

For educational or internal use only

Luka Bostick 2023

Contents of The Lectures

Chapter 1, Introduction to PHP

It talks about the history of PHP and gives a lightning-fast overview of what is possible with PHP programs.

Chapter 2, Language Basics

It concisely guides PHP program elements such as identifiers, data types, operators, and flow-control statements.

Chapter 3, Functions

Discusses user-defined functions, including scope, variable-length parameter lists, and variable and anonymous functions.

Chapter 4, Strings

It covers your PHP code's functions when building, dissecting, searching, and modifying strings.

Chapter 5, Arrays

It covers your PHP code's functions when building, dissecting, searching, and modifying strings.

Chapter 6, Objects

Covers PHP's updated object-oriented features. In this chapter, you'll learn about classes, objects, inheritance, and introspection.

Chapter 7, Dates and Times

Discusses date and time manipulations like time zones and data math.

Chapter 8, Web Techniques

It talks about techniques most PHP programmers eventually want to use, including processing web form data, maintaining state, and dealing with SSL.

For educational or internal use only

Luka Bostick 2023

Chapter 9, Databases

Discusses PHP's modules and functions for working with databases, using MySQL database as examples. Also, SQLite and PDO database interfaces are covered. NoSQL concepts are also covered here.

Chapter 10, Graphics

Demonstrates how to create and modify image files in various formats within PHP.

Chapter 11, PDF

Explains how to create dynamic PDF files from a PHP application.

Chapter 12, XML

Introduces PHP's extensions for generating and parsing XML data.

Chapter 13, JSON

It covers JavaScript Object Notation (JSON), a standardized data-interchange format that is extremely lightweight and human-readable.

Chapter 14, Security

Provides valuable advice and guidance for programmers creating secure scripts. You'll learn programming best practices to help you avoid mistakes that can lead to disaster.

Chapter 15, Application Techniques

It discusses coding techniques like implementing code libraries, dealing with output in unique ways, and error handling.

Chapter 16, Web Services

Describes techniques for dealing with external communication via REST tools and cloud connections.

Chapter 17, Debugging PHP

Discusses techniques for debugging PHP code and for writing debuggable PHP code.

Chapter 18, PHP on Disparate Platforms

Discusses the tricks and traps of the Windows port of PHP. It also discusses some of the features unique to Windows, such as COM.

CHAPTER 1 – Introduction to PHP

PHP is a simple yet powerful language designed for creating HTML content. This chapter covers essential background on the PHP language. It describes the nature and history of PHP, which platforms it runs on, and how to configure it. This chapter ends by showing you PHP in action, with a quick walkthrough of several PHP programs that illustrate common tasks, such as processing form data, interacting with a database, and creating graphics.

What Does PHP DO?

PHP can be used in two primary ways:

Server-side scripting

PHP was originally designed to create dynamic web content and is still best suited for that task. To generate HTML, you need a PHP parser and a web server to send the coded document files. PHP has also become popular for generating dynamic content via database connections, XML documents, graphics, PDF files, and more.

Command-line scripting

PHP can run scripts from the command line like Perl, awk, or the Unix shell. You might use the command-line scripts for system administration tasks, such as backup and log parsing; even some SRON job-type scripts can be done this way (As nonvisual PHP tasks).

In this document, however, we concentrate on the first item: using PHP to develop dynamic web content.

PHP runs on all major operating systems, from Unix variants (including Linux, FreeBSD, Ubuntu, Debian, and Solaris) to Windows and macOS. It can be used with all leading web servers, including Apache, Nginx, and OpenBSD servers, to name a few, even cloud environments like Azure and Amazon.

The language itself is extremely flexible. For example, you aren't limited to outputting just HTML or other text files- any document format can be generated. PHP has built-in support for generating PDF files, GIF, JPEGs, and PNG images.

One of PHP's most significant features is its wide-ranging support for databases. PGP supports all major databases (including MySQL, PostgreSQL, Oracle, Sybase, MS-SQL, DB2, and ODBC-compliant databases),

For educational or internal use only
Luka Bostick 2023

and even many obscure ones. Even the more recent NoSQL-style databases like CouchDB and MongoDB are also supported. Creating web pages with dynamic content from a database with PHP is remarkably simple.

Finally, PHP provides a library of PHP code to perform common tasks, such as database abstraction, error handling, and so on, with the PHP Extension and Application Repository (PEAR). PEAR is a framework and distribution system for reusable PHP components.

A Brief History of PHP

Rasmus Lerdorf first conceived of PHP in 1994, but the PHP that people use today differs from the initial version. To understand how PHP got where it is now, it is useful to know the historical evolution of the language. Here's that story, with ample comments and emails from Rasmus himself.

The Evolution of PHP

Here is the PHP 1.0 announcement that was posted to the Usenet newsgroup (*comp.infosystems.www.authoring.cgi*) in June 1995:

From: rasmus@io.org (Rasmus Lerdorf)
Subject: Announce: Personal Home Page Tools (PHP Tools)
Date: 1995/06/08
Message-ID: <3r7pgp\$aa1@ionews.io.org>#1/1
organization: none
newsgroups: comp.infosystems.www.authoring.cgi
Announcing the Personal Home Page Tools (PHP Tools) version 1.0.
These tools are a set of small tight cgi binaries written in C.
They perform a number of functions including:

- . Logging accesses to your pages in your own private log files
- . Real-time viewing of log information
- . Providing a nice interface to this log information
- . Displaying last access information right on your pages
- . Full daily and total access counters . Banning access to users based on their domain
- . Password protecting pages based on users' domains
- . Tracking accesses ** based on users' e-mail addresses **
- . Tracking referring URL's - HTTP_REFERER support
- . Performing server-side includes without needing server support for it
- . Ability to not log accesses from certain domains (ie. your own)
- . Easily create and display forms
- . Ability to use form information in following documents

Here is what you don't need to use these tools:

For educational or internal use only

Luka Bostick 2023

- . You do not need root access - install in your ~/public_html dir
- . You do not need server-side includes enabled in your server
- . You do not need access to Perl or Tcl or any other script interpreter
- . You do not need access to the httpd log files

The only requirement for these tools to work is that you have the ability to execute your own cgi programs. Ask your system administrator if you are not sure what this means.

The tools also allow you to implement a guestbook or any other form that needs to write information and display it to users later in about 2 minutes.

The tools are in the public domain distributed under the GNU Public License. Yes, that means they are free!

For a complete demonstration of these tools, point your browser at: <http://www.io.org/~rasmus>

--

Rasmus Lerdorf

rasmus@io.org

<http://www.io.org/~rasmus>

Note that this message's URL and email address are long gone. The language of this announcement reflects people's concerns at the time, such as password-protecting pages, easily creating forms, and accessing forms data on subsequent pages. The announcement illustrates PHP's initial positioning as a framework for several use cases.

The announcement talks only about the tools that came with PHP, be behind the scenes, the goal was to create a framework to make it easy to extend PHP and add more tools. The business logic for these additions was written in C, a simple parser picked tags out of the HTML and called the various C functions. It was never really part of the plan to create a scripting language.

So what happened?

Rasmus started working on a rather large project for the University of Toronto that needed a tool to gather data from various places and present a nice web-based administration interface. Of course, he used PHP for that task, but for performance reasons, the various small tools of PHP 1.0 had to be brought together better and integrated into the web server.

For educational or internal use only
Luka Bostick 2023

Initially, some hacks to the NCSA web server were made to patch it to support the core PHP functionality. The problem with this approach was that you had to replace your web server software with this special, hacked-up version as a user. Fortunately, Apache was also gaining momentum around this time, and the Apache API made adding functionality like PHP to the server easier.

Over the next year or so, a lot was done, and the focus changed quite a bit. Here's the PHP 2.0 (PHP/FI) announcement that was sent out in April 1996:

From: rasmus@madhaus.utcs.utoronto.ca (Rasmus Lerdorf)
Subject: ANNOUNCE: PHP/FI Server-side HTML-Embedded Scripting Language
Date: 1996/04/16
Newsgroups: comp.infosystems.www.authoring.cgi

PHP/FI is a server-side HTML embedded scripting language. It has built-in access logging and access restriction features and also support for embedded SQL queries to mSQL and/or Postgres95 backend databases. It is most likely the fastest and simplest tool available for creating database-enabled web sites.

It will work with any UNIX-based web server on every UNIX flavour out there. The package is completely free of charge for all uses including commercial.

Feature List:

. Access Logging

Log every hit to your pages in either a dbm or an mSQL database.
Having hit information in a database format makes later analysis easier.

. Access Restriction

Password protect your pages, or restrict access based on the refering URL plus many other options.

. mSQL Support

Embed mSQL queries right in your HTML source files

. Postgres95 Support

Embed Postgres95 queries right in your HTML source files

. DBM Support

DB, DBM, NDBM and GDBM are all supported

. RFC-1867 File Upload Support

Create file upload forms

. Variables, Arrays, Associative Arrays

. User-Defined Functions with static variables + recursion

. Conditionals and While loops

Writing conditional dynamic web pages could not be easier than with the PHP/FI conditionals and looping support

. Extended Regular Expressions

Powerful string manipulation support through full regexp support

. Raw HTTP Header Control

Lets you send customized HTTP headers to the browser for advanced features such as cookies.

. Dynamic GIF Image Creation

Thomas Boutell's GD library is supported through an easy-to-use set of tags.

It can be downloaded from the File Archive at: <URL:http://www.vex.net/php>

--

Rasmus Lerdorf
rasmus@vex.net

This was the first time the term scripting language was used. PHP 1.0's simplistic tag-replacement code was replaced with a parser to handle a more sophisticated embedded tag language. By today's standards, the tag language wasn't particularly sophisticated, but compared to PHP 1.0, it certainly was.

The main reason for this change was that only some people who used PHP 1.0 were actually interested in using the C-based framework for creating add-ons. Most users were much more interested in being able to embed logic directly in their web pages for creating conditional HTML, custom tags, and other such features. PHP 1.0 users constantly requested the ability to add the hit-tracking footer or send different HTML blocks conditionally. This led to the creation of an if tag. Once you have if, you need else as well, and from there, it's a slippery slope to the point where you write an entire scripting language, whether you want to or not.

By mid-1997, PHP version 2.0 had grown quite a bit and had attracted a lot of users, but there were still some stability problems with the underlying parsing engine. The project was also still mostly a one-man effort, with a few contributions here and there. At this point, Zeev Suraski and Andi Gutmans in Tel Aviv, Israel, volunteered to rewrite the underlying parsing engine, and we agreed to make their rewrite the base for PHP version 3.0. Other people also volunteered to work on other parts of PHP, and the project changed from a one-person effort with a few contributors to a true open-source project with many developers worldwide.

Here is the PHP 3.0 announcement from June 1998:

June 6, 1998 -- The PHP Development Team announced the release of PHP 3.0, the latest release of the server-side scripting solution already in use on over 70,000 World Wide Web sites.

This all-new version of the popular scripting language includes support for all major operating systems (Windows 95/NT, most versions of Unix, and Macintosh) and web servers (including Apache, Netscape servers, WebSite Pro, and Microsoft Internet Information Server).

PHP 3.0 also supports a wide range of databases, including Oracle,

Sybase, Solid, MySQL, mSQL, and PostgreSQL, as well as ODBC data sources.

New features include persistent database connections, support for the SNMP and IMAP protocols, and a revamped C API for extending the language with new features.

"PHP is a very programmer-friendly scripting language suitable for people with little or no programming experience as well as the seasoned web developer who needs to get things done quickly. The best thing about PHP is that you get results quickly," said Rasmus Lerdorf, one of the developers of the language.

"Version 3 provides a much more powerful, reliable, and efficient implementation of the language, while maintaining the ease of use and rapid development that were the key to PHP's success in the past," added Andi Gutmans, one of the implementors of the new language core.

"At Circle Net we have found PHP to be the most robust platform for rapid web-based application development available today," said Troy Cobb, Chief Technology Officer at Circle Net, Inc. "Our use of PHP has cut our development time in half, and more than doubled our client

For educational or internal use only

Luka Bostick 2023

satisfaction. PHP has enabled us to provide database-driven dynamic solutions which perform at phenomenal speeds."

PHP 3.0 is available for free download in source form and binaries for several platforms at <http://www.php.net/>.

The PHP Development Team is an international group of programmers who lead the open development of PHP and related projects.

For more information, the PHP Development Team can be contacted at core@php.net.

After the release of PHP 3.0, usage really started to take off. Version 4.0 was prompted by a number of developers who were interested in making some fundamental changes to the architecture of PHP. These changes included abstracting the layer between the language and the web server, adding a thread-safety mechanism, and adding a more advance two-stage parse/execute tag-parsing system. This new parser, primary written by Zeev and Andi, was named the Zend engine. After a lot of work by a lot of developers, PHP 4.0 was released on May 22,2000.

As this document is transcribed, PHP version 7.3 has been released for some time. There have already been a few minor "dot" releases, and the stability of this current version is quite high. As you will see in this document, there have been some major advances made in this version of PHP, primarily in code processing on the server side. Many other minor changes, function additions, and feature enhancements have also been incorporated.

The Widespread Use Of PHP

Figure 1-1 shows the usage of PHP as compiled by W3 Techs as of July 2023. The most interesting piece of data here is that 77% of all the surveyed websites use it, with version 7 being the most widely used. If you look at the methodology used in the W3Techs surveys, you will see that they select the world's top 10 million sites (based on traffic; website popularity). As it evident, PHP has a very broad adoption indeed.

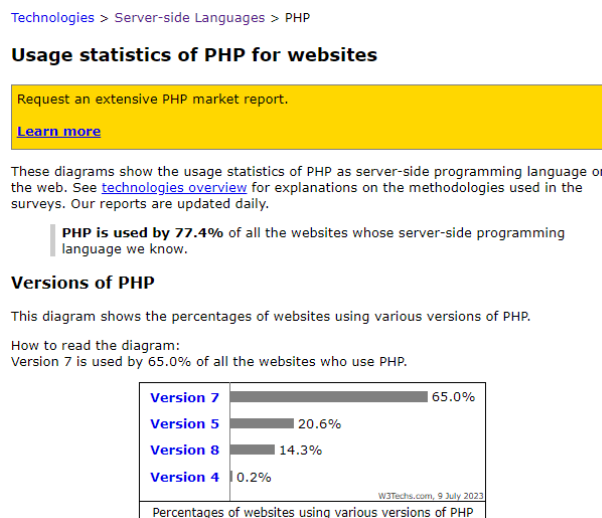


Figure 1-1 PHP usage as of July 2023

Installing PHP

As mentioned, PHP is available for many operating systems and platforms. Therefore, you are encouraged to consult the PHP documentation to find the environment that most closely fits the one you will be using and follow the appropriate setup instructions.

From time to time, you may also want to change the way PHP is configured. To do that, you will have to change the PHP configuration file and restart your web (Apache) server in order for those changes to take effect.

PHP's configuration settings are usually maintained in a file called `php.ini`. The settings in this file control the behavior of PHP features, such as session handling and form processing. Later chapters refer to some of the `php.ini` options, but generally, the code in this Document does not require a customized configuration. See the PHP documentation for more information on configuring `php.ini`.

A Walk Through of PHP

PHP pages are generally HTML pages with PHP commands embedded in them. This is in contrast to many other dynamic web page solutions, which are scripts that generate HTML. The web server processes the PHP commands and sends their output (and any HTML from the file) to the browser. Example 1-1 shows a complete PHP page

```
<html>
  <head>
    <title> Look Out World</title>
  </head>

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

Example 1-1. Hello_world.php

Save the contents of Example 1-1 to a file, `hello_world.php`, and point your browser to it. The results appear in Figure 1-2.

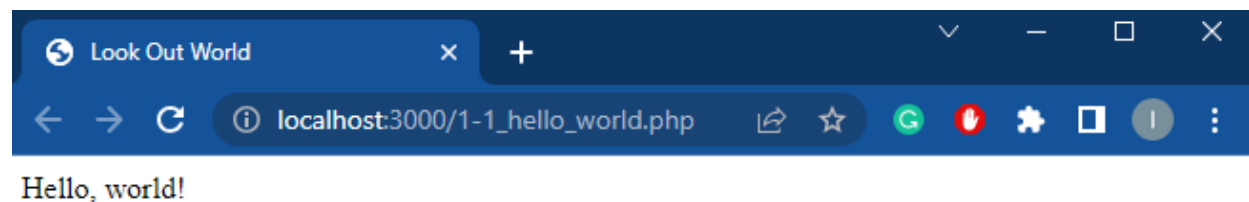


Figure 1-2. Output of `hello_world`

For educational or internal use only
Luka Bostick 2023

The PHP echo command produces output (the string “Hello, world!” in this case) inserted into the HTML file. In this example the PHP code is placed between the `<?php` and `?>` tags. There are other ways to tag your PHP code – see Chapter 2 for a full description.

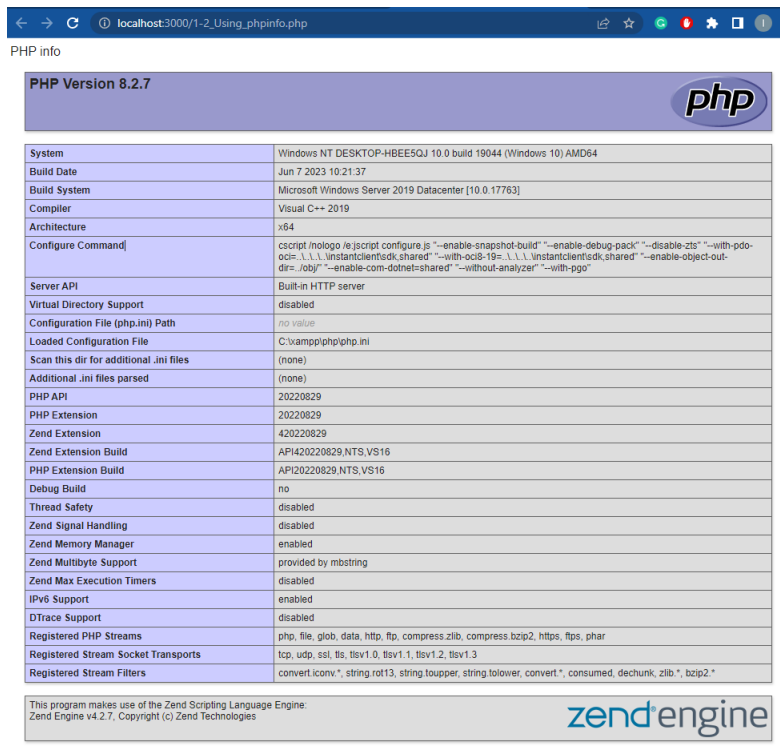
Configuration Page

The PHP function `phpinfo()` creates an HTML page full of information on how PHP was installed and is currently configured. You can use it to see whether you have a particular extensions installed, or whether the `php.ini` file has been customized. Example 1-2 is a complete page that displayed the `phpinfo()` page.

```
<html>
<head>PHP info</head>

<body>
    <?php phpinfo();?>
</body>
</html>
```

Example 1-2. Using `phpinfo()`



PHP Version 8.2.7	
System	Windows NT DESKTOP-HBEE5QJ 10.0 build 19044 (Windows 10) AMD64
Build Date	Jun 7 2023 10:21:37
Build System	Microsoft Windows Server 2019 Datacenter [10.0.17763]
Compiler	Visual C++ 2019
Architecture	x64
Configure Command	csconfig /nologo /e:jsconfig.js "--enable-snapshot-build" "--enable-debug-pack" "--disable-zts" "--with-pdo-oci=\\.\\.\\instantclient\ sdk,shared" "--with-oci8-19=\\.\\.\\instantclient\ sdk,shared" "--enable-object-out-dir=\\.obj" "--enable-com-dotnet=shared" "--without-analyzer" "--with-pgo"
Server API	Built-in HTTP server
Virtual Directory Support	disabled
Configuration File (php.ini) Path	no value
Loaded Configuration File	C:\xampp\php\php.ini
Scan this dir for additional .ini files	(none)
Additional .ini files parsed	(none)
PHP API	20220829
PHP Extension	20220829
Zend Extension	420220829
Zend Extension Build	API420220829,NTS,VS16
PHP Extension Build	API20220829,NTS,VS16
Debug Build	no
Thread Safety	disabled
Zend Signal Handling	disabled
Zend Memory Manager	enabled
Zend Multibyte Support	provided by mbstring
Zend Max Execution Timers	disabled
IPv6 Support	enabled
DTrace Support	disabled
Registered PHP Streams	php, file, glob, data, http, ftp, compress.zlib, compress.bzip2, https, ftps, phar
Registered Stream Socket Transports	tcp, udp, ssl, tls, tlsv1.0, tlsv1.1, tlsv1.2, tlsv1.3
Registered Stream Filters	convert.iconv*, string.rot13, string.toupper, string.tolower, convert.*, consumed, dechunk, zlib.*, bzip2.*

This program makes use of the Zend Scripting Language Engine:
Zend Engine v4.2.7. Copyright (c) Zend Technologies

Figure 1-3. Partial output of `phpinfo()`

Forms

Example 1-3 creates and processes a form. When the user submits the form, the information typed into the name field is sent back to this page via the `$_SERVER['PHP_SELF']` form action. The PHP code tests for a name field and displays a greeting if it finds one.

```
<html>
  <head>
    <title>Personalized Greeting Form</title>
  </head>

  <body>
    <?php if(!empty($_POST['name'])) {
      echo "Greetings, {$_POST['name']}, and welcome.";
    } ?>

    <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
      Enter your name: <input type="text" name="name" />
      <input type="submit" />
    </form>
  </body>
</html>
```

The form and the message are shown in Figure 1-4.

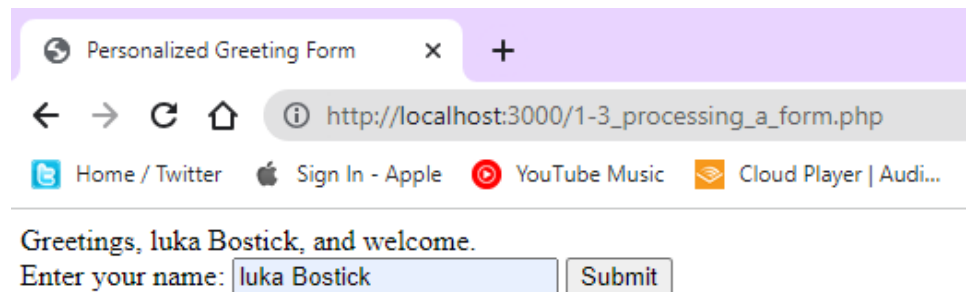


Figure 1-4. Form and greeting page

PHP programs access form values primarily through the `$_POST` and `$_GET` array variables. Chapter 8 discusses form and form processing in more detail.

Databases

PHP supports all the popular database systems, including MySQL, PostgreSQL, Oracle, Sybase, SQLite, and ODBC-compliant databases. Figure 1-5 shows part of a MySQL database query run through a PHP script, displaying the results of a book search on a book review site. It lists the book title, the year the book was published, and the book's ISBN.

Figure 1-5. A MySQL book list query run through a PHP script

These Books are currently available		
Title	Year Published	ISBN
Executive Orders	1996	0-425-15863-2
Forward the Foundation	1993	0-553-56507-9
Foundation	1951	0-553-80371-9
Foundation and Empire	1952	0-553-29337-0
Foundation's Edge	1982	0-553-29338-9
I, Robot	1950	0-553-29438-5
Isaac Asimov: Gold	1995	0-06-055652-8
Rainbow Six	1998	0-425-17034-9
Roots	1974	0-440-17464-3
Second Foundation	1953	0-553-29336-2
Teeth of the Tiger	2003	0-399-15079-X
The Best of Isaac Asimov	1973	0-449-20829-X
The Hobbit	1937	0-261-10221-4
The Return of The King	1955	0-261-10237-0
The Sum of All Fears	1991	0-425-13354-0
The Two Towers	1954	0-261-10236-2

The code in Example 1-4 connects to the data base, uses a query to retrieve all available books with the WHERE clause), and produces a table as output for all returned results through a while loop. The SQL code for this sample database is in the provided file library. SQL. You can drop this code into MySQL after you create the library database and have the sample database at your disposal for testing out the following code sample as well as the related samples in Chapter 9.

```
<?php

$db = new mysqli("localhost", "peterwin", "password", "library");
// fictitious database connect function

// Check connection
if ($db->connect_errno) {
    die("Connect Error (" . $db->connect_errno . ") " . $db->connect_error);
}

$sql = "SELECT * FROM books WHERE available = 1 ORDER BY title";
$result = $db->query($sql);
?>

<!DOCTYPE html>
<html>
<head>
    <style>
        table {
            margin-left: auto;
            margin-right: auto;
            border-collapse: collapse;

            th, td {
                padding: 6px;
                border: 1px solid black;

                th {
                    text-align: center;
                }
            }
        }
    </style>
</head>
<body>
<table>
    <tr>
        <td colspan="3">
            <h3 align="center">These Books are currently available</h3>
        </td>
    </tr>
```

```
<tr>
  <th>Title</th>
  <th>Year Published</th>
  <th>ISBN</th>
</tr>

<?php while ($row = $result->fetch_assoc()) { ?>
  <tr>
    <td><?php echo htmlspecialchars($row['title']); ?></td>
    <td align="center"><?php echo
htmlspecialchars($row['pub_year']); ?></td>
    <td><?php echo htmlspecialchars($row['ISBN']); ?></td>
  </tr>
<?php } ?>
</table>
</body>
</html>
```

Database-provided dynamic content that drives the news, blogs, and e-commerce sites at the heart of the web. More details on accessing databases from PHP are given in Chapter 9. (Instructors note: if you receive the following error -Uncaught Error: Call to undefined function imagecreatefrompng()

Try the followingThe error you're encountering is caused by the GD library not being enabled in your PHP configuration. The GD library is required for image processing functions like imagecreatefrompng() and imagettftext(). To resolve this issue, you need to enable the GD library.

Here are the steps to enable the GD library:

Locate the PHP configuration file (php.ini). The location of this file can vary depending on your operating system and PHP installation.

Open the php.ini file in a text editor. Search for the following line: ;extension=gd

Remove the semicolon at the beginning of the line to uncomment it: extension=gd Save the php.ini file. Restart your web server to apply the changes.)

Graphics

With PHP, you can easily create and manipulate images using the GD extension. Example 1-5 provides a text entry field that lets the user specify the text for a button. It takes an empty button image file, and centers the text passed as the GET parameter 'message' on it. The result is sent back to the browser as a PNG image.

The form generated by example 1-5 is shown in Figure 1-6. The button created is shown in Figure 1-7.

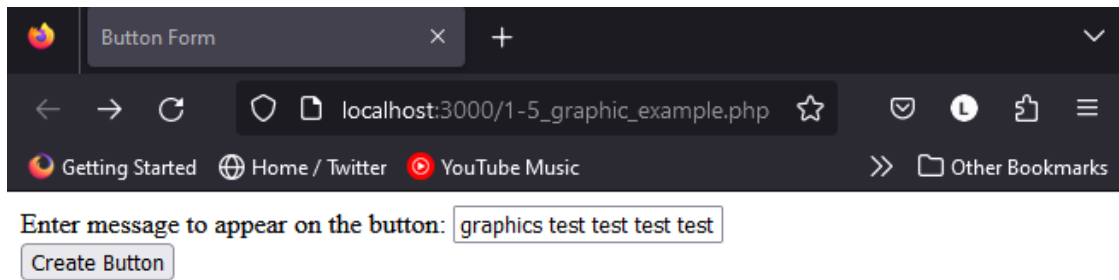


Figure 1-6. Button creation form

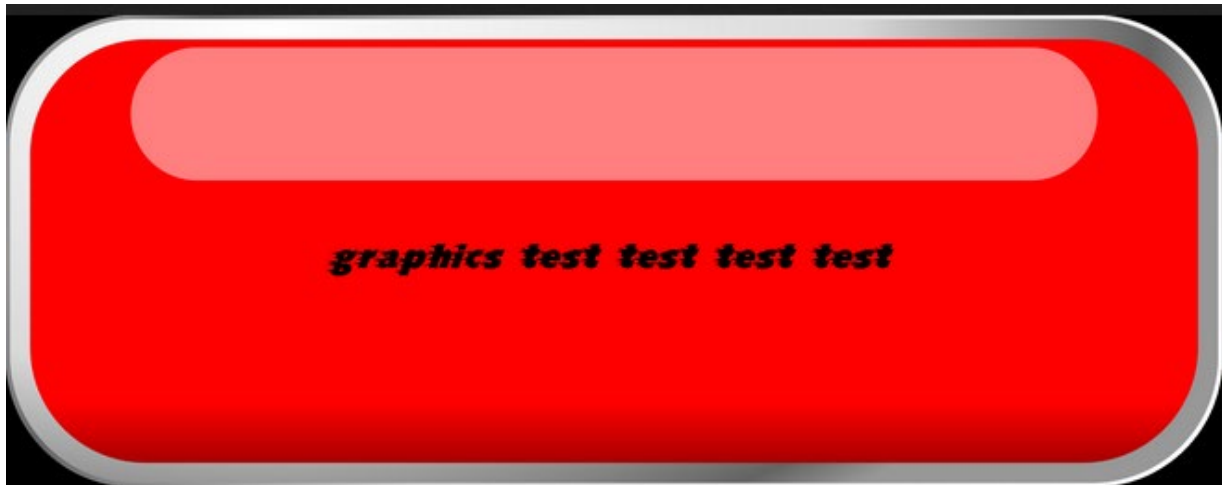


Figure 1-7. Button created

You can use GD to resize images dynamically, produce graphs, and much more. PHP also has several extensions to generate documents in Adobe's popular PDF format. Chapter 10 covers dynamic image generation in depth, while Chapter 11 provides instructions on creating Adobe PDF files.

What's Next

Now that you've tasted what is possible with PHP, you are ready to learn how to program in the language. We start with its basic structure, focusing on user-defined functions, string manipulation, and object-oriented programming. Then we move to specific application areas, such as the web, databases, graphics, XML, and security. We finish with quick references to the built-in functions and extensions. Master these chapters, and you will have mastered PHP!

CHAPTER 2 – Language Basics

This chapter provides a whirlwind tour of the core PHP language, covering such basic topics as data types, variables, operators, and flow-control statements. PHP is influenced by other programming languages such as Perl and C, so if you've had experience with those languages, PHP should be easy to

pick up. If PHP is one of your first programming languages, don't panic. We start with the basic units of a PHP program and build up your knowledge from there.

Lexical Structure

The lexical structure of a programming language is the set of basic rules that governs how you write programs in that language. It is the language's lowest-level syntax and specifies what variables look like, what characters are used for comments, and how program statements are separated from each other.

Case Sensitivity

The names of user-defined classes and functions, as well as build-in constructs and keywords (such as echo, while, class, etc), are case-insensitive. Thus, these three lines are equivalent:

```
<html>
  <head>
    <title> Look Out World</title>
  </head>

  <body>
    <?php echo "Hello, world!";
        ECHO "Hello, world!";
        EcHo "Hello, world!";?>
  </body>
</html>
```

Variables, on the other hand, are case-sensitive. That is \$name, \$NAME, and \$NaME are three different variables.

Statements and Semicolons

A statement is a collection of PHP code that does something. It can be as simple as a variable assignment or as complicated as a loop with multiple exit points. Here is a small sample of PHP statements, including function calls, some variable data assignments, and an if statement

```
<body>
  <?php echo "Hello, world";
  myFunction(42,"Web-Design-Tools");
  $a = 1;
  $name = "Elephant";
  $b=$a/25.0;
  if($a==$b)
  {
    echo "Rhyme? And Reason?";
  }?>
</body>
```


PHP uses semicolons to separate simple statements. A compound statement that uses curly braces to mark a block of code, such as a conditional test or loop, does not need a semicolon after a closing brace. Unlike in other languages, in PHP, the semicolon before the closing brace is not optional:

```
<?php if($needed)
{
    echo "We must have it!";// semicolon required here

    }// no semicolon required here after the brace
?>
```

The semicolon, however, is optional before a closing PHP tag:

```
<?php if($a==$b)
{
    echo "Rhyme? And Reason?";
}
echo"Hello, world"// no semicolon required before closing tag
?>
```

It's good programming practice to include optional semicolons, as they make it easier to add code later.

Whitespace and Line Breaks

In general, whitespace doesn't matter in a PHP program. You can spread a statement across any number of lines, or lump a bunch of statements together on a single line. For example, this statement;

```
<?php raisePrices($inventory, $inflation, $costOfLiving, $greed);?>
```

Could just as well be written with more whitespace:

```
<?php raisePrices(
    $inventory,
    $inflation,
    $costOfLiving,
    $greed);?>
```

Or with less whitespace:

```
<?php raisePrices($inventory,$inflation,$costOfLiving,$greed);?>
```

You can use this flexible formatting to make your code more readable (by lining up assignments, indenting, etc.) Some lazy programmers use this freeform formatting and create unreadable code. For example, you could say theatrically. Give each character in your code a space making it impossible to read.

Comments

Comments give information to people who read your code, but PHP ignores them at execution time. Even if you think you're the only person who will ever read your code, it's a good idea to include comments in your code- in retrospect, code you wrote months ago could easily look like a stranger wrote it.

A goof practice is to make your comments sparse enough not to get in the way of the code itself but plentiful enough that you can use the comments to tell what's happening. Don't comment on obvious things, lest you bury the comments that describe tricky things. For example this is worthless:

```
<?php $x = 17; // store 17 into the variable $x?>
```

Whereas the comments on this complex regular expression will help whoever maintains your code:

```
<?php $text =preg_replace('/&#(0-9)+;/','chr(\'\\1\')');// convert &#nnn;  
entities into characters?>
```

The purpose of this code is to convert HTML numeric character references (e.g., **A** representing the character 'A') into their corresponding characters. For example, if the input text contains **A**, the regular expression will match it, and **chr(65)** will convert it to the character 'A'.

PHP provides several ways to include comments within your code, all of which are borrowed from existing languages such as C, C++, and the Unix shell. In general, use C-style comments to comment out code, and C++ style comments to comment on code.

Shell-style comments

When PHP encounters a hash mark character (#) within the code, everything from the has mark to the end of the line or end of the section of PHP code (whichever comes first) is considered a comment. This method commenting is found in Unix shell scripting languages and is useful for annotating single lines of code or making short notes.

Because the hash mark is visible on the page, shell-style comments are sometimes used to mark off blocks of code:

```
<?php #####  
    ## Cookie functions  
    #####?>
```

Sometimes they're used before a line of code to identify what that code does, in which case they're usually indented to the same level as the code for which the comment is intended:

```
<?php if($doubleCheck){  
    # create an HTML form requesting that the user confirm the  
    action  
    echo confirmationForm();  
}?>
```

Short comments on a single line of code are often put on the same line as the code:

```
<?php $value = $p * exp($r * $t); # calculate compounded interest?>
```

When you're tightly mixing HTML and PHP code, it can be useful to have the closing PHP tag terminate the comment:

```
<?php $d=4; #Set $d to 4.?> then another <?php echo $d; ?>
```

C++ comments

When PHP encounters two slashes (//) within the code,, everything from the slashes to the end of the line or the end of the code section, whichever comes first., is considered a comment. This method of commenting is derived from C++. The result is the same as the shell comment style.

Here are the shell-style comment example, rewritten to use C++ comments:

```
<?php      ////////////////////////////////////////////////////
            // Cookie functions
            ////////////////////////////////////////////////////
            if($doubleCheck)
            {
                // create an HTML form requesting that the user confirm
//the action
                echo confirmationForm();
            }
            $value = $p * exp($r * $t); // calculate compounded interest
            ?>
```

C comments

While shell-style and C++-style comments are useful for annotating code or making short notes, longer comments require a different style. Therefore, PHP supports block comments whose syntax comes for the C programming language. When PHP encounters a slash followed by an asterisk (/*), everything after that until it encounters an asterisk followed by a slash (*), is considered a comment. Unlike those shown earlier, this kind of comment can span multiple lines.

Here's an example of a C-style multiline comment:

```
<?php /* In this section, we take a bunch of variables and
        assign numbers to them. There is no real reason to
        do this, we're just having fun.
        */
        $a = 1;
        $b = 2;
        $c = 3;
        $d = 4;
        ?>
```

For educational or internal use only
Luka Bostick 2023

Because C-style comments have specific start and end markers, you can tightly integrate them with code. This tends to make your code harder to read and is discouraged:

```
<?php /* These comments can be mixed with code too,  
        see? */ $e = 5; /* This works just fine. */  
?>
```

C-style comments, unlike the other types, can continue past the end PHP tag markers. For example:

```
<?php  
$l = 12;  
$m = 13;  
/* A comment begins here  
?>  
<p>Some stuff you want to be HTML.</p>  
<?= $n = 14; ?>  
*/  
echo("l=$l m=$m n=$n\n");  
?><p>Now <b>this</b> is regular HTML...</p>  
l=12 m=13 n=  
<p>Now <b>this</b> is regular HTML...</p>  
?>
```

You can indent comments as you like:

```
/* There are no  
special indenting or spacing  
rules that have to be followed, either.  
  
*/
```

C-style comments can be useful for disabling sections of code. In the following example, we disabled the second and third statements and the inline comments by including them in a block comment. To enable the code, all we must do is remove the comment markers:

```
<?php  
    $f=6;  
    /**  
     * $g = 6; # This is a different style of comment  
     * $g = 8;  
     */  
?>
```

However, you must be careful not to attempt not to nest block comments:

```
<?php
    $i=9;
    /*
    $j=10; /* This is a comment*/
    $k=11;
    Here is some comment text.
    */
?>
```

In this case, PHP tries (and fails) to execute the (non) statement Here is some comment text and returns an error.

Literals

A literal is a data value that appears directly in a program. The following are all literals in PHP:

```
<?php
2001
0xFE
2.4142
"Hello World"
'Hi'
true
null
?>
```

Identifiers

An identifier is simply a name. In PHP, identifiers name variables, functions, constants, and classes. The first character of an identifier must be an ASCII letter (uppercase or lowercase), the underscore character (_), or any other characters between ASCII 0x7F and ASCII 0xFF. After the initial character, these characters and the digits 0-9 are valid.

Variable names

Variable names always begin with a dollar sign (\$) and are case-sensitive. Here are some valid variable names:

```
<?php
$bill
$head_count
$MaximumForce
$I_HEART_PHP
$_underscore
$_int
?>
```

For educational or internal use only

Luka Bostick 2023

Here are some illegal variable names:

```
<?php
$not valid
$|
$3wa
?>
```

These variables are all different due to case sensitivity:

```
<?php
$hot_stuff
$Hot_stuff
$hot_Stuff
$HOT_STUFF
?>
```

Function names

Function names are not case-sensitive (functions are discussed in more detail in Chapter 3). Here are some valid function names:

```
<?php
tally
list_all_users
deleteTclFiles
LOWERCASE_IS_FOR_WIMPS
_hide
?>
```

These function names all refer to the same function:

```
<?php
howdy
HoWdY
HOWDY
HOWdy
howdy
?>
```

Class names

Class names follow the standard rules for PHP identifiers and are also not casesensitive. Here are some valid class names:

```
<?php
Person
account
?>
```

Note: The class name stdClass is a reserved class name.

Constants

A constant is an identifier for a value that will not be changed; scalar values (Boolean, integer, double, and string) and arrays can be constants. Once set, the value of a constant cannot change. Constants are referred to by their identifiers and are set using the define() function:

```
<?php
define('PUBLISHER', 'NOT ME I STOLE THIS');
echo PUBLISHER;
?>
```

Keywords

A keyword (or reserved word) is a word set aside by the language for its core functionality-you cannot give a function, class, or constant the same names as a keyword. Table 2-1 lists the keywords in PHP which are case-insensitive.

Table 2-1. PHP core language keywords

__CLASS__	echo	insteadof
__DIR__	else	interface
__FILE__	elseif	isset()
__FUNCTION__	empty()	list()
__LINE__	enddeclare	namespace
__METHOD__	endfor	new
__NAMESPACE__	endforeach	or
__TRAIT__	endif	print
__halt_compiler()	endswitch	private
abstract	endwhile	protected
and	eval()	public
array()	exit()	require
as	extends	require_once
break	final	return
callable	finally	static
case	for	switch
catch	foreach	throw
class	function	trait
clone	global	try
const	goto	unset()
continue	if	use
declare	implements	var
default	include	while
die()	include_once	xor
do	instanceof	yield
		yield from

In addition, you cannot use an identifier that is the same as a built-in PHP function. For a complete list of these, see the Appendix.

Data Types

PHP provides eight types of values or data types. Four are scalar (single-value) types:

Integers, floating-point numbers, strings, and Booleans. Two are compound (collection) types: arrays and objects. The remaining two are special types: resource and NULL. Numbers, Booleans, resources, and NULL, are discussed in full here, while strings, arrays, and objects are big enough topics to get their chapters (Chapters 4.5 and 6, respectively).

Integers

Integers are whole numbers, such as 1,12, and 256. The range of acceptable values varies according to the details of your platform but typically extends from -2,127,483,648 to +2,127,483,647—specifically, the range of the long data type of your C compiler. Unfortunately, the C standard doesn't specify what long type range should have, so on some systems; you might see a different integer range. Integer literals can be written in decimal, octal, binary, or hexadecimal. A sequence of digits represents decimal values without leading zeros. The sequence may begin with a plus (+) or a minus(-). If there is no sign, a positive is assumed. Examples of decimal integers include the following:

```
<?php
1998
-641
+33
?>
```

Octal numbers consist of a leading 0 and a sequence of digits from 0 to 7. Like decimal numbers, octal numbers can be prefixed with a plus or minus. Here are some examples of octal values and their equivalent decimal values:

```
<?php
0755 // decimal 493
+010 // decimal 8
?>
```

Hexadecimal values begin with 0x, followed by a sequence of digits (0-9) or letters (A-F). The letters can be upper- or lowercase but are usually written in capitals. As with decimal and octal value, you can include a sign in hexadecimal numbers:

```
0xFF // decimal 255
0x10 // decimal 16
-0xDAD1 // decimal -56017
```

Binary numbers begin with 0b, followed by a sequence of digits (0 and 1). As with other values, you can include a sign in binary numbers:


```
0b01100000 // decimal 96  
0b00000010 // decimal 2  
-0b10 // decimal -2
```

If you try to store a variable that is too large to be stored as an integer or is not a whole number, it will automatically be turned into a floating-point number.

Use the `is_int()` function (or its `is_integer()` alias) to test whether a value is an integer

```
if(is_int($x))  
{  
    // $s is an integer  
}
```

Floating-Point Numbers

Floating-point number (often referred to as “real” numbers) represent numeric values with decimal digits. Like integer, their limits depend on your machine’s details. PHP floating-point numbers are equivalent to the range of the double data type of your C compiler. Usually, this allows numbers between $1.7E-308$ and $1.7E+308$ with 15 digits of accuracy. You can use the BC or GMP extensions if you need more accuracy or a wider range of integer values.

Php recognizes floating-point numbers written in two different formats. There’s the one we all use every day:

```
3.14  
0.17  
-7.1
```

But PHP also recognizes numbers in scientific notation:

```
0.314E1 // 0.314*10^1, or 3.14  
17.0E-3 // 17.0*10^(-3), or 0.017
```

(Instructors NOTE: Make a demonstration of this concept)

Floating-point values are only approximate representations of numbers. For example, on many systems, 3.5 is represented as 3.499999999. This means you must take care to avoid writing that assumes floating-point numbers are represented completely accurately, such as directly comparing two floating-point values using `==`. The normal approach is to compare to several decimal places:

```
if(inval($a * 1000) == intval($b * 1000))  
{  
    // numbers equal to three decimal places  
}
```

Use the `is_float()` function (or its `is_real()` alias) to test whether a value is a floating-point number:

```
if(is_float($x))
{
    // $x is a floating-point number
}
```

Strings

Because strings are so common in web applications, PHP includes core-level support for creating and manipulating strings. A string is a sequence of characters of arbitrary length. String literals are delimited by either single or double quotes:

```
'big dog'
"fat hog"
```

Variables are expanded (interpolated) within double quotes, while within single quotes they are not:

```
$name = "Guido";
echo"Hi, $name <br/>";
echo'Hi, $name';
Hi, Guido
Hi, $name;
```

Double quotes also support a variety of string escapes, as Listed in Table 2-2.

Table 2-2. Escape sequences in double-quoted strings

Escape sequence	Character represented
\"	Double quotes
\n	Newline
\r	Carriage return
\t	Tab
\\	Backslash
\\$	Dollar sign
\{	Left brace
\}	Right brace
\[Left bracket
\]	Right bracket
\0 through \777	ASCII character represented by octal value
\x0 through \xFF	ASCII character represented by hex value

A single-quoted string recognizes \\ to get a literal backslash and \' to get a literal single quote:

```
$dosPath = 'C:\\WINDOWS\\SYSTEM';
$publisher = 'Tim O\\'Reilly';
echo "$dosPath $publisher";
// OUTPUT C:\WINDOWS\SYSTEM Tim O'Reilly
```

To test whether two strings are equal, use the == (double equals) comparison operator:

```
if($a==$b)
{
    echo"a and b are equal";
}
```

Use the is_string() function to test whether a value is a string:

```
if(is_string($x))
{
    //$x is a string
}
```

PHP provides operators and functions to compare, disassemble, assemble, search, replace, and trim strings, as well as a host of specialized string functions for working with HTTP, HTML, and SQL encodings. Because there are so many string-manipulation functions, we've devoted a whole chapter (Chapter 4) to covering all the details.

Booleans

A Boolean value represents a truth value—it says whether something is true or not. Like most programming languages, PHP defines some value as true and other as false. Truthfulness and falseness determine the outcome of conditional code such as:

```
if ($alive) { ... }
```

In PHP, the following values all evaluate to false:

- The keyword false
- The integer 0
- The floating-point value 0.0
- The empty string ("") and the string "0"
- An array with zero elements
- The NULL value

A value that is not false is true, including all resource values (which are described later in the section "Resources").

PHP provides true and false keywords for clarity:

```
$x = 5;        //$x has a true value
$x = true;     // clearer way to write it
$y="";        // $y has a false value
$y = false;    // clear way to write it
```

Use the is_bool() function to test whether a value is a Boolean:

```
if (is_bool($x)) {
    // $x is a Boolean
}
```

Arrays

An array holds a group of values, which you can identify by position (a number, with zero being the first position) or some identifying name (a string), called an associative index:

```
$person[0]="Edison";  
$person[1]="Wankel";  
$person[2]="Crapper";  
  
$creator['Light bulb']="Edison";  
$creator['Rotary Engine'] = "Wankel";  
$creator['Toilet'] = "Crapper";
```

The array() construct creates an array. Here are two example:

```
$person = array("Edison","Wankel","Crapper");  
  
$creator = array('Light bulb' => "Edison",  
                 'Rotary Engine'=> "Wankel",  
                 'Toilet'=> "Crapper");
```

There are several ways to loop through arrays, but most common is a foreach loop:

```
foreach($person as $name){  
    echo"Hello,{ $name}<br/>";  
}  
  
foreach($creator as $invention => $inventor){  
    echo"{ $inventor} invented the { $invention} <br/>";  
}  
  
/*OUTPUT: Hello, Edison  
Hello, Wankel  
Hello, Crapper  
Edison created the Light bulb  
Wankel created the Rotary Engine  
Crapper created the Toilet  
*/
```

You can sort the elements of an array with the various sort functions:

```
sort($person);  
// $person is now array("Crapper", "Edison", "Wankel")  
  
asort($creator);  
// $creator is now array ('Toilet' => "Crapper",  
// 'Light bulb' => "Edison",  
// 'rotary Engine' /> "Wankel");
```

Use the `is_array()` function to test whether a value is an array:

```
if(is_array($x))
{
    //$x is an array
}
```

There are functions for returning the number of items in the array fetching every value in the array, and much more. Arrays are covered in depth in Chapter 5.

Objects

PHP also supports object-oriented programming (OOP). OOP promotes clean, modular design; simplifies debugging and maintenance and assists with code reuse. Classes are the building blocks of object-oriented design. A class is a definition of a structure that contains properties (variables) and methods (functions). Classes are defined with the `class` keyword:

```
class Person{
    public $name='';

    function name($newname = NULL)
    {
        if(!is_null($newname)){
            $this->name = $newname;
        }
        return $this->name;
    }
}
```

Once a class is defined, any number of objects can be made from it with the `new` keyword, and the object's properties and methods can be accessed with the `->` construct:

```
$ed = new Person;
$ed->name('Edison');
echo "Hello, {$ed->name}<br/>";
$tc = new Person;
$tc->name('Crapper');
echo "Look out below{$tc->name}";

/*
OUTPUT:
Hello, Edison
Look out below Crapper
*/
```

Use the `is_object()` function to test whether a value is an object:

Chapter 6 describes classes and objects in much more detail, including inheritance, encapsulation, and introspection.

Resources

Many modules provide several functions for dealing with the outside world. For example, every database extension has at least a function to connect to the database, a function to close the connection to the database, and a function to query the database. Because you can have multiple database connections open simultaneously, the connect function gives you something to identify that unique connection when you call the query and close functions: a resource (or a handle).

Every active resource has a unique identifier. Each identifier is a numerical index into an internal PHP lookup table containing information about all the active resources. PHP maintains information about each resource in this table, including the number of references to (or uses of) the resource throughout the code. When the last reference to a resource value goes away, the extension that created the resource is called to perform tasks such as freeing any memory or closing any connection for that resource:

```
$res = database_connect();// fictitious database connect function
    database_query($res);
$res = "boo";
// database connection automatically closed because $res is redefined
```

The benefit of this automatic cleanup is best seen within functions when the resource is assigned to a local variable. When the function ends, the variable's value is reclaimed by PHP:

```
function search(){
    $res=database_connect();
    database_query($res);
}
```

When there are no more references to the resource, it's automatically shut down. That said, most extensions provide a specific shutdown or close function, and it's considered good to call that function explicitly when needed rather than relying on variable scoping to trigger resource cleanup.

Use the `is_resource()` function to test whether a value is a resource:

Callbacks

Callbacks are functions or object methods some functions use, such as `call_user_func()`. Callbacks can also be created by the `create_function()` method and through closures (described in Chapter 3):

```
$callback = function()
{
    echo "callback achieved";
};
call_user_func($callback);
```

NULL

There's only one value of the NULL data type. That value is available through the case-insensitive keyword NULL. The NULL value represents a variable that has no value. (similar to Perl's undef or Python's None):

```
$alpha="beta";  
$aleph = null; // variable's value is gone  
$alpha=NULL; // same  
$aleph = NULL; // same
```

Use the `is_null()` function to test whether a value is NULL- for instance, to see whether a variable has a value:

```
if(is_null($x)){  
    // $x is NULL  
}
```

Variables

Variables in PHP are identifiers prefixed with a dollar sign (\$). For example:

```
$name  
$Age  
$_debugging  
$MAXIMUM_IMPACT
```

A variable may hold a value of any type. There is no compiler-time or runtime type checking on variables. You can replace a variable's value with another of a different type:

```
$what = "Fred";  
$what = 35;  
$what = array("Fred",35,"Wilma");
```

There is no explicit syntax for declaring variables in PHP. The first time the value of a variable is set, the variable is created in memory. In other words, setting a value to a variable also functions as a declaration. For example, this is a valid complete PHP program:

```
$day = 60*60*24  
echo "There are {$day} seconds in a day,";  
//Output There are 86400 seconds in a day.
```

A variable whose value has not been set behaves like the NULL value:

```
if($uninitializedVariable === NULL){  
    echo "Yes!";  
}  
//OUTPUT: Yes!
```

Variable Variables

You can reference the value of a variable whose name is stored in another variable by prefacing the variable reference with an additional dollar sign (\$). For example:

```
$foo = "bar";  
$$foo = "baz";
```

After the second statement executes, the variable \$bar has the value "baz".

Variable References

IN PHP, references are how you create variable aliases or pointers. To make \$black an alias for the variable \$white, use:

```
$black =& $white;
```

The old value of \$black, if any, is lost. Instead, \$black is now another name for the value that is stored in \$white:

```
$bigLongVariableName = "PHP";  
$short =& $bigLongVariableName;  
$bigLongVariableName .= " rocks!";  
print "\$short is $short <br/>";  
print "Long is $bigLongVariableName";  
/*  
OUTPUT:  
$short is PHP rocks!  
Long is PHP rocks!  
*/  
  
$short = "Programming $short";  
print"\$short is $short <br/>";  
/*  
OUTPUT:  
$short is Programming PHP rocks!  
Long is Programming PHP rocks!  
*/
```

After the assignment, the two variables are alternate names for the same value. Unsetting an aliased variable does not affect other names for that variable's value, however:

```
$while = "snow";  
$black =& $while;  
unset($while);  
print $black;  
/*  
* OUTPUT: snow  
*/
```


Functions can return values by reference (for example, to avoid copying large strings or arrays, as discussed in Chapter 3):

```
$while = "snow";
$black =& $white;
unset($white);
print $black;
/*
 * OUTPUT: snow
 */
```

Variable Scope

The scope of a variable, which is controlled by the location of the variable's declaration, determines those parts of the program that can access it. There are four types of variable scope in PHP: local, global, static, and function parameters.

Local scope

A variable declared in a function is local to that function. It is visible only to code in that function (excepting nested function definitions); it is not accessible outside the function. In addition, by default, variables defined outside a function (global variables) are not accessible inside the function. For example, here's a function that updates a local variable instead of a global variable:

```
function updateCounter()
{
    $counter++;
}
$counter = 10;
updateCounter();
echo $counter;
/*
 * OUTPUT:
 * 10
 */
```

The `$counter` inside the function is local to that function because we haven't said otherwise. The function increments its private `$counter` variable, which is destroyed when the subroutine ends. The global `$counter` remains set at 10.

Only functions can provide local scope. Unlike in other languages, in PHP you can't create a variable whose scope is a loop, conditional branch, or other type of block.

Global scope

Variables declared outside a function are global. That is, they can be accessed from any part of the program. However, by default, they are not available inside functions. To allow a function to access a

global variable within the function. Here's how we can rewrite the `updateCounter()` function to allow it to access the global `$counter` variable:

```
function updateCounter()
{
    global $counter;
    $counter++;
}
$count = 10;
updateCounter();
echo $counter;
/**
 * OUTPUT: 11
 */
```

A more cumbersome way to update the global variable is to use PHP's `$GLOBALS` array instead of accessing the variable directly:

```
function updateCounter()
{
    $GLOBALS['counter']++;
}
$counter = 10;
updateCounter();
echo $counter;
/*
 OUTPUT 11
 */
```

Static variables

A static variable retains its value between calls to a function but is visible only within that function. You declare a variable static with the `static` keyword. For example:

```
function updateCounter()
{
    static $counter = 0;
    $counter++;

    echo "Static counter is now {$counter}<br/>"
}
$counter = 10;
updateCounter();
updateCounter();

echo "Global counter is {$counter}";
```

```
/*  
    OUTPUT:  
    Static counter is now 1  
    Static counter is now 2  
    Global counter is 10  
*/
```

Function parameters

As we'll discuss in more detail in Chapter 3, a function definition can have named parameter:

```
function greet($name)  
{  
    echo"Hello, {$name}";  
}  
greet("Janet");  
  
/*  
OUTPUT: Hello, Janet  
*/
```

Function parameters are local, meaning they are available only inside their functions. In this case, \$name is inaccessible from outside greet().

Garbage Collection

PHP uses reference counting and copy-on-write to manage memory. Copy-On-write ensures that memory isn't wasted when you copy values between variables, and reference counting ensures that memory is returned to the operating system when it is no longer needed.

To understand memory management in PHP, you must first understand the idea of a symbol table. There are two parts to a variable –its name (e.g, \$name), and its value (e.g, "Fred"). A symbol table is an array that maps variable names to the position of their values in memory.

When you copy a value from one variable to another, PHP doesn't get more memory for a copy of the value. Instead, it updates the symbol table to indicate that "both of these variables are names for the same chunk of memory." So the following code doesn't actually create a new array:

```
$worker = array("Fred", 35, "Wilma");  
$other = $worker; // array isn't duplicated in memory
```

If you subsequently modify either copy, PHP allocates the required memory and makes the copy:

```
$worker[1] = 36; // array is copied in memory,. value changed
```

By delaying the allocation and copying, PHP saves time and memory in many situations. This is copy-on-write.

Each value pointed to by a symbol table has a reference count, representing the number of ways to get to that piece of memory. After the initial assignment of the array to \$worker and \$worker to \$other, the array pointed to by the symbol table entries for \$worker and \$other has a reference count of 2(it is

actually three if you are looking at the reference count from the C API, but for this explanation and from a user-space perspective, it is easier to think of it as 2). In other words, that memory can be reached through `$worker` or `$other`. But after `$worker[1]` is changed, PHP creates a new array for `$worker`, and the reference count of each array is only 1.

When a variable goes out of scope at the end of a function, such as function parameters and local variables, the reference count of its value is decreased by one. When a variable is assigned a value in a different area of memory, the reference count of the old value is decreased by one. When the reference count of a value reaches 0, its memory is released. This is reference counting.

Reference counting is the preferred way to manage memory. Keep variables local to functions, pass in values that the functions need to work on, and let reference counting take care of the memory management. If you do insist on trying to get a little more information or control over freeing a variable's value, use the `isset()` and `unset()` functions.

To see if a variable has been set to something—even the empty string—use `isset()`:

```
$s1 = isset($name); // $s1 is false
$name = "Fred";
$s2 = isset($name); // $s2 is true
```

Use `unset()` to remove a variable's value:

```
$name = "Fred";
unset($name); // $name is NULL
```

Expressions and Operators

An expression is a bit of PHP code that can be evaluated to produce a value. The simplest expressions are literal values and variables. A literal value evaluates to itself, while a variable evaluates to the value stored in the variable. More complex expressions can be formed using simple expressions and operators.

An operator takes some values (the operands) and does something (e.g., adds them together). Operators are sometimes written as punctuation symbols—for instance, the `+` and `-` are familiar to us from math. Some operators modify their operands, while most do not.

Table 2-3 summarizes the operators in PHP, many of which were borrowed from C and Perl. The column labeled “P” gives the operator's precedence; the operators are listed in precedence order, from highest to lowest. The column labeled “A” gives the operator's associativity, which can be L (left-to-right), R (right-to-left), or N (nonassociative).

Table 2-3. PHP operators

P	A	Operator	Operation
24	N	clone, new	Create new object
23	L	[Array subscript
22	R	**	Exponentiation
21	R	~	Bitwise NOT
	R	++	Increment
	R	--	Decrement
	R	(int), (bool), (float), (string), (array), (object), (unset)	Cast
	R	@	Inhibit errors
20	N	instanceof	Type testing
19	R	!	Logical NOT
18	L	*	Multiplication
	L	/	Division
	L	%	Modulus
17	L	+	Addition
	L	-	Subtraction
	L	.	String concatenation
16	L	<<	Bitwise shift left
	L	>>	Bitwise shift right
P	A	Operator	Operation
15	N	<, <=	Less than, less than or equal
	N	>, >=	Greater than, greater than or equal
14	N	==	Value equality
	N	!=, <>	Inequality
	N	===	Type and value equality
	N	!==	Type and value inequality
	N	<=>	Returns an integer based on a comparison of two operands: 0 when left and right are equal, -1 when left is less than right, and 1 when left is greater than right.
13	L	&	Bitwise AND
12	L	^	Bitwise XOR
11	L		Bitwise OR
10	L	&&	Logical AND
9	L		Logical OR
8	R	??	Comparison
7	L	?:	Conditional operator
6	R	=	Assignment
	R	+=, -=, *=, /=, .=", %=", &=", =, ^=, ~=", <=<=", >=>="	Assignment with operation
5		yield from	Yield from
4		yield	Yield
3	L	and	Logical AND
2	L	xor	Logical XOR
1	L	or	Logical OR

Number of Operands

Most operators in PHP are binary operators; they combine two operands (or expressions) into a single, more complex expressions. PHP also supports several unary operators, converting a single expression into a more complex one. Finally, PHP supports a few ternary operators that combine numerous expressions into a single expression.

Operator Precedence

The order in which operators in an expression are evaluated depends on their relative precedence. For example, you might write:

```
2+4*3
```

Table 2-3 shows that the addition and multiplication operators have different precedence, with multiplication higher than addition. So the multiplication happens before the addition, giving $2 + 12$, or 14, as the answer. If the precedence of addition and multiplication were reversed, $6 * 3$, or 18, would be the answer.

To force a particular order, you can group operands with the appropriate operator in parentheses. In our previous example, to get the value 18, you can use the expression:

```
(2 + 4) * 3
```

It is possible to write all complex expressions (expressions containing more than a single operator) simply by putting the operands and operators in the appropriate order so that their relative precedence yields the answer you want. Most programmers, however, write the operators in the order that they feel makes the most sense to them, and add parentheses to ensure it makes sense to PHP as well. Getting precedence wrong leads to code like:

```
$x + 2 / $y >= 4 ? $z : $x << $z
```

This code is hard to read and is almost definitely not doing what the programmer expected it to do.

One way many programmers deal with the complex precedence rules in programming languages is to reduce precedence down to two rules:

- Multiplication and division have higher precedence than addition and subtraction.
- Use parentheses

Operator Associativity

Associativity defines the order in which operators with the same order of precedence are evaluated. For example, look at:

```
2 / 2 * 2
```

The division and multiplication operators have the same precedence, but the result of the expression depends on which operation we do first:

```
2 / (2 * 2) // 0.5  
(2/2) * 2   // 2
```

The division and multiplication operators are left-associative; this means that in cases of ambiguity, the operators are evaluated from left to right. In this example, the correct result is 2.

Implicit Casting

For instance, many operators have expectations of their operands; binary math operators typically require both operands to be of the same type. PHP's variables can store integers, floating-point numbers, strings, and more and to keep as much of the type details away from the programmer as possible, PHP converts values from one type to another as necessary.

The conversion of a value from one type to another is called casting. This kind of implicit casting is called type juggling in PHP. The rules of the type juggling done by arithmetic operators are shown in Table 2-4.

Table 2-4. Implicit casting rules for binary arithmetic operations

Type of first operand	Type of second operand	Conversion performed
Integer	Floating point	The integer is converted to a floating-point number.
Integer	String	The string is converted to a number; if the value after conversion is a floating-point number, the integer is converted to a floating-point number.
Floating point	String	The string is converted to a floating-point number.

Some other operators have different expectations of their operands and thus have different rules. For example, the string concatenation operator converts both operands to strings before concatenating them:

```
3 . 3.74//gives the string 32.74
```

You can use a string anywhere PHP expects a number. The string is presumed to start with an integer or floating-point number. If no number is found at the start of the string, the numeric value of that string is 0. If the string contains a period (.) or upper- or lowercase e, evaluating it numerically produces a floating-point number. For example:

```
"9 Lives" - 1; // 8 (int)
"3.14 Pies" * 2; // 6.28 (float)
"9. Lives" - 1; // 8 (float / double)
"1E3 Points of Light" + 1; // 1001 (float)
```

Arithmetic Operators

The arithmetic operators are operators you'll recognize from everyday use. Most arithmetic operators are binary; however, the arithmetic negation and assertion operators are unary. These operators require numeric values, and non-numeric values are converted into numeric values by the rules described in the section "Casting Operators." The arithmetic operators are:

Addition (+)

The result of the addition operator is the sum of the two operands.

Subtraction (-)

The result of the subtraction operator is the difference between the two operands—that is, the value of the second operand subtracted from the first.

Multiplication (*)

The result of the multiplication operator is the product of the two operands. For example, $3 * 4$ is 12.

Division (/)

The result of the division operator is the quotient of the two operands. Dividing two integers can give an integer (e.g., $4 / 2$) or a floating-point result (e.g., $1/2$).

Modulus(%)

The modulus operator converts both operands to integers and returns the remainder of the division of the first operand by the second operand. For example, $10 \% 6$ gives a remainder of 4.

Arithmetic negation(-)

The arithmetic negation operator returns the operand multiplied by -1, effectively changing its sign. For example, $-(3 - 4)$ evaluates to 1. Arithmetic negation differs from the subtraction operator, even though both are written as a minus sign. Arithmetic negation is always unary and before the operand. Subtraction is binary and between its operands.

Arithmetic assertion (+)

The arithmetic assertion operator returns the operand multiplied by +1, which has no effect. It is used only as a visual cue to indicate the sign of a value. For example, $+(3 - 4)$ evaluates to -1, just as $(3-4)$ does.

Exponentiation ()**

The exponentiation operator returns the result of raising \$var1 to the power of \$var2.

```
$var1 = 5;  
$var2 = 3;  
  
echo $var1 ** var2; //outputs 125
```


String Concatenation Operator

Manipulating strings is such a core part of PHP applications that PHP has a separate string concatenation operator(.). The concatenation operator appends the righthand operand to the lefthand operand and returns the resulting string. Operands are first converted to strings, if necessary. For example:

```
$n = 5;
$s = 'There were ' . $n . ' ducks.';
/*
//OUTPUT
$s is 'There were 5 ducks'
*/
```

The concatenation operator is highly efficient because so much of PHP boils down to string concatenation.

Auto-Increment and Auto-Decrement Operators

IN programming, one of the most common operations is to increase or decrease the value of a variable by one. The unary auto-increment (++) and auto-decrement (--) operators provide shortcuts for these common operations. These operators are unique in that they work only on variables; the operators change their operands' values and return a value.

There are two ways to use auto-increment or auto-decrement in expressions. If you put the operator in front of the operands, it returns the new value of the operands (incremented or decremented). If you put the operator after the operand, it returns the original value of the operand (before the increment or decrement). Table 2-5 lists the different operations.

Table 2-5. Auto-increment and auto-decrement operations

Operator	Name	Value returned	Effect on \$var
\$var++	Post-increment	\$var	Incremented
++\$var	Pre-increment	\$var + 1	Incremented
\$var--	Post-decrement	\$var	Decrement
--\$var	Pre-decrement	\$var - 1	Decrement

These operators can be applied to strings as well as numbers. Incrementing an alphabetic character turns it into the next letter in the alphabet. As illustrated in Table 2-6, incrementing “z” or “Z” wraps it back to “a” or “A” and increments the previous character by one (or inserts a new “a” or “A” if at the first

character of the string), as through the characters were in a base-26 number system.

Table 2-6. Auto-increment with letters

Incrementing this	Gives this
"a"	"b"
"z"	"aa"
"spaz"	"spba"
"K9"	"L0"
"42"	"43"

Comparison Operators

As their name suggests, comparison operators compare operands. The result is always either true if the comparison is truthful or false otherwise.

Operands to the comparison operators can be both numeric, both string, or one numeric and one string. The operators check for truthfulness in slightly different ways based on the types and values of the operands, whether using strictly numeric comparisons or using lexicographic (textual) comparisons. Table 2-7 outlines when each type of check is used.

Table 2-7. Type of comparison performed by the comparison operators

First operand	Second operand	Comparison
Number	Number	Numeric
String that is entirely numeric	String that is entirely numeric	Numeric
String that is entirely numeric	Number	Numeric
String that is entirely numeric	String that is not entirely numeric	Lexicographic
String that is not entirely numeric	Number	Numeric
String that is not entirely numeric	String that is not entirely numeric	Lexicographic

One important thing to note is that two numeric strings are compared as if they were numbers. If you have two strings that consist entirely of numeric characters and you need to compare them lexicographically, use the `strcmp()` function.

The comparison operators are:

Equality (==)

If both operands are equal, this operator returns true; otherwise, it returns false.

Identity (===)

If both operands are equal and are of the same type, this operator returns true; otherwise, it returns false. Note that this operator does not do implicit type casting. This operator is useful when you don't know if the values you're comparing are of the same type. Simple comparison may involve value conversion. For instance, the strings "0.0" and "0" are not equal. The == operator says they are, but === says they are not.

Inequality (!= or <>)

If the operands are not equal, this operator returns true; otherwise, it returns false.

Not identical (!==)

If the operands are not equal, or they are not of the same type, this operator returns true; otherwise, it returns false.

Greater than (>)

If the lefthand operand is greater than the righthand operand, this operator returns true; otherwise, it returns false.

Greater than or equal to (>=)

If the lefthand operand is greater than or equal to the righthand operand, this operator returns true; Otherwise, it returns false.

Less than (<)

If the lefthand operand is less than the righthand operand, this operator returns true; otherwise, it returns false.

Less than or equal to (<=)

If the lefthand operand is less than or equal to the righthand operand, this operator returns true; otherwise, it returns false.

Spaceship (< = >)

When the lefthand and righthand operands are equal, this operator returns 0; when the lefthand operand is less than the righthand, it returns -1; and when the lefthand operand is greater than the righthand, it returns 1.

```
$var1= 5;  
$var2= 65;  
  
echo $var1 <=> $var2; //outputs -1  
echo $var2 <=> $var1; //outputs 2
```

Null coalescing operator (??)

This operator evaluates to the righthand operand if the lefthand operand is NULL;
Otherwise, it evaluates to the lefthand operand.

```
$var1= 5;
$var2= 65;

echo $var1 <=> $var2; //outputs -1
echo $var2 <=> $var1; //outputs 2

$var1 = null;
$var2 = 31;

echo $var1 ?? $var2; //outputs 31
```

Bitwise Operators

The bitwise operators act on the binary representation of their operands. Each operand is first turned into a binary representation of the value, as described in the bitwise negation operator entry in the following list. All the bitwise operators work on numbers as well as strings, but they vary in their treatment of string operands of different lengths. The bitwise operators are:

Bitwise negation (~)

The bitwise negation operator changes 1s to 0s and 0s to 1s in the binary representations of the operands. Floating-point values are converted to integers before the operation takes place. If the operand is a string, the resulting value is a string the same length as the original, with each character in the string negated.

Bitwise AND (&)

The bitwise AND operator compares each corresponding bit in the binary representation of the operands. If both bits are 1, the corresponding bit in the result is 1; otherwise, the corresponding bit is 0. For example, 0755 & 0671 is 0651. This is easier to understand if we look at the binary representation. Octal 0755 is binary 111101101, and octal 0671 is binary 110111001. We can then easily see which bits are in both numbers and visually come up with the answer:

```
111101101
& 110111001
- - - - -
110101001
```

The binary number is 110101001. It is octal 0651. (Here's a tip: split the binary number into three groups—6 is binary 110, 5 is binary 101, and 1 is binary 001;

thus, 0651 is 110101001.) You can use the PHP functions `bindec()`, `decbin()`, `octdec()`, and `decoct()` to convert numbers back and forth when you are trying to understand binary arithmetic.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise AND operation between the two corresponding characters in the operands. The resulting string is the length of the shorter of the two operands; trailing extra characters in the longer string are ignored. For example, "wolf" & "cat" is "cad".

Bitwise OR (|)

The bitwise OR operator compares each corresponding bit in the binary representations of the operands. If both bits are 0, the resulting bit is 0; otherwise, the resulting bit is 1. For example, 0755 | 020 is 0775.

If both operands are strings, the operator returns a string in which each character is the result of a bitwise OR operation between the two corresponding characters in the operands. The result string is the length of the longer of the two operands, and the shorter string is padded at the end with binary 0s. For example, "pussy" | "car" is "suwsy".

Bitwise XOR (^)

The bitwise XOR operator compares each corresponding bit in the binary representation of the operands. If either of the bits in the pair, but not both, is 1, the resulting bit is 1; otherwise, the resulting bit is 0. For example, 0755 ^ 023 is 776. If both operands are strings, this operator returns a string in which each character is the result of a bitwise XOR operation. If the operands have different lengths, the resulting string is the length of the shorter operand, and extra trailing characters in the longer string are ignored. For example, "big drink" ^ "AA" is "#{".

Left shift (<<)

The left-shift operator shifts the bit in the binary representation of the lefthand operand left by the number of places given in the righthand operand. Both operands will be converted to integers if they aren't already. Shifting a binary number to the left inserts a 0 as the rightmost bit of the number and moves all other bits to the left one place. For ex, 3 << 1 (or binary 11 shifted one place left) results in 6 (binary 110).

Note that each place to the left where a number is shifted results in doubling the numbers. The result of left shifting is multiplying the lefthand operand by 2 to the power of the righthand operand.

Right shift (>>)

The right-shift operator shifts the bits in the binary representation of the lefthand operand right by the number of places given in the righthand operand. Both operands will be converted to integers if they aren't already. Shifting a positive binary number to the right inserts a 0 as the leftmost bit of the number and moves all other bits to the right one place. The rightmost bit is discarded. For ex, 13 >> 1 (or binary 1101 shifted one bit to the right, results in 6 (binary 110).

Logical Operators

Logical operators provide ways for you to build complex logical expressions. Logical operators treat their operands as Boolean values and return a Boolean value. There are both punctuation and English versions of the operators (|| and or are the same operator). The logical operators are:

Logical AND (&&, and)

The result of the logical AND operation is true if and only if both operands are true; otherwise, it is false. If the value of the first operand is false, the logical AND operator knows that the resulting value must also be false, so the righthand operand is never evaluated. This process is called short-circuiting, and a common PHP idiom uses it to ensure that a piece of code is evaluated only if something is true. For ex, you might connect to a database only if some flag is not false:

```
$result = $flag and mysqlconnect();
```

The && and 'and' operators differ only in their precedence: && comes before 'and'.

Logical OR (||, or)

The result of the logical OR operation is true if either operand is true; otherwise, the result is false. Like the logical AND operator, the logical OR operator is short-circuited. If the lefthand operator is true, the result of the operator must be true, so the right hand operator is never evaluated. A common PHP idiom uses this to trigger an error condition if something goes wrong. For example:

```
$result = fopen($filename) or exit();
```

The || and or operators differ only in their precedence.

Logical XOR (xor)

The result of the logical XOR operation is true if either operand, but not both, is true; otherwise, it is false.

Logical negation

The logical negation operator returns the Boolean value true if the operand evaluates to false, and false if the operand evaluates to true.

Casting Operators

Although PHP is a weakly typed language, there are occasions when it's useful to consider a value as a specific type. The casting operators (int), (float), (string), (bool), (object), and (unset), allows you to force a value into a particular type. To use a casting operator, put the operator to the left of the operand. Table 2-8 lists the casting operators, synonymous operators, and the type to which the operator changes the value.

Table 2-8. PHP casting Operators

Operator	Synonymous operators	Changes type to
(int)	(integer)	Integer
(bool)	(boolean)	Boolean
(float)	(double), (real)	Floating point
(string)		String
(array)		Array
(object)		Object
(unset)		NULL

Casting affects the way other operators interpret a value rather than changing the value in a variable. For example, the code:

```
$a = "5";  
$b = (int)$a;
```

Assigns \$b the integer value of \$a; \$a remains the string "5". To cast the value of the variable itself, you must assign the result of a cast back to the variable:

```
$a = "5";  
$b = (int)$a; // now $a holds an integer
```

Not every cast is useful. Casting an array to a numeric type gives 1 (if the array is empty, it gives 0), and casting an array to a string gives "Array" (Seeing this in your output is a sure sign that you've printed a variable that contains an array).

Casting an object to an array builds an array of the properties, the mapping property names to values:

```
class Person{  
    var $name = "Fred";  
    var $age = 35;  
}  
  
$o = new Person;  
$a = (array)$o;  
  
print_r($a);  
Array ([name] => Fred [age] => 35)
```

You can cast an array to an object to build an object whose properties correspond to the array's keys and values. For example:

```
$a = array('name' => "Fred", 'age' => 35, 'wife' => "Wilma");  
$o = (object) $a;  
echo $o->name;  
//output Fred
```

Keys that are not valid identifiers are invalid property names and are inaccessible when an array is cast to an object but are restored when the object is cast back to an array.

Assignment

The basic assignment operator (=) assigns a value to a variable. The lefthand operand is always a variable. The righthand operand can be any expression—any simple literal, variable, or complex expression. The righthand operand's value is stored in the variable named by the lefthand operand.

Because all operators are required to return a value, the assignment operator returns the value assigned to the variable. For example, the expression `$a = 5` not only assigns 5 to `$a`, but also behaves as the value 5 if used in a larger expression. Consider the following expressions:

```
$a = 5;  
$b = 10;  
$c = ($a = $b);
```

The expression `$a=$b` is evaluated first, because of the parentheses. Now, both `$a` and `$b` have the same value, 10. Finally, `$c` is assigned the result of the expression `$a = $b`, which is the value assigned to the lefthand operand (in this case, `$a`). When the full expression is done evaluating, all three variables contain the same value 10.

Assignment with operation

In addition to the basic assignment operator, there are several assignment operators that are convenient shorthand. These operators consist of binary operators followed directly by an equals sign, and their effect is the same as performing the operation with the full operands, then assigning the result value to the lefthand operand. These assignment operators are:

Plus-equals (+=)

Adds the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand. `$a += 5` is the same as `$a=$a+5`.

Minus-equals (-=)

Subtracts the righthand operand from the value of the lefthand operand, then assigns the result to the lefthand operand.

Divide-equals (/=)

Divides the value of the lefthand operand by the righthand operand, then assigns the result to the lefthand operand.

Multiply-equals (*=)

Multiplies the righthand operand by the value of the lefthand operand, then assigns the result to the lefthand operand.

Modulus-equals(%)

Performs the modulus operation on the value of the lefthand operand and the righthand operand, then assigns the results to the lefthand operand.

Bitwise-XOR-equals(^=)

Performs a bitwise AND on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

Bitwise-AND-equals(&=)

Performs a bitwise AND on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

Bitwise-OR-equals(|=)

Performs a bitwise OR on the value of the lefthand operand and the righthand operand, then assigns the result to the lefthand operand.

Concatenate-equals(.=)

Concatenates the righthand operand to the value of the lefthand operand, then assigns the result to the lefthand operand.

Miscellaneous Operators

The remaining PHP operators are for error suppression, executing an external command, and selecting values:

Error suppression(@)

Some operators or functions can generate error messages. The error suppression operator, discussed in full in Chapter 17, is used to prevent these messages from being created.

Execution('...')

The backtick operator executes the string contained between the backticks as a shell command and returns the output

. For example:

```
$listing = `ls -ls /tmp`;  
echo $listing;
```

Conditional (? :)

The conditional operator is, depending on the code you look at, either the most overused or most underused operator. It is the only ternary (three-operand) operator and is therefore sometimes just called the ternary operator. The conditional operator evaluates the expression before the ?. If the expression is true, the operator evaluates the expression before the ?. If the expression is true, the

operator returns the value of the expression between the ? and ;; otherwise, the operator returns the value of the expression after the .. For instance:

```
a href="<? echo $url; ?>"><? echo $linktest ? $linktest : $url; ?> </a>
```

If text for the link \$url is present in the variable \$linktest, it is used as the test for the link; otherwise, the URL itself is displayed.

Type(instanceof)

The instanceof operator tests whether a variable is an instantiated object of a given class or implements an interface (see Chapter 6 for more information on object and interfaces):

```
$a = new Foo;  
$isAFoo = $a instanceof Foo; //true  
$isABar = $a instanceof Bar; // false
```

Flow-Control Statements

PHP supports a number of traditional programming constructs for controlling the flow of execution of a program. Conditional statements, such as if/else and switch, allow a program to execute different pieces of code, or none at all, depending on some condition. Loops, such as while and for, support the repeated execution of particular segments of code.

If

The if statement checks the truthfulness of an expression and, if the expression is true, evaluates a statement. An if statement looks like:

```
if (expression) statement
```

To specify an alternative statement to execute when the expression is false, use the else keyword:

```
if(expression)  
    statement  
else statement
```

For example:

```
if($user_validate)  
    echo "Welcome!";  
else echo "Access Forbidden!";
```

To include more than one statement within an if statement, use a block—a set of statements enclosed by curly braces:

```
if($user_validate)  
{  
    echo "Welcome!";  
    $greeted = 1;  
}
```

```
else {  
    echo "Access Forbidden!";  
    exit;  
}
```

PHP provides another syntax for blocks in tests and loops. Instead of enclosing the block of statements in curly braces, end the if line with a colon (:) and use a specific keyword to end the block (endif, in this case). For example:

```
if($user_validate):  
    echo "welcome!";  
    $greeted = 1;  
else:  
    echo "Access Forbidden!";  
    exit;  
endif;
```

Other statements described in this chapter also have similar alternate syntax styles (and ending keywords); they can be useful if you have large blocks of HTML inside your statements. For example:

```
<?php if ($user_validated) : ?>  
    <table>  
        <tr>  
            <td>First Name</td><td>sophia</td>  
        </tr>  
  
        <tr>  
            <td>Last Name:</td><td>Lee</td>  
        </tr>  
    </table>  
  
    <?php else: ?>  
        Please log in.  
    <?php endif ?>
```

Because if is a statement, you can chain(embed) more than one. This is also a good example of how the blocks can be used to help keep things organized:

```
if($good){  
    print("Dandy!");  
}  
else{  
    if($error){  
        print("Oh, no!");  
    }  
    else{  
        print("I'm ambivalent...");  
    }  
}
```

```
}  
}
```

Such chains of if statements are common enough that PHP provides an easier syntax: the elseif statement. For example, the previous code can be rewritten as:

```
if($good){  
    print("Dandy!");  
}  
elseif($error){  
    print("Oh,no!");  
}  
else {  
    print("I'm ambivalent...");  
}
```

The ternary conditional operator (?:) can shorten simple true/false tests. Take a common situation, such as checking to see if a given variable is true and printing something if it is. With a normal if/else statement, it looks like this:

```
<td><?php if($active) {echo "yes";} else {echo "no";} ?></td>
```

With the ternary conditional operator, it looks like this:

```
<td><?php echo $active ? "yes" : "no";?></td>
```

Compare the syntax of the two:

```
if(expression){true_statement} else {false_statement}  
(expression) ? true_expression : false_expression
```

The main difference here is that the conditional operator is not a statement at all. This means that it is used on expression, and the result of a completed ternary expression is itself an expression. In the previous example, the echo statement is inside the if condition, while when used with the ternary operator, it precedes the expression.

Switch

The value of a single variable may determine one of a number of different choices (e.g., the variable holds the username and you want to do something different for each user). The switch statement is designed for just this situation.

A switch statement is given an expression and compares its value to all cases in the switch; all statements in a matching case are executed up to the first break keyword it finds. If none match, and a

default is given, all statements following the default keyword are executed, up to the first break keyword encountered. For example, suppose you have the following:

```
if ($name == 'ktatrow'){  
    // do something  
}else if($name == 'dawn'){  
    // do something  
}else if($name == 'petermac'){  
    // do something  
}else if($name == 'bobk'){  
    // do something  
}
```

You can replace that statement with the following switch statement

```
switch($name){  
    case 'ktatrow':  
        // do something  
        break;  
    case 'dawn':  
        // do something  
        break;  
    case 'petermac':  
        // do something  
        break;  
    case 'bobk':  
        // do something  
        break;  
}
```

The alternative syntax for this is:

```
switch($name):  
    case 'ktatrow':  
        // do something  
        break;  
    case 'dawn':  
        // do something  
        break;  
    case 'petermac':  
        // do something  
        break;  
    case 'bobk':  
        // do something  
        break;
```

```
endswitch;
```

Because statements are executed from the matching case label to the next break keyword, you can combine several cases in a fall-through. In the following example, “yes” is printed when \$name is equal to sylvie or Bruno:

```
switch($name){  
    case'sylvie':// Fall-through  
    case'bruno':  
        print("yes");  
        break;  
    default:  
        print("no");  
        break;  
}
```

Commentating the fact that you are using a fall-through case in a switch is a good idea, so someone doesn't come along at some point and add a break thinking you had forgotten it. You can specify an optional number of levels for the break keyword to break out of. In this way, a break statement can break out of several levels of nested switch statements. An example of using break in this manner is shown in the next section.

While

The simplest form of loop is the while statement

```
while(expression)statement
```

if the expression evaluates to true, the statement is executed and then the expression is re-evaluated (if it is still true, the body of the loop is executed again, and so on). The loop exits when the expression is no longer true (i.e evaluates to false).

As an example, here's some code that adds the whole numbers from 1 to 10:

```
$total = 0;  
$i = 1;  
  
while($i <= 10){  
    $total += $i;  
    $i++;  
}
```

The alternative syntax for while has this structure:

```
while(expr):  
    statement;  
    more statements;  
endwhile;
```

For example:

```
$total =0;
$i = i;

while($i<= 10):
    $total +=$i;
    $i++;
endwhile;
```

You can prematurely exit a loop with the break keyword. In the following code, \$i never reaches a value of 6, because the loop is stopped once it reaches 5:

```
while($i <= 10){
    if($i == 5){
        break; // breaks out of the loop
    }
    $total +=$i;
    $i++;
}
```

Optionally, you can put a number after the break keyword indicating how many levels of loop structures to break out of. In this way, a statement buried deep in nested loops can break out of the outermost loop. For example:

```
$i = 0;
$j = 0;

while($i <10){
    while($j <10){
        if($j == 5){
            break 2; // breaks out of two while loops
        }
        $j++;
    }
    $i++
}

echo "{$i}, {$j}";
//OUTPUT 0,5
```

The continue statement skips ahead to the next test of the loop condition. As with the break keyword, you can continue through an optional number of levels of loop structure:

```
while($i < 10){
    $i++;

    while($j < 10){
        if($j ==5){
```

```
                continue 2; // continues through two levels
            }

            $j++;
        }
    }
```

In this code, \$j never has a value above 5, but \$i goes through all values from 0 1- 9. PHP also supports a do/while loop, which takes the following form:

```
do
    statement
while (expression)
```

Use a do/while loop to ensure that the loop body is executed at least once (the first time):

```
$total = 0;
$i = 1;

do{
    $total += $i++;
}while ($i <= 10);
```

You can use break and continue statements in a do/while statement just as in a normal while statement.

The do/while statement is sometimes used to break out of a bloc of code when an error condition occurs. For example:

```
do{

    // do some stuff

    if($errorCondition){
        break;
    }

    // do some other dtuff
}while (false);
```

Because the condition for the loop is false, the loop is executed only once, regardless of what happens inside the loop. However, if an error occurs, the code after the break is not evaluated.

For

The for statement is similar to the while statement, except it adds counter inicialarion and counter manipulation expressions, and is often shorter and easier top read than the equivalent while loop.

Here's a while loop that counts from 0 to 9, printing each number:

```
$counter = 0;

while($counter< 10)
```



```
{  
    echo "Counter is {$counter} <br/>";  
    $counter++;  
}
```

Heres the corresponding, more concise for loop:

```
for($counter = 0; $counter < 10; $counter++){  
    echo "Counter is $counter <br/>";  
}
```

The structure of a for statement is:

```
for(start; condition; increment){statement(s);}
```

The expression start is evaluated once, at the beginning of the for statement. Each time through the loop, the expression condition is tested. If it is true, the body of the look is executed if it is false, the loop ends. The expression increment is evaluated after the loop body runs.

The alternative syntax of a for statement is:

```
for(expr1; expr2; expr3):  
    statement;  
    ...;  
endfor;
```

This program adds the numbers from 1 to 10 using a for loop:

```
for($i = 1; $i<=10; $i++){  
    $total += $i;  
}
```

Here's the same loop using the alternate syntax:

```
for ($i = 1; $i <= 10; $i+):  
    $total += $i;  
endfor;
```

You can specify multiple expressions for any of the expressions in a for statement by separating the expressions with commas. For example:

```
$total = 0;  
for($i=0, $j=1; $i<=10; $i++, $j*=2){  
    $total =0;  
}
```

You can also leave an expression empty, signaling that nothing should be done for that phase. In the most degenerate form, the for statement becomes an infinite loop. Yoy probably don't want to run this example as it never stops printing:

```
for(;;){  
    echo "Can't stop me <br />";  
}
```

For educational or internal use only

Luka Bostick 2023

In for loops, as in while loops, you can use the break and continue keywords to end the loop or current iteration.

Foreach

The foreach statement allows you to iterate over elements in an array. The two forms of the foreach statement are further discussed in Chapter 5, where we talk in more depth about arrays. To loop over an array, accessing the value at each key use:

```
foreach($array as $current){  
    //...  
}
```

The alternate syntax is

```
foreach($array as $current):  
    //...  
endforeach;
```

To loop over an array, accessing both key and value, use:

```
foreach($array as $key => $value)  
{  
    //...  
}
```

The alternate syntax is

```
foreach($array as $key => $value):  
    //...  
endforeach;
```

try...catch

The try...catch construct is not so much a flow-control structure as it is a more graceful way to handle system errors. For example, if you want to ensure that your web application has a valid connection to a database before continuing, you could write code like this:

```
try{  
    $dbhandle = new PDO('mysql:host=localhost;  
dbname=library', $username, $pwd);  
    doDB_Work($dbhandle); // call function on gaining a  
connection  
    $dbhandle = null;  
}  
catch (PDOException $error){  
    print "Error!: " . $error->getMessage() . "<br/>";  
    die();  
}
```

Here the connection is attempted with the try portion of the construct and if there are any errors with it, the flow of the code automatically falls into the catch portion, where the PDO exception class is instantiated into the \$error variable. It can then be displayed on the screen, and the code can “gracefully” fail, rather than making an abrupt end. You can even redirect to try connecting to an alternate database, or respond to the error any other way you wish within the catch portion.

Declare

The declare statement allows you to specify execution directives for a block of code. The structure of a declare statement is:

```
declare (directive) statement
```

Currently, there are only three declare forms: the ticks, encoding, and strict_types directives. You can use the ticks directive to specify how frequently (measured roughly in number of code statements) a tick function is registered when register_tick_function() is called. For example:

```
register_tick_function("someFunction");  
    declare(ticks = 3){  
        for($i=0; $i <10; $i++){  
            //do something  
        }  
    }
```

In this code, someFunction() is called every third statement within the block is executed. You can use the encoding directive to specify a PHP script’s output encoding. For example:

```
declare(encoding="UTF-8");
```

This form of the declare statement is ignored unless you compile PHP with the –enable-zend-multibyte option. Finally you can use the strict_type directive to enforce the use of a strict data type when defining and using variables.

exit and return

As soon as it is reached the exit statement ends the script’s execution. The return statement returns from a function or, at the top level of the program, from the script. The exit statement takes an optional value. If this is a number, it is the exit status of the process. If it is a string, the value is printed before the process terminates. The function die() is an alias for this form of the exit statement

```
$db = mysql_connect("localhost", $USERNAME,$PASSWORD);  
    if(!$db){  
        die("Could not connect to database");  
    }
```

This is more commonly written as:

```
$db = mysql_connect("localhost",$USERNAME,$PASSWORD) or  
    die("could not connect to database");
```

See Chapter 3 for more information on using the return statement in functions.

Goto

The goto statement allows execution to “jump” to another place in the program. You specify execution points by adding a label which is an identifier followed by a colon (:). You then jump to the label from another location in the script via the goto statement:

```
for($i=0; $i < $count; $i++)  
    {  
        //oops, found an error  
        if($error){  
            goto cleanup;  
        }  
    }  
  
cleanup:  
    // do some cleanu
```

You can only goto a label within the same scope as the goto statement itself, and you can't jump into a loop or switch. Generally, anywhere you might use a goto (or multilevel break statement, for that matter), you can rewrite the code to be cleaner without it.

Including Code

PHP provides two constructs to load code and HTML from another module: require and include. Both load a file as a PHP script runs, work in conditionals and loops, and complain if the file being loaded cannot be found. Files are located by an included file path as part of the directive in the user of the function, or based on the setting of include_path in the php.ini file. The include_path can be overridden by the set_include_path() function. If all these avenues fail, PHP's last attempt is to try to find the file in the same directory as the calling script. The main difference is that attempting to require a nonexistent file is a fatal error, while attempting to include such a file produces a warning but does not stop script execution.

A common use of include is to separate page-specific content from general site design. Common elements such as headers and footers go in separate HTML files, and each page then looks like:

```
<?php include "header.html";?>  
content  
<?php include "footer.html";?>
```

We use include because it allows PHP to continue to process the page even if there's an error in the site design file(s). The required construct is less forgiving, and it is more suited to loading code libraries when the page cannot be displayed if the libraries do not load. For example:

```
require "codelib.php";  
mysub(); // defined in codelib.php
```

A marginally more efficient way to handle headers and footers is to load a single file and then call functions to generate the standardized site elements:

```
<?php require "design.php"; header();?>
```

```
content
<?php footer();
```

If PHP cannot parse some part of a file added by include or require, a warning is printed and executed continues. You can silence the warning by prepending the call with the silence operator(@)-for example, @include. If the allow_url_fopen option is enabled through PHP's configuration file, php.ini, you can include files from a remote site by providing a URL instead of a simple local path:

```
include "http://www.example.com/codelib.php";
```

If the filename begins with http://, https://, or ftp://, the file is retrieved from a remote site and loaded. Files included with include and require can be arbitrarily named. Common extensions are .php,.php5,and .html.

NOTE: that remotely fetching a file that ends in .php from a web server that has PHP enabled fetches the output of that PHP script. It executes the PHP code in that file.

If a program uses include or require to include the same file twice (mistakenly done in a loop, for example), the file is loaded, and the code is run, or the HTML is printed twice. This can result in errors about the redefinition of functions or multiple copies of headers or HTML being sent. To prevent these errors from occurring, use the include_once and require_once constructs. They behave the same as include and require the first time a file is loaded, but quietly ignore subsequent attempts to load the same file. For example, many page elements, each stored in separate files, need to know the current user's preferences. The element libraries should load the user preferences library with require_once. The page designer can then include a page element without worrying about whether the user preference code has already been loaded.

Code in an included file is imported at the scope that is in effect where the include statement is found, so the included code can see and alter your code's variables. This can be useful-for instance, a user-tacking library might store the current user's name in the global \$user variable:

```
// main page
include "userprefs.php";
echo"Hello, {$user}.";
```

The ability of libraries to see and change your variables can also be a problem. You have to know every global variable used by a library to ensure that you don't accidentally try to use one of them for your own purposes, thereby overwriting the library's value and disrupting how it works.

If the include or require construct is in a function, the variables in the included file become function-scope variables for that function.

Because include and require are keywords, not real statements, you must always enclose them in curly braces in conditional and loop statements:

```
for($i=0; $i<10; $i++){
    include "repeated_element.html";
}
```

Use the get_included_files() function to learn which files your script has included or required. It returns any array containing the full system path filenames of each included or required file. Files that did not parse are not included in this array.

Embedding PHP in Web Pages

Although it is possible to write and run standalone PHP programs, most PHP code is embedded in HTML or XML files. This is, after all, why it was created in the first place/ Processing such documents involves replacing each chunk of PHP source code with the output it produces when executed.

Because a single file usually contains PHP and non-PHP source code, we need a way to identify the regions of PHP code to be executed. PHP provides four different ways to do this.

As you'll see, the first and preferred method looks like XML. The second method looks like SGML. The third method is based on ASP tags. The fourth method uses the standard HTML `<script>` tag; this makes it easy to edit pages with enabled PHP using a regular HTML editor.

Standard (XML) Style

Because of the advent of the eXtensible Markup Language (XML) and the migration of HTML to an XML language (XHTML), the currently preferred technique for embedding PHP uses XML-compliant tags to denote PHP instructions.

Coming up with tags to demark PHP commands in XML was easy, because XML allows the definition of new tags. To use this style, surround your PHP code with `<?php` and `?>`. Everything between these markers is interpreted as PHP, and anything outside the markers is not. Although it is not necessary to include spaces between the markers and the enclosed test, doing so improves readability. For example, to get PHP to print "Hello, world," you can insert the following line in a web page:

```
<?echo "Hello, world"; ?>
```

The trailing semicolon on the statement is optional, because the end of the block also forces the end of the expression. Embedded in a complete HTML file, this looks like:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset='utf-8'>
  <meta http-equiv='X-UA-Compatible' content='IE=edge'>
  <title>This is my first PHP program</title>
  <meta name='viewport' content='width=device-width, initial-scale=1'>
  <link rel='stylesheet' type='text/css' media='screen' href='main.css'>
  <script src='main.js'></script>
</head>

<body>
  <p>
    Look, ma! It's my first PHP program:<br/>
    <?php echo "Hello, world";?><br/>
    how cool is that?
```

```
</p>

</body>
</html>
```

Notice that there's no trace of the PHP course code from the original file. The user sees only its output. Also, notice that we switched between PHP and non-PHP, all in the space of a single line. PHP instructions can be put anywhere in a file, even within valid HTML tags. For example:

```
<input type="text" name="first_name" value="<?php echo Peter"; ?>" />
```

When PHP is done with this text, it will read:

```
<input type="text" name="first_name" value="Peter" />
```

The PHP code within the opening and closing markers does not have to be on the same line. If the closing marker of a PHP instruction is the last thing on a line, the line break following the closing tag is removed as well. Thus, we can replace the PHP instructions in the “Hello, world” example with:

```
<?php
echo "Hello, world";
?><br/>
```

With no change in the resulting HTML.

SGML Style

Another style of embedding PHP comes from SQLML instruction processing tags. To use this method, simply enclose the PHP in `<? And ?>`. Here's the “Hello, world” example again:

```
<? echo "Hello, world";?>
```

This style, known as short tags, is off by default. You can turn on support for short tags by building PHP with the `--enable-short-tags` option or `enable_short_open_tag` in the PHP configuration file. This is discouraged as it depends on the state of this setting; if you export your code to another platform, it may or may not work. The short echo tag, `<?= ... ?>`, is available regardless of the availability of short tags.

Echoing Content Directly

Perhaps the single most common operation within a PHP application is displaying data to the user. In the context of a web application, this means inserting into the HTML document information that will become HTML when viewed by the user. To simplify this operation, PHP provides a special version of the SQLML tags that automatically take the value inside the tag and insert it into the HTML page. To use this feature, add an equal sign (=) to the opening tag. With this technique, we can rewrite our form example as:

```
<input type="text" name="first_name" value="<?= "Dawn"; ?>" />
```

What's Next

Now that you have the basics of the language under your belt—a foundational understanding of what variables are and how to name them, what data types are, and how code flow control works—we will move on to some finer details of the PHP language. Next we'll cover three topics that are so important to

For educational or internal use only

Luka Bostick 2023

PHP that they each have their own dedicated chapters: how to define functions (Chapter 3), manipulate strings (Chapter 4), and manage arrays (Chapter 5).