CSC2620 Object Oriented Design — Spring 2024
Unit 9 – Socket Programming

9.1 Sockets

- To utilize the network you will be required to use the *Socket API*.
  - A socket is an abstraction of the network that allows a process to send and receive messages from a process on another host.
  - A socket allows for *peer-to-peer* process communication, or server-client communication.
  - Sockets are used by Java programs to communicate with programs running in a different JRE.
- Java socket programs can be connected (so they are in constant communication) or connectionless (where two programs check mutual resources periodically to communicate)...we're going to concentrate on connected sockets
- The `Server` and `ServerSocket` classes allow for socket connections
- Clients and servers are two types of machines in a network...they communicate back and forth and have different purposes
  - The client is usually the program run by the end user
  - The server is the program that sends information to the clients and controls communication between them
- To write a server application two sockets are needed.
  1. A `ServerSocket` which waits for the client requests (when a client makes a new `Socket()`)
  2. A plain old `Socket` to use for communication with the client

9.2 Object Streams and Sockets

- **Object Streams** support the passing of objects as input/output, just like any other I/O stream.
- The object stream classes are `ObjectInputStream` and `ObjectOutputStream`. These classes implement `ObjectInput` and `ObjectOutput`, which are subinterfaces of `DataInput` and `DataOutput`.
- Object Streams are heavyweight and unnecessary given JSON, but you may see them in the wild.

Review Server-Client and Server-Client-Threaded examples.

9.3 JSON with Sockets

Now let's look at how to use **JSON** (JavaScript Object Notation) with Java Sockets. JSON is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. Java Sockets are a mechanism for communicating between two endpoints over a network. By combining these two technologies, we can create powerful networked applications that exchange data in a standardized format.

9.3.1 JSON Basics

Before we dive into using JSON with Java Sockets, let's review some basics of JSON.

JSON is a text format that uses a simple syntax to represent data as a collection of key-value pairs. For example, the following JSON object represents a person's information:

```
{
  "name": {
     "first": "John",
     "second": "Smith"
  }
  "emails": [
    "john.smith@example.com",
    "jsmith@dextrous.com",
    "smithj@merrimack.edu"
  ],
  "age": 30
}
```

For more information, look here.

Here, the keys are strings and the values can be of various types such as strings, numbers, or even other JSON objects or arrays.

9.4 Using JSON with Java Sockets

Now that we've reviewed the basics of JSON and Java Sockets, let's see how we can use them together.

The general idea is that we will send and receive JSON-encoded data over the network using the input and output streams of the `Socket` object.

9.4.1 Encoding and Decoding JSON

To encode and decode JSON in Java, we can use a library such as Jackson or Gson. These libraries provide convenient methods for converting between JSON and Java objects.

Here's an example of how to encode a Java object as JSON using Jackson:

```
ObjectMapper mapper = new ObjectMapper();
MyObject obj = new MyObject();
byte[] jsonBytes = mapper.writeValueAsBytes(obj);
```

Here, `MyObject` is a Java class that we want to encode as JSON. The `ObjectMapper` class provides methods for encoding and decoding JSON. We create a new instance of `MyObject`, then use the `writeValueAsBytes` method to encode it as a byte array.

To decode JSON back into a Java object, we can use the `readValue` method:

```
MyObject obj = mapper.readValue(jsonBytes, MyObject.class);
```

Here, `jsonBytes` is a byte array containing JSON-encoded data. The `readValue` method converts this data back into a Java object of the specified class.

9.5 Sending and Receiving JSON

To send JSON over a Java Socket, we first encode the Java object as JSON, then send the resulting byte array over the network using the `OutputStream` of the `Socket`.

Here's an example of how to send a JSON-encoded object over a Java Socket:

```
ObjectMapper mapper = new ObjectMapper();
MyObject obj = new MyObject();
byte[] jsonBytes = mapper.writeValueAsBytes(obj);
OutputStream out = socket.getOutputStream();
out.write(jsonBytes);
```

Here, we're creating a new instance of `MyObject`, encoding it as JSON using Jackson, then writing the resulting byte array to the `OutputStream` of the `Socket`.

To receive JSON over a Java Socket, we first read data from the `InputStream` of the `Socket` into a byte array, then decode the byte array back into a Java object using Jackson.

Here's an example of how to receive a JSON-encoded object over a Java Socket:

```
InputStream in = socket.getInputStream();
byte[] buffer = new byte[1024];
int bytesRead = in.read(buffer);
byte[] jsonBytes = Arrays.copyOf(buffer, bytesRead);
MyObject obj = mapper.readValue(jsonBytes, MyObject.class);
```

Here, we're reading up to 1024 bytes from the `InputStream` of the `Socket` into a byte array called `buffer`. We then create a new byte array called `jsonBytes` containing only the bytes that were actually read.

We then decode `jsonBytes` back into a Java object using Jackson.

9.6 Practice

1. Design and create a Tic-Tac-Toe game played over a network governed by a server.
   (a) Hint: Start with the `ServerClientThreadExample` in `merrimack_cs_examples`, and instead of a chat window, create a canvas or a text area for an ASCII version of the TTT board
   (b) Create a class design (using UML) for the game.
   (c) Have a centralized server that collects the moves made by the players, and pushes them out to both clients (or have a client make its own move, and push that out to the other client, like in the chat example)