3

Control Structures and Loops

Writing a program would be a pretty useless exercise if we didn't have control structures. Essentially, a control structure lets a program decide how to react based on a given circumstance or piece of data. When moving from RPG to PHP, you'll find that most control structure syntax is fairly similar. Structures such as if-then-else are pretty much the same across languages. You will encounter some minor syntactical differences in PHP, but the concepts are virtually identical. In fact, if you were to use older PHP if-then-else syntax, you could almost copy the code line for line.

Conditional Structures

PHP's conditional structures are built around the **if** and **switch** statements. Let's take a look at some examples to illustrate their syntax and use.

The if Statement

To work with an **if** statement, you define a block of code that should be executed only if a given condition is true. For example:

```
<?php

$var = 1;

if ($var === 1) {
    echo "The variable equals one\n";
}
```

This sample code produces the following output:

```
The variable equals one
```

If we were to input a different variable, as in

```
<?php

$var = 2;

if ($var === 1) {
    echo "The variable equals one\n";
}
```

we would not get any output. That's because in this case the condition within the parentheses evaluates to false, and therefore the **echo** statement is not executed.

Often, you want to require that multiple conditions evaluate to true before a particular block of code is executed. Take, for instance, the example of validating a birth year:

```
<?php

$month = 15;

if ($month < 1 || $month > 12) {
    die('That is an invalid month');
}
```

Because the month value in this example is outside the specified range, this program prints out

```
That is an invalid month
```

If your logic requires that you also have a default block of code that is executed if the condition is not satisfied, you can use an if-else structure:

```
<?php

$month = 11;

if ($month < 1 || $month > 12) {
    die('That is an invalid month');
} else {
    echo "That is an excellent choice for a month";
}
```

In this **if** statement, the first condition evaluates to false, so the output is

```
That is an excellent choice for a month
```

Using an if-else-if structure, you can specify additional conditional statements to be executed in-line. No practical restriction exists on the number of conditional statements you can tie together, so, technically, you could call this an "if-else if-else if-else if etc." conditional. Two methods are available for implementing this type of logic. You can specify it using the two words (**else if**), or you can specify it by compounding both words (**elseif**). Either way works:

```
<?php

$month = 4;
$name = 'not in the list';

if ($month == 1) {
    $name = 'January';
} else if ($month == 2) {
    $name = 'February';
} elseif ($month == 3) {
    $name = 'March';
}

echo "The month is $name";
```

The preceding code produces the following output:

```
The month is not in the list
```

Notice the difference between the first and second else-if statements. The first uses both keywords, while the keywords are compounded in the second.

Earlier, we mentioned the older method of writing conditional statements. There are two primary differences between the old method and the current method, which is more C-like. The old method omits the curly braces(**{}**), and it appends a colon (**:**) to non-terminating conditional expressions (**else**, **elseif**). To illustrate these differences, let's rewrite the previous example using the older syntax. Note that **else if** is not valid using this syntax; you must use **elseif**. Notice also the use of the **endif** keyword.

```
<?php

$month = 4;
$name = 'not in the list';

if ($month == 1):
    $name = 'January';
elseif ($month == 2 ):
    $name = 'February';
elseif ($month == 3):
    $name = 'March';
endif;

echo "The month is $name";
```

The switch Statement

Sometimes, simply comparing values is easier than writing full conditional statements. As you can imagine from the preceding code examples, things can get a bit verbose in some conditional statements. Instead of evaluating several conditionals, each of which could have their own set of bugs (e.g., a mistyped variable name), **switch** simplifies the logical structure and can make your code a little easier to read. For expressing simple conditional statements, developers often prefer to use **switch**.

The **switch** block itself consists of the **switch** keyword followed by one or more **case** keywords. Each **case** keyword represents a potential true evaluation that could be executed. After each **case** statement, an optional **break** keyword is used. You use the **break** keyword (which we'll look at in more detail in just a bit) to jump out of a particular place in the code. Omitting the **break** lets statements flow from one to the next.

The **switch** statement is similar to RPG's **SELECT/WHEN** opcodes. One key difference to be aware of with **switch** is that unless you use a **break** statement to exit, each of the conditional **case** statements will be evaluated. This behavior holds true even after one of the **case** statements has been evaluated as true.

That's a fair amount of information that calls for a code example:

```
<?php

$month = 4;
$name = 'not in the list';

switch ($month) {
    case 1:
           $name = 'January';
           break;
    case 2:
           $name = 'February';
           break;
    case 3:
           $name = 'March';
           break;
}

echo "The month is $name";
```

Here is the equivalent RPG code:

```
d $month          s            10i 0 inz(4)
d $name           s            20a   inz('not in the list')

 /free
   select;
```

```
      when $month = 1;
         $name = 'January';

      when $month = 2;
         $name = 'February';

      when $month = 3;
         $name = 'March';

   endsl;

   dsply ('The month is '+%trim($name));

   return;
 /end-free
```

Note that the switch block is specified using matching parentheses, whereas the **case** blocks are not. This PHP code functions essentially the same way as what you saw earlier with **if** statements. Let's look at another example that shows what you can do with a **switch** statement:

```
<?php

$month = 2;
$name = 'not in the list';

switch ($month) {
    case 1:
    case 2:
    case 3:
           $name = 'earlier than April';
           break;
}

echo "The month is $name";
```

This code prints out

```
The month is earlier than April
```

This example illustrates what we noted earlier about flowing from one case to another. Because the **case 2** statement contains no break, statement execution continues until a **break** statement is encountered. Because the "break" function

is always implied in the RPG **SELECT/WHEN** structure, the RPG code to accomplish the same output would most likely not use the **SELECT/WHEN** structure.

As with the final **else** statement following an **if** statement, you can define a default condition when you use **switch**. You do so using the **default** keyword, which serves the same function as the **OTHER** opcode in RPG. You code the **default** in place of a distinct **case** statement, as follows:

```
<?php

$month = 2;

switch ($month) {
    case 1:
    case 2:
    case 3:
           $name = 'earlier than April';
           break;
    default:
           $name = 'not in the list';
           break;
}

echo "The month is $name";
```

Here is the equivalent RPG code:

```
d $month          s            10i 0 inz(2)
d $name           s            20a   inz('not in the list')

 /free

   select;

      when $month = 1 or $month = 2 or $month = 3;
         $name = 'earlier than April';

      other;
         $name = 'not in the list';

   endsl;

   dsply ('The month is '+%trim($name));

   return;
 /end-free
```

Loops

PHP's looping structures implement the same type of iterative logic you can code in RPG to execute a block of code a number of times. PHP provides three looping statements: **for**, **while**, and **do-while**.

The for Loop

When you need to iterate over a defined number of values in PHP, you typically use a **for** loop. In the initialization portion of the loop, you provide a value that is iterated over a given number of times, usually with the value being incremented via a coded or internal operation known as a *post-op* (because it is executed after the loop operations). In RPG's **FOR/ENDFOR** operation (available only in the free-format version of the language), the **BY** clause implements the post-op. In PHP, the post-op is an actual assignment expression that is executed at the end of each iteration until the conditional statement evaluates to false. This may sound complicated, but it's really quite simple, even though the syntax differs somewhat from RPG.

Rather than providing opcodes to determine how you will iterate over the loop, you use conditions and operations to determine the looping characteristics. The syntax is much like C or Java. As with most PHP operations, you enclose the block of code over which you are iterating within curly braces:

```
<?php

for ($count = 1; $count <= 10; $count++) {
    echo "The count is {$count}\n";
}

?>
```

This code produces the following output:

```
The count is 1
The count is 2
The count is 3
The count is 4
```

```
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
```

Here is the equivalent RPG code:

```
d $count          s             10i 0

 /free

  // Note that the "by 1" post-op is optional in this case
  // and is included only for clarity.
  // for $count = 1 to 10; would function exactly the same.

   for $count = 1 by 1 to 10;
      dsply ('The count is '+%trim(%editc($count:'Z')));
   endfor;

   return;
 /end-free
```

In PHP, iteration over the loop continues until the condition evaluates to false. This approach differs a little from the RPG implementation. Indeed, in PHP you can easily create an infinite loop:

```
<?php

for ($count = 1; $count > 0; $count++) {
   echo "The count is {$count}\n";
}
```

If you write this code, make sure you're able to easily kill the process because you'll be able to cook an egg on the processor afterwards!

In RPG, when you count down in a loop, you use the **DOWNTO** opcode:

```
FOR i=10 by 1 DOWNTO 1
  ...code to be executed
ENDFOR
```

When writing this logic in PHP, you simply change the post-op expression and the comparison:

```
<?php

for ($i = 10; $i >= 1; $i--) {
    echo "The number is {$i}\n";
}
```

Or, if you wanted to iterate using a hexadecimal:

```
<?php

for ($i = 1; $i < 50; $i += 0x0b) {
    echo "The number is {$i}\n";
}
```

This code produces the output

```
The number is 1
The number is 12
The number is 23
The number is 34
The number is 45
```

Sometimes, you may want to intentionally create an infinite loop. You can accomplish this goal by omitting all parts of the **for** loop. Chances are you won't want to do this when handling an HTTP request because you usually want such requests to be short-lived, and infinity is not short. This technique is useful, however, if you're using PHP to run as a server daemon, or service that is listening on a socket:

```
<?php

for (;;) {
    $client = socket_accept($server);
    echo "I got someone's socket!!\n";
}
```

The while Loop

You typically use the **while** loop when you don't have a specific iterative sequence you want to go over but you have code that needs to be executed zero or more times — in other words, code that might not be executed. A PHP **while** loop is roughly equivalent to **DOW/ENDDO** in RPG.

```
<?php

$printAuthors = $_GET['printAuthors'];
$kevinPrinted = false;
$jeffPrinted = false;

while ($printAuthors) {

    if (!$jeffPrinted) {
           echo "Jeff\n";
           $jeffPrinted = true;
    } else if (!$kevinPrinted) {
           echo "Kevin\n";
           $kevinPrinted = true;
    }

    $printAuthors = !($jeffPrinted && $kevinPrinted);
}
```

The preceding code will be run only if someone provides a **GET** parameter of **printAuthors** that can be evaluated as a Boolean true. *If* someone were to provide that value, the following output would result:

```
Jeff
Kevin
```

The do-while Loop

PHP's **do-while** loop is similar to the **while** loop, except that rather than executing its code zero or more times, it executes the code one or more times. In other words, a **do-while** loop will always be executed at least once, similar to a **DOU/ENDDO** in RPG. Say you want to count through the first 10 values of a Fibonacci sequence:

```
<?php

$num1 = 0;
$num2 = 0;
$fs = 0;

do {
    $currentNum = $num1 + $num2;
    echo "F{$fs} = {$currentNum}\n";
    if ($fs == 0 || $fs == 1) {
           $num2 = 1;
    } else {
           $tmp = $num2;
           $num2 = $num2 + $num1;
           $num1 = $tmp;
    }
    $fs++;
} while ($fs < 10);
```

The resulting output:

```
F0 = 0
F1 = 1
F2 = 1
F3 = 2
F4 = 3
F5 = 5
F6 = 8
F7 = 13
F8 = 21
F9 = 34
```

Another use for a **do-while** loop is when iterating over the result set of a **LEFT OUTER** JOIN SQL query. We'll look at database access later on.

Modifying Loop Iteration

PHP provides two different ways to modify loop behavior that is outside the conditional statements in the loop: the **continue** keyword and the **break** keyword.

Continue

You use the **continue** keyword in a loop when you want to stop executing the code in the current iteration and continue on to the next iteration. The **continue** statement works exactly like RPG's **ITER** opcode. Although you can imitate this functionality by using a big **if** statement, that approach doesn't always make for the neatest code. Let's take a look at code that prints out the odd numbers between 1 and 10:

```
<?php

for ($i = 1; $i <= 10; $i++) {
    if ($i % 2 != 0) {
            echo "The number is {$i}\n";
    }
}
```

We could write this example a little more neatly using the **continue** keyword:

```
<?php

for ($i = 1; $i <= 10; $i++) {
    if ($i % 2 == 0) continue;

    echo "The number is {$i}\n";
}
```

Here is the equivalent RPG code:

```
d $i              s             10i 0

 /free

   for $i = 1 by 1 to 10;
```

```
      if %rem( $i : 2 ) = 0;
         iter;
      endif;

      dsply ('The number is '+%trim(%editc( $i : 'Z' )) );

   endfor;

   return;
 /end-free
```

Although this example is a little simplistic, **continue** can become more useful when your logic is more complex than simply writing the odd numbers from 1 to 10.

The **continue** keyword is quite helpful when used alone, but it can also take an optional parameter that extends its functionality beyond that of **ITER**. The parameter notes how many levels of loops to continue from. You might think this option lets you note the number of iterations to skip over, but this is not true. Used with a parameter, **continue** jumps to the next iteration of the specified depth in a nested loop.

Consider the following code:

```
<?php

for ($i1 = 1; $i1 <=2; $i1++) {
    echo "Exec outer loop\n";
    for ($i2 = 1; $i2 <=2; $i2++) {
          echo "Exec middle loop\n";
          for ($i3 = 1; $i3 <=2; $i3++) {
                echo "$i1 $i2 $i3\n";
          }
    }
}
```

Here is the output produced by this code:

```
Exec outer loop
Exec middle loop
1 1 1
1 1 2
Exec middle loop
1 2 1
1 2 2
```

```
Exec outer loop
Exec middle loop
2 1 1
2 1 2
Exec middle loop
2 2 1
2 2 2
```

Here is the equivalent RPG code:

```
d $i1             s             10i 0
d $i2             s             10i 0
d $i3             s             10i 0

 /free

   for $i1 = 1 to 2;
      dsply ('Exec outer loop');
      for $i2 = 1 to 2;
         dsply ('Exec middle loop');
         for $i3 = 1 to 2;
            dsply (%trim(%editc($i1:'Z')) + ' ' +
                   %trim(%editc($i2:'Z')) + ' ' +
                   %trim(%editc($i3:'Z')) );
         endfor;
      endfor;
   endfor;

   return;
 /end-free
```

Now, let's change the example to specify a parameter with **continue**:

```
<?php

for ($i1 = 1; $i1 <=2; $i1++) {
    echo "Exec outer loop\n";
    for ($i2 = 1; $i2 <=2; $i2++) {
          echo "Exec middle loop\n";
          for ($i3 = 1; $i3 <=2; $i3++) {
                echo "$i1 $i2 $i3\n";
                continue 2;
          }
    }
}
```

Here is the output that results:

```
Exec outer loop
Exec middle loop
1 1 1
Exec middle loop
1 2 1
Exec outer loop
Exec middle loop
2 1 1
Exec middle loop
2 2 1
```

The specified **continue** statement causes the PHP interpreter to skip to the bottom of the loop that is two levels up in the nesting. To be clear, when **continue 2** is encountered in the sample code, execution jumps to the bottom of the current **$i3** loop and to the bottom of the **$i2** loop structure (two levels). Execution then resumes at the top of the **$i2** loop. If we changed the **continue** statement to **continue 3**, execution would jump to the bottom of the **$i3** loop, to the bottom of the **$i2** loop, and to the bottom of the **$i1** loop and then would resume at the top of the **$i1** loop.

No opcode in RPG is equivalent to this use of **continue**. In RPG, we must simulate this functionality using the **LEAVE** opcode:

```
d $i1             s             10i 0
d $i2             s             10i 0
d $i3             s             10i 0

 /free
   for $i1 = 1 to 2;
      dsply ('Exec outer loop');
      for $i2 = 1 to 2;
         dsply ('Exec middle loop');
         for $i3 = 1 to 2;
            dsply (%trim(%editc($i1:'Z')) + ' ' +
                   %trim(%editc($i2:'Z')) + ' ' +
                   %trim(%editc($i3:'Z')) );
                   leave;
         endfor;
      endfor;
   endfor;

   return;
 /end-free
```

If we change the example to specify a variable on the **continue** statement:

```
<?php

for ($i1 = 1; $i1 <=2; $i1++) {
    echo "Exec outer loop\n";
    for ($i2 = 1; $i2 <=2; $i2++) {
           echo "Exec middle loop\n";
           for ($i3 = 1; $i3 <=2; $i3++) {
                    echo "$i1 $i2 $i3\n";
                    continue $i1;
           }
    }
}
```

The following output results:

```
Exec outer loop
Exec middle loop
1 1 1
1 1 2
Exec middle loop
1 2 1
1 2 2
Exec outer loop
Exec middle loop
2 1 1
Exec middle loop
2 2 1
```

Break

Where **continue** goes to the next iteration of the loop, **break** completely jumps out of the loop. This functionality is the same as that of RPG's **LEAVE** opcode. Let's look at the same code we saw beforehand and see how it works differently with **break**:

```
<?php

for ($i1 = 1; $i1 <=2; $i1++) {
    echo "Exec outer loop\n";
    for ($i2 = 1; $i2 <=2; $i2++) {
```

```
            echo "Exec middle loop\n";
            for ($i3 = 1; $i3 <=2; $i3++) {
                        echo "$i1 $i2 $i3\n";
                        break;
            }
    }
}
```

This code prints out:

```
Exec outer loop
Exec middle loop
1 1 1
Exec middle loop
1 2 1
Exec outer loop
Exec middle loop
2 1 1
Exec middle loop
2 2 1
```

Notice that the third number never gets above **1**. That's because **break** is jumping completely out of that loop. The RPG equivalent for this code is the same as the previous example that uses the **LEAVE** opcode in place of **continue**.

Just like **continue**, **break** can accept an optional parameter that notes the level that it is supposed to break out of:

```
<?php

for ($i1 = 1; $i1 <=2; $i1++) {
    echo "Exec outer loop\n";
    for ($i2 = 1; $i2 <=2; $i2++) {
           echo "Exec middle loop\n";
           for ($i3 = 1; $i3 <=2; $i3++) {
                        echo "$i1 $i2 $i3\n";
                        break $i1;
           }
    }
}
```

The resulting output:

```
Exec outer loop
Exec middle loop
1 1 1
Exec middle loop
1 2 1
Exec outer loop
Exec middle loop
2 1 1
```

In this case, there simply is not an applicable RPG equivalent. Obviously, we could create code to achieve the same results, but we would need to use completely different logic.

Exam and Exercise

1. Create a script that yields the same output as the following RPG program.

```
   // Chapter 3 - Exercise 1
d $a               s             10i 0
d $b               s             10i 0

 /free

  $a = 1;
  $b = 2;

  if $a = 1 or $b = 1;
     dsply ('Condition one true');
  endif;

  if $a = 1 and $b = 2;
     dsply ('Condition two true');
  endif;

  *inlr = *on;
  return;
 /end-free
```

2. Create a script that yields the same output as the following RPG program.

```
  // Chapter 3 - Exercise 2
d $count          s             10i 0

 /free

  // for those familiar with the RPG implementation of FOR
  for $count = 5 by 4 to 35;
     dsply $count;
  endfor;


  // and for those who are not
  $count = 5;
  dow $count <= 35;
     dsply $count;
     $count += 4;
  enddo;

  *inlr = *on;
  return;
 /end-free
```

3. Create a script that yields the same output as the following RPG program.

```
  // Chapter 3 - Exercise 3
d $count          s             10p 5

 /free

  $count = 7;
  dow $count < 40;
     dsply %trim(%editc($count:'1'));
     $count += $count / 2;
  enddo;

  *inlr = *on;
  return;
 /end-free
```

4. Create a script that yields the same output as the following RPG program.

```
  // Chapter 3 - Exercise 4
d $count          s             10i 0 inz(1)

 /free

  if $count > *zeros;

     dow $count < 10;
        dsply %trim(%editc($count:'Z'));
        $count += 1;
     enddo;

  endif;

  *inlr = *on;
  return;
 /end-free
```

5. Create a script that yields the same output as the following RPG program.

```
  // Chapter 3 - Exercise 5
d $counter        s             10i 0
d $c1             s             10i 0
d $c2             s             10i 0

 /free

  for $c1 = 0 by 1 to 4;

     for $c2 = 0 by 1 to 4;

        $counter += 1;
        if $counter = 10;
           dsply 'I''m done with this';
           leave;
        endif;
        dsply (%trim(%editc($c1:'1'))
             +' '+%trim(%editc($c2:'1')));

     endfor;

     if $counter = 10;
        leave;
     endif;

  endfor;

  *inlr = *on;
  return;
 /end-free
```