**ISTE-121 Day22 – Networking with Object I/O**

**Object Transfer**

So far, we’ve transmitted simple data types like byte, int, long, and UTF (Strings) using DataInputStream and DataOutputStream. Now, we’re going to look at how we can send objects across the network using ObjectInputStream and ObjectOutputStream.

We’ll be modifying TCPClientData.java and TCPServerData.java, in today’s downloads, so that the client sends a file called test.txt (also in today’s downloads) to the server as a File object. When you look at the JavaDocs for ObjectInputStream and ObjectOutputStream, you’ll find that these classes not only have the ability to transfer basic data types the same way as DataInputStream and DataOutputStream, but also entire objects at once by using readObject and writeObject.

Before you get too far, rename TCPClientData to TCPClientObject and TCPServerData to TCPServerObject.

**Fair Warning about Object Transfer**

It turns out that when you open a Socket’s input stream as an ObjectInputStream, Java reads ahead right away, without waiting, to see if the stream is at EOF (the stream equivalent of being closed). If there is no input on that stream, your program will block indefinitely.

If the process at the other end of the Socket that has already opened its ObjectOutputStream, this blocking does not occur, as there is a short “handshake” between the two processes to avoid this.

What does this mean for you? When using Object I/O over Sockets, always open the ObjectOutputStream first, then the ObjectInputStream. If both parties do this, it guarantees that nobody gets stuck waiting for the other.

out = new ObjectOutputStream(socket.getOutputStream());  
in = new ObjectInputStream(socket.getInputStream());

One final warning: each time you do a write, whether it be writeInt, writeUTF, writeObject, or something else, make sure you **flush** the output stream. If you don’t do this, the other party waiting for data will continue to block indefinitely.

**Modifying the Client and Server**

Recall that TCPClientData and TCPServerData did a simple data transfer as follows:

1. Client sends the size of a file called test.txt to the server.
2. Server sends this file size back to the client.
3. Client sends the file byte by byte to the server.
4. Server responds to the client with a string that says “DONE”.
5. Server closes the connection.

Your task today is to modify the client and server programs to use Object I/O and to add some additional functionality to the client and server to do the following:

1. Client sends the size of a file called test.txt to the server.
2. Server sends this file size back to the client.
3. Client sends a File object to the server.
4. Server will print out the size of the File object to its log and write the File object byte by byte to a file called output.txt.
5. Server responds to the client with a string that says “DONE”.
6. Client will send an object of type NameTag to the server.
7. Server will print out the contents of the NameTag object to its log.
8. Server responds to the client with a string that says “DONE”.
9. Server closes the connection.

To properly implement and test this code, you will need to make gradual changes to the client and server. Use lots of log messages to show your progress!

For an Object of any type to be transferred using the Object I/O libraries, the Object must be Serializable. That is, the Object class must implement Serializable. Notice in the JavaDocs that File does implement this, so it can be transferred using these libraries already.

To give you some practice with Serializable objects, you’ll create a class called NameTag. It will have String members to represent a person’s first name, last name, home town, home state, and email address as well as methods to access those members. NameTag should also have a constructor that allows the calling method to pass all of this data directly to the class upon object creation.

import java.io.Serializable; // make sure to do this first!

When receiving an Object, it is good practice to determine what type of Object it is before attempting to use it. The following code snippets are quite valuable.

Object obj = in.readObject();

if( obj instanceof File )

File rcvFile = (File) obj;

The first line stores the object received on an ObjectInputStream called “in” into a generic Object, while the second line is an if statement to determine if the expected type of Object was received. In this case, a File object. To then store the Object for future use, the third line converts the generic Object to a File object using casting.

If the Object received is not of the expected type (a transfer error likely occurred), it’s a good idea to throw an exception.

throw new Exception();