CSC142 Supplemental Reading: Week03

Contents

1 Boolean Variables and Expressions

Recall that type ***boolean*** is one of the 8 primitive types in Java. A variable or expression of type *boolean* will have one of only two possible values: ***true*** and ***false***. Java has many operators that evaluate to type *boolean* and a number that operate on *boolean* values. This note describes 9 of these operators, listed in two categories.

1.1 Relational Operators

Relational operators are used to compare primitives. There are six relational operators in Java:

Relational Operator	Meaning
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

Notice that the operator for testing equality (==) is **different** from the assignment operator (=).

When relational operators are used to compare numbers, the results are exactly what you would expect:

Given:

```
int x = 2;
int y = 1;
```

Expression	Value
x > y	*true*
x < y	*false*
x < 2	*false*
x <= 2	*true*
x == 2	*true*
y == x	*false*
x != y	*true*
x != 2	*false*

To hone your logical thinking, it is useful to think of the relational operators as three pairs of opposites:

Pairs of Opposites
`<` and `>=`
`>` and `<=`
`==` and `!=`

So, the exact opposite of `x < y` is `x >= y`.

The last two operators on this list (== and !=) are sometimes called **equality operators**. They can be used with any kind of data, whereas the other four relational operators are limited to the 6 numeric types plus type *char*.

1.1.1 Comparing Characters

Relational operators may also be used to compare *char* values (characters). In this case, the Unicode character codes determine the ordering. There is no need to memorize any character codes, but it is useful to know some features of the code set:

- The space character has a very low character code (32) below all letters and digits
- The upper case letters are sequential. For example, the code for 'A' is 65, the code for 'B' is 66, the code for 'C' is 67, etc.
- The lower case letters are sequential and this set comes AFTER the upper case letters. For example, the code for 'a' is 97, the code for 'b' is 98, etc. Notice how these codes are larger than the codes for the uppercase characters.
- The digits are sequential. The code for the character '0' is 48, the code for the character '1' is 49,etc. **Be careful!** We are talking type *char* here, '0' and '1', not the numbers 0 and 1.

Here are a few examples:

`'X' > 'A'` is ***true*** (because the code for 'X' is bigger than the code for 'A')
`'3' > '0'` is ***true*** (because the code for '3' is bigger than the code for '0')
`'b' > 'W'` is ***true*** (because all lower-case letters have higher codes than any upper-case letter)

You can determine the Unicode character code for a *char* by casting to type *int*. For example:

```
(int)'A'
```

evaluates to 65.

1.1.2 Comparing Object References

Two of the relational operators (== and !=) may also be used with object references. However, it is important to understand what they test. Comparing object references with == or != compares the references (memory addresses) directly. That means that we are testing to see if they are referring to the same object. We can call this test **identity equality**. This is distinct from **content equality** -- a test to see if two objects are considered to be equivalent. For example, hold two different $20 bills in your hand. They are obviously distinct objects -- two separate bills. However, they are considered to be equivalent to each other. Note that they may be equivalent even if they have slight differences (wear, etc.). A class may include code that specifies how to perform the content equality test, as we will learn in a few weeks.

Comparing object references	
Identity Equality:	**obj1 == obj2**
Content Equality:	**obj1.equals(obj2)**

Here's a very simple program that demonstrates the difference:

```
String s1 = "Hello";
String s2 = "He";
s2 = s2 + "llo"; // s2 is now "Hello"
System.out.println( s1 == s2 ); // prints false
System.out.println( s1.equals( s2 ) ); // prints true
```

The variables s1 and s2 do not refer to the same String object, so the test for identity equality (==) is *false*. However, the two Strings are considered to be equal to each other from a content standpoint -- they both have exactly the same characters in the same order -- so the test for content equality (.equals) evaluates to *true*.

1.2 Compound Boolean Expressions

You can also make multiple tests in one boolean expression. You do this by combining 2 or more boolean expressions into a larger one. For this you need to use **boolean operators**. Here are some examples:

Compound Boolean Expressions
`x == 55 || y < 23`
`userHeight > 60 && userAge >= 8`
`! (size == 1.3)`

There are three boolean operators as seen in the examples above: **&& ('and'), || ('or'), and ! ('not')**. (Note that the '**or**' operator (||) is written using two "vertical bar" characters with no space between them.) The "vertical bar" character is located on the key just above the 'Enter' key on a normal keyboard -- it is shift-backslash. Some keyboards may show a gap in the bar symbol.) The boolean operations have the same logical meaning as the words 'and', 'or', and 'not' do in English. For example, the first expression in the table above will be true if x == 55 OR it would also be true if y < 23. For the second expression, both simple expressions must be true: userHeight must be greater than 60 AND userAge must be greater than or equal to 8 for this to be true. In the third example, the ! operator turns a *true* into a *false*, and a *false* into a *true*. These three operators may be used **only** with boolean operands.

Here are 3 **truth tables** to help explain how these logical operators work. The *true* and *false* in the tables below can be any boolean expression:

&& (and)	
Expression	**Value**
true && true	***true***
true && false	***false***
false && true	***false***
false && false	***false***

|| (or)	
Expression	**Value**
true || true	***true***
true || false	***true***
false || true	***true***
false || false	***false***

! (not)	
Expression	**Value**
! true	***false***
! false	***true***

Here are some expressions and their values:

Given:
```
int x = 3;
double y = 17.5;
```

Expression	**Value**
`x > y && y < 20`	***false***
`x != 3 || y != 3`	***true***
`y > x && x == 3`	***true***
`!(x > y)`	***true***
`x > y || x < 10`	***true***

1.2.1 Operator Precedence

Here is a summary of the precedence for these operators: arithmetic, relational, and logical:

logical not (!) [highest precedence -- executed first]
arithmetic operators
relational operators (< >= > <=)
equality operators (== !=)
the remaining logical operators
assignment operators [lowest precedence -- executed last]

For more detail, take a look at the complete operator precedence table in last weeks' supplemental reading section.

You can create large complex tests by using one or more boolean operators. As a challenge, decide if the program code below will show a message.

```
int x = 4;
int y = -2;
if ( x * y < 0 && ( x > 0 || y > 0 ) )
   System.out.println( "Yes, it works!" );
```

1.3 Common Errors Writing Compound Conditions

A **compound condition** is one that makes use of logical operators to join 'sub conditions'. We use constructions like this all the time in our daily lives, and the logic works the same way. However, students new to programming sometimes make mistakes translating spoken conditions into code. Consider the following statement: *"If the number is between 7 and 11, tell the user that they win!"* How would you write the logical expression to meet these requirements? Here are two **common errors**:

```
( 7 <= number <= 11 )
( number >= 7 && <= 11 )
```

A **correct** form of the expression is

```
( number >= 7 && number <= 11 )
```

Notice in the correct expression that each relational operator has its own set of values to compare, and the logical operator is used to join these sub-expressions.

Here's another one to try: *"If either of the two occupants of the hotel room is at least 55 years old, use the 'senior citizen rate'."* How would you write the expression?

Here is a correct version:

```
( age1 >= 55 || age2 >= 55 )
```

One more very common error in logical expressions is to mistakenly use the **assignment operator (=)** in place of the **equality operator (==)**. This is not always a syntax error, but will not produce the desired results, so check your code carefully!

1.4 Flag Variables and Predicate Methods

A *boolean* variable may be used to remember that some event occurred or some condition was satisfied so that information can be used at a later time. A variable like this is called a **flag variable**, or just a **flag**. We will use flags when we explore applications of loops.

Instance variables of type *boolean* are initialized to *false* by default.

A method that returns type *boolean* is called a **predicate method**. The **equals** method used to test for content equality is an example of a predicate method.

2 Additional Notes on 'if' Statements

2.1 The if Statement with a Block of Code

If you need to have two or more statements executed conditionally, they may be enclosed inside curly braces, which define a **block of code**. If the boolean expression is *true*, **all statement in the block of code** will be executed in sequence; if the boolean expression is *false*, the block of code will be skipped.

```
public void checkNum2( double num ) {
   if ( num > 0 ) {
      System.out.println( "The number is greater than zero" );
      // do something else
   }
}
```

2.2 Nested if Structures and Multi-way Selection

Your textbook does a good job explaining how do use nested *if* statements to perform multi-way selection. Here are additional notes on multi-way selection:

- A '**multi-way selection**' situation is one in which <u>**one**</u> **choice** is made from a set of three or more possibilities. There are always **multiple conditions** in multi-way selection structures.
- When nested *if* statements are used for multi-way selection, the final '*else*' clause in the *if* pattern serves as a **'default' action** that will be executed if none of the conditions is matched. This final '*else*' clause is optional. Here are the important points:
    - The switch statement is another control structure used to perform multi-way selection. In some situations, the switch statement may be a better choice, but if statements can always get the job done.
    - If a final '*else*' clause <u>**is**</u> included, then <u>**exactly one**</u> **action will be executed** during each pass through the *if* structure. It will be either the action associated with the <u>**first**</u> **condition** that is satisfied, or the default action if no condition is satisfied.
    - If <u>**no**</u> final '*else*' clause is included, then **it is <u>possible</u> that <u>no</u> action will be performed** -- this will happen if no condition is satisfied. We can say for sure that **either no action or one action will be executed** during each pass through the *if* structure.

Control structures may be nested in any number of different ways. The only rule that needs to be followed is this: **the 'inner' control structure must be contained completely within one 'action' of the outer control structure**. Not all nested *if* statements qualify as simple 'multi-way' selection -- the structure could be much more complex. If you need help visualizing the structure required to solve a particular problem, consider drawing a flow chart.

2.3 if Statements in Methods that Return a Value

When *if* statements are used in a method, they create multiple paths that program flow can take through that method. For methods that return a value, one important rule to keep in mind is that **every path through the method must end with a *return* statement (or a *throw* statement)**. This makes sense when you realize two things: 1) the value to be returned must be specified somehow, and 2) when a *return* statement is executed, that ends the method call -- control is transferred immediately back to the calling statement.

It is common for an *if* statement to be used to provide an **early exit** from a method when some special condition is met. For example, consider the following method from the LineSegment2 sample program:

```
public double slope() {

   double dx, dy;
   dx = this.p1.getX() - this.p2.getX();

   if ( dx == 0 )
      return Double.POSITIVE_INFINITY;

   dy = this.p1.getY() - this.p2.getY();
   return dy / dx;
}
```

There are two paths through this program. When the value of **dx** is zero, the method returns the special value Double.POSITIVE_INFINITY and the method call immediately ends with this statement. Otherwise, execution continues and the last two lines of the method are executed. Notice that, as required, each path through the method ends with a *return* statement. Of course, frequently multiple paths through a method will end at the same *return* statement, which is fine. In fact, it is considered better programming practice in some circles for every method to have just one *return* statement. How could you re-write the method above so that it has just one *return* statement? Look for the method called 'slope1' in the LineSegment2 sample program in this week's samples folder for one possible solution.

Here is a simple method that has 3 possible paths, two of which end at the same return statement:

```
public String showNumber( int num ) {

   String s;
   if ( num == 0 )
      return "Zero!";
   else if ( num < 0 )
      s = "Minus ";
   else
      s = "Plus ";
   return s + num;
}
```

2.4 Testing Programs with Selection Structures

When testing a program that uses selection structures, test cases must be chosen to carefully check the program's operation **right around the threshold of each of the conditions**. Consider the following code as an example:

```
if ( score >= 90 )
   System.out.print( "Grade: A" );
else if ( score >= 80 )
   System.out.print( "Grade: B" );
else if ( score >= 70 )
   System.out.print( "Grade: C" );
else if ( score >= 60 )
   System.out.print( "Grade: D" );
else
   System.out.print( "Grade: F" );
```

To fully test the grade examples above, you'd want to test 59, 60, 69, 70, 79, 80, 89 and 90 to make sure they fall into the right categories. Notice that these values are just below and just above each decision point. It is very easy for **'off-by-one' errors** to creep into conditions (for example, using '>' when '>=' is needed), and the only way to catch these **logic errors** is to test the code thoroughly.

3 The 'switch' Statement

The ***switch* statement** is another control structure used to perform multi-way selection when the decision can be defined by **matching specific values** of a variable or expression. The ***switch* statement** does not make use of logical expressions like an *if* statement -- it actually works in a very different way. The *switch* statement tries to **match** the value of an expression with one of several specific values listed in *case* statements. The action associated with the **first match** is executed.

Here is a simple example that illustrates the syntax of the switch statement:

```
public void checkGrade( char letterGrade ) {

   String message;
   switch ( letterGrade ) {
   case 'A':
      message = "very good";
      break;
   case 'B':
   case 'C':
   case 'D':
      message = "OK";
      break;
   default:
      message = "not very good";
   }
System.out.println( "Your grade is " + message );
}
```

Here are some things to notice about this example:

- The identifiers ***switch***, ***case***, ***default*** and ***break*** are all Java keywords.
- The basic syntax is keyword ***switch***, followed by required parentheses containing a **switch expression**, followed by curly braces defining a block of code. The block of code contains a series of ***case*** statements and (optionally) a ***default*** statement, interspersed with executable statements.
- The switch expression must be type ***int*** or type ***char*** only. With Java 7, you can also use Strings.
- Every **case label** must be a constant expression -- an expression made up of literal values and/or named constants (*final* variables) only.
- A ***break*** statement ends the *switch* instruction and transfers control to the first line of code following the *switch* instruction's block -- the println statement in this example. In order to keep all actions mutually exclusive, **be sure to end every action with a *break* statement**. However, a *break* statement is not needed at the end of the *default* action, because it is already at the end of the block.
  Also, notice how there is no break statement for cases 'B' and 'C'. With the syntax, the 'B', 'C', and 'D' cases will all have the same action: the "OK" message.
- **The *default* clause is optional.** If default is included, then **exactly one action** will be executed -- the one associated with the first matching case, or the default action if no case matches. If default is not included, then **one action or no action** will be executed.

- **Basic operation:**
    - When the *switch* statement is encountered, the switch expression is evaluated. Its value is then compared to each case label in sequence.
    - Until a match is found, only case labels are checked -- all other statements are skipped.
    - As soon as a match is found, the execution of normal statements begins (and all further case labels are ignored).
    - Normal statements continue to be executed in sequence until a *break* statement is encountered or the end of the block of code is reached.
- **Execution example:**
    - Let the value of letterGrade be 'C'.
    - When the *switch* statement is executed, case 'A' does not match, so the two executable statements following it are skipped.
    - Case 'B' is checked and also does not match.
    - Case 'C' is checked and matches. Now only executable statements are executed; therefore, case 'D' is skipped.
    - `message = "OK";` is executed.
    - `break;` is executed -- this ends the switch statement and transfers control the first line of code after it.
    - `println` is executed and shows the message.

3.1 switch or if?

Here are some guidelines for choosing between an *if* statement and a *switch* statement for a particular selection application:

- **A *switch* should be used only for multi-way selection.** If there are fewer than three possible actions, use an *if* statement.
- **The choice of an action must be determined solely by the value of a single variable or expression of type *int* or *char*, *or String* .** If the selection criteria are more complicated, or if another data type is involved, use *if* statements.
- Each of the possible values of the switch expression must be specified by **constant expression** used as a case label. If the possible values are variable, then a *switch* statement cannot be used.

In situations where these criteria are met, a *switch* statement may be a good choice. A *switch* statement can be easier to use than nested *if* statements, and it makes the criteria for selection extremely clear.

4 Repetition

Our coverage of control structures began with selection structures and now continues with repetition structures (also called iteration structures or loops). One of your goals for this quarter should be complete mastery of the operation of all Java control structures.

4.1 Repetitive Logic

Let's think for a moment where we find repetition in our daily lives.

- If we try to call someone and it is busy, we repeat the process until we get frustrated or we get through (notice the logic operator OR in there!).
- When we balance our check book, we repeatedly add or subtract transactions until we have processed them all.
- When we want to access our account at an ATM, we enter a password and re-enter it when we get it wrong.
- When we look through a photo album, we examine one photo at a time, turning each page to see if there are more photos to see, until we reach the end. (Notice that it is not necessary to know in advance how many photos there are in the collection.)
- When we tell a 2 year old to count from 1 to 10, she says "1, 2, 3, 4, ... 10", a repetitive process.

Let's think about the 2 year old and the process her brain goes through.

- Step 1 -- Her mind starts at number 1 (because that's where you told her to start)
- Step 2 -- Is the number <= 10? If yes continue to Step 3, otherwise STOP (jump to Step 6)
- Step 3 -- say the number
- Step 4 -- go to the next number (we as adults know this is + 1)
- Step 5 -- go back to Step 2
- Step 6 -- DONE!! (can I have a cookie now?)

This is the algorithm to count to 10. The 2 year old knows it; we need to tell the computer how to do it. Let's rewrite the algorithm in pseudocode (not actual Java statements -- just 'English-like' statements that represent the logic of what we want to accomplish):

```
number = 1 // INITIALIZE -- where to start
while number <= 10 // LOOP TEST -- continue or not
print number
number = number + 1 // UPDATE
go back to the top of the loop and repeat
execute this line when the loop finishes (and then continue with the rest of the algorithm)
```

When the repetitive process (or loop) is done, execution jumps to the line after the loop.

All loops have the 3 pieces, or **elements** you see commented: **INITIALIZE**, **LOOP TEST** and **UPDATE**. The loop test is the easiest one to think about first. What needs to be true in order to repeat? (Think back to the examples above).

The variable that has its value tested controls the repetitions of the loop. This variable is known as the **loop control variable**. The loop control variable must be initialized -- be given a starting value -- before it is tested. That's why the initialization must come before the test. In some situations, there may be more than one loop control variable.

Finally, there must be an update -- something that changes the value of the loop control variable. We want the loop test to be false at some point, so the variable's value must change; at some point that value will cause the loop test to fail. The update needs to be inside the body of the loop!

If the loop test never fails (meaning the update is not working properly) this is known as an infinite loop. It is an error in your algorithm. Think again about the 2 year old counting to 10. What if the child doesn't go to the next number? She'd just repeat "1...1...1..." forever!

4.2 The Five Elements of a Loop

There are five key elements that should be considered during the design of any loop, and we have just finished discussing three of them. Here is a complete definition of all five:

- **Initialization**- Set up the proper conditions for the start of the loop.
  - Identify the **loop control variable**. This is the variable that will have its value changed with each iteration (repetition) of the loop. It must be assigned an initial value before the loop begins. Make any other initializations needed for the loop to operate properly.
- **Loop Test** - Specify the conditions that must be satisfied for the loop body to be executed.
- **Main Work** - Statements to be executed during each iteration of the loop. (Saying the number in the example of the child counting.)
- **Update** or "Change" or "Making Progress Toward Termination" step.
  - The value of the **loop control variable** should be **changed** inside the body of the loop.
  - The pattern of changes must be such that the loop eventually terminates! In other words, the loop must continually make progress toward termination. In the example above, the number is repeatedly increased by one, making it get closer to 10 with each iteration of the loop.
- **Finalization** - Add any statements needed after the end of the loop to properly wrap things up (e.g., asking for a cookie -- an optional step.)

The **order** in which these elements appear in a loop **may vary**. Initialization will always be first, and finalization (if used) will always be last (as their names imply), but the other elements may appear in any order -- depending on the iteration structure used and the design of the algorithm being implemented.

4.3 Accumulator variables

Think about the checkbook example from above. As you process each individual entry, you keep track of a running total -- how much you have in the account so far. Once you've processed all the data you then have the final account balance. Same with programs.

Checkbook example:

If I have the following entries:	Here is the Accumulated Value:
deposit $40	$ 40.00
check for $19.95	$ 20.05
check for $6.05	$ 14.00
deposit $80	$ 94.00

If I want a program to add the numbers 1 to 10, the program has to **accumulate** the sum and keep track of how much it has added up so far. Here is some pseudocode:

```
sum = 0 // accumulator variable being initialized
count = 1 // loop control variable being initialized
while count <= 10
sum = sum + count // add the new value to the running total (accumulate)
count = count + 1 // update for the loop
repeat
```

Notice there are two variables now, each with a different job. The variable count is the one controlling the loop (notice that it is the variable checked in the loop test). The variable sum is known as the **accumulator**.

4.4 Design Patterns

Think of how you might change this to add the numbers from 200 to 500? You would change the initial value of the count variable, and the test for the upper limit. What if you want to calculate the sum of all numbers from 3 to 330, counting by 3s? You'd change the initial and test values for the count variable, and you'd add 3 to the count variable in the update. Notice that you only have to change some constants. What if I said multiply instead of sum up? You'd only change an arithmetic operator (and the accumulator's initialization). The **design** of the loop is the same. This is important in programming. You may be asked to write many different loops; but the patterns will all be similar.

4.5 Definite and Indefinite Loops

The counting loops above are known as **definite loops**. You can tell from the algorithm how often they will repeat. The programmer may not know in advance the actual number of repetitions that will occur (that may depend on the data). However, in every definite loop, the number of repetitions could be calculated from data values before the loop begins.

An **indefinite loop** is one for which the number of repetitions cannot be known at the time the program is written or calculated from data values. This situation arises a lot. Consider the example at the beginning of this note of repeatedly calling someone until the phone is answered. This process is repeated until some condition is satisfied -- the phone is answered -- and it is not possible to know in advance how many repetitions will be required.

The ATM password example illustrates an important situation where indefinite loops are often used: **input validation**.

The photo viewing example illustrates the process of **iterating through a collection** -- another situation where an indefinite loop is often used. We will study design patterns related to collections soon.

5 Loop Statements

Loop Control Structures: *while*, *for*, and *do* ... *while*

Java provides three different control structures for creating loops. These are the ***while***, ***for***, and ***do* ... *while*** statements. **Any loop control structure can be used to create any loop in a program**, no matter what its purpose (with the possible addition of some if statements). This note explains how the three loop control structures work and how to choose the best control structure for a specific situation.

5.1 The while Statement

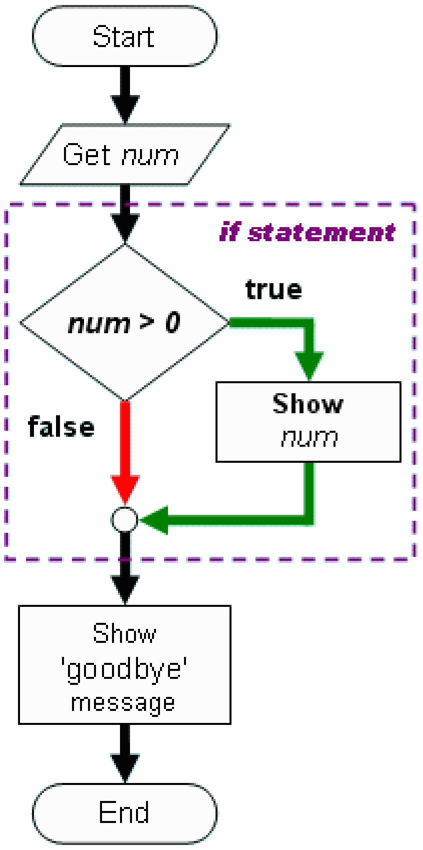
The basic concept behind a loop in programming is simple. **In a loop, a set of statements is <u>repeated</u> as long as a specified <u>condition</u> is satisfied (evaluates to *true*). As soon as the condition is no longer satisfied, the loop terminates.** The structure of a *while* loop most clearly illustrates this concept.

(continues…)

Let's compare the behavior of a *while* loop with that of a simple, single-alternative *if* statement. In an *if* statement, a set of statements is executed if a specified condition is satisfied (evaluates to *true*); otherwise the statements are not executed. This has a lot in common with the behavior of a *while* loop -- the figures below use a flow chart to illustrate an example:

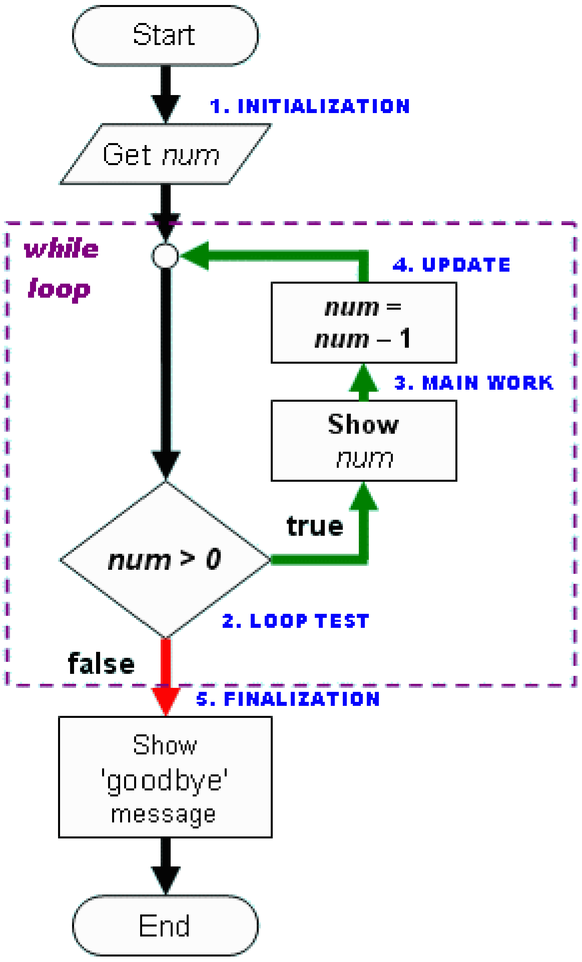
```
Scanner in = new Scanner( System.in );
System.out.print( "Enter a number: " );
int num = in.nextInt();

if ( num > 0 ) {
    System.out.println( "<br>" + num );
}
System.out.println( "<p>Goodbye..." );
```



```
Scanner in = new Scanner( System.in );
System.out.print( "Enter a number: " );
int num = in.nextInt();

while ( num > 0 ) {
    System.out.println( "<br>" + num );
    num = num - 1; // ( or num--; )
}
System.out.println( "<p>Goodbye..." );
```



Here are some important things to notice about these figures:

- The *if* statement and the *while* statement have very similar syntax -- a **condition** is specified inside parentheses after the keyword (*if* or *while*) that begins the statement, and that is followed by a **block of code** (a set of statements inside { }) called the 'action' of the *if* statement or the **'body' of the *while* loop**.
- The 'decision' is the first thing that happens in both statements -- follow the flow chart arrows to convince yourself of this. In a loop, this decision is called the **loop test**.

- If the value of the condition is *false*, the 'action' of the *if* statement and the 'body' of the loop are never executed. Follow the red arrows (also labeled 'false') of the flow chart to convince yourself of this.
- If the value of the condition is *true*, the 'action' of the *if* statement and the 'body' of the loop are executed. Follow the green arrows (also labeled 'true') of the flow chart...
- **Here's the big difference**: with an *if* statement, after the action is executed, control is transferred to the first line of code after the end of the *if* statement. With a loop, **after the body of the loop is executed, control is transferred back to the beginning of the loop (which is the loop test in a *while* loop)**. In this way, the statements in the loop may be repeated! This process keeps repeating (body / loop test / body / loop test...) until the loop test is *false*. Once the loop test is *false*, the flow of control **skips over** the body of the loop and continues on with the rest of the program.
- The variable that is tested in the loop test (**num** in this example) is called the **loop control variable**. **The body of the loop must include a statement that updates the value of the loop control variable.** Notice that there is one additional statement in the *while* loop example: '`num = num - 1;`' (the statement '`num--;`' would do the same thing). This statement ensures that the value of the variable num is moved ever closer to a point where the loop test will fail.
- What is the value of the variable **num** after the loop finishes? It would be 0, not 1. The last number shown on the web page is 1, but the final value of **num** is 0. The final value of the variable will be the value that made the test *false*.
- Each time the body of the loop is executed, we say that **one pass through the loop** or **one iteration of the loop** has occurred.

Notice that the **five elements of a loop** discussed in the Note on Repetition are labeled on the flow chart for the while loop above. Here is a reminder of their significance:

- **Initialization** - Set up the proper conditions for the start of the loop; give an initial value to the **loop control variable**.
- **Loop Test** - Specify the condition that must be satisfied for the loop body to be executed.
- **Main Work** - Statements to be repeated with each iteration of the loop.
- **Update** - Change the value of the **loop control variable** in such a way that the loop must continually make progress toward termination.
- **Finalization** - Add any statements needed after the end of the loop to properly wrap things up.

Hopefully you can see that the meaning of the keyword *while* is the same as in the English language. The word while means "as long as this is true". Once the expression is *false*, the loop is done, and the program executes the code following the loop.

The operation of the other two iteration statements is similar to the *while* statement, and the fundamental principles apply to all three kinds of loops.

5.2 The for Statement

Many loops are specifically written to count, and **the *for* statement is often a particularly good choice for a counting loop**. When used to write a counting loop, the *for* loop is easier to read because it puts the **initialization** and **update** elements together with the **loop test** in the loop 'header' -- right in the parentheses following the keyword *for*.

The illustration below shows a *while* loop used for counting and an equivalent *for* loop. Both loops would display the integers 1 through 10 inclusive.

// *while* loop that counts:
`int counter = 1;`
`while ( counter <= 10 ) {`
`    System.out.println( counter );`
`    counter++;`
`}`
	// *for* loop that counts:
	`int counter;`
	`for ( counter = 1; counter <= 10; counter++ ) {`
	`    System.out.println( counter );`
	`}`

Notice the 3 key elements of a loop in red, green and blue: the **initialization** of counter, the **loop test**, and the **update** step. In the *for* loop, these 3 parts (critical to any loop) are visually next to each other, separated by semicolons. Here is the basic format of the *for* loop:

```
for ( initialization_expression ; test_expression ; update_expression ) {
        // body of the loop
}
```

- The rule for the curly braces is the same for all other control structures: curly braces are optional if there is only one statement in the body of the loop.
- The execution of the loop is **exactly the same** as the *while* loop. That is, the initialization happens first, then the loop test. If the test expression is *true*, the body is executed, and then the update. (The update statement is not executed until after the body of the loop.) Following the update, we go back to the test and repeat.
- The **initialization** expression is an **assignment statement**.
- The **loop test** expression must be a **boolean expression**.
- The **update** is any expression that **changes the value of the loop control variable**. This includes using the operators ++, --, +=, *=, -=, /= and %=
- Any of the 3 expressions (initialization, loop test, or update) are optional. (It may not be obvious as to why you'd want to use this, but this way you know). However, the semicolons are not optional. There must always be 2 semicolons between the parentheses. (See also Note 1.)
- This point is important: the initialization happens only once. Students new to the for loop get confused since the initialization is written within the loop header itself. But INITIALIZATION HAPPENS ONLY ONCE AT THE BEGINNING OF THE FOR LOOP.

Both **the *while* loop and the *for* loop are called <u>pre-test loops</u>** because **the loop test is executed before the body of the loop**. This allows for the *possibility* that the body of the loop will *never* be executed. If the loop test evaluates to *false* the first time it is executed, then the body of the loop is never executed. So, **in a pre-test loop, the body of the loop is executed zero or more times**. Very often this is the desired behavior.

5.3 The do...while Statement

The ***do...while* statement** is a **post-test loop**. This statement is structured so that the body of the loop appears before the loop test when the code is written, and **the body of the loop is executed once before the loop test is executed for the first time**. This guarantees that the body of the loop will be executed at least one time. **In a post-test loop, the body of the loop is executed one or more times**. Sometimes, this is the behavior that is required.

Here is the basic format of the *do ... while* loop:

```
do
{
   body of the loop
} while ( test_expression );
```

- Nothing happens when the keyword *do* is encountered -- it's just used to mark the beginning of the body of the loop.
- The body of the loop is executed once before the loop test is encountered.
- As long as the loop test evaluates to *true*, control transfers from the line with the keyword *while* back up to the line marked with *do* -- the beginning of the body of the loop.
- As soon as the loop test evaluates to *false*, the program continues with the first statement after the end of the loop.

A *do ... while* example is shown in the next section.

5.4 Counting Loops

One of the simplest kinds of loops to understand is the counting loop. Counting loops are very widely used in programming, although that fact is not always obvious to the user. We will show examples where the counting is quite explicit to illustrate the concept.

Any of the iteration statements can be used to create a counting loop. Here is a simple example showing three ways to show the numbers from 1 to 10 in the Java console (each assumes that the variable count has already been declared):

```
for ( int count = 1; count <= 10; count++ ) { // INIT; LOOP TEST; UPDATE
    System.out.println( count ); // MAIN WORK
}
```

```
int count = 1; // INITIALIZATION
while ( count <= 10 ) { // LOOP TEST
    System.out.println( count ); // MAIN WORK
    count++; // UPDATE
}
```

```
int count = 1; // INITIALIZATION
do {
    System.out.println( count ); // MAIN WORK
    count++; // UPDATE
} while ( count <= 10 ); // LOOP TEST
```

Here are some important things to notice about these programs:

- Each program has an **initialization step** that reads 'count = 1;'. In these programs, ***count*** is the **loop control variable**. In a counting loop, **the loop control variable should always be initialized to the starting count**. If the loop counts up, this will be the lowest number; if it counts down, the highest number will be the starting count.
- Each of these programs has a **loop test** that reads 'count <= 10;'. In a counting loop, **the loop test should always check for the ending count**. When counting up, a test of '<=' or '<' is appropriate because the loop should keep repeating as long as the count is lower than the ending count. When counting down, a '>=' or '>' test is appropriate.
- Each of these programs has an **update step** that reads 'count++;'. This means 'increment the value of count by 1' (same as: count = count + 1;). An update step is always required in a loop. When counting down, 'count--;' could be used. It is also possible for a loop to count using some **interval** other than one. To count "by 2's", use an update step like this: 'count += 2;'.
- There is nothing special about the word 'count' in this example -- it is just the name of a variable. Any variable could be used.

Counting is not usually the primary purpose of a loop, but rather a means to achieving some other goal. The **main work** in the body of the loop determines what the loop accomplishes. Often, other variables are used in a loop, and three categories of use are so important that they have names...

5.5 Counters, Accumulators and Flags

The variable 'count' in the examples above is an example of **a counter -- a variable that has its value changed by a fixed amount with each iteration of a loop**. Counters are often used as loop control variables, but it is also possible for a counter to be used in a loop for some other purpose.

An accumulator is a variable used to 'accumulate' information (by having its value changed) during the execution of a loop. Accumulators are commonly numeric variables, but could also hold string data or any other kind of data.

A **flag** is a variable used to remember if a certain condition has been satisfied during the execution of the loop. Flags are usually boolean variables, and this is the same concept touched on earlier in the quarter.

NOTES:

1. It is possible for a *for* statement to have 2 or more initialization statements in the 'initialization' region inside its parentheses. These would be separated by commas, all before the first semicolon. It is also possible to have 2 or more update statements. These would be separated by commas, all after the second semicolon. Use this capability with care, as it can reduce the readability of a program. It is also a good design practice to **include in the *for* statement itself only those initializations and updates that are specifically related to the loop's repetition behavior**. That means include only initializations and updates of **loop control variables**. Initializations and updates related to the main work of the loop, such as calculating sums, should be done outside of the *for* statement. Extra initializations should appear immediately before the *for* statement; extra updates at an appropriate place in the body of the loop. Below is an example of a loop that has a complicated control structure. The variables x and z are both loop control variables, and so are initialized and updated in the *for* statement; sum is an accumulator, so it is initialized immediately before the *for* statement and updated inside the body of the loop. Can you figure out what the final sum will be?

```
int sum = 0;
for( int x = 18, z = 2; z < x; x--, z *= 2 ) {
   sum += z;
}
```

6 Indefinite Loops and Data Validation

6.1 Data Validation Loop Algorithms

Think about using an ATM. After you insert your card you must enter a password. What happens if you get it wrong? The ATM asks you to try again. If you get it wrong a second time? The ATM will ask again. Assuming there is no limit, this process will repeat while the user input is invalid. No counting or accumulation going on here, just repetition until the user gets it right. An algorithm for that might be:

```
input guess // user's first attempt at the password
while guess != stored password

display an error message
input guess // repeated attempts at the password

repeat
```

Can you find the **initialization**, **test** and **update** for this loop? The loop control variable is 'guess'. The initialization occurs at the first input statement (above the loop); the update occurs at the other input statement (within the loop). This type of loop is known as a **data validation** loop. While the data is *invalid*, the process repeats.

If you're ready to look at some code, jump here.

6.2 Input Loop Algorithms with Exit Sentinels

How many of you have been asked to enter a code over the phone (like a pin) followed by the '#' symbol (the pound symbol)? This symbol is an example of an **exit sentinel** or **exit code.** An exit sentinel is used to signal the program that the user is done entering input. It is used when the programmer has no idea ahead of time how much data the user has, so the user must signal when they are done.

Here is another example. Assume a program's job is to add up the total cost of grocery items input by the user. But how many items are there? The programmer has no way of knowing ahead of time. One user may have 2 items while the next user has 100. So the algorithm has to be flexible enough to stop whenever the user wants to. Take a look at the pseudocode below.

```
totalCost = 0
groceryItemCost = input the cost of a grocery item, or enter "stop" as the
      exit code
while groceryItemCost != "stop"
convert from string to number
totalCost += groceryItemCost // totalCost is an accumulator variable
groceryItemCost = input the cost of another grocery item, or enter stop as
      the exit code
repeat
```

What variable is controlling the loop operation? -- 'groceryItemCost'. Where are the initialization, loop test and update now? Remember, initialization happens before the loop and update happens in the loop body.

Note that the exit code must be a value that would not be a valid piece of data. You wouldn't want an exit code of 1.5 because a grocery item might cost 1.50. But you could have an exit code of -99 (no item will have a negative cost). Make sure that the exit code does not get processed as a piece of the data set.

The data validation loop and this last loop are examples of **indefinite loops**. From looking at the code, you cannot tell how often they will repeat. It is all up to the user. These loops may execute 0 or more times. Think about it, the user could enter a value that would cause the test to be false the very first time, so the body of the loop would never execute.

If you're ready to look at some code, jump here.

6.3 Data Validation Syntax: if Statement vs. Loop

Data validation can be done in different ways, depending upon the method used to obtain the data. In some situations, a value is presented, and all that can be done is accept it or reject it. For example, if a method is called and data is passed in through the parameter list, there is not much the method can do -- other than signal the problem -- if the data is invalid. Similarly, if a program is interacting with a user through an event-driven interface, and the data entered in a form element is invalid, the program can signal the problem by showing a message. It is then up to the user to make a correction and submit the data again. You have certainly encountered this when interacting with a program in a 'windows' environment (an event-driven environment). Tests like this would be done using an ***if* statement**.

If the user input is being obtained interactively -- for example, by using a Scanner object to obtain keyboard input -- then a **loop**must be used to repeatedly ask the user for the correct answer. In other words, you don't want the program to continue until the input is correct.

Here is an example of each.

6.4 Input Validation Using an if Statement

The method below calculates the total pay for an employee. This only makes sense if the number of hours is greater than or equal to zero. Otherwise, the method should signal an error, which is done by throwing an exception:

```
public static double calcPay( double hours ) {
   if ( hours < 0 )
      throw new IllegalArgumentException( "Negative hours not allowed" );
   double totalPay = hours * HOURLY_RATE; // assume HOURLY_RATE is a class constant
   if ( hours > 40 )
      totalPay += (hours - 40) * HOURLY_RATE / 2;
   return totalPay;
}
```

There is no need to "repeat" code here -- there is no way for the method to get a replacement value. The best that can be done is to signal the error and hope that it is 'caught' further up the line (more about catching exceptions in CSC 143).

6.5 Input Validation Using a Loop

This next example uses a Scanner and the standard input stream to ask the user for a number between 1 and 12. In this case, we must force the user to enter a valid number before the method can continue. We use a *while* loop control structure:

```
System.out.print("Enter the hour:> "); //notice use of print instead of println
int hr = keyboard.nextInt();
while (hr < 1 || hr > 12) {
   System.out.print("ERROR: bad input for hour. Enter an hour between 1 and 12:> ");
   hr = keyboard.nextInt();
}
```

This application of the while loop is known as an **indefinite loop**. We can't tell by looking at the loop how often it will execute. It all depends on the user. In fact, the body of the loop may **never** execute! Think about it for a minute, the user may enter a valid number on the very first try. The test of the loop would be false and so we'd skip the body of the loop and just continue with the simulation.

Notice that the loop test above checks two different ways in which the value of 'hr' might be bad. This illustrates an important point about performing validation using a loop. **All ways in which a variable's value might be bad must be checked in the same loop; you cannot use one loop after another.** This is necessary because the replacement value entered by the user might be bad in any of the possible ways, so they all must be checked every time you get a new value from the user.

6.6 Loop Syntax with Exit Sentinels

Here's an example from the EchoTest.java sample program. This program repeatedly accepts input from the user and echoes each word back to the user until the user enters a "0". "0" is the **exit sentinel** or **exit code**. Notice how the loop test in the *while* statement tests for the exit code:

```
Scanner in = new Scanner( System.in );
System.out.print( "Enter a word, or type 0 to quit: " );
String s = in.next();
while ( ! s.equals("0") ) {
   System.out.println( "You entered: " + s );
   System.out.print( "Enter another word, or type 0 to quit: " );
   s = in.next();
}
System.out.println( "Good-bye" );
```

Is this loop a definite or indefinite loop? Can you tell just by looking how many times it will repeat or is it based on the user? If you said indefinite loop, you'd be right!

An indefinite loop employing an exit sentinel is the perfect control structure for implementing an interactive **menu** of choices or commands in a program that communicates with the user by using 'standard input' / 'standard output' (System.in / System.out).

7 Problem Solving with Loops

Any time repetition is used in a program, a loop is the right tool for the job. Soon we will see loops applied to manipulate **collections** of information. To wrap up our review of loops, we will look at another category of problems that can be solved using loops: simple simulations.

7.1 Simulating a Process

Some real-world processes are naturally repetitive in nature. For example, if you owe money on a credit card and make a payment each month, your account balance is updated each month to reflect the accrual of interest and the payment made. Other processes can be modeled, or simulated, by approximating a continuous process (such as a ball flying through the air) as a series of discrete steps. Simulations like this are used routinely by engineers to model real-world behavior.

7.2 Counters, Accumulators and Flags

A counter is a variable that has its value changed by a fixed amount with each iteration of a loop. Counters are often used as loop control variables, but it is also possible for a counter to be used in a loop for some other purpose.

An accumulator is a variable used to 'accumulate' information (by having its value changed) during the execution of a loop. Accumulators are commonly numeric variables, but could also hold string data or any other kind of data.

A **flag** is a variable used to remember if a certain condition has been satisfied during the execution of the loop. Flags are usually boolean variables, and this is the same concept touched on earlier in the quarter.

7.3 Simulations

Writing a simulation loop is a straightforward process. Start by thinking about how the process works, and how each aspect corresponds to the five elements of a loop.

- What information in this simulation is constant and what information varies? What variables do we need to perform this simulation? (counters, accumulators, flags)
- What must be done to get things started? (initialization)
- What parts are repeated? (main work)
- How many times, or under what conditions, is the process repeated? (loop test)
- Is there anything that needs to be done to wrap things up at the end? (finalization)

Consider the following simple example: get 5 numbers from the user and report the average. Here is a method that accomplishes this:

```
public static void averageFive() {

   Scanner in = new Scanner( System.in );
   final int maxCount = 5;
   int count;
   double sum = 0; // initialize accumulator
   for ( count = 0; count < maxCount; count++ ) {
      System.out.print( "Enter a number: " );
      double num = in.nextDouble();
      sum += num;
      }

   double ave = sum / maxCount;
   System.out.println( "Average = " + ave );
}
```

Initializations include setting sum = 0. The repeated part includes getting a number from the user and adding it to the running sum. A finalization step -- calculating the average -- is also needed.

Let's apply this approach to a more complex simulation. Let's write a method that will project the future balance of a credit card account after a period of time. Here's the information we need to get started: initial balance, annual interest rate, monthly payment, and the number of months to simulate. Our simulation will assume that no new charges will be made (the user has already cut up the credit card).

What happens each month? Just two things: interest is added to the account balance, and the payment is deducted from the account balance. This is the main work of the simulation. How do we know when the simulation is finished? In this case, it runs for a specified number of months.

Here is a program that solves this problem:

```
public static double creditCardSim( double initialBalance,
        double annualPercentageRate, double monthlyPayment,
        int numberOfMonths ) {

   double monthlyMultiplier = annualPercentageRate / 1200;
   double balance = initialBalance; // initialize accumulator
   for( int month = 1; month <= numberOfMonths; month++ ) {
      balance = balance + balance * monthlyMultiplier;
      balance = balance - monthlyPayment;
   }
   if ( balance < 0 )
      balance = 0; // final balance cannot be less than 0
   return balance;
}
```

Notice the following:

- This simulation is all about a changing **balance**. The balance is initialized before the beginning of the loop, it is updated inside the loop, and it is checked in a finalization step before its value is returned by the method.
- This simulation uses a **definite loop** -- the number of repetitions is known in advance -- specified by a parameter of the method. So, a simple counting loop is used.
- The loop counter itself (the variable count) is used for no other purpose in this program. The numbers are not printed out or summed up or anything. Rather, the counting loop simply controls how many times the statements in its body are executed as they simulate the effect of months passing.
- Notice also that the statements inside the body of the loop reflect exactly what happens each month: interest is added to the account balance, and the payment is deducted from the account balance.
- It is possible for the balance to fall below zero as the loop progresses. This doesn't really make sense in the context of the problem, so a little adjust is performed in a finalization step. Suppose you wanted to adjust the loop so that it terminates early if the balance falls to zero (or less) -- how would you accomplish that? By changing the loop test so that it accurately reflects **the conditions required for the body of the loop to be executed**. For example:

  ```
  month <= numberOfMonths && balance > 0
  ```

7.4 Simulations with Indefinite Loops

Not every simulation makes use of a definite loop. Sometimes the conditions of the simulation determine how many times the loop executes -- and sometimes the **goal** of the simulation is to determine how many cycles are needed. In these cases, an **indefinite loop** would be used.

Consider a variation of the example above. This time, instead of specifying the number of months, we want the method to determine how many months are required to pay off the credit card balance. The process -- what happens each month -- is the same. However, the loop control is different. Here's a solution:

```
public static double monthsToPayOffBalance( double initialBalance,
                        double annualPercentageRate, double monthlyPayment ) {

   double monthlyMultiplier = annualPercentageRate / 1200;
   double balance = initialBalance;
   int month = 0;
   do {
      balance = balance + balance * monthlyMultiplier;
      balance = balance - monthlyPayment;
      month++;
   } while ( balance > 0 && month < 1200 );
   return month;
}
```

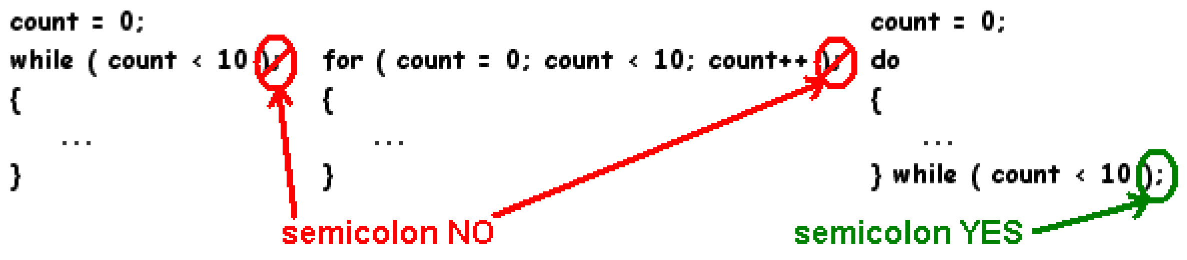
Here are some things to notice:

- Many things are the same in this version as in the previous one. However, this time, although the balance is still the critical simulation variable, it is the number of months that is returned by the method.
- Notice the **loop test**. The main condition is **balance > 0** -- we want to continue the simulation as long as the balance is greater than zero. However, if the payment is too low to cover the monthly interest, the balance will actually increase each month, so an upper limit on the number of months allowed is needed to prevent an infinite loop situation.
- To be complete, both of these programs should have tests added to verify that the values of the parameters are valid -- this is called precondition checking. The method should throw an exception if a parameter is invalid, e.g. a negative value for the monthly payment.

8 Common Problems with Loops

Several problems commonly occur when loops are used. Here is a short list, along with some tips to avoid these errors:

- **A missing update step** is a common cause of **infinite loops**. These are loops that never stop, because the loop test is always *true*. To prevent this, think about the update step when you are designing a loop. Make sure that every path through the loop includes an update step that changes the value of the control variable.
- **A loop test with an incorrect 'sense' (< rather than > or vice versa)** can also cause an infinite loop. When writing the loop test, always **ask yourself "Under what conditions should the body of the loop be executed?"** The loop test must evaluate to *true* when the body of the loop should be executed.
- A loop may contain an **off-by-one error** due to an incorrect relational operator (<= rather than < or similar). When writing the loop test, always **ask yourself "Should the body of the loop be executed when the loop control variable is <u>equal to</u> the limit?"**
- The use of a floating-point number (type *float* or *double*) as a loop control variable can cause errors in several ways. A better choice is to use one of the **<u>integer</u>** data types as a loop counter if at all possible. If a floating-point number must be used as the loop control variable, never test its value using == or !=; better to use <= or >=. Also ask yourself if the loop will behave properly if there is a slight representational error in the number (e.g. its value is 6.3999999999997 or 6.4000000000002 instead of 6.4).
- Programmers just learning Java sometimes put a semicolon at the end of a *while* or *for* statement where one should not be present. The compiler will not signal an error in this situation, but the program will almost certainly not do what you want it to do. Remember the following about the **three types of loops**:



```
count = 0;
while ( count < 10 );
{
    ...
}
```

```
for ( count = 0; count < 10; count++ );
{
    ...
}
```

```
count = 0;
do
{
    ...
} while ( count < 10 );
```