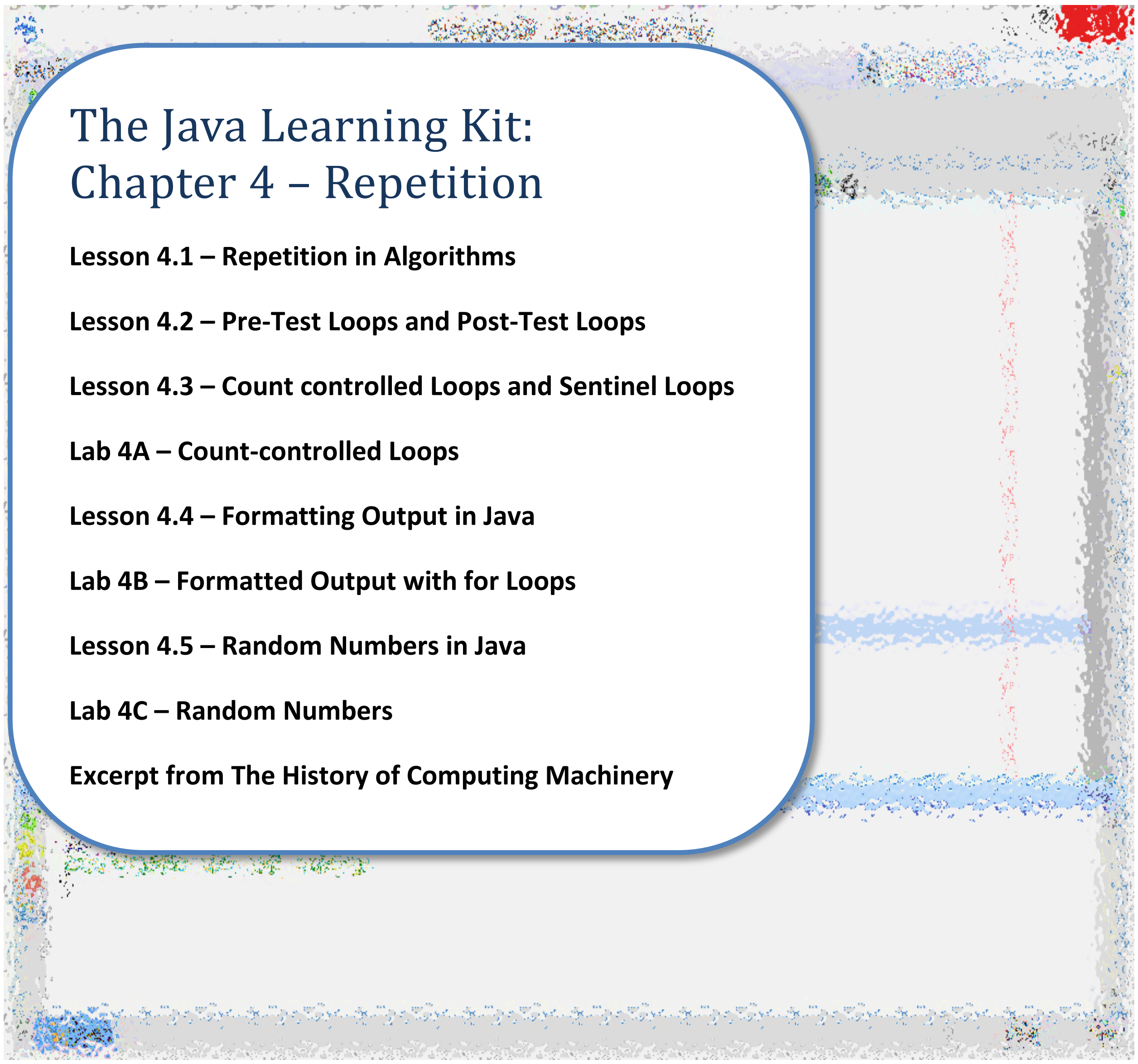


The Java Learning Kit: Chapter 4 – Repetition

Contents

Chapter 4 – Repetition

This chapter examines repetition in algorithms and in Java programming, which is also known as looping.

The difference between pre-test loops with the test for repetition at the beginning of the loop and post-test loops with the test for repetition at the end of the loop is discussed. The difference between loops that are count-controlled and loops that are not count-controlled is also discussed.

Formatted output is presented, particularly for use in loops that create tables of data.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

- describe the concept of repetition in an algorithm;
- describe the difference between a pre-test loop and a post-test loop and why a pre-test loop is often preferred;
- list and describe the four major components of a pre-test loop in a computer program;
- describe the difference between a count-controlled loop and a sentinel loop, and the three pieces of information needed for the counter in a count-controlled loop;
- create Java code that includes pre-test loops, post-test loops, count-controlled loops, sentinel loops, and nested loops using the *while, do…while,* and *for* statements appropriately;
- describe the use of format Strings in Java's System.out.format() and System.out.printf() methods and write code to format output using them;
- describe the difference between a deterministic pseudo-random number
- generator and a true random number generator;
- write Java code to generate random numbers using the math.Random method, including both random integers and random floating point numbers within specific ranges.

Lesson 4.1 Repetition in Algorithms

Repetition occurs whenever part of an algorithm is repeated, forming a loop in the algorithm. The **loop** is a *repetition sequence*.

In Chapter 3 we saw conditional execution – algorithms that branched into different paths based on some condition in the form of a Boolean expression. A loop is created when an algorithm branches backward following such a condition, as shown by the pseudocode and flowchart below.

```
start

x = 1;

while ( x ≤ 10) {
    print x;
    x++;
    }

stop
```

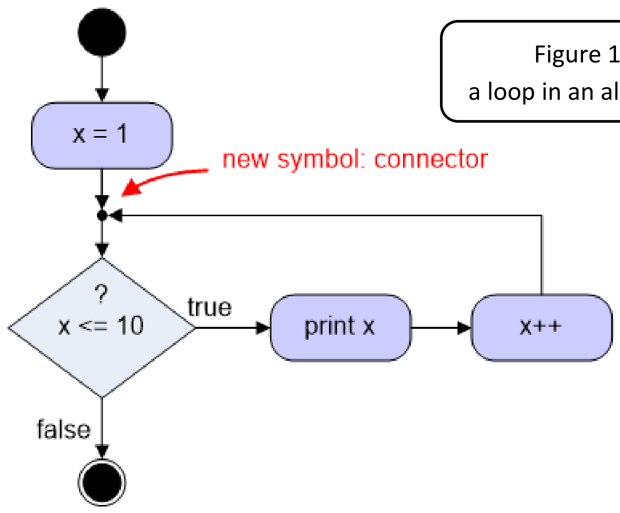


Figure 1 –
a loop in an algorithm

The Boolean condition in this algorithm is (x <= 10). The condition appears following the word *while* instead of the word *if* that is used in binary branching. In pseudo-code, as in many programming languages, the word while indicates that the path of the algorithm returns to the conditional expression when the block of code following the *while* statement is finished being executed. This use of the word *while* originated in the Algol language, so it is common in Algol-derived languages such as *C*, C+, Java, Python, Ruby, and Swing.

A **while statement** forms a test to see if a loop should be executed. Each time the Boolean condition in the while statement is true, the computer will execute the block of code, then go back to the condition again. When the condition is no longer true, the block of code will be ignored and the computer will move on to whatever comes next in the algorithm.

It is possible that the while condition could be false the first time it is encountered, which means the loop will not be executed in this run through the algorithm. It is also possible that the while condition will always be true, which means the loop will always be executed, forming an infinite loop.

A small round circle is used on a flowchart to show where the path that forms the loop rejoins the previously executed part of the algorithm. This symbol is called a *connector*, and is only used to show where separate paths of logic in an algorithm come together.

The algorithm illustrated here prints the numbers from 1 to 10. Before the loop starts, x is set to 1. If the condition *(x ≤ 10)* is true, the value of x is printed, x is incremented by 1, then the condition is checked again. This continues until the condition is false – until x is no longer less than or equal to 10.

The last time through the loop, 10 will be printed, then x is set to 11. When the condition is checked this time around, the loop terminates because the condition is now false, 11 is not less than or equal to 10.

A while statement is not the only statement used for branching. Just as there are several different kinds of branching routines, there are several different kinds of loops. The loop shown in Figure 1 on the previous page is an example of a *count-controlled pre-test* loop. Loops are either *count-controlled* loops or *sentinel* loops. They are either *post-test* loops or *pre-test* loops. We will examine these differences in this chapter.

Loop Control Variables

As we just saw, a condition is tested each time through a loop to determine whether to repeat the loop again or terminate the loop and move on. The condition should be either a boolean expression or a function that returns a Boolean value. If the condition is a Boolean expression, it should include at least one loop control variable. A **loop control variable** is a variable used in a boolean expression to determine if a loop should be repeated. Sometimes the acronym LCV is used to refer to a loop control variable. Any type of variable may be used in a loop control variable, as long as the overall expression evaluates to a Boolean value, as in the condition from above, (X<=10).

Here are some examples of while statements with Boolean expressions containing loop control variables using Java syntax:

- while (count < 10)   // count is an integer.
- while (!(name == "END"))   // name is a String.
- while (age >= 18.0 && age <= 65.0)   // age is a double. This checks for a range of values.
- while (citizen)   // citizen is a Boolean variable.
- while (angle == Math.PI)   // angle is a double. This is a troublesome condition.
- while (Math.abs(angle – Math.PI) < .001) // angle is a double variable. This works better.

Why is *(angle == PI)* troublesome? Tests for equality of floating point variables can be troublesome because any floating point value could be rounded off differently, so two floating point values may not be exactly equal. This is especially true for irrational numbers, which cannot be expressed as a ratio of integers -- as a fraction with an integer numerator and an integer denominator. 3.141592 does not equal 3.14159. Instead of testing for equality, we can test to see if they are really close to one another, as in the last condition above: ***(Math.abs(angle – Math. PI) < .001)*** The absolute value function is used in the condition to see if the difference between *angle* and π is less than .001. It tests to see if *angle* is really close to π. We can make this test as accurate as needed. In general, this is a good way to compare floating point values that may be irrational or that may be rounded off for some other reason.

A loop may have more than one loop control variable, such as a loop with the following while statement:

while (all the doors are not painted AND we still have paint left)

In programming terms, the statement might look more like this:

```
while ( (unpaintedDoorCount > 0) && (amountOfPaint > 0) )
```

Both *unpaintedDoorCount* and *amountOfPaint* are loop control variables. The loop will only repeat if both parts of the condition are true.

Here is an example using an OR condition:

while (age >= 12 OR height >= 60)

In programming terms, the statement might look more like this:

```
while ( (age >= 12) || (height >= 60) )
```

Both *age* and *height* are loop control variables. In this case, the loop will repeat if either of the two parts of the condition is true.

In the next few sections we will look at the different kinds of loops and how to implement them in Java.

CheckPoint 4.1

1. How does a *while* statement in Java compare to an *if* statement?
2. How is a loop control variable used in a while statement?
3. What type of variable can be used as a loop control variable?
4. Why are conditions such as (x == Math.sqrt(2) troublesome?
5. How can we test to see if the values of two floating point variables are very close to one another?

Lesson 4.2 Pre-Test Loops and Post-Test Loops

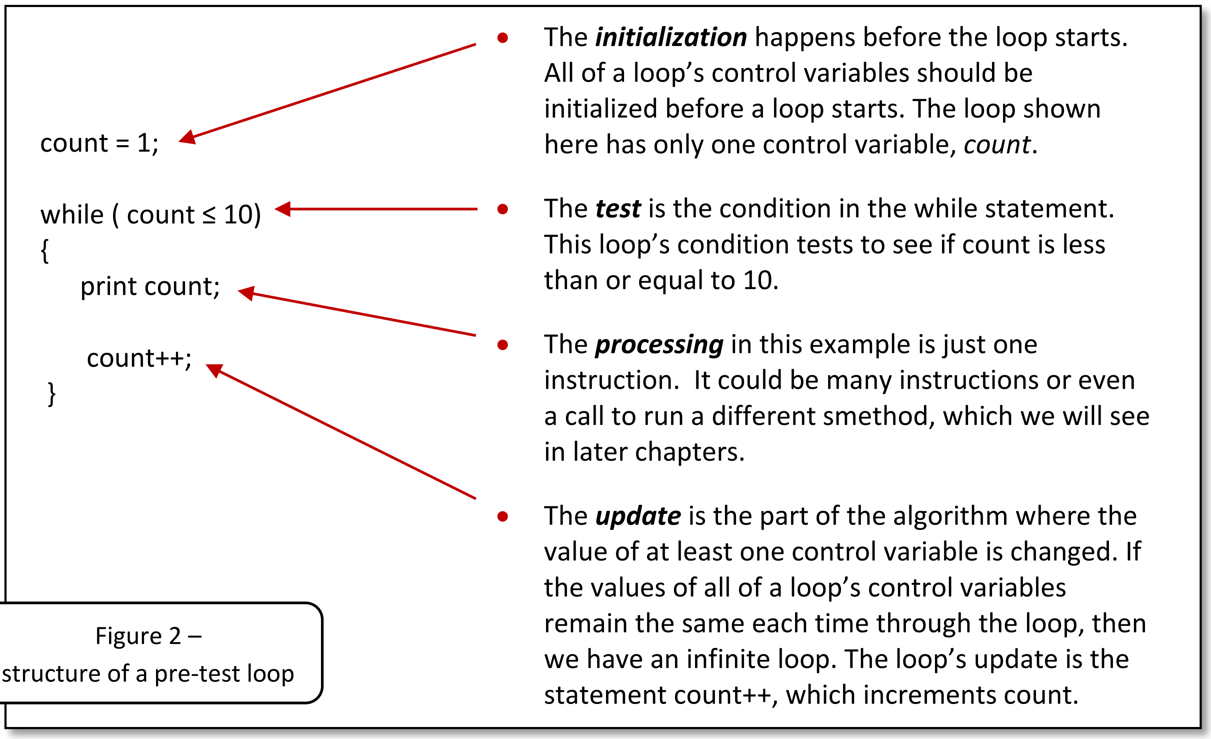
The test to determine if part of an algorithm should be repeated could occur at the beginning of a loop or at the end of a loop. If the test occurs at the beginning of a loop, the loop is known as a *pre-test loop*. If the test occurs at the end of the loop, the loop is known as a *post-test* loop. In this section we will look at the structure of pre-test loops and post-test loops.

Pre-Test Loops

A **pre-test loop** is a loop in which the test to determine whether or not to go through the loop comes before the process that is to be executed within the loop. There are four parts to every pre-test loop:

- **Loop Initialization:** set the first value of the loop control variable
- **Loop Test:** test the loop control variable to see if the loop should be executed
- **Loop Processing:** instructions defining the part of the algorithm to be repeated
- **Loop Update:** change the value of at the loop control variable

The **body of a loop** includes all of the instructions in the loop's block of code – the *processing* and the *update*. The four parts of the loop are identified in the following pseudocode:



```
count = 1;

while ( count ≤ 10)
{
    print count;

     count++;
 }
```

- The ***initialization*** happens before the loop starts. All of a loop's control variables should be initialized before a loop starts. The loop shown here has only one control variable, *count*.
- The ***test*** is the condition in the while statement. This loop's condition tests to see if count is less than or equal to 10.
- The ***processing*** in this example is just one instruction. It could be many instructions or even a call to run a different smethod, which we will see in later chapters.
- The ***update*** is the part of the algorithm where the value of at least one control variable is changed. If the values of all of a loop's control variables remain the same each time through the loop, then we have an infinite loop. The loop's update is the statement count++, which increments count.

Figure 2 –
structure of a pre-test loop

The update in a pre-test loop generally occurs at the end of the body of the loop, after the processing. This is a convention adopted by many programmers to make the logic in an algorithm easier to understand. It's a good idea to do things this way to avoid confusion, but it is not absolutely necessary to do so.

Pre-test Loops in Java

Just as in pseudocode, the *while* statement is used to establish pre-test loops in Java. Here is the same program from above as Java code:

```
/*
 * preTest.java
 * Sample of a simple pre-test loop using while
 * last edited Jan. 25, 2015 by C. Herbert
 */
package pretest;

public class Pretest {

public static void main(String[] args) {
    int count; // the control variable for the loop

    // initialization -- set count to 1
    count = 1;

    // test -- continue while count ≤ 10
    while (count <= 10) {

        // processing -- print a message with the value of count
        System.out.println("The value of count is " + count );

        // update the control variable -- increment count
        count++;
        } // end while (count <= 10)

    } // end main()

} // end class Pretest
```

Figure 3 – pre-test loop

In Java, *while* statements are very similar to *if* statements, except the flow of the program goes back to the *while* command each time the block of code related to the *while* statement is executed.

Loop Invariants

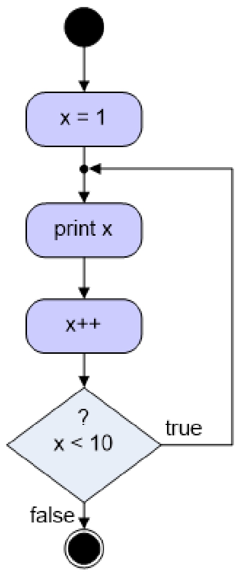
A *loop invariant* is a condition which is true each time through a loop. Computer scientists and software engineers sometimes use loop invariants to test a loop to see if it functioning properly. As an example, consider the algorithm on the left that finds the minimum value in a group of 10 numbers being input by a user. A loop invariant in this example is that min is the lowest value so far.

```
count = 1;
input n;
min = n;
while (count <= 10) {
   input n;
   if (n < min) then min = n;
   count++
   } // end while(count <= 10)
```

Post-test Loops

A **post-test loop** is a loop in which the test to determine whether or not to repeat the loop comes after the body of the loop. Here is our *one to ten* algorithm with a post-test loop:

```
start
x = 1;
do {
    print x;
     x++;
 } while (x <=  10);
stop
```



This post-test algorithm produces the same results as the pre-test algorithm we saw earlier. The **do…while** structure establishes a post-test loop, with *do* at the beginning of the loop and the *while* condition at the end of the loop.

Notice the semicolon at the end of the *while* statement. This should be used in pseudocode (and in Java) to show that the *while* terminates the loop and does not start a new loop inside the first loop.

```
count = 1;
do {
    print count;
     count++;
 } while ( count ≤ 10);
```

- The ***initialization*** happens before the loop starts just as in a pre-test loop.
- The ***processing*** in a post-test loop comes first within the body of the loop.
- The ***update*** in a post-test loop could occur before or after the processing within the loop. Be careful, this affects the nature of the processing. Consider how this loop would be different if the update occurred before the processing.
- The ***test*** in the while statement follows the brace closing the loop. It is usually on the same line as the brace.

Figure 4–
structure of a post-test loop

The braces in a post-test loop usually line up so that the word *do* lines up with the brace in front of the while statement that marks the end of the loop.

Post-test Loops in Java

The *do …while* statement is used to establish a post-test loop in Java, just as it is in the pseudocode above. The following program is a post-test version of the pre-test program shown earlier to print the numbers from 1 to 10:

```
/*
 * postTest.java
 * Sample of a simple post-test loop using do…while
 * last edited Jan. 25, 2015 by C. Herbert
 */
package posttest;

public class PostTest {

 public static void main(String[] args) {
    int count;      // the control variable for the loop

    count = 1;      // initialization set count to 1

    do {   // while (count <= 10)

        // processing -- print a message with the value of count
        System.out.println("The value of count is " + count );

        // update -- increment count
        count++;

    // test -- continue while count ≤ 10
    } while (count <= 10);

  } // end main()

} // end class Posttest
```

Figure 5 – post-test loop

The *do* statement in this code has a comment indicating which *while* command it matches. This is a good idea in case there are several *do...while* structures in the same method.

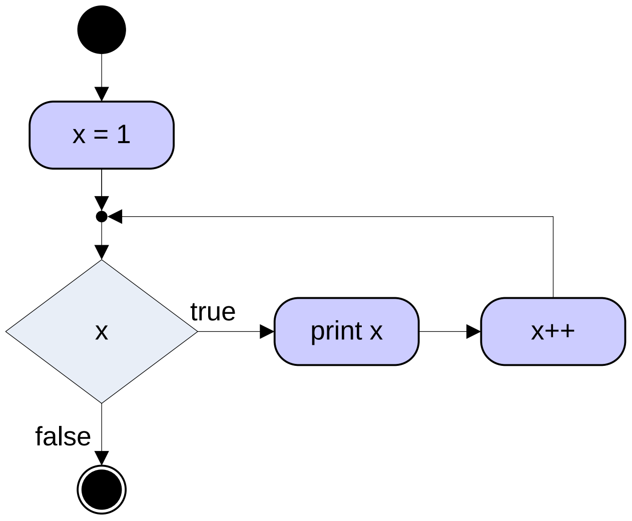
The opening brace for the block of code is on the same line as the *do* statement. The *Format* command on NetBean's *Source* menu which lines up code in a NetBean's Java source code file, including fixing indenting in code, will line up the code this way. It will also indent the brace at the end of a *do…while* loop to line up the indenting for the closing brace on the same line as the *while* statement with beginning of the *do* statement, also as in the code above.

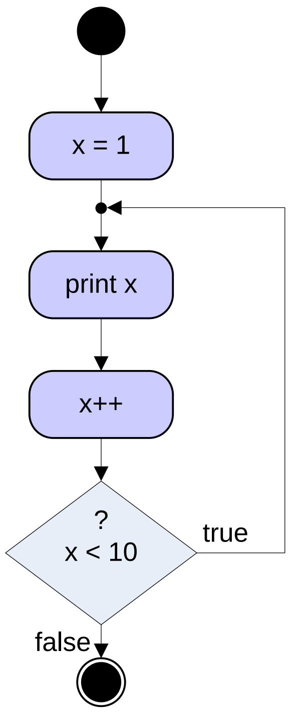
It is especially important in Java to be careful about using a semicolon at the end of a while statement. We do not put a semicolon on a while statement at the beginning of a pre-test loop, but we do need a semicolon at the end of the while statement at the end of a post-test do … while structure.

How do we choose when to use a pre-test loop and when to use a post-test loop? We will see specific examples later in this course where one format or the other is better to use, but in general pre-test loops do tend to be used far more often than post-test loops. In fact, some computer scientists argue strongly that we should never use post -test loops. In fact, the Python language has no do…while structure specifically to discourage the use of post-test loops.

Pre-test loops tend to be safer than post-test loops because the condition is checked before the body of the loop is executed. A pre-test loop is generally better if we want to avoid situations in which the loop might execute once when we don't want it to do so. Imagine a program on a network that erases hard disk drives. Which is better, a program that, when we run it, erases a hard disk drive then asks if we want to repeat the process for another disk, or a program that asks first if we really want to erase a hard disk drive? What if someone runs the program by mistake?

We can see this in the two flowcharts side by side below. What would happen if x were accidentally initialized x to 11 instead of 1? In the pre-test loop on the left, the loop would not be executed; but in the post-test loop on the right, 11 would be printed.





Some programming languages have a *repeat…until* structure for post-test loops. Instead of checking for a continuation condition, a *repeat…until* structure checks for a terminating condition. A **continuation condition** tells a computer when to continue repeating a loop. A **terminating condition** tells a computer when to stop executing a loop. Continuation conditions and terminating conditions are the logical opposite of each other for any one loop. In the *one to ten* programs above, (x <= 10) is the continuation condition and (x > 10), the logical opposite of (x <= 10), would be a terminating condition.

Java does not support a *repeat…until* structure.

CheckPoint 4.2

1. Where can the test to determine if a loop should be repeated be located?
2. What are the four parts of a pre-test loop and in what order should they occur?
3. What happens in a loop update?
4. What are the four parts of a post-test loop and in what order should they occur?
5. What language is used in Java for a post-test loop?

Lesson 4.3 Count-Controlled Loops and Sentinel Loops

Count-controlled Loops

A **count-controlled loop** is a loop in which the loop control variable is an iterated numeric variable. An **iterated numeric variable,** also known as a **counter**, is a numeric variable that changes in discrete steps from one value to another. When we count to ten, we are changing a value from one to ten in steps of one. The pre-test and post-test loops we saw earlier in this chapter were count-controlled loops.

There are three things we need to know about the counter to set up a count-controlled loop:

- the initial value of the counter
- the final value of the counter
- the increment – the change in the counter each time through the loop

In the one to ten loops we saw, the initial value is 1, the final value is 10, and the increment is 1.

We could have a negative increment, also called a decrement, but only if the final value is less than the starting value. For example, we could have an initial value of 10, a final value of 1, and an increment of -1, which would cause our loop to go from 10 down to 1.

Be careful – if the initial value and the final value don't match the increment – for example if the initial value is 10 the final value is 1 and the increment is +1 – then we could end up with an infinite loop. the values of the counter would start at 10, then be 11, then 12, then 13, and theoretically never reach 10.

However, if we are using a computer, then the program will eventually stop. Normally, one of three things will happen:

- The computer will timeout automatically if nothing is happening. Some operating systems have a built in mechanism for this.
- The computer will run out of memory. Of course, this only applies if more memory is used each time through a loop.
- The loop control variable will go out of bounds. You may recall from chapter two that each data type has an acceptable range of values. Eventually the value of the loop control variable will be out of bounds -- out of the acceptable range for that data type.

One of these three things will eventually happen, but it could take a while – variables of the *int* data type, for example, have a range of values from -2,147,483,648 to +2,147,483,647. Variables of Java's *long* data type have a range of values from roughly 2^{-63} to 2^{+63}. At one billion iterations per second, it would take a loop almost 300 years to count from 1 to 2^{63}.

The increment for a counter in a count-controlled loop does not need to be 1 or -1. It could be almost any value. Here are some examples. Can you identify the output from each one?

```
x = 1;              x = 10;             x = 1;              x = 2;              x = 0;

while ( x ≤ 10) {   while ( x ≥ 1) {    while ( x  ≤ 100)   while ( x ≤ 100)    while ( x ≤ 10) {
    print x;            print x;            print x;            print x;            print x;
     x++;                x--;                x = x +2;           x = x +2;           x = x + 0.5;
} // end while      } // end while      } // end while      } // end while      } // end while
```

Count-controlled Loops in Java using the *for* Statement

Java, like many programming languages, has a special statement that can be used to set up a count-controlled loop – the *for* statement. The **for statement** is a single statement that establishes a count-controlled loop, allowing a programmer to specify the initial value, final value, and increment for the loop's counter.

[Note: There are two forms of the *for* statement, the *basic for* statement and the *enhanced for* statement. Here we are only discussing the *basic for* statement. The *enhanced for* statement is primarily used with arrays and will be discussed later in Chapter 6.]

Here is the *for* statement format:

```
for (initialization ; continuation expression ; update)
{
    body of the loop
}
```

The information in parentheses following the word *for* is called the **for header**. It contains an *initialization*, a *continuation expression*, and an *update*.

The **initialization** in a *for header* is an assignment statement specifying the loop control variables and its initial value. A loop control variable may be declared in the initialization if it will only be used within the loop as the loop counter.

The **continuation expression** in a *for header* must be a boolean expression, similar to the continuation condition in a *while* statement. The loop will continue to execute each time the continuation expression is true. The loop will terminate and move on to whatever comes next when the continuation expression is false.

The **update** in a *for header* is an assignment statement, which should change the value of the counter. The most common forms of a for statement update are the increment and decrement expressions

counter++; and **counter--;** The body of the loop should not include an update because of the update in the for header.

A *for* statement works as follows:

1. The *initialization* is executed.
2. The continuation expression is evaluated.
    a. If the continuation expression is true, the body of the loop is executed. Following the execution of the body of the loop, the update is executed and the continuation expression is again evaluated. This continues until the continuation expression is false, or until something within the body of the loop, such as a *break* statement, causes the loop to terminate.
    b. If the continuation expression is false, the body of the loop is bypassed, and the computer moves on to whatever comes after the loop.

Almost any valid assignment statements will work as the initialization and update portions of the *for* header. Any valid boolean expression should work as the continuation expression. The JLS does not specify data types that may be used with a *for* statement. Numeric variables are most often used.

Here is a one-to- ten program with a *for* loop:

```
public static void main(String[] args)  {

    for (int i=1; i <= 10; i++)
        System.out.println("The value of count is " + i);

    } // end main()
```

Figure 6 – for loop

No braces are needed to mark the body of the loop because it is a single instruction. A block of code with multiple statements could also form the body of the loop, marked by beginning and ending braces.

The convenience and ease of programming using the *for* statement becomes immediately obvious. Compare this to the programs in Figure 3 on page 8 and Figure 5 on page 10 that do the same thing using the *while* statement. The loop's *initialization*, *test*, and *update* are all contained in the *for* header. It is much more desirable to use a *for* statement for count-controlled loops than to use a *while* statement. It is easier to program, less prone to errors, and easier to read in source code.

Sentinel Loops

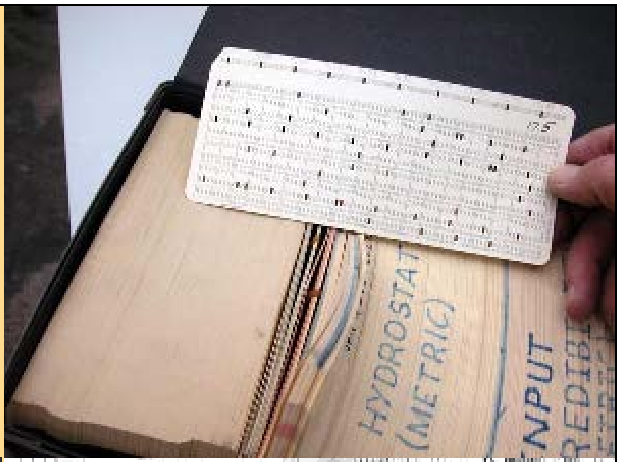
A **sentinel** is a marker that signals when a loop should be terminated. A **sentinel loop** continues until it encounters a sentinel. Sentinels can be used to signal the end of a loop with data input, either data input by a user, or from a data file. The sheep program in Figure 7 on the next page shows an example of a sentinel loop in Java.

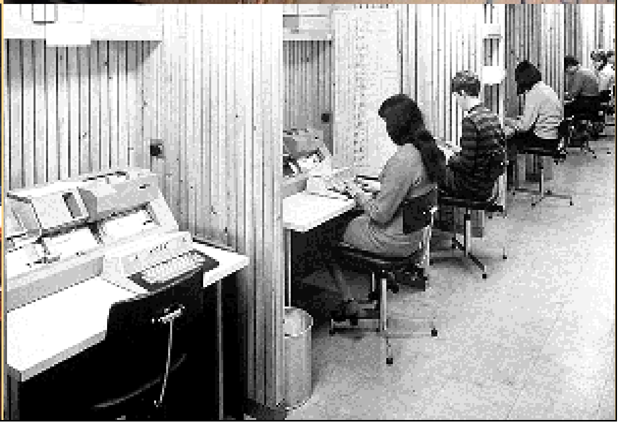
Count controlled loops will be executed a specific number of times, whereas a sentinel loop will continue to execute unit a specific sentinel is encountered. Consider a program print mailing labels. A count controlled loop could be used to print labels from a mailing list with 1,000 names, if we know ahead of time that there are 1,000 names in the list. A sentinel loop could be used to print mailing labels until a sentinel is reached at the end of the list. The number of times a sentinel list repeats depends on the data, not on a pre-determined count. Of course, we could also use a count controlled loop in which the count is read in before the loop starts, but there are many cases where a sentinel loop is more useful.

Trailer Cards

Sentinel loops originated with trailer cards. The first generation of electronic computers used read data in from a deck of punched cards, with one record on each card. The last card in the deck, called a trailer card, marked the end of the deck. Sentinel values are sometimes still referred to as trailer values.

These images from a British university circa 1970 show a deck of cards, students creating decks of cards with keypunch machines, and an operator loading a deck of cards into a card reader.





```
/* This method will help a farmer keep track of the weight of a group of sheep
 * It asks for the weight of each sheep and calculates the count and average
 * weight. The sentinel value -99 is used to terminate the loop.
 */

  public static void main(String[] args) {
    int weight;                 // the weight of an individual sheep
    int count = 0;              // the number of sheep
    double totalWeight = 0;     // the total weight of a group of sheep
    double averageWeight;       // the average weight of a group of sheep

    // declare a Scanner for the keyboard
    Scanner kb = new Scanner(System.in);

    // opening message to the user
    System.out.println("This program will help keep track of the weight of your sheep.\n");
    System.out.println("Enter the weight of each animal.");
    System.out.println("Enter  -99 to indicate the end of the list.");

    // get first weight   -- priming read
    System.out.println("Weight?");
    weight = kb.nextInt();

    while (weight != -99) {

        // increment count and add weight to total
        count = count + 1;
        totalWeight = totalWeight + weight;

        // get next weight  -- "\n separates prompt and capture input sets
        System.out.println("\nWeight?");
        weight = kb.nextInt();

        } // end while while (weight != -99)

    // calculate average weight
    averageWeight = (double) totalWeight / count;

    // display results
        System.out.println("Data for " + count + " sheep:");
        System.out.println("\ttotal weight:\t" + totalWeight);
        System.out.println("\taverage weight:\t" + averageWeight);

    } // end main()

} // end class Sheep
```

Figure 7 – sentinel loop

The value *-99* serves as the sentinel in the loop in the sheep program. The *do...while* structure will continue looping until the user enters the value *-99* as the weight of a sheep. It is important to use a value for the sentinel that could not possibly be a real data element. In this example, assuming that sheep must have a positive weight, -99 is an acceptable sentinel value.

The concept of a sentinel is important in reading data files. Usually there is a special value at the end of a file called an *end of file marker*. We can write a loop to read data in from a data file until the end of file marker is encountered, just as the loop above continues until it encounters the value *-99*. We will see more about this when we study data files later in the semester.

CheckPoint 4.3

1. What is an iterated numeric variable?
2. What three things do we need to know about a counter in a count controlled loop?
3. Must a count controlled loop's increment always be positive? Must it always be 1?
4. What are the parts of a *for header* in a *for loop*?
5. What is the difference between a count-controlled loop and a sentinel loop?

While Loops in Other Languages

While loops in the *C*, *C++*, *Microsoft C#*, and the *Apple Swift* programming languages are almost identical to *while* loops in Java. Here are some examples of a while loop to print the numbers from one to ten in several programming languages. Almost all use syntax derived from ALGOL 68.

Java
```
while (n <= 10) {
   System.out.println(n);
   n++;
   }
```

C
```
while (n <= 10) {
   printf("%d\n",n);
   n++;
   }
```

C++
```
while (n <= 10) {
   printf("%d\n",n);
   n++;
   }
```

FORTRAN
```
do while (n .LE. 10)
     print *, n
     n= n + 1
   end do
```

VB.net
```
While n <= 10
   Console.WriteLine(n)
   n += 1
End While
```

C#.net
```
while (n <= 10)
{
   Console.WriteLine(n);
   n++;
}
```

Python
```
while n <= 10:
    print n
    n +=1
```

Ruby
```
while n <= 10 do
   puts n
   n +=1
end
```

Swift
```
while n <= 10  {
   println("\(n)")
   n++
}
```

Lisp
```
(loop while (<= x 10) do
    (write x) (terpri)
    (setq  + n 1)
)
```

PHP
```
while ($n <= 10):
    echo $n;
    $n++;
endwhile;
```

Perl
```
while ($n > 0) {
   say $n;
   $n +=1;
   }
```

Lab 4A – Count-controlled Loops

The code for the count-controlled pre-test loop shown in Figure 3 on page 8 is saved as a NetBeans project in the folder *pretest.zip* in the files for Week 4. Open the file in NetBeans and try the program.

The examples below were first shown on page 13. Change the *pretest* code to count as each example indicates. See if the output is what you predicted. (In which loop should x be a double variable?)

```
x = 1;               x = 10;              x = 1;                x = 2;               x = 0;
while ( x ≤ 10) {    while ( x ≥ 1) {     while ( x  ≤ 100) {   while ( x ≤ 100) {   while ( x ≤ 10) {
     print x;             print x;             print x;              print x;             print x;
      x++;                 x--;                 x = x +2;             x = x +2;             x = x + 0.5;
 } // end while       } // end while       } // end while        } // end while       } // end while
```

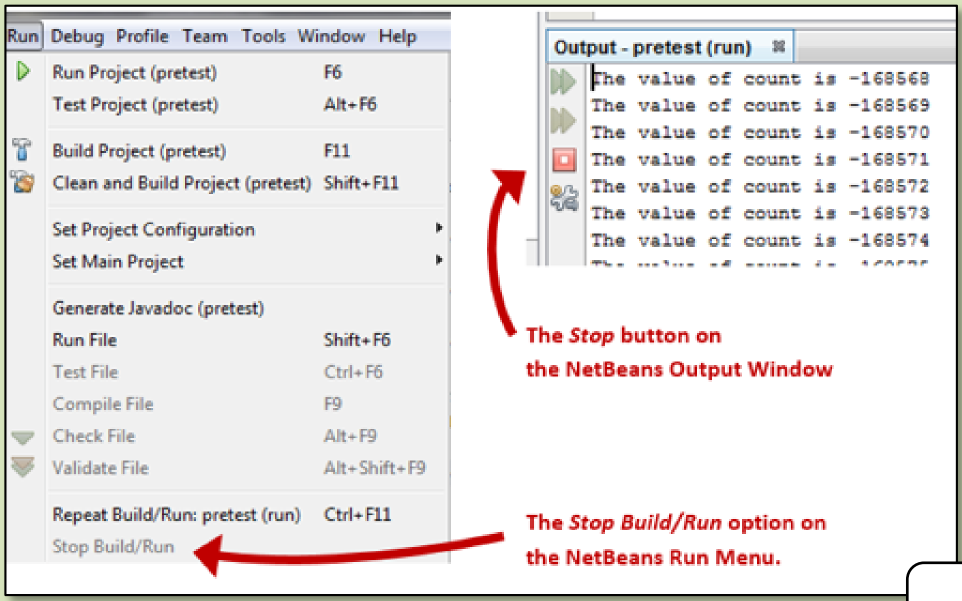
You should also be able to create *for* loops equivalent to the *while* loops in these examples.

You can set up an infinite loop on purpose to see what happens. An initial value of 1, a final value of 10, and an increment of -1 will cause an infinite loop.

As mentioned above, no loop will run infinitely, but it could take a while to stop. The output part of the program is the slowest part, because output statements communicates with the video controller, which is slower than operations that happen only in the CPU.

NetBeans Notes – Stopping a Runaway Program

As we saw in Chapter 2, NetBeans has two mechanisms to stop a runaway program. The Run menu has a Stop Build/Run option at the bottom of the menu, and the NetBeans output window has a stop button. Either one can be used to stop a program running in NetBeans.


Run Debug Profile Team Tools Window Help
Run Project (pretest) F6
Test Project (pretest) Alt+F6
Build Project (pretest) F11
Clean and Build Project (pretest) Shift+F11
Set Project Configuration
Set Main Project
Generate Javadoc (pretest)
Run File Shift+F6
Test File Ctrl+F6
Compile File F9
Check File Alt+F9
Validate File Alt+Shift+F9
Repeat Build/Run: pretest (run) Ctrl+F11
Stop Build/Run
Output - pretest (run)
The *Stop* button on the NetBeans Output Window
The *Stop Build/Run* option on the NetBeans Run Menu.

```
The value of count is -168568
The value of count is -168569
The value of count is -168570
The value of count is -168571
The value of count is -168572
The value of count is -168573
The value of count is -168574
```

Figure 8 – NetBeans stop button

Lesson 4.4 Formatting Output in Java

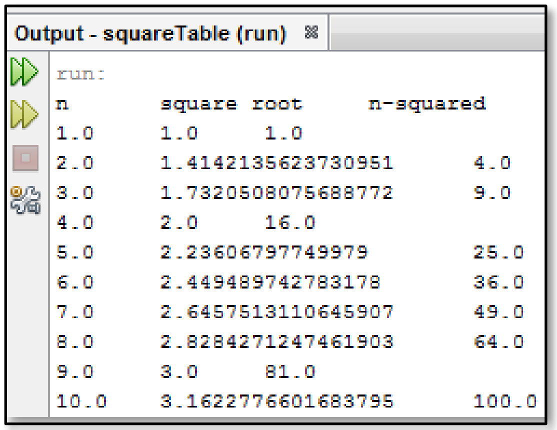
For loops are routinely used to print tables of data, such as a table of squares and square roots, trigonometric tables, or an amortization table for a bank loan or home mortgage. The code below prints a table of squares and square roots for the numbers from one to ten.

```
// print column headers -- \t is the tab character
System.out.println("n\tsquare root\tn-squared");

for (double i = 1; i <= 10; i++)
    System.out.println(i + "\t" + Math.sqrt(i) + "\t" +  i*i);
```

Figure 9 – poorly formatted output

Unfortunately, the output from this code, shown on the right, is not the nice neat table we might expect, even though we use the Unicode tab character to format the data.

Lines for 1, 4, and 9 line up with each other, but not with the header. Lines for 2, 3, 5, 6, 7, 8, and 10 line up with each other, but not with lines for 1, 4, and 9.



```
run:
n       square root     n-squared
1.0     1.0     1.0
2.0     1.4142135623730951      4.0
3.0     1.7320508075688772      9.0
4.0     2.0     16.0
5.0     2.23606797749979        25.0
6.0     2.449489742783178       36.0
7.0     2.6457513110645907      49.0
8.0     2.8284271247461903      64.0
9.0     3.0     81.0
10.0    3.1622776601683795      100.0
```

This happens because floating point numbers are printed in Java using only the significant digits. We need to format the output better to get a nice looking table.Java has two methods to format printed output: **System.out.format()** and **System.out.printf()**. They format values in the same way, and can be used to specify how wide a column should be, how many digits are used to print a number, and even whether floating point numbers are printed in decimal form, scientific notation, or in base 16 or base 2.

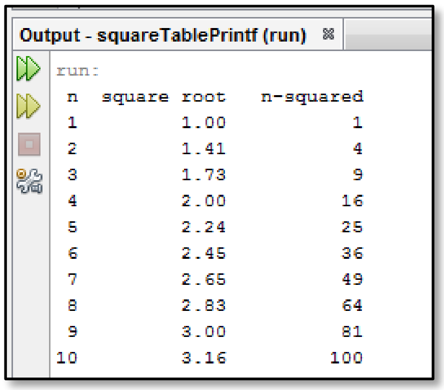
We will discuss the *System.out.printf()* method. The techniques for formatting data are the same for the *System.out.format()* method.

The *printf()* method requires a format String, followed by a variable or set of variables. A **format String** contains codes describing how output should be formatted. Here is an example using *printf*:

```
// print column headers -- \t is the tab character
System.out.printf("%2s%13s%12s%n", "n", "square root", "n-squared");

for ( double i = 1; i <= 10; i++ )
   System.out.printf( "%2.0f%13.2f%12.0f%n", i, Math.sqrt(i), i*i );
```

Figure 10 – better output formatting

The String *"%2.0f%13.2f%12.0f%n"* in the loop is an example of format String for *printf*. It has a series of percent signs, followed by format specifiers. They are described in detail on the next page.



```
run:
 n  square root   n-squared
 1         1.00           1
 2         1.41           4
 3         1.73           9
 4         2.00          16
 5         2.24          25
 6         2.45          36
 7         2.65          49
 8         2.83          64
 9         3.00          81
10         3.16         100
```

The table of squares and square roots formatted by using *printf* is shown on the right. It was generated by the code on the last page.

The format String *"%2.0f%13.2f%12.0f%n"* was used to print the data in this table. It is composed of a series of print specifiers that tell the computer how to display values.

Each print specifier starts with a percent sign % and includes conversion codes, format flags, and numbers specifying the field width and precision of data. **Conversion codes** indicate the general format to use in printing data. **Format flags** turn on certain features, such as printing leading zeros with numbers.

The String *"%2.0f%13.2f%12.0f%n"* has four print specifiers -- *%2.0f*, *%13.2f*, *%12.0f* and *%n* .

In the first three, *f* is the conversion code, telling the computer to display a floating point number (double or float) as a decimal. The number before the *f* tells the computer the field width and precision to use when printing a value. *2.0* means that the field for the number should take up 2 columns, with no decimal places. In the second specifier, *%13.2f*, the field width is 13 and the precision is 2 places after the decimal. The width should include the decimal point. Values are right justified.

The *%n* print specifier at the end of the String tells the computer to start a new line.

Java printf and print format specifiers		
code	**description**	**examples**
f	print a floating point value as a decimal number right justified in the field. It has two numbers, *width* (which includes decimal points) and *precision*, the number of places to the right of the decimal	8.2f width is 8, precision is 2 value 123.4567 prints as 123.45 7.3f width is 7, precision is 3 value 123.4567 prints as 123.456
e	print a floating point value in scientific notation right justified in the field. It has two numbers: *width* (which includes decimal points, etc.) *precision*, the number of places to the right of the decimal in the mantissa	8.2e width is 8, precision is 2 value 1234.567 prints as 1.23e+03 7.3e width is 7, precision is 3 123456.7891 prints as 1.234e+05
d	print an integer value as a signed decimal integer right justified in the field. number specifies *width*	12d width is 12 value 123 prints as 123
s	print a String as a UTF-16 string right justified in the field. number specifies *width*	14s width 14 value "Smith" prints as Smith
c	print a char as a single UTF-16 character	8c width 8 value 'A' prints as A
n	newline character -- similar to "\n"	go to beginning of next output line

Several format flags may be used to modify a print specification. Flags are used in immediately after the % and before any numbers or letters in the specifier. for example, if hrs = 37.5, then the flag '+' in

```
    System.out.printf( "%+7.2f", hrs);
```

prints:

```
     +37.50
```

Here is a list of flags commonly used in *printf()* and *format()* specifiers:

flag	symbol name	effect
-	minus sign	left justifies values using the specified field width
+	plus sign	value is preceded by a sign: - for negative values, + for non-negative (0 could be displayed with + or -,depending on the context.)
	blank space	leading space for positive numbers, minus sing for negative values
0	zero	values are zero-padded to fill the specified field width
,	comma	a comma will precede every third digit to the right of the decimal point for values > 1,000
(	left parenthesis	negative values will be enclosed in parentheses (0 could be displayed as positive or negative, depending on the context.)

The *printf* statement has the following syntax:

```
        System.out.printf( "format string", arg1, arg2, … etc. );
```

The number of arguments and their data types should match the specifications in the format String. The following examples show how *printf* works:

```
System.out.printf( "%8s%f7.6%n", "hours:", hrs);    // hrs = 37.5
  hours:37.50
System.out.printf( "%-8s%f7.2%n", "hours:", hrs);    // hrs = 37.5
hours:  37.50
System.out.printf( "%-8s%0f6.2%n", "hours:", hrs);    // hrs = 37.5
hours: 037.50
System.out.printf( "%8s%f8.2%n", "feet:", ft);    // ft = 5280
feet:    5280.00
System.out.printf( "%8s%,f8.2%n", "feet:", ft);    // ft = 5280
feet:   5,280.00
System.out.printf( "%8s%,f8%n", "feet:", ft);    // ft = 5280
feet:   5,280
System.out.printf( "%8s%,+f8%n", "feet:", ft);    // ft = 5280
feet:  +5,280
System.out.printf( "%8s%e8.2%n", "feet:", ft);    // ft = 5280
feet:5.28e+03
System.out.printf( "%8s%8d%8d%8d%8d%n", "change:", q, d, n, p); // works with integers
change:       2       1       3       4
```

The information in this section about using printf only discusses the most commonly used format specifications. There are many others for a variety of situations, including formats for time, date, system memory locations in hexadecimal code, and so on. Some of these will be discussed later this semester and in CSCI 112 and 211 where appropriate.

Full specifications for all Java format specifiers and flags are available in Oracle's Java SE Documentation, online at: http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax

CheckPoint 4.4

1. What are two methods that can be used in Java code to format output?
2. What is a format String and how is it used to format output?
3. What is the difference between a conversion code and a format String in a Java print specifier? Describe several examples of each.
4. What is the output from each of the following:
   a. System.out.printf("%20s%f5.2%n", name, gpa); // gpa = 3.6666667
   b. System.out.printf("%8s%4d%n", "count", count); // count = 12
   c. System.out.printf("%8.2e", AVAGADRO); // AVAGADRO = 6.02214129 x 10^{23}
5. Create print statements to format values similar to those shown here:
   a. 1,024 b. 3.14159 c. -48.62 d. 0016 (left-justified)

Lab 4B – Formatted Output with *for* Loops

The following code prints a table of feet, yards, and meters, but the output does not look nice.

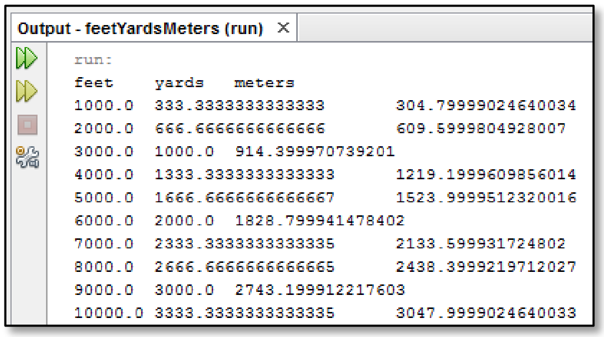
```
// print column headers -- \t is the tab character
System.out.println("\tfeet\tyards\tmeters");

for ( double feet = 1000; feet <= 10000; feet = feet + 1000)
       System.out.println(feet + "\t" + feet/3 + "\t" +  feet/3.28084 );
```

Figure 11 –
for loop with bad formatting

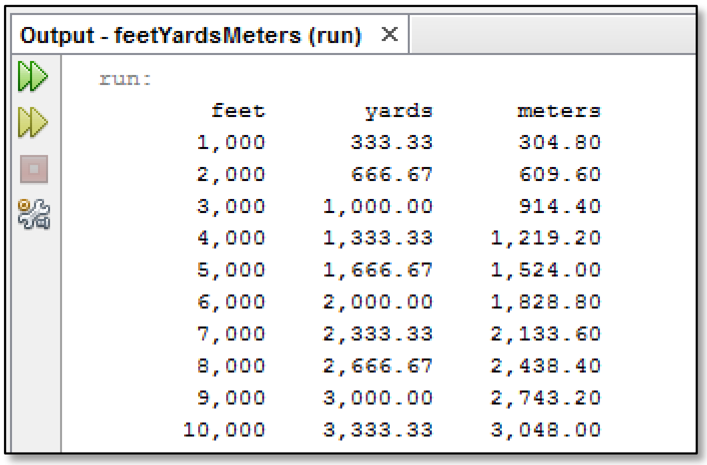
The output is shown on the right. The complete NetBeans Project is in the file *feetYardsMeters.zip* in the Canvas Week 4 files.

Your task is to modify the code to use the *printf* method instead of *println*, and to create a format String to print nice looking output, such as what's shown below.

The numbers in each column should be lined up and numbers greater than 1,000 should include a comma, but the exact format is up to you.


Output - feetYardsMeters (run)

```
run:
feet     yards    meters
1000.0   333.3333333333333        304.7999024640034
2000.0   666.6666666666666        609.5999804928007
3000.0   1000.0   914.399970739201
4000.0   1333.3333333333333       1219.1999609856014
5000.0   1666.6666666666667       1523.9999512320016
6000.0   2000.0   1828.799941478402
7000.0   2333.3333333333335       2133.599931724802
8000.0   2666.6666666666665       2438.3999219712027
9000.0   3000.0   2743.199912217603
10000.0 3333.3333333333335       3047.999902464033
```


Output - feetYardsMeters (run)

```
run:
          feet       yards      meters
         1,000      333.33      304.80
         2,000      666.67      609.60
         3,000    1,000.00      914.40
         4,000    1,333.33    1,219.20
         5,000    1,666.67    1,524.00
         6,000    2,000.00    1,828.80
         7,000    2,333.33    2,133.60
         8,000    2,666.67    2,438.40
         9,000    3,000.00    2,743.20
        10,000    3,333.33    3,048.00
```

Lesson 4.5 Random Numbers in Java

The pseudocode for a guessing game algorithm from Chapter 3 is repeated below. Sample output is on page 20 in Chapter 3, and a flowchart for the algorithm follows on page 21.

```
int pick         // the number the computer picks
int guess = 0    // the user's guess, initialized to zero
int count = 0    // the number of guesses
pick = a random integer  between 1 and 100
print  "I am thinking of a number between 1 and 100. Try to guess what it is."
print" Your guess?"
user inputs guess                  // get value of guess from the keyboard input
while (guess ≠ pick) {             // set up a loop to repeat until the user guesses the number
      increment count              // keep track of how many guesses
          if (guess < pick)        // in the loop guess cannot equal pick: it will be low or high
            print "Too low. Try again."
      else
            print "too high. Try again."
      print "Try again."
      print "Your guess?"
      user inputs guess            //get value of guess from the keyboard input
          } //end while (guess ≠ pick)      // the loop ends when guess = pick
print "Correct. The number Is " + pick
print "It took you " + count + " guesses"
```

Figure 12 –
guessing game pseudocode

One line near the beginning of the pseudocode reads "pick = a random integer between 1 and 100". In this section we will see how to generate random numbers in Java for use in algorithms such as this.

The Java Math class has a method to generate random numbers: **Math.random()**, which actually returns a pseudorandom double value greater than or equal to 0.0 and less than 1.0. A **pseudorandom number generator** returns a sequence of numbers that are evenly distributed like random numbers, but the generator is deterministic. This means that the sequence of numbers can be determined and repeated, given the initial state of the generator and the algorithm it uses to generate numbers. Given the correct information, the next number in the series can be predicted with certainty.

For many purposes, a pseudorandom number generator is fine. Later we will learn to generate more truly random numbers.

The value returned by Math.Random() is in the range $0.0 \le n < 1.0$. What if we want to pick an integer from 1 to 10, including 1 and 10? We start by multiplying by the number 10. 0 times 10 is 0. 1 times 10 is 10. If $0.0 \le n < 1.0$, and we multiply n by 10, then $0.0 \le n < 10.0$.

But we still don't have an integer, and notice the range is < 10, so ten is not included in the possible values yet. Our next step is to type cast the result into an integer. Remember, *y = (int) x;* turns y into an integer by truncating x. It does not round off x, it chops off the decimal. If 0.0 ≤ *n* < 10.0, and we truncate the value, we get 0 ≤ n < 10. The number will be an integer in the set {0, 1, 2, 3,… 9} .

Next we just add 1 to the value and now it is in the set {1,2,3…,10} which is what we want.

So, to generate an integer in the range from 1 to 10, including 1 and 10 with Math.random() use:

```
x= 1+ (int) (Math.random() * 10);
```

This does exactly what we want – picks a number in the range 0.0 ≤ *n* < 1.0, multiply by 10, truncate it to be an integer, then add 1.

So, to set a variable x equal to a random integer between *1* and *b*, including *1 and b*, the assignment statement should be:

x= *1* + (int) (Math.random() * *b*);   where *b* is the high endpoint of the range of values.

The following examples show how to generate random integers for specific ranges of values. Generating random floating point numbers is similar, but we do not typecast the values to integers.

pick a door – 1, 2, or 3

door = *1* + (int) (Math.random() * *3*);

simulate rolling a pair of dice, each between 1 and 6

d1 = *1* + (int) (Math.random() * *6*); d2 = *1* + (int) (Math.random() * *6*); total = d1 +d2;

pick a card from a deck of cards

suit = *1* + (int) (Math.random() * *4*); // if 1 print "hearts", if 2 print "clubs", etc.
rank = *1* + (int) (Math.random() * *13*); // 1 is an "ace", 11 is a "jack", 12 is a "queen", 13 is a "king"

What about picking a number between 100 and 500?

number = *100* + (int) (Math.random() * *400*);
In this case the lowest number is 100, and there are 400 numbers in the range.

In general, to generate an integer in the range a ≤ number ≤ b
number = *a* + (int) (Math.random() * *b*); // *a* is the lowest value and *b* is the number of values in the range.

CheckPoint 4.5

1. What values are generated by the *Math.random()* function?
2. What is a pseudorandom number generator?
3. Create a Java statement to generate pseudorandom integers from 1 to 20.
4. Create a Java statement to generate pseudorandom floating point numbers such that for each number *n*, 2.25 ≤ *n* ≤ 2.75.
5. What is the range of values generated by the following?
   number = *128* + (int) (Math.random() * 64);

Lab 4C – Random Numbers

The Java NetBeans project in the file *testran.zip* has the code below to generate and print 100 integers between 1 and 6.

STEP 1.

Download and unzip the file *testran.zip* from the week 4 files in Canvas.

STEP 2.

Open the Project in NetBeans.

STEP 3.

Examine the program and run it to see what it does and how it works.

STEP 4.

Modify the program to pick numbers in a different range. Try these ranges:

numbers between 1 and 25; numbers between 1 and 12; numbers between 10 and 30;
numbers between -9 and 9

STEP 5.

Modify the program to print numbers in a different pattern, by changing the number of rows and columns printed and by changing the field width in the *printf* statement.

STEP 6.

Modify the program to print random double values accurate to 2 places after the decimal point. To do this, *number* needs to be a double value and the typecasting to an *int* value in the assignment statement needs to be removed. Also, don't forget to change the literal values, such as 1, to numbers with decimal points, such as 1.0. You can decide on the range of values and number of rows columns. The *printf()* format String will need to be changed to print floating point values.

```
    // This method prints 100 random numbers in five rows of 20 numbers each
    public static void main(String[] args) {
        int number;       // the number to be printed
        int row;          // the row number
        int col;          // the column number

        for (row = 1; row <= 5; row++) {
            for (col = 1; col <= 20; col++) {
                number = 1 + (int) (Math.random() * 6);
                System.out.printf("%4d", number);
            } // end col

            System.out.println();
        } // end row

    } // end main()
```

Figure 13 – generating random numbers

Excerpt from The History of Computing Machinery

The following text, about 1,200 words, is from a much longer chapter about the history of computing machinery. It is the basis for this week's class discussion.

The Earliest Languages

Language is a tool we use to share our ideas with other people. It is also a tool we use to shape our ideas in each of our own minds. Language is a tool for processing ideas; it is a tool we use for processing information.

Most human languages are verbal, but languages can also be composed of primitive sounds, like whistling or clicking, or hand gestures, or other symbols, especially written symbols. Written languages are important because they allow us to preserve ideas more accurately and for a longer time than spoken languages. Written languages allow us to store information.

We have an obvious problem when trying to trace the roots of language that existed before written language – nothing was written down (or otherwise recorded). The time before written language is called the prehistoric period, and we have no record of the development of language in prehistoric times. Most specialists in the field believe that languages developed slowly as people evolved from more primitive organisms. One noted exception is the famous linguist Noam Chomsky, who said he believes language appeared suddenly when modern man appeared.[1]

The oldest preserved written language we do have today is Sumerian, a cuneiform script that dates back about 5,000 years to 3,000 BCE. Sumerian was to the ancient world what Latin is to the modern world – the spoken form appears to have died out in about 2,000 BCE, but the written form survived and influenced other languages for centuries afterwards.[2] It was replaced by the Akkadian language, which developed into the Assyrian and Babylonian languages of the ancient Middle East. Sumerian and Akkadian writing was composed of logograms – symbols that stood for ideas.

The written form of the ancient Egyptian language seems to have appeared just after Sumerian, but there was something different about Eqyptian – it had phonetic symbols, called phonograms, in addition to logograms like those found in Sumerian. The difference between logograms and phonograms is significant. **Logograms** are primitive, with rough symbols to represent things such as cow, horse, man, and so on. Logograms mainly represent physical things. **Phonograms** represent spoken language, which can be richer in meaning since the phonograms represent syllables that can be combined to form new words related to the meaning of old words. Phonograms make it easier to represent more abstract

[1]. For more information about Chomsky, see: http://www.chomsky.info . His *Generative Theory of Linguistics* contributed to the development of computer languages. Chomsky was born in East Oak Lane and studied at Penn.

[2] The Sumerian language appeared in the southern part of ancient Mesopotamia. The Encyclopedia Britannica has a good article on Sumerian online at: http://www.britannica.com/EBchecked/topic/573229/Sumerian-language The University of Pennsylvania Museum at 32nd and South has almost 30,000 clay tablets inscribed in ancient Sumerian and Akkadian cuneiform.

concepts. Sumerian was the language of Mesopotamian accounts, whereas ancient Egyptian was the language of Egyptian priests. In fact, the ancient Egyptian name for their written language, which we call hieroglyphics, translates as "*the writing of the words of god*".[3]

The technology of written language is also important. Writing on a cave wall works, but it's not very portable. Writing on stone tablets is a little better, but imagine trying to send a detailed report of a military battle or a lengthy harvest report from one city to another on stones. The Sumerians invented cuneiform – writing with marks on clay tablets. The Egyptians wrote on papyrus – a primitive paper-like material formed from reeds that grow in wetlands along the Nile. Papyrus was cheaper, lighter, and easier to use; cuneiform tablets lasted longer. As a result, we have more early writing samples today in Sumerian than in the earliest Egyptian, even though more was probably written in Egyptian because of the ease of use of papyrus.

We can see many parallels in the earliest written languages to issues we still face with computers today. The way in which we represent information affects how we use language, or other tools, to process the information. This link between the way we represent or store data and the way we process data – the connection between the format of data and the algorithms we use to process data – is as important today as it was when written language first appeared. So, too, is the choice of the technology we use to store data. Logograms or phonograms? Clay or Papyrus? Apple or Android? PC or tablet? Of course, we have an advantage today because of our scientific approach to making such decisions. That's the *science* part of Computer Science.

Numbers

The history of numbers dates back much farther than the history of written language. In fact, there is a lot we don't know about the history of numbers since the use of numbers does predate written language.

The earliest recorded numbers that we have evidence of today are marks on bones, wood, or stone to count things. These marks, called tally marks, usually had one mark for each item, so seven notches on a bone, for example, might mean that seven nights had passed. We can find tally marks on many ancient human artifacts. The presence of these tally marks shows us that early humans counted things. They most likely also performed simple arithmetic, such as addition and subtraction.

The **Ishango Bone**[4] is an important ancient artifact – 20 to 25 thousand years old – that may indicate a more sophisticated early understanding of mathematics. It is now in the *Royal Belgian Institute of Natural Sciences* in Brussels, but was found in the *Democratic Republic of the Congo*, then the *Belgian Congo*, in the late 1950s. It is a baboon's fibula fashioned into a tool with a jewel embedded in one end, most likely for engraving other items. There are three columns of tally marks engraved on the bone, which may indicate an awareness of multiplication or other more sophisticated numerical concepts. Some experts have described a pattern of prime numbers in the tally marks, while others have linked

[3] Sherman, Dennis. *The West in The World, 2/e*. New York: McGraw-Hill, 2004. pg. 84

[4] A 2011 *CNN* video report on the Ishango Bone is on *YouTube* at: http://www.youtube.com/watch?v=r_YkdxdAGX8

the patterns to lunar calendars. Similar artifacts have been found in Europe and Asia, although the Ishango Bone, and a similar 35,000 year old tool from southern Africa called the *Lebombo Bone*, are among the oldest.

The real meaning of the marks on the Ishango Bone is still debated, but clearly people were storing and processing numerical information – and developing new methods and technologies to do so – a thousand generations before written language.

The story of how humanity got from the Ishango Bone to the iPhone is long and complicated. The cast of characters alone will fill many volumes – the Egyptians, the Sumerians, the Babylonians, Thales, Pythagoras, Euclid, Archimedes, al-Khwarizmi, Chu Shih-Chieh, Brahmagupta, Joost Burgi, John Napier, Blaise Pascal, Gottfried Leibniz, George Boole, Augustus De Morgan, Charles Xavier Thomas de Colmar, Charles Babbage, Ada Augusta Lovelace, Herman Hollerith, William S. Burroughs, William Henry Eccles, Vannevar Bush, Kurt Gödel, Alan Turing, Claude Shannon, John Atanasoff, Howard Aiken, John Mauchly, Presper Eckert, William Shockley, John Bardeen, Walter Brattain, Jack Kilby, John Backus, Grace Hopper, Douglas Engelbart, Gordon Moore, Robert Noyce, Bill Gates, Steve Wozniak, Steve Jobs, and Tim Berners-Lee – just to name a few. Many of the names and parts of the story – such as the real use of the Ishango Bone – may never be known. Around 200 BCE, the Chinese Qin emperor ordered that all books without a royal warrant should be burned, erasing much of the history of early China. Other more recent events, such as the Second World War, have obscured other parts of the story.

Key Terms in Chapter 4

After completing this chapter, You should be able to define each of the following key terms:

Chapter Questions

1. What is the difference between using the *while* statement and using the *if* statement in Java?
2. What is the difference between a pre-test loop and a post-test loop? Which of the two is generally preferred and why?
3. What are the four parts of every pre-test loop? How does this compare with the parts of a post-test loop?
4. What is initialized and tested in a loop? What needs to be initialized, then updated and tested each time through a loop?
5. What is the difference between the way *while* and *do…while* are used in looping? When is a semicolon used after a *while* statement?
6. What is the difference between a continuation condition and a terminating condition? How are they related to one another? Which is used in while statements in Java? How is a repeat…until statement implemented in Java?
7. What can be used as a sentinel to signal that a loop should be terminated? What is often used as a sentinel when reading in data from a data file? How is this related to a trailer value?
8. What is used as the loop control variable, or counter, in a count-controlled loop? What three things do we need to know about the counter? How can we end up with an infinite loop when trying to establish a count-controlled loop?
9. When does a loop control variable go out of bounds? What does it mean to go out of bounds?
10. In what two ways can we stop a runaway program, such as an infinite loop, running in NetBeans.
11. What values can be used for the increment in a count controlled loop?
12. What is the information in parentheses following the word *for* in a *for* statement called? What does it contain?
13. What can be used as initialization and update portions of a *for* header?
14. What data types does the JLS specify must be used with a *for* statement?
15. Why are the variables *i*, *j*, and *k*; or *m* and *n* often used as counters in count controlled loops?
16. What methods can be used instead of *print* and *println* to print formatted output? What is used to tell these methods how to format the output?
17. What do the following conversion codes mean in a format String: f, e, d, and s? What do the numbers used with each of these in a print specifier mean? What does the conversion code *n* mean when used as a print specifier?
18. What do format flags do when used with a print specifier? How can we left justify output using a format flag? How can we specify that a plus sign should be printed in front of positive numbers by using a format flag? How can we put comas in numbers equal to 1,000 or higher?

19. What method is used to generate pseudorandom numbers in Java? What is a pseudorandom number?
20. What is the range of numbers generated by Java's pseudorandom number generator? How can we generate numbers in a specified range, such as integers from 1 to 6?

Chapter Exercises

1. Multiplication Table

   A set of nested *for* loops can be used to print a multiplication table using *System.out.printf* statements.

```
     1   2   3   4   5   6   7   8   9
  1  1   2   3   4   5   6   7   8   9
  2  2   4   6   8  10  12  14  16  18
  3  3   6   9  12  15  18  21  24  27
  4  4   8  12  16  20  24  28  32  36
  5  5  10  15  20  25  30  35  40  45
  6  6  12  18  24  30  36  42  48  54
  7  7  14  21  28  35  42  49  56  63
  8  8  16  24  32  40  48  56  64  72
  9  9  18  27  36  45  54  63  72  81
```

   Write a program to print such a multiplication table.

2. Celsius to Fahrenheit Table

   Write a program to print a Celsius to Fahrenheit conversion table that a traveler could use to look up temperatures while traveling. The formula is $F = (\frac{9}{5}C) + 32$.

   The table should include integer Celsius degrees from 0 to 40. Your program does not need to use integers, but the table should display the Celsius temperature, then the Fahrenheit accurate to one-tenth of a degree. The first four entries from the table are shown below:

Celsius	Fahrenheit
0	32.0
1	33.8
2	35.6
3	37.4
4	39.2

3. Guessing Game

   The guessing game that asks a user to pick a number from 1 to 100 is described in Chapter 3, on pages 20 and 21 with pseudocode, sample output, and a flowchart. It uses branching, repetition, and the generation of random numbers. Your task is to create the game in Java. You should submit the game along with your programming lab report as described in Chapter 3.

4. Printing Patterns

   Nested for loops can print patterns of stars, such as the example below:

```
for(i=1; i<=5; i++) {
    for(j=1; j<=10; j++) {
       System.out.print("*");
    } // end for j
    System.out.println();
 }  // end for i
```

   Output from this program:

```
**********
**********
**********
**********
**********
```

   Your task is to write similar Java code to print each the following patterns:

```
*           *   *            *         *
**         **   ***        ***       ***
***       ***   *****     *****     *****
****     ****   ******   *******   *******
*****   *****                        ***
                                     ***
                                     ***
```

5. International Gymnastics Scoring

   We wish to write a program to quickly calculate gymnastics scores for an international competition. The program should have a loop to ask the user for the scores from each of the six judges, numbered 1 through 6 for anonymity. Your program should add each score to a sum inside the loop, then calculate and display the average score after the loop is finished. Scores are in the range 0 to 10 with one decimal place, such as 8.5. The average should be displayed with two decimal places.

   Here is a sample of the output:

```
Score for Judge 1:  8.5
Score for Judge 2:  8.7
Score for Judge 3:  8.4
Score for Judge 4:  9.1
Score for Judge 5:  8.9
Score for Judge 6:  9.0
The average score is:  8.77
```

6. Write a program similar to number 5, above, but in this case it should work no matter how many judges there are. The user will enter a -1.0 as the score to signal that there are no more judges.

7. Fibonacci Numbers

   The Fibonacci sequence is one of the most commonly found patterns in nature. It starts with 0 and 1, then each term in the sequence is generated by adding the previous two terms. The first few terms are 0, 1, 1, 2, 3, 5, 8, 13, 21, and 34. Numbers that appear in the sequence are called Fibonacci numbers. Write a program that uses a loop to generate and display the first 20 Fibonacci numbers. Your program can start by printing 0 and 1, then work from there for the next 18 Fibonacci numbers.

8. Trigonometric Table

   Write a program to print the sine and cosine for angles between 0 degrees and 90 degrees in increments of 10 degrees. Your program should use the functions in the Math library – Math.sin() and Math.cos(). The trigonometric functions work in radians, so for each angle, your program will need to convert degrees to radians, then find and print the sine and cosine.
   For `example    y = Math.sin(Math.toradians(x) );   // calculates the sin of x degrees`

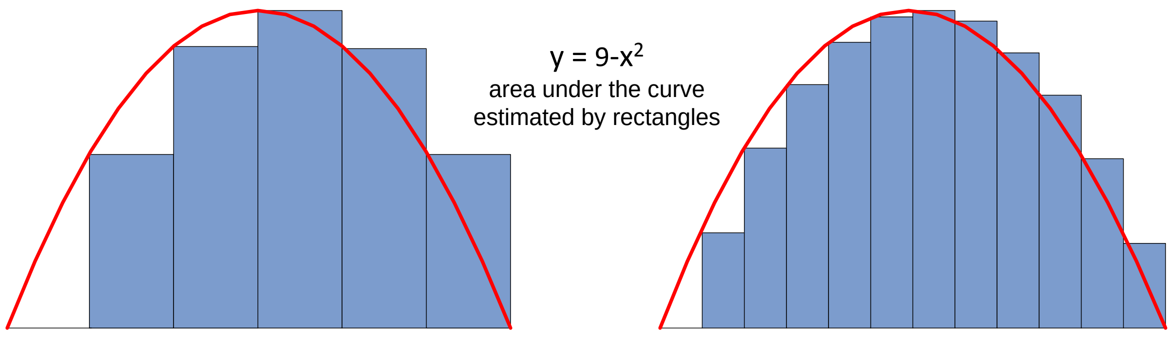
```
angle     sin     cos
   10   0.174   0.985
   20   0.342   0.940
   30   0.500   0.866
   40   0.643   0.766
   50   0.766   0.643
   60   0.866   0.500
   70   0.940   0.342
   80   0.985   0.174
   90   1.000   0.000
```

9. Test for Divisibility
   If the remainder from dividing A by B is zero, then A is a multiple of B (A is evenly divisible by B). For example, the remainder from dividing 6, 9, or 12 by 3 is 0, which means 6, 9, and 12 are all multiples of 3. Write a program to print the numbers from 1 to 25, and on the line next to each number, print messages if the number is a multiple of 2, 3, or 5. Here is a sample of the output:

```
1
2    multiple of 2
3    multiple of 3
4    multiple of 2
5    multiple of 5
6    multiple of 2   multiple of 3
7
8    multiple of 2
9    multiple of 3
10   multiple of 2   multiple of 5
```

10. Estimate of a Definite Integral

    A loop in a computer program can be used to approximate the value of a definite integral. We can break the integral into rectangles, find the area of each rectangle then add the areas together to get the total area, as shown in the accompanying diagram. In practice, we could make the width of the rectangles progressively smaller, then run the loop again, until we achieved the desired accuracy.



As the width of the rectangles gets progressively smaller, their total area gets closer to the total area under the curve. This is an old method that predates Calculus. Both Leibnitz and Newton used this method, eventually developing calculus from the concept of infinitesimals, which, in the case of our rectangles, would amount to asking, what would the total area be if we had very many rectangles with very small width?

This was tedious by hand, but we now have computers. We can set the width of the rectangle. The height of the rectangle is the y value at point x. So for example, if we need to solve:

$$\int_{+3}^{-3} 9 - x^2\, dx$$

The starting point is -3, the ending point is +3, and the height of the rectangle at each value of x in between the two is $9 - x^2$ where y at each x is the height of the rectangle at that x. We can start with a width of 0.1, and write a loop like this:

```
totalArea = 0

for x = -3 to +3 step 0.1   // x is -3.00, then -2.9, then -2.8, etc.
  {
    y = $9 - x^2$;          // the height is the y value at each x
    area = y * 0.1;      // the width is the increment, in this case 0.1
    totalArea = totalArea + area;
      }
```

When the loop is finished, totalArea is the total area of the rectangles, which is an approximation of the area under the curve.

Your task is to write a Java program that implements this algorithm for the definite integral used in this example. Run the loop several times, each time with a smaller increment, until your result does not change by more than .01 from the previous run of the program.

Your lab report should explain what you did, and the result of your work.

— End of Chapter 4 —