Math Modeling in Java: An S-I Compartment Model

Basic Concepts

What is a compartment model?

A compartment model is one in which a population is modeled by treating its members as if they are separated (logically, if not physically) into distinct compartments, with some number moving between compartments in each time step. The number moving from one compartment to another in any given time step is generally a function of the number in one or both compartments at the start of that time step, and may also be affected by the numbers in other compartments and other factors. (Population members may also enter or leave the model entirely in each time step — e.g. through birth and death processes.)

This approach is often useful for modeling the spread of infectious diseases: we can define compartments for the members of the population who are susceptible, infected, immune, etc., and observe (analytically, or through simulation) the dynamics of these compartments over time.

The S-I Model

In the simplest compartment model of infectious disease, we divide the population into two compartments: The Infected (I) compartment includes all members of the population who are currently infected, and the Susceptible (S) compartment includes all members of the population who are not currently infected, and who may become infected through contact with those in the Infected compartment. In this simple model, we do not take into account variations in susceptibility among segments of the population (due to age, general health, etc.), and we assume that there is no recovery (i.e. infection is permanent). Also, in this model, we won't be considering death or birth processes — i.e. the population remains constant.

In our model, we will use the following basic variables:

P = total population size

I_t = number of individuals in the Infected compartment at time t

I_0 = number of individuals in the Infected compartment at time $t = 0$

$S_t = P - I_t$ = number of individuals in the Susceptible compartment at time t

$S_0 = P - I_0$ = number of individuals in the Susceptible compartment at time $t = 0$

Δt = length of the time step

In this model, the number of new infections (those moving from the Susceptible compartment to the Infected compartment) in each time step is considered to be approximately equal to the number of number of opportunities for infection (i.e. significant contacts between infected and susceptible individuals), multiplied by the probability of infection, given the opportunity for infection. In some cases, we might speak of the "infection force", which is the factor that, when multiplied by the number of infected, the number of susceptible, and the length of the time step, gives the number of new infections:

f = infection force

$I_{t + \Delta t} \approx I_t + fI_tS_t\Delta t$

(Note: even for a very contagious infection, in a densely populated area, the infection force is generally a very small fractional value.)

The number of individuals are added to the Infected compartment in each time step have to come from somewhere — namely, the Susceptible compartment. Thus, we can also write:

$S_{t + \Delta t} \approx S_t - fI_tS_t\Delta t$

Questions for discussion:

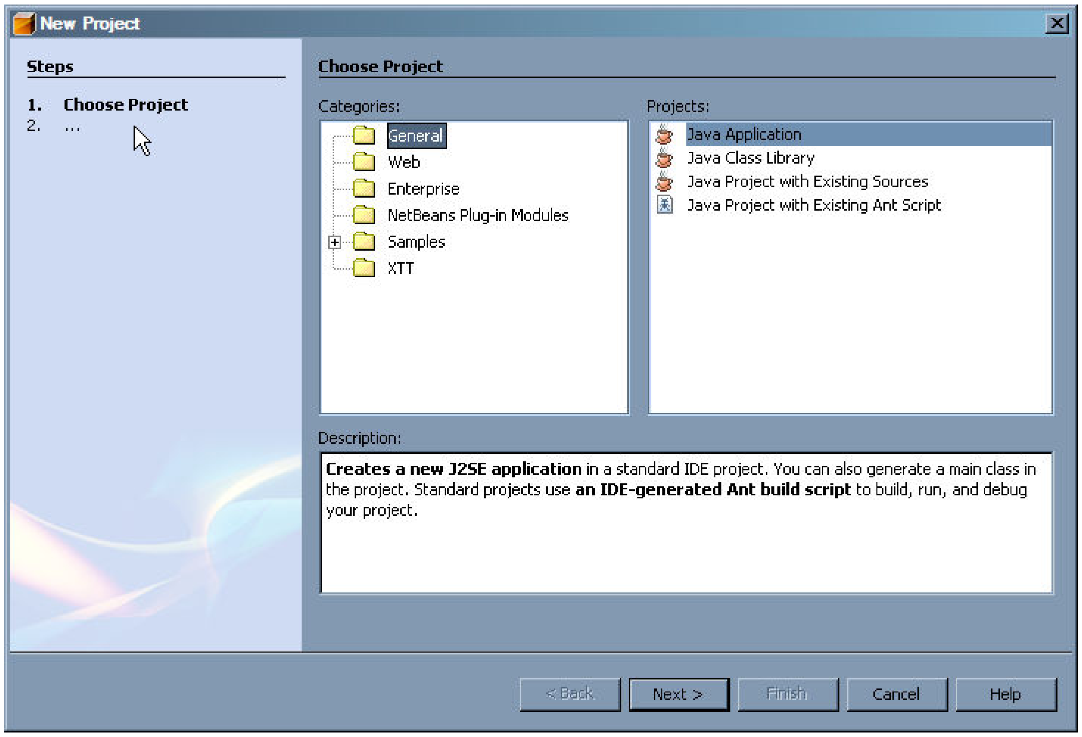
- *What factors might contribute to the infection force?*
- *This model assumes "uniform mixing" — in other words, all individuals, whether infected or susceptible, meet (and may infect or be infected by) others at the same rate. Do you think a compartment model could be adapted to allow for such concepts as quarantine (where symptomatic infected individuals are not allowed to come in contact with susceptible individuals), or at least decreased mobility of infected individuals who are significantly weakened?*

Implementing the S-I Model in Java

Exercise #1: Creating a Java application project in NetBeans

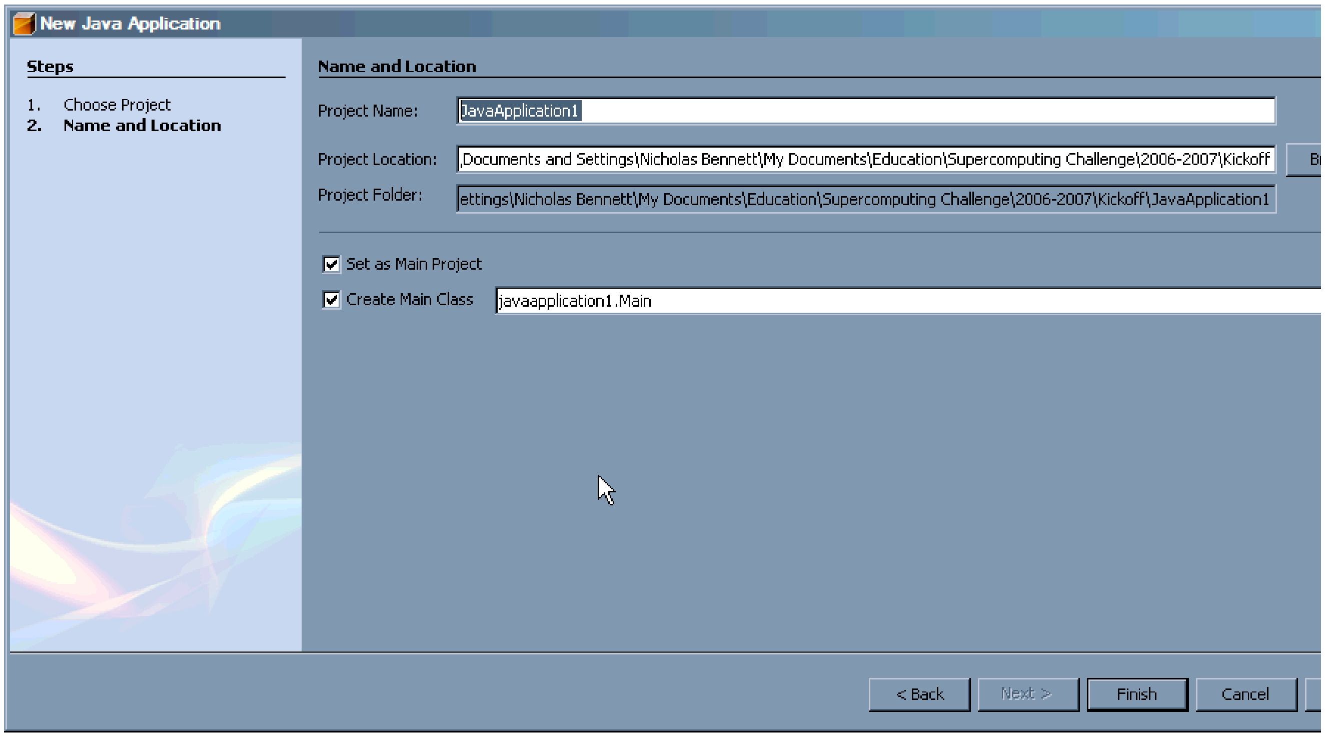
During the rest of the session, we'll be building an S-I model in Java. As a first step, let's create a NetBeans project, in which we will build our program.

In NetBeans, select **New Project…** from the **File** menu, to open the New Project wizard:



Creates a new J2SE application in a standard IDE project. You can also generate a main class in the project. Standard projects use **an IDE-generated Ant build script** to build, run, and debug your project.

Select **General** from the list of categories, and **Java Application** from the list of projects. Then, press the **Next** button, to go to the following screen:



Project Name: JavaApplication1

Project Location: ,Documents and Settings\Nicholas Bennett\My Documents\Education\Supercomputing Challenge\2006-2007\Kickoff

Project Folder: ettings\Nicholas Bennett\My Documents\Education\Supercomputing Challenge\2006-2007\Kickoff\JavaApplication1

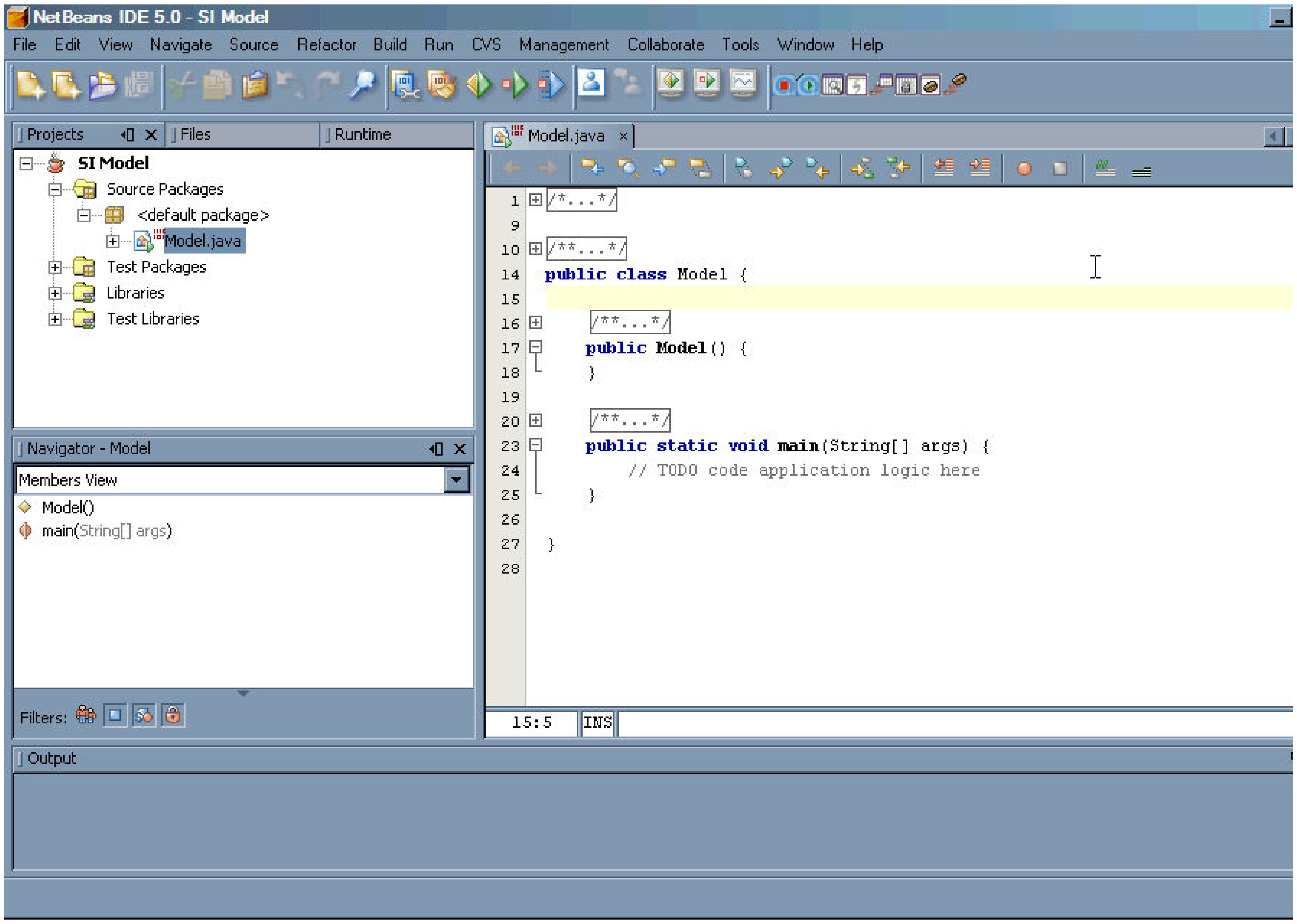
☑ Set as Main Project

☑ Create Main Class javaapplication1.Main

Follow these steps to complete the **New Java Application** screen:

1. For **Project Name**, type "SI Model" (don't worry about spelling or case for this one — the name you type is the name NetBeans will use when displaying the project in its workspace, but is has little to do with the compilation and execution of your program).
2. The value provided by default in **Project Location** is probably fine; but if you want to be able to copy your work to a floppy or USB drive, it might be easier if you specify a directory (by clicking the **Browse…** button, and navigating from there) that will be easily found later (the Desktop is pretty good for this purpose). In any event, you will want to write this value down for later use.
3. Make sure that **Set as Main Project** is checked.
4. Make sure that **Create Main Class** is checked, and type "Model" for the class name (this time, spelling and case are *very* important).
5. Click the **Finish** button.

Now, you should see a new NetBeans project, with the source file for the class you specified (in step #4) open and ready for editing. (Don't worry if your NetBeans screen isn't laid out in exactly the same way: as long as you see your new project on the left, and the `Model.java` file open on the right, you are doing fine.)



```
/*...*/

/**...*/
public class Model {

    /**...*/
    public Model() {
    }

    /**...*/
    public static void main(String[] args) {
        // TODO code application logic here
    }

}
```

We could have created this class by hand, without much difficulty. However, NetBeans, like virtually all Java development environments, makes some tasks a little easier (and a few tasks a *lot* easier), so we'll take advantage of it.

Here are some things to notice about this code:

- The Java class for our program is `Model`; similarly, the file in which we define the class is `Model.java`. It is *very* important that these two match — not just in name (except for the `.java` file extension), but also in case; otherwise, the Java "executive" (the software responsible for loading and running programs written in Java) will not be able to locate our program, to run it.
- The `Model` class contains a method called `main`. Every Java application class (i.e. the main class of a standalone Java program) must contain this method, declared in the same way (more about this later).

- The `Model` class contains a constructor; this is the code that begins with `public Model()`. A constructor is used for creating individual instances based on the class definition. Even though constructors are a fundamental feature of Java (and most object-oriented languages), we won't be covering them today; for now, we can ignore this constructor.
- As it stands, the code compiles and runs cleanly (even though it doesn't really do anything when it runs); we can check this by selecting **Run Main Project** from the **Run** menu (if the code has been modified, this will also cause the code to be recompiled before it is run). In general, it is a good practice to write code in such a way that you can compile and test often — even before the code does everything it is supposed to do. Similarly, when changing existing code, make and test small changes — rather than making a large number of changes, and only then trying to compile and test, and finding out that there are several errors in the code.

Now that we've got a place to write Java code, we need to think about what the program will actually do.

Turning a mathematical model into a computational model

Previously, we defined a number of model parameters (e.g. initial population, initial susceptible & infected, infection force, time step duration), and variables (number of susceptible & infected, over time). We also wrote equations describing how the variables change in each time step. The work of implementing a mathematical model, in general, consists of translating such definitions and equations into program code. In our case, we will translate them into Java code, and add them to class file we just created.

Of course, we can't simply write the formulas above in Java terms, and expect our program to do what we want it to do. Instead, we will start our model at a point in simulated time, and keep applying the formulas repetitively (to update the current time, the number of susceptible, and the number of infected), until we reach a stopping point.

Questions for discussion:

- *Under what conditions should our S-I model stop execution?*
- *How can we execute program instructions repetitively, as many times as needed, if we don't know ahead of time exactly how many repetitions we will need?*

Exercise #2: Creating Java variables for our model

Now, we will create program variables for the variables and parameters we defined in our mathematical model. We have a lot of leeway in how we can do this, but there are a few guidelines we should follow:

- Variable names cannot contain spaces; this is not a guideline, but a rule that the compiler enforces.
- Variables should be given descriptive names; there is little value in being cryptic, or in using very short names in order to save typing, if it makes our program hard to read and understand.
- By convention, Java variables are given names that begin with lower-case letters; when a variable name includes multiple words, each successive word is capitalized.

For this exercise, we will define all of our variables at the class level — i.e. they will be defined as members of the class, and not as local variables within a method. Also, since we will not be dealing with definition of additional classes, nor even with the instance constructor of the class we have created, we will define all of our variables (and the methods we will define later) as `static` variables; this makes them accessible from `static` methods (such as the `main` method already in our code).

To define a variable in a class, we write the definition inside the brackets enclosing the close, but outside any of the class methods. Typically (but not necessarily) we write them near the top of the class, before the definition of the constructor and any methods.

A variable definition begins with an optional accessibility modifier (which controls if & how the variable can be accessed from outside the class); if the variable is to be `static`, this modifier comes next; there are a few different modifiers which might come next (e.g. `final`, `volatile`), but we won't deal with these for now; next comes the variable type, which can be an intrinsic type (a simple type, which only holds data, and does not define any of its own behavior), or a class type (which can define methods that operate on the data in the variable); then comes the name of the variable; finally, and optionally, there can be an initial assignment statement. (Also, don't forget: all simple Java statements end with a semicolon.)

As an example, if we want to define an integer variable called "x", which will not be accessible outside the class, and which will have static scope, and which will have an initial value of 1, we could write the following:

Example:

```
    private static int x = 1;
```

Now, add variable definitions for the model variables and parameters to the `Model` class. At the same time, try to make some reasonable initial value assignments, where doing so makes sense to you. Here is one way this might look, when complete (the comments have been omitted from the code):

Example:

```
public class Model {

    private static int population = 1000000;
    private static int initialInfected = 10;
    private static double timeStep = 1;
    private static double infectionForce = 0.0000001;
    private static int currentInfected = initialInfected;
    private static double currentTime = 0;

    public Model() {
    }

    public static void main(String[] args) {
    }

}
```

After modifying the code, use the **File/Save** menu option to save your changes. Then, use the **Build/Build Main Project** menu option to compile your program. If any compiler errors are reported, ask the instructor for help.

Questions for discussion:

- *Note that there are a number of parameters and variables defined for the mathematical model, which do not appear in the code. Can these values be calculated, as needed, from the variables that are included?*
- *From the code above, is it clear what each variable will be used for? If you used different variable names, are they more or less clear to you?*

Exercise #3: Creating a method to update the model variables

Now, we will write a method that will use the formula given above to update the number of infected. For now, we won't worry about how we will repeat this calculation at every time step; instead, we will just focus on how the calculation will be performed in any single time step.

In writing this method, we will take advantage of one of a special set of operators, available in C, C++, Java, JavaScript, and a few other languages: the "compound assignment" operators. These operators allow us to replace a statement like:

Example:

```
x = x + 10;
```

with an equivalent, shorter statement:

Example:

```
x += 10;
```

In both cases, the result is the same: the value obtained by adding 10 to the current value of `x` is placed into `x`; the effect is to increase the value of `x` by 10.

In our case, it is the number of infected that will be updated, but the technique is identical.

Methods are defined in much the same way as variables: first, an optional accessibility modifier may be specified; next, `static` may be used to specify class scope, vs. instance scope; then, the type of value returned by the method is specified (if the method is not intended to return a value, the special type specifier `void` is used); the name of the method then follows. After that, method declarations become very different from variable declarations: a list of parameters (values passed to the method, to give it any additional information it needs) is specified, within parentheses (if no parameters are specified, empty parentheses are used); the statements making up the body of the method are specified, inside of curly braces.

Now, create a new `static` method in the `Model` class. It will only need the variables we have already to defined, so the method definition will not include parameters.

Inside the method body (i.e. inside the curly braces following the method name and the parenthesized list of parameters), write the statement that updates the number of infected, based on the formula given above. Also, include a statement to update the current time.

Again, there are a few guidelines you should consider following:

- Method names cannot contain spaces; this is not a guideline, but a rule that the compiler enforces.
- Methods should be given descriptive names; it is a common (and generally valuable) practice to start a method name with a verb (i.e. methods do something; use the key verb for what a method does in its name).
- By convention, method names are given names that begin with lower-case letters, with an upper-case letter used for the first letter in each successive word in a multi-word method name.

When complete, you should have something like this (note that, by convention, class methods are generally written to follow the class constructors):

Example:

```
    public class Model {

        private static int population = 1000000;
        private static int initialInfected = 10;
        private static double timeStep = 1;
        private static double infectionForce = 0.0000001;
        private static int currentInfected = initialInfected;
        private static double currentTime = 0;

        public Model() {
        }

        private static void updateModel() {
            currentInfected += infectionForce * currentInfected * (population - currentInfected) * timeStep;
            currentTime += timeStep;
        }

        public static void main(String[] args) {
        }

    }
```

Once again, save and compile your program — correcting errors and repeating as necessary.

Exercise #4: Completing the `main` method

We're almost done with our work on our (very basic) S-I model. However, we're left with the questions we had before: How will we repeatedly update the number of infected? Just as important, how will we know when to stop? And we have to answer another question, that we didn't even ask before: How we will we show the results of our model?

In this case, knowing when to stop is actually not so difficult: in our model, once someone is infected, they stay that way; thus, we can simply run our model until everyone is infected. One complication with this is that our model, as written so far, allows for "fractional individuals": we can (and will) end up with a non-integral values for the number of infected, and the number of susceptible, at various moments in time. For now, let's assume that when the number of susceptible drops to below 0.5, everyone in the population is infected.

As to the mechanics of iteration in Java, we actually have a number of choices. Most important among them are the `for` and `while` statements. Actually, these two are virtually similar, and any use of the `for` statement can be rewritten with a `while` statement; however, the former is more clear and concise for many purposes — particularly when we are iterating across the elements of an array or collection. For our purposes, we will use the `while` statement, which has this general form:

Example:

```
    while (repeat-condition) {
        statements
    }
```

We'll replace the italicized portions with the condition and statements that are relevant to our model. The *repeat-condition* is an expression that has a `boolean` (true or false) value. It is evaluated before the *statements* in the curly braces are repeated: if it evaluates to `true`, the statements are executed, and the *repeat-condition* is evaluated again, etc.; if/when the *repeat-condition* evaluates to false, the *statements* are not executed, and the program continues after the closing curly brace of the `while` statement.

So our task should be coming into clearer view: we must write a `while` statement, which will loop repeatedly until all of the population is infected; each time it repeats, the statements within the loop must update the number of infected and the current time (of course, to do that part, we simply need to call the `updateModel` method).

Let's not forget, however, that we should also generate some kind of output as the model is running, to show the population dynamics. Java libraries such as Repast give us easy access to graphs for presenting simulation model output (not to mention control panels for experimenting with different values of model parameters). There are also any number of general purpose graph plotting libraries, that give more flexibility, at the price of additional development effort. Finally, of course, we could write all of the code to draw our own graphs in Java, in exactly the desired style; that would require that we devote an even larger portion of the programming effort to the presentation of output, possibly at the expense of working on the model itself. Unfortunately, all of these options fall outside the scope of what we can accomplish today, in the time allotted.

For now, we will fall back on the lowest-common-denominator approach: text output. After each time step, we will use the text output functions of Java's `PrintStream` class to display the current time, and the current number infected. As it turns out, the Java executive creates an instance of the `PrintStream` class when launching a Java program, and makes it available as `System.out` (in other words, the `System` class has a `static` member called `out`, which is a `PrintStream` instance); we will use this "standard output" `PrintStream` for our output.

There are three methods of the `PrintStream` class which are most commonly used for this kind of output: the `print` method, which can write values of many different types to the output; the `println` method, which does exactly the same thing as `print`, but appends a newline character to the output (so that the next data written to the output is sent to the next line); the `printf` method, which writes a formatted string of characters to the output, with placeholder tokens in the string being replaced at runtime by expression values. Here are some examples of the invocation of these methods, using the `PrintStream` instance referred to by `System.out`:

Example:

```
// Writes out the values of x and y on the same line.
System.out.print(x);
System.out.print(y);

// Writes out the values of x and y on different lines.
System.out.println(x);
System.out.print(y);

// Writes out the values of x and y on the same line, embedded in a line of text.
System.out.printf("X = %s; Y = %s", x, y);
```

Now, fill in the `main` method, to include a `while` statement that repeatedly calls the `updateModel` method, and then uses `System.out.print`, `System.out.println`, and/or `System.out.printf` method to write the current time and current number infected to standard output. (Don't forget to write the condition of the `while` statement so that it evaluates to `true` as long as the number of susceptible is greater than or equal to 0.5, and `false` otherwise.)

Here is one way to complete the `main` method:

Example:

```
public class Model {

    private static int population = 1000000;
    private static int initialInfected = 10;
    private static double timeStep = 1;
    private static double infectionForce = 0.0000001;
    private static int currentInfected = initialInfected;
    private static double currentTime = 0;

    public Model() {
    }

    private static void updateModel() {
        currentInfected += infectionForce * currentInfected * (population - currentInfected) * timeStep;
        currentTime += timeStep;
    }

    public static void main(String[] args) {
        while (population - currentInfected >= 0.5) {
            updateModel();
            System.out.printf("Time: %f; Infected: %f\n", currentTime, currentInfected);
        }
    }

}
```

(The `\n` at the end of the format string in the `System.out.printf` method call denotes a newline character; this is a handy way of forcing subsequent output to the next line, without using the `System.out.println` method.)

Save and compile your program — correcting errors and repeating as necessary.

Exercise 5: Running the Model

Now that your program compiles successfully, it's time to try it out. Use the **Run/Run Main Project** menu option to execute your program; the output of your program will appear in the NetBeans Output pane (which is probably located below the code editing pane).

If your program seems to lock up, or run without stopping, it is likely that the condition in your `while` statement is written in such a way that an "infinite loop" results.

If the current time and the current number of infected individuals aren't changing, check to make sure that your `updateModel` method is written correctly, and that it is being called from the body of the `while` statement in your `main` method.

If the current time is changing, but not the current number of infected individuals, check to make sure that you have assigned an initial value (either in the variable declaration, or in the `main` method) to the current infected.

(The NetBeans environment includes a fairly robust debugger, which can be used to set breakpoint and watches in the code, and to step through the code line by line. An explanation of the use of this debugger is outside the scope of this session, but the NetBeans documentation for its use is reasonably complete.)

Questions for discussion:

- *How might we modify the model and the program, to allow those in the Infected compartment to recover, and return to the Susceptible compartment?*
- *How might we modify the model and the program, to introduce the concept of immunity?*
- *In the current version, the model parameter values are specified explicitly in the program code. How could we change this, to allow new values to be specified and used, without requiring recompilation of the code?*