Loops

Loops provide the mechanism to do simple operations repeatedly. In this section, we'll examine Java's three loops statements: the `while` loop, the `do/while` loop, and the `for` loop. Let's begin with the `while` statement.

while Loop

The syntax for a while loop is

```
while (relational_expression)
   statement;
```

It is more common to see a `while` loop with a statement block:

```
while (relational_expression)
{
   statement1;
   statement2;
   ...
}
```

A `while` statement consists of the reserved word `while` followed by a relational expression in parentheses. The relational expression is used for "loop control". Following the relational expression is a statement or statement block.

Here's how a `while` statement works: First the loop control relational expression is evaluated. If the result is `true` the statement or statement block executes. Then the relational expression is evaluated again. If the result is still `true`, the statement(s) executes again. This continues until the relational expression is evaluated and yields `false`, whereupon the statement(s) is skipped and execution continues after the statement or statement block.

If the relational expression yields `false` the first time it is evaluated, then the statement(s) never executes. If the relational expression yields `true` the first time it is evaluated, then the statement(s) executes at least once. Clearly, the relational expression must eventually yield `false`, otherwise an infinite loop results.

The flowchart for the while statement is shown in Figure 1.

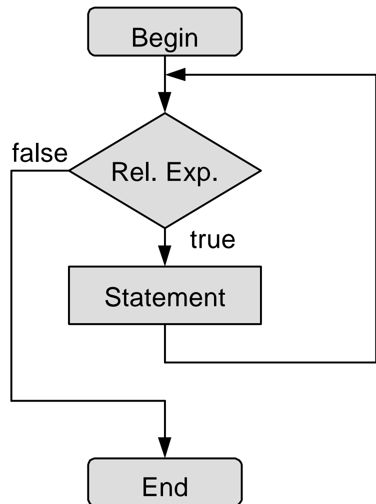

Figure 1. Flowchart for `while` statement

The program `TableOfSquares.java` outputs a table of the squares of integers from 0 to 5 using a `while` statement (see Figure 2).

```
 1  public class TableOfSquares
 2  {
 3     public static void main(String[] args)
 4     {
 5        System.out.print("===============\n"
 6                      +  " x  Square of x\n"
 7                      +  "--- -----------\n");
 8        int x = 0;
 9        while (x < 6)
10        {
11           System.out.println(" " + x + "       " + (x * x));
12           x++;
13        }
14        System.out.println("===============\n");
15     }
16  }
```

Figure 2. `TableOfSquares.java`

This program generates the following output:

```
===============
 x  Square of x
--- -----------
 0       0
 1       1
 2       4
 3       9
 4       16
 5       25
===============
```

The `int` variable `x` is used for loop control. It is declared and initialized to zero in line 8, just before the `while` loop begins. In line 9, the relational expression "`x < 6`" is evaluated. Since `x` was just initialized to zero, the result is `true`, so the statement block in lines 10-13 executes.

A print statement outputs x and x * x with the appropriate spacing for the table to look good. Then, x is increment by one using the postfix increment operator in line 12. Execution then moves back to the loop control expression. x is now 1, so the loop control expression again equals true and the statement block executes again. Eventually x is incremented to 6, and at this point the loop control expression yields false (6 is not less than 6!). Execution then proceeds out of the while loop to line 14, where the bottom rule for the table is printed.

It is, of course, the int variable x that is crucial for the while loop to achieve the desired effect. It is a general rule for loops that something in the loop must alter a variable in the loop control expression, otherwise an infinite loop occurs.

do/while Loop

The main difference between a while loop and a do/while loop is that the statement or statement block executes *at least once* with a do/while loop. This is illustrated in the flowchart in Figure 3.

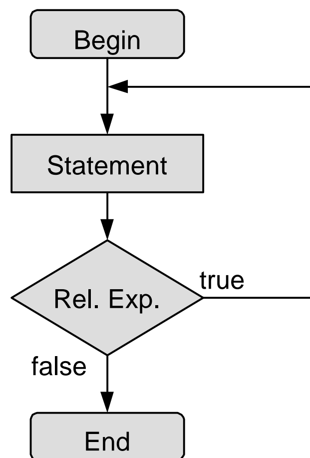

Figure 3. Flowchart for the do/while loop

The syntax for a do/while loop is a little trickier than for a while loop:

```
do
   statement;
while (relational_expression);
```

or

```
do
{
   statement1;
   statement2;
   ...
} while (relational_expression);
```

The loop begins with the reserved word do. This is followed by a statement or statement block. The loop control relational expression appears at the end of the loop in parentheses following the reserved word while. Note that a semicolon is required after the closing parenthesis. A do/while loop cannot be constructed without the closing while.

Figure 4 shows a program using a `do/while` loop to print the message "`Java is fun!`" five times.

```
 1  public class JavaIsFun
 2  {
 3     public static void main(String[] args)
 4     {
 5        int i = 0;
 6        do
 7        {
 8           System.out.println("Java is fun!");
 9           i++;
10        } while (i < 5);
11     }
12  }
```

Figure 4. `JavaIsFun.java`

The `int` variable `i` is declared and initialized with 0 in line 5, just before the `do/while` loop begins. Note that the statements in the `do/while` loop in lines 8 and 9 in execute *at least once*, since the loop control relational expression is not evaluated until line 10. This is the key difference between a `while` loop and a `do/while` loop and it is the primary characteristic that determines which type we choose when setting up a loop.

for Loop

A popular alternative to the `while` loop is the `for` loop. More often than not, a loop uses a *loop control variable*. This variable is *initialized* before the loop begins, it is used in the *loop control* relational expression, and it is *adjusted* at the end of the loop. When these conditions exist, a `for` loop can often do the job. This is illustrated in Figure 5.

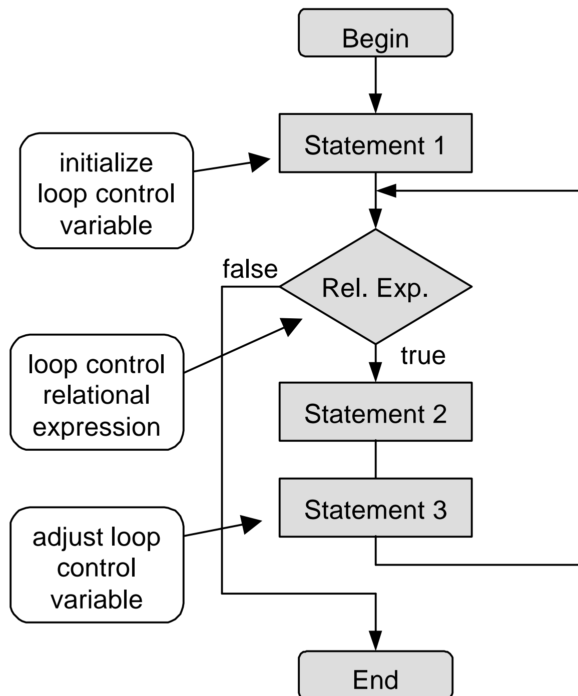

Figure 5. Flowchart for `for` loop

The general syntax for the `for` loop is

```
for (exp1; rel_exp; exp2)
   statement;
```

This is the same as

```
exp1;
while (rel_exp)
{
   statement;
   exp2;
}
```

Expression #1 and expression #2 can be any type of expression, but, most commonly, expression #1 is a statement that initializes a loop control variable, and expression #2 is a statement that adjusts the loop control variable. Within parentheses is the loop control relational expression that determines if the loop is to execute again.

The following `for` loop will print "Java is fun!" five times:

```
int i;
for (i = 0; i < 5; i++)
   System.out.println("Java is fun!");
```

This is equivalent to

```
int i = 0;
while (i < 5)
{
   System.out.println("Java is fun!");
   i++;
}
```

The initialization clause of a `for` loop can include both the declaration and initialization of the loop control variable:

```
for(int i = 0; i < 5; i++)
   System.out.println("Java is fun!");
```

This is a concise and handy way to declare simple loop control variables. This feature is not supported for the relational test clause of `for` or `while` loops. If the loop control variable `i` exists only for the duration of the `for` statement. We will elaborated on this later in our discussion of variable scope.

Sometimes the loop control variable is initialized "high" and then adjusted "backward". This is illustrated in the `StringBackwards` demo program in Figure 6.

```
 1  public class StringBackwards
 2  {
 3     public static void main(String[] args)
 4     {
 5        String s = "Hello, world";
 6        int length = s.length();
 7        for (int i = length; i > 0; i--)
 8           System.out.print(s.substring(i - 1, i));
 9     }
10  }
```

Figure 6. `StringBackwards.java`

The output of this program is

```
dlrow ,olleH
```

In this example, the loop control variable `i` is initialized with the length of the string (lines 6-7). As the loop progresses, `i` is decremented using the postfix decrement operator. The loop terminates when the relational expression "`i > 0`" is `false`. Recall from our discussion of strings in Chapter 2 that the length of a string is "one greater than" the index of the last character in a string. For this reason, the `substring()` method in line 8 retrieves the character starting at `i - 1`, rather than `i`.

Line 6 is not necessary in `StringBackwards.java`, but it was included to avoid clutter in the `for` loop. The loop could just as easily take the following form

```
for (int i = s.length(); i > 0; i--)
   System.out.print(s.substring(i - 1, i));
```

The adjustment expression needn't be limited to a simple increment or decrement of the loop control variable. The following `for` loop counts by threes from zero to one hundred:

```
for (int i = 0; i < 100; i += 3)
   System.out.println(i);
```

The adjustment expression "`i += 3`" increments the loop control variable by three after each print statement.

Loop control needn't be restricted to expressions involving the loop control variable. The following loop prints the cubes of integers starting at zero, increasing until the result approaches but does not exceed 500.

```
int j = 0;
for (int i = 0; j <= 500; i++)
{
   System.out.println(i + "\t" + j);
   j = i * i * i;
}
```

The output is

```
0     0
1     1
2     8
3     27
4     64
5     125
6     216
7     343
```

Since we want to terminate the loop "when the *result* approaches but does not exceed 500" the test does not use the loop control variable, `i`. The loop control relational expression "`j <= 500`" ensures the loop stops when the result – an integer cube – approaches but does not exceed 500. Note the use of the tab character escape sequence (`\t`) to conveniently align the two columns of output.

Comma Operator

It is possible to include more than one expression in the loop initialization and loop adjustment parts of the `for` statement. This is a accomplished using the comma (`,`) operator. The previous example of printing integer cubes could be coded as follows using the comma operator:

```
for (int i = 0, j = 0; j < 500; i++, j = i * i * i)
   System.out.println(i + "\t" + j);
```

Although use of the comma operator is never essential, it occasionally springs to mind as a convenient way to setup a simple loop.

break Statement

We have already met the `break` statement as a means to exit a `switch` after the statements for a selected "case" execute. The break statement also works as an "early" exit from a loop; that is, to exit *before* the loop control relational expression yields "`false`". Suppose we want to process characters in a string starting at the beginning of the string and proceeding through the string. However, assume we want to stop as soon as a period (`.`) is read. The program `DemoBreak` shows how this can be done (see Figure 7).

```
 1  import java.io.*;
 2
 3  public class DemoBreak
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter a line of characters:");
11        String s = stdin.readLine();
12        for (int i = 0; i < s.length(); i++)
13        {
14           if (s.charAt(i) == '.')
15              break;
16           else
17              System.out.print(s.charAt(i));
18        }
19        System.out.println("\nThank you");
20     }
21  }
```

Figure 7. `DemoBreak.java`

A sample dialogue with this program follows:

```
PROMPT>java DemoBreak
Enter a line of characters:
Yes! No. Maybe?
Yes! No
Thank you
```

The `for` loop is setup in line 12 to process the line character-by-character. Within the loop (lines 12-18), the first task is to determine if the character at index `i` is a period (line 14). If so, the `break` statement in line 15 executes and the loop is immediately exited. In this demo, "processing" is the simple act of printing each character (line 17).

A `switch` statement cannot be setup properly without using `break` statements. However, as a means to exit a loop early, a `break` statement is never essential. A workaround is always possible. For example, lines 12-18 in `DemoBreak.java` can be replaced with

```
for (int i = 0; i < s.length() && !(s.charAt(i) == '.'); i++)
   System.out.print(s.substring(i, i + 1));
```

This approach saves a few lines of code, but it is also harder to understand. Note the use of lazy evaluation in the loop control relational expression to ensure that the `charAt()` method does not execute unless the index `i` represents a valid character in the string.

There is an on-going debate in the computer science community on whether the `break` statement is good news or bad news. As it turns out, programs that use `break` statements to exit loops are extremely difficult (arguably "impossible") to verify for correctness. In very large, safety-critical projects, it is probably best to avoid the `break` statement. However, in our ever-expanding bag of Java programming tools, the `break` statement is often the best "fit" for our mental model of a problem. We will continue to use `break` statements in these notes.

By definition, the `break` statement exits the inner-most loop or `switch` where it is located. If a `break` statement is located in a `switch` statement that is inside a loop, it serves only to exit the

switch statement. If a break statement is inside a loop that is inside another loop, it serves only to exit the inner loop.

continue Statement

The continue statement provides a mechanism to skip statements in a loop and proceed directly to the loop control relational expression. The program DemoContinue inputs a line of characters and then processes (i.e., prints) only the digit characters in the line (see Figure 8).

```
 1  import java.io.*;
 2
 3  public class DemoContinue
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter a line of characters:");
11        String s = stdin.readLine();
12        for (int i = 0; i < s.length(); i++)
13        {
14           if (s.charAt(i) < '0' || s.charAt(i) > '9')
15              continue;
16           System.out.print(s.charAt(i));
17        }
18        System.out.println("\nThank you");
19     }
20  }
```

Figure 8. DemoContinue.java

A sample dialogue follows:

```
PROMPT>java DemoContinue
Enter a line of characters:
Process only 0123456789 characters
0123456789
Thank you
```

The character is processed (i.e., printed) character by character; however, line 14 determines if the current character is a digit. If it is not, the continue statement in line 15 executes to immediately begin processing the next character.

A continue statement can be avoided by reversing the relational test and adding another level of indentation to the statement(s) that follows. Lines 14-16 in DemoContinue can be replaced with

```
if ( !(s.charAt(i) < '0' || s.charAt(i) > '9') )
   System.out.print(s.charAt(i));
```

The ! (NOT) logical operator reverses the relational test. Note that the print statement above is indented farther than the same statement in line 16 in Figure 8. The difference is minor; however, if the print statement is replaced by a large statement block, the cosmetic effect of the added indentation may be undesirable. The continue statement can help prevent nested loops, etc., from creeping too far to the right, with lines extending off the viewable area of the editor's display.

For the most part `continue` is used simply because it is the most convenient tool at hand to solve a particular programming problem. If it suits your purpose, by all means use it.

Note that a `continue` statement has no particular use in a `switch` statement (unlike `break`). If a `continue` statement does appear in a `switch` statement, execution proceeds to the loop control relational expression of the inner-most loop containing the `switch`.

Infinite Loops

Infinite loops were mentioned earlier as a possible side effect of incorrectly setting up a loop control relational expression. In these situations, the infinite loop is definitely a programming bug, because the program never terminates. However, infinite loops can be exploited as a simple way to set-up a loop that executes indefinitely until some condition occurs.

The program `DemoInfiniteLoop` reads lines from the standard input and echoes the lines to the standard output. This continues in a loop until a line beginning with 'Q' is inputted (see Figure 9).

```
 1  import java.io.*;
 2
 3  public class DemoInfiniteLoop
 4  {
 5     public static void main(String[] args) throws IOException
 6     {
 7        BufferedReader stdin =
 8           new BufferedReader(new InputStreamReader(System.in), 1);
 9
10        System.out.println("Enter lines of characters:\n" +
11                           "(Line beginning with 'Q' quits)");
12        while (true)
13        {
14           String s = stdin.readLine();
15           if (s.length() > 0 && s.charAt(0) == 'Q')
16              break;
17           System.out.println(s);
18        }
19        System.out.println("Thank you");
20     }
21  }
```

Figure 9. `DemoInfiniteLoop.java`

A sample dialogue with this program follows:

```
PROMPT>java DemoInfiniteLoop
Enter lines of characters:
(Line beginning with 'Q' quits)
Hello
Hello
Good bye
Good bye
Quit
Thank you
```

This program uses a `while` loop of the following form:

```
while (true)
   statement;
```

Since the boolean constant "`true`" is always "`true`", the loop is an infinite loop. The trick is to use a `break` statement to exit the loop when a certain condition occurs. In this example, that condition is a line beginning with 'Q'. Note the use of lazy evaluation in line 15 to ensure that the `charAt()` method doesn't execute if a blank line is inputted.

Setting up infinite loops, as above, is simple and effective. As with `break`, `continue`, and the comma operator, there is always an alternate way to create the same effect. Of course, if you are of the conviction that `break` statements are bad news, then you are unlikely to buy-into the deliberate use of infinite loops, as above.

For completeness, let's examine two alternatives to the `while` loop in `DemoInfiniteLoop`. Lines 12-18 can be replaced with the following statements:

```
boolean moreData = true;
while (moreData)
{
   String s = stdin.readLine();
   if (s.length() > 0 && s.charAt(0) == 'Q')
      moreData = false;
   else
      System.out.println(s);
}
```

In this approach a `boolean` variable `moreData` is used as a "flag" to signal the end of input. The initial state of `moreData` is `true`, so the loop executes at least once. When a line is inputted that begins with 'Q' `moreData` is set to `false`. The next time the loop control relational expression is evaluated, the result is `false` and the loop terminates.

For the ultimate in dense code, the same lines can be further reduced as follows:

```
String s;
while ((s = stdin.readLine()).length() != 0 && !(s.charAt(0) == 'Q'))
   System.out.println(s);
```

The `while` loop now includes a large loop control relational expression containing an assignment expression, `(s = stdin.readLine())`. This is perfectly legal; however, since assignment has the lowest precedence, parentheses are added. The result is simply the string assigned and, so, it is also perfectly legal to append "`.length()`" to determine the length of the string. Further appending "`!= 0`" simply implements a relational expression which is `true` as long as the line inputted is not a blank line. This is followed by the logical `&&` operator. Lazy evaluation ensures that the right-hand argument is not evaluated unless a non-blank line was inputted. If indeed the line was non-blank, the expression `!(s.charAt(0) == 'Q')` further ensures that the line does not begin with 'Q'. If the final result of this mega-expression is `true`, the inputted line is printed on the standard output in the next line — and then we start all over again!

We have just examined three ways to solve the same problem. The first used an infinite loop with a break statement. The second used a `boolean` flag to exit a loop. The third used a one-statement loop with a very dense loop control relational expression. So, which of these is "the best". If you are using these notes as a student in a Java programming course, your instructor will, no doubt, have an opinion on this. Your best bet is to adopt the style suggested by your

instructor. In fact, subscribing to, and sticking with, a consistent programming philosophy is the best advice we can offer.

Variable Scope

Earlier, we noted that a variable can be declared anywhere, provided it is declared before it is used. This seems simple enough, and it is, as long as programs execute in a straight line. Now, our programs include choices and loops. Due to a characteristic known as *variable scope*, a bit more attention is needed on the placement of variable declarations when loops are involved. Once a variable is defined it exists from that point on — *but only within the block where it is defined*. This point is illustrated using a simple demo program (see Figure 10).

```
 1  public class DemoVariableScope
 2  {
 3     public static void main(String[] args)
 4     {
 5        String advice = "Haste makes waste";
 6        int i;
 7        for (i = 0; i < 2; i++)
 8           System.out.println(advice);
 9        System.out.println("Good advice is worth giving...");
10        System.out.println(i + " times");
11     }
12  }
```

Figure 10. `DemoVariableScope.java`

This program generates the following output:

```
Haste makes waste
Haste makes waste
Good advice is worth giving...
2 times
```

The "2" in the final line of output is the value of the `int` variable `i`. Now, have a look at line 6 in the listing. Note that the `int` variable `i` was declared *before* the `for` loop. This is important. If `i` were defined inside the initialization of the `for` loop, as commonly done, a compile error will occur at line 10, because `i` only exists for the duration of the loop. The error message will indicate "`Undefined variable: i`". This is illustrated in Figure 11.

```
for (int i = 0; i < 2; i++)
   System.out.println(advice);
System.out.println("Good advice is worth giving...");
System.out.println(i + " times");
```


Figure 11. Variable scope example

If a variable is not needed after the loop, then the approach in Figure 11 is fine.

For nested loops, even more care is warranted. The same rule applies, but greater opportunity for error exists. An example of variable scope with nested loops is shown in Figure 12.

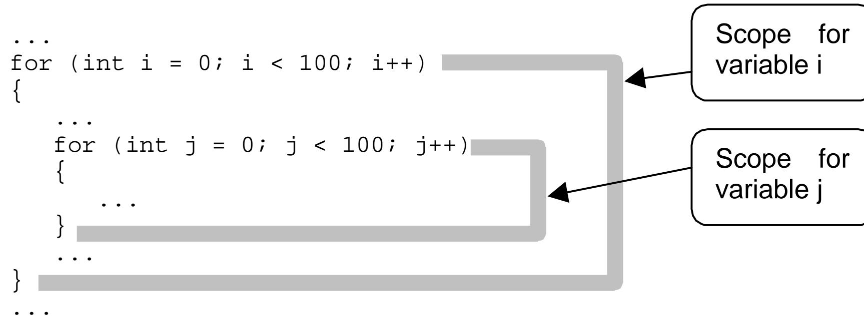

Figure 12. Variable scope for nested loops

This completes our discussion of "choice" and "loops" in the Java language. In summary,

- "Choices" may be implemented using the `if` statement or the `switch` statement.
- "Loops" may be implemented using the `while`, `do/while`, or `for` statements.
- A `break` statement is used to immediately exit a `switch` or a loop.
- A `continue` statement inside a loop causes an immediate branch to the loop control relational expression.
- A variable has scope, or visibility, only in the block where it is defined.