CS112 Lecture: Making Choices

Last revised 1/19/06

Objectives:

1. To review the Java if and if ... else statements
2. To introduce relational expressions and boolean operators
3. To discuss nested if statements
4. To review rules for transforming if statements

Materials:

1. Dr. Java for demos
2. BlueJ project containing WhatADrag,
3. Projectable of operator precedence table
4. Projectable - Gordon catalog rules for minimum GPA to avoid probation
5. Projectable showing various ways of structuring this as nested ifs
6. Projectable from Karel Conditionals lecture showing various transformations

I. Introduction

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

```
if ( <condition> )
    <statement>


if ( <condition> )
    <statement>
else
    <statement>
```

B. 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.

C. Recall that we saw with Karel that the <statement> governed by the true or else part of an if may either be:

1. A single statement

2. A compound statement enclosed in braces.

   This is true in general in Java, not just with Karel

II. Introduction to Boolean Expressions

A. In the general form of the if statement, the condition can be any boolean expression - i.e. any expression whose ultimate value is either true or false.

1. In Karel's world, we met a variety of built-in boolean methods - e.g. frontIsClear(), nextToABeeper()

2. In Karel's world, we also learned how to defined boolean methods, such as rightIsClear()

3. The objectdraw library used by the book includes various geometric shapes, all of which have a contains() method that can be used to test whether the shape contains a given point.

4. In the general world of Java programming, there are many ways to form boolean expressions

B. One of the simplest forms of boolean expression is a *relational expression*. A relational expression consists of two arithmetic expressions compared using one of the *relational operators*:

1. == - true iff the two expressions being compared are equal.

   Note the difference between == and =. The relational operator *asks* if two expressions are equal; the assignment operator *causes* the variable on the left be equal to the expression on the right.

2. != - true iff the two expressions being compared are *not* equal

   Note:

```
if (A == B)          and      if (A != B)
   S1;                           S2;
else                          else
   S2;                           S1;
```

   are functionally identical

3. < - true if the first expression is less than the second

4. > - true if the first expression is greater than the second

   Question: are the following two statements identical:

```
if (A < B)          and        if (A > B)
    S1;                            S2;
else                           else
    S2;                            S1;
```

ASK

No! - they are different in the case A and B are equal

5. <= - true if the first expression is less than or equal to the second (exact opposite of >)

6. >= - true if the first expression is greater than or equal to the second (exact opposite of <)

Example: Print a warning if a student's gpa is below 2.0.

```
if (gpa < 2.0)
    System.out.println("Warning - gpa too low!");
```

DEMO with Dr. Java

C. In addition to the relational operators, a class may define a method whose return type is boolean

Example: `rightIsClear()` in project 1

D. It is also possible to have boolean variables.

Example: WhatADrag example from the book - variable boxGrabbed

DEMO/PROJECT

E. More complicated boolean expressions may be built up by combining simpler expressions using the boolean operators &&, ||, ^, and !

1. The && operator is read "AND", and has the following meaning: the expression is true just when *both* of its subexpressions are true. This can be represented by the following *truth table:*

A	B	A && B
FALSE	FALSE	FALSE
FALSE	TRUE	FALSE
TRUE	FALSE	FALSE
TRUE	TRUE	TRUE

2. The || operator is read "OR" and has the following meaning: the expression is true just when *either* of its subexpressions is true. This can be represented by the following *truth table:*

A	B	A || B
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	TRUE

3. The ^ operator is read "Exclusive or" or "XOR" and has the following meaning: the overall expression is true just when *exactly one* of the two subexpressions is true. This can be represented by the following *truth table:*

A	B	A ^ B
FALSE	FALSE	FALSE
FALSE	TRUE	TRUE
TRUE	FALSE	TRUE
TRUE	TRUE	FALSE

4. The ! operator is read "NOT" and has the following meaning: the overall expression is true just when the subexpression is *not* true. This can be represented by the following *truth table*:

A	! A
FALSE	TRUE
TRUE	FALSE

5. As a further point to note, both && are || are defined as *short circuit* operators - which means that each first evaluates its left subexpression, and then evaluates its right subexpression if and only if necessary to determine the final result -

   a) e.g.

      (1) If we have A && B, and A is false, B is not even evaluated - the value of the overall expression is necessarily false, and the value of B could not impact that.

      (2) If we have A || B, and A is true, B is not even evaluated - the value of the overall expression is necessarily true, and the value of B could not impact that.

   b) Short circuit evaluation becomes important if the second subexpression involves a computation that might not always be legal, or that involves calling a method - e.g.

   (1)
```
if (credits != 0 && gradePoints/credits < 2.0)
...
```
   gradePoints / credits would result in a division by zero error if credits is zero, but short-circuit evaluation guarantees that the division will not even be done if credits == 0.

   (2)
```
if (! frontIsClear() ||
      nextSquareContainsABeeper())
...
```
   (assume nextSquareContainsABeeper() is a method that advances one block, checks for a beeper, then turns around and goes back.)

   nextSquareContainsABeeper() would result in an error shutoff if the robot's front is not clear, but short-circuit evaluation guarantees that the method will not be called if the robot's front is not clear.

   c) You should be aware that there are non short-circuit forms of the and and or operators: & and | - though you will likely never need to use them. (They are a relic of a slightly different way of handling boolean values in C and C++).

6. When we build up boolean expressions containing a variety of different operators, it becomes important to keep the relative precedence of the operators in mind. In the following list, operators we have seen thus far (there are quite a few others) are listed from highest precedence to lowest.

PROJECT

unary operators	- !
multiplicative operators	* / %
additive operators	+ -
comparison operators	> < >= <=
equality operators	== !=
boolean and	&&
boolean or	||
assignment	=

As always, parentheses can be used for grouping to override precedence and for clarity (and generally should be used with complex expressions!)

F. An example of a complex boolean expression

The following is an illustration of a complex boolean expression. To exercise our understanding of operator precedence, unnecessary parentheses have been omitted. *This is probably not a good style for general programming!* What is the order of operator evaluation?

ASK

```
a + 3 < b - 2 ||  ! (a > 7 && b < 6)
  1   3   2   4   8    5   7    6
```

Note that the parentheses are needed after the not operator, because the unary operator ! takes higher precedence than the binary operators. If we had omitted the parentheses, we would have had ! a - an illogical operation, since a is presumably a number and ! only applies to booleans. Also note that computations 5,6,7, and 8 are not done at all if the left hand side is true.

III. Nested If Statements

A. We saw, in conjunction with Karel, that it is possible to have if statements within other if statements - what are called *nested* if statements. Once again, what we did with Karel is true in general in Java.

B. Actually, it is often the base that the same problem can be solved in more than one way by using a combination of boolean expressions and nested if statements.

Example: Gordon has a policy that a student must have a gpa of 2.0 to graduate. A student with a gpa of less than 2.0 is subject to being put on probation or being suspended. However, new students are held to a less stringent standard in their first couple of years - though they must eventually reach 2.0. This policy is summarized in the following table in the college catalog:

TRANSPARENCY- College catalog p. 35

Suppose we wish to write a program fragment that outputs one of the following messages, given a student's gpa and credits: OK, OK for now, or Below standard. (Assume that gpa and credits are variables) (PROJECT examples)

1. One way to do this would be with nested if statements - e.g. the following would work:

```
if (gpa < 2.0)
    if (credits <= 26)
        if (gpa < 1.6)
            System.out.println("Below standard");
        else
            System.out.println("OK for now");
    else
        if (credits <=55)
            if (gpa < 1.8)
                System.out.println("Below standard");
            else
                System.out.println("OK for now");
        else
            System.out.println("Below standard");
else
    System.out.println("OK");
```

2. This could be indented using an alternate style which may be a bit more readable:

```
if (gpa < 2.0)
    if (credits <= 26)
        if (gpa < 1.6)
            System.out.println("Below standard");
        else
            System.out.println("OK for now");
    else if (credits <=55)
        if (gpa < 1.8)
            System.out.println("Below standard");
        else
            System.out.println("OK for now");
    else
        System.out.println("Below standard");
else
    System.out.println("OK);
```

3. Another approach would also work. (As above, we used an alternate style of indentation for the inner ifs)

```
if (credits <= 26)
    if (gpa < 1.6)
        System.out.println(“Below standard”);
    else if (gpa < 2.0)
        System.out.println(“OK for now”);
    else
        System.out.println(“OK”);

else if (credits <= 55)
    if (gpa < 1.8)
        System.out.println(“Below standard”);
    else if (gpa < 2.0)
        System.out.println(“OK for now”);
    else
        System.out.println(“OK”);
else
    if (gpa < 2.0)
        System.out.println(“Below standard”);
    else
        System.out.println(“OK”);
```

4. Yet another approach makes use of complex conditions to reduce the number of statements:

```
if (gpa >= 2.0)
    System.out.println("OK");
else if (gpa >= 1.6 && credits <= 26 ||
        gpa >= 1.8 && credits <= 55)
    System.out.println("OK for now");
else
    System.out.println("Below standard");
```

5. Which approach is the most understandable? Why?

   ASK

C. One problem that arises with nested ifs that we have already discussed in connection with Karel is the *dangling else problem.* What is it?

   ASK

   1. Example:

```
if (condition-1)
    if (condition-2)
        statement-1;
else
    statement-2;
```

   Under what circumstances is statement-2 done?

   a) According to the programmer's apparent intention as specified by the indentation?

      ASK

   b) According to the way the Java compiler actually interprets the statement?

      ASK

   2. How do we solve the dangling else problem?

      ASK

Assuming we really intend the interpretation implied by the indentation, we must use braces even though no compound statements are involved

```
if (condition-1)
{
    if (condition-2)
        statement-1;
}
else
    statement-2;
```

3. For this reason, and others, some authors recommend that you always use braces around the true and else parts of an if statement, even when they are not mandatory. I personally don't like that style, because I think it makes the program harder to read. However, you can follow whichever convention works best for you. (Just be consistent)

IV. Guidelines for Using If Statements

A. Given the different ways of using if statements in Java (if, if .. else, nested if), how should we decide which kind to use for a given situation?

The general rule is to use the kind of statement that produces the most understandable code.

B. Nested ifs tend to be difficult to read, and are prone to the "dangling else" problem.

1. A good general rule is that nested ifs should be avoided when possible - especially if nested more than two deep.

2. An exception to that general statement arises when nested ifs are used to achieve a multi way selection - as in the following (based on the rule in Gordon's catalog for determining freshman, sophomore etc. standing)

```
if (credits > 85)
   year = "Senior";
else if (credits > 55)
   year = "Junior";
else if (credits > 26)
   year = "Sophomore";
else
   year = "Freshman";
```

Note how we use a different style of indentation to represent the coordinate nature of the various tests. In fact, some programming languages include a special reserved word for this situation - e.g. Ada has the reserved word elsif.

3. Sometimes there is a tradeoff between using compound conditions and nested ifs. In such cases, it may make sense to sketch out both alternatives on scratch paper and go with the one that is more readable.

C. When we discussed the if statement in connection with Karel, we learned about some transformations that can be used to clarify if statements:

ASK CLASS

1. Test reversal

```
if (condition)
    instruction1;
else
    instruction2;
```

is equivalent to

```
if (! condition)
    instruction2;
else
    instruction1;
```

This is especially useful if the we don't want to do anything if the condition is true

2. Another improvement is *bottom factoring*. If the last thing done in both the true and else part is the same, it can be factored out and moved to after the if statement:

3. A third improvement is *top factoring*. If the first thing done in both the true and else parts is the same - *and it does not affect the test* - it can be factored out and moved to before the if statement

4. A final improvement is removing redundant tests. If some condition must necessarily be true when we reach it, we don't need to test it (and shouldn't, for reasons of efficiency and clarity.)

PROJECT Example of all from earlier Karel lecture