OOPJAV – Object-Oriented Programming in Java

Seminar 3 – Control Statements

At first this lecture seems like a lot of work; however the material covered is fairly straightforward and therefore should not be too time consuming nor should it cause any major problems. By now we now have the basics required to write Java programs. Yet so far our programs have been rather flat, meaning that they all generally have a single *flow of control* path passing through them. In this seminar, we are first going to look at how we can introduce choice points into this flow of control path so that we can have several paths going through a program. We will also then look at repetition.

Before we can look at selection and repetition in detail, we must first understand the concepts of expressions and statements.

I. Expressions

An *expression* in the context of computer programming languages is a formula. As such, expressions are composed of one or more operands whose values are combined by operators. Typical operators include the mathematical operators (+, -, *, /), that we have already seen, for example:

numA + (2 * numB)

Evaluation of an expression produces a value. Expressions can therefore be used to assign values to variables by embedding them into an assignment statement.

A. Boolean Expressions

Expressions that use Boolean operators return a data item of type Boolean which can handles the values true or false. Well-known Boolean operators are given in **Figure 1** below. The first six are sometimes called *comparison* or *relational* operators and the remaining four are known as *logica*l operators.

Operation	Java encoding
Equals	==
Not equals	!=
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Logical and	& or &&
Logical or	| or ||
Exclusive logical or	^
Logical not	!

Figure 1: *Boolean operations*

Examples:

operand1 < 50

operand2 == 0

operand3 >= 10 & operand3 <= 20

operand3 < 10 | operand3 > 20

operand1 < 20.0 && (operand2 != 'q' | myInstance.testFun(operand1,operand4) == 1)

These expressions can be interpreted as follows:

1. Return true if value for data item operand1 is less than 50, otherwise return false.

2. Return true if value for data item operand2 is equal to 0, otherwise return false.

3. Return true if value for data item operand3 is greater than or equal to 10 and less than or equal to 20 (i.e. operand3 is within the range 10-20), otherwise return false.

4. Return true if value for data item operand3 is less than 10 or greater than 20 (i.e. operand3 is not within the range 10-20), otherwise return false.

5. Return true if value for data item operand1 is less than 20.0 and either the value for operand2 is not the character 'q', or the value returned by the test function (which has two arguments, operand1 and operand4) is equal to 1, otherwise return false.

The comparison operators may be applied to both numeric types and to characters. The result of a comparison on characters will depend on the *Unicode* values associated with the characters in question. For example, 'B' < 'a' is true ('B' has a Unicode value of 66 while 'a' has a Unicode value of 97).

B. Evaluation

Consider the expression:

operand1 & operand2

If operand1 is false, then the expression will return false regardless of the value of operand2. Similarly, in the case of:

operand1 | operand2

If operand1 is true, then the expression will return true regardless of the value of operand2. In both cases, the second operand has been evaluated needlessly. Java therefore provides two short cut versions of & and |, namely && and ||, which avoid this needless evaluation. Thus:

operand1 && operand2

operand1 || operand2

The mechanism whereby we only evaluate operands if we need to is called *lazy evaluation* or *short circuit evaluation.*

II. Statements

Statements are the commands in a language that perform actions and change the state of an object. The state of an object is described by the values held by its variables at a specific point

during the program execution. We can identify six broad categories of statements common to most OOP languages:

1. Assignment - changes the value (*state*) of a field.

2. Method calls - causes a method to be performed. In object-oriented parlance (although not in Java) this is sometimes referred to as *message passing*.

3. I/O (Input/Output) - statements to control input and output (from various sources and to various destinations).

4. Selection - choices between alternative statements or groups of statements to be performed.

5. Repetition - perform a certain statement or group of statements in an iterative manner until some end condition is met.

6. Exception - detect and react to some unusual circumstance.

Statements are usually separated by an operator or symbol. In Java, this is the semi-colon character (;).

III. Selection

We are now going to look at how we can introduce choice points into this flow of control path so that we can have several paths going through a program. This is done using a selection statement. The path followed on any particular invocation will then depend on the input variables supplied. We can identify two broad types of selection constructs:

1. If-else statements

2. Switch statements

A. If-else Statements

An if-else statement, sometimes referred to as a conditional, can be used in two forms. The first is often referred to as a *linear if:*

1.
```
if (<Condition X>)
        { <Action Y> }
```

2.
```
if (<Condition X>)
        { <Action Y> }
else
        { <Action Z> }
```

The condition is a Boolean expression (or a sequence of Boolean expressions) as described above, and the action part consists of one or more statements. Note that where the action part consists of only a single statement, we can omit the curly brackets. We can also identify a number of variations on the above. For example, we can write:

```
if (<Condition X1>)
        {<Action Y1>};
else if (<Condition X2>)
        {<Action Y2>};
```

```
else if (<Condition Xn>)
        {< Action Yn>};
else
        {<Action Z>};
```

It is sometimes useful to produce a *control flow diagram* indicating flow of control through a method. Such a diagram is presented in **Figure 2**. The choice point (selection) is indicated by a diamond, and other sequences of statements are indicated by simple boxes. Flow of control is indicated by the directed lines. The flow starts at the start oval (referred to as the *riser*) and ends at the end oval (referred to as the *sink*). The principal advantage of such a diagram is that it clearly indicates the potential *paths* through a software system.

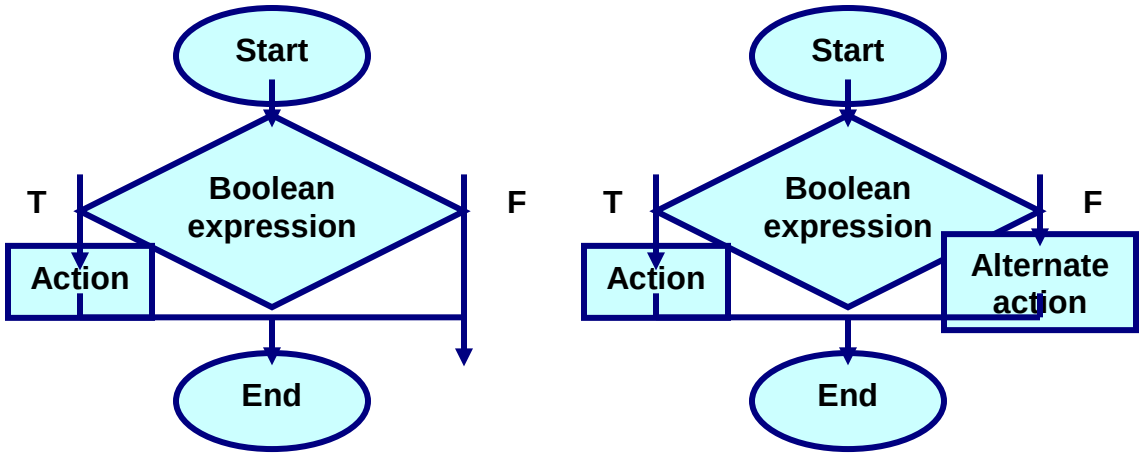

Figure 2: *Control flow diagrams for if-else statements (both versions)*

B. Example Program - Linear Equation

A linear equation with one unknown can be expressed as follows:

bx + c = 0;

where b and c represent numeric quantities. A Java program which will solve equations of this form and find a value for x - the root of the equation, is presented in **Figure 3**. The code assumes that b and c are integers and that the output is to be presented in the form of a real number. Note that to find a value for x we must rearrange the above equation:

x = -c / b;

Consequently, b must not have a value of 0, otherwise a divide by zero error will occur. We therefore include an if-else statement to check for this condition.

```
1  //LINEAR EQUATION CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  public class LinearEquation{
7
8      //-------ATTRIBUTES-------
9      private double bValue, cValue, xValue;
10
11     //-------CONSTRUCTOR-------
12     public LinearEquation(int bInput, int cInput){
13         bValue = (double)bInput;
14         cValue = (double)cInput;
15     }
16
17     //-------METHODS-------
18     //Resolve linear equation, return true if successful and false otherwise
19     public boolean resolveLinearEquation(){
20         //Test for divide by 0 error; if found, return false
21         if (bValue != 0.0)
22         {
23             //Determine xValue and return true
24             xValue = (-cValue)/bValue;
25             return (true);
26         }
27         else return(false);
28     }
29
30     //Get xValue
31     public double getXValue(){
32         return (xValue);
33     }
34 }
```

Figure 3: *Source code for linear equation class*

Note that on the code presented in **Figure 3**:

1. In the constructor LinearEquation, there is no need to expressly include the casts

   bValue = (double) bInput;
   cValue = (double) cInput;

   because an integer can be upgraded to a double automatically. However, from a software engineering point of view (readability, understandability, etc.), it makes good sense to include an explicit cast.
2. The brackets included in the arithmetic statement containing the monadic (unary) - operator, xValue = (-cValue)/bValue (contained in the resolveLinearEquation method) are superfluous; however they have been included to again enhance readability/understandability.

An accompanying application class is given in **Figure 4**. Notice the way that the selection statement is encoded. The condition part of the statement is evaluated first and then one or other of the action parts is invoked according to the result. It is considered bad practice (and also pointless) to test the result of a Boolean method against itself, meaning we should not write the condition part given in **Figure 4** as:

(newEquation.resolveLinearEquation() == True)

```java
//LINEAR EQUATION APPLICATION CLASS
//Y. Jing
//March 2007
//The University of Liverpool, UK

import javax.swing.*;

public class LinearEquationApp{

    //-------Main Method-------

    public static void main(String[ ] args)
    {
        int newBValue, newCValue;
        //Input
        String inputB = JOptionPane.showInputDialog("Please input an int value for B:");
        newBValue = new Integer(inputB).intValue();
        String inputC = JOptionPane.showInputDialog("Please input an int value for C:");
        newCValue = new Integer(inputC).intValue();

        //Create linear equation instance
        LinearEquation newEquation = new LinearEquation(newBValue, newCValue);

        //Calculation if method returns true, output result; otherwise produce error message
        if(newEquation.resolveLinearEquation())
            JOptionPane.showMessageDialog(null, "Result is " + newEquation.getXValue());
        else
            JOptionPane.showMessageDialog(null, "Error 1:  Divide by zero error!");
    }
}
```

Figure 4: *Source code for linear application class*

A control flow diagram for the entire system is shown in **Figure 5**.

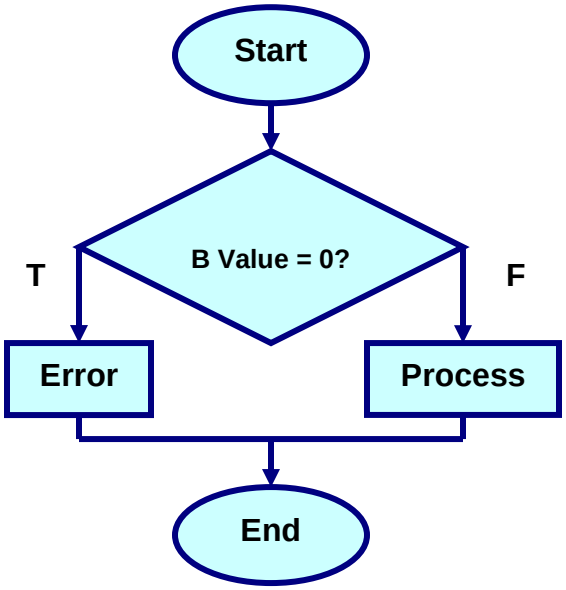

Figure 5: *Control flow diagram for linear equation problem*

C. Switch Statements

An if-else statement supports selection from only two alternative choices of action. We can, of course, nest such statements or use constructs such as if ... else if ... else, but it is usually more succinct to use a *switch* statement when there are many alternate choices of action that may be made (also sometimes referred to as a *case* statement). Switch statements allow selection from many alternatives where each alternative is linked to a predicate (referred to as a *selector)*,

which, when evaluated to true causes an associated program statement (or statements) to be executed. In Java, the general format of a switch (case) statement is as follows:

```
switch <SELECTOR> {
        case <ALTERNATIVES>:
                <STATEMENTS>;
        ...
}
```

Note that the colon is important. Selections may be made according to:

1. A distinct value of a given selector, or

2. A default value.

Selectors **must** be of a discrete type (such as an integer or a character). An example Java class containing a switch statement is given in **Figure 6**.

```
1  //SWITCH EXAMPLE APPLICATION
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class SwitchExampleApp{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int number;
14
15         //Input
16         String input = JOptionPane.showInputDialog("Please input an integer:");
17         number = new Integer(input).intValue();
18
19         //Process number using a case statement
20         switch (number){
21             case 0:
22                 JOptionPane.showMessageDialog(null, "Number is 0");
23             case 1:
24                 JOptionPane.showMessageDialog(null, "Number is 1");
25             case 2:
26             case 3:
27             case 4:
28                 JOptionPane.showMessageDialog(null, "Number is 2, 3, or 4");
29                 break;
30             default:
31                 JOptionPane.showMessageDialog(null, "Number is less than 0 or more than 4");
32         }
33     }
34 }
```

Figure 6: *Switch example program*

Where N is of type integer, the code in **Figure 6** states that:

1. When the value of N is equal to 0, the output is 'Number is 0.'

2. When the value of N is equal to 1, the output is 'Number is 1.'

3. When the value of N is within the range 2 to 4 inclusive, the output is 'Number is 2, 3 or 4.'

4. Otherwise by default, the output is 'Number is less than 0 or greater than 4.'

Notice the use of the break statement in the code. It is considered good practice to always include a default alternative to act as a catch-all in a switch statement.

D. Menu Interfaces

A common interface requirement is to allow a user to select from a number of alternatives. A simple way of doing this is to present the user with a list of possibilities from which to choose (this type of interface is less error-prone). In sophisticated window interfaces, the selection is made through mouse clicks. An alternative and simpler interface is to allow the user to indicate a selection by entering a number or letter at the keyboard. The disadvantage associated with this simpler approach is that the user may make erroneous inputs and therefore the interface must include some error recovery mechanism.

Consider the Java application program given in **Figure 8**. Notice that it consists of a top level main() method and two other methods, outputMenu() and processSelector(). The outputMenu() method outputs a list of options from which the user can make a single selection by entering the appropriate number associated with the desired option. The selection is then processed by the processSelector() method, which includes an error recovery mechanism that will cause the program to repeat until a valid option is entered. A control flow diagram for the code is presented in **Figure 7**.

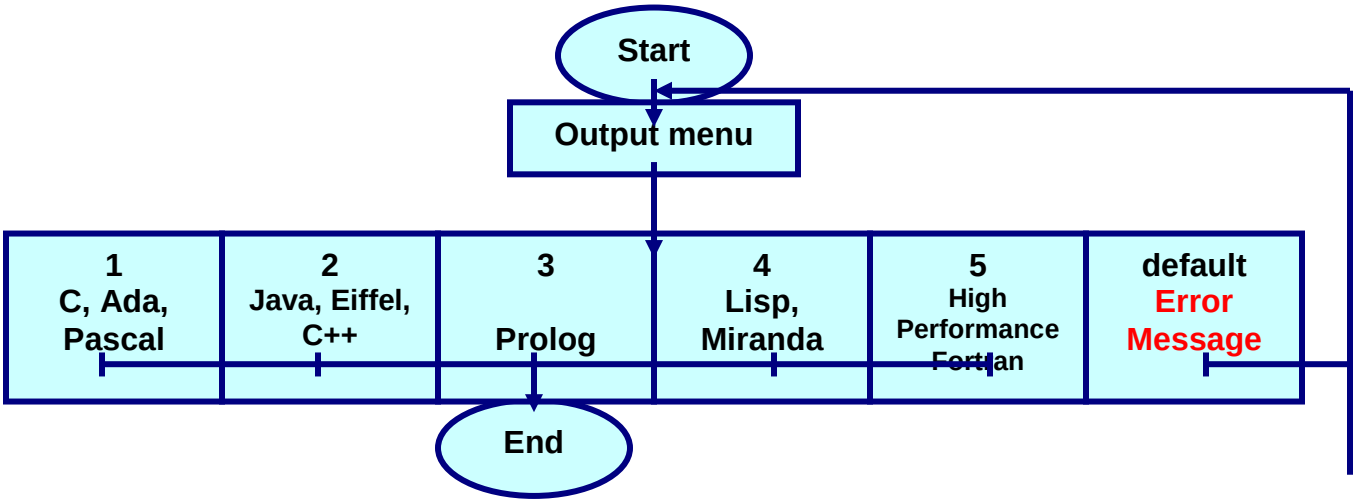

Figure 7: *Control flow diagram for code presented in* ***Figure 8***

Incidentally, the process whereby a method calls itself, as in the code presented in **Figure 8**, is known as recursion.

```
1  //MENU INTERFACE EXAMPLE APPLICATION
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5  import javax.swing.*;
6  public class MenuInterfaceExApp{
7      //-------Main Method-------
8      public static void main(String[ ] args) {
9          int selector;
10         selector = outputMenu();
11         processSelector(selector);
12     }
13     public static int outputMenu(){
14         String input = JOptionPane.showInputDialog("Please select a number representing a programming paradigm:
15         1 - Imperative, 2 - Object-oriented, 3 - Logical, 4 - Functional, 5 - Parallel.");
16         return(new Integer(input).intValue());
17     }
18     public static void processSelector(int selector) {
19         switch(selector){
20             case 1:
21                 JOptionPane.showMessageDialog(null, "Example languages include C, Ada and Pascal.");
22                 break;
23             case 2:
24                 JOptionPane.showMessageDialog(null, "Example languages include Java, Eiffel and C++.");
25                 break;
26             case 3:
27                 JOptionPane.showMessageDialog(null, "Example languages include Prolog.");
28                 break;
29             case 4:
30                 JOptionPane.showMessageDialog(null, "Example languages include Lisp and Miranda.");
31                 break;
32             case 5:
33                 JOptionPane.showMessageDialog(null, "Example languages include high performance Fortran.");
34                 break;
35             default:
36                 JOptionPane.showMessageDialog(null, "Unrecognized menu selection " + selector + "!");
37                 selector = outputMenu();
38                 processSelector(selector);
39         }
40     }
41 }
```

Figure 8: *Example application illustrating menu interface*

IV. Repetition

A repetition construct causes a group of one or more program statements to be invoked repeatedly until some *end condition* is met. We can identify two main forms of repetition:

1. Fixed count loops - repeat a predefined number of times (i.e. defined prior to compilation/interpretation).

2. Variable count loops (also sometimes referred to as conditional loops) - repeat an unspecified number of times.

A. Pre-test and Post-test Loops

Fixed and variable count loops can be further classified according to whether they are *pre-test* or *post-test* loops. In the case of a pre-test loop, the end condition is tested prior to each repetition, while in the case of a post-test loop, the end condition is tested after each repetition. With respect to Java, both pre-test fixed and variable count loops are supported, as well as post-test fixed and variable count loops.

We can illustrate the distinction between pre-test and post-test loops by considering the flow of control associated with each. In the case of a pre-test loop, we test before entering the repeat statement. Therefore, if on the first attempt, the test fails, then no repetitions will be made. In the case of a post-test loop, the repeat statement is called at least once. We can conceive of flow of control in terms of *control flow diagrams* as shown in **Figure 9**.

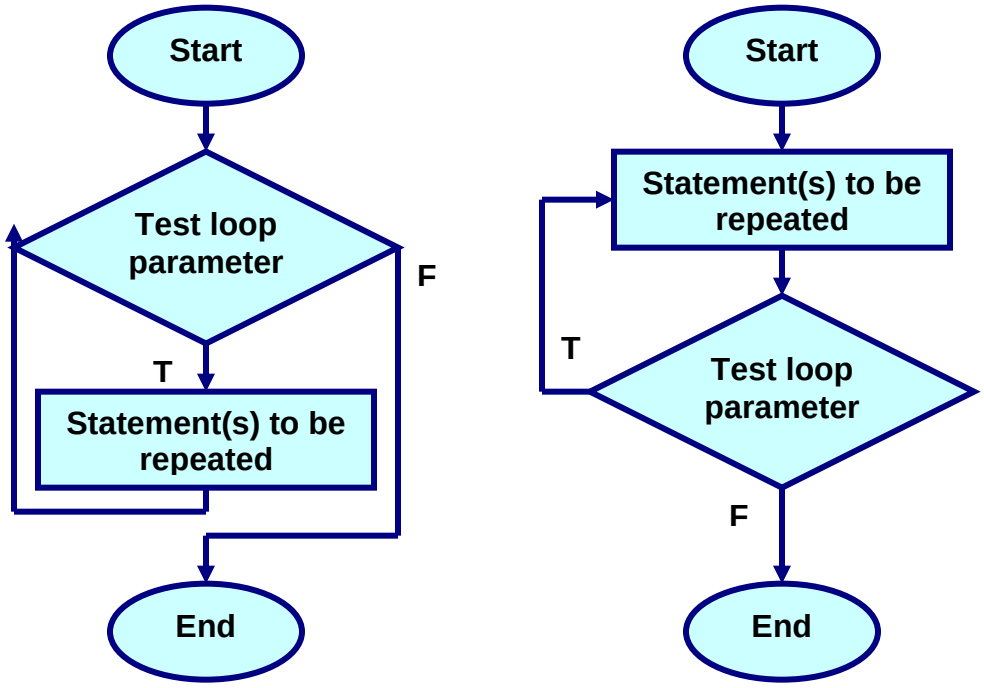

Figure 9: *Distinction between pre-test and post-test loops*

B. Java Loops

Thus, from the above, we can identify four basic types of loop:

1. Fixed count pre-test loops.
2. Fixed count post-test loops.
3. Variable count pre-test loops.
4. Variable count post-test loops.

In Java, three types of loop construct are supported:

1. **For** loops --- To define pre-test fixed or variable count loops.
2. **While** loops --- To also define pre-test fixed and variable count loops.
3. **Do-while** loops --- To define post-test fixed and variable count loops.

We therefore have two mechanisms for constructing variable and fixed count pre-test loops. Whichever you choose to use and when is entirely up to you.

V. For Loops in Java

The general form of a for loop in Java is:

for (<startExpression>;<testExpression>;<updateExpression>) {
<sequenceOfStatements>
}

This is a very concise method of defining a loop that is sometimes criticised because of its lack of easy interpretation. The counter-argument to this complaint is that this form of loop definition is very versatile. Whatever the case, from the above:

1. **<startExpression>:** Introduces one or more *loop parameters* (also referred to as the *control variables* or *loop counters*). These are introduced using one or more assignment expressions which serve to assign the start value(s) to the loop parameter(s). Where more than one loop parameter is introduced, each must be separated by a comma.

2. **<testExpression>:** Defines the end condition for terminating the loop. Usually this is a simple comparison expression. Again, there can be more than one test expression in which case they must again be separated by commas.

3. **<updateExpression>:** To avoid infinite repetition, the loop parameter(s) must be updated on each repetition. This is usually done using an arithmetic expression. The most straightforward approach is to simply increment the loop parameter by one on each repetition. As before, where there is more than one update expression, they must be separated by commas.

Note: loop parameters are usually of some scalar numeric type. It is good practice to use parameters of type int or long. In **Figure 10,** a trivial Java program is presented incorporating a for loop that outputs a sequence of 10 smiley faces.

```
1  //SMILEY FACE APPLICATION
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class SmileyFace{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int loopParameter;
14         final int startCondition = 0;
15         final int endCondition = 10;
16
17         //For loop
18         for (loopParameter = startCondition; loopParameter < endCondition; loopParameter++){
19             JOptionPane.showMessageDialog(null, ":-D");
20         }
21     }
22 }
```

Figure 10: *Smiley face for loop application example*

Notice that the start and end values for the loop are defined as constants and not as *magic numbers*. This is for sound software engineering reasons because the use of a name conveys much more meaning to a reader than a number.

A. Using the Loop Parameter

In the example given in **Figure 10,** the loop parameter serves no other purpose than to control the loop. In addition, we can make use of the parameter as part of an output statement or as part of some simple arithmetic. **Figure 11** gives an example of the first usage, and **Figure 12** an example of the second. The code in **Figure 11** is used to simply output a sequence of numbers (the loop parameter).

```
1  //SEQUENCE NUMBERS CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class SequenceNumbers{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int loopParam;
14         final int startCondition = 0;
15         final int endCondition = 10;
16
17         //For loop
18         for (loopParam = startCondition; loopParam < endCondition; loopParam++){
19             JOptionPane.showMessageDialog(null, loopParam + "");
20         }
21     }
22 }
```

Figure 11: *Sequence numbers for loop application example*

```
1  //SEQUENCE  EVEN NUMBERS CLASS
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class SequenceEvenNumbers{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int loopParam;
14         final int startCondition = 0;
15         final int endCondition = 10;
16
17         //For loop
18         for (loopParam = startCondition; loopParam < endCondition; loopParam++){
19             JOptionPane.showMessageDialog(null, loopParam * 2 + "");
20         }
21     }
22 }
```

Figure 12: *Sequence of even numbers for loop application example*

The code in **Figure 12** performs some arithmetic on the loop parameter so that a sequence of even numbers is produced.

B. Processing a Loop in Reverse

It is sometimes desirable to process a fixed count loop backwards, or in *reverse*. In Java, all we need to do is decrement the loop parameter instead of the more usual increment (i.e. – instead of ++).

VI. While Loops in Java

While loops are traditionally used to implement variable count loops (although this can also be achieved with a for loop). As we will see later, while loops can also be used to implement fixed count loops. A variable count loop is used when the number of iterations associated with a loop construct is not known in advance. What is known, however, is that the action should continue, provided a certain condition is true. When the condition becomes false, the repetition stops. The Java while loop statement has the following format:

```
while (<Boolean expression>) {
    <sequence of statements>
}
```

The Boolean expression is used to test some control variable. When the expression evaluates to false, the loop terminates. So that the loop has a chance of terminating, the control variable must be updated each iteration.

The example fragment of code in **Figure 13** uses a variable count while loop to monitor input. We keep looping around until the user inputs a number between 1 and 50. Note the use of the break statement here to exit from the loop when a number satisfying the conditions has been entered. We discuss the break statement further in section VIII.

```
1  //VARIABLE COUNT WHILE LOOP EXAMPLE
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class WhileLoopExample{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int number = 2;
14         final int maximum = 50;
15         final int minimum = 1;
16
17         //While loop
18         while (number > minimum && number < maximum){
19             String input = JOptionPane.showInputDialog("Please enter an integer between "
20             + minimum + " and " + maximum + " (inclusive):");
21             number = new Integer(input).intValue();
22             JOptionPane.showMessageDialog(null, "Input =" + number);
23         }
24     }
25 }
```

Figure 13: *Input loop using while statement*

VII. While Loop or For Loop?

The code presented in **Figure 13** could equally well have been implemented using a Java for loop. Similarly the smiley face program described in **Figure 10** can also be implemented using a while loop. Try to do this for practice.

So when should we use a for loop and when a while loop? Both are pretest loops, and both can be used for fixed and variable count loops. In Java, where for and while statements can be used to describe both fixed and variable count loops, it is simply a matter of personal choice!

VIII. Continuous Loops and the Break Statement

Sometimes there is a need to terminate a loop somewhere in the middle. Many programming languages therefore support a *break* statement, including Java. This has already been introduced in connection with switch statements, but it can also be used to break out of a loop. Wherever possible, loop break statements should be avoided, as they alter the flow of control associated with loop statements from a clear 'in at the top, out at the bottom' structure to something which is less obvious. In Java, a break statement can sometimes legitimately be used to break out of a continuous loop. However, in most cases, there is an alternative strategy that will avoid the use of a break.

In Java we may write continuous loops using a for statement:

```
for ( ; ;) {
   <STATEMENTS TO BE REPEATED INDEFINITELY>
}
```

or a while statement:

```
while (true) {
   < STATEMENTS TO BE REPEATED INDEFINITELY >
}
```

Note that the condition is the Boolean value, true, which always succeeds or evaluates to true. We can also use a do-while post-test loop construct to define a continuous loop - but more on this later. Continuous loops are sometimes used when awaiting a specific input that we test for in the iteration. If found, a break statement may then be used to break out of the loop:

```
if ( <SOME CONDITION> ) break;
```

XI. Missing Control Statements in For Loops

Although a for loop usually consists of (1) a control variable instantiation expression, (2) a termination test expression and (3) a control variable increment (or decrement) expression, it is not necessary to include all three. Some examples are presented in the code given in **Figure 14**, which consists of four for loops.

1. The first loop uses the standard format with all expressions present and produces the result of adding up the sequence {1, 2, 3, 4, 5, 6, 7, 8, 9}.

2. The second loop has no instantiation expression since an existing data item, total, which is instantiated on the previous loop, is used. The loop is used to count down from 45 to 5 in steps of five.

3. The third loop has no end expression as this is included within the loop statements (coupled with a break). It produces a sequence where each element is produced by adding the loopCounter to the previous element in the sequence, starting with the value of the data item total as set by the previous loop. The break statement is used to jump out of the loop when the sequence gets above 50.

4. The last loop has no increment expression. It is included within the body of the loop statement.

As noted above, if we omit all the expressions, a continuous loop effect is produced. For example:

```
for ( ; ; ) {
   System.out.println("This loop will go on for ever");
}
```

Note that we still have to include the semicolon separators.

The resulting output from the code presented in **Figure 14** is as follows:

Loop 1: 45
Loop 2: 45, 40, 35, 30, 25, 20, 15
Loop 3: 11, 13, 16, 20, 25, 31, 38, 46, 55
Loop 4: 64

```
1  //SWITCH EXAMPLE APPLICATION
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class SwitchExampleApp{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int number;
14
15         //Input
16         String input = JOptionPane.showInputDialog("Please input an integer:");
17         number = new Integer(input).intValue();
18
19         //Process number using a case statement
20         switch (number){
21             case 0:
22                 JOptionPane.showMessageDialog(null, "Number is 0");
23             case 1:
24                 JOptionPane.showMessageDialog(null, "Number is 1");
25             case 2:
26             case 3:
27             case 4:
28                 JOptionPane.showMessageDialog(null, "Number is 2, 3, or 4");
29                 break;
30             default:
31                 JOptionPane.showMessageDialog(null, "Number is less than 0 or more than 4");
32         }
33     }
34 }
```

Figure 14: *For loops with missing expressions*

X. Post-test Loops and the Java Do-While Loop

Remember that whereas for a pre-test loop, the control variable is tested prior to the start of each iteration, in post-test loops, the control variable is tested at the end of each iteration. From an efficiency point of view, the advantage is that we can reduce the number of tests required provided that we understand the loop will be exercised at least once. If there is a possibility that the loop should not be exercised at all, then a post-test loop will not be appropriate (a pre-test loop will be required instead). In Java, a post-test loop is implemented using a do-while construct. This has
the general form:

```
do {
        <STATEMENTS TO BE REPEATED>
} while <TEST CONDITION>;
```

An example of the use of this construct is presented in **Figure 15** where we present a third implementation of the integer input example program introduced earlier. As before, during the iteration the user is asked to input an integer between 1 and 50 inclusive. If correct input is made the loop terminates. If we compare the code presented in **Figure 15** with that given in **Figure 13** we can note that:

1. In the case of the code presented in **Figure 13**, the program will test the number variable at least twice (as opposed to at least once using the do-while construct).

2. Also, again with respect to the code presented in **Figure 13**, we must initialize the number variable with an appropriate value to ensure that the user is allowed the opportunity to input a number. This is not the case when using a do-while loop construct.

```
1  //DO WHILE LOOP EXAMPLE
2  //Y. Jing
3  //March 2007
4  //The University of Liverpool, UK
5
6  import javax.swing.*;
7
8  public class DoWhileExample{
9
10     //-------Main Method-------
11
12     public static void main(String[ ] args){
13         int number = 0;
14         final int maximum = 50;
15         final int minimum = 1;
16
17         //Do-while loop
18         do{
19             String input = JOptionPane.showInputDialog("Please enter an integer
20             number = new Integer(input).intValue();
21             JOptionPane.showMessageDialog(null, "Input =" + number);
22         }while (number > minimum && number < maximum);
23     }
24 }
```

Figure 15: *Variable count do while loop example*

Do-while loops can equally well be used to describe fixed count loops. Try reworking the smiley faces program presented earlier using a fixed count for loop and then using a fixed count do-while loop. Note that the test expression, unlike the previous examples, should include the <= operator so that the same number of smiley faces are printed as before. The distinction is that since this is a post test loop, the loop parameter is incremented before we get to the test expression. Therefore if you do not alter the test expression, only 9 smiley faces will be produced instead of 10!

As with for and while loops, we can define a continuous loop using a do-while construct:

```
do {
        < STATEMENTS TO BE REPEATED INDEFINITELY>
} while(true);
```

XI. Nested Loops

As with selection statements, loops can be nested. Care should be taken when nesting loops as this tends to decreases the readability of the code. Unless we are undertaking some fairly trivial nesting, it is normally better to encode the nested loops in the form of further methods called from within the parent loop.

Required Reading

In **Chapter 4** of the text, read the following sections:

4.1-4.7, 4.10-4.13.

In **Chapter 5** of the text, read the following sections:

5.1-5.9.

You will notice in each chapter various sections entitled Common Programming Errors, Testing and Debugging, Performance Tips and Software Engineering Observation, etc. These sections are meant to be skimmed over or even ignored on a first reading, but bear them in mind as you may wish to return to them later.

Helpful Additional Readings

If you happen to get really stuck on a concept we've introduced in this week's material, please post a message as a response to the Q & A thread in the main folder to let me know. I have extra materials and instruction guides that I can post that may help clarify things for you. **All you have to do is ask!**

Helpful Websites

Some sites you may wish to visit this week for clarification of the ideas presented in the lecture and in the book are:

For control flow and decision structures:

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/flow.html

You may notice that the above website comes directly from the Sun Microsystems Java web. I encourage you to use this site frequently as there are many tutorials, short courses, white papers, discussion groups, etc., that you may use to your advantage. You may wish to start getting used to the Java API files as soon as possible - many of your questions about syntax can be answered here.