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 introduce the Java switch statement
5. To review rules for transforming if statements, and discuss the tradeoff between if and switch statements

Materials:

1. Transparency - Wu Table 6.2
2. Transparency - Gordon catalog rules for minimum GPA to avoid probation
3. Handout showing various ways of structuring this as nested ifs

I. Introduction

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

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

```java
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


   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. 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: Pop up a message box containing a warning if a student’s gpa is below 2.0. (Assume message box object is already created)

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

C. In addition to the relational operators, there are certain other operations that yield a boolean variable.

1. The == and != operators can be applied to objects as well as to arithmetic expressions. In this case, they test to see whether the objects have the same identity.

Note the distinction between comparing identities of objects and comparing values:

    Float a = new Float(1);  
    Float b = new Float(1);  
    if (a == b)              // false  
      if (a.floatValue() == b.floatValue()) // true  
      if (a == b.floatValue())              // illegal

2. Some classes implement a method named equals() that compares two objects of that class to see if they have the same value.
Continuing the above:

```java
if (a.equals(b)) // true
```

3. The `instanceof` operator can be used to test if an object is an instance of a given class (including one of its subclasses) - e.g.

```java
RightTurnerRobot karel = new RightTurnerRobot ... if (karel instanceof RightTurnerRobot) // true
if (karel instanceof Robot) // true
// assuming RightTurnerRobot extends Robot
if (karel instanceof World) // false
```

4. A class may define a method whose return type is `boolean`

   *Example:* `nextToTheTreasure()` in project 1

D. 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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A &amp;&amp; B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A ^ B</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>A</th>
<th>! A</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>TRUE</td>
</tr>
<tr>
<td>TRUE</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

5. As a further point to note, both && and || 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.

   **TRANSPARENCY - Wu Table 6.2**

E. An example of a complex boolean expression

The following is an illustration of a complex boolean. 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**

\[
\begin{align*}
   a + 3 &< b - 2 \land \neg (a > 7 \land b < 6) \\
   1 &\quad 3 &\quad 2 &\quad 4 &\quad 8 &\quad 5 &\quad 7 &\quad 6
\end{align*}
\]

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)

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

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

```java
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:

```java
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:

```java
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

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

3. For this reason, and others, Wu recommends 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. 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:

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

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

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

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

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

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

    case 4:
        yearLabel.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:

```c
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”)

12
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)

```java
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!

V. Guidelines for Using Selection Statements

A. Given the different kinds of selection statements in Java (if, if .. else, nested if, switch), 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):

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

1. E.g. a switch statement could not have been easily used in the case above 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:
   ...
   ...
D. 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.)