HOUR 3

Controlling the Program’s Flow

What You’ll Learn in This Hour:

- Block statements
- Operators
- Looping
- Labels
- Exiting Perl after a program execution

In Hour 2, “Perl’s Building Blocks: Numbers and Strings,” you learned about statements, operators, and expressions. All the examples in that hour had one thing in common: All the statements were executed in order from top to bottom and were executed only once.

One of the reasons that you use computers is that computers are very good at performing repetitive tasks—over and over again—without getting tired or bored and without developing carpal tunnel syndrome. So far, you haven’t had any way of telling Perl to “do this task X times” or to “repeat this task until it’s done.” In this hour, you will learn about Perl’s control structures. Using them, you can group statements into something called a statement block and run the group of statements repeatedly until they’ve done what you want.

The other chore that computers excel at is making decisions quickly. It would be tiresome—not to mention silly—if a computer had to ask you every time it made a decision. The very act of retrieving and reading your email causes your computer to make millions of decisions that you really don’t want to deal with: how to assemble network traffic, which colors to make each pixel on your screen, how your incoming mail should be
pulled apart and displayed, what should be done when your mouse cursor moves even a tiny bit, and countless others. All these decisions are made up of other decisions, and some of them are made thousands of times per second. In this hour, you will learn about **conditional statements**. Using these statements, you can write blocks of code that will be executed or not, depending on decisions made in your Perl program.

## Blocks

The fundamental way to group statements in Perl is the **block**. To group statements in a block, just surround the statements with a matched set of curly braces, as shown here:

```perl
{  statement_a;
    statement_b;
    statement_c;
}
```

Within the block, statements execute from the top down, as they have until now. You can have other, smaller blocks of statements nested within a block, as you can see here:

```perl
{  statement_a;
    {  statement_x;
        statement_y;
    }
}
```

The format of the block, like the rest of Perl, is free-form. The statements and curly braces can be on one line or on several lines, as shown here, and with any kind of alignment you want, as long as you always have a matched set of curly braces:

```perl
{ statement;  { another_statement; }
    { last_statement;  }
}
```

Although you can arrange blocks any way you would like, programs can be hard to read if they’re just thrown together. Good indenting, although not necessary for Perl to understand your code, makes for human-readable Perl. It can help you keep track of your program’s logic.

Blocks that occur by themselves within a program are called **bare blocks** or **naked blocks**. Most of the time, however, you will encounter blocks attached to other Perl statements.
The if Statement

To control whether statements are executed based on a condition in a Perl program, you usually use an if statement. The syntax of an if statement is as follows:

\[ \text{if (expression) block} \]

The statement works like this: If the expression evaluates to true, the block of code is run. If the expression is false, the block of code is not run. Remember that the block includes the braces. Consider this example:

```
if ( $r == 5 ) {
    print 'The value of $r is equal to 5.';
}
```

The expression being tested is \( \text{r == 5} \). The == symbol is an equality operator. If the two operands on either side—\( \text{r} \) and 5—are numerically equal to one another, the expression is considered to be true, and the print statement is executed. If \( \text{r} \) is not equal to 5, the print statement is not executed.

The if statement can also run one set of statements if a condition is true and another set of statements if it's not. That structure is called an if-else statement. The syntax looks like this:

```
if (expression) # If expression is true...
    block1       # ...this block of code is run.
else
    block2      # Otherwise this block is run.
```

The first block, block1, is run only if the expression is true; if the expression is not true, block2, following the else, is run. Now consider this example:

```
$r=<STDIN>;
chomp $r;
if ($r == '10') {
    print "$r is 10", "\n";
} else {
    print "$r is something other than 10...", "\n";
    $r=10;
    print "$r has been set to 10", "\n";
}
```

In the preceding example, notice that to assign a value to \( \text{r} \), I used the assignment operator, =. To test the value of \( \text{r} \), I used the numeric equality test operator, ==. Do not confuse them in your programs because debugging can be very difficult. Remember that = assigns a value and == tests for equality. If you use the -w option to turn warnings on, Perl can sometimes warn you if you have made this error.
Yet another way of structuring an if statement is to check multiple expressions and run code depending on which expressions are true:

```perl
if (expression1)     # If expression1 is true ...
    block1        # ...run this block of code.
elsif (expression2)  # Otherwise, if expression2 is true...
    block2       # ...Run this block of code.
else
    block3       # If neither expression was true, run this.
```

You can read the preceding block like this: If the expression labeled `expression1` is true, the block `block1` is run. Otherwise, control falls to the `elsif` and `expression2` is tested; if it’s true, `block2` is run. If neither `expression1` nor `expression2` is true, `block3` is run. The following is an example of real Perl code that demonstrates this syntax:

```perl
$r=10;
if ($r==10) {
    print '$r is 10!';
} elsif ($r == 20) {
    print '$r is 20!';
} else {
    print '$r is neither 10 nor 20';
}
```

### The Other Relational Operators

So far, you’ve been comparing numeric quantities in your if statements with the equality operator, `==`. Perl actually has quite a few operators for comparing numeric values, most of which are listed in Table 3.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>==</code></td>
<td>$x == $y</td>
<td>True if $x equals $y</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>$x &gt; $y</td>
<td>True if $x is greater than $y</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>$x &lt; $y</td>
<td>True if $x is less than $y</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>$x &gt;= $y</td>
<td>True if $x is greater than or equal to $y</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>$x &lt;= $y</td>
<td>True if $x is less than or equal to $y</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>$x != $y</td>
<td>True if $x is not equal to $y</td>
</tr>
</tbody>
</table>

To use these operators, you can simply put them in anywhere that your program needs to test relations between numeric values. An example of the use of these operators in an if statement is shown in Listing 3.1, which you can type in and run (do not type the line numbers—"1:" and so forth). The various parts of the program work as follows:
LISTING 3.1 A Small Number Guessing Game

1:   #!/usr/bin/perl -w
2:   $im_thinking_of=int(rand 10);
3:   print "Pick a number:";
4:   $guess=<STDIN>;
5:   chomp $guess;   # Don't forget to remove the newline!
6:   if ($guess>$im_thinking_of) {
7:       print "You guessed too high!
";
8:   } elsif ($guess < $im_thinking_of) {
9:       print "You guessed too low!
";
10:  } else {
11:      print "You got it right!
";
12:  }

Line 1: This line is the standard first line of a Perl program; it indicates the interpreter you want to run and the -w switch to enable warnings. See Hour 1, “Getting Started with Perl”; your first line may need to look slightly different.

Line 3: The (rand 10) function picks a number between 0 and 10, and the int() function truncates it so that only integers 0 to 9 are assigned to $im_thinking_of.

Lines 4–6: These lines ask the user for the guess, assign it to $guess, and remove the trailing newline character.

Lines 8–9: If $guess is greater than the number in $im_thinking_of, these lines print an appropriate message.

Lines 10–11: Otherwise, if $guess is less than the number in $im_thinking_of, these lines print that message.

Lines 12–13: The only choice left is that the user guessed the number.

The operators in Table 3.1 are used only for testing numeric values. Using them to test nonalphabetic data results in behavior that you probably don’t want. Consider this example:

```
$first='Simon';
$last='simple';
if ($first == $last) {   # == is not what you want!
    print "The words are the same!\n";
}
```

The two values $first and $last actually test equal to each other. The reason was explained in Hour 2: If nonnumeric strings are used when Perl is expecting numeric values, the strings evaluate to zero. So the preceding if expression looks something like this to Perl: if ( 0 == 0 ). This expression evaluates to true, and that’s probably not what you wanted.
If warnings are turned on, trying to test two alphabetic values (simple and Simon in the preceding snippet) with `==` will generate a warning message when the program runs to alert you to this problem.

If you want to test nonnumeric values, you can use another set of Perl operators, which are listed in Table 3.2.

**TABLE 3.2  Alphanumeric Relational Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>$s eq $t</td>
<td>True if $s is equal to $t</td>
</tr>
<tr>
<td>gt</td>
<td>$s gt $t</td>
<td>True if $s is greater than $t</td>
</tr>
<tr>
<td>lt</td>
<td>$s lt $t</td>
<td>True if $s is less than $t</td>
</tr>
<tr>
<td>ge</td>
<td>$s ge $t</td>
<td>True if $s is greater than or equal to $t</td>
</tr>
<tr>
<td>le</td>
<td>$s le $t</td>
<td>True if $s is less than or equal to $t</td>
</tr>
<tr>
<td>ne</td>
<td>$s ne $t</td>
<td>True if $s is not equal to $t</td>
</tr>
</tbody>
</table>

These operators decide “greater than” and “less than” by examining each character left to right and comparing them in ASCII order. This means that strings sort in ascending order: most punctuation first, and then numbers, uppercase, and finally lowercase. For example, `1506` compares less than `Happy`, which compares less than `happy`.

**What Truth Means to Perl**

Up to this point, you’ve been reading about “if this expression is true...” or “…evaluates to true…,” but you haven’t seen any formal definition of what Perl thinks “true” is. Perl has a few short rules about what is true and what is not true, and the rules actually make sense when you think about them for a bit. The rules are as follows:

- The number 0 is false.
- The empty string (“”) and the string “0” are false.
- The undefined value `undef` is false.
- Everything else is true.

Make sense? The only other point to remember is that when you’re testing an expression to see whether it’s true or false, the expression is simplified—functions are called, operators are applied, math expressions are reduced, and so on—and
then converted to a scalar value for evaluation to determine whether it is true or false.

Think about these rules, and then take a look at Table 3.3. Try to guess whether the expression is true or false before you look at the answer.

### Table 3.3 True or False Examples

| Expression | True or False?
|------------|-----------------------
| 0          | False. The number 0 is false.
| 10         | True. It is a nonzero number and therefore true.
| 9>8        | True. Relational operators return true or false, as you would expect.
| -5+5       | False. This expression is evaluated and reduced to 0, and 0 is false.
| 0.00       | False. This number is another representation of 0, as are 0x0, 00, Ob0, and 0e00.
| "          | False. This expression is explicitly mentioned in the rules as false.
| "          | True. There's a space between the quotes, which means they're not entirely empty.
| "0.00"     | True. Surprise! It's already a string, but not "0" or "". Therefore, it is true.
| "00"       | True also, for the same reason as "0.00"
| "0.00" + 0 | False. In this expression, 0.00+0 is evaluated, the result is 0, and that is false.

Until now, you've seen only expressions with relational operators as the conditions in if statements. Actually, you can use any expression that will evaluate to true or false the way you would want:

```perl
# The scalar variable $a is evaluated for true/false
if ($a) { ... }

# Checks the length of $b. If nonzero, the test is true.
if (length($b)) { .... }
```

Recall from Hour 2 that the assignment operator = returns a value—the value that was assigned. That value, of course, is also true or false:

```perl
$s = 1;
$b = 2;
print qq(The statement "$s = $b" is );
if ($s = $b) { # value is 2, therefore true
  print "true";
} else {
  print "false";
}
```
This code prints the statement "1 = 2" is true. Now you see why using = when you meant == is such a pitfall.

The value undef is a special value in Perl. Variables that have not yet been set have the value of undef, and some functions return undef on failure. It’s not 0, and it’s not a regular scalar value. It’s kind of special. In a test for truth, undef always evaluates to false. If you try to use the undef value in a math expression, it’s treated as though it were 0.

Using variables that haven’t been set yet is usually a sign of a programming error. If you’re running your Perl programs with warnings enabled, the value undef in an expression or as an argument to some functions causes Perl to generate the warning Use of uninitialized value.

Logical Operators

When you’re writing programs, you sometimes need to code something like the following: Do this if $x is true and if $y is true, but not if $z is true. You can code this example into a series of if statements, but it’s not pretty:

```perl
if ($x) {
    if ($y) {
        if ($z) {
            # do nothing
        } else {
            print "All conditions met.\n";
        }
    }
}
```

Perl has a whole class of operators for connecting together true and false statements like this, called logical operators. The logical operators are shown in Table 3.4.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Alternative Name</th>
<th>Example</th>
<th>Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
<td>$s &amp;&amp; $t</td>
<td>True only if $s and $t are true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$q and $p</td>
<td>True only if $q and $p are true</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$c or $d</td>
<td>True if $c is true or $d is true</td>
</tr>
<tr>
<td>!</td>
<td>not</td>
<td>! $m</td>
<td>True if $m is not true</td>
</tr>
<tr>
<td></td>
<td></td>
<td>not $m</td>
<td>True if $m is not true</td>
</tr>
</tbody>
</table>
Using the operators in Table 3.4, you could rewrite the previous snippet much more concisely as follows:

```perl
if ($x and $y and not $z ) {
    print 'All conditions met.
';
}
```

Expressions connected with logical operators are evaluated from left to right, until a value of true or false can be determined for the entire expression. Examine the following code:

```perl
1:   $a=0;
2:   $b=1;
3:   $c=2;
4:   $d="";
5:  if ($a and $b) {  print '$a and $b are true'; }
6:  if ($d or $b) { print 'either $d or $b is true'; }
7:  if ($d or not $b or $c)
8:      { print '$d is true, or $b is false or $c is true'; }
```

Lines 1–4: These lines give the variables default values.

Line 5: $a is evaluated first. It is false, so the and expression cannot possibly be true. $b is never evaluated; it doesn’t have to be because the truth of the expression is known after evaluating $a. The print is not executed.

Line 6: $d is evaluated first. It is false. Even if $d is false, the expression might still be true—because it contains a logical or—so $b is examined next. $b turns out to be true; therefore, the expression is true, and the print happens.

Line 7: $d is evaluated first. It is false. But although $d is false, the expression might still be true—as seen in line 4—because it contains a logical or. Next, the truth of $b—1, so true—is negated, so this expression becomes false. The truth of the or statement cannot be determined yet, so $c is evaluated. $c turns out to be true, so the whole expression is true, and the print happens.

This behavior—stopping the evaluation of a logical expression as soon as the truth can be determined—is called **short-circuiting**. This feature is used by Perl programmers to construct simple flow-control statements out of logical operators and to avoid the if statement entirely:

```perl
$message="A and B are both true.
"  ($a and $b) or $message="A and B are not both true.
";
```

In the preceding example, if either $a or $b is false, the right side of the or must be evaluated, and the message is changed. If both $a and $b are true, the or must be true, and it's not necessary to evaluate the right side. The truth value of the entire expression isn't used at all; this example uses the short-circuit side effects of the and or operators to manipulate $message.
The `||` operator and `or` aren't completely alike. They differ in that `||` has higher precedence than `or`. This means that in an expression, `||` tends to be evaluated sooner than `or`. This is similar to multiplication having higher precedence than addition in normal mathematical expressions. The same caution applies to `&&/and` and `/not`. If you're unsure, use parentheses to guarantee the order in which the expression will be evaluated.

An interesting property of Perl's logical operators is that they don't simply return true or false. They actually return the last value evaluated. For example, the expression `5 && 7` doesn't just return true—it returns 7. This allows constructs like this:

```perl
# Set $new to old value if $old is true, # otherwise use the string 'default'. $new=$old || 'default';
```

which is a little more concise than the code

```perl
$new=$old;
if (! $old) { # was $old empty (or false)?  $new="default";
}
```

This trick can make your code less readable. Understanding how it works can be helpful, though, if you are going to be looking at much Perl code written by others.

## Looping

As you read in the Introduction, sometimes just making decisions and running code conditionally are not enough. Often you need to run pieces of code over and over again. The exercise presented in Listing 3.1 wasn't much fun because you could take only one guess (well, that and because it's a pointless game). If you want to be able to take multiple guesses, you need to be able to repeat sections of code conditionally, and that's what looping is all about.

### Looping with while

The simplest kind of loop is a `while` loop. A `while` loop repeats a block of code as long as an expression is true. The syntax for a `while` loop looks like this:

```perl
while (expression) block
```

When Perl encounters the `while` statement, it evaluates the `expression`. If the `expression` is true, the `block` of code is run. Then, when the end of the block is reached, the `expression` is re-evaluated. If it's still true, the `block` is repeated, as in the following snippet:
LISTING 3.2  Sample *while* Loop

1:   $counter=0;
2:   while ($counter < 10 ) {
3:       print "Still counting...$counter\n";
4:       $counter++;
5:   }

Line 1: $counter is initialized to zero.
Line 2: The expression $counter < 10 is evaluated. If it's true, the code in the block is run.
Line 4: The value of $counter is incremented by 1.
Line 5: The ) marks the end of the block started on line 2 with a {. At this point, Perl returns to the top of the *while* loop and re-evaluates the conditional expression.

**Looping with *for***

The *for* statement is the most complicated and versatile of Perl's looping constructs. The syntax looks like this:

```
for ( initialization; test; increment ) block
```

The three sections of the *for* statement—*initialization*, *test*, and *increment*—are separated by semicolons. When Perl encounters a *for* loop, the following sequence takes place:

1. The *initialization* expression is evaluated.
2. The *test* expression is evaluated; if it's true, the *block* of code is run.
3. After the *block* is executed, the *increment* is performed, and the *test* is evaluated again. If the *test* is still true, the *block* is run again. This process continues until the *test* expression is false.

The following is an example of a *for* loop:

```
for( $a=0; $a<10; $a=$a+2 ) {
    print 'a is now $a\n';
}
```

In this snippet, $a is set to 0, and the test $a<10 is performed and found to be true. The body of the loop prints a message. The increment is then run—$a=$a+2—which increases the value of $a by 2. The test is performed again, and the loop repeats. This particular loop repeats until the value of $a is 10, when the test will be false and the program will continue running after the *for* loop.
You don’t have to use the increment in the for statement for counting; it simply iterates until the test is false. In fact, you should be aware that each of the three parts of the for statement is optional, although the two semicolons are required. The following for statement is missing some pieces but is still perfectly valid:

```perl
$i=10;              # initialization
for( ; $i>-1; ) {
  print "$i..";
  $i--;          # actually, a decrement.
}
print "Blast off!\n";
```

Omitting the test portion means that you have to have some other way to exit the loop, or it will loop forever.

**Other Flow Control Tools**

Controlling the way your program executes with loops and condition statements is fine, but other flow control statements are needed to make readable programs. For example, Perl has statements to exit a while loop early, to skip certain portions of a for loop, to exit an if statement before the end of a block, or even to exit your program without falling off the end. Using some of the constructs explained in this section can make your Perl programs more concise and easier to read.

**Odd Arrangements**

The if statements have one more possible syntax. If you have only one expression inside the if block, the expression can actually precede the if statements. So, instead of writing

```perl
if (test_expression ) {
  expression ;
}
```

you can write

```perl
expression if (test_expression );
```

The following are a couple of examples of this variation of the syntax:

```perl
$correct=! if ($guess == $question);  
print "No pi for you!" if ( $ratio != 3.14159);
```

You usually use this syntax in Perl code for clarity; sometimes reading the code is easier if you see the action before the condition. The expression preceding the if must be a single expression. The if statement must also be followed by a semicolon.
Fine-Grained Control

In addition to blocks, for, while, if, and other flow-control statements that control blocks of code, you can use Perl statements to control the flow within the blocks.

The simplest statement that gives you this control is last. The last statement causes the innermost currently running loop block to be exited. Consider this example:

```perl
while($i<15) {
    last if ($i==5);
    $i++;
}
```

The last statement causes the while loop to exit when the value of $i is 5, instead of normally when the while test is false. When you have multiple nested loop statements, last exits the loop currently running.

**LISTING 3.3  Example of the last Statement**

```perl
  for($i=0; $i<100; $i++) {
    for($j=0; $j<100; $j++) {
      if ($i * $j == 140) {
        print "The product of $i and $j is 140\n";
        last;
      }
    }
  }
```

The set of nested loops in Listing 3.3 finds all the whole numbers less than 100 whose products are 140—2 and 70, 4 and 35, and so on—rather inefficiently. The point to note here is the last statement. When a product is found, the result is printed, and the inner loop (the loop iterating over $j) is exited. The outer loop continues executing (by incrementing $i) and reruns the inner loop.

The next statement causes control to be passed back to the top of the loop and the next iteration of the loop to begin, if the loop isn't finished:

```perl
for($i=0; $i<100; $i++) {
    next if (not $i % 2);
    print "An odd number=$i\n";
}
```

This loop prints all the even numbers from 0 to 98. The next statement causes the loop to go through its next iteration if $i is not even; the $i % 2 expression is the remainder of $i divided by 2. In this case, the print statement is skipped. (A much more efficient way to write this loop would be simply to increase $i by 2, but that wouldn't demonstrate next, would it)
The *redo* statement is similar to *next*, except that the condition isn’t re-evaluated. Perl resumes execution back at the beginning of the block and doesn’t check whether the termination condition has been met yet.

**Labels**

Perl allows blocks and some loop statements (*for, while*) to be labeled. That is, you can place an identifier in front of the block or statement:

```perl
MYBLOCK: {
}
```

The preceding block is labeled as *MYBLOCK*. Label names follow the same conventions as variable names, with one small exception: Label names do not have an identifying character—%, $, @—as variables do. It’s important to make sure that label names do not clash with Perl’s built-in keywords. As a matter of style, it’s best if label names are all uppercase. You should not have any conflicts with any current or future Perl keywords that way. The *for* and *while* statements can all have labels as well.

```perl
OUTER: while($expr ) {
    INNER: while($expr) {
        statement;
    }
}
```

The *last*, *redo*, and *next* statements can each take a label as an argument. You therefore can exit a specific block. The code in Listing 3.2 found two factors of 140 by using a nested pair of *for* loops. Suppose you wanted to exit the loops as soon as a factor is found. Without labels, you would need a complex arrangement of flag variables (variables whose only purpose is to store a true or false value for program flow control) and if statements between the two loops because you cannot exit the outer loop from within the inner loop. Labels solve this problem:

```perl
OUTER: for($i=0; $i<100; $i++) {
    for($j=0; $j<100; $j++) {
        if ($i * $j == 140) {
            print 'The product of $i and $j is 140\n';
            last OUTER;
        }
    }
}
```

Now the last statement can specify which loop it wants to exit—in this case, the *OUTER* loop. This snippet prints only the first pair of factors of 140 that it finds.
Leaving Perl

The exit statement is the ultimate flow-control tool. When Perl encounters an exit statement, the program stops executing, and an exit status is returned by Perl to the operating system. This exit status is usually used to indicate successful completion of the program. You’ll learn more about exit statuses in Hour 11, “System Interaction.” For now, an exit status of zero means everything went okay. The following is an example of exit:

```perl
if ($user_response eq 'quit') {
    print "Good Bye!\n";
    exit 0; # Exit with a status of 0.
}
```

The exit statement has some side effects that are important to your operating system. When an exit is performed, any open files are closed, file locks are released, memory allocated by Perl is released to the system, and the Perl interpreter performs a clean shutdown.

Exercise: Finding Primes

What computer language primer would be complete without this little gem? In this exercise, you will examine a small program to find and print prime numbers. Prime numbers are divisible only by 1 and themselves; for example, 2 is prime, 3 is prime, 4 is not (because it is divisible by 1, 4, and 2), and so on. The list of primes is infinite, and they take a lot of computer power to find.

Using your text editor, type the program from Listing 3.4 and save it as Primes. Do not type in the line numbers. Make the program executable according to the instructions you learned in Hour 1.

When you’re done, try running the program by typing the following at a command line:

```
perl -w Primes
```

LISTING 3.4  The Complete Source for Primes

```perl
#!/usr/bin/perl -w
$maxprimes=20;  # Stop when you've found this many
$value=1;
$count=0;
while($count < $maxprimes) {
    $value++;
```
HOUR 3: Controlling the Program’s Flow

LISTING 3.4  Continued

8:     $composite=0;
9:     OUTER: for ($i=2; $i<$value; $i++) {
10:         for($j=$i; $j<$value; $j++) {
11:             if (($j*$i)==$value) {
12:                 $composite=1;
13:                 last OUTER;
14:             }
15:         }
16:     }  
17:     if (! $composite) {
18:         $count++;
19:         print "$value is prime\n";
20:     }
21: }

Line 1: This line contains the path to the interpreter (you can change it so that it's appropriate to your system) and the -w switch. Always have warnings enabled!

Line 3: $maxprimes is the maximum number of primes you want to find.
Line 4: $value is the value you're going to test for primeness.
Line 5: $count is the number of primes so far.
Line 6: The while loop continues as long as the program hasn't found enough primes.
Line 7: $value is incremented, so the first number to be checked for prime quality is 2.
Line 8: $composite is a flag used in the for loops to indicate that the number found is composite, not prime.
Lines 9–10: The for loops iterate through all the possible factors of $value. If $value were 4, the loops would produce 2 and 2, 2 and 3, 2 and 3, 3 and 3.
Lines 11–14: The values of $i and $j are multiplied together; if the product is $value, then $value is composite. The flag $composite is set, and both for loops are exited.
Lines 17–20: After the for loops, the $composite flag is checked. If it's false, the number is prime. These lines then print a message and increment the counter.

By the Way

The algorithm used here to find primes isn't particularly speedy or efficient—but it makes for a good demonstration of looping. A better method can be found in a good book on numerical algorithms.
Summary

In this hour, you learned about Perl's many flow control constructs. Some constructs, such as if and the logical operators, are used to control whether portions of the program run, depending on true or false values. Other constructs, such as while, until, and for, are used for looping over pieces of code as many times as necessary. You also learned what Perl's particular idea of truth is, which is used by virtually all test conditions in Perl.

Q&A

Q I'm familiar with another programming language, C, which has a switch (or case) statement. Where is Perl's switch statement?

A Perl doesn't have one! Perl provides such a variety of tests that figuring out the best syntax for a switch statement is nightmarish. The simplest way to emulate a switch statement is as follows:

```perl
if ($variable_to_test == $value1) {
    statement1;
} elsif ($variable_to_test == $value2) {
    statement2;
} else {
    default_statement;
}
```

The online syntax manual page—which you can view by typing `perldoc perlsyn` at a command prompt—contains many clever examples of how to emulate a switch statement in Perl, some with very switch-like syntax.

Q How many for (while, if) blocks can I nest inside each other?

A As many as you like, within memory restrictions of your system. Usually, however, if you have deeply nested loops, it is a sign that you should approach the problem differently.

Q Help! Perl is giving me the message Unmatched right bracket (or Missing right bracket). The line number reported is the end of the file!

A Somewhere in your program, you've used an open brace ({} without a close brace (}), or vice versa. Perl can sometimes guess where the typo is in your program, but sometimes not. Because control structures can nest arbitrarily deeply, Perl doesn't know you've made a mistake until it unexpectedly reaches the End of File without finding the balancing brace. A good program editor (such as vi, Emacs, or UltraEdit) has features to help you find mismatched braces. Use one.
Workshop

Quiz

1. The **while** statement loops as long as a condition is true. What statement loops as long as a condition is false?
   - a. `if (not ) {}`
   - b. `while (! condition) {}`

2. Is the following expression true or false?
   \[(0 \text{ and } 5) \lor (("0" \text{ or } 0 \text{ or ""}) \text{ and } (6 \text{ and } "Hello")) \lor 1\]
   - a. True
   - b. False

3. What is the value of `$i$` after this loop is run?
   ```
   for($i=0; $i<=10; $i++) {   }
   ```
   - a. 10
   - b. 9
   - c. 11

Answers

1. b. The **while** `{}` syntax loops until the condition is false.

2. a. The expression can be reduced in these steps:
   ```
   (false) || (( false ) and ( true )) or true
   false || false or true
   true
   ```

3. c. The test is `$i\leq10$`, so when the test is finally false, `$i$` must be 11. If you got this one wrong, don’t worry. It’s such a common mistake that it even has a special name among programmers: a **fence post error** or an **off-by-one error**.
Activities

- Modify Listing 3.1 to keep playing the game until a successful guess is made.

- Listing 3.3, as it is written, is actually quite inefficient at finding primes. For example, it analyzes all the even numbers above 2—which cannot possibly be prime. Make additions to the algorithm to make the Primes program more efficient.