5. PHP Loops

Often we may want certain code that satisfies a particular condition to be repeated again and again. Instead of adding several almost equal lines in a script we can use loops to perform a task like this.

In PHP, we have the following looping statements:

- **while** - loops through a block of code while a specified condition is true
- **do...while** - loops through a block of code once, and then repeats the loop as long as a specified condition is true
- **for** - loops through a block of code a specified number of times
- **foreach** - loops through a block of code for each element in an array

Each time the code in the loop executes, it is called an iteration. It's useful for many common tasks such as displaying the results of a query by looping through the returned rows. Each of the loop constructs requires three basic pieces of information. First initialization of loop variable is done. Then secondly, when to stop looping based on loop condition is defined just like the comparison in an `if` statement. Third, the loop variable is incremented or decremented in order to make the loop fails so it comes out and execute the statement next to end of the loop. Within the loop the code to perform is also required and specified either on a single line or within curly braces.

5.1. while Loops

The `while` loop takes the expression followed by the code to execute.

The syntax is for a `while` loop is:

```
while (expression)
{
  code to
 execute;
}
```

An example is shown in **Example 5.1.**

Example 5-1. A sample while loop that counts to 10

```
<?php
$num = 1;

while ($num <= 10){
    print "Number is $num<br />\n";
    $num++;
}
print 'Done.';
?>
```

Example 5.1 produces:

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
Number is 8
Number is 9
Number is 10
Done.
```

Before the loop begins, the variable `$num` is set to 1. This is called initializing a counter variable. Each time the code block executes, it increases the value in `$num` by 1 with the statement `$num++;`. After 10 iterations, the evaluation `$num <= 10` becomes `FALSE` and the loop stops, which then prints `Done`..

5.2. do . . . while Loops

The `do . . . while` loop takes an expression such as a `while` statement but places it at the end. The syntax is:

```
do
 {
  code to execute;
} (expression);
```

This loop is useful when you want to execute the block of code once regardless of the value in the expression. For example, let's count to 10 with this loop, as in **Example 5.2**.

Example 5-2. Counting to 10 with do...while

```
<?php

$num = 1;

do {
    echo "Number  is ".$num."<br />";
    $num++;
} while ($num <= 10);

echo "Done.";

?>
```

Example 5.2 produces the same results as **Example 5-1**; if you change the value of `$num` to 11, the loop processes differently.

```
<?php

$num = 11;

do {
    echo $num;
    $num++;
} while ($num <= 10);

?>
```

This produces:

```
11
```

The code in the loop displays 11 because the loop always executes at least once. Following the pass, `while` evaluates to `FALSE`, causing execution to drop out of the `do . . . while` loop.

5.3. for Loops

`for` loops provide the same general functionality as `while` loops, but also provide for a predefined location for initializing and changing a counter value. Their syntax is:

```
for (initialization expression; test expression; modification expression){
  code that is executed;
}
```

An example `for` loop is:

Example 5-3. Counting to 10 with do...while

```
<?php
for ($num = 1; $num <= 10; $num++) {
    print "Number is $num<br />\n";
}
?>
```

This produces:

```
Number is 1
Number is 2
Number is 3
Number is 4
Number is 5
Number is 6
Number is 7
```

```
Number is 8
Number is 9
Number is 10
```

When your PHP program process the `for` loop, the initialization portion is evaluated. For each iteration of the portion that increments, the counter executes, followed by a check to see whether you're done. The result is a much more compact and easy-to-read statement.

5.4 Breaking Out of a Loop

PHP provides the equivalent of an emergency stop button for a loop: the `break` statement. Normally, the only way out of a loop is to satisfy the expression that determines when to stop the loop. If the code in the loop finds an error that makes continuing the loop pointless or impossible, you can break out of the loop by using the `break` statement. It's like getting your shoelace stuck in an escalator. It really doesn't make any sense for the escalator to keep going!

Possible problems you might encounter in a loop include running out of space when writing to a file or attempting to divide by zero. In **Example 5.4**, we simulate what can happen if you divide based on an unknown entry initialized from a form submission (that could be a user-supplied value). If your user is malicious or just plain careless, she might enter a negative value where you are expecting a positive value (although this should be caught in your form validation process). In the code that is executed as part of the loop, the code checks to make sure `$counter` is not equal to zero. If it is, the code calls `break`.

Example 5.4. Using break to avoid division by zero

```
<?php

$counter = -3;

for (; $counter < 10; $counter++){
    // Check for division by zero
    if ($counter == 0){
        echo "Stopping to avoid division by zero.";
        break;
    }

    echo "100/$counter<br />";
}

?>
```

This displays:

```
100/-3
100/-2
```

```
100/-1
Stopping to avoid division by zero.
```

Of course, there may be times when you don't want to just skip one execution of the loop code. The `continue` statement performs this for you.

5.5 continue Statements

You can use the `continue` statement to stop processing the current block of code in a loop and jump to the next iteration of the loop. It's different from `break;` in that it doesn't stop processing the loop entirely. You're basically skipping ahead to the next iteration. Make sure you are modifying your test variable before the `continue` statement, or an infinite loop is possible.

Example 5-5 shows the preceding example using `continue` instead of `break`.

Example 5-5. Using continue instead of break

```
<?php

$counter =- 3;

for (; $counter < 10; $counter++){
    // Check for division by zero
    if ($counter == 0){
        echo "Skipping to avoid division by zero.<br />";
        continue;
    }

    echo "100/$counter<br />";
}

?>
```

Example 5-5 displays:

```
100/-3
100/-2
100/-1
Skipping to avoid division by zero.
100/1
100/2
100/3
100/4
100/5
100/6
100/7
100/8
100/9
```

Notice that the loop skipped over the `$counter` value of zero but continued with the next value.

We've now covered all of the major program flow language constructs. We've discussed the building blocks for controlling program flow in your programs. Expressions can be as simple as `TRUE` or `FALSE` and as complex as relational comparison with logical operators. The expressions combined with program flow control constructs like the `if` statement and `switch` make decision making easy.

We also discussed `while`, `while . . . do`, and `for` loops. Loops are very useful for common dynamic web page tasks such as displaying the results from a query in an HTML table.

The foreach Loop

The foreach loop is used to loop through arrays.

Syntax

```
foreach ($array as $value)
  {
  code to be executed;
  }
```

For every loop iteration, the value of the current array element is assigned to $value (and the array pointer is moved by one) - so on the next loop iteration, you'll be looking at the next array value.

Example

The following **Example 5.6** demonstrates a loop that will print the values of the given array:

```
<html>
<body>

<?php
$x=array("one","two","three");
foreach ($x as $value)
  {
  echo $value . "<br>";
  }
?>

</body>
</html>
```

Output:

```
one
two
three
```