CS 4: Lecture 11
Monday, February 27, 2006

"switch" STATEMENTS

Some long chains of if-then-else clauses can be simplified by using a "switch" statement. "switch" is appropriate only if every condition tests whether a variable x is equal to some constant.

```
    switch (month) {
    case 2:
      days = 28;
      break;
    case 4:
    case 6:
    case 9:
    case 11:
      days = 30;
      break;
    default:
      days = 31;
      break;
    }
```

```
    if (month == 2) {
      days = 28;
    } else if ((month == 4) || (month == 6) ||
               (month == 9) || (month == 11)) {
      days = 30;
    } else {
      days = 31;
    }
```

// These two code fragments do exactly the same thing.

IMPORTANT: "break" jumps to the end of the "switch" statement. If you forget a break statement, the flow of execution will continue right through past the next "case" clause. If month == 12 in the following example, both Strings are printed:

```
    switch (month) {
    case 12:
      output("It's December.");
      // Just keep moving right on through.
    case 11:  case 1:  case 2:
      output("It's cold.");
    }
```

However, this is considered bad style, because it's hard to read and understand. If there's any chance that other people will need to read or modify your code (which is very common when you program for a business), don't code it like this. Use break statements in the switch, and use subroutines to reuse code or clarify the control flow.

Observe that the last example doesn't have a "default:" case. If "month" is not 12 nor 11 nor 1 nor 2, Java jumps right to the end of the "switch" statement (just past the closing brace) and continues execution from there.

THE "break" AND "continue" STATEMENTS

A "break" statement immediately exits the innermost loop or "switch" statement enclosing the "break", and continues execution at the code following the loop or "switch".

To motivate the "break" statement, consider the following code.

```
  do {
    s = keybd.readLine();
    process(s);
  } while (s.length() > 0);              // Exit loop if s is an empty String.
```

But we don't want to call "process(s)" when s is a signal to exit (in this case, an empty String). We want a "time-and-a-half" loop--we want to enter the loop at a different point in the read-process cycle than we want to exit the loop at. Here are two alternative loops that do the right thing. These two loops behave identically. Each has a different disadvantage.

```
  s = keybd.readLine();
  while (s.length() > 0) {
    process(s);
    s = keybd.readLine();
  }
```

Disadvantage: The line "s = keybd..." is repeated twice. It's not a big disadvantage here, but if input took 100 lines of code, the duplication would make the code harder to maintain. Why? Because a programmer improving the code might change one copy of the duplicated code without noticing the need to change the other to match.

```
  while (true) {       // Loop forever.
    s = keybd.readLine();
    if (s.length() == 0) {
      break;
    }
    process(s);
  }
```

Disadvantage: Somewhat obfuscated for the reader, because the loop isn't aligned with its natural endpoint.

Both of the methods above are used in practice. Neither is perfect, and it's a matter of taste which one you should use in any given situation.

Some loops have more than one natural endpoint. Suppose we want to iterate the read-process loop at most ten times. In the example at left below, the "break" statement cannot be criticized, because the loop has two natural endpoints. We could get rid of the "break" by writing the loop as at right below, but the result is longer and harder to read.

```
  for (int i = 0; i < 10; i++) {
    s = keybd.readLine();
    if (s.length() == 0) {
      break;
    }
    process(s);
  }
```

```
  int i = 0;
  do {
    s = keybd.readLine();
    if (s.length() > 0) {
      process(s);
    }
    i++;
  } while (i < 10 && s.length() > 0);
```

There are anti-break zealots who will claim that the loop on the right is the "correct" way to do things. Some of them feel this way because "break" statements are a little bit like the "go to" statements found in some languages like Basic and Fortran (plus the "machine language" that microprocessors really execute). "go to" statements allow you to jump to any line of code in the program. It sounds like a good idea at first, but it invariably leads to insanely unmaintainable code. And what happens if you jump to the middle of a loop? Edsger Dijkstra wrote a famous article in 1968 entitled "Go To Statement Considered Harmful", which is part of the reason why most modern languages like Java don't have "go to" statements.

WARNING: It's easy to forget exactly where a "break" statement will jump to. For example, "break" does not jump to the end of the innermost enclosing "if" statement. An AT&T programmer introduced a bug into telephone switching software in a procedure that contained a "switch" statement, which contained an "if" clause, which contained a "break", which was intended for the "if" clause, but instead jumped to the end of the "switch" statement. As a result, on January 15, 1990, AT&T's entire U.S. long distance service collapsed for eleven hours.

That code was actually written in C, but Java's loop syntax and "break" semantics are identical.

For this reason, Java (unlike C) allows you to attach labels to any kind of enclosing statement, including "if" statements and any group of statements placed { in braces }. Then, "break" can jump to the end of any labeled enclosure that encloses the "break" statement.

```
  test:
  if (x == 0) {
    loop:
    while (i < 9) {
      stuff: {
        switch(z[i]) {
        case 0: break;           // Jump to statement1
        case 1: break stuff;     // Jump to statement2
        case 2: break loop;      // Jump to statement4
        case 3: break test;      // Jump to statement5
        case 4: continue;        // Jump to location 3
        default: continue loop;  // Jump to location 3
        }
        statement1();
      }
      statement2();
      i++;
      // location 3
    }
    statement4();
  }
  statement5();
```

The "continue" statement is akin to the "break" statement, except
(1) it only applies to loops (so you can't write "continue stuff" or "continue test" above--look out, AT&T), and
(2) it doesn't necessarily exit the loop; another iteration may commence (if the condition of the "while"/"do"/"for" loop is satisfied).

I suggest you always use labeled break and continue statements (except perhaps in uncomplicated "switch" statements).

Finally, I told you that "for" loops are identical to certain "while" loops, but there's actually a subtle difference when you use "continue". What's the difference between the following two loops?

```
  int i = 0;                          | for (int i = 0; i < 10; i++) {
  while (i < 10) {                    |   if (condition(i)) {
    if (condition(i)) {               |     continue;
      continue;                       |   }
    }                                 |   call(i);
    call(i);                          | }
    i++;                              |
  }                                   |
```

Answer: when "continue" is called in the "while" loop, "i++" is not executed. In the "for" loop, however, i is incremented at the end of _every_ iteration, even iterations where "continue" is called.