Programming via PHP Conditional execution

6.1. `if` statements

Suppose we wanted our random number generation program to work even when the user enters the two ends of the range in the opposite order. What we would want is a way to test whether they are inverted, and then to silently swap them back into the proper order before proceeding. We can do this using what is called an **`if`**statement. The following solution illustrates how it would work.

```
<?php import_request_variables("pg", "form_"); ?>
<html>
<head>
<title>Generate Random Number</title>
</head>

<body>
<p>From the range <?php
    if($form_begin > $form_end) {
        $old_begin = $form_begin;
        $form_begin = $form_end;
        $form_end = $old_begin;
    }
    echo $form_begin;
?> to <?php
    echo $form_end;
?> I have selected the random number <?php
    echo rand($form_begin, $form_end);
?>.</p>
</body>
</html>
```

An **`if`** statement is always followed by a set of parentheses enclosing some yes/no expression. Following that is a set of braces enclosing a list of statements. PHP will execute these statements only when the answer to the question in the parentheses is yes. After completing these statements, it will continue after the braces. But if the answer is no, PHP will skip over the statements inside the braces (without executing them) and immediately go on to the first statement following the braces.

One thing you should be careful of: An `if` statement does not involve any semicolons. A typical beginner's mistake is to write something like `if($a > $b); {`..., with a semicolon inserted between the close parenthesis and the opening brace. Unfortunately, PHP allows this but interprets it very differently from what you will intend: If you insert a semicolon there, PHP will think you mean that the body of the `if` statement is the statement consisting of only a semicolon — a statement with nothing in it. The end result is that if `$a` exceeds `$b`, the computer will perform this empty statement (doing nothing) and then proceed into the braces; but if `$a` does not exceed `$b`, the computer will skip over the empty statement and proceed into the braces anyway.

In the above example, the three assignment statements in the braces end up swapping the values associated with `$form_begin` and `$form_end`: We first assigning`$old_begin` to refer to the original value of `$form_begin`; then we assign `$form_begin` to the original value of `$form_end`; and finally we assign `$form_end` to the value of`$old_begin`, which is the original value of `$form_begin`.

6.2. Booleans and conditions

The portion in parentheses is supposed to be a yes/no value. PHP has a special type for such values called the **boolean**. (This is the fourth type we've seen, on top of integers, floating-point values, and strings.) There are only two different boolean values, written `TRUE` and `FALSE`.

In fact, the expression in parentheses doesn't need to be a boolean. If it isn't, then PHP will first convert it to `TRUE` or `FALSE` to see whether to execute what in the braces. The rule it uses is that every integer, floating-point value, and string value is equivalent to `TRUE` except for the following, which are `FALSE`:

- the integer 0
- the floating-point value 0.0
- the string with nothing in it (`""`)
- the string containing just the 0 digit (`"0"`)

Generally speaking, I recommend avoiding any dependence on these rules; instead, you would use an assignment operator to compare values.

PHP provides a number of operators for comparing values:

< less than
> greater than

<= less than or equal (i.e., at most)
>= greater than or equal (i.e., at least)
== equal
!= not equal

The usage of the exclamation point '`!`' for the word *not* is a bit odd, but it's not hard to grow accustomed to it.

Note, though, that the way to check whether two things are equal is to use two equals signs. PHP does this because a single equals sign is already being used in PHP to represent assigning a value to a number.

In fact, the single equals sign is actually an operator, too, whose value is whatever is assigned. This can lead to weird behavior. Suppose you write the following.

```
if($form_begin = $form_end) {                          // WRONG!
    echo "Error: The range doesn't contain any numbers.";
}
```

In the parentheses, we use a single equals sign, and so PHP takes this to mean that we actually wish to change `$form_begin` so that its value is the same as `$form_end`'s value. Moreover, the value of the expression in the parentheses is this assigned value, which PHP will interprets as `TRUE` unless `$form_end` happened to be the empty string or the string `0`. So in fact the program will probably execute what is in the braces, reporting the error. And then of course it will go on and do whatever follows the braces... but now `$form_begin` and `$form_end` will represent the same value.

Needless to say, this is a fairly big error in our program. Beginning PHP programmers are almost guaranteed to encounter it, and it can be hard to see if you don't know about this distinction: *A single equals means to change a variable; a double equals means to compare two values to see whether they are equal.*

With experience, you can avoid the problem by following one simple rule: Inside parentheses on an **`if`**, you should *never* use a single equals sign. And in assignment statements, ou should always use a single equals sign.

6.3. **`else`** and **`elseif`** clauses

Sometimes you want one thing to happen in one case but a very different thing to happen in other cases. For these, PHP provides an option of including an **else** clause following the body of an **if**.

```
if($form_begin != $form_end) {
    $choice = rand($form_begin, $form_end);
    echo "I have selected $choice from the range.";
} else {
    echo "The range includes only one choice: $form_begin.";
}
```

It also sometimes happens that there are several cases. For these, you can include an **elseif** clause, optionally followed by an **else** clause. An **elseif** clause contains an additional condition in parentheses, followed by the set of statements to execute in that case if the condition turns out to be true.

```
if($form_begin < $form_end) {
    $choice = rand($form_begin, $form_end);
    echo "I have selected $choice from the range.";
} elseif($form_begin > $form_end) {
    $choice = rand($form_end, $form_begin);
    echo "I have selected $choice from the inverted range.";
} else {
    echo "The range includes only one choice: $form_begin.";
}
```

You can have have any number of **elseif** clauses in the same **if** statement; but the **else** clause, if there is one, must come last among them.

In executing an **if** statement with **elseif** clauses, PHP will consider each clause's condition in sequence, until it finds one that is true; then it will skip over all other clauses. Looked at another way: The computer will check an **elseif** clause only when the initial **if** condition and any preceding **elseif** conditions all turn out to fail. The body of the **else** will be executed only when the **if** and all **elseif** conditions fail.

The following illustrates this. Note that if `$grade` were 85, it would display only that a B was received (even though later **elseif** clauses also seemingly apply), since PHP doesn't consider later **elseif** clauses in the same statement once it finds one that applies.

```
if($grade >= 90) {
    echo "You aced the course.";
```

```
} elseif($grade >= 80) {
    echo "You received a B.";
} elseif($grade >= 70) {
    echo "You received a C.";
} else {
    echo "You failed.";
}
```

6.4. A note on indentation

You've probably noticed that the above examples are very systematic about how they are indented. In fact, PHP doesn't care about indentation, and you could just as easily place the entire program on the same line.

```
if($grade>=90){echo "You aced the course.";}elseif($grade>=80){echo
"You received a B.";}elseif($grade>=70){echo"You received a C.";}else
{echo "You failed.";}
```

However, code written in this way is very difficult to repair when there is a problem with it, and for those who haven't just typed it, it is fairly difficult to read. For these two reasons, you should follow a similar convention, where inside each set of braces, you indent each line`a bit further than where the `if` statement starts, so that it's easy to see where the body is.

You'll also notice that I'm very systematic about how I place the braces: An opening brace is usually put off at the end of a line, and the corresponding closing brace is placed at the start of the line where it appears, indented just as far as the initial `if`. This convention isn't *quite* as strong as the convention about indentation, but arranging the braces in some sort of system is very well-established among programmers — and the particular technique that I'm illustrating is the most popular system that experienced programmers use. You should follow some well-established system of indenting code and dealing with braces.

Source: http://www.toves.org/books/php/ch06-if/index.html