STUDENT OUTLINE

Lesson 10 – for, do-while, Nested Loops

INTRODUCTION: We continue our study of looping control structures with a look at the **`for`** and **`do-while`** loops. The concept of nested loops to solve two-dimensional problems will also be covered.

The key topics for this lesson are:

A. The **`for`** Loop
B. Nested Loops
C. The **`do-while`** Loop
D. Choosing a Loop Control Structure

VOCABULARY: FOR DO-WHILE
NESTED LOOP

DISCUSSION: A. The **`for`** Loop

1. The **`for`** loop has the same effect as a while loop, but using a different format. The general form of a **`for`** loop is:

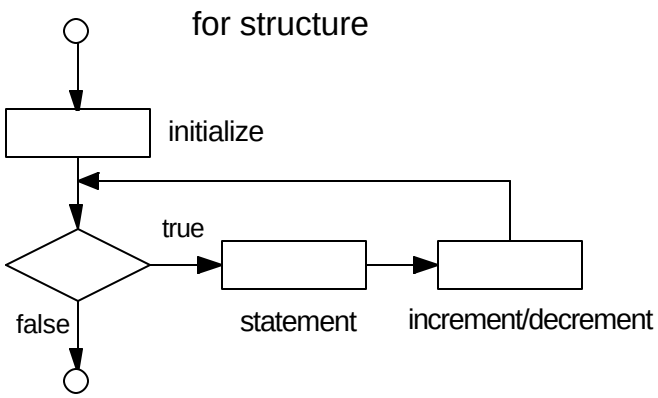
```
for (statement1; expression2; statement3)
  statement
```

`statement1` initializes a value
`expression2` is a **`boolean`** expression
`statement3` alters the key value, usually via an increment/decrement statement

2. Here is an example of a **`for`** loop, used to print the integers 1-10.

```
for (int loop = 1;  loop <= 10;  loop++)
  System.out.print( loop);
```

3. The flow of control is illustrated:



Notice that after the statement is executed, control passes to the increment/decrement statement, and then back to the Boolean condition.

4. Following the general form of section 1 above, the equivalent **`while`** loop would look like this:

```
statement1;            //  initializes variable
while (expression2)    //  Boolean expression
{
  statement;
  statement3;          //  alters key value
}
```

Coded version:

```
loop = 1;
while (loop <= 10)
{
  System.out.print( loop);
  loop++;
}
```

5. A **`for`** loop is appropriate when the initialization value and number of iterations is known in advance. The above example of printing 10 numbers is best solved with a **`for`** loop because the number of iterations of the loop is well-defined.

6. Constructing a **`for`** loop is easier than a **`while`** loop because the key structural parts of a loop are contained in one line. The initialization, loop boundary, and increment/decrement statement are written in one line. It is also easier to visually check the correctness of a **`for`** loop because it is so compact.

7. A **`while`** loop is more appropriate when the boundary condition is tied to some input or changing value inside of the loop.

8. Here is an interesting application of a **for** loop to print the alphabet:

```
char letter;

for (letter = 'A'; letter <= 'Z'; letter++)
  System.out.print( letter);
```

The increment statement `letter++` will add one to the ASCII value of `letter`.

9. A simple error, but time-consuming to find and fix is the accidental use of a null statement.

```
for (loop = 1;  loop <= 10;  loop++);   // note ?;?
  System.out.print(loop);
```

The semicolon placed at the end of the first line causes the **for** loop to do "nothing" 10 times. The output statement will only happen once after the **for** loop has done the null statement 10 times. The null statement can be used as a valid statement in control structures.

B. Nested Loops

1. To nest a loop means to place one loop inside another loop. The statement of the outer loop will consist of another inner loop.

2. The following example will print a rectangular grid of stars with 4 rows and 8 columns.

```
for (int row = 1;  row <= 4;  row++)
{
  for (col=1;  col <= 8;  col++)
    System.out.print("*");
  System.out.println( );
}
```

Output:

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

3. For each occurrence of the outer `row` loop, the inner `col` loop will print its 8 stars, terminated by the newline character.

4. The action of nested loops can be analyzed using a chart:

row	col
1	1 to 8

2	1 to 8
3	1 to 8
4	1 to 8

5. Suppose we wanted to write a method that prints out the following 7-line pattern of stars:

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

6. Here is an analysis of the problem, line-by-line.

Line #	# spaces	# stars
1	0	7
2	1	6
3	2	5
...		
7	6	1
L	L - 1	N - L + 1

For a picture of `N` lines, each line `L` will have (`L-1`) spaces and (`N-L+1`) stars.

7. Here is a pseudocode version of the method.

A method to print a pattern of stars:

Print `N` lines of stars, each Line `L` consists of
(`L-1`) spaces
(`N-L+1`) stars
a line feed

8. Code version of the method.

```
void  picture (int n)
{
  int  line, spaces, stars, loop;

  for (line = 1; line <= n; line++)
  {
    spaces = line - 1;
    for (loop = 1; loop <= spaces; loop++)
      System.out.print (" ");           // print a blank space
    stars = n - line + 1;
    for (loop = 1; loop <= stars; loop++)
      System.out.println ("*");
    System.out.println ( );
  }
}
```

C. The **do-while** Loop

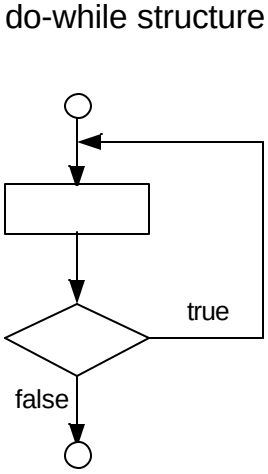
1. There are conditional looping situations where it is desirable to have the loop execute at least once, and then evaluate an exit expression at the end of the loop.

2. The **do-while** loop allows you to do a statement first, and then evaluate an exit condition. The **do-while** loop complements the **while** loop that evaluates the exit expression at the top of the loop.

3. The general form of a **do-while** loop is:

```
do
  statement;
while (expression);
```

4. The flow of control for a **do-while** loop is illustrated:



5. The following fragment of code will keep a running total of integers, terminated by a sentinel - 1 value.

```
int number, total = 0;
do
{
  System.out.print ("Enter an integer (-1 to quit) --> ");
  number = console.readInt();
  if (number >= 0)
    total += number;
}
while (number >= 0);
```

In contrast to the **while** loop version, the **do-while** has the advantage of using only one input statement inside of the loop. Because the Boolean condition is at the bottom, you must pass through the main body of a **do-while** loop at least once.

6. The same strategies used to develop **`while`** loops apply to **`do-while`** loops. Make sure you think about the following four sections of the loop: initialization, loop boundary, contents of loop, and the state of variables after the loop.

D. Choosing a Loop Control Structure

See Handout H.A.10.1, ***Programming pointers.***

1. If you know how many times a loop is to occur, use a **`for`** loop. Problems that require execution of a pre-determined number of loops should be solved with a **`for`** statement.

2. The key difference between a **`while`** and **`do-while`** is the location of the boundary condition. In a **`while`** loop, the boundary condition is located at the top of the loop. Potentially a **`while`** loop could happen zero times. If it is possible for the algorithm to occur zero times, use a **`while`** loop.

3. Because a **`do-while`** loop has its boundary condition at the bottom of the loop, the loop body must occur at least once. If the nature of the problem being solved requires at least one pass through the loop, use a **`do-while`** loop.

SUMMARY/ REVIEW: Learning to translate thoughts into computer algorithms is one of the more challenging aspects of programming. We solve repetitive and selection problems constantly without thinking about the sequence of events. Use pseudocode to help translate your thinking into code.

ASSIGNMENT: Lab Exercise, L.A.10.1, *Pictures*
Lab Exercise, L.A.10.2, *Payments*
Lab Exercise, L.A.10.3, *ParallelLines*

LAB EXERCISE

Pictures

Assignment:

1. Write a method that takes in, through its parameters, the number of rows and columns to print in a multiplication table.

2. There must be a margin of row and column headings for the table.

3. The answer columns should be separated by a field-width of 5 columns.

4. A blank line should be inserted between the column heading and the first row of the table.

5. A precondition of the procedure is that the value of the row or column will be from 1..12.

6. Sample run output for `printTable(4,6)`:

```
      1    2    3    4    5    6

1     1    2    3    4    5    6
2     2    4    6    8   10   12
3     3    6    9   12   15   18
4     4    8   12   16   20   24
```

---

7. Write another method to print the following pyramid pattern of stars. If the number of lines = 5, then:

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

8. Such a method should take in, through its parameter list, the number of lines to print. The precondition of this method, lines <= 30.

Instructions:

1. The `main` method will consist of only 4 method calls separated by `readLine()` calls.

```
class PicturesTest
{
  public static void main (String[] args)
  {
    Pictures pic = new Pictures();
    ConsoleIO keyboard = new ConsoleIO();
    String get;

    pic.printTable(4,6);
    get = keyboard.readLine(); // freezes the output screen to see the picture
    pic.printTable(11,12);
    get = keyboard.readLine();
    pic.pyramid(10);
    get = keyboard.readLine();
    pic.pyramid(25);
    get = keyboard.readLine();
  }
}
```

2. The `readLine()` between the four method calls will hold the picture until a return key is entered.

3. Your source code must include a pseudocode version of each method. Write your pseudocode above your actual code as documentation. Please follow the process and develop your pseudocode first, then translate it to code. You need to practice this system of problem solving on relatively easy problems so that you can apply it to more difficult problems.

LAB EXERCISE

Payments

Background:

Borrowing money for expensive items has become a way of life for most Americans. To illustrate the high cost of borrowing and how such loans work, you will write a program to calculate the following monthly analysis of a loan.

Month	Principal	Interest	Payment	New Balance
1	10000.00	100.00	300.00	9800.00
2	9800.00	98.00	300.00	9598.00
3	9598.00	95.98	300.00	9393.98
4	9393.98	93.94	300.00	9187.92
and many months later ...				
39	809.46	8.09	300.00	517.55
40	517.55	5.18	300.00	222.73
		2222.73	total interest	

The loan analysis above started with the following information:

Principal (amount borrowed) = 10000.00
Annual interest rate = 12.0 %
Monthly payment = 300.00

The monthly interest rate is found by dividing the annual rate among 12 months. For the above example the monthly rate is 1.0 %. The last three values of each line are calculated as follows:

Interest = Principal * Monthly interest rate
Payment = amount set at beginning of problem
New Balance = Principal + Interest - Payment

The new balance becomes the starting principal amount for the next month. As you can see, progress toward decreasing the principal is slow at the beginning of the loan.

Assignment:

Write a program to analyze a loan as described above using the five column format. Your program must accomplish the following:

1. Data input: The program should ask for the appropriate starting information.

2. Printing of analysis: The program must print the month-by-month analysis until the remaining principal is less than the monthly payment. You must use a **do-while** loop to solve this problem. At the bottom of the analysis you must print the total interest paid to the lending institution.

Instructions:

1. After completing your program, test it using the data given in the example. Your answers should agree within a few cents.

2. You are to analyze the following loan data:

   principal = 12000
   annual interest rate = 8.80
   monthly payment = 500.00

3. Turn in your source code and run output.

Extending the Lab:

1. We now reverse the idea. Suppose we wish to study the effect of time and compounding interest on investments. Revise the program to ask the user for:

   starting principal to invest
   annual rate of return (5%, 10%, etc)
   monthly addition to the principal
   number of months to iterate

2. The printout will be similar except the column called "payment" will be changed to investment. You should still calculate and print out the total interest and final balance.

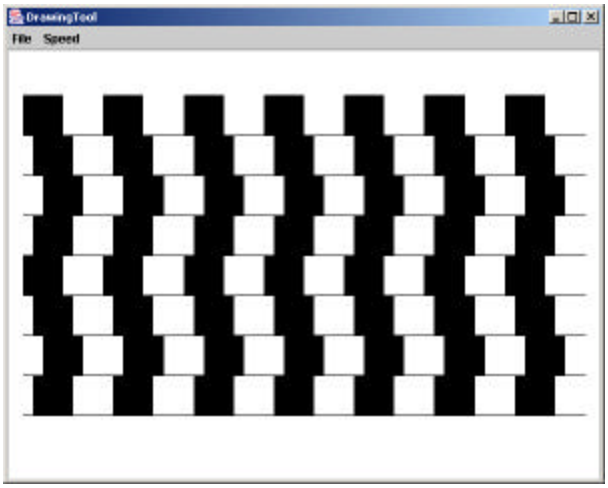
3. This lab exercise should encourage you to start investing early in life.

LAB EXERCISE

ParallelLines

Background:

In this lab, you will practice using nested for loops to recreate this image using the `DrawingTool` class:



To do this efficiently, there are a few suggestions to consider. First, there are eight rows each containing seven filled boxes. This suggests a nested **`for`** loop, like this:

```
for (int row = 0; row < 8; row++)
{
  // calculate the start of the row of squares

  // calculate and add a horizontal offset

  for (int col = 0; col < 7; col++)
  {
    // draw the square
    // calculate and position for the next square
  }

  // calculate the location and draw the line
}
```

Calculating the square position will take some work. You might want to make a quick sketch, with coordinates, to save frustration. Hint: there will be a negative x offset and a positive y offset to have the image start in the upper left corner as shown.

The `DrawingTool` class includes a method for drawing filled rectangles:

```
fillRect(double width, double height)
```

For instance, with a `DrawingTool` called `pen`, the call `pen.fillRect(40, 40)` would create a 40x40 filled square centered about the current drawing position.

Assignment:

1. Using the information given in the Background section above, write a program using nested for loops to display the image shown above.

2. Write and debug your work as *ParallelLines.java*.

3. Turn in your source code and run output.

PROGRAMMING POINTERS, LESSON 10

Syntax/correctness issues

10-1 Make sure you initialize counters and totals before entering a loop.

10-2 Be careful to avoid infinite loops. For both **`while`** and **`do-while`** loops, make sure that the Boolean expression eventually becomes false.

10-3 Be sure that you are not confusing "less than" with "less than or equal"; or "greater than" with "greater than or equal", when you are coding the Boolean condition for loops.

Formatting suggestions

10-4 Use formatting to indicate subordination of control structures. The statements that belong inside a control structure should be indented about 3 spaces. Good formatting is especially needed with nested control structures to make the code more readable. For example:

```
for (int row = 1; row <= 5; row++)
{
   // if row is odd, print a row of '*', otherwise print a row of '-'
   if (row % 2 == 1)
      c1 = '*';
   else
      c1 = '-';

   for (int col = 1; col <= 10; col++)
      System.out.print(c1);

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

Software engineering

10-5 Here are some basic guidelines for selecting the appropriate looping tool in Java:

`for` loop – used when the number of iterations can be determined before entering the loop.
`while` loop – used when the loop could potentially occur zero times.
`do-while` loop – used when the loop body should be executed at least once.

10-6 A valuable strategy for developing the Boolean expression for a conditional loop is the idea of negation. This technique consists of two important steps:

1. What will be true when the loop is done? This will be stated as a Boolean expression.
2. Then write the opposite of this Boolean expression.

For example, suppose we wanted to read positive integers from the keyboard until the running total is greater than 50.

What will be true when the loop is done? total > 50
Negate this expression: total <= 50.

We use the negated expression as the boundary condition of our loop.

```
total = 0;
do
{
   System.out.println("Enter an integer ---> ");
   num = keyboard.readInt();
   total += num;
}
while (total <= 50);
```

The expression (`total <= 50`) can be thought of as "keep doing the loop as long as total is less than or equal to 50."

This technique will be developed more completely in a later lesson.