PHP 101 (part 3) − Chocolate Fudge And Time Machines

By Vikram Vaswani and Harish Kamath

Table of Contents

Looping The Loop

If you've been paying attention, you'll remember that last time, we gave you a quick crash course in PHP's conditional statements and operators; we also showed you how PHP can be used to process the data entered into a Web form. This week, we're going to delve deeper into the topic of form processing by showing you how PHP handles form elements like lists and checkboxes, together with a demonstration of the other type of variable − the array variable.

First, though, we're going to toss you for a loop...

Back To The Future

For those of you unfamiliar with the term, a loop is a control structure which enables you to repeat the same set of PHP statements or commands over and over again; the actual number of repetitions may be dependent on a number you specify, or on the fulfillment of a certain condition or set of conditions.

The first − and simplest − loop to learn in PHP is the so−called "while" loop, which looks like this:

```
while (condition) { do this! }
```

or, in English,

```
while (it's raining) { carry an umbrella! }
```

In this case, so long as the condition specified evaluates as true − remember what you learned last time? − the PHP statements within the curly braces will continue to execute. As soon as the condition becomes false − the sun comes out, for example − the loop will be broken and the statements following it will be executed.

Here's a quick example which demonstrates the "while" loop:

```
<? // if form has not been submitted, display initial page if
(!$submit) { ?> <html> <head> </head> <body> <h2>The
Incredible Amazing Fantabulous Time Machine</h2> <form
action="tmachine.php4" method="POST"> Which year would you
like to visit? <input type="text" name="year" size="4"
maxlength="4"> <input type="submit" name="submit" value="Go">
</form> </body> </html> <? } else // else process it and
generate a new page { ?> <html> <head> </head> <body> <? //
current year $current = 2000; // check for dates in future and
generate appropriate message if ($year > $current) { echo
"<h2>Oops!</h2>"; echo "Sorry, this time machine can only
travel backwards at the moment. But leave your phone number
and we'll call you when the new improved model goes on sale.";
} else { // or echo "<b>Going back to $year...</b><br>"; //
use a while loop to print a series of numbers (years) // until
the desired number (year) is reached while($year < $current) {
$current = $current − 1; echo "$current "; } echo "<br><b>The
past definitely isn't all it's cracked up to be!</b>"; } ?>
</body> </html> <? } ?>
```

In this case, we first ask the user for the year he'd like to visit − this year is stored as the variable $year, and passed to the PHP script. The script first checks the year to ensure that it in the past [hey, we're working on it!] and then uses a "while" loop to count backwards from the current year − 2000, stored in the variable

$current − until the values of $current and $year are equal.

Note our usage of the $submit variable to use the same PHP page to generate both the initial form and the subsequent pages − we explained this technique in detail last time.

Revisiting The Past

So that's the "while" loop − it executes a set of statements until a specified condition is satisfied. But what happens if the condition is satisfied on the first iteration itself? In the illustration above, for example, if you were to enter the year 2000, the "while" loop would not execute even once. Try it yourself and you'll see what we mean.

So, if you're in a situation where you need to execute a set of statements *at least* once, PHP offers you the "do−while" loop. Here's what it looks like:

```
do { do this! } while (condition) Let's take a quick example:
<?php $bingo = 366; while ($bingo == 699) { echo "Bingo!";
break; } ?>
```

In this case, no matter how many times you run this PHP script, you will get no output at all, since the value of $bingo is not equal to 699. But, if you ran this version of the script

```
<?php $bingo = 799; do { echo "Bingo!"; break; } while ($bingo
== 699); ?>
```

you'd get one line of output.

Thus, the construction of the "do−while" loop is such that the statements within the loop are executed first, and the condition to be tested is checked after. This implies that the statements within the curly braces would be executed at least once.

Let's modify the time machine above to use a "do−while" loop instead:

```
<? // if form has not been submitted, display initial page if
(!$submit) { ?> <html> <head> </head> <body> <h2>The
Incredible Amazing Fantabulous Time Machine</h2> <form
action="tmachine.php4" method="POST"> Which year would you
like to visit? <input type="text" name="year" size="4"
maxlength="4"> <input type="submit" name="submit" value="Go">
</form> </body> </html> <? } else // else process it and
generate a new page { ?> <html> <head> </head> <body> <? //
current year $current = 2000; // check for dates in future and
generate appropriate message if ($year > $current) { echo
"<h2>Oops!</h2>"; echo "Sorry, this time machine can only
travel backwards at the moment. But leave your phone number
and we'll call you when the new improved model goes on sale.";
} else { // or echo "<b>Going back to $year...</b><br>"; // in
this case, the "do−while" loop // ensures that at least one
```

```
line of output is generated // even when you enter the year
2000 do { $current = $current − 1; echo "$current "; }
while($year < $current); echo "<br><b>The past definitely
isn't all it's cracked up to be!</b>"; } ?> </body> </html> <?
} ?>
```

Doing It By The Numbers

Both the "while" and "do−while" loops continue to iterate for so long as the specified conditional expression remains true. But there often arises a need to execute a certain set of statements a specific number of times − for example, printing a series of thirteen sequential numbers, or repeating a particular set of <TD>> cells five times. In such cases, clever programmers reach for the "for" loop...

The "for" loop typically looks like this:

```
for (initial value of counter; condition; update counter) { do
this! }
```

Looks like gibberish? Well, hang in there for a minute...the "counter" here is a PHP variable that is initialized to a numeric value, and keeps track of the number of times the loop is executed. Before each execution of the loop, the "condition" is tested − if it evaluates to true, the loop will execute once more and the counter will be appropriately incremented; if it evaluates to false, the loop will be broken and the lines following it will be executed instead.

And here's a simple example that demonstrates how this loop can be used:

```
<html> <head> <basefont face="Arial"> </head> <body>
<center>Turning The Tables, PHP−Style!</center> <br> <? //
define the number $number = 7; // use a for loop to calculate
tables for that number for ($x=1; $x<=15; $x++) { echo
"$number X $x = " . ($number*$x) . "<br>"; } ?> </body>
</html>
```

Let's dissect this a little bit:

The first thing we've done here is define the number to be used for the multiplication table; we've used 7, since we're particularly poor at math − you might prefer to use another number.

Next we've constructed a "for" loop with $x as the counter variable − we've initialized it to 1 and specified that the loop should run no more than 15 times.

The $x++ you see in the "for" statement is an interesting little operator known as the auto−increment operator − more on this in the "Miscellaneous Notes" section below. For the moment, all you need to know is that it automatically increments the counter by 1 every time the loop is executed.

Finally, the line that does all the work − the "echo" statement, which takes the number specified, multiplies it by the current value of the counter, and displays the result on the page.

As you can see, a "for" loop is a very interesting − and useful − programming construct. Our next example

illustrates its usefulness in a manner that should endear it to any HTML programmer.

```
<html> <head> <basefont face="Arial"> </head> <body>
<center>Turning The Tables, Part II</center> <br> <table
border=1> <? // the purpose of this exercise is // to generate
a 4x4 grid via an html table // this is accomplished via two
nested "for" loops // the first one is for the table rows or
<tr>s // we need four of them for ($alpha=1; $alpha<=4;
$alpha++) { ?> <tr> <? // the second one is for the cells in
each row or <td>s // we need four of them too for ($beta=1;
$beta<=4; $beta++) { ?> <td> <? // print the coordinate of
each cell echo("row $alpha, column $beta"); ?> </td> <? } ?>
</tr> <? } ?> </table> </body> </html>
```

And here's the output:

```
<html> <head> <basefont face="Arial"> </head> <body>
<center>Turning The Tables, Part II</center> <br> <table
border=1> <tr> <td> row 1, column 1 </td> <td> row 1, column 2
</td> <td> row 1, column 3 </td> <td> row 1, column 4 </td>
</tr> <tr> <td> row 2, column 1 </td> <td> row 2, column 2
</td> <td> row 2, column 3 </td> <td> row 2, column 4 </td>
</tr> <tr> <td> row 3, column 1 </td> <td> row 3, column 2
</td> <td> row 3, column 3 </td> <td> row 3, column 4 </td>
</tr> <tr> <td> row 4, column 1 </td> <td>s row 4, column 2
</td> <td> row 4, column 3 </td> <td> row 4, column 4 </td>
</tr> </table> </body> </html>
```

As you'll see if you try coding the same thing by hand, PHP's "for" loop just saved you a whole lot of work!

Anyone For Apple Pie?

PHP offers one more type of loop, the "foreach" loop, which is designed specifically for use with array variables. Logic would suggest that we explain array variables before we attempt to teach you the "foreach" loop...and you know what slaves we are to logic!

Thus far, the variables you've used contain only a single value − for example,

```
$i = 0
```

However, array variables are a different kettle of fish altogether. An array variable can best be thought of as a "container" variable, which can contain one or more values. For example,

```
$desserts = array("chocolate mousse", "tiramisu", "apple pie",
"chocolate fudge cake");
```

Here, $desserts is an array variable, which contains the values "chocolate mousse", "tiramisu", "apple pie", and "chocolate fudge cake".

Array variables are particularly useful for grouping related values together − names, dates, phone numbers of ex−girlfriends et al.

The various elements of the array are accessed via an index number, with the first element starting at zero. So, to access the element

```
"chocolate mousse"
```

you would use the notation

```
$desserts[0]
```

while

```
"chocolate fudge cake"
```

would be

```
$desserts[3]
```

− essentially, the array variable name followed by the index number enclosed within square braces. Geeks refer to this as "zero−based indexing".

Defining an array variable is simple via the array() function − here's how:

```
$past_flames = array("Jennifer", "Susan", "Tina", "Bozo The
Clown");
```

The rules for choosing an array variable name are the same as those for any other PHP variable − it must begin with a letter, and can optionally be followed by more letters and numbers.

Alternatively, you can define an array by specifying values for each element in the index notation, like this:

```
$past_flames[0] = "Jennifer"; $past_flames[1] = "Susan";
$past_flames[2] = "Tina"; $past_flames[3] = "Jennifer";
```

The Generation Gap

You can add elements to the array in a similar manner − for example, if you wanted to add the element "apricot fritters" to the $desserts array, you would use this:

```
$desserts[4] = "apricot fritters";
```

and the array would now look like this:

```
$desserts = array("chocolate mousse", "tiramisu", "apple pie",
"chocolate fudge cake", "apricot fritters");
```

In order to modify an element of an array, you would simply assign a new value to the corresponding scalar variable. If you wanted to replace "chocolate fudge cake" with "chocolate chip cookies", you'd simply use

```
$desserts[3] = "chocolate chip cookies";
```

and the array would now read

```
$desserts = array("chocolate mousse", "tiramisu", "apple pie",
"chocolate chip cookies", "apricot fritters");
```

PHP arrays can store both string and numeric data − in fact, the same array can store both types of data, a luxury not available in many other programming languages.

How about a quick example?

```
<!doctype html public "−//W3C//DTD HTML 4.0 Transitional//EN">
<html> <head> <basefont face="Arial"> </head> <body> <? //
define some array variables $then = array("The Beatles", "Roy
Orbison", "Frank Sinatra"); $now = array("Britney Spears",
"N−Sync", "Jennifer Lopez", "Blink 182"); ?> While Mom and Dad
listen to <? // use loop to extract and display array elements
for ($x=0; $x<sizeof($then); $x++) { echo $then[$x] . ", "; }
?> in the living room, I'm grooving to <? for ($y=0;
$y<sizeof($now); $y++) { echo $now[$y] . ", "; } ?> in the
basement! </body> </html>
```

And here's what you'll see:

```
While Mom and Dad listen to The Beatles, Roy Orbison, Frank
Sinatra, in the living room, I'm grooving to Britney Spears,
N−Sync, Jennifer Lopez, Blink 182, in the basement!
```

In this case, we've first defined two arrays, and then used the "for" loop to run through each one, extract the elements using the index notation, and display them one after the other. Note our usage of the sizeof() function − this function is typically used to return the size or length of an array variable, and is used here to ensure that the loop iterates as many times as there are elements in each array.

What's That Noise?

Of course, there is a simpler way of extracting all the elements of an array − PHP4 has a "foreach" loop designed specifically for this purpose and similar in syntax to the Perl construct of the same name. Here's what it looks like:

```
foreach ($array as $temp) { do this! }
```

or, in English, "take each element of the array variable $array, place it in the variable $temp, and execute the set of statements within the curly braces on $temp"

Let's demonstrate this by re−writing the example above in terms of the "foreach" loop:

```
<html> <head> <basefont face="Arial"> </head> <body> <? //
define some array variables $then = array("The Beatles", "Roy
Orbison", "Frank Sinatra"); $now = array("Britney Spears",
"N−Sync", "Jennifer Lopez", "Blink 182"); ?> While Mom and Dad
listen to <? // use loop to extract and display array elements
foreach ($then as $artist) { echo $artist . ", "; } ?> in the
living room, I'm grooving to <? foreach ($now as $artist) {
echo $artist . ", "; } ?> in the basement! </body> </html>
```

Checking The Boxes

In addition to their obvious uses, arrays and loops also come in handy when processing forms in PHP. For example,

Here's a simple example which demonstrates how arrays can be used when processing checkboxes in an HTML form:

```
<html> <head> </head> <body> <? // check for $submit if
(!$submit) { // and display form ?> <form
action="jukebox.php4" method="GET"> <input type="checkbox"
name="artist[]" value="Bon Jovi">Bon Jovi <input
type="checkbox" name="artist[]" value="N'Sync">N'Sync <input
type="checkbox" name="artist[]" value="Boyzone">Boyzone <input
type="checkbox" name="artist[]" value="Britney Spears">Britney
Spears <input type="checkbox" name="artist[]" value="Jethro
Tull">Jethro Tull <input type="checkbox" name="artist[]"
value="Crosby, Stills &Nash">Crosby, Stills &Nash <br> <input
type="submit" name="submit" value="Select"> </form> <? } // or
display the selected artists else { ?> <b>And here are your
selections:</b> <br> <? // use a "for" loop to read and
display array elements for($count = 0; $count <
sizeof($artist); $count++) { echo
"<i>$artist[$count]</i><br>"; } } ?> </body> </html>
```

In this case, when the form is submitted, PHP will automatically create an array variable named $artist, and populate it with the items selected. This is useful when you need to group related form data together.

The same technique can be applied to "multiple select" list boxes:

```
<html> <head> </head> <body> <? // check for $submit if
(!$submit) { // and display form ?> <form
action="jukebox.php4" method="POST"> <select name="artist[]"
size="6" multiple> <option value="Bon Jovi">Bon Jovi</option>
<option value="N'Sync">N'Sync</option> <option
value="Boyzone">Boyzone</option> <option value="Britney
Spears">Britney Spears</option> <option value="Jethro
Tull">Jethro Tull</option> <option value="Crosby, Stills
&Nash">Crosby, Stills &Nash</option> </select> <br> <input
type="submit" name="submit" value="Select"> </form> <? } // or
display the selected artists else { ?> <b>And here are your
selections:</b> <br> <? // use a "for" loop to read and
display array elements for($count = 0; $count <
sizeof($artist); $count++) { echo
"<i>$artist[$count]</i><br>"; } } ?> </body> </html>
```

Miscellaneous Notes

The auto−increment and auto−decrement operators
−−

In our explanation of the "for" loop above, we mentioned the auto−increment operator [++] in passing. The auto−increment operator is a PHP operator designed to automatically increment the value of the variable it is attached to by 1.

This snippet of code should explain it.

```
<? // define $mercury as 10 $mercury = 10; // increment it
$mercury++; // $mercury is now 11 echo $mercury; ?> Thus,
$mercury++; is functionally equivalent to $mercury = $mercury
+ 1;
```

And there's also a corresponding auto−decrement operator [−−], which does exactly the opposite:

```
<? // define $mercury as 10 $mercury = 10; // decrement it
$mercury−−; // $mercury is now 9 echo $mercury; ?>
```

These operators are frequently used in loops to automatically update the value of the loop counter.

break and continue
−−−−−−−−−−−−−−−−−−−−−−−−

When dealing with loops, there are two important keywords you should be aware of: "break" and "continue".

The "break" keyword is used to exit a loop when it encounters an unexpected situation. A good example of this is the dreaded "division by zero" error − when dividing one number by another one[which keeps decreasing], it is advisable to check the divisor and use the "break" statement to exit the loop as soon as it becomes equal to zero.

The "continue" keyword is used to skip a particular iteration of the loop and move to the next iteration immediately − it's demonstrated in the following example:

```
<? for ($x=1; $x<=10; $x++) { if ($x == 7) { continue; } else
{ echo "$x <br>"; } } ?>
```

In this case, PHP will print a string of numbers from 1 to 10 − however, when it hits 7, the "continue" statement will cause it to skip that particular iteration and go back to the top of the loop. So your string of numbers will not include 7 − try it and see for yourself.

And that's all we have time for this week. We'll be back soon with more PHP 101, same time, same channel. Make sure you tune in!