Yuwen Sang
COMP 2355
March, 16th, 2020
Instructor: Professor Daniel Stevenson

Notice:

There are some details that the instruction does not speak clearly, but I think all the functions that the instruction clearly required has been achieved in this project. You can run the program first, then when you read code, if you find some code is hard to understand, then refer to this cover sheet (since I know this is a long cover sheet…)

Program 3 Socket Painter Cover Sheet

This program can successfully run all functions, including the bonus point (making the drawing process more realistic by using *MouseMotionListener*). Please run **Hub class** first, and then run **Painter class** any times you want. Feel free to close any window that you want to exit, and feel free to run a new Painter any time. The functions are described as follows:

1. Hub Class:

Hub class process and Exchange all painters' information. It also save the historical information and create a painter to the new participated painter. The class uses two hashtables to save painter's socket and account information separately. Both hashtables use name as key. The structure of Hub class described as follows:

- **void paintingSharing(PaintingPrimitive newPrimitve, Socket self){}** & **void textSharing(String str, Socket self){}**

  These two methods sharing new painting primitive/text message received from some painter to other painters. The two methods have the same logic. They go through all online painter's socket address and account information saved in the hashtables and using ObjectOutputStream to send newPrimitve to every online painter.

- **void startServerSocket(){}**

  This method create ServerSocket and using while(true) to make it always waiting new socket connect into the hub. When it accept a new client (painter), it will generate a new **HubSerivce** (the next class) thread for the new painter. This method used in the main method.

- **private class HubService implements Runnable{}**

  This is the thread part of the Hub class. The thread keeps running until the painter offline (close the window). **HubService constructor** used to create a new client account which saving the client's name, ObjectOutputStream and ObjectInputStream information. Then it send the current painting primitive ArrayList to the new Painter. Then the constructor also added the new client's socket and account information into the two hashtables.

  Override **run()** always keep opening by using while(ture) to wait receiving information from its painter. When painter send information to this thread, it will identify the type of the information and decide use **paintingSharing()** or **textSharing()** to share the received information to other online painters. Also, it will save the new painting primitive into the ArrayList pntpmts so when a new painter come to the hub, the hub will send this updated ArrayList to the new painter.

  When this thread's client exits (closes the window), run() will tell other painters that this painter leaves the hub, and also close the related socket, oos, ois, and also remove this painter's information from the two hashtables. Then this thread will die.

The **run()** method has been synchronized to avoid race condition.

- **void addPrimitive(){}**

  A simple method to add the new painting primitive into the hub's PaintingPrimitive ArrayList.

2. **Painter Class:**

   The client class that used to generate painting board and other structures of the window. This class also communicate with hub class to get other painters' sending information. There are two button, "Send Text" and "Send Image". This is used to avoid send text and image at the same time (though I can do that since the hub will use **interfaceof** to identify the information type). I just think this looks good to make different buttons represent different functions.

   When a new painter connected to the hub, the painter will also generate a thread (see private class **Receiver**) to receive information from the hub. The painter will send their painting/text when they click related button, so the *writeobject()* method was call under private **MyActionListener** class which implements ActionListener.

   Receiver thread always run until the user close the window (exit). While WindowListener (under Painter constructor) listened that the window closed, the thread will close and the painter's socke, oos&ois will also closed. Instead of show an IOException, the console will show the information that the painter successfully exit and appreciate the painter's use. The **run()** method has been synchronized, to avoid race condition.

   Notice: *the painter can only draw one shape at each time.* If the painter does not send the shape he/she created and start draw a new shape, the old shape will not be saved anywhere and just disappeared. I can create a PaintingPrimitive ArrayList to allow painters draw multiple shapes and then send them together to other painters' the board, but I don't do that since the instruction does not require me do it.

   Notice2: *sometimes the new coming painter's chatting board may disappear*. I think this is the JFrame's format problem. If you maximize the new painter's window, you will see the chatting board shows up. Then you can unmaximize the window to see that the chatting board will appear.

3. **PaintingPrimitive, Circle, Line Class:**

   These classes are written based on canvas instruction. **Circle** and **Line** are subclasses of the **PaintingPrimitive** class. The structure is easy to understand. You can also create new shape class as **PaintingPrimitive**'s new child class.

4. **PaintingPanel Class:**

   This class used to create center canvas. The class written based on canvas instruction.

5. **Account Class:**

During my programming, it is easy to report StreamCorruptedException: invalid type code: AC error, which caused by try to build two ObejctOutputStream (or ObjectInputStream) for one socket. To avoid this Exception, I build a new Account for each client and saved the built ObjectOutputStream & ObejctInputStream (oos & ois) into it. When I want to write or read something, I can directly use each client's oos/ois by using getOOS()/getOIS() methods. *Notice that Hub and Painter have different account for a same painter.* When a new painter come in, the Hub class would create an account for this painter that include the painter's name, and the ois&oos that used to read from/write to the painter. This painter would also have a local account that has this painter's name, local ois&oos to read from/write to the hub.