Computer Science 202
Introduction to Programming
The College of Saint Rose
Fall 2013

# Topic Notes: Repetition

---

## 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 preformed when a mouse event occurs are quite short and then the computer has to wait for us again. The computer works for a fraction of a second then waits. We could get the computer to do more work in response to our mouse events by writing methods with thousands or millions of instructions, but this would be painful.

As a simple example, 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. Think of the scribble program or the Spirograph program. Each time we drag the mouse in these programs, the computer "does the same thing" in the sense that it executes the same instructions — the body of `onMouseDrag`. Each time these instructions are executed, however, the computer actually does something different because the meaning of at least one of the variables referenced in the instructions, `point`, has changed.

Consider this example, where we get some "interesting" behavior through repeating the same instructions without depending on changes in the mouse position.

See Example: RailroadClick

Here, we draw a railroad track, one railroad tie at a time, by clicking the mouse.

Each time the mouse is clicked, the `onMouseClick` method does the same things. It creates a `FilledRect` that looks a bit like a railroad tie and it increases the value associated with the variable `tiePosition`. Because `tiePosition` is increased with each click, the next click

draws its tie a little farther over in the screen. To prevent the program from wasting time by drawing ties no one will ever see, an `if` statement is included that skips the creation of new ties once `tiePosition` gets large enough.

It is painful to have to click repeatedly to get the ties drawn. Instead we would like the computer to continue drawing ties while they are still on the screen. 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
 }
```

Armed with this construct, we can draw all of our railroad ties in the `begin` method.

See Example: Railroad

As in this example, 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 no computer can, in general) determine if a loop will not stop.

Armed with this construct, we return to one of our motivating examples to see a `while` loop in a Java application rather than a graphical applet:

See Example: PerfectSquares

Two examples to develop in class:

Graph paper and random placement of random ovals.

# Loops for Error Checking

To motivate the use of loops for error checking, consider this Java application:

See Example: MassPikeTolls

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

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 main problem with this program is that it simply exits with an error if invalid inputs are presented. We can use loops to reissue prompts and reread input when an invalid value is entered.

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.

Java provides a construct we can use for this purpose – 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.

This example also introduces the declaration, construction, and use of `DecimalFormat` objects to format our floating-point output. We will see more examples later. But 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.

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*.

Our first example will be a straightforward one: calculating the sum of the 10 integers.

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 (or sometimes, a value beyond that)
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 organizes these components in a very particular format:

```
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

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.

---

## The `VisibleImage` object

We take a break from loops for a brief look at a new Objectdraw object type that you're likely to enjoy using.

See Example: Snowman

In the program above, we drag a picture of a snowman around the screen. The picture comes from a "gif" file named `snowman.gif`.

The image is certainly too complex to draw using our ObjectDraw primitives. Fortunately, we can read an image from a file and save it as an object with type `Image`. `Image` is a built-in Java class from the library `java.awt`. Hence you need to make sure that any program using `Image` imports `java.awt.Image` or `java.awt.*`.

The first line of the `begin` method of the `Snowman` class shows how to do this when given a "gif" file (a particular format for holding images on-line):

```
snowManPic = getImage("snowman.gif");
```

where `snowManPic` is an instance variable declared to have type `Image`. Downloading a "gif" file can often be slow, so we usually will want to create an image from the "gif" file at the beginning of a program and save it until we need it. If you download "gif" files in the middle of your program, you may cause a long delay while the computer brings it in from a file on a local disk or fileserver.

While objects of class `Image` can hold a picture, they can't do much else. We would like to create an object that behaves like our other graphics objects (*e.g.*, `FramedRect`) so that it can be displayed and moved around on our canvas.

The class `VisibleImage` from the ObjectDraw library allows you to treat an image roughly as you would a rectangle. In fact, imagine a `VisibleImage` to be a rectangle with a picture embedded in it. You can do most things you can do with a rectangle, except that there's a neat picture on top.

To create a new `VisibleImage`:

```
new VisibleImage( anImage, xLocation, yLocation, canvas);
```

For example, `new VisibleImage(snowManPic, 10, 10, canvas);` would create an object of type `VisibleImage` from the image in `snowManPic` and place it on `canvas` at location (10,10), with size equal to the size of the image it contains.

If you associate a name with your `VisibleImage`, you can manipulate it using some familiar methods:

```
VisibleImage snowMan;
```

And then later:

```
snowMan = new VisibleImage(snowManPic, 10, 10, canvas);

snowMan.setWidth(124);
snowMan.setHeight(144);
```

Our original snow man image is large: 619x718 pixels, but we shrunk him down to a more reasonable size.

What do you think happens if we say:

```
snowMan.setColor(Color.green);
```

Nothing! It's not an error, but nothing is done for you either! Because the picture already has its own colors, it wouldn't make sense to change it to a solid color. Similarly, the value returned by `snowMan.getColor()` is always `Color.black`, no matter what colors are in the image!

The rest of the code for the `Snowman` class is just a variation on the earlier programs that allowed us to drag around squares and T-shirts.

Another example of the use of a `VisibleImage` that also uses a loop, demonstrating that we only need to use `getImage` once, and can then create as many `VisibleImage`s as we want with that one `Image`.

See Example: SnowyNight

See the comment near the bottom of the `begin` method for more about how we ensure that we can see all of the snowflakes while allowing them to be partially obscured on either side or the top of the canvas.

---

# More Advanced Loops

Now that we have seen how important loops are, and have practiced with them in so many contexts, we step back and discuss more complex loops.

See Example: Knitting

Here, each time the mouse is clicked, we knit a scarf.

If you look carefully at the pictures generated, you will see that the scarf is formed by overlapping circles. It is easiest to develop this by first writing code to generate a row, then expand it to generate the correct number of rows, in the correct positions.

To draw a row, we will have a `while` loop. Each time through the loop we increase the value of `x` position as well as bump up our counter of the number of columns drawn so far, `numCols`.

That wasn't too hard, but now we'd like to create successive rows. Each time we start a new row, there are a number of things that we will need to take care of:

1. We need to reset the value of `x` so that we start drawing at the beginning of the row rather than where we left off.
2. We to increase the value of `y` so that rows won't be drawn on top of each other.
3. We need to reset `numCols` back to 0 so that it will keep the correct count when we restart drawing a row.
4. We need to bump up `numRows` each time through.

Now all we need to do is to repeatedly execute the code for drawing a row by placing it inside an enclosing `while` loop. This is our first example of a *nested loop* structure: a loop that executes within a loop.

There is nothing mysterious about a nested loop. The inner loop is simply part of what the outer loop does over and over.