Computer Science 202
Introduction to Programming
The College of Saint Rose
Fall 2012

Topic Notes: Conditional Execution

All of our programs so far have had one thing in common: they are entirely *sequential*. The statements in our `main` methods all execute, one after another, in the order they are encountered in our program.

While this is useful in many cases, we would like to move beyond that and have our programs start to *make choices*. We would like to be able to check for a certain condition. If the condition is "true" we would like to do one thing, if it's "false" we would do something else or possibly nothing at all.

Thinking of every day algorithms, this is something we do all the time.

- If I am still hungry, I will go for seconds.
- If it is a weekday, I will set my alarm to get up for class. Otherwise, I will sleep as long as I would like.
- If the dough is too watery, add more flour.
- If the student's score is at least 95% on the spelling test, put a sticker at the top.

Visual Logic's If Condition

We will begin our exploration of this idea with Visual Logic, using the "If Condition".

We will develop a small program that asks what year you were born, and prints out a message that depends on that.

In the "If Condition" in Visual Logic, we can specify a *boolean condition* – something that evaluates to either true or false – and depending on that result, the execution will continue on one of two paths.

Our program will print out your age, and an extra message if you were born before 1995 (when Java was born).

Java's `if` Statement

Java has a construct that does the same thing. We will look at a simplified version first: one that either does something or not.

The basic form is

```
if (<boolean condition>) {
   statement1;
   statement2;
   ...
}
```

where `<boolean condition>` is some Java expression that evaluates to `true` or `false`. If it evaluates to `true`, the statements inside the curly braces following the condition are executed. Otherwise, they are skipped.

Let's use this to develop our program.

See Example: OlderThanJava

There are many things that can be used to construct a boolean expression, but we will start with the standard relational operators and use them to compare numeric values.

Expression	When is it `true`?
`x > y`	when `x` is greater than `y`
`x < y`	when `x` is less than `y`
`x >= y`	when `x` is greater than or equal to `y`
`x <= y`	when `x` is less than or equal to `y`
`x == y`	when `x` is equal to `y`
`x != y`	when `x` is not equal to `y`

Java's `if-else` Construct

The `if` statement we saw above allows us to execute a statement or group of statements if the condition is true. Often, we want to execute one set of statements if the condition is true and another set if the condition is false.

In Visual Logic, we can do this by placing flowchart symbols on the "false" side of the If Condition symbol as well as on the "true" side.

We will expand our `OlderThanJava` program to do this. Here, we print a different message if the person is younger than or the same age as Java (in addition to the previous message when the person is older).

In Java, we can use the `if-else` construct.

```
if (<boolean condition>) {
   statement1A;
   statement1B;
   ...
}
else {
```

```
    statement2A;
    statement2B;
    ...
}
```

where <boolean condition> is some Java expression that evaluates to `true` or `false`.

See Example: OlderYoungerThanJava

Another Example, Adding in `JOptionPane` I/O

We will next consider another example of `if` statements in Java, but with the added bonus of using a different mechanism for input and output.

The problem: we wish to write a program that calculates the number of full payments needed for a no-interest loan where we are given a loan amount and desired monthly payment. This number is reported. If additional funds are due after those full payments are made, that is reported as well.

See Example: NoInterestLoan

The comments in that example describe in detail three new items:

- The use of `JOptionPane.showInputDialog` to bring up a dialog box with a message and a text box for input, and returning the text typed into the box as a `String`.
- The use of Java's `Integer.parseInt` method to convert a `String` to an `int`, which is necessitated here because the `JOptionPane.showInputDialog` only returns `String` values.
- The use of Java's `JOptionPane.showMessageDialog` to bring up a dialog box to display some program output.

Nested Conditionals

There is nothing stopping us from putting conditionals inside of conditionals. Consider this decision-making problem of whether it's a good idea to cancel classes tomorrow so we can go skiing.

Suppose we are only willing to go skiing if the temperature will be no higher than 50 and there is at least 6 inches of snow on the ground in the mountains.

We will first build a Visual Logic flowchart for this. We can ask either of the questions (temperature or snow cover) first, and if that response doesn't disqualify the day as a ski day, only then will we ask the other. Let's ask temperature first, then if the temperature is cool enough, ask about the snow cover.

All we really need to understand here is that any flowchart element, including an If Condition, can be placed on one of the branches of the If Condition.

See Example: ShouldWeSki

When we convert this to Java, we will place prompt that reads in and the entire condition that checks for a deep enough snow cover within the "if part" (the part that happens if the temperature is below 50).

Java's `if-else-if` Construct

Our next example is a program that asks for the user's name and hometown, then displays a message that indicates whether the length of (number of characters in) the name is more than, less than, or the same as the length of the town.

Again, we can construct a Visual Logic flowchart. We know most of what we need to do this, the exception being how we can compute the length of a string.

In Visual Logic, we can compute the length of a string (if it's called `s`) with

```
length(s)
```

We can then use the number returned as part of our condition to decide which branch of an If Condition we should take.

Note that there are 3 possible cases: the name is shorter, the town is shorter, or they are the same length. Since an If Condition only has two choices, we will need more than one If Condition.

See Example: NameAndTown

For this conversion to Java, we could do a nested `if` as we did for the previous example. However, there is a variant on the `if-else` construct that allows us to check multiple conditions in a sequence and (optionally) perform an "otherwise" case at the end.

It is sometimes called the "if else-if" construct, and looks like this:

```
if (cond1) {
   // cond1 true stuff
}
else if (cond2) {
   // cond2 true stuff (only can happen if cond1 false)
}
else if (cond3) {
   // cond3 true stuff (only can happen if cond1 and cond2 false)
}
...
else if (condn) {
```

```
    // condn true stuff (only can happen if all previous conds false)
}
else {
    // "otherwise" -- will happen if all previous conds false
}
```

In our program we can see that construct where we first check if the name is shorter. If not, we check if the name is longer. If neither was true, then they must have been equal in length, so the final `else` is executed.

Also notice that we also have a mechanism in Java to compute the length of a `String`.

So far, we have made use of only a fraction of the capabilities of Java's `String` class. All we have done is to declare variables capable of holding `String` references, assign `String` values to them, and use those values in constructing outputs.

There are many methods provided by Java's `String`s and we will see a lot of them in coming weeks. For now, we just need the one that can give us a `String`'s length.

If we have a `String` in a variable `s`, we can compute its length with

```
s.length();
```

With this available to us, we can complete the example program.

Boolean data and boolean expressions

Our discussion of conditional execution needs to include a look at more complex boolean expressions.

The common boolean expression operators are

- arithmetic comparisons: `==` to test for equality, `!=` to test for inequality, and the inequality tests: `<`, `<=`, `>`, and `>=`.
- `&&`, which is the *and* operator. Its result is `true` if both of its operands evaluates to `true`.
- `||`, which is the *or* operator. Its result is `true` if either of its operands evaluates to `true`.
- `!` – which evaluates to the boolean opposite of its only operand.

We will encounter all of these in meaningful examples going forward, but for now, we can see many of them in action in this example.

See Example: BooleanDemo

See the comments therein to see some details.

In particular, note the precedence of these operators: `&&` is evaluated before `||`, much like multiplication is evaluated before addition in an arithmetic expression.

Important note: you need to be very careful that you do specify these operators as `&&` and `||` rather than `&` and `|`. The single-character operators will perform a bitwise and (or) rather than a logical and (or), which is not usually what you want..

Armed with these constructs and a few more we will see in this example, we can now tackle a more complicated problem.

See Example: MassPikeTolls

The comment at the top of the Java program describes the problem.

This is a complex enough decision problem that we should first sketch out a flowchart in Visual Logic.

We will need to use the boolean operators in Visual Logic, which are a bit different:

- arithmetic comparisons: `=` to test for equality, `<>` to test for inequality, and the inequality tests: `<`, `<=`, `>`, and `>=`.
- `AND`, which is the *and* operator. Its result is `true` if both of its operands evaluates to `true`.
- `OR`, which is the *or* operator. Its result is `true` if either of its operands evaluates to `true`.
- `!` – which evaluates to the boolean opposite of its only operand.

We end up with 3 possible outputs:

- There is a full toll if both entry and exit were at an interchange numbered 6 or higher, or if we are driving a truck.
- There is no toll if both entry and exit were at an interchange numbered 6 or lower, and we are not driving a truck.
- There is a toll on only part of the trip (east of interchange 6) if we entered or exited on one side of interchange 6

See the comments throughout the Java program for more information. Note in particular these new Java methods and constructs:

- The use of `System.exit(1)` to terminate the program when an error occurs (in this case, an invalid input was encountered).
- The use of a more complex form of `JOptionPage.showMessageDialog` to more clearly indicate an error message as opposed to an informational message like those we have used previously.

- The use of the `String`'s `equals` method to compare `String` values. We cannot use `==` to compare `String`s for equality in most cases. Java will accept it, but it does not have the meaning we wish it to have in this context. More on this later in the semester.

The `switch` Statement

A common pattern in programming is to have a series of statements of the form:

```
if (x == 0) {
  // do stuff for x == 0
}
else if (x == 1) {
  // do stuff for x == 1
}
else if (x == 2) {
  // do stuff for x == 2
}
...
else if (x == 8) {
  // do stuff for x == 8
}
else {
  // do stuff when x is none of the above
}
```

Let's look at an example where this occurs. Consider a program that tells you which Computer Science faculty member you can find in each of the offices in the Albertus 400 suite.

See Example: CSOfficesIfElse

Java (and many other languages) provide a special construct we can use in situations like this that can be a bit more convenient.

```
switch (x) {
  case 0:
    // do stuff for x == 0
    break;
  case 1:
    // do stuff for x == 1
    break;
  case 2:
    // do stuff for x == 2
    break;
...
```

```
  case 8:
    // do stuff for x == 8
    break;
  default:
    // do stuff when x is none of the above
    break;
}
```

This works only when the comparison if for equality and we are using one of these data types: `char`, `byte`, `short`, or `int`. So far, we have only used `int` variables from among this group. Note that it does not work for `double` or `String` values.

Also note that each `case` is ended by a special statement: `break;`

If we rewrite the example to use a `switch` statement, it would look like this:

See Example: CSOfficesSwitch

If we mistakenly leave out a `break;` statement, Java will "fall through" to the next `case`. Sometimes this is handy and just what we want, but the vast majority of the time, we want a `break;` at the end of case `case`.

One situation where this does come in handy is when we want to do the same thing for multiple cases:

See Example: LittlePrimes

Formatting Output

Our next example has more conditionals, but also shows how we can nicely format output that contains floating point numbers.

See Example: Payroll

The key points to notice from this example:

- The use of named constants for numbers that are unlikely to change from one execution of the program to the next.
- The declaration of variables that will be assigned inside the `if-else` before the `if-else`. If they were defined within the body of the `if` parts and/or `else` part, those variables would exist only *within* those blocks of code.
- The declaration, construction, and use of `DecimalFormat` objects to format our floating-point output. See the text for more examples. The essentials:
  - Like `Scanner` and `JOptionPane`, we need to tell Java if we intend to use a `DecimalFormat` with

```
import java.text.DecimalFormat;
```

- Before we make use of one, we need to declare a variable of type `DecimalFormat` and construct an instance. The parameter we pass to this *constructor* is the number of digits and any other characters we want. There are two examples in this proram, more in the text.
- When we want to print out a floating point value as formatted by one of these `DecimalFormat` objects, we pass the floating point value to the object's `format` method. This returns a `String` representation of that value using the specified format.