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

Topic Notes: Repetition

People find repetition boring. Fortunately, computers don't feel this way. This is fortunate because repetition is the only way we can exploit the full power of a computer. As we discussed in the first class, part of the computer's power comes from the fact that it can follow the instructions stored within its memory rapidly without waiting for a human being to press a button or flip a switch.

In all of the examples we have considered so far, the sequences of instructions performed are quite short. The only time our programs have taken more than a few microseconds to execute are when we have the program wait for input from a `Scanner` or `JOptionPane`. The computer works for a fraction of a second then waits for input or finishes. We could get the computer to do more work by writing methods with thousands or millions of instructions, but this would be painful.

As a simple example (which we won't write), suppose we want to compute all of the perfect squares (*i.e.*, 1, 4, 9, 16, *etc.*) that are less than 100. We could write a program to print them out one by one, each with its own output statement. But what about all perfect squares less than 1,000? Or 1,000,000? Or 1,000,000,000? None of us are signing up to write that program with thousands of printouts.

But we can get the computer to execute thousands or millions of instructions without writing thousands or millions of instructions ourselves: we can have the computer execute the same instructions over and over and over again.

At first, this may seem like a boring and inefficient use of the computer. In fact, when in comes to following instructions, doing the same thing over and over again can be very interesting.

Moreover, we may not always know exactly how many times our statements will need to be executed, so it would be nearly impossible to write the program with a sequence of instructions without some repetition.

`while` Loops in Visual Logic and in Java

We begin in Visual Logic, and we will develop a flowchart for a program that prints perfect squares up to some limit.

We will use a new flowchart construct called a "While Loop". A While Loop is essentially an If Condition that repeatedly executes its "True" branch until the condition becomes False, in which case it continues to the next statement.

In our case, we will read in our upper limit, and start computing the squares of numbers, continuing until the next square is greater than our upper limit.

We will also make use of one more feature of Visual Logic here – console output. In any "Output Expression" element, we have the choice of having the output in a dialog box or in a console window (like we get when we use `System.out.println` in Java). Since we will potentially be outputting many numbers, it will be much easier to use our program if we can avoid having a popup dialog box to click on for each number generated.

The Visual Logic flowchart for this example is available with the Java example below.

Once our flowchart is working, we can convert to Java. As usual, we will have to worry a bit more about syntax in Java.

Java provides the `while` statement, or "while loop", to perform repeated actions. Java includes other looping constructs that we will see later in the semester.

The syntax of a `while` statement is:

```
while (condition)
{
    ...
}
```

As in the `if` statement, the condition used in a while must be some expression that produces a `boolean` value. The statements between the open and closed curly brackets are known as the *body* of the loop.

A common way the while loop is used is as follows:

```
while (condition)
{
    do something
    change some variable so that next time you do
                    something a bit differently
 }
```

The condition controlling the `while` loop will usually involve the variable that's changing. If nothing in the condition changes, then the loop will never terminate. Such a condition is called an *infinite loop*. We avoid this, in general, by ensuring that our loops have a precise stopping condition. While we might be able to look at an algorithm and say "hey, we should stop now", Java will not (and in fact cannot, in general) determine if a loop will not stop.

Armed with this construct, we can convert our flowchart into Java

See Example: PerfectSquares

Other than the `while` statement itself, we see one additional Java construct here:

```
nextNumber++;
```

Since increment and decrement operations on variables are extremely common in programming, the designers of Java (and the designers of C before them), included a shorthand notation for these.

The above has the same effect as if we had written

```
nextNumber = nextNumber + 1;
```

Loops for Error Checking

We will use loops in many contexts, one of which is to allow us to reissue prompts and reread input when an invalid value is entered.

To demonstrate this, we will improve on one of our old examples: the one where we determined whether a trip on the Massachusetts Turnpike was toll free, partially tolled, or fully tolled.

See Example: MassPikeTollsBetter

The changes are all at the start of the program while we input values.

See the comments there for details.

The `do-while` Loop

The `while` loop we saw in the last few examples is called a *pre-test loop*. That is, we check the condition before we enter the first time. This allows a `while` loop to execute its body 0 times if the condition is initially false.

In some circumstances, we want to execute the loop at least once. Such a loop is called a *post-test loop*.

Consider the problem where we have a sequence of numbers to read in, say prices of items at a supermarket checkout, for which we want to keep a running total to report at the end.

In Visual Logic, we can accomplish this by choosing the "Post-test" option in the While Loop component.

We use this to develop a flowchart for the problem.

Notice how the condition test is now at the "bottom" of the loop construct.

Java provides a construct we can use for this purpose as well – the `do-while` loop.

It is basically the same as a `while` loop, except we begin it with the keyword `do`, follow with the body of the loop, and end it with a `while` keyword and condition.

```
do
{
    ...
} while (condition);
```

See Example: Checkout

This example demonstrates the `do-while` construct.

One other new Java construct here is the `+=` assignment operator:

```
total += itemPrice;
```

Much like the `++` we saw recently for the increment operation (and the corresponding `--` operation for decrement), this is a shorthand notation for a common programming task: adding a value to a variable and storing the result back in that variable:

```
total = total + itemPrice;
```

This shorthand exists for all of the standard arithmetic operators: `-=`, `*=`, `/=` and `%=`.

For example, if we wanted to double the value in a variable `x`, we could use the shorthand:

```
x *= 2;
```

You are never going to be required to use these shorthand operators, but they are convenient, and you will need to recognize them in my examples.

Counting Loops

All of the loops we wish to have in our programs can be written using the `while` and `do-while` constructs we have just seen.

However, most programming languages include another construct that is typically used for *counting loops*. Both Visual Logic and Java have such a construct, called a *for loop*.

Let's look at it in Visual Logic first. Our first example will be a straightforward one: calculating the sum of the 10 integers.

When we create a "For Loop" element in Visual Logic, we are presented with a window that lets us control how this counting loop will count. There are four pieces of information needed here:

1. The name of a variable that will contain the values as we count
2. The first value to be given to the variable
3. The last value to be given to the variable
4. The amount by which we change the value each time around the loop (allowing us to count backwards, or by 2's or any number of other variations)

Java's `for` loop looks a bit different, but essentially has all of the same components.

```
for (int number = 1; number <= 10; number++)
{
   // do stuff - but omit number++ at end
}
```

The code in the parentheses consists of 3 parts; it is not just a condition as in `if` or `while` statements. The parts are separated by semicolons. The first part is executed once when we first reach the `for` loop. It is used to declare and initialize the counter. The second part is a condition, just as in `while` statements. It is evaluated before we enter the loop (ı,e it is a pre-test loop) and before each subsequent iteration of the loop. It defines the stopping condition for the loop, comparing the counter to the upper limit. The third part performs an update. It is executed at the *end* of each iteration of the `for` loop, just before testing the condition again. It is used to update the counter.

See Example: Sum1To10

Notice how the for localizes the use of the counter. This has two benefits. First, it simplifies the body of the loop so that it is somewhat easier to understand the body. More importantly, it becomes evident, in one line of code, that this is a counting loop.

Other variations

Many variations are possible and we will see them frequently throughout the remainder of the course. For example, we could *count down* instead of up:

See Example: Countdown

We can see the loop in Visual Logic as well as in Java.

This includes not only a count down loop, but a loop whose starting condition depends on the value in a variable instead of an integer constant. We can use any arithmetic expression for the initialization and any boolean expression for the stopping condition.

If we wanted to count by 2's to add up the even numbers:

See Example: Sum2ToNBy2

We can compute some number of terms of the geometric sum

$$\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + ...$$

If we continue this sum infinitely, the series sums to 1 (can you prove it?).

See Example: GeometricFractionalSum

This example has a straightforward counting loop structure, but has more work to do each time around the loop. Not only do we need to make sure we iterate the proper number of times, we also

need to update the value of the next term to be added each time around.