Chapter 4 - Loops

Sections	Pages	Review Questions	Programming Exercises
4.1-4.7, 4.9-4.10	104-117, 122-128	2-9, 11-13,15-16,18-19,21	2,4,6,8,10,12,14,18,20,24,26,28,30,38

Loops

Loops are used to make a program do something over and over, until some condition is met. For instance, we may want to read and add test scores from the user *until a 0 is entered*. There are three kinds of loops in Java: for, while, and do-while.

while Loop

1. The syntax for a *while* loop is:

```
while ( continuation_condition )
{
   // The body of the loop
   statement(s);
}
```

Operation is simple, the *while loop* is like an *if* block that repeats over-and-over until the condition is false. When the *while* statement is encountered for the first time, the *continuation_condition* expression is evaluated. If it is *true*, the *body* of the loop is executed. This is referred to as an *iteration*. Next, execution returns to the *while* statement and the *continuation_condition* expression is evaluated again. This process repeats until the *condition_condition* is *false*. Thus, there must be some way to change the value of the *condition_condition* inside the body of the loop; otherwise, the loop would run forever. The number of times the code in the body of the loop executes is referred to as the number of *iterations*. We will see examples of these things shortly.

So, the *continuation_condition* must be *true* for the loop to execute. When it changes to *false*, then the loop will terminate.

2. Example – What value does this program print out? How many iterations did this loop make? What makes this loop end? Now, suppose x=10 initially (instead of 0). Answer these questions again.

```
int x = 0;

while ( x <= 3 )
{
   x++;
}

System.out.println( x );
```

3. An *infinite loop* occurs when the condition is , *i.e.* a loop that never terminates. Thus, we must make sure that something inside the loop will eventually change the *continuation_condition* to false.

   This is an example of an infinite loop, why?

```
int x = 0;

while ( x < 10 )
     System.out.println( x );
```

   Sometimes, this infinite loop is useful:

```
while ( true )
{
   // Do something
}
```

   However, this loop generates a compile error, “unreachable statement”. In other words, the compiler has detected that the while loop is superfluous.

```
while ( false )
{
   // Do something
}
```

4. Example: Write a program that generates a random integer between 0 and 100. Have the user guess the number until they figure it out. Provide informative messages along the way to indicate if their guess is too high or too low.

   a. Algorithm:

```
Generate a random number
While number has not been guessed correctly
     Read guess
     If guess = number
          Print success message
          Make sure loop terminates
     Else If guess < number
          Print “too low” message
     Else
          Print “too high” message
     End  If
End While
```

b. The code, also in the text is shown below:

```
int guess = 0;
int number = (int)(Math.random()*101);
boolean correctGuess = false;

while ( guess != number )
{
   System.out.print( "Guess a number between 0 and 100: " );
   guess = scanner.nextInt();

   if ( guess == number )
      System.out.println( "You got it!” );
   else if ( guess < number )
      System.out.println( "Your guess is too low!" );
   else
      System.out.println( "Your guess is too high!" );
}
```

Another way to terminate the loop above is:

```
int guess = 0;
int number = (int)(Math.random()*101);
boolean correctGuess = false;

while ( ! correctGuess )
{
   System.out.print( "Guess a number between 0 and 100: " );
   guess = scanner.nextInt();

   if ( guess == number )
   {
      correctGuess = true;
      System.out.println( "You got it!” );
   }
   else if ( guess < number )
      System.out.println( "Your guess is too low!" );
   else
      System.out.println( "Your guess is too high!" );
}
```

Suppose we want to print out how many guesses it took. How would we do this?

while Loop – Sentinel Value

1. Sometimes, especially when we are reading values from a user, we use a special value to signal the end of the loop. In other words, we may have the user enter -1 to indicate that they don't have any more data to enter. This is called a *sentinel value*.
2. Example: Write a program to read test scores from the console. The user will enter -1 when they are finished entering tests. Compute and display the average of the test scores.
   a. Algorithm:

```
While user did not enter -1
    s = read score
    If score is not -1 then
        sum += s;
        count++;
    End If
End While
avg = sum / count
Print avg
```

   b. Code:

```java
int score = 0;
double sum = 0;
int count = 0;

while ( score != -1 )
{
   System.out.print( "Enter a test score or -1 to quit" );
   score = scanner.nextInt();

   if ( score != -1 )
   {
      count++;
      sum += score;
   }
}
double average = sum / count;
System.out.println( average );
```

What happens when the user enters -1 immediately without entering any test scores? How would we fix this program?

c. Another way to write the code:

```
System.out.print( "Enter a test score or -1 to quit: " );
score = s.nextInt();

while ( score != -1 )
{
   count++;
   sum += score;

   System.out.print( "Enter a test score or -1 to quit: " );
   score = s.nextInt();
}

if ( count > 0 )
{
   double average = sum / count;
   System.out.println( average );
}
else
   System.out.println( "No test scores entered" );
```

do-while Loop

1. The syntax for a *do-while* loop is:

```
do
{
   statement(s);

} while ( continuation_condition )
```

   This is the same as the while loop, except the *continuation_condition* is at the end of the loop instead of the beginning. Thus, the body of a *do-while* loop always executes at least once.

   When should you use a *while* loop or a *do-while* loop. First, mostly, people use a *while* loop over the *do-while* just because it feels natural in the context of the problem. However, you can always use either approach. Sometimes, however, a *do-while* loop is the *natural* thing to do. For instance, when we have some code (the body of the loop) that must always be executed at least once.

2. Example: Prompt the user for an amount of money. Compute the amount of money at the end of one year assuming 7% interest. Then, ask the user to enter 1 if they want to compute the value for another year or 0 if they want to quit.

```
double value = 0.0;
int years = 0;
int again = 0;

System.out.print( "Enter an amount of money: " );
value = s.nextDouble();

do
{
    years++;
    value *= 1.07;

    System.out.printf( "After %d years you have $%.2f\n", years, value );
    System.out.print( "Compute for another year (1=yes, 0=no)? " );
    again = s.nextInt();

} while ( again == 1 );
```

3. Example: What is the output of this code? Suppose x=10?

```
int x = 0;

while ( x < 3 )
   System.out.println( x++ );

do
   System.out.println( x++ );
while ( x < 3 );
```

for Loop

1. A *for* loop is useful when you know (or can compute) in advance how many times you want a loop to iterate. A simplified syntax for a *for* loop is:

```
for ( beginning; end; increment )
{
   // The body of the loop
}
```

2. For example:

```
for( int i=0; i<5; i++ )
   System.out.println( i );
```

How many iterations does this loop execute? What is the *scope* of *i*?

How would you write this loop with a *while* loop?

```
int i=0;
while ( i<5 )
   System.out.println( i++ );
```

A *do-while* loop?

```
int i=0;
do
   System.out.println( i++ );
while ( i<5 );
```

3. A little more careful way to describe the syntax of the *for* loop is (can be generalized further) shown below. In most situations, this version is appropriate.

```
for ( index_initial_value; continuation_condition; index_increment )
{
   // The body of the loop
}
```

4. Example, print out a table of the values of $y = 2x^2 - 4x + 1$ for the positive integer values of x through 10.

```
for( int x=1; x<=10; x++ )
   System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
```

Output

```
1, -1.0
2, 1.0
3, 7.0
4, 17.0
...
```

5. Example, print out a table of the values of $y = 2x^2 - 4x + 1$ for the positive integer values of x from 10 down to 1.

```
        for( int x=10; x>=1; x-- )
           System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
```

Output

```
    10, 161.0
    9, 127.0
    8, 97.0
    7, 71.0
    ...
```

6. Example, print out a table of the values of $y = 2x^2 - 4x + 1$ for the positive, odd, integer values of x from 1 to 11.

```
        for( int x=1; x<=11; x+=2 )
           System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
```

Output

```
    1, -1.0
    3, 7.0
    5, 31.0
    ...
```

7. Example, print out a table of the values of $y = 2x^2 - 4x + 1$ for x=1 to 3 by increments of 0.1

```
        for( double x=1; x<=2; x+=0.1 )
           System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
```

Output:

```
    1.0, -1.0
    1.1, -0.98
    1.2000000000000002, -0.9199999999999999
    1.3000000000000003, -0.8199999999999998
    1.4000000000000004, -0.6799999999999993
    1.5000000000000004, -0.4999999999999991
    1.6000000000000005, -0.2799999999999985
    1.7000000000000006, -0.019999999999998685
    1.8000000000000007, 0.280000000000002
    1.9000000000000008, 0.6200000000000028
```

Why doesn't it display for x=2.0? Warning: need to be very careful if using decimal numbers in the statements for the *for* loop.

8. A solution to the preceding problem:

```
double x;
for( int i=1; i<=11; i++ )
{
   x = 1.0 + (i-1)*0.1;

   System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
}
```

Output:

```
1.0, -1.0
1.1, -0.98
1.2, -0.9199999999999999
1.3, -0.8199999999999998
1.4, -0.6800000000000002
1.5, -0.5
1.6, -0.27999999999999936
1.7000000000000002, -0.019999999999999574
1.8, 0.28000000000000025
1.9, 0.6200000000000001
2.0, 1.0
```

Another way to write the loop:

```
for( int i=0; i<=10; i++ )
{
   x = 1.0 + i*0.1;

   System.out.println( x + ", " + (2*Math.pow(x,2)-4*x+1) );
}
```

9. Example, print out a table of the values of $y = 2x^2 - 4x + 1$ between two values specified by the user using increments of 1.

```
System.out.print( "Enter beginning value: " );
begin = scanner.nextInt();
System.out.print( "Enter ending value: " );
end = scanner.nextInt();

for( int i=begin; i<=end; i++ )
   System.out.println( i + ", " + (2*Math.pow(i,2)-4*i+1) );
```

What happens when you enter 3 & 1, respectively?

Nested Loops

1. We can have loops within loops. For example:

```
int k;

for( int i=0; i<3; i++ )
{
   for( int j=0; j<3; j++ )
   {
      k = i*j;
      System.out.println( k );
   }
}
```

Trace of program

i	j	k
0	0	0
	1	0
	2	0
1	0	0
	1	1
	2	2
2	0	0
	1	2
	2	4

Output

```
0
0
0
0
1
2
0
2
4
```

2. Example:

```
for( int i=0; i<3; i++ )
{
  for( int j=i; j<3; j++ )
   {
      k = i*j;
      System.out.println( k );
   }
}
```

Trace of program

i	j	k
0	0	0
	1	0
	2	0
1	1	1
	2	2
2	2	4

Output

```
0
0
0
1
2
4
```

3. Example, Multiplication Table 1:

```
for( int i=1; i<4; i++ )
{
  for( int j=1; j<4; j++ )
   {
      k = i*j;
      System.out.print( k + " " );
   }
   System.out.println();
}
```

Output:

```
1 2 3
2 4 6
3 6 9
```

4. Example, Multiplication Table 2

```
System.out.print( "    " );

for( int i=1; i<4; i++ )
   System.out.print( i + " " );

System.out.println();
System.out.println( "----------" );

for( int i=1; i<4; i++ )
{
   System.out.print( i + " | " );

   for( int j=1; j<4; j++ )
   {
      k = i*j;
      System.out.print( k + " " );
   }
   System.out.println();
}
```

Output:

```
    1 2 3
----------
1 | 1 2 3
2 | 2 4 6
3 | 3 6 9
```

5. Example, Multiplication Table 3. Same as previous.

```
String multTable;

multTable = "    ";

for( int i=1; i<4; i++ )
   multTable += i + " ";

multTable += "\n----------\n";

for( int i=1; i<4; i++ )
{
   multTable += i + " | ";

   for( int j=1; j<4; j++ )
   {
      multTable += (i*j) + " ";
   }
   multTable += "\n";
}

System.out.println( multTable );
```

6. Example. Suppose we want to print out this table using loops:

```
1 2 3 4 5
  1 2 3 4
    1 2 3
      1 2
        1
```

   a. Analysis/Design

      In a situation like this we need nested for loops. The *outer* loop will iterate over the *rows*. The *inner* loop will iterate over the columns.

      Notice that there are 5 rows. This means that we need an outer loop like this:

```
for( int row=1; row<=5; row++ )
{
    // Print row
}
```

      Now, think in terms of an algorithm. What do we need to do for each loop?

```
For each row
    Print spaces
    Print numbers
```

Since the number of spaces is different for each row, and the number of numbers is also different for each row, we can use two *inner* loops, one to print spaces and one to print the numbers. Notice that each number takes up two spaces (the number and a space). So, let's refer to 2 spaces as a *blank.*

```
For each row
    For i=1 to number of blanks
            Print blank
    For i=1 to end number
            Print number
```

Now, the tricky part: we need to figure out the *number of blanks* and the *end number*. Look carefully at the example table that was given in the problem statement above and then consider this table:

Row	Blanks	Ending Number
1	0	5
2	1	4
3	2	3
4	3	2
5	4	1

Now, the really tricky part. We need an *expression* for the number of blanks so that we can use it as a *continuation_condition* for the first inner loop. Consider: *Blanks=Row-1*. Verify this in the table above.

Now, we need to do the same thing for the ending number. Consider: *Ending Number = 5 – (Row-1)*. Verify this in the table above.

Finally, our algorithm looks like this:

```
For row=1 to 5
        For col=1 to row-1
                Print Blank
        For col=1 to 5-(row-1)
                Print col
```

b. Code

```
// Loop over each row
for( int row=1; row<=5; row++ )
{
   // Print blanks
   for( int col=1; col<=row-1; col++ )

      System.out.print( "  " );


   // Print numbers
   for( int col=1; col<=5-(row-1); col++ )

      System.out.print( col + " " );

   System.out.println();
}
```

break Keyword

1. The *break* keyword is used to break out of a loop (or switch statement). It transfers control to the next executable line after the loop. This should only be used when it makes the code easier to understand. You can always write code that doesn't require a *break*.
2. Example – Suppose j=1, what is the output? j=2? j=5?

```
int sum = 0;

while( j < 6 )
{
   sum += j;

   if( sum > 8 ) break;

   j++;
}
System.out.println( j + ", " + sum );
```

continue Keyword

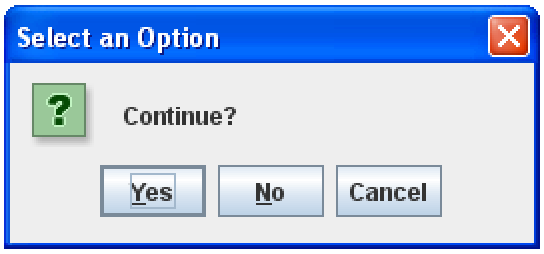
1. The *continue* keyword is used to break out of the current iteration of a loop (or switch statement). It transfers control to the end of the loop. This should only be used when it makes the code easier to understand. You can always write code that doesn't require a *continue*.
2. Example. Suppose j=1, what is the output? j=2?

```
sum = 0;

while( j < 6 )
{
   sum += j;

   if( j==2 || j==3 )
   {
      j+=2;
      continue;
   }

   j++;
}
System.out.println( j + ", " + sum );
```

Confirmation Dialog

1. A *confirmation dialog* is a GUI element that allows the user to choose, Yes, No, or Cancel:

```
JOptionPane.showConfirmDialog( null, "Continue?" );
```



Continue?

2. Usually we *catch* the response from the dialog so that we can make a decision in our program

```
int option;

option = JOptionPane.showConfirmDialog( null, "Continue?" );

if( option == JOptionPane.YES_OPTION )
{
   // Do something
}
else if ( option == JOptionPane.NO_OPTION )
{
   // Do something
}
else if ( option == JOptionPane.CANCEL_OPTION )
{
   // Do Something
}
```

3. A *framework* for allowing the user to run your program again:

```
do
{
    // Do something

    option = JOptionPane.showConfirmDialog( null, "Run again?" );

} while ( option == JOptionPane.YES_OPTION );
```

4. Example – Allow user to enter test scores until -1 is entered. Display the average and see if the user wants to run the program again.

```
int score = 0;
double sum = 0;
int count = 0;

do
{
    System.out.print( "Enter a test score or -1 to quit: " );
    score = s.nextInt();

    while ( score != -1 )
    {
        count++;
        sum += score;

        System.out.print( "Enter a test score or -1 to quit: " );
        score = s.nextInt();
    }

    if ( count > 0 )
    {
        double average = sum / count;

        System.out.println( average );
    }
    else
    {
        System.out.println( "No test scores entered" );
    }

    option = JOptionPane.showConfirmDialog( null, "Run again?" );

   } while ( option == JOptionPane.YES_OPTION );
```

There is a logic error in this program. What is it?

PowerPoint

Cover PowerPoint slides for Chapter 4 in class.

More Examples

If time permits, another handout will be provided that gives additional problems we will work in class.