PHP 101 (part 3): Looping The Loop

Going Deeper

If you've been paying attention, you remember that in Part Two I gave you a quick crash course in PHP's basic control structures and operators. I also showed you how PHP can be used to process the data entered into a Web form. In this tutorial, I'm going to delve deeper into PHP's operators and control structures, showing you two new operators, an alternative to the `if-else()` family of conditional statements, and some of PHP's more interesting loops. So keep reading… this is just about to get interesting!

Switching Things Around

An alternative to the `if-else()` family of control structures is PHP's `switch-case()` statement, which does almost the same thing. It looks like this:

```
switch (decision-variable) {
   case first condition is true:
      do this!
      case second condition is true:
      do this!
     ... and so on...
}
```

Depending on the value of the decision variable, the appropriate `case()` block is executed. A `default` block can also be created, to handle all those occasions when the value of the decision variable does not match any of the listed `case()` conditions.

I'll make this a little clearer by re-writing one of my earlier examples in terms of the `switch()` statement:

```
<html>
<head></head>
<body>
<?php
// get form selection
$day = $_GET['day'];
// check value and select appropriate item
switch ($day) {
    case 1:
        $special = 'Chicken in oyster sauce';
        break;
    case 2:
        $special = 'French onion soup';
        break;
    case 3:
        $special = 'Pork chops with mashed potatoes and green salad';
        break;
    default:
        $special = 'Fish and chips';
            break;
}
?>
<h2>Today's special is:</h2>
<?php echo $special ?>
</body>
</html>
```

There are a couple of important keywords here:

•The `break` keyword is used to break out of the `switch()` statement block and move immediately to the lines following it.

•The `default` keyword is used to execute a default set of statements when the variable passed to `switch()` does not satisfy any of the conditions listed within the block.

A common newbie mistake here is to forget the `break` at the end of every `case()` block. Remember that if you forget to break out of a `case()` block, PHP will continue executing the code in all the subsequent `case()` blocks it encounters.

For more on the `switch()` statement, see http://www.php.net/manual/en/control-structures.switch.php.

Creative Conditionals

Normally, when creating and processing forms in PHP, you would place the HTML form in one file, and handle form processing through a separate PHP script. However, with the power of conditional statements at your disposal, you can combine both pages into one (俱體怎麼弄, 不重要, 故沒劃).

How do you do this? Simple. All you need to do is assign a name to the form submit control, and then check whether the special $_POST container variable contains that name when the script first loads up. If it does, the form has already been submitted, and you can process the data; if it does not, that the user has not submitted the form and you therefore need to generate the initial, unfilled form. Thus, by testing for the presence or absence of this submit variable, a clever PHP programmer can use a single PHP script to generate both the initial form, and the output after it has been submitted, as appropriate.

Here's a simple example:

```
<html>
<head></head>
<body>
<?php
/* if the "submit" variable does not exist, the form has not been submitted -
display initial page */
if (!isset($_POST['submit'])) {
?>
    <form action="<?php echo $_SERVER['PHP_SELF']; ?>" method="post">
    Enter your age: <input name="age" size="2">

    <input type="submit" name="submit" value="Go">

        </form>

<?php

    }

else {

/* if the "submit" variable exists, the form has been submitted - look for and

process form data */

    // display result

     $age = $_POST['age'];

        if ($age >= 21) {

        echo 'Come on in, we have alcohol and music awaiting you!';

        }

    else {

        echo 'You're too young for this club, come back when you're a little older';

        }

}

?>

</body>

</html>
```

As you can see, the script contains two pages: the initial, empty

form and the result page generated after hitting the `submit` button. In

order to decide which page to display, the script first tests for the

presence of the `$_POST['submit']` variable. If it doesn’t find it, it assumes that the form has yet to be submitted, and displays the initial list of days. Once the form has been submitted, the same script will be called to process the form input. This time, however, the `$_POST['submit']` variable *will* be set, and so PHP will not display the initial page, but rather the page containing the result message.

Note that for this to work, your `submit` button must have a value assigned to its "name" attribute, and you must check for that value in the primary conditional statement. And in case you were wondering, the `$_SERVER` array is a special PHP variable which always holds server information, including the path and name of the currently executing script.

Next up, loops.

One by One

For those of you unfamiliar with the term, a loop is a control structure that enables you to repeat the same set of php statements or commands over and over again (the actual number of repetitions can be a number you specify, or depend on the fulfillment of one or more conditions).

Now, last time out you saw a few comparison and logical operators, which help in building conditional statements. Since this segment of the tutorial is going to focus on loops, this is an appropriate time to introduce you to PHP's auto-increment and auto-decrement operators, which see a lot of use in this context.

The auto-increment operator is a PHP operator designed to automatically increment the value of the variable it is attached to by

1. It is represented by two “plus” signs (++). This snippet of code should explain it:

```
<?php
// define $total as 10
$total = 10;
// increment it
$total++;
// $total is now 11
echo $total;
?>
```

Thus, `$total++` is functionally equivalent to `$total = $total + 1`.

There’s a corresponding auto-decrement operator (–), which does exactly the opposite:

```
<?php
// define $total as 10
$total = 10;
// decrement it
$total--;
// $total is now 9
echo $total;
?>
```

These operators are frequently used in loops, to update the value of the loop counter, speaking of which…

Being Square

The first – and simplest – loop to

learn in PHP is the so-called `while()` loop, which looks like this:

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

In this case, so long as the condition specified evaluates as true
– remember what you learned in Part Two?
– the PHP statements within the curly braces will continue to execute. As soon as the condition becomes false, the loop will be broken and the statements following it will be executed.

Here's a quick example which demonstrates the `while()` loop:

```
<html>
<head></head>
<body>
<form action="squares.php" method="POST">
Print all the squares between 1 and <input type="text" name="limit" size="4"
maxlength="4">
<input type="submit" name="submit" value="Go">
</form>
```

```
</body>

</html>
```

This is a simple form which asks the user to enter a number. When the form is submitted, the PHP script that is invoked should take this number and print the squares of all the numbers between 1 and the entered value. With a `while()` loop, this is simplicity itself:

```
<html>
<head></head>
<body>
<?php
// set variables from form input
$upperLimit = $_POST['limit'];
$lowerLimit = 1;
// keep printing squares until lower limit = upper limit
while ($lowerLimit <= $upperLimit) {
    echo ($lowerLimit * $lowerLimit).'&nbsp;';
        $lowerLimit++;
}
// print end marker
echo 'END';
?>
</body>
</html>
```

This script uses a `while()` loop to count forwards from 1 until the values of `$lowerLimit` and `$upperLimit` are equal.

Loop First, Ask Questions Later

The `while()` loop executes a set of statements while a specified condition is true. But what happens if the condition is true on the first iteration of the loop itself? In the previous example, if you were to enter the value 0in the form, the `while()` loop would not execute even once. Try it yourself and you’ll see what I mean.

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 is true)
```

Let’s take a quick example to better understand the difference between `while()` and `do-while()`:

```
<?php
$x = 100;
// while loop
while ($x == 700) {
        echo "Running...";
    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 `$x` is not equal to 700.

But, if you ran this version of the script:

```
<?php
$x = 100;
// do-while loop
do {
    echo "Running...";
        break;
} while ($x == 700);
?>
```

you would see *one* line of output, as the code within the `do()`

block would run once.

Let’s now revise the previous PHP script so that it runs at least

once, regardless of what value is entered into the form:

```
<html>
<head></head>
<body>
```

```
<?php
// set variables from form input
$upperLimit = $_POST['limit'];
$lowerLimit = 1;
// keep printing squares until lower limit = upper limit
do {
    echo ($lowerLimit * $lowerLimit).'&nbsp;';
    $lowerLimit++;
} while ($lowerLimit <= $upperLimit);
// print end marker
echo ' END';
?>
</body>
</html>
```

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 afterwards. This implies that the statements within the curly braces would be executed at least once.

Read more about the `while()` and `do-while()` loops at http://www.php.net/manual/en/control-structures.while.php and http://www.php.net/manual/en/control-structures.do.while.php.

Doing it by Numbers

Both the `while()` and `do-while()` loops continue to iterate for as long as the specified conditional expression remains true. But what if you 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; new value of 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.

Here’s a simple example that demonstrates how this loop can be used:

```
<html>

<head>

<basefont face="Arial">

</head>
```

```
<body>

<?php

// define the number

$number = 13;

// use a for loop to calculate tables for that number

for ($x = 1; $x <= 10; $x++) {

        echo "$number x $x = ".($number * $x)."<br />";

}

?>

</body>

</html>
```

The first thing I’ve done here is define the number to be used for the multiplication table. I’ve used 13 here – for no reason other than that it rhymes with “green”.

Next, I’ve constructed a `for()` loop with `$x` as the counter variable, initialized it to 1. and specified that the loop should run no more than 10 times. The auto-increment operator (discussed earlier) automatically increments the counter by 1 every time the loop is executed. Within the loop, the counter is multiplied by the number, to create the multiplication table, and `echo()` is used to display the result on the page.

Turning the Tables

As you just saw, a `for()` loop is a very interesting – and useful – programming construct. The next example illustrates its usefulness in a manner that should endear it to any HTML programmer.

```
<html>
<head></head>
<body>
<form method="post" action="<?php echo $_SERVER['PHP_SELF']; ?>">
Enter number of rows <input name="rows" type="text" size="4"> and columns
<input name="columns" type="text" size="4"> <input type="submit"
name="submit" value="Draw Table">
</form>
<?php
if (isset($_POST['submit'])) {
    echo "<table width = 90% border = '1' cellspacing = '5' cellpadding = '0'>";
    // set variables from form input
    $rows = $_POST['rows'];
    $columns = $_POST['columns'];
    // loop to create rows
    for ($r = 1; $r <= $rows; $r++) {
        echo "<tr>";
        // loop to create columns
        for ($c = 1; $c <= $columns;$c++) {
            echo "<td>&nbsp;</td>
";
```

```
            }
    echo "</tr>
";
    }
    echo "</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! And it looks good too – take a look at the

source code of the dynamically generated table, and you'll see that

it's nicely formatted, with line breaks at the end of every table cell

and row. This magic is accomplished by forcing a carriage return

with

in every call to `echo()`.

For more examples of the `for()` loop in action, visit

http://www.php.net/manual/en/control-structures.for.php.

Loops are frequently used in combination with one of PHP's more

complex data types, the animal known as the array. That's a whole topic in itself, and in fact

I'm going to discuss it in detail in the next segment of this tutorial.

Then I'm going to show you how arrays, loops and forms all work together to make

the creation of complex Web forms as easy as eating pie. All that and more in

Part Four!