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

MAY DECLARE A VARIABLE HERE

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
```

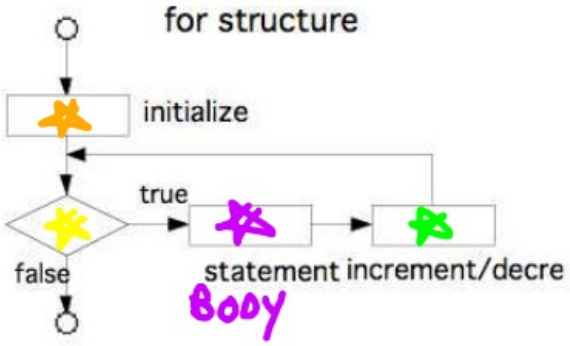
1 2 3 4

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

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:

```
*******
.******
..*****
...****
....***
.....**
      *
```

How many nested loops? 2? 3? 4?

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.

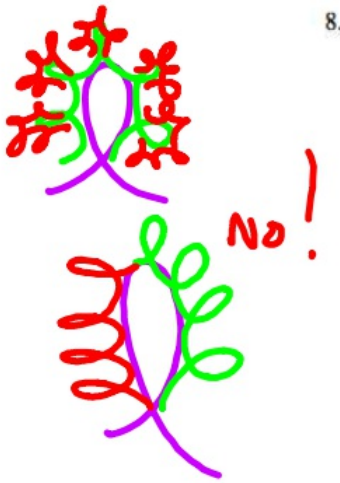
A method to print a pattern of stars:

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.print ("*");
    System.out.println ( );
  }
}
```



AAACK! Sorry!

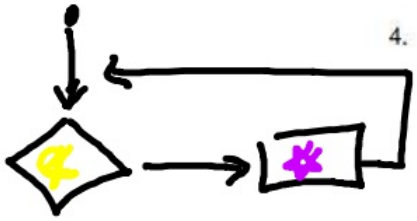
C. The do-while Loop

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.

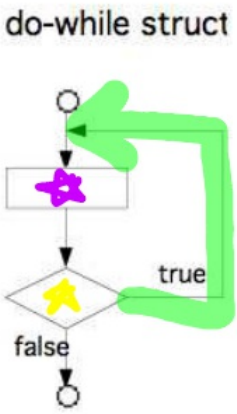
2. The **do-while** loop allows you to do a statement first, and then evaluate an exit condition. The **do-while** loop complements the **while** loop that evaluates the exit expression at the top of the loop.

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:



GREAT FOR AN INPUT SENTINEL VALUE.

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. CONDITIONAL

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.10.3, *ParallelLines*