CS112 Lecture: Loops

Last revised 3/11/08

Objectives:

1. To introduce some while loop patterns
2. To introduce and motivate the java do .. while loop
3. To review the general form of the java for loop.
4. To discuss pitfalls in writing loops
5. To introduce the use of continue and break in loop bodies

Materials:

1. Projectable of priming the pump and input validation pattern examples.
2. BlueJ project with 2 versions of RelayRobot relay race: original option 2 version, plus FixedRelayRobot, and 2 main programs, each running a three robot race.
3. Executable of Dr. Java

I. Introduction

A. In our discussion of Karel the Robot, we were introduced to the Java repetition statements

```
while ( <condition> )
    <statement>
```

AND

```
for (int <var> = <initial>; <var> < <final>; <var> ++)
    <statement>
```

As we noted at that time, these are general Java statements, and are useful for a variety of purposes beyond moving robots around in their world. In the case of the for statement, the Java statement is actually much more general than the form we learned about. In addition, there is another Java repetition statement we didn't even discuss.

B. More recently, we talked about while loops.

C. Today, we say a bit more about while loops, talk a bit more about for loops, and the introduce a third kind of java loop: the do .. while.

II. More about the while Statement

A. There are some important and useful patterns that arise in conjunction with while loops.

1. One such pattern is the *priming the pump* pattern that frequently arises in conjunction with *sentinel values*.

   Example: Suppose we wanted to write a program to calculate the average grade on the exam. Rather than asking the user to count the number of grades ahead of time, we will allow the user to just enter the scores, entering a value of -1 when finished. We'll assume we have a GUI dialog called `inputBox` with a method called `getDouble()` that gets a double value (don't worry about how this is done - that's a different topic)

   PROJECT

```
double sum = 0;
double count = 0;
double grade = inputBox.getDouble("Enter first grade:");
while (grade >= 0)
{
    sum = sum + grade;
    count ++;
    grade = inputBox.getDouble(
        "Enter next grade or - 1 if done:");
}
double average = sum / count;
```

   a) The value -1 that the user enters to terminate the loop is called the *sentinel* value.

      (1) The sentinel value must be chosen to be one that cannot possibly be a legitimate value for the program to process. Here, we assume that a negative grade on an exam is impossible.

      (2) The sentinel value can either be couched as a single value, or as a set of values that meet some criterion. (E.g. as the example program is written, any negative value the user enters would suffice. This is good defensive programming.)

      (3) Care must be used in choosing the sentinel value - lest surprising things happen.

Example: One time a company's payroll system stopped paying employees with names beginning with DR and up. Upon examination, it was found that the company had recently hired a person whose name was DONE. Unfortunately, the authors of the payroll program had used an employee name of "DONE" as the sentinel for indicating the end of the list of employees!

(4) In some cases, if all possible values of input could be legitimate, a sentinel-controlled loop cannot be used.

b) A characteristic of the sentinel-controlled loop - called the *priming the pump pattern* is that the step that ultimately produces the sentinel value (here the call to inputBox.getDouble) occurs *twice*:

(1) Once before the initial entry to the loop (to prime the pump)

(2) Once at the very end of the loop body (to get the value that decides whether or not we go around again.)

(3) This is a consequence of the fact that the statement that produces the sentinel is done one more time than the rest of the statements in the loop body - because the last time it is done, it produces the sentinel value and the loop body is not done.

2. Another pattern could be called the *input validation pattern*. Suppose we wanted to ensure that the grade entered was never greater than 100.

PROJECT

```
double sum = 0;
double count = 0;
double grade = inputBox.getdouble("Enter first grade:");
while (grade >= 0)
{
    while (grade > 100)
        grade = inputBox.getDouble(
          "Please enter a value not greater than 100:");
    sum = sum + grade;

    count ++;
    grade = inputBox.getDouble(
        "Enter next grade or - 1 if done:");
}
double average = sum / count;
```

a) This code actually has a problem. What is it?

If the user enters a grade greater than 100, and then, upon re-entry, enters a negative grade, the value is processed as if it were a valid grade.

b) To fix, we would need to validate each time input is requested

PROJECT

```
double sum = 0;
double count = 0;
double grade = inputBox.getDouble("Enter first grade:");
while (grade > 100)
    grade = inputBox.getDouble(
       "Please enter a value not greater than 100:");
while (grade >= 0)
{
    sum = sum + grade;
    count ++;
    grade = inputBox.getDouble(
        "Enter next grade or - 1 if done:");
    while (grade > 100) grade = inputBox.getDouble(
        "Please enter a value not greater than 100:");
}
double average = sum / count;
```

Note: the input validation pattern is distinct from the priming the pump pattern - e.g. we might use it without the priming the pump pattern if we did not need the former.

III. The do ... while loop

A. One important characteristic of the while loop is that its body can be done zero or more times. Related to that is the necessity of ensuring that the variables involved in the condition being tested have appropriate initial value set before encountering the loop (e.g. priming the pump).

B. There are some situations in which the logic of what is being done mandates that the loop body be done at least once. Often, the first time through the loop is special in some way in these cases.

1. *Example:* Consider the "Karel the Robot" steeple chase project. As you recall, option two involved turning this into a relay race, using two robots. It turns out that this change could be made by creating two classes of robots, one of which extends the other and overrides one method.

   Show code for `runRace()` method of `RacerRobot` and `RelayRobot`. Note that the only difference is that the former runs until it is next to a beeper, and the latter until it is next to a robot. All other methods of `RelayRobot`. are inherited

2. Now consider a further modification to use *three* robots. The first runs to the second, the second to the third, and the third to the finish. It would appear at first that we can do this by the same approach, using two classes, with two instances of the "relay" class and one of the finisher class.

   Show code for main method of `Project1Modified`

   DEMO - run slowly near handoff to second relay robot

3. Why did this code exhibit the problem?

   ASK

   The program crashes because the second robot (doofus) stops where he starts, dropping the beeper "torch" there, and leading the third robot to try to pick up a beeper "torch" that has not yet been delivered, leading to an error shutoff.

   Recall that the runRace() loop for the relay robot looks like this:

   ```
   while (! nextToARobot())
        advanceOneStep();
   ```

   Now consider what happens to the doofus. We want him to run until he is next to zelda. However, when he starts out, he is still next to the anthony (who has just handed off the beeper "torch".) Since the `nextToARobot()` test succeeds if a robot is next to *any* other robot, the while loop terminates immediately, leading to the error condition.

C. Situations like this can be handled by using a different sort of loop - the do .. while loop:

```
do
    <statement>
while <condition>
```

1. In this form of loop, the rules for <statement> and <condition> are the same as for the ordinary while loop
   a) The statement can either be a single statement or a compound statement (enclosed in braces).
   b) The condition can be any boolean expression. Note that, as in the ordinary while loop, the condition being true causes us to continue executing the loop.

2. In this form of loop, the first iteration is special, and is always done. The loop condition is not tested until *after* the first iteration completes.

   Our failed relay race could be handled successfully by changing the while loop in the runRace() method of the initial robots to a do..while

   SHOW CODE for runRace() method of FixedRelayRobot. Of course, we also need to change the main program to create this new kind of robot - SHOW Project1Fixed

   DEMO

3. Note that each type of loop (while and do .. while) has its appropriate uses, and some thought should be given as to which type of loop is most appropriate in any given case.

   EXAMPLE: Show code for runRace() method of SteepleChaseRobot. It still uses a while loop, not a do .. while. Why?

   ASK

   Because we do want to allow for the possibility of this loop being done zero times, if the robot starts out on the square containing the beeper that marks the end of the race (as in one of the test worlds for this project). Note that if a while loop is used here, a special case if statement (as some students used in a previous year) is *not* needed.

4. For the grade averaging example we did earlier, could we change the while loop to a do .. while ? Should we have?

   ASK

a) Yes - a do .. while is appropriate in this case, because we cannot compute a meaningful average if there are no grades. (We would get a divide by zero error when we divide by count).

b) Note that changing to a do .. while does *not* eliminate the need for the priming read in this case. We still have to read one more grade than we actually use.

IV. The Java for Loop

A. The final kind of loop we need to consider is the for loop. It has the general form:

```
for ( < initialization > ; < test > ; < increment > )
    < statement >
```

where

1. Initialization is zero or more initializations of variables (separated by commas) to be done *before* the loop is entered the first time.

   a) It is possible to *declare* variables in the initialization - in which case the scope of the variable is limited to just the loop

   EXAMPLE:

   ```
   for (int i = 0; i < 10; i ++)
       System.out.println(i);

   // Declaration of i is no longer in effect here
   ```

   b) The initialization part of the loop can be empty, but the semicolon is still required.

2. Test is a condition to be tested before entering the loop each time (as in a while loop)

3. Increment is zero or more assignments of variables (separated by commas) to be done *after* each time through the loop body.

   a) Frequently, the increment part makes use of the Java increment (++), decrement (--), and/or shorthand assignment operators

   b) However, it is also possible to increment by amounts other than +/- 1.

Example: the following code prints out all odd numbers between 1 and 20.

```
for (int i = 1; i < 20; i = i + 2)
    System.out.println(i);
```

c) The increment part of the loop can be empty, but the semicolon is still required.

4. As with other kinds of loop, statement can either be a single statement or a compound statement enclosed in braces.

B. In fact, a for loop lis exactly equivalent to the following while loop:

```
< initialization >
while ( < test > )
{
    < statement >;
    < increment >;
}
```

C. Most frequently, the for loop is used for *counter-controlled loops*, in which case

1. The initialization creates a counter variable and initializes it (most often to 0 or 1; sometimes to a maximum value if we want to count backwards)

2. The test checks to see if the counter variable has reached its limit yet.

3. The increment steps the counter to the next value

4. *EXAMPLE:* Print the square roots of numbers in the range of 1.0 up to (but not including) 2.0 in steps of 0.1

```
for (double x = 1.0; x < 2.0; x += 0.1)
    System.out.println("sqrt(" +x+ ") = "+Math.sqrt(x));
```

DEMO using Dr. Java

5. *EXAMPLE:* Calculate and print the powers of 2 up to 2^{30} (the largest power of 2 representable as a Java int) . (Note the use of two local variables declared in the initialization and modified by the increment)

```
for (int i = 0, n = 1; i <= 30; i ++, n *= 2)
    System.out.println("2 to the power " +i+ "=" +n);
```

DEMO using Dr. Java

D. Java 1.5 introduced another version of the for loop (called the enhanced for loop) that can be used with collections, when we want to do something with every member of a collection The book does not discuss this here, nor will we, since we haven't met any kind of collections. We will cover it (as the book does) when we deal with arrays.

E. As a general rule of thumb, to which there are certainly exceptions, one should use the various types of loops as follows:

1. For *indefinite iteration*, where the loop body can be done zero or more times: use a while loop

2. For *indefinite iteration,* where the loop body must be done at least once: use a do ... while loop

3. For *definite iteration*: use a for loop

V. Pitfalls in Writing Loops

There are several pitfalls to watch out for when writing loops

A. Infinite loops: Always be sure that you can convince yourself that the loop will eventually terminate - i.e. that the loop condition will eventually become false.

B. Exact count with real numbers: beware of a loop body whose test is designed to terminate the loop on *exact equality* with some value, especially when the value is a real number (double or float)

EXAMPLE: Demonstrate the following loop with Dr. Java:

```
for (double d = 1.0; d != 2.0; d += 0.1)
    System.out.println(d);
```

(Use Reset Interactions in Tools menu to stop)

To solve the problem, we must switch the end test from requiring exact equality to <=

DEMO

C. Off by one errors: a very common error with loops is to make a slight error that results in the loop body being executed one too many times

EXAMPLE:

```
// turnLeft() three times

for (int i = 0; i <= 3; i ++)
    turnLeft();
```

What does this actually do?

ASK

Turns left *four* times.

VI. Using the continue and break Statements in Loops

A. All three of the standard java loops incorporate a boolean expression that is tested *once* per iteration in order to decide whether to (re) enter the loop.

B. Sometimes, a condition may arise in the middle of a loop that calls for an immediate exit from the loop. *EXAMPLE:* Suppose we were writing a loop that repeatedly reads and totals the cost of individual items purchased. Suppose further that we want to exit the loop immediately if the customer exceeds some predetermined spending limit.

1. Java allows a break statement (just like the one we saw in conjunction with switch) to occur inside the body of a loop. If it is executed, the loop containing it is terminated immediately.

   EXAMPLE: (Continuing the above - using pseudocode)

```
total = 0;
do  // Note - loop terminates early if total would
    // exceed limit
{
    purchase = get next purchase amount from user;
    if (purchase + total > limit)
    {
        tell user (s)he'd go over the limit;
        break;
    }
    total = total + purchase;
    ask if user wants to enter another item
}
while (user wants to enter another)
```

2. In the case of nested loops, a break always exits the innermost loop containing it. It is also possible to attach a label to a loop and exit a loop other than the innermost one by specifying the label, but we won't go into this here.

C. Java also has a similar statement called continue that can be used inside a loop to specify that we *start a new iteration* of the loop immediately, rather than terminating the loop. (The loop may still terminate if the condition for staying in it is false.)

D. Most texts do not discuss either continue or the usage of break within a loop, for good reason. The misuse of these statements can easily lead to confusing code - especially continue. In general, anytime you contemplate using one of these, you should ask if it's *really* the clearest way to write the code. Some writers would argue that the continue statement is *never* the best way to write something and so should not be used at all. (It's a carryover from C).

EXAMPLE: The above loop written without using break:

```
total = 0;
overLimit = false;
do
{
    purchase = get next purchase amount from user;
    if (purchase + total > limit)
    {
        tell user (s)he'd go over the limit;
        overLimit = true;
    }
    else
    {
        total = total + purchase;
        ask if user wants to enter another item
    }
}
while (! overLimit and wants to enter another)
```

Which solution is clearer and more readable? *ASK*

Note use of comment to highlight possibility of mid-exit in first form