Serialization and Sockets

"Serialization" refers to reading and writing between objects in memory and persistent storage, such as a file. The earlier XML read/write code example is another instance of this strategy.

Serialization / Archiving

- State in memory -- objects
- Write objects to streamed state
  - To a disk file, or across the network, or to the system clipboard
  - The notion of "address space" does not hold in the streamed form -- there are no pointers. Just a block of bytes.
- Read
  - Read the streamed form, and re-create the object in memory
- There are many words for writing an object to a streamed form
  - Writing
  - Persisting
  - Pickling
  - Flattening
  - Streaming
  - Dehydrate (Rehydrate = read)
  - Archiving

Old Style Memory/Disk Streaming

- Translate back and forth by hand
- Typically use an ASCII text format
  - Custom arrangement between your data structures and some ASCII format for reading and writing.
  - Write custom code to read and write between the memory form and the streamed form
  - e.g. DBRecord

Java Automatic Serialization

- Serializable interface
  - By implementing this interface, a class declares that it is willing to be read/written by the automatic serialization machinery.
  - This use of an interface is a bit of a hack, but it works. It marks which classes participate. (In the future, this sort of thing will be done with Java 5 annotations -- attach extra information to a class.)
  - Serialization is not the most efficient in terms of cpu cost or space, but it is very easy.
- Automatic Writing
  - The system knows how to recursively write out the state of an object
  - Recursively follows pointers and writes those objects out too (they must in turn be serializable). In this way, it writes out the whole tree of objects.
- Built-Ins
  - Most built ins know how to serialize: int, array, Point, ...
- Fields declared with the "transient" modifier are skipped by serialization
- Use the "transient" reserved word on an instance variable to prevent the serialization from recurring down a branch you do not want written out. A transient ivar comes back as null after reading.
- Override: readObject(), writeObject() -- to put in more customized reading-writing
• Versioning of the class
  - Serialization can detect version changes to the class when reading and refuse to read. Programmer can control this if they wish.
  - This make serialization fragile without care -- data written with class code version N will fail to read back into the class version N+1 by default.

Strategy -- Data Struct
• Create a little public struct class that contains plain data you want to send
• Make it serializable
• Use object streams below to read and write it easily

ObjectOutputStream
• Create an object output stream wrapped around any other stream. Then can write objects onto that stream.
• e.g. out = new ObjectOutputStream( <regular file or socket output stream> );
• This is the "decorator" pattern -- wrapping something of interface X in another thing, also of interface X

out.writeObject(obj)
• Suppose "out" is an ObjectOutputStream
• out.writeObject(obj);
• This one line calls the automatic serialization machinery to write out everything rooted at the given object.
• Classes
  - Each written object will be identified by its class -- the reading code will need those same classes to read the stream.
  - The written class should be public (so the receiver can know it)
  - The written class should not be inner, since that will try to write the outer object too. It can be a nested (static) class however.
• Array
  - For a collection of things, it may be easier to arrange all the things into a single array that can be written in one operation.
• Transient
  - Fields should be declared transient if they should not be written. They will read back in as null.
• MVC
  - Write out the data model, not the view.
  - May also want to write out a simplified or canonical version of the data model -- so you can revise your real internal data model over time, without breaking file compatibility.

No Duplicates / Automatic Detection
• The automatic serialization detects duplicates in the stream -- objects that are written more than once. That is, a single object that is written multiple times is only recorded once in the stream.
• If a second or later instance of an object is written to the stream, a reference to the first instance is instead put in the stream, instead of a 2nd copy.
• The reading code uses this information to correctly re-create the pointer graph in memory.

out.writeUnshared(obj)
• writeUnshared(obj) -- writes out a fresh copy of the given object, even if that object was previously written to the stream. However, the no-duplicates property will still hold for objects referenced from inside the given object.

ObjectInputStream
• Create an ObjectInputStream wrapped around any type of stream
• ObjectInputStream in = new ObjectInputStream(<input stream>);

in.readObject()

• CT type
  - Read back with the same compile time type it was written (Object[] or String[])

• Class
  - If a class was written that is not present at read-time, there will be an error.
  - If the class has the same name but a changed implementation there will be an error.
  - It's safest to serialize classes that are stable everywhere such as Array and Point

Sockets

• Sockets make network connections between machines, but you just read/write/block on them like there were plain file streams. The Internet is basically built on sockets.

Client Socket

• Make connection to host name "127.0.0.1" or "localhost" (the local machine itself) or "elaine26.stanford.edu" (machine on the internet) + a port number on that machine.
  - Socket toServer = new Socket(host, port); // make connection
  - OutputStream out = toServer.getOutputStream(); // write to this
  - InputStream in = toServer.getInputStream; // read from this

• Reads will block if there is no data (do not do on swing thread!)
• Writes go through fast, so ok to do on swing thread (could fork off a thread to do it)
• Can wrap each stream in ObjectInputStream/ObjectOutputStream to send whole objects -- a low budget way to do network i/o without a lot of parsing

Server Sockets / accept()

• The server thread creates a sever socket and calls accept() to wait (block) for incoming client connections on a particular port number.
• On unix, ports under 1024 are "privileged" so regular users must use high port numbers, like 8000 or 3456.
• The accept() call blocks waiting for an incoming connection, and then returns a new socket, one for each incoming client. Typically you deal with the new connection, and then loop around and block in accept again.
• Get input and output streams, as above, for each client
• See the ServerAccepter example below.

Blocking / Flushing

• Reading on a socket when there is no data will block -- so you can't do that on the swing thread
• Likewise, the server blocks in accept(), waiting for new client connections
• Writing on a socket may "buffer" the data to send it all in a big chunk. Use flush() on a stream to force the accumulated data to go out now. When you close() on a stream when you are done with it, that does an implicit flush() to send all the data.

Ticker GUI/Socket Example

• Message -- a little struct that contains a Date and a String
• Server button -> accepts client connections. Starts a ServerAccepter thread.
• Server keeps a list of all the connections to clients -- sends messages to all of them.
• Client button -> connects to a server and listens for incoming messages, posts them to its GUI.
• Complete code available in hw directory

Ticker GUI/Socket Code

//TickerExample.java
/*
Demonstrates using client and server sockets with a GUI.
One server ticker can support any number of client tickers -- sortof a primitive, one-way instant messenger.
Uses serialization to send a little data struct object.
*/
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.*;
import java.util.*;
import java.io.*;
import java.net.*;

public class TickerExample extends JFrame {
    private JTextArea textArea;
    private JTextField field;
    private JLabel status;
    // The are thread inner classes to handle
    // the networking.
    private ClientHandler clientHandler;
    private ServerAccepter serverAccepter;
    // List of object streams to which we send data
    private java.util.List<ObjectOutputStream> outputs =
        new ArrayList<ObjectOutputStream>();

    public static void main(String[] args) {
        // Prefer the "native" look and feel.
        try {
            UIManager.setLookAndFeel(UIManager.
                getSystemLookAndFeelClassName());
        } catch (Exception ignored) { }
        for (int i=0 ;i<3; i++ ) { // for testing, handy to make a few at a time
            new TickerExample();
        }
    }

    public TickerExample() {
        setTitle("Ticker");
        JComponent box = new JPanel();
        box.setLayout(new BoxLayout(box, BoxLayout.Y_AXIS));
        box.add(new JScrollPane(textArea), BorderLayout.CENTER);

        JButton button;
        button = new JButton("Start Server");
        box.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doServer();
            }
        });

        button = new JButton("Start Client");
        box.add(button);
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                doClient();
            }
        });
    }
}
5

```java
field = new JTextField(15);
JPanel panel = new JPanel();
panel.setMinimumSize(new Dimension(200, 30));
panel.add(field);
box.add(panel);
field.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        doSend();
    }
});
status = new JLabel();
box.add(status);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
pack();
setVisible(true);
}
```

// Struct object just used for communication -- sent on the object stream.
// Declared "static", so does not contain a pointer to the outer object --
// that we don't also serialize the whole outer object.
// The contained String and Date objects are both Serializable, otherwise
// the serialization would fail.

```java
public static class Message implements Serializable {
    public String text;
    public Date date;
    public transient TickerExample ticker; // transient = do not send

    public Message(String text, Date date, TickerExample ticker) {
        this.text = text;
        this.date = date;
        this.ticker = ticker;
    }

    public String toString() {
        return "message: " + text;
    }
}
```

// Appends a message to the local GUI (must be on swing thread)
```java
public void sendLocal(Message message) {
    textArea.setText(textArea.getText() + message.text + "\n" + message.date + "\n\n" + message.ticker);
}
```

// Initiate message send -- send both local and remote (must be on swing thread)
// Wired to text field.
```java
public void doSend() {
    Message message = new Message(field.getText(), new Date(), this);
    sendLocal(message);
    sendRemote(message);
    field.setText("");
}
```

// Client runs this to handle incoming messages
// (our client only uses the inputstream of the connection)
```java
private class ClientHandler extends Thread {
    private String name;
    private int port;

    ClientHandler(String name, int port) {
        this.name = name;
        this.port = port;
    }

    // Connect to the server, loop getting messages
    public void run() {
        try {
            // make connection to the server name/port
            Socket toServer = new Socket(name, port);
        }
    }
}
```
// get input stream to read from server and wrap in object input stream
ObjectInputStream in = new ObjectInputStream(toServer.getInputStream());
System.out.println("client: connected!");

// we could do this if we wanted to write to server in addition
// to reading
// out = new ObjectOutputStream(toServer.getOutputStream());

while (true) {
    // get object from server; blocks until object arrives.
    Message message = (Message) in.readObject();
    System.out.println("client: read " + message);
    // note message.ticker is null, since "transient"
    invokeToGUI(message);
}

catch (Exception ex) { // IOException and ClassNotFoundException
    ex.printStackTrace();
}

// Could null out client ptr.
// Note that exception breaks out of the while loop,
// thus ending the thread.

// Given a message, puts that message in the local GUI.
// Can be called by any thread.
public void invokeToGUI(Message message) {
    final Message temp = message;
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            status.setText("Client receive");
            sendLocal(temp);
        }
    });
}

// Sends a message to all of the outgoing streams.
// Writing rarely blocks, so doing this on the swing thread is ok,
// although could fork off a worker to do it.
public synchronized void sendRemote(Message message) {
    status.setText("Server send");
    System.out.println("server: send " + message);
    Iterator<ObjectOutputStream> it = outputs.iterator();
    while (it.hasNext()) {
        ObjectOutputStream out = it.next();
        try {
            out.writeUnshared(message);
            out.flush();
            // writeUnshared() is like writeObject(), but always writes
            // a new copy of the object. The flush (optional) forces the
            // bytes out right now.
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

// Adds an object stream to the list of outputs
// (this and sendToOutputs() are synchronized to avoid conflicts)
public synchronized void addOutput(ObjectOutputStream out) {
    outputs.add(out);
}

// Server thread accepts incoming client connections
class ServerAccepter extends Thread {
    private int port;
    ServerAccepter(int port) {
        this.port = port;
    }

    public void run() {
        try {
            ServerSocket serverSocket = new ServerSocket(port);
            while (true) {
                Socket toClient = null;
                // this blocks, waiting for a Socket to the client
                toClient = serverSocket.accept();
                System.out.println("server: got client");
                // Get an output stream to the client, and add it to
                // the list of outputs
                // (our server only uses the output stream of the connection)
                addOutput(new ObjectOutputStream(toClient.getOutputStream()));
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}

// Starts the server accepter to catch incoming client connections.
// Wired to Server button.
public void doServer() {
    status.setText("Start server");
    String result = JOptionPane.showInputDialog("Run server on port", "8001");
    if (result!=null) {
        System.out.println("server: start");
        serverAccepter = new ServerAccepter(Integer.parseInt(result.trim()));
        serverAccepter.start();
    }
}

// Runs a client handler to connect to a server.
// Wired to Client button.
public void doClient() {
    status.setText("Start client");
    String result = JOptionPane.showInputDialog("Connect to host:port", "127.0.0.1:8001");
    if (result!=null) {
        String[] parts = result.split(":");
        System.out.println("client: start");
        clientHandler = new ClientHandler(parts[0].trim(), Integer.parseInt(parts[1].trim()));
        clientHandler.start();
    }
}