Starting Networking

Assumed knowledge:

- *The basics of java*
- *What a package is*
- *What a main method is*

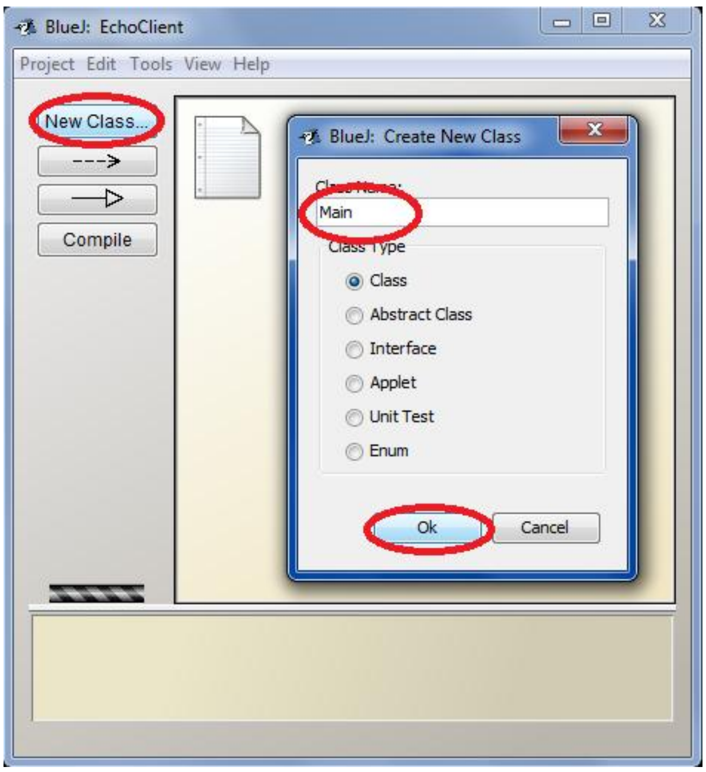
Covered in this lesson:

- *Creating simple networked server-client applications*
- *Using BlueJ*
- *Creating .bat files to run .jar files using command prompt*
- *Threading (briefly)*

In this guide I will cover the basics of networking, and walk through the creation of some programs.

First off, I should introduce the java package we're going to use for this: The java.net.* package. You can go right ahead and look at its documentation but it might make slightly more sense to see some of the methods in action, as that will probably be clearer. We're also going to be using the java.io.* package. When I use classes from java.io I won't fully explain how they work, as this is about networking. If you want to understand these uses, there will be material on the internet, and I _might_ make a lesson in the future covering it.

Now, although this is a course for programming in Greenfoot, I feel it will be much clearer if first we make a program or two in BlueJ (If you don't have this, download it here). BlueJ (actually made by the same people as Greenfoot, so you may recognise the code editor interface) is a simple IDE for creating programs rather than games (you can create games in it, but it's much, much more complex than doing it in Greenfoot).



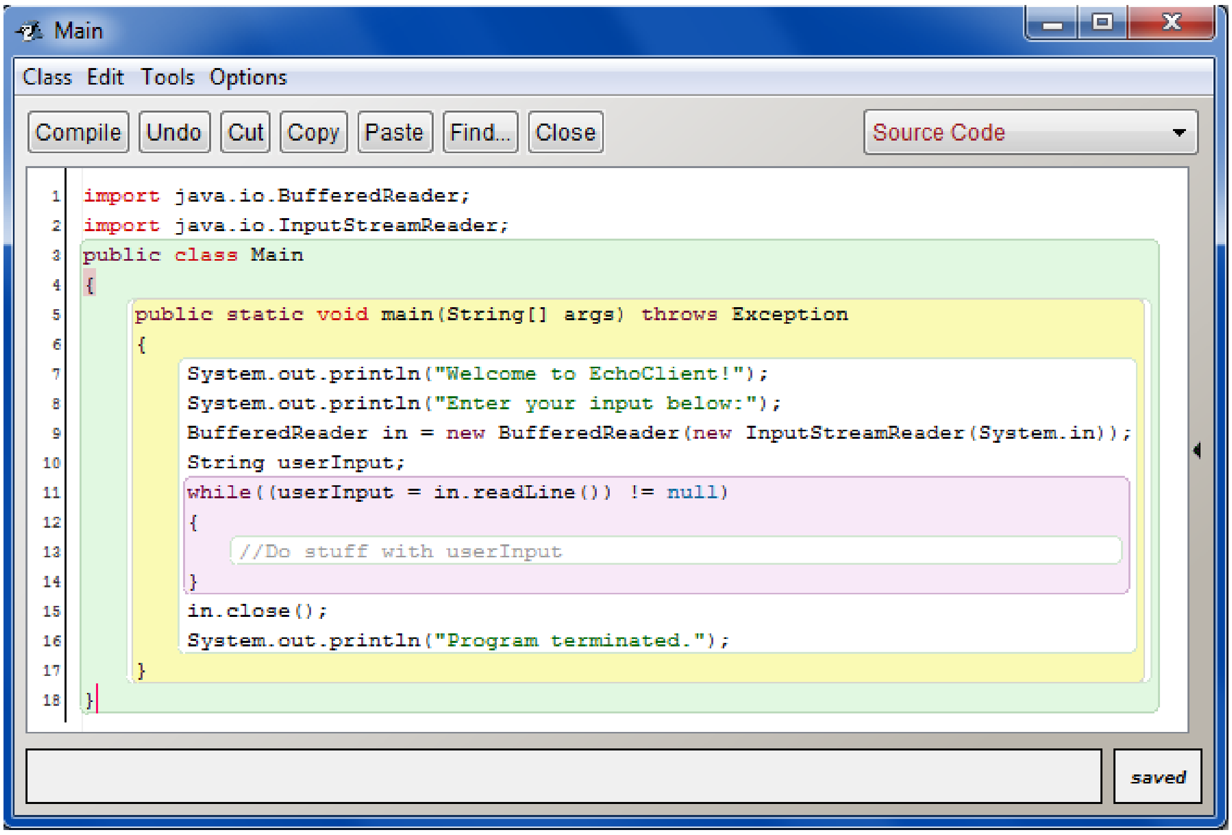
The first program we're going to be making will be called "Echo". What it will do (as you might have guessed) is take a message from the user, send it to the server, and display the message the server sends back (which will be, in this case, the same message with "Echo: " added before it).

Now for this we're going to have to create 2 separate programs (named "EchoClient" and "EchoServer" - I recommend naming networked programs like this to avoid confusion). The first program I'm going to walk through the creation of is "EchoClient".

Firstly what we're going to do is create a class. To do this, click the "New Class..." button, enter the name for it (We're going to call it "Main"), and hit OK.

Once you've done this, click on the box that has the title "Main" on it to access the code editor, and the fun part can begin.

Now the first part we should do is getting the user's input. Here's my approach to it:



```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Main
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Welcome to EchoClient!");
        System.out.println("Enter your input below:");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String userInput;
        while((userInput = in.readLine()) != null)
        {
            //Do stuff with userInput
        }
        in.close();
        System.out.println("Program terminated.");
    }
}
```

What's going on here is effectively:

1. Print a welcome message
2. Create a (Buffered)reader that reads from the System input stream
3. Create the userInput variable
4. Read a line from the user and store it as userInput
5. Check the user doesn't want to exit the program (indicated by typing "Stop")
6. If the user does want to continue, do stuff with the input and then go back to step 4
7. If the user wants to quit, print a termination message and stop the program

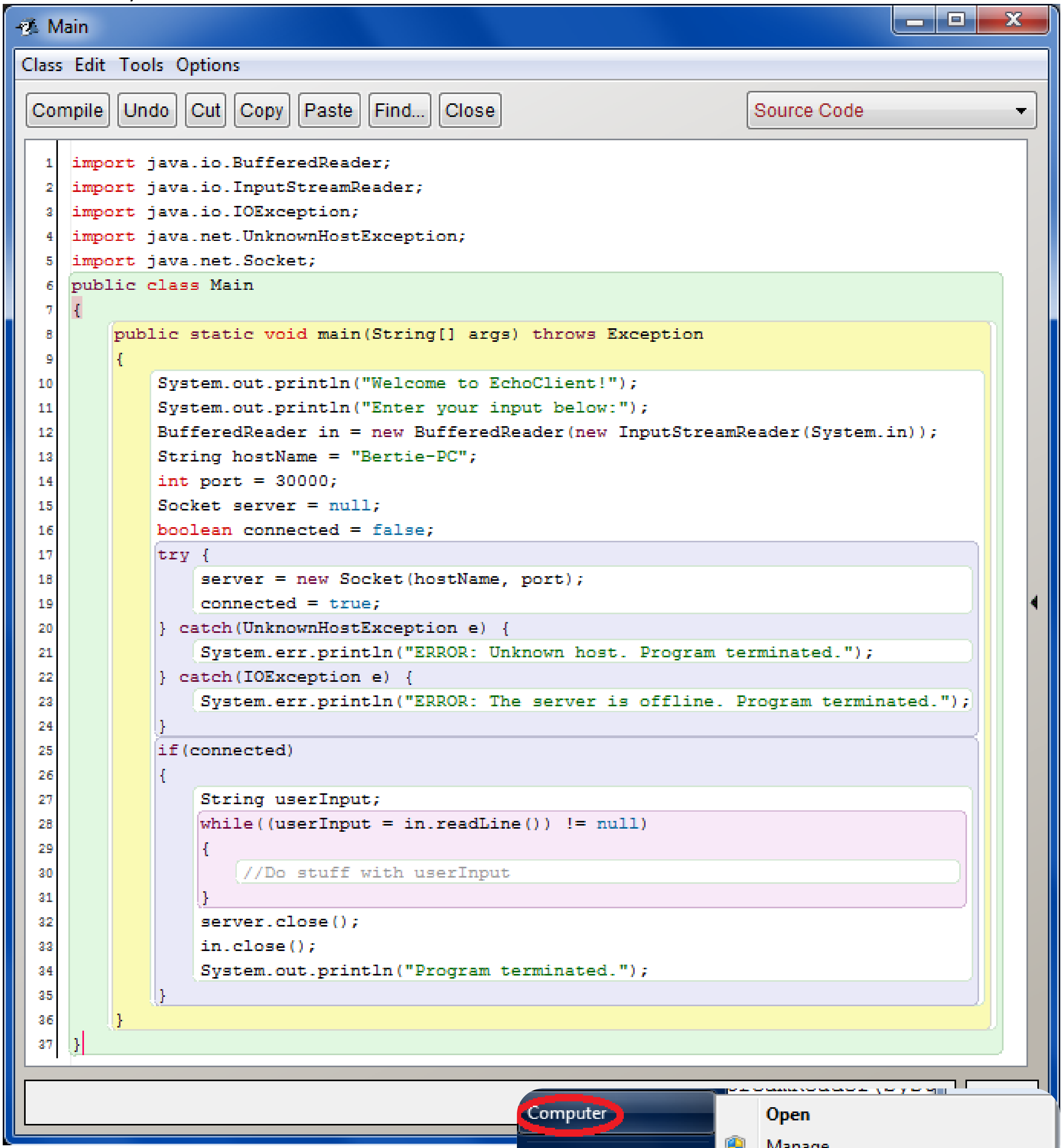
Now we need to send the user input to the server, wait for a response and display the response received.

Thinking about how we will do this, the process could be concisely represented as such:

1. Attempt to establish a connection with the server
2. If successful, continue, if not, print an error message and exit
3. Send our input to the server
4. Wait for a response from the server

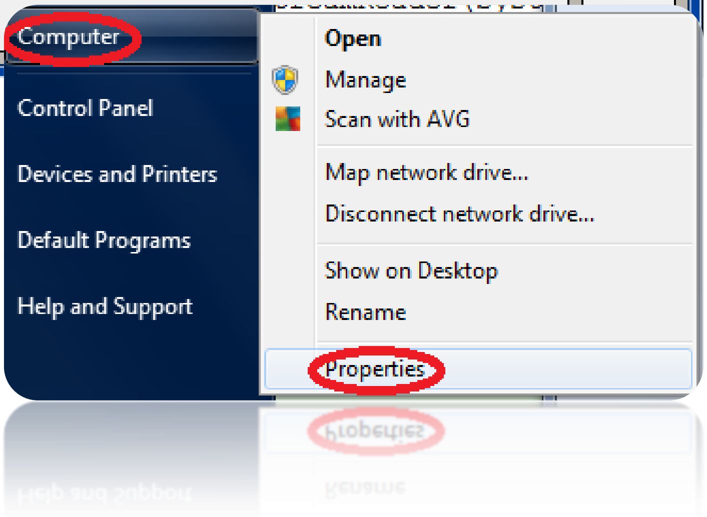
5. Print the response

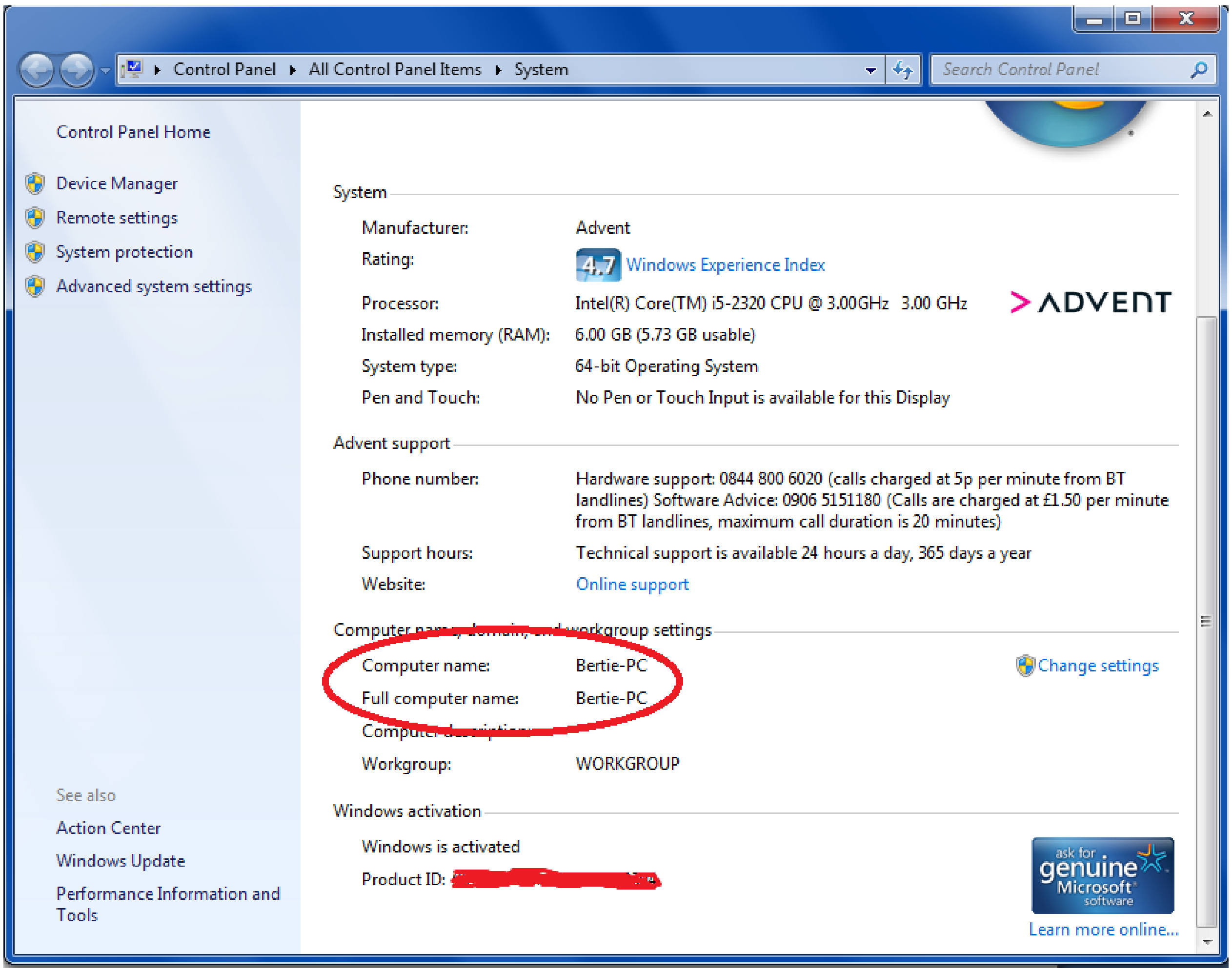
So the first part required is to establish a connection with the server. The following code does exactly that:



```
 1 import java.io.BufferedReader;
 2 import java.io.InputStreamReader;
 3 import java.io.IOException;
 4 import java.net.UnknownHostException;
 5 import java.net.Socket;
 6 public class Main
 7 {
 8     public static void main(String[] args) throws Exception
 9     {
10         System.out.println("Welcome to EchoClient!");
11         System.out.println("Enter your input below:");
12         BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
13         String hostName = "Bertie-PC";
14         int port = 30000;
15         Socket server = null;
16         boolean connected = false;
17         try {
18             server = new Socket(hostName, port);
19             connected = true;
20         } catch(UnknownHostException e) {
21             System.err.println("ERROR: Unknown host. Program terminated.");
22         } catch(IOException e) {
23             System.err.println("ERROR: The server is offline. Program terminated.");
24         }
25         if(connected)
26         {
27             String userInput;
28             while((userInput = in.readLine()) != null)
29             {
30                 //Do stuff with userInput
31             }
32             server.close();
33             in.close();
34             System.out.println("Program terminated.");
35         }
36     }
37 }
```

First things first, I should give a couple of points to note, specifically about lines 13 and 14. For your program to work you'll need to change line 13 to the host name of **your** machine. If you don't know your host name, you'll want to view your computer's details




Control Panel ▸ All Control Panel Items ▸ System

Search Control Panel

Control Panel Home

Device Manager

Remote settings

System protection

Advanced system settings

See also

Action Center

Windows Update

Performance Information and Tools

System

Manufacturer:	Advent
Rating:	4.7 Windows Experience Index
Processor:	Intel(R) Core(TM) i5-2320 CPU @ 3.00GHz 3.00 GHz
Installed memory (RAM):	6.00 GB (5.73 GB usable)
System type:	64-bit Operating System
Pen and Touch:	No Pen or Touch Input is available for this Display

Advent support

Phone number:	Hardware support: 0844 800 6020 (calls charged at 5p per minute from BT landlines) Software Advice: 0906 5151180 (Calls are charged at £1.50 per minute from BT landlines, maximum call duration is 20 minutes)
Support hours:	Technical support is available 24 hours a day, 365 days a year
Website:	Online support

Computer name, domain, and workgroup settings

Computer name:	Bertie-PC
Full computer name:	Bertie-PC
Computer description:	
Workgroup:	WORKGROUP

Change settings

Windows activation

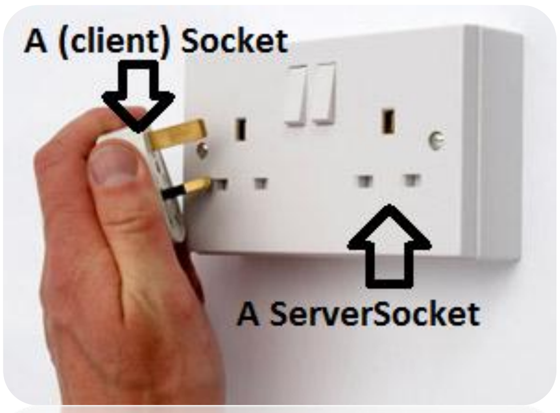
Windows is activated

Product ID: [redacted]

Learn more online...

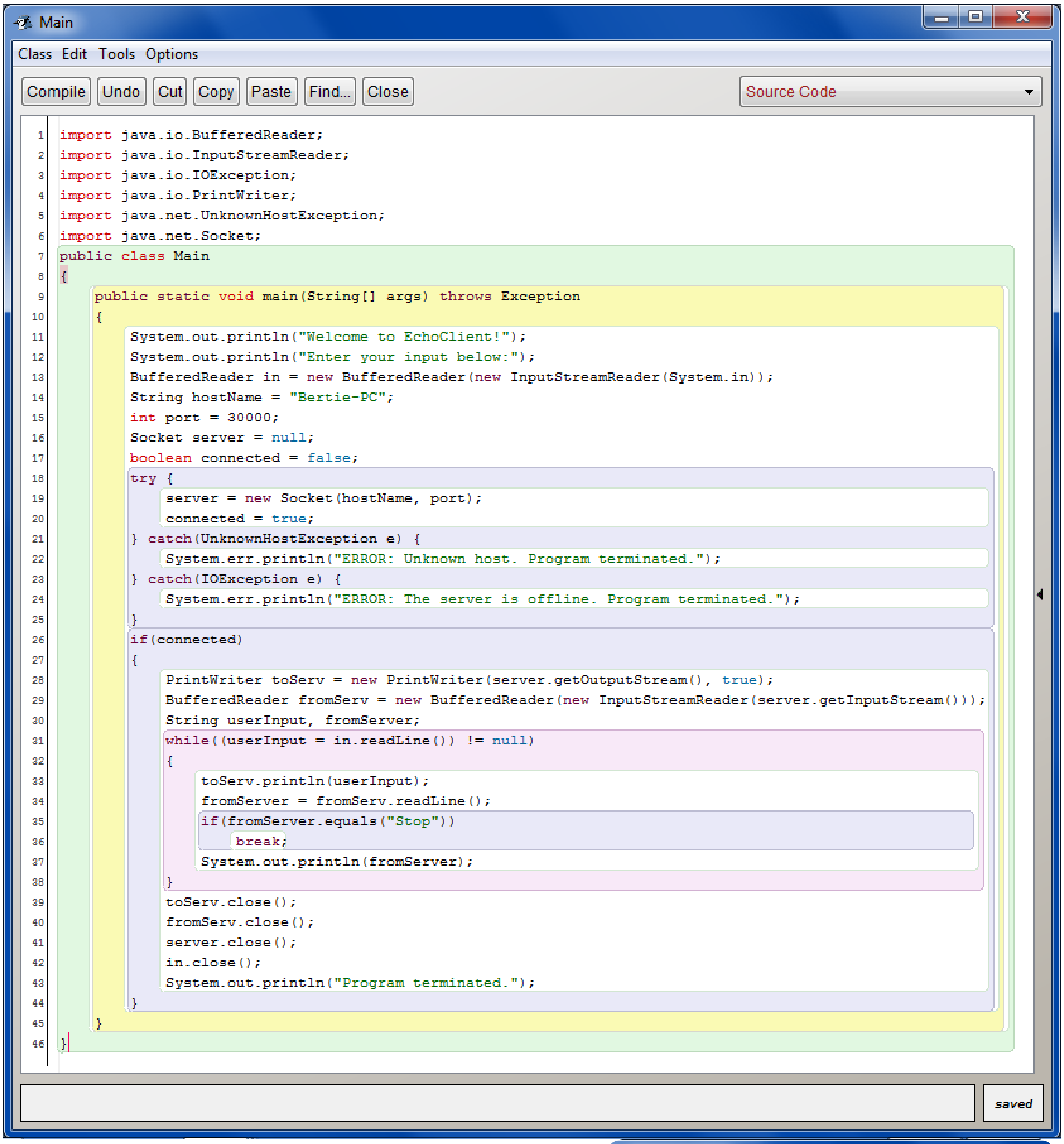
by doing *Start Menu* → *Computer* → *Properties*. Then, scroll down until you see your host name.

Moving on to line 14, the port number for our application is going to be 30000 because it is free (or at least it is on my machine, you may have to change it later on when we're making our server application if you find that port 30000 is busy for you... Anyway, we'll get onto that in time).

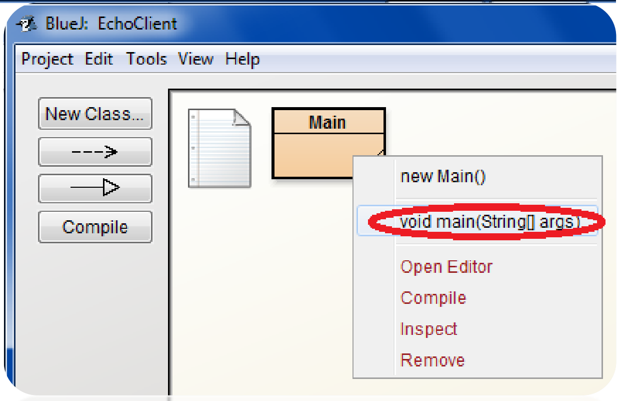


Going back to our client application, you'll notice that our connection to the server is in the form of a Socket object. In Java, connections are established between clients and servers in the form of Sockets and ServerSockets, where the Socket is the client's end of the connection and the ServerSocket is, well, the server's end of things. Imagine (for now) that Sockets are plugs, and ServerSockets are plug sockets (to help visualize this, see right). My point being that you not only need both of them, but they - in the plug analogy - must be the same type. What I mean is the port number in your client program must be the same as the one your server is 'listening' on.

Anyway, back to the code. Left to do on the client is sending our input to the server and getting the response - here's my solution to this:



```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.net.UnknownHostException;
import java.net.Socket;
public class Main
{
    public static void main(String[] args) throws Exception
    {
        System.out.println("Welcome to EchoClient!");
        System.out.println("Enter your input below:");
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        String hostName = "Bertie-PC";
        int port = 30000;
        Socket server = null;
        boolean connected = false;
        try {
            server = new Socket(hostName, port);
            connected = true;
        } catch(UnknownHostException e) {
            System.err.println("ERROR: Unknown host. Program terminated.");
        } catch(IOException e) {
            System.err.println("ERROR: The server is offline. Program terminated.");
        }
        if(connected)
        {
            PrintWriter toServ = new PrintWriter(server.getOutputStream(), true);
            BufferedReader fromServ = new BufferedReader(new InputStreamReader(server.getInputStream()));
            String userInput, fromServer;
            while((userInput = in.readLine()) != null)
            {
                toServ.println(userInput);
                fromServer = fromServ.readLine();
                if(fromServer.equals("Stop"))
                    break;
                System.out.println(fromServer);
            }
            toServ.close();
            fromServ.close();
            server.close();
            in.close();
            System.out.println("Program terminated.");
        }
    }
}
```



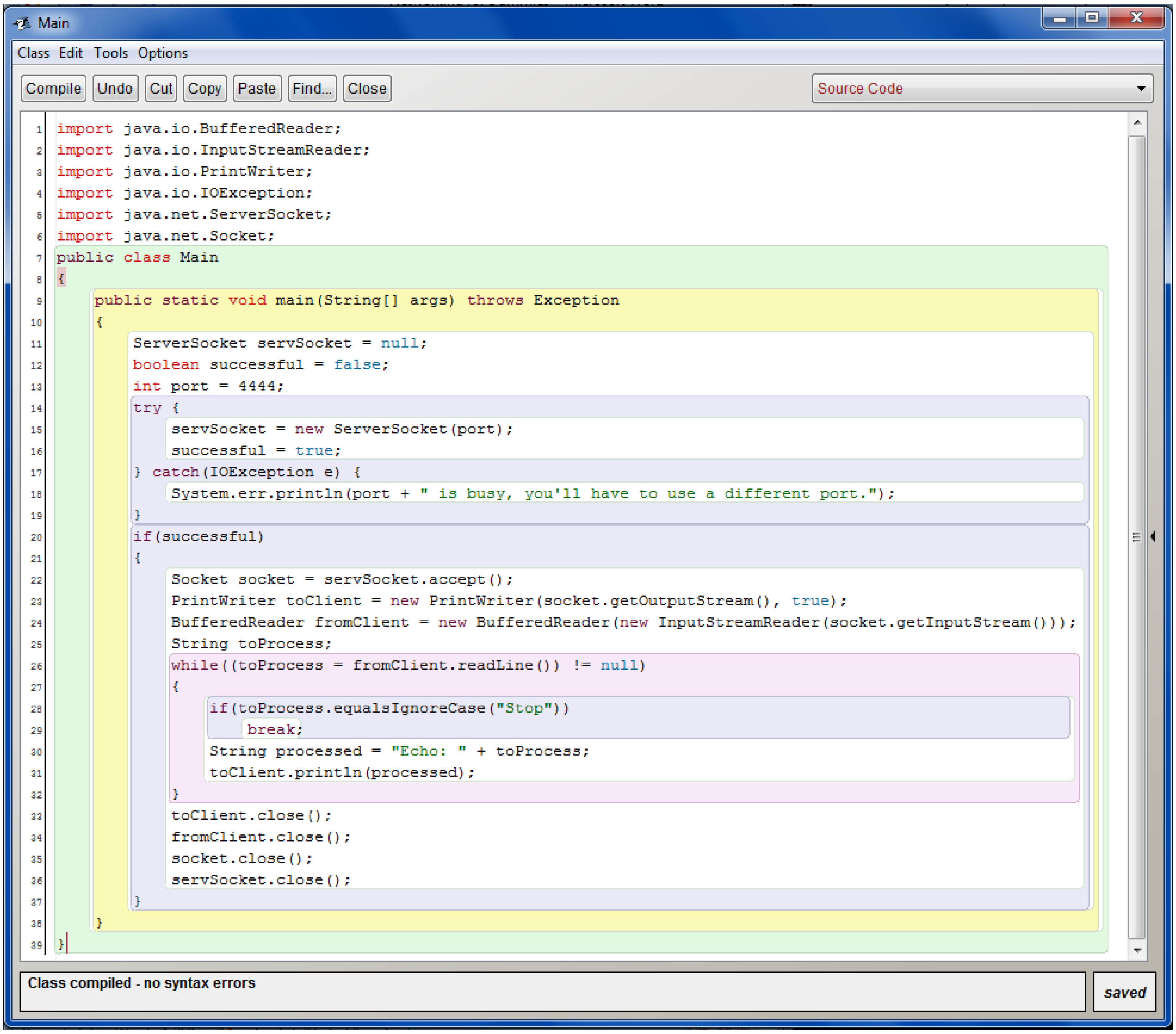
Our client program is now completed, however if you run it now, you'll get an error telling you that the server is offline - which of course it is, as we haven't even made it yet! I'm not going to go over this code, as instead I'm going to move on to making the server application, and once that's made I'll talk about how they are interacting, which should make more sense.

Before I do that, I'm just quickly going to cover running program(s). In BlueJ, you can (similar to Greenfoot) actually run them within the IDE, rather than having to go to the pain of exporting it. To do this, right click on the box which says "Main", and then invoke our method ("void main(String[] args)"). I will cover exporting programs later, once we have created our server program (it's slightly more complicated than you initially may think).

Starting the server program, we could come up with an idea of how it might work, like this:

1. Try to make a ServerSocket
2. If successful, wait until someone connects to the server, else quit
3. When someone connects, wait for messages from the client
4. When a message is received, manipulate it and send back the reply
5. Keep repeating steps 3&4

Put into code, this looks like the following (also I'm assuming you've made a new project called "EchoServer" with, again, a class called "Main"):



```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
public class Main
{
    public static void main(String[] args) throws Exception
    {
        ServerSocket servSocket = null;
        boolean successful = false;
        int port = 4444;
        try {
            servSocket = new ServerSocket(port);
            successful = true;
        } catch(IOException e) {
            System.err.println(port + " is busy, you'll have to use a different port.");
        }
        if(successful)
        {
            Socket socket = servSocket.accept();
            PrintWriter toClient = new PrintWriter(socket.getOutputStream(), true);
            BufferedReader fromClient = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            String toProcess;
            while((toProcess = fromClient.readLine()) != null)
            {
                if(toProcess.equalsIgnoreCase("Stop"))
                    break;
                String processed = "Echo: " + toProcess;
                toClient.println(processed);
            }
            toClient.close();
            fromClient.close();
            socket.close();
            servSocket.close();
        }
    }
}
```

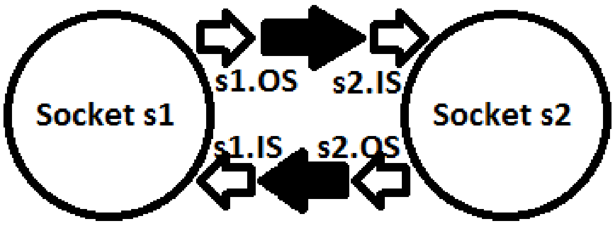
Class compiled - no syntax errors

saved

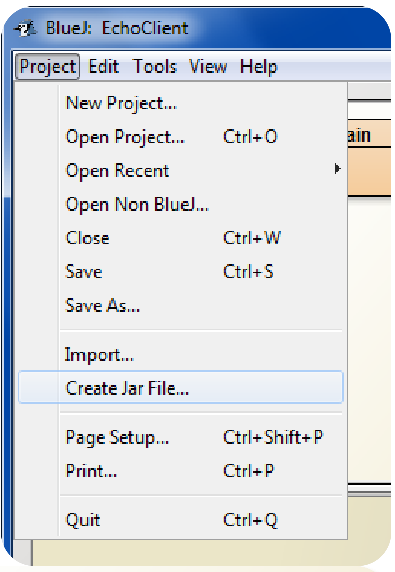
Now, while it might seem a big jump for me to go from nothing to the entire program, don't worry - I'm going to go over it!

One thing you may or may not notice from my code, is that I wasn't entirely telling the truth in my plug socket picture earlier - for a correct analogy, the ServerSocket is the actual power supply (which there's only one of), and the Socket in the server program is the plug socket (of which we only have one at the moment, but there is the potential for many to all stem from the same ServerSocket). The client Socket is, however, still the plug. A difference between plugs and Sockets is that a plug connected to a plug socket can only have transfer of power (data) in one direction - from the socket to the plug - however when you have two connected Sockets it can go in both directions (I talk about this more in the following paragraph).

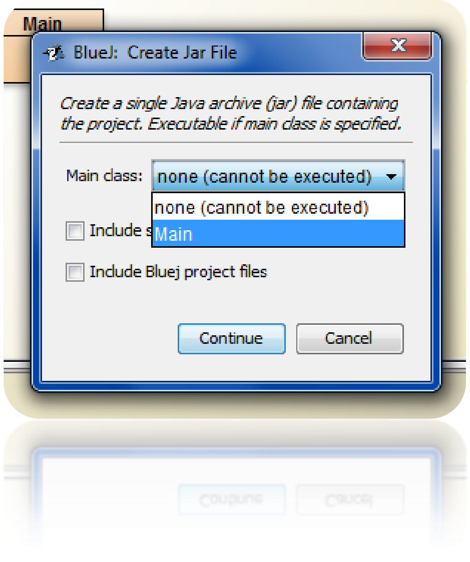
Walking through this code, we're creating a ServerSocket with a specified port. We then do `Socket socket = servSocket.accept();`. What this does is that `servSocket.accept()` listens for a connection request to our ServerSocket servSocket, and delays within the call to accept() until a client attempts to connect, at which point it makes a server-side Socket which is linked to the client who connected's Socket. When I say they're connected, I mean that one's InputStream is the other's OutputStream (for clarity, see right). Say that s1 in the diagram is our client Socket. When we do , the message goes to the right-facing black arrow - the stream of data between the two Sockets. So, when our server Socket (s2 in this case) reads from the stream, the message from the client is received. And, obviously, vice versa (when sending from s2 to s1). Anyway, it then stores that as a variable called "socket". Once we have our Socket linked to the client's one, we set up a PrintWriter and BufferedReader so we can easily access these streams talked about above. After that, we keep looping round, reading lines from the client, modifying them to say "Echo: <message>" as opposed to "<message>", and sending them back to the client. Once the server receives the message "Stop" from the client, it exits this loop and, for good habits, closes our connections.



**Key: s1.OS = left socket's output stream,
s2.IS = right socket's input stream, etc**



Now that we have both our client and our server programs finished, we're going to quickly test them. Instead of running them within the IDE (which you can of course do), we're going to export our programs. To do this, we're going to create two .jar files - one for our client, one for our server - and then create two .bat files that will run the programs in command prompt (so you can use System.in/System.out).

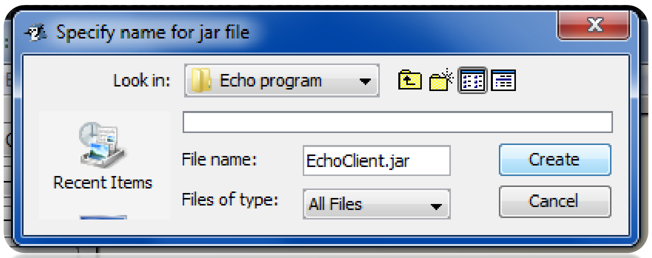


Create a single Java archive (jar) file containing the project. Executable if main class is specified.

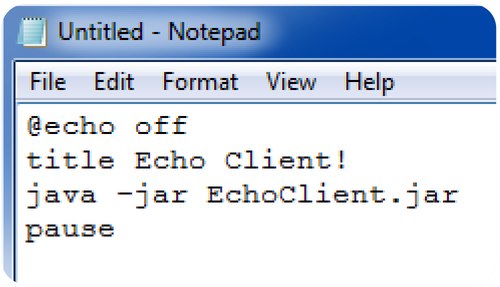
Main class: none (cannot be executed)
none (cannot be executed)
Main

☐ Include s
☐ Include Bluej project files

To make the .jar files, open up both of the projects and click *File → Create Jar File*. Once presented with the creation screen, select "Main" as the main class, and name the file either EchoClient.jar or EchoServer.jar (obviously depending on which you are exporting at the time). Once you've done this for both programs, you're going to need to open up Notepad.

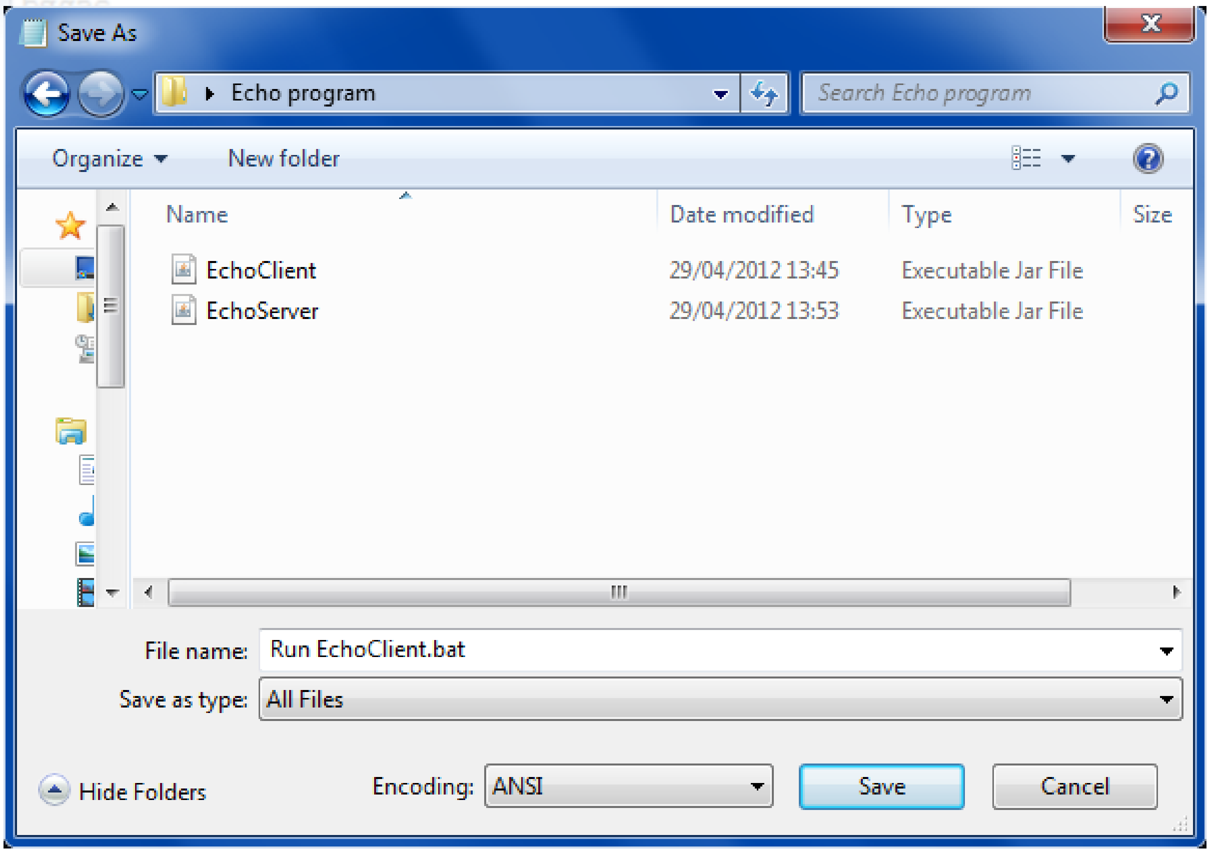


In Notepad, type the following:



```
@echo off
title Echo Client!
java -jar EchoClient.jar
pause
```

And save it as a .bat file in the same directory as your .jar files:



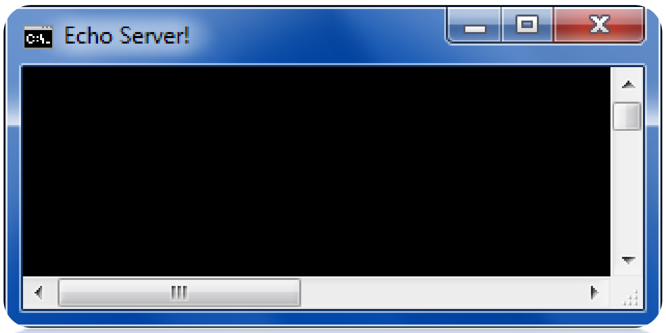
Name	Date modified	Type	Size
EchoClient	29/04/2012 13:45	Executable Jar File	
EchoServer	29/04/2012 13:53	Executable Jar File	

File name: Run EchoClient.bat
Save as type: All Files
Encoding: ANSI

Once you've done that for the client, replace any mentions of "Client" in the text to "Server" and save accordingly.

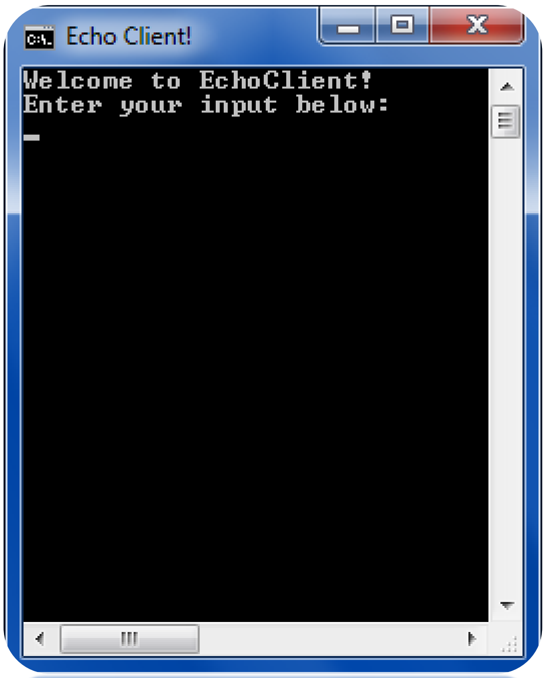
Time to run the programs! Navigate to the directory where your files are located, and run the server program's .bat file.

Hopefully, you should be presented with the following:



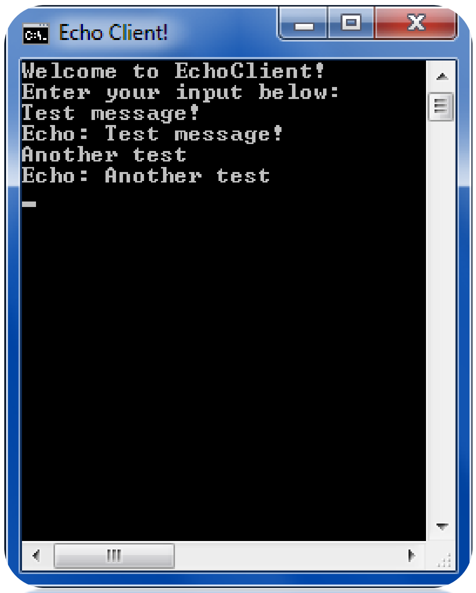
If you get the error message we included saying that port 30000 is busy, just change the port to a different one (not only in the server program but also the client one!) and re-export.

Once your server is running happily with no problems, run your client's .bat file - it should look like the following when running:



```
Welcome to EchoClient!
Enter your input below:
```

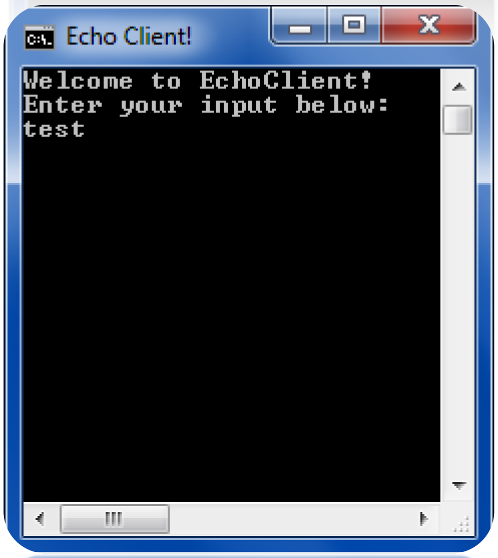
Enter a couple of test messages (press enter to send a message), and you should receive responses like so:



```
Welcome to EchoClient!
Enter your input below:
Test message!
Echo: Test message!
Another test
Echo: Another test
```

It works! Congratulations on your first networked program - however we're not done yet!

A problem with this is if you attempt to run a second instance of the client program, this happens:



```
Welcome to EchoClient!
Enter your input below:
test
```

If not clear from the picture, what occurs is the program loads up, and lets you send a message (it thinks that its connected to another - server - socket), however it then stalls, waiting for a response from the server that its never going to get, because the server is dealing with the first connection it received after running - the first time we ran the client program.

So, we need to support multiple clients.

To make this change, all we need to do is modify the server program - the client program doesn't care whether it's the only one connected or if there are a thousand others connected. What we are going to do to support this functionality is every time we have an incoming connection request, we create and dispatch a new Thread to deal with it. To quickly cover how Threads work, here's an excerpt from the documentation:

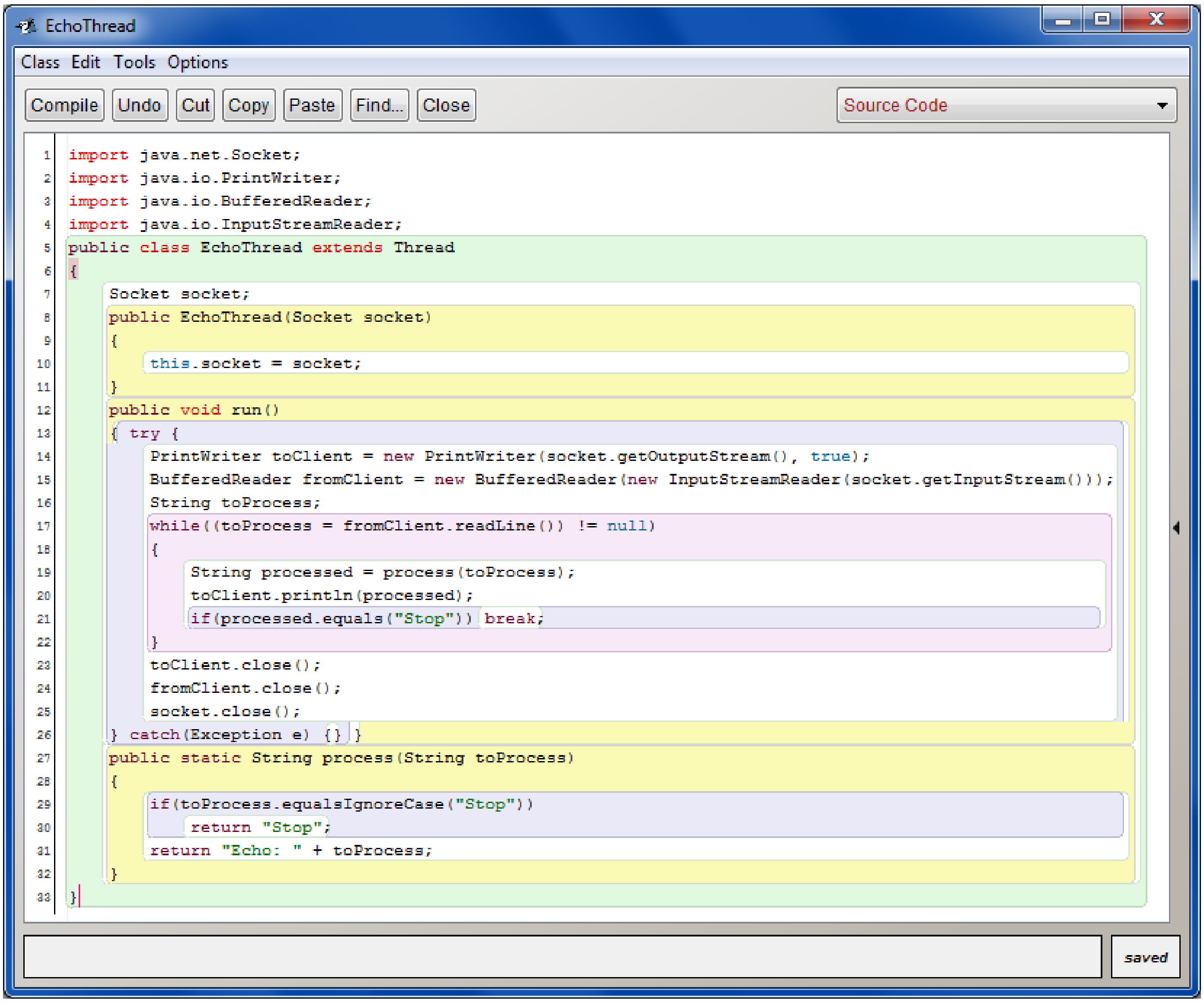
"declare a class to be a subclass of Thread. This subclass should override the run method of class Thread. An instance of the subclass can then be allocated and started. For example, a thread that computes primes larger than a stated value could be written as follows:

```
class PrimeThread extends Thread {

      long minPrime;

      PrimeThread(long minPrime) {

            this.minPrime = minPrime;

      }

      public void run() {

            // compute primes larger than minPrime

             . . .

      }

}
```

The following code would then create a thread and start it running:

```
PrimeThread p = new PrimeThread(143);

p.start();"
```

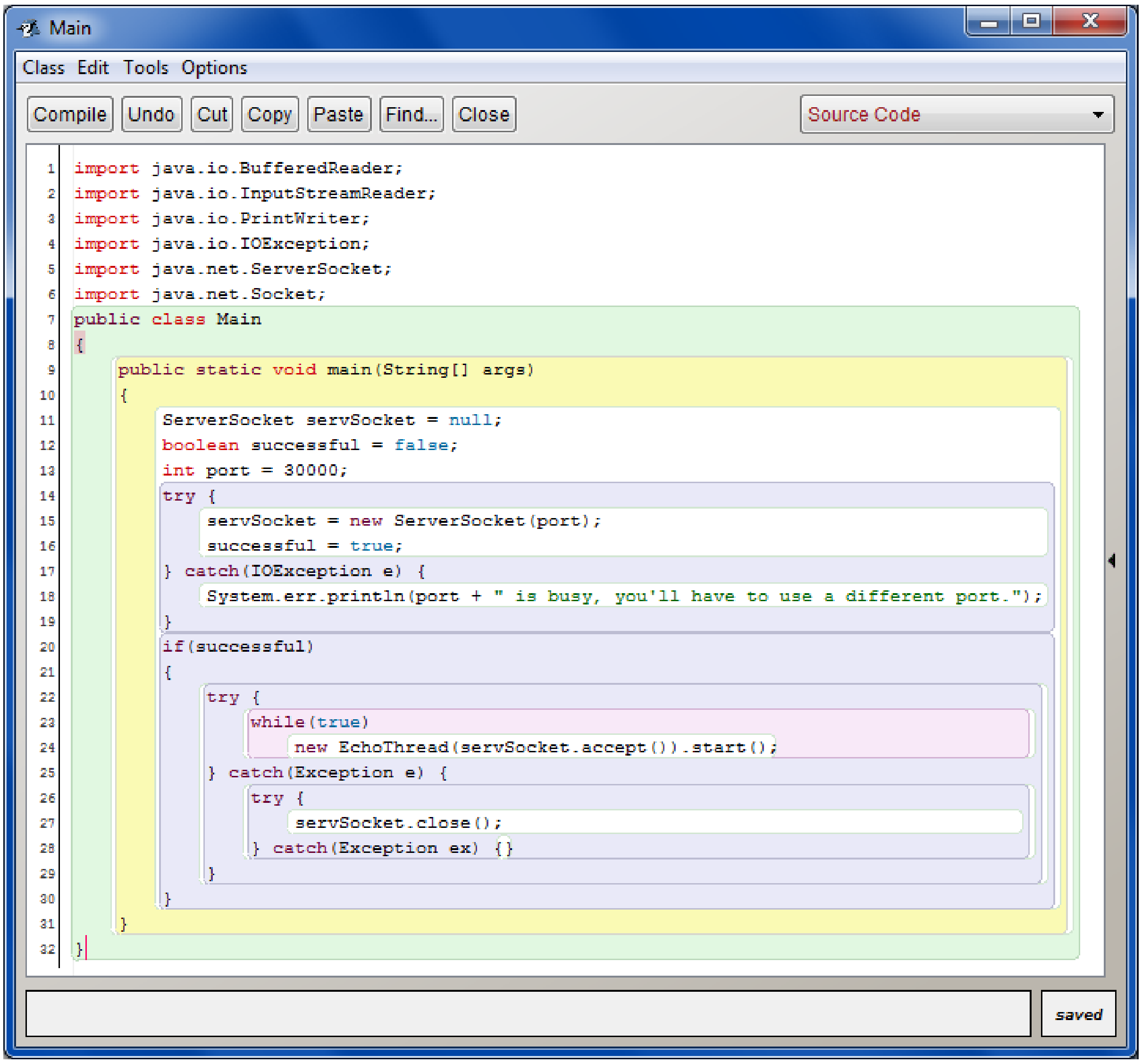
So, we're going to firstly create a new class named "EchoThread", with the following code:



```java
import java.net.Socket;
import java.io.PrintWriter;
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class EchoThread extends Thread
{
    Socket socket;
    public EchoThread(Socket socket)
    {
        this.socket = socket;
    }
    public void run()
    { try {
        PrintWriter toClient = new PrintWriter(socket.getOutputStream(), true);
        BufferedReader fromClient = new BufferedReader(new InputStreamReader(socket.getInputStream()));
        String toProcess;
        while((toProcess = fromClient.readLine()) != null)
        {
            String processed = process(toProcess);
            toClient.println(processed);
            if(processed.equals("Stop")) break;
        }
        toClient.close();
        fromClient.close();
        socket.close();
    } catch(Exception e) {} }
    public static String process(String toProcess)
    {
        if(toProcess.equalsIgnoreCase("Stop"))
            return "Stop";
        return "Echo: " + toProcess;
    }
}
```

I'm not going to go over this code because you'll notice a lot of this code is copy-pasted from our original "Main" class. Basically, what we're doing is just moving the processing to this new class.

Of course, we've now got to modify Main.class. What we're going to change is how we deal with incoming requests. I've covered what we're going to do already, so let's jump right into the code:



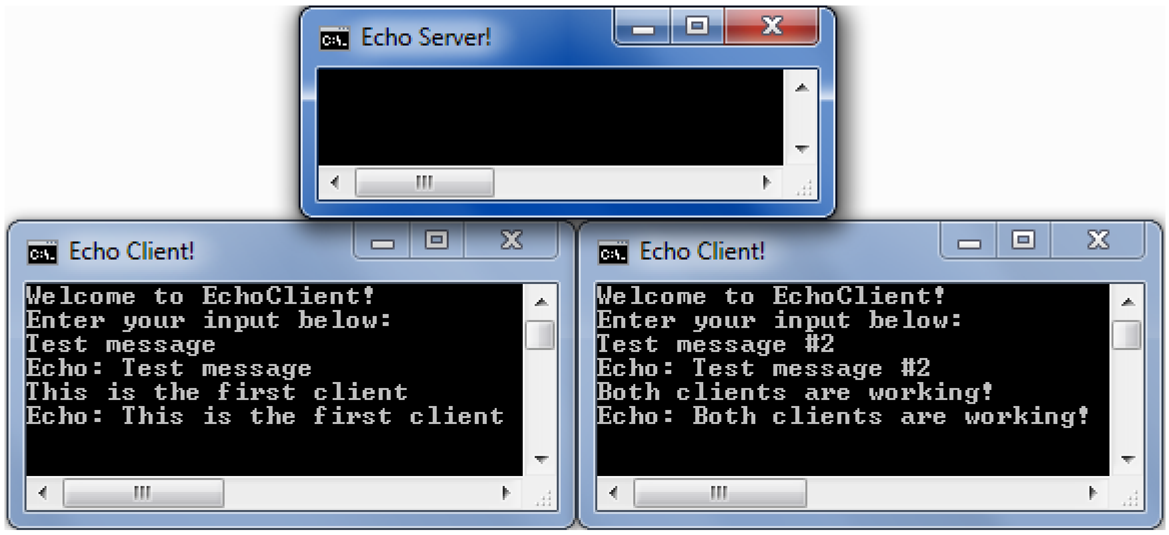
```
1  import java.io.BufferedReader;
2  import java.io.InputStreamReader;
3  import java.io.PrintWriter;
4  import java.io.IOException;
5  import java.net.ServerSocket;
6  import java.net.Socket;
7  public class Main
8  {
9      public static void main(String[] args)
10     {
11         ServerSocket servSocket = null;
12         boolean successful = false;
13         int port = 30000;
14         try {
15             servSocket = new ServerSocket(port);
16             successful = true;
17         } catch(IOException e) {
18             System.err.println(port + " is busy, you'll have to use a different port.");
19         }
20         if(successful)
21         {
22             try {
23                 while(true)
24                     new EchoThread(servSocket.accept()).start();
25             } catch(Exception e) {
26                 try {
27                     servSocket.close();
28                 } catch(Exception ex) {}
29             }
30         }
31     }
32 }
```

This code loops round (in lines 23-24), listening for a connection, and once it gets a link to a client, creates a connected Socket, passes it to a newly-created Thread, and starts the Thread. The Thread, like we saw in EchoThread.class then deals with the client.

If it's not clear how line 24 is doing that, it might be more obvious in a longer form:

```
while(true)
{
        Socket socket = servSocket.accept();
        EchoThread eT = new EchoThread(socket);
        eT.start();
}
```

Of course, now is the time to check it works! Re-export the server and get it running, then (same as before) get a client running and test it works. Now, to show our update was successful, run a second instance of the client. You should see that both the clients work simultaneously!



```
Welcome to EchoClient!
Enter your input below:
Test message
Echo: Test message
This is the first client
Echo: This is the first client
```

```
Welcome to EchoClient!
Enter your input below:
Test message #2
Echo: Test message #2
Both clients are working!
Echo: Both clients are working!
```

Great! We have a working, networked program!

That's all for this lesson, however I intend to make a future one covering how you can network your Greenfoot games to have network best scores, chat services, and more in them!

I recommend that you try it - at least in a simple form - first: Just make a server program in BlueJ, and then have your game connect to it like we did with our client program.

I hope you learnt from this, and happy coding!