOAD_ERR_NO_TMP_DIR` | `6` | There is no temporary folder specified in *php.ini*. (This error code was added as of PHP *5.0.3*). |

Thus, only if the error code in `$_FILES['userfile']['error']` is `UPLOAD_ERR_OK` (*0*) should we continue processing the file at all. In this case, we could do some validation, depending on how advanced our system is and what requirements we had. If we were allowing users to upload arbitrary binary data, we might want to run a virus scanner on the file to make sure it is safe for our networks. We might otherwise just wish to make sure the file is an image file and reject other types.

Once we have done this, we need to move the file from its temporary location to its final resting place (at least as far as this page is concerned). While this can be done with any file functions such as `copy` or `rename`, it is best done with the `move_uploaded_file` function. This ensures that the file being moved truly was one of those uploaded to the server with the request.

This helps to prevent possible situations where a malicious user could try to trick us into moving a system file *(/etc/passwd*, *c:\windows\php.ini*) into the location where we eventually put uploaded files. The `move_uploaded_file` function actually makes sure that the specified file was uploaded fully and successfully. Using this function and then checking the error result in the `$_FILES` superglobal significantly reduces the exposure to attacks through file uploading.

Our code in the file to process the uploaded thus becomes something along the following lines:

```
//
// Did the upload succeed or fail?
//
if ($_FILES['userfile']['error'] == UPLOAD_ERR_OK)
{
  //
  // Verify (casually) that this appears to be an image file.
  //
  $ext = strtolower(pathinfo($_FILES['userfile']['name'],
                             PATHINFO_EXTENSION));
  switch ($ext)
  {
    case 'jpg': case 'jpeg': case 'gif':
    case 'png': case 'bmp':
      break;   // File type is okay!
    default:
      throw new PSInvalidFileTypeException($ext);
  }

  //
  // Move the file to the appropriate location.
  //
  $destfile = '../user_files/' .
                basename($_FILES['userfile']['name'];
  $ret = @move_uploaded_file($_FILES['userfile']['tmp_name'],
                             $destfile);
  if ($ret === FALSE)
    echo "Unable to move user file!<br/>\n";
  else
    echo "Moved user file to appropriate directory<br/>\n";
}
else
{
  //
  // See what the error was.
  //
  switch ($_FILES['userfile']['error'])
  {
    case UPLOAD_ERR_INI_SIZE:
    case UPLOAD_ERR_FORM_SIZE:
      throw new PSFileSizeException();
      break;
```

```
    case UPLOAD_ERR_PARTIAL:
      throw new PSIncompleteUploadException();
      break;

    case UPLOAD_ERR_NO_FILE:
      throw new PSNoFileReceivedException();
      break;

    case UPLOAD_ERR_NO_TMP_DIR:
      throw new PSInternalError('no upload directory'); >\n";
      break;

    default:
      throw new PSInternalError('Unknown error!!');
      break;
  }
}
```

# File Functions

To read in the entire contents of a file into a variable in PHP, you can use one of two functions:

- `file_get_contents`—Reads in an entire file and puts all of its contents into the returned string.
- `file`—Reads in an entire file and returns an array—one element in the array for each line in the original file. Newline characters are included in these individual elements.

To write the contents of a string to a file, you can use the `file_put_contents` function in PHP. It takes the entire string and simply creates a file with those contents.

```
file_put_contents('userdata.txt', serialize($user));
```

## File Stream Functions

In addition to the simple one-line file functions, PHP provides a full set of functions to read, write, and seek on files. You begin by getting a *file pointer* with the `fopen` function. This function takes a filename to open and a mode in which it should be opened. Possible modes are shown in Table 16-2.

Table 16-2 Values for the Mode Parameter for the fopen Function

| Mode | Description |
|---|---|
| `r` | The file will be opened for reading (only), and the file pointer (see text) will be positioned at the beginning of the file. Writing to the file is not possible. |
| `r+` | The file will be opened for both reading and writing, and the file pointer will be positioned at the beginning of the file. Any writes before the end of existing content will overwrite that content. |

*continues*

Table 16-2 Values for the Mode Parameter for the fopen Function *(Continued)*

| Mode | Description |
|---|---|
| w | The file will be opened for writing (only). If the file exists, its contents will be deleted, and it will have a size of *0*. If the file does not exist, an attempt to create it will be made. The file pointer is placed at the beginning of the file. |
| w+ | The file will be opened for both reading and writing. If the file exists, its contents will be deleted, and it will have a size of *0*. If the file does not exist, an attempt to create it will be made. The file pointer is placed at the beginning of the file. |
| a | The file will be opened for appending (writing) only. If the file does not exist, an attempt to create it will be made. The file pointer is placed at the end of the file. |
| a+ | The file will be opened for reading and appending (writing). If the file does not exist, an attempt to create it will be made. The file pointer is placed at the end of the file. |
| x | The file will be created and opened for writing (only). If the file already exists, the function will return FALSE and a warning will be generated. If the file does not exist, an attempt to create it will be made. |
| x+ | The file will be created and opened for reading and writing. If the file already exists, the function will return FALSE and a warning will be generated. If the file does not exist, an attempt to create it will be made. |

Thus, to open an existing file for reading, we could use:

```
$f = fopen('userdata.txt', 'r');
```

To open a file for writing (which will create the file if it doesn't exist):

```
$f = fopen('newuserdata.txt', 'w');
```

The `fopen` function will fail if you attempt to open a file for reading that does not exist.

Once you have a file handle, the two key operations you will use on it are `fread` and `fwrite`, for reading and writing, respectively.

The `fread` function takes a file handler, and a number of characters to read, advancing the *file pointer* that many characters after each successful read. It returns those characters in the return value, a string. If the requested number of characters is greater than the remaining contents in the file, then a shortened string is returned and no further reads will succeed (they'll return `FALSE`).

```
$f = fopen('userdata.txt', 'r');
$entire_contents = '';

while (($str = fread($f, 512)) !== FALSE)
   $entire_contents .= $str;

$user = unserialize($entire_contents);
```

A handy cousin to the `fread` function is the `fgets` function, which assumes the opened file is a text file and reads one line at a time until there are no more. Newline characters remain in the returned string.

```
$f = fopen('userdata.txt', 'r');
$entire_contents = '';

while (($str = fgets($f)) !== FALSE)
   $entire_contents .= $str;

$user = unserialize($entire_contents);
```

To write to a file opened with one of the appropriate write flags in `fopen`, you use the `fwrite` function. You give the function a file handle and a string to write to the file:

```
fwrite($f, serialize($user));
```

Once you have finished working with a file using `fopen`, it is very important to close it properly using the `fclose` function. This ensures that all the file structures are cleaned up properly and any unsaved data are written to disk:

```
fclose($f);
```

# Browsing Directories

Viewing the contents of a directory is most easily done in PHP5 with a pseudo-object-oriented class called `dir`, which behaves largely like a regular PHP class, except in the way it is created:

```
$dir = dir('/path/to/some/directory');
  if ($dir === FALSE)
    throw new NoSuchDirectoryException();
```

Note that we do not use the `new` keyword for this class. To browse the contents of the directory, you call the `read` method until it returns `FALSE`.

```
  while (($entry = @$dir->read()) !== FALSE)
  {
    echo $entry;
  }
```

Note that we have to perform the explicit comparison `!==` instead of merely using `!=`. If we did not do this, and there was a directory entry named "`0`" (zero), the `!=` operator would say it is equal to `FALSE`! With the `!==` operator, there is no chance for this confusion, as it will verify that the value of `$entry` is typed as a Boolean.

Items returned by the `dir` class are in no particular order (they are not sorted for us). To go back to the beginning and start browsing through the contents again, we can call the `rewind` method on this class:

```
  $dir->rewind();
```

Finally, when we are done with the `dir` object, we call the `close` method on it:

```
  $dir->close();
```

lesson 17

# Formatted Output, Output Buffering, and Security

## Formatting Strings with printf and sprintf

One useful technique for customizing output in our programs is to use parameterized strings. In this scheme, you have a template string with some placeholders in it, and right before sending the string to output you insert data into those placeholders.

In PHP5, this functionality is obtained by using the `printf` and `sprintf` functions. They take a format string with placeholders and some parameters to insert into those placeholders. The placeholders all begin with the `%` character, and vary according to which type of data you are inserting (see Table 17-1).

Table 17-1 Type Specifiers for printf and sprintf

| Type Specifier | Description |
|---|---|
| `%` | Prints a % character. |
| `d` | Prints an integer value. |
| `f` | Prints as a (locale aware) floating-point number. |
| `s` | Prints as a string. |
| `b` | Prints an integer number in binary format. |
| `c` | Prints an integer number as the ASCII character with that value. |
| `e` | Prints a number in scientific notation (for example, 6.02214e23). |
| `u` | Prints an integer as an unsigned integer. |
| `F` | Prints a floating-point number in a non-locale-aware format (such as U.S. English). |
| `o` | Prints an integer number in octal format. |
| `x` | Prints an integer value in hexadecimal format (with letters in lower case). |
| `X` | Prints an integer value in hexadecimal format (with letters in upper case). |

To demonstrate a basic usage for inserting an integer and a string into a format string, we might execute the following:

```
<?php
  echo sprintf("There are %d books in %s's room.",
               $cbooks, $name);
?>
```

The output from this, if `$cbooks` were *123* and `$name` were *Michiko*, would be:

```
  There are 123 books in Michiko's room.
```

There are a number of key options we can include with this type specifier that further control how the output is generated.

- We can include a `+` sign before numeric type specifiers to indicate that positive numbers should have a number sign (instead of the default of only negative numbers having a sign). An example would be `%+d`.
- We can specify the number of decimal digits we would like for floating-point numbers by including `.##` before the type specifier `f`, where `##` is the number of decimal digits we want to see. For example, `%.10f` would cause us to show 10 digits after the decimal place.
- We can specify a width for the output data. This is a minimum width, and if the output is greater than this, it is not truncated at all. If the output is less than this, the output is padded from the left with spaces by default. An example would be `%10d`.
- We can specify the character to use for padding if our minimum width is greater than the width specified for the output. The default character is a space, but we can make it other characters by using a single quote ( `'` ) and the character we wish to use. This must be a single-byte character. For example, to use the _ character instead of spaces: `%'_10f`.
- Finally, we can also specify whether the padding should be on the right or on the left if our width is too wide. This is done with a minus sign ( `-` ), specified before the width specifier: `%-10f`. By default, padding occurs on the left.

To see all of these in action, we show some examples:

```
<?php

$floatv = 123456.78;
$negi = -123456;
$posi = 54829384;
$name = "Taleen";

echo sprintf("%d", $posi);          // prints: 54829384
echo sprintf("%d", $negi);          // prints: -123456
echo sprintf("+%d", $posi);         // prints: +54829384
echo sprintf("0x%x", $posi);        // prints: 0x344a148
echo sprintf("0x%X", $posi);        // prints: 0x344A148
```

```
echo sprintf("%e", $floatv);        // prints: 1.23457e+5
echo sprintf("%14.4f", $floatv);    // prints: 123456.7800    '
echo sprintf("%'_15.4f", $floatv);  // prints: ____123456.7800
echo sprintf("%-'_15.4f", $floatv); // prints: 123456.7800____
echo sprintf("%s", $name);          // prints: Taleen
echo sprintf("'%'_12s'", $name);    // prints: '______Taleen'

?>
```

# Date and Time Functions

## Time

The most common way to obtain a date/time value is with the `time` function. It, as well as the various file functions that give us time-based information about files (such as `filemtime`), return a 32-bit integer value, usually called a *timestamp*. The value represents the number of seconds that have elapsed since midnight on January 1, 1970 (sometimes referred to as "the Epoch"). This starting time is represented by the value 0, while the maximum positive 32-bit integer value of *2147483647* represents the evening of January 18, 2038.

Without some sort of processing of the integer value, the information is of limited use:

```
<?php

  $now = time();
  echo "The time is now: $now\n";

?>
```

The output of this would be something such as:

```
  The time is now: 1108437029
```

## Date

PHP provides a number of functions for the format and output of dates as strings. The most commonly used of these is the `date` function.

### The date Function

The `date` function allows for a very high degree of flexibility in the output of date and time information. The function has the following parameter list:

```
  string date($format[, $timestamp]);
```

The first parameter specifies the way in which we would like the date information formatted, while the second is optional and lets us specify the timestamp to format for output. If omitted, the current time returned by the `time` function is used.

The format string is basically a sequence of placeholder characters that represent various tidbits of information to be inserted, along with any extra white space or punctuation characters that are all ignored. The possible values of the placeholder characters are listed in Table 17-2.

Table 17-2 Format Placeholder Characters for the date() Function

| Character | Output | Description |
|---|---|---|
| a | *am* or *pm* | Lowercase *ante meridiem* (am) or *post meridiem* (pm) value (12-hour time only). |
| A | *AM* or *PM* | Uppercase *ante meridiem* (AM) or *post meridiem* value (PM) (12-hour time only). |
| B | *000* through *999* | *Swatch Internet Time* value. In this system, the day is divided into 1000 equal parts (each of which is 1 minute, 26.4 seconds long). |
| c | Something like: *2001-09-08T13:11:56-09:00* | The date printed in ISO 8601 format. |
| d | *01* to *31* | Day of the month, with leading zeros (if necessary). |
| D | *Mon* through *Sun* | Three-letter textual representation of the day of week. |
| F | *January* through *December* | Full textual representation of the month name. |
| g | *1* through *12* | 12-hour format of hour, without leading zeros. |
| G | *0* through *23* | 24-hour format of hour, without leading zeros. |
| h | *01* through *12* | 12-hour format of hour, with leading zeros. |
| H | *00* through *23* | 24-hour format of hour, with leading zeros. |
| i | *00* through *59* | Minutes, with leading zeros. |
| I | *1* if Daylight Savings Time, else *0* | Whether or not the date falls in daylight savings time. |
| j | *1* to *31* | Day of the month, without leading zeros. |
| l | *Sunday* through *Saturday* | Full textual representation of the day of the week. |
| L | *1* if it is a leap year, else *0* | Whether or not it is a leap year. |
| m | *01* through *12* | Numerical value of month, with leading zeros. |
| M | *Jan* through *Dec* | Three-letter textual representation of month. |
| n | *1* through *12* | Numerical value of month, without leading zeros. |
| O | Something similar to: *+0200* or *-0800* | Difference from Greenwich Mean Time (GMT), in hours. |
| r | Something similar to: *Mon, 13 Feb 1995 20:45:54 EST* | The date output in RFC 2822 format. |
| s | *00* through *59* | Number of seconds, with leading zeros. |
| S | *st*, *nd*, *rd*, or *th* | The English ordinal suffix for the day of the month (commonly used with the *j* format character). |
| t | *28* through *31* | The number of days in the month. |
| T | Something such as: *PST, EDT, CET* | The abbreviated name for the time zone used on the server machine. |

*continues*

Table 17-2 Format Placeholder Characters for the date() Function *(Continued)*

| Character | Output | Description |
|---|---|---|
| U | Output of `time` function | The number of seconds since January 1, 1970. |
| w | *0* (Sunday) through *6* (Saturday) | Numerical value of the day of the week. |
| W | *0* through *51* (*30* is the 30th week in the year, etc.) | The ISO-8601 week number in the year, where weeks always start on a Monday. |
| y | Something such as: *75* or *04* | A two-digit representation of the year. |
| Y | Something like: *1950*, *2049* | Full numerical value of the year, four digits. |
| z | *0* through *365* | The day of the year (starting at 0). |
| Z | *-43200* through *43200* | Time zone offset in seconds. West of GMT are negative values, east are positive. |

These placeholder characters can be put together in any order in the format string to create the output string:

```
<?php

  $time = strtotime('2005-09-12 21:11:15');  // see below

  echo date('Y-m-d H:i:s');  // 2005-09-12 21:11:15
  echo date('l, F jS, Y');   // Monday, September 12th, 2005
  echo date('c');            // 2005-09-12T21:11:15-09:00
  echo date('r');            // Mon, 12 Sep 2005 21:11:15 -0700

?>
```

The `date` function only prints output in the locale of the PHP server (that is, U.S. English). To see formatted output that correctly honors a different locale, you use the `setlocale` function and the `strtotime` function.

## Converting strings to timestamps

For those situations where somebody gives you a date, time, or timestamp in some sort of string format, you can use the `strtotime` function, which will go to Herculean lengths to try to parse it and determine the correct value. For most common formats and those standardized formats, the function performs exceedingly well:

```
<?php

  // yyyy-mm-dd
  $time = strtotime('2004-12-25');
  // mm/dd/yyyy
  $time = strtotime('12/25/2004');
  // RFC 2822 formatted date
  $time = strtotime('Mon, 13 Feb 1995 20:45:54 EST');
```

```
  // ISO 8601 formatted date
  $time = strtotime('2001-09-08T13:11:56-09:00');
  // it even understands some English words!
  $time = strtotime('yesterday');

?>
```

The RFC 2822 date format is frequently used in various Internet technologies, while the ISO 8601 format was specified by the International Standards Organization in an attempt to merge and reduce the myriad possible date/time formats currently in use.

# Output Buffering

We have repeatedly had to be careful throughout these LiveLessons to avoid sending any output before particular PHP functions, most notably the `setcookie` and `header` functions. A single blank line in any of our script files before a call to these functions causes a warning about headers already being sent and these functions not working correctly.

To help alleviate this problem, PHP includes a feature called *output buffering*, or output control. This is a group of functions that, when used, gather all of our output in a buffer before sending it to the client machine. This has the advantage of being able to wait for any and all `setcookie` and `header` function calls to execute properly before sending the output. Some users have even seen some performance improvements when using this extension.

## How It Works

Output buffering works by not sending the output headers and content right as they are emitted from within script (either through the `print` and `echo` functions, or by non-code blocks in the various script files being processed). Instead, it holds all of these in a memory area called a *buffer*, and only sends them (and any necessary headers) to the client when instructed.

There are functions to turn on buffering, get the current contents of the buffer, submit the current contents for output (also called *flushing* the buffers), discard the buffer, and close the output buffer, among others. This functionality is built in to PHP. No extra effort is required to compile or enable it at runtime.

Configuration of this feature area is limited to three options in *php.ini*, as follows:

1. `output_buffering` (default value of `"0"`)—If this is set to `'On'`, then output buffering will be enabled for all pages processed by the PHP engine. Instead of `'On'`, you can also specify a numeric value, which then sets the maximum size of the output buffer.

2. `output_handler` (defaults to `NULL`)—This controls through which function the buffered output is redirected before being sent to the client. By default, it is simply sent to the client when script execution ends or the buffers are flushed.

3. `implicit_flush` (defaults to `"0"`)—This controls whether the output buffers are flushed every time the user calls `print` or `echo`. This has some serious negative performance implications, and should only be used for debugging purposes.

## Using Output Buffering

Using output buffering in your pages is remarkably easy. You begin output buffering in your scripts by calling the `ob_start` function at the very top of them:

```
<?php
  ob_start();   // no output before me !!

  require_once('filea.inc');
  require_once('fileb.inc');

  etc ...
```

At the end of your script, when you are done with your output, you call the `ob_end_flush` function to end buffered output and cause the current buffered content to be sent to the client (along with any headers):

```
<?php
  ob_start();

  ...
  ..
  .

  echo "Thank you for visiting<br/>\n";

  ob_end_flush();
?>
```

Another way to end output buffering is to call the `ob_end_clean` function. This not only ends output buffering, but deletes the contents of the output buffers. This is most typically used in error situations when you want to then redirect the user to some other page, as follows:

```
<?php
  ob_start();

  // includes/requires here ...

  echo "Listing users from database:";

  $conn = @new mysqli('host', 'user', 'pwd', 'dbname');
  if (mysqli_connect_errno() != 0)
  {
    ob_end_clean();
    header("Location: showerror.php?err=" . $conn->errno);
    exit;
  }
```

```
  // otherwise, proceed as normal.

  ob_end_flush();
?>
```

One of the most compelling aspects of output buffering is that it is so easy to use! Apart from these functions, there are only a few others that we might occasionally make use of:

- `ob_flush`—Causes any current output in the buffers to be flushed (sent) to the client.
- `ob_clean`—Erases (cleans) the current contents of the output buffers. Output buffering remains enabled, however.
- `ob_get_length`—This returns the current size of the output buffer in bytes.
- `ob_get_content`—This function allows us to fetch the current contents of the output buffer, in case we wish to do additional processing on it before sending it to the client.

lesson 18

# When Things Go Wrong

## Errors in PHP

We will investigate four primary sources of errors in our Web applications.

### PHP Language Engine Errors

Although, theoretically, all errors in our Web application come from the PHP language engine (since it is what is running these applications), we will specially note those errors that occur because of problems with our code or the actual execution environment itself.

Examples of this include:

- Syntax errors in our code—We may have forgotten a semicolon, bracket, etc.
- Undeclared variable errors.
- Undefined reference errors—For example, if you try to create an instance of the `User` class, but PHP doesn't know about that class, it will throw an error.
- The PHP language engine couldn't start or run properly—Possible reasons include low system resources (memory) or an improperly configured *php.ini* file.

These errors are things we should try to hammer out as much as possible during development, and then take measures to note them carefully when executing our live Web applications.

### Application Errors (Bugs)

Most of the problems that we face when developing Web applications will be minor errors or incorrect assumptions in our code, more commonly known as bugs. For example, if we're not careful:

```
$x = load_int_value();
$y = 2523 / $x;        // will generate error if $x == 0

$x = 1
while ($x < 10)
{
    echo $x;           // whoops, forgot to increment $x;
}
```

If we don't check return values, or otherwise forget a simple little line of code, our application will have problems. Most of the time, when something goes horribly wrong, we'll either be told by PHP right away, or we'll notice that something isn't right—in the above infinite loop example, the script will take an unusually long time to execute (indeed, it will never stop until PHP terminates it).

### External Errors

Another source of errors in our applications will be from external components we might use, such as a MySQL database server or some other external resource, such as temporary files on the local file system. These will occasionally start to fail, and it's very important that our applications be able handle this gracefully and not start doing unpredictable things or, worse, creating security problems.

This is why, every single time we do a MySQL query or try to create a file in our Web applications, we should check the return codes very carefully and make sure things worked properly before continuing. Of course 99.99999 percent of the time it will, but you must be prepared for when it doesn't.

### User Errors

If you ask for a phone number in a form, and the user enters "Little Fluffy Bunnies", this is an error and something that must be flagged to them. This category of errors is a little different in that it's something the user can correct and we want to report it a little differently, but it is still something we must deal with in our applications.

## Handling Errors

Once an error condition occurs, we must be able to handle it correctly. Applications with random error messages littered across the pages look extremely unprofessional and usually don't impress visitors.

Here are some common ways to handle our errors.

### Thorough Debugging and Testing

For language and configuration errors, the best way to eliminate them will be to thoroughly test and debug our applications, making sure to cover as much code as possible. The more code we've run and tested, the more confident we can be that something will actually work. If we never run a particular piece of code, we might not find out until it's too late that it is broken.

### The @ Operator

For errors that come from external sources such as files, databases, or other external components, we can prefix the function call with the `@` operator, to tell PHP to not report any errors that occur when the function is called. This helps to keep our application nice and quiet and doesn't confuse the user, but has the big disadvantage of masking a potentially serious error condition of which we'd like to be aware.

We will typically only use the `@` operator for those situations where we know something is highly likely to fail, but for some reason, that's okay and we're willing to ignore it, or when nothing can be done about it.

### Setting Global Error Handlers

We will want to always set global error and exception handlers in our code that will log errors, perhaps e-mail key people, and otherwise tell the user something that something has gone wrong and you're aware of it. By doing this, you don't risk people learning unwanted details about your code, and you don't confuse your users with strange programming messages.

### Configuring php.ini Correctly

For development, we will always have errors turned on in *php.ini*, but for deployment to live servers, we will definitely turn these off. Instead, we will have PHP log the errors to a file so that we can periodically check it and see what is happening.

### Regular User of Structured Exception Handling

To handle user errors, and even a lot of system or resource errors, we will use structured exception handling in our applications. We can use `try` / `catch` blocks to make sure that we deal with the appropriate errors in the appropriate way, and we can let errors trickle up the call stack to let different functions handle different types of errors.

## Debugging with Xdebug

We've used `var_dump` a lot in the past to debug our Web applications, and while it is a very helpful way of doing things, it can prove a little limiting sometimes. To help us out, we'll use a popular debugging tool for PHP call *Xdebug*, developed by Derick Rethans.

### Installation

To install Xdebug on the Mac or on Unix, you can simply execute (as a super-user):

```
/usr/local/php5/bin/pecl install xdebug
```

PHP will do a bunch of downloading and compiling, and when it's done, in your */usr/local/php5/lib/php/extensions* directory, there will be a subdirectory with a complicated name like *no-debug-non-zts-20060613* or so, and in that will be an *xdebug.so* file.

Windows users should go to *http://xdebug.org* and download a precompiled version of *xdebug.dll* appropriate for their version of PHP.

For all platforms, to enable Xdebug in PHP, you have to add the following to your php.ini file:

```
zend_extension="/path/to/xdebug.so"
zend_extension="x:\path\to\xdebug.dll"
```

Note that you cannot just use the regular `extension=` syntax as for other PHP modules.

### Configuration

Xdebug is configurable to an insane degree, and has controllable options for just about everything. You are best off visiting http://xdebug.org for a full list of options, but as we look at the individual features, we will show one or two of the more common ones.

Xdebug is turned on by default when the extension is loaded. To turn it off, you should just comment out the `zend_extension` entry in *php.ini* by putting a semicolon in front of it.

## var_dump Extensions

Xdebug provides a replacement for the `var_dump` function which gives you nicer formatting and slightly better information than the regular one. It is also formatted to look better in the browser as opposed to forcing you to view the source all the time.

```
array
  0 => int 1
  1 => int 2
  2 => int 5636
  3 => int -3523
  4 => string 'cat' (length=3)
  5 => boolean true
  6 => float 342.23
  7 => int 325
  8 => float 5.55E+10
  9 => boolean false
  10 => int 234324
```

By using color and nicer formatting, we can see our variables more clearly.

For those cases where we have objects containing arrays containing objects and arrays and so on, Xdebug has configuration options that will let you control how many children to display in an array, and to what level, or depth, it will keep expanding objects and array values:

- `xdebug.var_display_max_depth` – Defaults to 3, controls how many levels of objects and array values deep to traverse.
- `xdebug.var_display_max_children` – Defaults to 128, controls the maximum number of values to display from arrays or member variables on objects.

You can set these values in *php.ini*, or you can use the `ini_set` function at the top of your scripts:

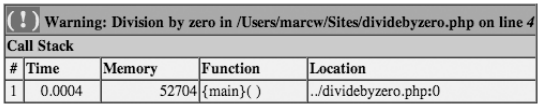
```
ini_set('xdebug.var_display_max_depth', 10);
```

## Better Error Handlers

The default error reporting in PHP tends to be quite sterile and not terribly helpful beyond the basic information, such as a basic error message along with file and line number information.

Xdebug replaces this with a much friendlier error handler that prints out robust HTML messages (when running for the browser) and also prints out a call stack, showing which functions were called before the current error message was generated (see Figure 18-1).

You can turn this off by calling `xdebug_disable` at the top of your scripts.



Warning: Division by zero in /Users/marcw/Sites/dividebyzero.php on line 4

Call Stack

| # | Time | Memory | Function | Location |
|---|---|---|---|---|
| 1 | 0.0004 | 52704 | {main}( ) | ../dividebyzero.php:0 |

Figure 18-1 Xdebug printing out error message information.

## Other Functionality

This extension contains a number of other useful features such as printing out a list of all functions called in your scripts, helping to analyze script execution times, and even functioning as a remote debugging program for your scripts. Spending a few hours reviewing the online documentation for it and playing around with these features is entirely a worthwhile use of time.