CS112 Lecture: Control Structures

Last revised 2/18/08

Objectives:

1. To review previously-covered material on while and for
2. To introduce DeMorgan's laws
3. To introduce the Java switch statement
4. To discuss the tradeoff between if and switch statements

Materials:

1. Projectable version of code illustrating switch
2. GPA Tester demo

I. The While Statement

A. Chapter 7 of the Bruce book introduces the while statement. Of course, we have previously met this in connection with Karel, so we just want to review this statement briefly here.

B. The general form of the while statement is

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

1. The condition can be any boolean expression - so all of the things we said about this in connection with if applies here as well. In particular, the condition can be any valid boolean expression

   Review kinds of boolean expressions -

   ASK

   a) a simple test for equality (==, !=)

   b) a relational comparison (>, >=, <, <=)

   c) a boolean variable

   d) a boolean method

   e) a compound condition formed from two or more booleans joined by &&, ||, or ^.

2. The statement - called the loop body - can be any Java statement, including another control structure of a compound statement enclosed in { }.

3. The condition is tested only when deciding to perform another iteration of the loop. It can become false during the execution of the body, but is not tested until it is time to consider another iteration of the loop.

   a) Example: In project 1, the following would not work:

```
while (! nextToABeeper())
{
   while (frontIsClear())
        move();
   climbHurdle();
}
```

   Why?

   ASK

   b) The following would work - though it is not good practice!

```
while (! nextToABeeper())
{
   while (frontIsClear() && ! nextToABeeper())
        move();
   if (! nextToABeeper())
        climbHurdle();
}
```

   Why does the if statement not need to test to see if the robot's front is blocked?

   ASK

   We would not have exited the inner while loop unless at least one of the two conditions were false. If `! nextToABeeper()` is true, then it must be the case that `frontIsClear()` is false - hence the robot must be up against a hurdle.

II. DeMorgan's Laws

A. While compound conditions can be confusing, it is often the case that they cannot be avoided. In such cases, it is helpful to understand some basic properties of such expressions.

B. One of the helpful things to know is a pair of equalities known as DeMorgan's laws

1. ! (A && B) ≡ A || B

2. ! (A || B) ≡ A && B

III. The Switch Statement

A. The if and if ... else statement allows the specification of one or two alternatives based on a single true or false test. Sometimes, we need to do one of a large number of options based on a value that can assume several different values.

Example: Suppose that a student object includes an instance variable called year which has the value 1 for freshman, 2 for sophomore ... Suppose, further, that we want to display a student's class as a word, rather than as a number - i.e. we want to display the word "Freshman" etc.

1. We could do this using nested ifs, as follows: (PROJECT)

```
if (year == 1)
     yearDisplay.setText("Freshman");
else
     if (year == 2)
          yearDisplay.setText("Sophomore");
     else
          if (year == 3)
               yearDisplay.setText("Junior");
          else // assume year must be 1..4
               yearDisplay.setText("Senior");
```

2. As we saw earlier, in a case like this, there is a slightly different way of formatting the nested if statement that still has the same meaning, but makes what is being done a bit clearer:

```
if (year == 1)
     yearDisplay.setText("Freshman");
else if (year == 2)
     yearDisplay.setText("Sophomore");
else if (year == 3)
     yearDisplay.setText("Junior");
else // assume year must be 1..4
     yearDisplay.setText("Senior");
```

3. However, another alternative is to use a different Java statement known as switch:

```
switch(year)
{
     case 1:
          yearDisplay.setText("Freshman");
          break;

     case 2:
          yearDisplay.setText("Sophomore");
          break;

     case 3:
          yearDisplay.setText("Junior");
          break;

     case 4:
          yearDisplay.setText("Senior");
          break;
}
```

B. A switch statement may be used when you have an expression that assumes one value from a discrete set of possible values - which must be integers or characters. (A switch statement cannot be used with floats or doubles, because the set of possible values is mathematically infinite, even if actually finite in the computer implementation; nor can it be used with Strings or objects for the same reason)

C. A switch statement has the following form: (PROJECT)

```
switch( <expression>)
{
    case < value > :
        < statement >
        < statement >
        ...

    case < value > :
        < statement >
        < statement >
        ...

    case < value > :
        < statement >
        < statement >
        ...

    ...

    [ default:
        < statement >
        < statement >
        ... ]
}
```

Where:

1. The type of the expression and the types of the values associated with each case label must all be the same

2. Any given value occurs in at most one label

3. The last statement in each group of statements following a case label is typically a break statement. (More on this shortly).

4. The default part is optional. (This is why the example encloses it in brackets - you don't literally use the brackets in the program - they are a device for indicating "this is optional")

D. The switch statement is evaluated as follows:

1. The expression is evaluated

2. A check is made to see if the resulting value occurs in any of the case labels. If it does, then execution continues at the statement just after the case label - otherwise, it continues at the statement just after the default: label or (if there is no default) the switch statement terminates (no case is done) and execution continues after the closing right brace.

3. Once execution begins at a certain statement, it proceeds sequentially until either

   a) A break statement is encountered - in which case the switch statement terminates after the closing right brace.

   b) The last statement in the entire switch statement (just before the closing right brace) is executed.

E. The need for ending each case with a break statement is a common source of errors, so one might ask why the switch statement was designed this way - i.e. why doesn't the start of each new case terminate the processing of the previous case? The answer is that the approach that Java takes allows us to deal with situations where several cases are handled the same way - e.g.

Suppose we want to output the word "Vowel", "Consonant", or "Vowel or consonant" depending on what a given letter is. We could code this as follows (assuming letter is always an letter of the alphabet) (PROJECT)

```
switch(letter)
{
   case 'a': case 'e': case 'i':  case 'o': case 'u':
        System.out.println("Vowel");
        break;

   case 'y':|
        System.out.println("Vowel or consonant");
        break;

   default:
        System.out.println("Consonant");
}
```

F. As a final note, it is possible to nest switches in various ways:

1. One can nest a switch statement inside one case of another switch statement

2. One can nest a switch statement inside the true or else part of an if statement.

3. One can nest an if statement inside one case of a switch statement

4. etc.

5. Though such nesting is possible, it must be approached with some caution!

G. The switch statement can only be used when the values being tested form a discrete set of fixed integer or character values.

1. In an earlier class, we developed an example of implementing Gordon's "minimum GPA" rule, where the minimum acceptable GPA is based on the number of credits earned (0-26, 27-55; 56 and up)

   In this case, a switch statement could not be used unless we were prepared to explicitly list all possible values of credits - e.g.

   case 0: case 1: case 2: *remaining values between 3 and 25* case 26:

   year = "Freshman"

2. However, when a switch statement can be used, it is often the clearest way to handle a particular situation.

3. Sometimes, it pays to create enumerated constants to facilitate using a switch statement

   e.g. if we frequently had statements like:

```
if (year.equals("Freshman"))
    ...
else if (year.equals("Sophomore"))
    ...                                                    ...
```

   It might pay to define constants:

```
public static final int FRESHMAN = 1;
public static final int SOPHOMORE  = 2;
public static final int JUNIOR = 3;
public static final int SENIOR = 4:
```

to allow creation of statements like:

```
switch(year)
{
    case FRESHMAN:
        ...
```

PROJECT

H. A further demonstration of use of switch with symbolic constants

GPATester

DEMONSTRATE

PROJECT, discuss code