MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Spring Semester, 1997

Problem Set 7

JAVA!!

Issued: Tuesday, April 2

Due: In recitation on Wednesday, April 9 for recitations held at 10 and 11.;
and on Friday, April 11 for recitations held at 9, 12, 1 and 2.

Tutorial Preparation: For the week of April 7.

Reading Assignment: *Learn Java Now*, Chapters 1-7, 11, 12

1. Tutorial exercises

For this problem set there are not explicit tutorial exercises. Rather, you should come to tutorial prepared to work examples of simple Java coding.

Note that we have outlined what seems like a lot of reading (though also notice that this reading is much faster than say `SICP`). Thus, we suggest that you do as much of the background reading as you can before Tutorial.

2. Working together

*****WARNING***: This is an experimental problem set. As we noted in lecture, we are experimenting with the use of Java in 6.001. While we believe that this Problem Set is (mostly) bug free, we have had limited interaction with this version of Java, and can't guarantee that everything will run smoothly. Please bear with us as we work out the kinks in this system.**

Because this is an experiment, we are only going to use one platform, in this case the Microsoft J++ environment, running on a PC. We understand that not everyone in the class has the ability to run this system. Thus, in section on April 3, we will be clustering people into groups that will work together on this problem. Each group will have at least one person with a machine capable of running the software – PC with 486 or higher, running Windows 95 or Windows NT version 4.0 or higher, 8 MB of memory if Windows 95, 16 MB if Windows NT, 20 MB hard disk, CD-ROM drive.

3. Laboratory Assignment: Exploring Java

So why are we giving you a problem set on Java, with only 1 week's worth of lectures and recitations? Our goals in this problem set are several:

- we want you to see another computer language, especially to see some of the tradeoffs one makes in designing languages – e.g., the role of types in the language, the fundamental units for organizing modular systems, and so on,
- we want you to learn enough of the basics of another language so that when we turn shortly to the topic of building evaluators for languages, you will have a chance to see how one language can be used as a substrate upon which to build an interpreter for another language.
- you may find parts of this problem set frustrating in that we don't necessarily have enough time to give you a thorough and fundamental understanding of all the details of Java, Microsofts Visual J++, etc. In part this is deliberate. Designers of computer systems must generally have such a deep understanding. People who want to use these systems to design their own custom system, on the other hand, often must be able to work by assuming abstractions for some components, and then using those abstractions without seeing what is "under the hood". Such is the case here. We expect you to be able to use the infrastructure provided by Microsoft's system to accomplish your task, even if you have not been directly exposed to all of that system. Thus, you should use the Help feature to lookup things you need, you should explore the options provided by Microsoft's user interface and so on, to gain the knowledge you need. Indeed, any practicing engineer in the real world does this on a regular basis, and you should gain some experience in working this way.

Problem 1: The first thing you need to do is get the Microsoft J++ system loaded onto your PC. To do this, simply follow the instructions that appear after you load the CD into your system. We believe this should be straightfoward, especially if you are running NT. However, some of the staff experienced some difficulties in installing Visual J++ on a 486 running Win95 with CD-ROM. We suspect that if you have a machine that was upgraded from Win 3.1 to Win 95, you might have this problem (although we think that machines that shipped with Win95 will be okay).

The bug was that the machine was using a "legacy" CD ROM driver, and thus the CD-ROM did not support "auto-play". In this case, the fix was to "manually" install the right Win95 driver (Win95 comes with an extensive collection of drivers for legacy equipment). This is fairly straightforward, by using the "install new hardware" facility of the "control panel", which is accessible from the "settings" menu item of the start button.

Problem 2: To get started with Java, let's do the standard simple thing – create a program to say "Hello, World". Initially, let's do this as an application, that is, as Java code intended to run locally on your machine. To do this, you can simply follow the description on pages 3 and 4 of the **Learn Java Now** book. However, since there is some discrepancy between the book and the latest version of the system released by MicroSoft, a better solution is to use the example provided by the system.

Thus, open up the MS Development Studio application. Note that the left hand pane of the window includes a tree of information files, called **InfoView**. You can open the **Read This** book (by clicking on the + sign next to it). Go to the **Simple Java Application** page, open it, and follow the instructions. This should enable you create a simple Java program to print out a greeting into a window.

This problem is mostly intended to get you used to the interface that you are going to use throughout the rest of the problem set. Also note that by writing an application, we are basically just treating Java as a different language.

Turn in a printout of your code.

Problem 3: The previous problem basically resembles the kinds of things we have been doing in Scheme, but simply in another language. One of the main selling points of Java, however, is that it can be used to create platform independent code that can be run over the Web.

Thus, if you use VisualJ++ to compile your program, it creates instructions not for a specific machine, but for a virtual machine that doesn't really exist. By doing so, we can then provide this virtual code to a Java interpreter that is written explicitly to interpret these virtual instructions for a specific machine. When you execute applets over the Web, the browser takes care of interpreting the virtual code. This means that someone else can concentrate on building one interpreter for each platform, which only needs to be done once. Then, users can simply program in one language, Java, independent of underlying machine type and the interpreter will automatically convert the virtual code.

Thus, let's go back and redo our "Hello, World!" example, but now as an applet. Chapter 12 of the Java book details how to do this, but again probably the easiest way is to use the InfoView component of J++. Now, under **Read This** open up the **Simple Java Applet** folder and follow the instructions.

Note how J++ provides a very thorough template for you. It already includes standard applet methods, such as `init, start, paint, ...`. This is exactly the advantage of using such a toolkit, as many of the bookkeeping details are handled automatically for you.

Turn in a copy of your code.

We have now seen some simple examples of building applications and applets in Java. A key advantage of Java is its ability to support Web applets, so that programs can be run on remote machines, largely independent of the type of machine, or its operating system. A second strength of Java is that it supports object oriented programming, in a manner somewhat different from Scheme.

In the rest of this problem set, we are going to explore a simple system of Java objects that lets us examine tradeoffs in how languages support creation and manipulation of object classes.

Our system is going consist of simple geometric objects in a two dimensional world. The simplest such objects are stationary "things", which have a position in the world, but not much else. Below is a definition of this class:

```
public class Thing
        {// internal parameters to hold state
         // idea is that each Thing has a position in the 2D plane

        private double m_dXPos, m_dYPos;

        //      methods for creating an instance
        //      this is the default constructor
        Thing ()
        { this(0, 0); // invoke more general constructor
        }

        // this is the more general constructor that takes input arguments
        Thing (double dxpos, double dypos)
        { // setup state variables
                m_dXPos = dxpos;
                m_dYPos = dypos;
        }

        // methods for allow other methods to access values of variables
        public double XPos ()
        {return m_dXPos;}

        public double YPos ()
        {return m_dYPos;}

        // methods to update parameters
        public void ChangeXPos (double delta)
        { m_dXPos = m_dXPos + delta;}

        public void ChangeYPos (double delta)
        { m_dYPos = m_dYPos + delta;}
}
```

You can load this code into your system by downloading `Thing.java` from the 6.001 Web Page. Alternatively, you can use the J++ Applet Wizard to create a new class. Hold off on either of these for a moment though, because we want to build an applet to handle these things.

Problem 4 Thus, just as you did for the "Hello, World!" example, use the J++ Applet Wizard to create a new project, called `Test`. This will set up a template for this new applet. We are going to add a few details to this template, as shown below, where we have marked the things to be added. You can, if you wish, simply open your own new project, then type in these changes. Alternatively, you can download the `Test.java` file from the 6.001 Web site and install it directly into J++ Applet Wizard.

```
//****************************************************************************
// Test.java:   Applet
//
//****************************************************************************
import java.applet.*;
import java.awt.*;

//============================================================================
// Main Class for applet Test
//
//============================================================================
//
// note that by extending the Applet class we get all the base
// behaviors working within the Web

public class Test extends Applet
{
        //********************************************************************
        // HERE IS ONE OF THE ADDITIONS

        // set up size of simulation

        int NumberOfThings = 5;

        // create an array of things

        Thing[] thingArray = new Thing[NumberOfThings];

        // some constants for offsetting the display

        int xDispOff = 100;
        int yDispOff = 100;

        // setting for time sampling

        double dTimeIncrement = 1.0;

        // ADDITION TO HERE
        //********************************************************************

        // Test Class Constructor
        //--------------------------------------------------------------------------
        public Test()
        {
                // for now, we are not doing anything special when we
```

```
                // construct this class

        }

        // APPLET INFO SUPPORT:
        //              The getAppletInfo() method returns a string describing the applet's
        // author, copyright date, or miscellaneous information.
    //--------------------------------------------------------------------------
        public String getAppletInfo()
        {
                return "Name: Test\r\n" +
                       "Author: Eric Grimson\r\n" +
                       "Created with Microsoft Visual J++ Version 1.0";
        }


        // The init() method is called by the AWT when an applet is first loaded or
        // reloaded.  Override this method to perform whatever initialization your
        // applet needs, such as initializing data structures, loading images or
        // fonts, creating frame windows, setting the layout manager, or adding UI
        // components.
    //--------------------------------------------------------------------------
        public void init()
        {
                resize(320, 240);

                // TODO: Place additional initialization code here
        }

        // Place additional applet clean up code here.  destroy() is called when
        // when you applet is terminating and being unloaded.
        //-------------------------------------------------------------------------
        public void destroy()
        {
                // TODO: Place applet cleanup code here
        }

        // Test Paint Handler
        //--------------------------------------------------------------------------
        public void paint(Graphics g)
        {
          //*********************************************************************
          // HERE IS A SECOND ADDITION

          // put up an announcement of what we are doing
```

```
        g.drawString("Painting things onto image", 10, 20);

        for (int i = 0; i < NumberOfThings; i++)
          { // for each of the things stored in our array,
            // get their position, and draw them on the screen
          Thing t = thingArray[i];     // get each thing
          double dX = t.XPos();  // note how we reference the
                                 // thing's method
          double dY = t.YPos();

          g.setColor( Color.red);

          g.fillOval( (int) dX + xDispOff, (int) dY + yDispOff, 3, 3);

        }
  // ADDITION TO HERE
  //*********************************************************************
}

//              The start() method is called when the page containing the applet
// first appears on the screen. The AppletWizard's initial implementation
// of this method starts execution of the applet's thread.
//--------------------------------------------------------------------------
public void start()
{
 //*********************************************************************
 // HERE IS A THIRD ADDITION

        // need to cycle over objects, setting up at random

        for (int i = 0; i < NumberOfThings; i++)
        { thingArray[i] = new Thing(startValue(0, 100), startValue(0, 100));
        }
 // ADDITION TO HERE
 //*********************************************************************
}

//              The stop() method is called when the page containing the applet is
// no longer on the screen. The AppletWizard's initial implementation of
// this method stops execution of the applet's thread.
//--------------------------------------------------------------------------
public void stop()
{
}
```

```
        //*********************************************************************
        // AND HERE IS A FOURTH ADDITION

        public double startValue(int low, int high)
        { double test = (low + ((high - low) *  Math.random()));
        return test;
        }
        // ADDITION TO HERE
        //*********************************************************************
}
```

Note, by the way, that you can use the Help command of J++ to check on details of methods and classes provided in Java. For example, check out the `graphics` package to see what kinds of drawing capabilities are provided.

Note the use of explicit typing throughout the code – we must tell Java what kind of thing is returned by a method, as well as declaring the kinds of arguments methods take, and as well as declaring the kinds of internal variables used by methods. Also note that if we want to explicitly change the type of a variable, e.g. convert a floating point number to an integer, we must do so using a `cast`, e.g. the use if `(int)` in the `paint` method to convert a floating point number into an integer.

Now, either start your own `Test` project, and type in the changes, or download our version into your system. Similar, either insert a new `Thing` class into the project (using the Insert command), and type in the details for Things above, or download our version of Things into your project.

To test out this simple system, you must `Compile` each file (by first switching to that window, then executing Compile). Then you must `Build` the applet (use the mouse menu to do this), and finally you can `Execute` this project. This should bring up a web browser that then displays your stationary Things in a window.

Problem 5 So far, we have the basis for a simple system. Let's get some practice in Java by making some minor changes. First, our class of `Things` all have the same size. Let's change this. First, modify the definition of `Thing` to allow for a `size` variable. This means you'll need to add a new internal state variable, as well as changing the constructors. Also, you should add a method for returning the value of the `size` parameter, and a method for changing the `size` parameter.

Once you have made these changes and Compiled the new version of the Thing file, let's go back to our `Test` class. Change the calls to the `Thing` constructors in the `start` method of Test to allow for variable sizes. Use `startValue` with some parameters to select a range of sizes. Note, by the way, that one can easily find the right code to change by using the Class View tree structure of classes (see Page 193 of the Java book). Also be sure to adjust the `paint` method to get the size of each `Thing` before drawing it.

Compile, Build and Execute your new version of the code.

Turn in a copy of your code changes.

Problem 6 This is still a pretty boring system. So let's take advantage of the object oriented nature of Java to build a new kind of class. This is a `MoveableThing` which will extend the class of `Things`. The idea is that a moveable thing inherits the properties of a thing (e.g. a position and a size), but it also has additional properties of its own, in particular an x and y velocity.

Install a new class of `MoveableThing` which extends `Thing` into your project. Add appropriate state variables for the velocity components, as well as adding constructors to make instances of a `MoveableThing`. Be sure to inherit the properties of the base class (check out the use of `super` for example). As with our other class, create methods to return the values of the internal parameters and methods to change the values of those parameters. Finally, add a method called `UpdateState`, which takes a single argument, namely a time increment. This method should change the position of the object by moving $v_x * \Delta t$ in the x direction and $v_y * \Delta t$ in the y direction, where Δt is the time increment, and v_x and v_y are the x and y velocities of the object.

Once your new class file properly compiles, we can turn to installing these new objects into our project. Go back to `Test` and make the following changes:

- add an array of MoveableThings to the structure,
- change the `start` method to install a set of MoveableThings at random,
- change the `paint` method to draw the MoveableThings onto the window – use a new color to distinguish these objects from static ones.

Check out your changes by Executing this new version.

Problem 7 Now, we want to have some way to get the moveable things to move. Here is one simple way – use the mouse to step through each time increment of this simulation. While the **Learn Java Now** book provides details on how to do this, we simply give you the code below. Add this code to the Test file as additional methods. The idea now is that when the mouse is positioned **over the display window**, clicking on the left mouse button will cause each moveable object to change state. Add this to your system (within the Test class) and check out its performance.

```
// Mouse support
// -------------------------------------------------------------------------

public boolean mouseDown(Event evt, int x, int y)
{// if down, ask each object to move, then repaint

        for (int i = 0; i < 5; i++)
        { MoveableThing mt = moveableThingArray[i];
        mt.UpdateState(dTimeIncrement);

       // be sure to add dTimeIncrement as a state variable
       // with some value (e.g. 1.0)
        }
```

```
            repaint();
            return true;
    }

    public boolean mouseUp(Event evt, int x, int y)
    { return true;
    }
```

Now try out your system, checking that as you click the mouse button, your objects move through the window.

Problem 8 Of course, so far, these moving objects are a bit uninteresting as they will simply drift across the window and disappear. So let's fix that. To your definition of a `MoveableThing`, add a maximum x and y position. Then, in your `UpdateState` method, you can incorporate boundary reflection. This means that if an object moves past the maximum x position, for example, then it should bounce off of that boundary – this simply means that its x position is currently n units beyond the maximum value, then the x position should be changed to lie n units inside the maximum. As well, change the x velocity to be in the opposition direction. Do the same for y. Also, you should check for the $x = 0$ and $y = 0$ boundaries as well.

Now try out your system, and check that the moveable objects bounce within the confines of the window.

Problem 9 Let's add another wrinkle to our system. Suppose that two objects collide. In that situation, let's have them cancel each other out. By this, we mean that the smaller of the two objects should have its size reduced to 0, and the larger of the objects should have its size reduced by the size of the smaller one. To implement this idea, we need to change the `MouseDown` method of `Test`. In particular, add code that will check for each pair of objects in our system, whether during this step of the simulation the two objects touch. There are several ways of doing this. One is to use the following mathematical observations.

Let's let the position of the center of one of our objects be p (a 2D vector), with radius r and velocity v. Let the other object have center q, radius s and velocity u. Let the time increment for each step of the simulation be represented by T. Then the distance between the two points at any stage of the simulation t is given by the distance between $p + vt$ and $q + ut$. Some standard calculus can be used to show that the two centers of the objects are closest when

$$t = \frac{(q_x - p_x)(v_x - u_x) + (q_y - p_y)(v_y - u_y)}{(v_x - u_x)(v_x - u_x) + (v_y - u_y)(v_y - u_y)}$$

provided, of course, that the denominator is not zero (and were q_x refers to the x or first component of the vector).

Thus, you can simply code up this test for any pair of objects in your simulation (using a zero velocity for a stationary object) finding the time at which the objects are closest. Remember that if the solution to the above equation gives a number less than 0, then you should use $t = 0$ as the time of closest approach, and if the solution to the above equation is greater than T then you should use $t = T$ as the time of closest approach.

Given such a time t, the actual squared distance between the centers of the objects is given by

$$\begin{aligned} & (p_x - q_x)(p_x - q_x) + (p_y - q_y)(p_y - q_y) \\ + \quad & 2t[(p_x - q_x)(v_x - u_x) + (p_y - q_y)(v_y - u_y)] \\ + \quad & t^2[(v_x - u_x)(v_x - u_x) + (v_y - u_y)(v_y - u_y)] \end{aligned}$$

Thus to see if there is a collision, simply check to see if this squared distance is less than

$$\left(\frac{r+s}{2}\right)^2$$

Implement and test this change.

Problem 10 Now let's add another class to our system, a class of AccelerableThings that extends the class of MoveableThings. This class should inherit the methods and members of its underlying class, with the following additions. It's constructor should take as input, in addition to the parameters needed to build the base class, `float` arguments that specify an X and a Y acceleration rate, and `int` arguments that specify a start and end time. These objects have the property that when initialized, they set an internal clock to 0. Then each time state is updated, the clock increments by 1. When the clock reaches the object's `start` time, acceleration kicks in and the objects velocity is updated in a manner similar to how velocity is used to update position. This continues for each time interval until the `end` time is reached, at which point acceleration is turned off.

Implement this class, add a few such objects to your simulation and test it out. Turn in a listing of your code.

Problem 11 Finally, add something new to your simulation. For example, you could add a new class of objects with a different behavior, e.g. when one of these new objects touches another smaller object it absorbs that object and continues moving.

Or you could add something completely different. Be imaginative!