

3.9: Structures

Structures

Referred to as selection, control, or loop structures, this set of elements dictate conditions under which certain sections code are executed. They allow our program to be flexible, instead of restricting it to the same actions every iteration. Here, we will consider all of them together, as structures in general.

If

“If” is perhaps the simplest test we can apply in logic. In programming, the “then” side is implied in the code contained in the definition of our statement—If [some condition] is true, then do something. We can use this to put exceptions into our code, or to redirect to a different action. Within the If statement we can perform simple comparisons or have the interpreter perform calculations and get returns from functions as part of its comparison task.

To run some examples, we will check if a variable called coffee is equal to hot. If it is, we want to run the drink function.

```
1. if("hot" == $coffee){  
2. drink($coffee);  
3. }
```

Additional notes

Remember that = is an assignment operation! == or === must be used when testing if both sides of the operand are equivalent.

Else

Next we will assume our drink() function tells us if we want more or feel full, prompting us to take new actions, like cleaning up after ourselves. This means we need to take one of two actions, which can do by extending our If/Then by adding Else:

```
1. if(drink($coffee)=='full'){  
2. cleanUp();  
3. }  
4. else{ //We want more!  
5. drink($coffee);  
6. }
```

elseif

We can use elseif when we want to follow up a failed If statement with another check. While the same affect can be created by placing an If inside of our Else, we can eliminate a layer of nesting. We can still follow up an elseif with its own Else as well:

```
1. if ($a > $b) {  
2. echo "a is bigger than b";  
3. } elseif ($a == $b) {  
4. echo "a is equal to b";  
5. } else {  
6. echo "a is smaller than b";  
7. }
```

While

While statements will repeatedly execute until the condition we provide is no longer true. The code will start over, in a loop, as long as the requirement is met.

Be careful with While statements! You can easily create infinite loops using While, in which your code will continue to run until forced to stop by closing the page or resetting your web service. The first method of prevention is to ensure that you have at least one place in your While loop that can affect the value (or values) in your while condition. The second method of prevention is to make sure that at least one result of those changes satisfies your While condition.

While loops are very similar to If in terms of structure. Here we will pretend \$ourValue is a 4:

```
1. $ourValue=4;
2. while($ourValue!=6){
3. echo "Wait for it...<br/>";
4. $ourValue++;
5. }
6. echo "6!";
```

```
1. Wait for it...
2. Wait for it...
3. 6!
```

Let us try one that counts for us:

```
1. $ourValue=4;
2. while($ourValue!=10){
3. echo "$ourValue," ;
4. $ourValue++;
5. }
```

- 4, 5, 6, 7, 8, 9,

To see a broken While statement in action, change \$ourValue to 11 and refresh your page!

Do While

Do While loops are very similar to While loops with the difference that every Do While loop will run at least one time. This is because a Do While loop evaluates the equation after each loop, not before. Since it does not check if conditions are met until the end, the first loop will never be checked. For example, compare the following loops and their output:

```
1. do {
2. echo $i;
3. } while ($i > 0);
4. echo "<br/> done";
```

- 0
- done

```
1. while ($i>0){
2. echo $i;
3. }
4. echo "<br/> done";
```

- done

Learn more

Keywords, search terms: Control structures, logic

Alternative syntax: <http://www.brian2000.com/php/understanding-alternative-syntax-for-control-structures-in-php/>

More examples: <http://www.informit.com/articles/article.aspx?p=30092&seqNum=2>

For

For loops are very similar to While loops, but they execute code only while the condition is within the given range. When using a For loop, we need to provide it with the value we want to monitor, the condition we want to stop on, and how we want to change it, right in the declaration. As a result, the body of the loop is strictly what we want to happen while the For's conditions are true. Let us say we want to watch \$x (we will pretend it is 5 right now). We want to keep running until it is greater than 10, and we want to add 2 each time we finish the loop:

```
1. $x=5;
2. for($x;$x<10;$x+=2){
3. echo "$x is less than 10 <br/>";
4. }
5. echo "$x is greater than 10!";
```

```
1. 5 is less than 10
2. 7 is less than 10
3. 9 is less than 10
4. 11 is greater than 10!
```

Unlike the While loop, you will notice we did not have to change the value of `$x` inside our loop, it was done automatically each time the end of the loop was reached. If the termination condition (in our example `x` would be greater or equal to 10) is not met, the loop starts again. Let us look at another example. This time, if a secondary condition that we test for occurs, we will force the For to stop by changing `$x` ourselves to something that satisfies the termination condition:

<pre>1. \$x=5; 2. for(\$x;\$x<10;\$x+=2){ 3. echo "\$x is less than 10
"; 4. if(\$x==7)\$x=11; 5. } 6. echo "\$x is greater than 10!";</pre>	<pre>1. 5 is less than 10 2. 7 is less than 10 3. 11 is greater than 10!</pre>
--	--

Foreach

Although we took a brief look at Foreach when we reviewed [pseudo-code](#), we will take a look at an actual implementation now. This loop format is used to iterate over arrays, allowing us an opportunity to manipulate each element inside. Since it is designed specifically for interacting with arrays, these are the only objects you can pass to it. The function takes the array we want to look at as well as what we want to call each element as we look at it. Alternatively, we can ask that the function take each element as a key and value pair, which can be very useful when position value is needed or associative arrays are in play.

When we use a Foreach, the engine makes a copy of the array, leaving the original intact, eliminating the need to track and reset the original array's pointer or reset the array when we are done. This also means that while we are interacting with the elements inside our array, we need to keep in mind that we are dealing with a copy. No changes that are applied to our copy will persist unless we call the function while applying an ampersand (&) before the variable name, which instructs the function to apply changes directly to our original array.

In order to apply changes to our array, the original that is passed must be a variable already stored in memory, and not an array declared in the function call. Let us look at some examples to clear some of this up by mimicking `print_r()`.

<pre>1. \$array = array(1, 2, 3, 4, 5); 2. foreach(\$array as \$number){ 3. echo "Our value is \$number
"; 4. }</pre>	<pre>1. Our value is 1 2. Our value is 2 3. Our value is 3 4. Our value is 4 5. Our value is 5</pre>
---	--

To play with an associative array and see the key and the value, we will adjust both our starting array and what we pass in the function call:

<pre>1. \$array = array("Mike"=>42, "Frank"=>38, "Anne"=>28); 2. foreach(\$array as \$key=>\$value){ 3. echo "\$key is \$value years old.
"; 4. }</pre>	<pre>1. Mike is 42 years old. 2. Frank is 38 years old. 3. Anne is 28 years old.</pre>
--	--

Finally, we will take a look at applying our changes back to the array, multiplying our original array's values by 2:

<pre>1. \$array = array(1, 2, 3, 4, 5); 2. foreach(\$array as &\$number){ 3. \$number = \$number * 2; 4. } 5. print_r(\$array);</pre>	<pre>1. Array(2. 0 => 2 3. 1 => 4 4. 2 => 6 5. 3 => 8 6. 4 => 10 7.)</pre>
---	---

Switch

A switch statement lets us run several tests to determine the proper course of action, or to apply one or more changes to our variable or elsewhere in our code, when the condition of our case is met. Some indicators that a switch is appropriate are when you find yourself running several If Then statements on the same variable, or have nested multiple logic statements attempting to control the action of your script.

To create an example, let us write a switch that gives information about a number passed to it. First, we will design it to determine the smallest positive value of 2 through 9 that \$value is a multiple of, and then we will tweak it a bit so it can tell us all the values of 2 through 9 \$value is a multiple of. If we just want the smallest value, we only need to test 2, 3, 5, and 7 since 4, 6, 8 and 9 are multiples of these numbers anyway, so they could not be the smallest. Now, if the number we check is divisible by one of these numbers, it would not have a remainder. To check specifically for a remainder, we can use modular division, which in PHP is represented by a %, and returns a 0 or 1. If we get a zero, there is no remainder. So let us create a switch with empty test cases for 2, 3, 5 and 7:

```
1. switch($value){  
2. case ($value % 2 == 0):  
3. case ($value % 3 == 0):  
4. case ($value % 5 == 0):  
5. case ($value % 7 == 0):  
6. default:  
7. }
```

Each case can be followed by a set of parenthesis denoting our logical test or a value, which is followed by a colon that denotes the start of what we want to happen when the case is true. If we wanted a case where the value IS 2, we would use:

```
1. Case 2:
```

Or, if we wanted to test if the value is the WORD two, we would use:

```
1. Case "two":
```

By default, each case will be tested until the switch is complete, or we tell it we want it to stop. This ability will be useful in a moment, but for now, we want the switch to stop as soon as we find a value, since we are testing in smallest to largest order, and only need one test to be true. To do this, we put a “break;” wherever we want the execution of our switch to stop. Typically this is the last line before the following case, but if the case we are in has additional logic tests we might have more than one “break;” depending on the extra conditions we are testing.

You will also notice the “default:” that snuck in at the bottom. The default case must be last, but is optional, and gives us a “catch all” action in the event that none of our cases were met. We do not need a “break;” after our default, since it will always be the last case in our switch.

To complete our example, we will want to let the user know what the smallest value we found was, so we need to fill out our code:

```
1. switch($value){  
2. case ($value % 2 == 0):  
3. echo "$value is divisible by 2 <br/>";  
4. break;  
5. case ($value % 3 == 0):  
6. echo "$value is divisible by 3 <br/>";  
7. break;  
8. case ($value % 5 == 0):  
9. echo "$value is divisible by 5 <br/>";  
10. break;  
11. case ($value % 7 == 0):  
12. echo "$value is divisible by 7 <br/>";  
13. break;  
14. default:  
15. echo "$value is not divisible by 2 through 9.";
```

16. }

Additional notes

This does not mean our switch will find all prime numbers! Prime numbers have to be tested against the range 2 to $n/2$ to ensure there are no dividends.

With this example, if \$value was 4, we would get “4 is divisible by 2.” If \$value was 12, we would get “12 is divisible by 2” but would not get a response for 3, 4, or 6 since we included breaks after each test, and it stopped after 2. If \$value was 11, we would get all the way to “11 is not divisible by 2 through 9.” In this scenario, it is because 11 is a prime number, which by definition is only divisible by itself and 1.

Now let us tweak our switch statement so it tells us all of the values between 2 and 9 that can divide our number without a remainder. First, we will have to start testing the values we could skip earlier. For example, 8 is not divisible by 6 even though both are divisible by 2. Second, we no longer want to stop after one expression is true. To do this, we will get rid of all of our breaks except for the one in the case preceding the default, ensuring that if any of the cases were true, we will not still see our default statement. That gives us:

```
1. switch($value){
2. case ($value % 2 == 0 ):
3. echo "$value is divisible by 2 <br/>";
4. case ($value % 3 == 0 ):
5. echo "$value is divisible by 3 <br/>";
6. case ($value % 4 == 0 ):
7. echo "$value is divisible by 4 <br/>";
8. case ($value % 5 == 0 ):
9. echo "$value is divisible by 5 <br/>";
10. case ($value % 6 == 0 ):
11. echo "$value is divisible by 6 <br/>";
12. case ($value % 7 == 0 ):
13. echo "$value is divisible by 7 <br/>";
14. case ($value % 8 == 0 ):
15. echo "$value is divisible by 8 <br/>";
16. case ($value % 9 == 0 ):
17. echo "$value is divisible by 9 <br/>";
18. break;
19. default:
20. echo "$value is not divisible by 2 through 9.";
21. }
```

To repeat our examples using 4, 12, and 11, respectfully we would see the following responses:

1. 4 is divisible by 2
2. 4 is divisible by 4
3. 12 is divisible by 2
4. 12 is divisible by 3
5. 12 is divisible by 4
6. 12 is divisible by 6
7. 11 is not divisible by 2 through 9

3.9: Structures is shared under a [not declared](#) license and was authored, remixed, and/or curated by LibreTexts.

- [3.9: Structures](#) by [Michael Mendez](#) has no license indicated.