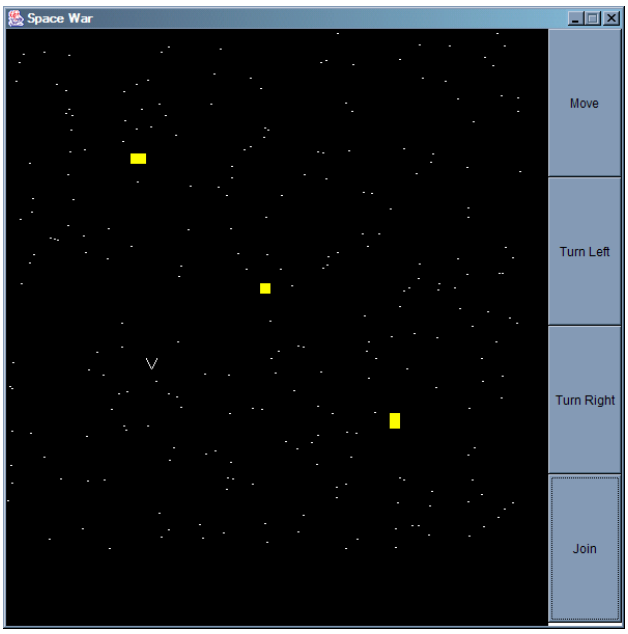
CSA 283 – Spring 2007
Program #1 Due: February 2, 2007
"Multi-User Game"

Objectives: In this assignment you will implement the communication portions of a multi-user computer game. The game will utilize of a client/server architecture to allow multiple players on interact in a shared 2D world. The server portion of this networked application will be responsible for receiving update messages from individual clients and forwarding those messages to all other clients. It will also attempt to ensure that each client is provided with a consistent set of obstacles on start up. Clients will be responsible for sending update messages for a single game entity and receiving update messages via the server for all other game entities. Game communication will be largely be based on the of best effort, connectionless, UDP/IP network communication. At your option you may use TDP/IP communication to ensure reliable game initialization.

Description:

Below is a screen capture of the space game. The game is played by maneuvering your own ship and ramming the ships of your opponents. In a head on collision both ships are destroyed. Otherwise, only the ship of the opponent is destroyed. The game also includes obstacles through which no ship can be moved. Your own ship is maneuvered by pressing the buttons to the right of the display. The starter code for the game can be run as is but has no communication with the server that provides the locations of obstacles and updates giving the position of other players.



The driving class is SpaceGame. Your own ship can be inserted into the game by pressing the "join" button. You can maneuver your ship around using the "Move," "Turn Left," and "Turn Right" buttons.

There are numerous classes included in the SpaceGame project. You will be adding code to the SpaceGame class. You may also decide to write Receiver, and Sender classes. The SpaceGame class handles user inputs by updating the Sector display for the user's ship. You must add code where you see a // TODO comment to send and receive messages and create a Datagram socket(s). Both the server (SpaceGameServer.java) and the client (SpaceGame.java) have been started for you.

The server is responsible for sending obstacles to new clients, and receiving and forwarding messages. It should maintain a list or array containing the IP addresses and port numbers of all participating clients. When a client joins the game. The server should add it to this list. When forwarding messages it should not send a copy back to the originating client.

During initialization the Clients sends a "start" message to the server and downloads the locations of all obstacles in the game. The client must know the IP address and port number of the Server a-priori. Once the obstacles are received the SpaceGame class must receive update messages from the server and use them to update the sector display and send update messages to the server each time the user presses the "Join," "Move," "Turn Left," or "Turn Right" button.

It is recommended that you write separate Sender and Receiver classes for your client. The Sender class could be responsible for receiving information from the SpaceGame object and placing that information into a message of the appropriate type and sending the message to the server. After initialization, the Receiver would use and infinite loop or run on a separate thread to continuously receive message from server and using the information contained in the messages to update the display encapsulated in a Sector object.

Getting Obstacles

The IP address of the server is known by client ahead of time. After receiving an initial "Join" message from a client, the server sends the x y locations of obstacles as pairs of ints. The client uses each pair of numbers to create an obstacle and add it to the sector display using the addObstacle method. The server indicates the end of obstacle location data by sending a negative number.

Updating the Display

Three types of messages should be used for in the game communication. The message types are "join," "update," and "remove." The messages have the following format. The first four fields and the last fields should be of type int. The IP fields should be of type byte.

Code	X position	Y position	Heading	IP1	IP2	IP3	IP4	Port

The codes for the “join,” “update,” and “remove” messages are respectively 0, 1, and 2. The entity to which the message refers is uniquely identified by the IP fields and the port number. In all cases the position and heading fields carry the position and heading information of the ship being referred to. The getByAddress(byte[] addr) method can be used to construct a InetAddress object using the IP bytes in the message.

The join message is sent by a user that has just joined or rejoined the game. The server receives this message and forwards it to all other clients. Upon receiving a join message, clients should construct an AlienSpaceCraft object using the information contained in the message and add the AlienSpaceCraft to the Sector display using the updateOrAdd method.

The update message is sent by a user each time a turn or move is made and forwarded by the sever. Upon receipt the client should construct an AlienSpaceCraft object using the information contained in the message to update the Sector display for the object using the updateOrAdd method.

The remove message is sent by a user when the user has rammed and thus destroyed another ship. If the collision was head on the user sends a remove message for itself as well. Upon receipt, the client should construct an AlienSpaceCraft object using the information contained in the message to update the Sector display using the remove method.

Advance Functionality

Successfully completing the above project will probably get you a grade in the high C low B range. To get more points consider implementing some of the following:

- Multi threaded Receiver class
- TCP/IP communication for passing obstacle locations.
- Quit messages from clients when they leave the game.
- Have server collect update messages and forward in a single Datagram

Starting files: AlienSpaceCraft.java, Constants.java, OwnSpaceCraft.java, Sector.java, SpaceCraft.java, SpaceGame.java, Obstacle.java, and ObstacleServer.java, and SpaceGui.java.

Input: User mouse button clicks and datagrams from other users.

Program Output: Game display and join, update, and remove messages to other users. Obstacle location data messages.

Grading. Scores will be assigned based on the correctness and completeness of your program, the logic and efficiency of your program design, and your ability to follow the stylistic conventions below.

What to turn in:

1. **Hardcopy of your any classes you write or modify:** Do not turn in hard copies of provided classes.

2. **Lab cover sheet:** Fill out the cover sheet and staple the cover sheet to the front of the Sender code

3. **Electronic copy of your Project:** Copy your project folder to the desktop. Change the name to your last name follow by "programOne". Place a copy of that folder in

   L:\Turnins\bachmaer\csa283\Programming Assignments\Programming Assignment 1.

   Your project must run with eclipse.

Stylistic Requirements:

Document your program with comments so that I can follow your logic. All method, class, and variable names should be meaningful. All variable declarations should be commented. Include your name on a separate line at the top of the program. Indent code three spaces. Use single blank lines within methods and white space to make code more readable. Leave two lines between method definitions. Put an "end" comment after the closing braces for each method and class as shown below

```
class ArnieHelp
{

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


} // end ArnieHelp class
```

Identifiers:

Be sure to use meaningful terms for all identifiers. Class names must begin with an upper case letter. Variable, class data members, and method name must begin with a lower case letter. Identifiers for constants should be all upper case with individual words separated by an under score. For example:

```
      final int DAYS_IN_YEAR = 365;
```

Comments:

All fields and method variables should have a comment indicating their purpose, such as:

```
    // Total bank account balance.
    private double accountBalance;
```

JavaDoc comments for each public class should look like the following (include the course, the project number, a description of the class, your name and version (use @version and @author):

```
/**
 * CSA 283 Project 1: Help out Arnie.
 *
 * Calculates distance and average speed while running.
 *
 * @author Eric Bachmann
 * @version 1.0 September 14, 2001
 */
class ArnieHelp...
```

JavaDoc comments for each method should look like the following. Include @return if the method returns a value. Include on @param tag for each of the arguments a method takes:

```
    /**
     * Increases the account balance by the amount
     *
     * @param the deposit amount
     * @return the updated balance
     */
    public double deposit(double amount)
```

JavaDoc comments for each public data member should look like the following:

```
    /** Holds the balance of the bank account */
    public double balance;
```

Program Indentation:

Every set of curly braces in your program defines a *code block*. In general every thing inside a code block is indented one tab space from the position where the code block began. This applies to the bodies of classes and methods as well as any code block associated with a loop, if or else statement. Follow the indentation conventions below. Again, there should be two blank lines between each method definition and no more that one blank line in a row within the methods. All if and else statements must have curly braces.

```
class SomeClass
{

   private double someDataMember;


   public void someMethod()
   {
      statement1;

      if (someCondition) {
         statement in if 1;

      }
      else {
```

```
         statement in else 1;
      }

      statement3;

   } // end someMethod


   public void switchMethod()
   {
      switch(some int expression) {

      case –1:
         statements;
         break;

      case 0:
      case 1:
         statements;
         break;

      default:
         statements;
         break;
      }

   } // end switchMethod


} // end SomeClass
```

There should be nothing in your Sender code beyond column 80. Otherwise, your indentation pattern will be destroyed by wrap arounds in the hard copy.

Some Design and Logic Considerations

1 Use many short methods. No method should be more than one page long. Less than half a page is even better.
2 Avoid the use of “magic numbers” in your code. Use constant variables instead.
3 Don’t burn memory by repeatedly instantiating objects when existing ones could be used.
4 Don’t make a variable a class data member if it could be a local variable to a single method.
5 Use public, package, protected, and private access specifiers in an appropriate manner.
6 More to come…

CSA 283 – Spring 2007
Program #1 Due: February 2, 2007
"Multi-User Game"
100 Points Possible

Name _____ Section ____A__ _____

List any enhanced features you implemented.

Comments by grader: