# CS360 Lecture 13
# Files and Networking

Tuesday, March 16, 2004

## *Reading*

Files and Streams: Chapter 17
Networking: Chapter 18

## Circular Buffer

The `synchronized` keyword that comes before a method means that the object is locked so no other `synchronized` method can run on that object at the same time.

```
public interface Buffer {
   public void set( int value );  // place value into Buffer
   public int get();              // return value from Buffer
}

import javax.swing.*;

public class CircularBuffer implements Buffer {

   private int buffers[] = { -1, -1, -1 };
   private int occupiedBufferCount = 0;
   private int readLocation = 0, writeLocation = 0;
   private JTextArea outputArea;

   public CircularBuffer( JTextArea output )
   {
      outputArea = output;
   }

   public synchronized void set( int value )
   {
      String name = Thread.currentThread().getName();

      while ( occupiedBufferCount == buffers.length ) {

         try {
            SwingUtilities.invokeLater( new RunnableOutput( outputArea,
               "\nAll buffers full. " + name + " waits." ) );
            wait();
         }

         catch ( InterruptedException exception )
         {
            exception.printStackTrace();
         }
      } // end while

      buffers[ writeLocation ] = value;
```

```
      SwingUtilities.invokeLater( new RunnableOutput( outputArea,
         "\n" + name + " writes " + buffers[ writeLocation ] + " ") );

      ++occupiedBufferCount;

      writeLocation = ( writeLocation + 1 ) % buffers.length;

      SwingUtilities.invokeLater( new RunnableOutput(
         outputArea, createStateOutput() ) );

      notify(); // return waiting thread (if there is one) to ready state

   } // end method set

   public synchronized int get()
   {
      String name = Thread.currentThread().getName();

      while ( occupiedBufferCount == 0 ) {

         try {
            SwingUtilities.invokeLater( new RunnableOutput( outputArea,
               "\nAll buffers empty. " + name + " waits.") );
            wait();
         }

         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }

      } // end while

      int readValue = buffers[ readLocation ];

      SwingUtilities.invokeLater( new RunnableOutput( outputArea,
         "\n" + name + " reads " + readValue + " ") );

      --occupiedBufferCount;

      readLocation = ( readLocation + 1 ) % buffers.length;

      SwingUtilities.invokeLater( new RunnableOutput(
         outputArea, createStateOutput() ) );

      notify(); // return waiting thread (if there is one) to ready state

      return readValue;

   } // end method get


   public String createStateOutput()
```

```
   {
      String output =
         "(buffers occupied: " + occupiedBufferCount + ")\nbuffers: ";

      for ( int i = 0; i < buffers.length; i++ )
         output += " " + buffers[ i ] + "  ";

      output += "\n         ";

      for ( int i = 0; i < buffers.length; i++ )
         output += "---- ";

      output += "\n         ";

      for ( int i = 0; i < buffers.length; i++ )

         if ( i == writeLocation && writeLocation == readLocation )
            output += " WR  ";
         else if ( i == writeLocation )
            output += " W   ";
         else if ( i == readLocation )
            output += "  R  ";
         else
            output += "     ";

      output += "\n";

      return output;

   } // end method createStateOutput
} // end class CircularBuffer

import javax.swing.*;

public class Producer extends Thread {
   private Buffer sharedLocation;
   private JTextArea outputArea;

   public Producer( Buffer shared, JTextArea output )
   {
      super( "Producer" );
      sharedLocation = shared;
      outputArea = output;
   }

   public void run()
   {
      for ( int count = 11; count <= 20; count ++ ) {
         try {
            Thread.sleep( ( int ) ( Math.random() * 3000 ) );
            sharedLocation.set( count );
         }
         catch ( InterruptedException exception ) {
```

```
            exception.printStackTrace();
         }
      }

      String name = getName();
      SwingUtilities.invokeLater( new RunnableOutput( outputArea, "\n" +
         name + " done producing.\n" + name + " terminated.\n" ) );

   } // end method run
} // end class Producer

import javax.swing.*;

public class Consumer extends Thread {
   private Buffer sharedLocation; // reference to shared object
   private JTextArea outputArea;

   public Consumer( Buffer shared, JTextArea output )
   {
      super( "Consumer" );
      sharedLocation = shared;
      outputArea = output;
   }

   public void run()
   {
      int sum = 0;

      for ( int count = 1; count <= 10; count++ ) {

         try {
            Thread.sleep( ( int ) ( Math.random() * 3001 ) );
            sum += sharedLocation.get();
         }

         catch ( InterruptedException exception ) {
            exception.printStackTrace();
         }
      }

      String name = getName();
      SwingUtilities.invokeLater( new RunnableOutput( outputArea,
         "\nTotal " + name + " consumed: " + sum + ".\n" +
         name + " terminated.\n ") );

   } // end method run
} // end class Consumer

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class CircularBufferTest extends JFrame {
```

```
   JTextArea outputArea;

   public CircularBufferTest()
   {
      super( "Demonstrating Thread Synchronizaton" );

      outputArea = new JTextArea( 20,30 );
      outputArea.setFont( new Font( "Monospaced", Font.PLAIN, 12 ) );
      getContentPane().add( new JScrollPane( outputArea ) );

      setSize( 310, 500 );
      setVisible( true );

      CircularBuffer sharedLocation = new CircularBuffer( outputArea );

      SwingUtilities.invokeLater( new RunnableOutput( outputArea,
         sharedLocation.createStateOutput() ) );

      Producer producer = new Producer( sharedLocation, outputArea );
      Consumer consumer = new Consumer( sharedLocation, outputArea );

      producer.start();  // start producer thread
      consumer.start();  // start consumer thread

   } // end constructor

   public static void main ( String args[] )
   {
      CircularBufferTest application = new CircularBufferTest();
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
   }

} // end class CirclularBufferTest

import javax.swing.*;

public class RunnableOutput implements Runnable {
   private JTextArea outputArea;
   private String messageToAppend;

   public RunnableOutput( JTextArea output, String message )
   {
      outputArea = output;
      messageToAppend = message;
   }

   public void run()
   {
      outputArea.append( messageToAppend );
   }

} // end class RunnableOutput
```

## File Processing

Java programs perform file processing by using classes from the Java package java.io.

Namely:

- `FileInputStream`: byte-based input from file
- `FileOutputStream`: byte-based output to a file
- `FileReader`: character-based input from file
- `FileWriter`: character-based output to a file

```
try
{
  BufferedReader input = new BufferedReader(
                              new FileReader( "Questions.txt" ));
  String text;
  text = input.readLine()
  catch( IOException e )
  {
    e.printStackTrace();
  }
}
```

## Java Networking

Java's networking capabilities are grouped into several packages, but the most important and fundamental one is java.net.

The simplest part of Java networking is socket-based communications, which enable applications to view networking as if it were file input/output. A program can read from a socket or write to a socket.

In networking there can be a client-server relationship. The client requests an action that can be performed and the server performs the action and responds to the client.

### *Creating a Java Server*

To create a Java server you must follow these five steps:

1. Create a ServerSocket object
   ServerSocket server = new ServerSocket( port, queueLength );
   The port number is bound to the application and the clients use this port number to connect with the server.
   The port number could be between 0-65,535 and should be above 1024.
   queueLength is the maximum number of clients that can wait to connect to the server.
2. Server needs to listen for client connections:
   Socket connection = server.accept();
3. Get the OutputStream and InputStream objects that enables the server to communicate with the client by sending and receiving bytes.
   ObjectInputStream input =
   new ObjectInputStream( connection.getInputStream() );
   ObjectOutputStream output =
   new ObjectOutputStream( connection.getOutputStream() );
4. This is the processing phase where the client and server communicate.
5. Closing the streams and the socket.

## *Creating a Java Client*

1. Create a Socket to connect to the server:
   Socket connection = new Socket( serverAddress, port );
   serverAddress is the server's IP address
   If a connection does not occur then a subclass of IOException is thrown
2. getInputStream and getOutputStream are used to obtain references to the Socket's InputStream and OutputStream.
3. This is the processing phase.
4. Close the streams and socket.

## *Client/Server Chat Application*

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Server extends JFrame {
   private JTextField enterField;
   private JTextArea displayArea;
   private ObjectOutputStream output;
   private ObjectInputStream input;
   private ServerSocket server;
   private Socket connection;
   private int counter = 1;

   public Server()
   {
      super( "Server" );

      Container container = getContentPane();

      enterField = new JTextField();
      enterField.setEditable( false );
      enterField.addActionListener(
         new ActionListener() {

            public void actionPerformed( ActionEvent event )
            {
               sendData( event.getActionCommand() );
               enterField.setText( "" );
            }
         }
      );

      container.add( enterField, BorderLayout.NORTH );

      displayArea = new JTextArea();
      container.add( new JScrollPane( displayArea ),
         BorderLayout.CENTER );

      setSize( 300, 150 );
      setVisible( true );
```

```
} // end Server constructor

public void runServer()
{
   try {

      server = new ServerSocket( 12345, 100 );

      while ( true ) {

         try {
            waitForConnection(); // Step 2: Wait for a connection.
            getStreams();        // Step 3: Get input & output streams.
            processConnection(); // Step 4: Process connection.
         }

         catch ( EOFException eofException ) {
            System.err.println( "Server terminated connection" );
         }

         finally {
            closeConnection();   // Step 5: Close connection.
            ++counter;
         }

      } // end while
   } // end try

   catch ( IOException ioException ) {
      ioException.printStackTrace();
   }

} // end method runServer

private void waitForConnection() throws IOException
{
   displayMessage( "Waiting for connection\n" );
   connection = server.accept(); // allow server to accept connection
   displayMessage( "Connection " + counter + " received from: " +
      connection.getInetAddress().getHostName() );
}

private void getStreams() throws IOException
{
   output = new ObjectOutputStream( connection.getOutputStream() );
   output.flush(); // flush output buffer to send header information

   input = new ObjectInputStream( connection.getInputStream() );

   displayMessage( "\nGot I/O streams\n" );
}
```

```
private void processConnection() throws IOException
{
   String message = "Connection successful";
   sendData( message );

   setTextFieldEditable( true );

   do { // process messages sent from client

      try {
         message = ( String ) input.readObject();
         displayMessage( "\n" + message );
      }

      catch ( ClassNotFoundException classNotFoundException ) {
         displayMessage( "\nUnknown object type received" );
      }

   } while ( !message.equals( "CLIENT>>> TERMINATE" ) );

} // end method processConnection

private void closeConnection()
{
   displayMessage( "\nTerminating connection\n" );
   setTextFieldEditable( false ); // disable enterField

   try {
      output.close();
      input.close();
      connection.close();
   }
   catch( IOException ioException ) {
      ioException.printStackTrace();
   }
}

private void sendData( String message )
{
   try {
      output.writeObject( "SERVER>>> " + message );
      output.flush();
      displayMessage( "\nSERVER>>> " + message );
   }

   catch ( IOException ioException ) {
      displayArea.append( "\nError writing object" );
   }
}

private void displayMessage( final String messageToDisplay )
{
   SwingUtilities.invokeLater(
```

```
         new Runnable() {  // inner class to ensure GUI updates properly

            public void run() // updates displayArea
            {
               displayArea.append( messageToDisplay );
               displayArea.setCaretPosition(
                  displayArea.getText().length() );
            }

         }  // end inner class

      ); // end call to SwingUtilities.invokeLater
   }

   private void setTextFieldEditable( final boolean editable )
   {
      SwingUtilities.invokeLater(
         new Runnable() {  // inner class to ensure GUI updates properly

            public void run()  // sets enterField's editability
            {
               enterField.setEditable( editable );
            }

         }  // end inner class

      ); // end call to SwingUtilities.invokeLater
   }

   public static void main( String args[] )
   {
      Server application = new Server();
      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.runServer();
   }

}  // end class Server

import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Client extends JFrame {
   private JTextField enterField;
   private JTextArea displayArea;
   private ObjectOutputStream output;
   private ObjectInputStream input;
   private String message = "";
   private String chatServer;
   private Socket client;
```

```
public Client( String host )
{
   super( "Client" );

   chatServer = host; // set server to which this client connects

   Container container = getContentPane();

   enterField = new JTextField();
   enterField.setEditable( false );
   enterField.addActionListener(
      new ActionListener() {

         public void actionPerformed( ActionEvent event )
         {
            sendData( event.getActionCommand() );
            enterField.setText( "" );
         }
      }
   );

   container.add( enterField, BorderLayout.NORTH );

   displayArea = new JTextArea();
   container.add( new JScrollPane( displayArea ),
      BorderLayout.CENTER );

   setSize( 300, 150 );
   setVisible( true );

} // end Client constructor

private void runClient()
{
   try {
      connectToServer(); // Step 1: Create a Socket to make connection
      getStreams();      // Step 2: Get the input and output streams
      processConnection(); // Step 3: Process connection
   }

   catch ( EOFException eofException ) {
      System.err.println( "Client terminated connection" );
   }

   catch ( IOException ioException ) {
      ioException.printStackTrace();
   }

   finally {
      closeConnection(); // Step 4: Close connection
   }

} // end method runClient
```

```
private void connectToServer() throws IOException
{
   displayMessage( "Attempting connection\n" );

   client = new Socket( InetAddress.getByName( chatServer ), 12345 );

   displayMessage( "Connected to: " +
      client.getInetAddress().getHostName() );
}

private void getStreams() throws IOException
{
   output = new ObjectOutputStream( client.getOutputStream() );
   output.flush(); // flush output buffer to send header information

   input = new ObjectInputStream( client.getInputStream() );

   displayMessage( "\nGot I/O streams\n" );
}

private void processConnection() throws IOException
{
   setTextFieldEditable( true );

   do { // process messages sent from server

      try {
         message = ( String ) input.readObject();
         displayMessage( "\n" + message );
      }

      catch ( ClassNotFoundException classNotFoundException ) {
         displayMessage( "\nUnknown object type received" );
      }

   } while ( !message.equals( "SERVER>>> TERMINATE" ) );

} // end method processConnection

private void closeConnection()
{
   displayMessage( "\nClosing connection" );
   setTextFieldEditable( false ); // disable enterField

   try {
      output.close();
      input.close();
      client.close();
   }
   catch( IOException ioException ) {
      ioException.printStackTrace();
   }
```

```
}

private void sendData( String message )
{
   try {
      output.writeObject( "CLIENT>>> " + message );
      output.flush();
      displayMessage( "\nCLIENT>>> " + message );
   }

   catch ( IOException ioException ) {
      displayArea.append( "\nError writing object" );
   }
}

private void displayMessage( final String messageToDisplay )
{
   SwingUtilities.invokeLater(
      new Runnable() {  // inner class to ensure GUI updates properly

         public void run() // updates displayArea
         {
            displayArea.append( messageToDisplay );
            displayArea.setCaretPosition(
               displayArea.getText().length() );
         }

      }  // end inner class

   ); // end call to SwingUtilities.invokeLater
}

private void setTextFieldEditable( final boolean editable )
{
   SwingUtilities.invokeLater(
      new Runnable() {  // inner class to ensure GUI updates properly

         public void run()  // sets enterField's editability
         {
            enterField.setEditable( editable );
         }

      }  // end inner class

   ); // end call to SwingUtilities.invokeLater
}

public static void main( String args[] )
{
   Client application;

   if ( args.length == 0 )
      application = new Client( "127.0.0.1" );
```

```
      else
         application = new Client( args[ 0 ] );

      application.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
      application.runClient();
   }

} // end class Client
```