Unit 1: Introduction to PHP

PHP with Framework (CMS & E-Commerce)

What is PHP? Why PHP?

- PHP is an open source, server-side, HTML-embedded web-scripting language that is compatible with all the major web servers (most notably Apache).
- PHP enables you to embed code fragments in normal HTML pages — code that is interpreted as your pages are served up to users.
- PHP also serves as a "glue" language, making it easy to connect your web pages to server-side databases.
- It's free, it's open source, it's full featured, it's cross-platform, it's stable, it's fast, it's clearly designed, it's easy to learn, and it plays well with others.

- PHP is the web development language written by and for web developers.
- PHP stands for PHP: Hypertext Preprocessor. The product was originally named Personal Home Page Tools.
- PHP is a server-side scripting language, usually used to create web applications in combination with a web server, such as Apache.
- PHP can also be used to create command-line scripts akin to Perl or shell scripts, but such use is much less common than PHP's use as a web language.

What Is MySQL?

- MySQL is an open source, SQL relational database management system (RDBMS) that is free for many uses (more detail on that later).
- MySQL found a broad, enthusiastic user base for its liberal licensing terms, perky performance, and ease of use.
- Its acceptance was aided in part by the wide variety of other technologies such as PHP, Perl, Python, and the like that have encouraged its use through stable, well-documented modules and extensions.

Deciding on a Web Application Platform

- There are many platforms upon which web applications can be built. Comparison can be done on following factors:
- Cost
- Ease of Use
- HTML-embeddedness
- Cross-platform compatibility
- Many extensions
- Fast feature development
- Not proprietary
- Strong user communities

Server- Side Scripting Overview

- Static HTML
- Client-Side Technologies: Cascading Style Sheets (CSS) and Dynamic HTML; client-side scripting languages, such as JavaScript; VBScript; Java applets; and Flash
- The best thing about client-side technologies is also the worst thing about them: They depend entirely on the browser.
- Wide variations exist in the capabilities of each browser and even among versions of the same brand of browser.
- Individuals can also choose to configure their own browsers in awkward ways: Some people disable JavaScript for security reasons

- **Server-Side Scripting**: server-side scripting is invisible to the user.
- Server-side web scripting is mostly about connecting web sites to backend servers, processing data and controlling the behavior of higher layers such as HTML and CSS. This enables the following types of two-way communication:
  - Server to client: Web pages can be assembled from backend-server output.
  - Client to server: Customer-entered information can be acted upon.

- Common examples of client-to-server interaction are online forms with some drop-down lists (that the script assembles dynamically on the server.
- Server-side scripting products consist of two main parts: the scripting language and the scripting engine. The engine parses and interprets pages written in the language.

What Is Server-Side Scripting Good For?

- Server-side scripting languages such as PHP perfectly serve most of the truly useful aspects of the web, such as the items in this list:
  - Content sites (both production and display)
  - Community features (forums, bulletin boards, and so on)
  - E-mail (web mail, mail forwarding, and sending mail from a web application)
  - Customer-support and technical-support systems
  - Advertising networks
  - Web-delivered business applications

- Surveys, polls, and tests
- Filling out and submitting forms online
- Personalization technologies
- Groupware
- Catalog, brochure, and informational sites
- Games (for example, chess) with lots of logic but simple/static graphics
- Any other application that needs to connect a backend server

Getting started with PHP

- Escaping from HTML: How does the PHP parser recognize PHP code inside your HTML document?
  - You tell the program when to spring into action by using special PHP tags at the beginning and end of each PHP section. This process is called ***escaping from HTML or escaping into PHP***.
  - Everything within these tags is understood by the PHP parser to be PHP code. The tags are known as Canonical PHP tags
  - <?php ?>

Hello World

```
<HTML>
   <HEAD>
      <TITLE>My first PHP program</TITLE>
   </HEAD>
   <BODY>
      <?php
            print("Hello, World<BR />\n");
            phpinfo();
      ?>
   </BODY>
</HTML>
```

Jumping in and out of PHP mode

- At any given moment in a PHP script, you are either in PHP mode or you're out of it in HTML.

```
<?php $id = 1; ?>
<FORM METHOD="POST" ACTION="registration.php">
<INPUT TYPE="HIDDEN" NAME="serial number"
   VALUE="<?php echo $id; ?>">
```

Including files

- Another way you can add PHP to your HTML is by putting it in a separate file and calling it by using
- PHP's include functions. There are four include functions:
  - include('/filepath/filename')
  - require('/filepath/filename')
  - include_once('/filepath/filename')
  - require_once('/filepath/filename')

- Include() and include_once() will merely generate a warning on failure, while require() and require_once() will cause a fatal error and termination of the script.
- As suggested by the names of the functions, include_once() and require_once() differ from simple include() and require() in that they will allow a file to be included only once per PHP script.
- The most common use of PHP's include capability is to add common headers and footers to all the web pages on a site.

Example

- Header File

```
<HTML>
  <HEAD>
    <TITLE>A site title</TITLE>
  </HEAD>
  <BODY>
```

- Footer File

```
    <P>Copyright 1995 - 2002</P>
  </BODY>
</HTML>
```

- They are called from a PHP page this way:

```
<?php
require_once($_SERVER['DOCUMENT_ROOT'].'/header.php');
?>
<P>This is some body text for this particular page.</P>
<?php
require_once($_SERVER['DOCUMENT_ROOT'].'/footer.php');
?>
```

Learning PHP Syntax and Variables

- Some points to remember
  - PHP is whitespace insensitive
  - In PHP all variables are case sensitive
  - Statements are expressions terminated by semicolons
  - Precedence, associativity, and evaluation order is same as C
  - C-style multiline comments /* */
  - Single-line comments: # and //

Variables

- Here are the most important things to know about variables in PHP:
  - All variables in PHP are denoted with a leading dollar sign ($)
  - The value of a variable is the value of its most recent assignment
  - Variables are assigned with the = operator, with the variable on the left-hand side and the expression to be evaluated on the right
  - Variables can, but do not need, to be declared before assignment.
  - Variables have no intrinsic type other than the type of their current value
  - Variables used before they are assigned have default values

- Assigning Variable
  - $pi = 3 + 0.14159; // approximately
  - $pi = "3 + 0.14159";
- Reassigning
  - $my_num_var = "This should be a number";
  - $my_num_var = 5;
- Unassigned variables : Default Values
  - In a situation where a number is expected, a number will be produced, and this works similarly with character strings.
  - In any context that treats a variable as a number, an unassigned variable will be evaluated as 0; in any context that expects a string value, an unassigned variable will be the empty string

- **Checking assignment with isset**

```
if (isset($set_var))
    print("set_var is set.<BR>");
else
    print("set_var is not set.<BR>");
```

- The function unset() will restore a variable to an unassigned state

Variable scope

- Variables assigned within a function are *local to that function, and unless you make a special declaration in a function, that* function won't have access to the global variables defined outside the function, even when they are defined in the same file.

Constants

- Constants, which have a single value throughout their lifetime. Constants do not have a $ before their names, and by convention the names of constants usually are in uppercase letters.
- Constants can contain only scalar values (numbers and string).
- Constants have global scope, so they are accessible everywhere in your scripts after they have been defined — even inside functions.
- For example, the built-in PHP constant E_ALL represents a number that indicates to the error_reporting() function that all errors and warnings should be reported. A call to error_reporting() might look like this:
  error_reporting(E_ALL);

- It's also possible to create your own constants using the define() function. The code:
  define(MY_ANSWER, 42);
  would cause MY_ANSWER to evaluate to 42 everywhere it appears in your code.
- There is no way to change this assignment after it has been made.
- Example: sales-tax rate or perhaps an exchange rate

Types in PHP

- PHP makes it easy not to worry too much about typing of variables and values, both because it does not require variables to be typed and because it handles a lot of type conversions for you.
- No variable type declarations: the type of a variable does not need to be declared in advance.
- Instead, the programmer can jump right ahead to assignment and let PHP take care of figuring out the type of the expression assigned:
- $first_number = 55.5;
- $second_number = “Not a number at all”;

- PHP will do the right thing when, for example, doing math with mixed numerical types. The result of the expression
- $pi = 3 + 0.14159;
- is a floating-point (double) number, with the integer 3 implicitly converted into floating point before the addition is performed

- The Simple Types:
  - integers,
  - doubles,
  - Booleans,
  - NULL, and
  - strings
- Examples of Boolean
  - $true_num = 3 + 0.14159;
  - $true_str = "Tried and true";
  - $true_array[49] = "An array element"; // see next section
  - $false_array = array();
  - $false_null = NULL;
  - $false_num = 999 – 999;
  - $false_str = ""; // a string zero characters long

Printing Output

- Echo and print: The two most basic constructs for printing to output are echo and print.
- Their language status is somewhat confusing, because they are basic constructs of the PHP language, rather than being functions. As a result, they can be used either with parentheses or without them.
- **Echo:** The simplest use of echo is to print a string as argument
  - echo "This is statement";
  - echo("This is statement");

- **Print:**
- The command print is very similar to echo, with two important differences:
  - Unlike echo, print can accept only one argument.
  - Unlike echo, print returns a value, which represents whether or not the print statement succeeded.
- The value returned by print is always 1.

Variables and strings:

```
$animal = "antelope";
$animal_heads = 1;
$animal_legs = 4;
print("The $animal has $animal_heads head(s).<BR>");
print("The $animal has $animal_legs leg(s).<BR>");
```

HTML and linebreaks:

\n will not work in PHP. We need to use HTML
 tag for new line

PHP Control Structures and Functions

- The two broad types of control structures we will talk about are *branches* and *loops.*
- *A branch is a fork in the road for a program's execution* — depending on some test or other, the program goes either left or right, possibly following a different path for the rest of the program's execution.
- A loop is a special kind of branch, where one of the execution paths jumps back to the beginning of the branch, repeating the test and possibly the body of the loop.

- **Boolean Expressions:** Every control structure in this chapter has two distinct parts: the *test (which determines which part* of the rest of the structure executes), and the *dependent code itself (whether separate branches or the* body of a loop).
- Tests work by evaluating a *Boolean expression, an expression with a result treated as* either true or false.
- **Boolean Constant:** The simplest kind of expression is a simple value, and the simplest Boolean values are the constants TRUE and FALSE.
- We can use these constants anywhere we would use a more complicated Boolean expression, and vice versa

```
if (TRUE)
    print("This will always print<BR>");
else
    print("This will never print<BR>");
```

Or equivalently:

```
if (FALSE)
    print("This will never print<BR>");
else
    print("This will always print<BR>");
```

- **Logical operators:** Logical operators combine other logical (aka Boolean) values to produce new Boolean values.
- The standard logical operations (and, or, not, and exclusive-or) are supported by PHP.
  - and
  - Or
  - !
  - xor
  - &&
  - ||

- **Comparison operators: ==, !=, <, >,<=,>=, ===**
- **String comparison:** The comparison operators may be used to compare strings as well as numbers
- **The ternary operator:**
  testExpression ? yesExpression : noExpression
  $max_num = $first_num > $second_num ? $first_num : $second_num;

Branching

- The two main structures for branching are if and switch.
- Switch is a useful alternative for certain situations where you want multiple possible branches based on a single value and where a series of if statements would be cumbersome.

```
if ($day == 5)
    print("Five golden rings<BR>");
elseif ($day == 4)
    print("Four calling birds<BR>");
elseif ($day == 3)
    print("Three French hens<BR>");
elseif ($day == 2)
    print("Two turtledoves<BR>");
elseif ($day == 1)
    print("A partridge in a pear tree<BR>");
```

- **Switch:** For a specific kind of multiway branching, the switch construct can be useful.
- Rather than branch on arbitrary logical expressions, switch takes different paths according to the value of a single expression.
- The expression can be a variable or any other kind of expression, as long as it evaluates to a simple value (that is, an integer, a double, or a string).
- The construct executes by evaluating the expression and then testing the result for equality against each case value
- As soon as a matching value is found, subsequent statements are executed in sequence until the special statement (break;) or until the end of the switch construct.
- A special default tag can be used at the end, which will match the expression if no other case has matched it so far.

```
switch(expression)
{
    case value-1:
    statement-1;
    statement-2;
    ...
    [break;]
    case value-2:
    statement-3;
    statement-4;
    ...
    [break;]
    ...
    [default:
    default-statement;]
}
```

Looping

- **Bounded loops versus unbounded loops:** A *bounded loop executes a fixed number of times — you can tell by looking at the code how many* times the loop will iterate, and the language guarantees that it won't loop more times than that.
- An *unbounded loop repeats until some condition becomes true (or false), and that condition is dependent* on the action of the code within the loop.
- Bounded loops are predictable, whereas unbounded loops can be as tricky as you like.

- **While:** The simplest PHP looping construct is while, which has the following syntax:
  while (*condition)*
  statement
- The while loop evaluates the *condition expression as a Boolean — if it is true, it executes statement* and then starts again by evaluating *condition.*
- *If the condition is false, the while loop terminates.*

```
$count = 1;
while ($count <= 10)
{
    print("count is $count<BR>");
    $count = $count + 1;
}
```

- **Do-while:** The do-while construct is similar to while, except that the test happens at the end of the loop.

do *statement*
while (*expression);*

- The statement is executed once, and then the expression is evaluated. If the expression is true, the statement is repeated until the expression becomes false.

```
$count = 45;
do
{
    print("count is $count<BR>");
    $count = $count + 1;
} while ($count <= 10);
```

- **For:** The most complicated looping construct is for, which has the following syntax:
  for (*initial-expression; termination-check; loop-end-expression)*
  statement
- In executing a for statement, first the *initial-expression is evaluated just once, usually to initialize* variables.
- Then *termination-check is evaluated — if it is false, the for statement concludes, and if it is* true, the statement executes.
- Finally, the *loop-end-expression is executed and the cycle begins again* with *termination-check.*
- *As always, by statement we mean a single (semicolon-terminated) statement, a* brace-enclosed block, or a conditional construct.

- **Break and continue:** The standard way to get out of a looping structure is for the main test condition to become false.
- The special commands break and continue offer an optional side exit from all the looping constructs, including while, do-while, and for:
  - The break command exits the innermost loop construct that contains it.
  - The continue command skips to the end of the current iteration of the innermost loop that contains it.

- **Terminating Execution:**
- Sometimes you just have to give up, and PHP offers a construct that helps you do just that. The exit() construct takes either a string or a number as argument, prints out the argument, and then terminates execution of the script.
- Everything that PHP produces up to the point of invoking exit() is sent to the client browser as usual, and nothing in your script after that point will even be parsed — execution of the script stops immediately.
- If the argument given to exit is a number rather than a string, the number will be the return value for the script's execution. Because exit is a construct, not a function, it's also legal to give no argument and omit the parentheses.

- The die() construct is an alias for exit() and so behaves exactly the same way.
- A better use for die() is to make your crashes informative. It's good to get into the habit of testing for unexpected conditions that would crash your script if they were true, and throw in a die() statement with an informative message.
- If you're correct in your expectations, the die() will never be invoked; if you're wrong, you will have an error message of your own rather than a possibly obscure PHP error.

```
$connection = make_database_connection();
if (!$connection)
   die("No database connection!");
use_database_connection($connection);
```

Using Functions

- The basic syntax for using (or *calling) a function is:* function_name(expression_1, expression_2, …, expression_n)
- This includes the name of the function followed by a parenthesized and comma-separated list of input expressions (which are called the *arguments to the function).*
- *Functions can be called with zero* or more arguments, depending on their definitions.
- When PHP encounters a function call, it first evaluates each argument expression and then uses these values as inputs to the function.
- After the function executes, the returned value (if any) is the result of the entire function expression.

- All the following are valid calls to built-in PHP functions:
  - sqrt(9); // square root function, evaluates to 3
  - rand(10, 10 + 10); // random number between 10 and 20
  - strlen("This has 22 characters"); // returns the number 22
  - pi(); // returns the approximate value of pi
- These functions are called with 1, 2, 1, and 0 arguments, respectively

Defining Your Own Functions

- What is a function?
  - A *function is a way of wrapping up a chunk of code and giving that chunk a name, so that you can* use that chunk later in just one line of code.
  - Functions are most useful when you will be using the code in more than one place, but they can be helpful even in one-use situations, because they can make your code much more readable.
- Function definition syntax

function *function-name ($argument-1, $argument-2, ..)*

{

statement-1;

statement-2;

...

}

- That is, function definitions have four parts:
  - The special word function
  - The name that you want to give your function
  - The function's parameter list — dollar-sign variables separated by commas
  - The function body — a brace-enclosed set of statements
- Just as with variable names, the name of the function must be made up of letters, numbers, and underscores, and it must not start with a number.
- Unlike variable names, function names are converted to lowercase before they are stored internally by PHP, so a function is the same regardless of capitalization.

1. PHP looks up the function by its name (you will get an error if the function has not yet been defined).
2. PHP substitutes the values of the calling arguments (or the *actual parameters) into the variables* in the definition's parameter list (or the *formal parameters).*
3. The statements in the body of the function are executed. If any of the executed statements are return statements, the function stops and returns the given value. Otherwise, the function completes after the last statement is executed, without returning a value.

- Formal parameters versus actual parameters
- Argument number mismatches
  - Too few arguments
  - Too many arguments
- Functions and Variable Scope: Each function is its own little world.
  - That is, barring some special declarations, the meaning of a variable name inside a function has nothing to do with the meaning of that name elsewhere.
- Global versus local: The scope of a variable defined inside a function is *local by default.*
  - Using the global declaration, you can inform PHP that you want a variable name to mean the same thing as it does in the context outside the function.
  - The syntax of this declaration is simply the word global, followed by a comma-delimited list of the variables that should be treated that way, with a terminating semicolon.

```
function SayMyABC ()
{
    global $count;
    while ($count < 10)
    {
        print(chr(ord('A') + $count));
        $count = $count + 1;
    }
    print("<BR>Now I know $count letters<BR>");
}
```

- **Static variables:** The static declaration causing variables to retain their values in between calls to the same function.
- **Recursion**

Passing Information with PHP

- **HTTP Is Stateless:** that is why form-handling technology like PHP comes in
- HTML forms are mostly useful for passing a few values from a given page to one single other page of a web site.
- There are more persistent ways to maintain state over many pageviews, such as cookies and sessions.
- The most basic techniques of information-passing between web pages, which utilize the GET and POST methods in HTTP to create dynamically generated pages and to handle form data.

GET Arguments

- The GET method passes arguments from one page to the next as part of the *Uniform Resource*
- *Indicator* (you may be more familiar with the term *Uniform Resource Locator*, or *URL*) query string.
- When used for form handling, GET appends the indicated variable name(s) and value(s) to the URL designated in the ACTION attribute with a question mark separator and submits the whole thing to the processing agent.
- Example
- When the user makes a selection and clicks the Submit button
- The browser thus constructs the URL string:

http://<your-server-name>/sports.php?Sport=Ice+Hockey&Submit=Select

- The PHP script to which the preceding form is submitted (sports.php) will grab the GET variables from the end of the request string, stuff them into the $_GET superglobal array (explained in a moment), and do something useful with them — in this case, plug one of two values into a text string.

- There are the two main methods for passing values: GET and POST (there are others).
- Each method has an associated superglobal array, which can be distinguished from other arrays by the underscore that begins its name.
- Each item submitted via the GET method is accessed in the handler via the $_GET array;
- each item submitted via the POST method is accessed in the handler via the $_POST array.
- The syntax for referencing an item in a superglobal array is simple and 100 percent consistent:
- $_ARRAY_NAME['index_name']
- where the index_name is the name part of a name-value pair (for the GET method), or the name of an
- HTML form field (for the POST method).

- The GET method of form handling offers one big advantage over the POST method: It constructs an actual new and differentiable URL query string. Users can now bookmark this page. The result of forms using the POST method is not bookmarkable.

- The disadvantages of GET for most types of form handling are so substantial that the original HTML 4.0 draft specification deprecated its use in 1997. These flaws include:
- The GET method is not suitable for logins because the username and password are fully visible onscreen as well as potentially stored in the client browser's memory as a visited page.
- Every GET submission is recorded in the web server log, data set included.
- Because the GET method assigns data to a server environment variable, the length of the URL is limited. You may have seen what seem like very long URLs using GET — but you really wouldn't want to try passing a 300-word chunk of HTML-formatted prose using this method.

POST Arguments

- POST is the preferred method of form submission today, particularly where there will be result in permanent changes, such as adding information to a database.
- The form data set is included in the body of the form when it is forwarded to the processing agent (in this case, PHP).
- No visible change to the URL will result according to the different data submitted.
- The POST method has one primary advantage:
  - There is a much larger limit on the amount of data that can be passed (a couple of megabytes rather than a couple of hundred characters).

- POST has these disadvantages:
  - The results at a given moment cannot be bookmarked.
  - Browsers exhibit different behavior when the visitor uses their Back and Forward navigation buttons within the browser.

Formatting Form Variables