

Programming in PHP

NYU/SCPS X52.9224-1 Spring 2005

Instructor : David Mintz <dmintz@davidmintz.org>

http://davidmintz.org/php_course/

Session One

Outline

What is PHP? The Long Answer.

High-level view of how web servers, HTTP and PHP work together.

Why PHP? Reasons to learn and use PHP.

Language Basics, Part One: PHP syntax, variables, control structures.

Assignment: Installing PHP, Apache, MySQL

PHP: A Brief History

Once upon a time, in Geneva around 1990, Tim Berners-Lee proposed an idea (not without inspiration from work and ideas of others) to facilitate collaboration among physicists working at Organisation Européenne pour la Recherche Nucléaire (CERN): a networked, platform-neutral system for sharing documents. The document format was to be called HTML – HyperText Markup Language -- and the transport protocol was to run over TCP/IP and be called HTTP – HyperText Transfer Protocol. The first HTML documents consisted of static text, and the earliest http clients – web browsers – were command line, text-based tools.

By the end of 1992 there were approximately 50 distinct physical web servers running in the world. By April 2001 there were over 24 million. You can do the math yourself and see why the word *explosion* comes to mind. One wonders how many people in 1992 imagined that the World Wide Web would eventually be used as it is today.

In the early days the web served static documents. In 1993 the Common Gateway Interface was developed at NCSA (National Center for Supercomputing Applications). CGI specified a standard for passing data between web clients and programs residing on web servers. The data sent by the client – e.g., submitted as HTML form variables – becomes the input to the program. The program's output is collected by the web server and sent back to the client. You could now use CGI for interactive applications like guestbooks, querying databases, and so on.

Early implementations of CGI typically used Perl – also known as the Duct Tape of the Internet – because it was a logical choice: powerful, well-suited for manipulating text, relatively fast, freely available. However, each individual http request to a Perl/CGI script required the web server to fire up a distinct, fresh copy of the Perl interpreter to run the CGI program. For busy sites, this overhead became prohibitive, and brought about a demand for more efficient implementations of CGI. Some of the solutions people eventually came up with include the Apache http server module `mod_perl` (which essentially blends the Perl interpreter with the web server); Java Servlets and Java Server Pages (JSP); and **PHP**.

Rasmus Lerdorf introduced the first version of Personal Home Page Tools (PHP Tools) in June 1995. At that point PHP was a suite of CGI binaries written in C with features including logging, form input manipulation, hit counters, and access control. A simple parser would pick certain tags out of the HTML and call the corresponding C functions. PHP was not yet a full-blown scripting language in its own right.

Rasmus became motivated to bring it to the next level by a project that demanded still greater efficiency, a need that happened to coincide with the rise of the venerable and now world-famous Apache http server. Apache made it straightforward for a skilled programmer to add functionality like PHP to the server itself. When the next major PHP version appeared in April 1996, it now fashioned itself a “server-side HTML-embedded scripting language” with support for such cool things as SQL queries (database access) embedded right in your HTML.

By 1997 PHP adoption had grown considerably, but the underlying parsing engine still had stability problems and the project was essentially a one-person affair. Zeev Suraski and Andi Gutmans, two programmers in Tel Aviv, volunteered to rewrite the parsing engine which became the basis of PHP 3. Other people stepped up to work on other parts of the project, and PHP evolved into a classic open source project.

In June of 1998 PHP 3 was released, now boasting support for all the major web servers and a wide range of databases. It was also a programmer-friendly language well suited to people with little or no programming experience. At this point PHP was in use on over 70,000 sites world wide; after PHP 3, the growth curve began to climb extremely steeply. PHP Version 4 came out in May 2000, featuring more sophisticated internals and the more advanced Zend parsing engine. By November 2001, according to a Netcraft survey, PHP was serving sites at over one million unique IP addresses. The number of actual domains using PHP in according to Netcraft was nearly 20% of the 36 million plus domains they surveyed – over 7 million. By April 2002, PHP was reportedly being used by over 24% of the sites on the Internet. Of the 37.6 million web sites reported worldwide (<http://www.netcraft.com/Survey/index-200204.html>), PHP was running on over 9 million sites and continuing to grow. You can see current figures at <http://www.php.net/usage.php>.

PHP 5 featured numerous substantial changes, perhaps most significantly in its support for Object Oriented Programming. The latest version as of this writing (February 2005) is 5.0.3, and although many consider it sufficiently stable for production use, PHP 4 is still more widely used.

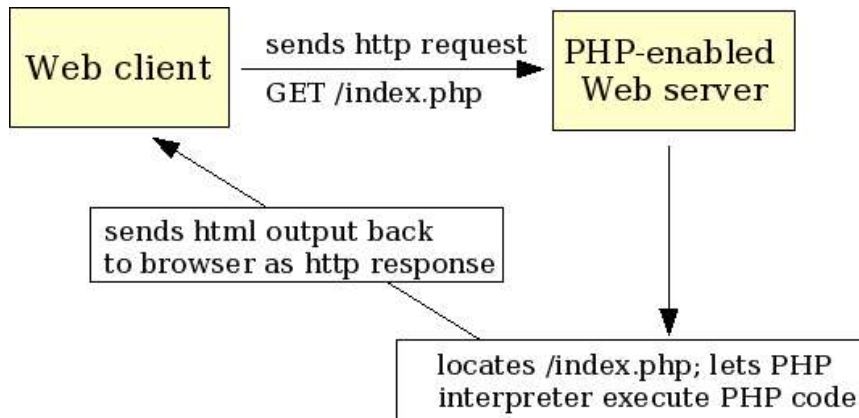
Why Use PHP?

- Ease of use. Unlike Perl or Java, PHP was designed for use on the WWW from day one; all kinds of facilities for web programming are built right in.
- With syntax based largely on C, and with influences from Perl and Java, PHP is easy for experienced programmers to learn quickly; for the inexperienced, it provides an excellent first language.
- While PHP is most often used for server-side processing (generating pages dynamically), it is also a command line tool and even supports client side graphical user interface (GUI) application programming with the PHP-GTK extension.
- Works with all major OSes.
- Works with all major web servers
- Flexibility: supports procedural, OOP, or both.
- Can generate alternative content types. You are not limited to outputting text; you can also generate graphics, PDF documents, and even Flash content on the fly.
- Solid support a vast range of databases.
- Powerful text processing: plenty of built-in string manipulation and regular expression functions.
- Outstanding support in the form of free code libraries (see e.g., <http://pear.php.net/>); countless online support and training resources in the form of articles, tutorials, forums and email lists; tons of books; magazines; user groups and a large, thriving international community.
- This is an excellent time to jump in. PHP is mature and is not going away.

PHP Language Basics

[Note: Parts of the following are substantially stolen from various sections of the official PHP manual <http://php.net/manual/en/>]

A PHP program is a text file containing zero or more *statements* (i.e., instructions) that tell the PHP interpreter what to do. When configured properly, the web server on which the program resides hands off certain files – most typically, files named *.php – to the PHP interpreter for execution of any PHP code contained in the file. Then the server sends the resulting *output* (rather than the PHP source code) back to the browser.



One consequence of the fact that PHP pages are dynamically generated is that if you want to develop PHP on your own system, you have to run a web server with PHP support. You test a PHP page by pointing your browser at it on your own server via http; you cannot view the output by accessing it directly through the file system (e.g., file:///C:/some_folder/some_file.php). Fortunately, running your own server is not at all difficult; more about that later.

You can have HTML and PHP code in the same physical file. Opening and closing tags tell the PHP parser where the PHP code begins and ends. Everything else is sent to the browser as is. Just as you can have PHP and HTML together in the same page, PHP will also gladly execute a file consisting of just HTML or just PHP code.

```

<p>Some plain old html</p>
<p>
<?php
    // outputs Hi everybody to the standard output -- usually a browser
    echo "Hi everybody.";
    // assigns the sum of 2 and 2 to the variable $sum
    $sum = 2 + 2;
    // you can probably guess the output of:
    echo " 2 + 2 is $sum";

?>
</p>
<p>More plain old html</p>
  
```

Short tags `<? echo "foo"; ?>` and asp-style tags `<% echo "foo" %>` can be turned on a setting in PHP's master configuration file, which by default is called `php.ini`. `<?php echo $foo ?>` and `<script language="PHP">//code</script>` are always available; first is most common, and preferred.

Each *instruction* is delimited with a semicolon. The closing `?>` tag is equivalent to a semicolon, and therefore optional. in this situation:

```

<?php
    echo "This is a test";
?>
  
```

```
<?php echo "This is a test" ?>
```

Whitespace is not significant; it is used to aid legibility.

Comments

Comments are exactly that. The PHP interpreter ignores them; they are for the benefit of human readers of the source code. They are typically used as a sort narrative to explain what is happening in your code, or like post-it notes to remind you of things to think about during development. Comments are also handy for temporarily disabling lines of code.

Unlike HTML comments, your PHP comments do not become part of the output visible to the browser.

So-called C-style and C++ style are supported:

```
<?php
    echo "This is a test"; // This is a one-line c++ style comment
    /* This is a multi line comment
       yet another line of comment */
    echo "This is yet another test";
?>
```

...as are shell- or Perl-style comments, with a leading # :

```
<?php
# here's a comment
    $foo = 23; # another comment here
?>
```

Variables

The notion of variables is fundamental to computer programming. They allow programs to behave dynamically and produce different output based on – well, variable – information that is decided at *runtime* (i.e., when the program is executed as opposed to when it is written (and possibly, compiled)). Variables are often supplied through user input. (Think of Google's search results) They can be compared to boxes with labels on them. The labels are the variable names. The stuff the boxes contain is data of some kind. We typically save stuff in these labeled boxes now so we can identify them and do something with their contents later.

A valid variable name starts with a letter or underscore, followed by any number of letters, numbers, or underscores. You assign a value to a variable with the assignment operator: = Hence:

```
$number = 7 ;
```

is how you would store the value 7 in a variable called \$number. We will discuss PHP operators later on.

Data types

PHP has eight "primitive" data types – primitive as in fundamental: anything fancier (more complex) is made out of these.

In general you do not have to be as concerned about the data types of your variables as you do with more strictly typed languages. PHP converts things for you automatically much of the time. There are nonetheless situations where you will need to be aware of the kind of data you are dealing with. (Also,

when you work with databases such as MySQL you will find that they are pickier about what types of data you are handing them, so it is useful to start thinking in these terms).

There are two compound data types: **array** and **object**. We will talk about arrays next week and objects in a few weeks.

There are also two special types: *resource* and *NULL*. Resources typically represent a link to something external to PHP, like a database connection or filehandle; we will get involved with these later. The special NULL value represents that a variable has no value at all. NULL is the only possible value of type NULL. It is useful in certain circumstances – we'll see examples in the weeks to come.

PHP has four *scalar* types: **boolean**; **integer**; **float** (aka double); and **string**. Scalar means a single thing, as opposed to a compound or complex thing.

boolean is the simplest type: it has only two possible values, either TRUE or FALSE. You often use some kind of expression that evaluates to a boolean which determines the flow of execution:

```
<?php if ($temperature > 75) { echo "it's too hot in here !" ; } ?>
```

integers are whole numbers (negative or positive). They can be notated in decimal, octal or hexadecimal notation. We'll be using decimal exclusively (or very nearly) so in this class. But you should be aware that in octal notation you precede the number with a zero, like so:

```
<?php
    $number = 03724;
    echo $number; // outputs: 2004
?>
```

And in hexadecimal, 2004 looks like this:

7d4

The max value depends on your operating system; for most applications you will find it is big enough (around 2 billion).

floating point numbers (or simply **floats**) are numbers like with a decimal point, like 1.432. The max value is about 1.8e308

a **string** is a series of characters. This is the data type you will probably work with most frequently, since a great many web pages consist mostly of text. The most common way to tell PHP something is a string is to put it in quotes – which makes sense, if you think about it. Single quotes and double quotes are both supported, but they can produce different results, so the choice is not just aesthetic.

Here, the contents of \$string1 and \$string2 will be identical.

```
<?php
$string1 = "Hello world";
$string 2 = 'Hello world';
?>
```

However, the double quotes support *variable interpolation* like so:

```
<?php
$string = 'world';
echo "Hello $string"; // prints "Hello world" because $string gets expanded
    // versus //
echo 'Hello $string'; // literally prints Hello $string
?>
```

Further, there are special characters like `\n` and `\t` that expand to newline and tab respectively in double- (but not single-) quoted strings. See <http://php.net/manual/en/language.types.string.php> for a complete list.

When you need to embed a literal dollar sign within a double-quoted string, you have to *escape* it by preceding it with a backslash.

```
<?php echo "The value of \$string is now $string"; ?>
```

The same is true when you need to embed a single quote in a single-quoted string, or a double-quote in a double-quoted string.

```
<?php echo 'John\'s cat\'s name is George'; ?>
```

The quotes, whether double or single, have to come in matching pairs, i.e., the first to signify the beginning and the second to signify the end of the string; otherwise you get a parse error. (The same is true of other punctuation characters that PHP has pressed into service: brackets, curly braces, and parentheses -- all of which we will encounter shortly.)

Remember our earlier comments about data types, and octal integer notation? Sometimes you do need to care about what kind of data you are working with:

```
<?php
$zip = 07302;
echo "zip is $zip\n";
    // versus
$zip = '07302';
echo "zip is $zip\n";
?>
```

Above, the quotes tell PHP 'this is a string.' Leave them out, and PHP thinks you mean an integer in octal notation.

You can -- and sometimes you have to -- use curly braces around string-embedded variables to "protect" them. For example:

```
<?php
$beers = 45;
$beer = 'Sierra Nevada Pale Ale';
echo "it would be excessive to drink $beers {$beer}s";
?>
```

Besides double and single quotes, there is a third way of writing string expressions known as the *heredoc syntax*. Provide an identifier after `<<<`, then the string, and then the same identifier to close the quotation. Heredoc text behaves just like a double-quoted string, without the double-quotes; variables are expanded. You do not have to escape quotes in your here docs, but you can still use escape codes like `\n` and `\t`.

```
<?php
$str = <<<EOD
Example of string spanning multiple lines
using heredoc syntax. You could have $foo
here and it would expand.
EOD;
?>
```

The closing identifier must begin in the first column (the beginning) of the line. Also, the identifier used must follow the same naming rules as any other label in PHP: it must contain only alphanumeric characters and underscores, and must start with a non-digit character or underscore.

The line with the closing identifier can contain no other characters, except possibly a semicolon (;). The identifier may not be indented, and there may not be any spaces or tabs after or before the semicolon. Further, the first character before the closing identifier must be a newline as defined by your operating system. (This is `\r` on Macintosh for example.)

If this rule is broken and the closing identifier is not "clean" then it's not considered to be a closing identifier and PHP will continue looking for one. If in this case a proper closing identifier is not found then a parse error will result with the line number reported as being at the end of the script.

Now that we've introduced (most of) PHP's data types, let's go back to the boolean type and talk about PHP's notion of truth and falsehood.

Truth and falsehood according to PHP

For PHP, only the following are false:

- the boolean value FALSE itself;
- the empty string "";
- the string "0";
- the integer 0;
- the float 0.0;
- the NULL value;
- an empty array (an array with no elements in it -- we will get to arrays later);
- an object with no properties set (hereagain, objects will come up later).

Absolutely everything else is TRUE, including negative numbers, the string "false", or anything else you can think of that you might intuitively assume to be false.

Operators

In this discussion we will run through the operators that you will use most frequently. For an exhaustive (and authoritative) treatment please see <http://us3.php.net/manual/en/language.operators.php>, from which much of the following is stolen.

The **precedence** of an operator specifies how "tightly" it binds two expressions together. For example, in the expression $1 + 5 * 3$, the answer is 16 and not 18 because the multiplication ("*") operator has a higher precedence than the addition ("+") operator. Parentheses may be used to force precedence. For instance: $(1 + 5) * 3$ evaluates to 18. If operator precedence is equal, left to right associativity is used to break the tie, so to speak.

associativity means the order in which PHP reads the expression -- either left to right, or right to left -- if precedence does not determine what gets evaluated first. For example, you might wonder whether the output of this would be 8 or -6:

```
<?php
    echo 4 - 3 + 7;
?>
```

Nobody who reads your code -- not even you -- wants to spend time pondering questions like the above, so a good rule of thumb is simply: always use parentheses, whether it's strictly necessary or not.

The **arithmetic operators** behave as you would expect: + - * / and % for addition, subtraction, multiplication, division, and modulus, respectively.

The most common of the **assignment operators** is surely =, as in

```
<?php $foo = 23; ?>
```

The above example assigns 23, the value on the right side (of the = operator), to the variable \$foo on the left. You can't do it the other way around because 23 is a literal value rather than a variable.

Note that an assignment expression itself evaluates to the value assigned. And since assignment associates from right to left, you will sometimes see things like

```
<?php
    $a = $b = 23;
?>
```

In addition to the basic assignment operator, there are "**combined operators**" for all of the binary arithmetic and string operators that allow you to use a value in an expression and then set its value to the result of that expression: += -= /= *= %=

You'll probably find the following example more comprehensible than the preceding sentence:

```
<?php
    $foo = 3;
    $foo += 2; // sets foo to 3 + 2, or 5
    $foo *= 2; // sets foo to 5 * 2
?>
```

Incrementing/Decrementing Operators: PHP (like other languages) provides the ++ and -- auto-increment and auto-decrement operators. They increase (or decrease) by the value of the variable that they are applied to. There are two flavors of each: pre- and post- . In the case of the former, the value is changed and assigned to the variable *before* it is evaluated in the expression where it appears; with the latter, the value changes afterwards. In both cases the value is increased/decreased by one; the difference is simply a matter of when. Example:

```
<?php
    $x = $y = 1;
    // increments y before the string concatenation
    echo "value of y is " . ++$y . "\n";
    // increments x after the string concatenation
    echo "value of x is " . $x++ . "\n";
?>
```

Output:

```
value of y is 2
value of x is 1
```

The **comparison operators** are

```
$a == $b    // true if $a and $b are equal
$a === $b   // true if $a and $b are equal AND the same data type
$a != $b    // true if $a and $b are not equal
$a <> $b    // same as above
$a !== $b   // true if $a and $b are not equivalent OR not of the same type
$a < $b     // true if $a is less than $b
$a > $b     // true if $a is greater than $b
$a <= $b    // true if $a is equal to or less than $b
$a >= $b    // true if $a is equal to or greater than $b
```

If you compare an integer with a string, PHP automatically converts the string to a number. If you compare two numerical strings, they are compared as integers.

TIP: Beware of saying = when you mean ==. Remember that an assignment expression itself evaluates to the value assigned. Remember also what we just noted about truth and falsehood. That means if you say 'assign' like so

```
<?php
    if ($a = 4) { // do something }
?>
```

when you really meant to say == ('compare'), your if-condition will always evaluate to true because 4 is considered true, regardless of whether \$a was true or false before that point in your program. (If you don't understand this now, don't worry: you will eventually learn it the hard way (-:))

The **logical operators** are analogous to our everyday understanding of the words 'and' and 'or' and 'not':

or	The expression (\$a or \$b) is true if either \$a or \$b is true.
and	The expression (\$a and \$b) is true if both \$a and \$b are true.
! (not/negation)	The expression (! \$a) is true if \$a is false.
xor (exclusive or)	The expression (\$a xor \$b) is true if <i>either</i> \$a <i>or</i> \$b is true but <i>not both</i> .

There are also the && and || operators that also mean 'and' and 'or', respectively, but have higher precedence than their English-like counterparts. (We will see an example later in which that fact allows you to write compact, clear and idiomatic code.) Here's an example of an expression involving 'or' and comparison operators:

```
<?php
    if ($temperature > 75 or $temperature < 60) {
        echo "I'm uncomfortable";
    } else {
        echo "I'm fine thanks";
    }
?>
```

There are just two **string operators** (the concatenation operator) and .= (concatenating assignment operator).

```
<?php
$string = "Hello";
$string = "Hello" . " "; // appends a space character to $string
$string .= "World";      // appends "World" to $string
echo $string;            // outputs "Hello World"
?>
```

Finally, there is the error suppression operator: @. It suppresses any error output from the expression that it precedes. We will understand this better later on when we talk about error handling.

Operators we are not talking about: because we will only very rarely use them in this class, for the sake of saving time we are skipping the **bitwise operators**. See <http://www.php.net/manual/en/language.operators.bitwise.php> if you are truly curious.

Control structures

If you are already familiar with other program languages, you will probably find PHP's control structures familiar. If on the other hand PHP is your first language, then you'll find this knowledge helpful if you ever decide to study another language.

Control flow structures manage the flow of execution of your program. They (along with *variables*) make your program smart enough to make decisions about what to do based on conditions that are not known to your program until runtime.

if

`if` is perhaps the most fundamental construct in computer programming: it allows for conditional execution of a fragment of code. In pseudocode, the syntax is:

```
if (expression) {
    statement
}
```

If `expression` is true, `statement` is executed; otherwise it is ignored, and the flow of execution drops through to whatever comes after the `if` block. You can have multiple statements, and you can nest `if` blocks as deep as desired.

Note the curly braces. They are optional for single line statements (but in my opinion omitting them is a bad habit). However, they must be paired, i.e., you need a left curly and a right curly, or you will have problems.

Also, note the indentation of `statement`. Although PHP does not require it, indenting your blocks aids legibility immensely; not indenting is considered poor style. These remarks apply to all the control structures, not just `if`.

This is important enough to bear repeating: indent your blocks. It will save you a world of pain.

else

`else` extends an `if` statement to execute a statement in case the expression in the `if` statement evaluates to FALSE. For example, the following code would display "a is bigger than b" if `$a` is bigger than `$b`, and "a is NOT bigger than b" otherwise.

```
<?php
if ($a > $b) {
    echo "a is bigger than b";
} else {
    echo "a is NOT bigger than b";
}
?>
```

elseif

`elseif`, as its name suggests, is a combination of `if` and `else`. Like `else`, it extends an `if` statement to execute a different statement in case the original `if` expression evaluates to FALSE. However, unlike `else`, it will execute that alternative expression only if the `elseif` conditional expression evaluates to TRUE. For example, the following code would display a is bigger than b, a equal to b or a is smaller than b:

```
<?php
if ($a > $b) {
    echo "a is bigger than b";
} elseif ($a == $b) {
    echo "a is equal to b";
} else {
    echo "a is smaller than b";
}
?>
```

while

`while` is for *looping*, that is, repeating some operation as long as some condition is true. In pseudocode, the syntax is:

```
while (expression)
    statement
```

`while` can be thought of as similar to `if`, except that instead of executing once, it keeps executing `statement` unless and until `expression` is false. If `expression` is false the first time it is evaluated, the code is executed zero times.

What do you suppose the output of the following fragment would be?

```
<?php
$i = 1;
while ($i <= 10) {
    echo $i++;
}
?>
```

If you said "12345678910" you'd be right.

do-while

`do-while` loops are similar to `while` loops, except the truth expression is checked at the end of each iteration instead of at the beginning. The main difference from regular `while` loops is that the first iteration of a `do-while` loop is guaranteed to run (the truth expression is only checked at the end of the iteration), whereas it may not necessarily run with a regular `while` loop (the truth expression is checked at the beginning of each iteration; if it evaluates to `FALSE` right from the beginning, the loop execution would end immediately).

```
<?php
$i = 0;
do {
    echo $i;
} while ($i > 0);
?>
```

You can get usually by without `do-while`; it is not frequently used.

for

`for` is the most complex of the PHP looping constructs, and it is frequently used.

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

The first expression (`expr1`) is evaluated (executed) once unconditionally at the beginning of the loop. In the beginning of each iteration, `expr2` is evaluated. If it evaluates to `TRUE`, the loop continues and the nested `statement(s)` are executed. If it evaluates to `FALSE`, the execution of the loop ends. At the end of each iteration, `expr3` is evaluated (executed).

`for` is often used for keeping track of counters, like so:

```
<?php
for ($i = 1; $i <= 10; $i++) {
    echo $i;
}
```

```
}  
?>
```

`$i` is initialized to 1; `$i` is compared to 10; `$i` is incremented if and only if `$i` is equal to or less than 10.

There is more to be said about `for`, but the simple usage illustrated above is the most common. Please see <http://us3.php.net/manual/en/control-structures.for.php> if you're curious about the fine print.

foreach

`foreach` is for iterating through arrays. We will discuss `foreach` when we discuss arrays.

break and continue

`break` immediately ends execution of the current `for`, `foreach`, `while`, `do-while` or `switch` structure (we haven't yet discussed `switch`). You use `break` whenever you are inside a loop and want to get out now for some reason, and move on to execution of whatever follows the loop structure.

`continue` is used within looping structures to skip the rest of the current loop iteration and continue execution at the beginning of the next iteration. In other words, you `continue` when you want to abort the current iteration and go on to the next iteration immediately.

A pseudo-code example might elucidate. Suppose you are a manager interviewing a series of job applicants for a position:

```
while (there are more job applicants) {  
    // bring in the next applicant for an interview  
  
    if (this applicant is a total loser) {  
        continue; // get out of my office. next, please?  
    }  
  
    if (this applicant is a genius) {  
        break; // stop interviewing people, this person is hired  
    }  
  
    // if neither of above conditions is met, execution gets to this point.  
    // carry on interview with current applicant.  
}
```

Installation (or: Your Assignment for Week One)

You don't actually code anything for this assignment. Instead, you will set up and test your environment so you can *start* coding. (And do some optional supplemental reading). You are certainly encouraged to try running the examples found on the class site once your Apache/PHP system is set up.

1. **Install Apache, MySQL and PHP on your system** and get it working.

Minimum versions: **PHP 5.0.3**; **Apache 1.3.33**; **MySQL 4.1.x** is strongly preferred. MySQL 4.0.x will do if you have trouble finding a PHP package for your platform with support for *mysql* (don't worry yet about what that means), and can't or don't want to try building your own from source. There are two general approaches: (1) download the source and compile it yourself, or (2) download a pre-compiled (binary) package for your platform.

If you are on either **Windows** or **Linux**: we recommend a justly popular and user-friendly package called XAMPP, available from <http://www.apachefriends.org/en/xampp.html>.

If you are on a Linux/Unix type system, you may wish to build Apache and PHP from source and use a binary MySQL installation (from <http://dev.mysql.com/downloads/mysql/4.1.html>). If you have never done this before it might be an adventure, but it really isn't rocket science, and gives a satisfying fuzzy open source feeling when it succeeds. See <http://httpd.apache.org/> and <http://php.net/>, download your sources, read your INSTALL files and have at it. You will need root privileges to complete the installation.

If you are on **Mac OSX**, you can also build from source, but it is said to be more challenging for the uninitiated than it is on Linux. For good binary packages and documentation, try <http://www.entropy.ch/software/macosx/>. Your Mac is likely to have Apache, PHP and perhaps MySQL already installed, but the versions may also be too old.

Setting up Apache/MySQL/PHP (fondly known as AMP) does not have to be difficult, but it can be if you are not paying extreme attention. Whichever installation method you choose on whatever platform, take great care to RTFM and follow instructions slavishly.

2. Copy the code below (or get it from the class website) and paste it into a new file called week1.php; save week1.php in your web document root directory (if, for example, you have installed XAMPP on a Windows system, that will be something like C:\xampp\htdocs)

```
<?php if (isset($_GET['phpinfo'])) {
    ob_start();
    phpinfo();
    $data = ob_get_clean();
    $data = preg_replace(
        '|</body>|i',
        "<p style=\"text-align:center\"><a href=\"$_SERVER[PHP_SELF]
\>Hide php info</a></p></body>",
        $data
    );
    echo $data;
    exit;
}
echo '<?xml version="1.0" encoding="iso-8859-1"?>?'
<!DOCTYPE html
PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head><title>Week 1</title>
<meta http-equiv="Content-Type" content="text/html; charset=iso-8859-1" />
</head>
<body style="font-family:sans-serif;margin-left:100px;margin-right:100px;">
<h2>Hello World</h2>
<p>
Current date and time: <?php echo date('l, F d, Y h:i:s a'); ?>.
I am proudly running PHP version <?php echo PHP_VERSION ;?> with
<?php echo $_SERVER['SERVER_SOFTWARE'];?>
<p>For full details, <a href="<?php echo $_SERVER['PHP_SELF'] . '?phpinfo' ?
>">view the output of my phpinfo().</a></p>
</body></html>
```

Point your web browser at <http://localhost/week1.php>

Click the link to display the verbose installation/configuration information.

Save the resulting output to a file called week1.html, and turn it in.

3. **Choose a competent code editor** or IDE (Integrated Development Environment) with PHP support, install it on your machine, and spend a few minutes learning your way around with it. For example, try opening week1.php in the editor and notice the syntax highlighting. If you dislike it, try another one; repeat as needed until you're satisfied. A list of editors for various platforms with various

licenses can be found at <http://www.php-editors.com/>. We cannot emphasize enough the importance of having a code editor that at a very minimum supports syntax highlighting and code completion. You will also want to turn on line numbering right away if it isn't already enabled by default.

Reading

Any of the following:

- *Learning PHP 5* (Sklar): Chapters 1, 2, 3
- *Programming PHP* (Lerdorf & Tatroe): Chapters 1, 2
- *Core PHP Programming* (Atkinson): Chapters 1, 2, 3
- <http://php.net/manual/en/langref.php> sections 9 - 15

Last update 03-Feb-2005 10:47 am