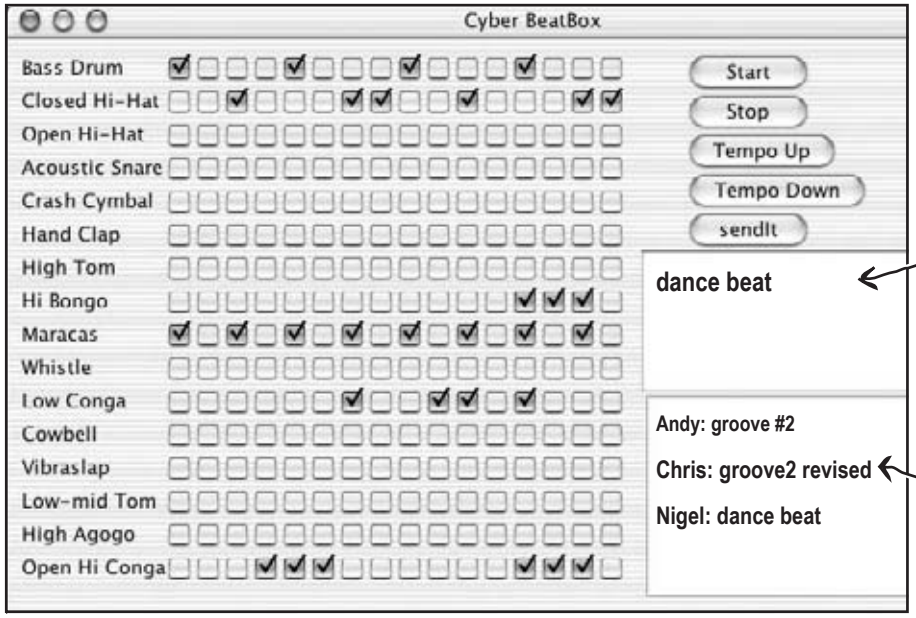
Appendix A:
Final Code Kitchen



Your message gets sent to the other players, along with your current beat pattern, when you hit "sendIt".

Incoming messages from players. Click one to load the pattern that goes with it, and then click 'Start' to play it.

Finally, the complete version of the BeatBox!

It connects to a simple MusicServer so that you can send and receive beat patterns with other clients.

Final BeatBox client program

Most of this code is the same as the code from the CodeKitchens in the previous chapters, so we don't annotate the whole thing again. The new parts include:

GUI - two new components are added for the text area that displays incoming messages (actually a scrolling list) and the text field.

NETWORKING - just like the SimpleChatClient in this chapter, the BeatBox now connects to the server and gets an input and output stream.

THREADS - again, just like the SimpleChatClient, we start a 'reader' class that keeps looking for incoming messages from the server. But instead of just text, the messages coming in include TWO objects: the String message and the serialized ArrayList (the thing that holds the state of all the checkboxes.)

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

public class BeatBoxFinal {

      JFrame theFrame;
      JPanel mainPanel;
      JList incomingList;
      JTextField userMessage;
      ArrayList<JCheckBox> checkboxList;
      int nextNum;
      Vector<String> listVector = new Vector<String>();
      String userName;
      ObjectOutputStream out;
      ObjectInputStream in;
      HashMap<String, boolean[]> otherSeqsMap = new HashMap<String, boolean[]>();

      Sequencer sequencer;
      Sequence sequence;
      Sequence mySequence = null;
      Track track;


      String[] instrumentNames = {"Bass Drum", "Closed Hi-Hat", "Open Hi-Hat","Acoustic
      Snare", "Crash Cymbal", "Hand Clap", "High Tom", "Hi Bongo", "Maracas", "Whistle",
      "Low Conga", "Cowbell", "Vibraslap", "Low-mid Tom", "High Agogo", "Open Hi Conga"};

      int[] instruments = {35,42,46,38,49,39,50,60,70,72,64,56,58,47,67,63};
```

```
public static void main (String[] args) {
   new BeatBoxFinal().startUp(args[0]);  // args[0] is your user ID/screen name
}

public void startUp(String name) {
    userName = name;
    // open connection to the server
    try {
      Socket sock = new Socket("127.0.0.1", 4242);
      out = new ObjectOutputStream(sock.getOutputStream());
      in = new ObjectInputStream(sock.getInputStream());
      Thread remote = new Thread(new RemoteReader());
      remote.start();
    } catch(Exception ex) {
        System.out.println("couldn't connect - you'll have to play alone.");
    }
    setUpMidi();
    buildGUI();
} // close startUp

 public void buildGUI() {

    theFrame = new JFrame("Cyber BeatBox");
    BorderLayout layout = new BorderLayout();
    JPanel background = new JPanel(layout);
    background.setBorder(BorderFactory.createEmptyBorder(10,10,10,10));

    checkboxList = new ArrayList<JCheckBox>();

    Box buttonBox = new Box(BoxLayout.Y_AXIS);
    JButton start = new JButton("Start");
    start.addActionListener(new MyStartListener());
    buttonBox.add(start);

    JButton stop = new JButton("Stop");
    stop.addActionListener(new MyStopListener());
    buttonBox.add(stop);

    JButton upTempo = new JButton("Tempo Up");
    upTempo.addActionListener(new MyUpTempoListener());
    buttonBox.add(upTempo);

    JButton downTempo = new JButton("Tempo Down");
    downTempo.addActionListener(new MyDownTempoListener());
    buttonBox.add(downTempo);

    JButton sendIt = new JButton("sendIt");
    sendIt.addActionListener(new MySendListener());
    buttonBox.add(sendIt);

    userMessage = new JTextField();
```

Add a command-line argument for your screen name.

Example: % java BeatBoxFinal theFlash

Nothing new... set up the networking, I/O, and make (and start) the reader thread.

GUI code, nothing new here

```
    buttonBox.add(userMessage);

    incomingList = new JList();
    incomingList.addListSelectionListener(new MyListSelectionListener());
    incomingList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    JScrollPane theList = new JScrollPane(incomingList);
    buttonBox.add(theList);
    incomingList.setListData(listVector); // no data to start with

    Box nameBox = new Box(BoxLayout.Y_AXIS);
    for (int i = 0; i < 16; i++) {
        nameBox.add(new Label(instrumentNames[i]));
    }

    background.add(BorderLayout.EAST, buttonBox);
    background.add(BorderLayout.WEST, nameBox);

    theFrame.getContentPane().add(background);
    GridLayout grid = new GridLayout(16,16);
    grid.setVgap(1);
    grid.setHgap(2);
    mainPanel = new JPanel(grid);
    background.add(BorderLayout.CENTER, mainPanel);

    for (int i = 0; i < 256; i++) {
        JCheckBox c = new JCheckBox();
        c.setSelected(false);
        checkboxList.add(c);
        mainPanel.add(c);
    } // end loop

    theFrame.setBounds(50,50,300,300);
    theFrame.pack();
    theFrame.setVisible(true);
  } // close buildGUI

  public void setUpMidi() {
     try {
     sequencer = MidiSystem.getSequencer();
     sequencer.open();
     sequence = new Sequence(Sequence.PPQ,4);
     track = sequence.createTrack();
     sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}

} // close setUpMidi
```

JList is a component we haven't used before. This is where the incoming messages are displayed. Only instead of a normal chat where you just LOOK at the messages, in this app you can SELECT a message from the list to load and play the attached beat pattern.

Nothing else on this page is new

Get the Sequencer, make a Sequence, and make a Track

```
public void buildTrackAndStart() {
   ArrayList<Integer> trackList = null; // this will hold the instruments for each
   sequence.deleteTrack(track);
   track = sequence.createTrack();

   for (int i = 0; i < 16; i++) {

     trackList = new ArrayList<Integer>();

     for (int j = 0; j < 16; j++) {
        JCheckBox jc = (JCheckBox) checkboxList.get(j + (16*i));
        if (jc.isSelected()) {
           int key = instruments[i];
           trackList.add(new Integer(key));
        } else {
          trackList.add(null);  // because this slot should be empty in the track
        }
     } // close inner loop
     makeTracks(trackList);
   } // close outer loop
   track.add(makeEvent(192,9,1,0,15)); // - so we always go to full 16 beats
   try {
      sequencer.setSequence(sequence);
      sequencer.setLoopCount(sequencer.LOOP_CONTINUOUSLY);
      sequencer.start();
      sequencer.setTempoInBPM(120);
  } catch(Exception e) {e.printStackTrace();}
 } // close method

public class MyStartListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      buildTrackAndStart();
   } // close actionPerformed
} // close inner class

public class MyStopListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      sequencer.stop();
  } // close actionPerformed
} // close inner class

public class MyUpTempoListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      float tempoFactor = sequencer.getTempoFactor();
      sequencer.setTempoFactor((float)(tempoFactor * 1.03));
  } // close actionPerformed
} // close inner class
```

Build a track by walking through the checkboxes to get their state, and mapping that to an instrument (and making the MidiEvent for it). This is pretty complex, but it is EXACTLY as it was in the previous chapters, so refer to previous CodeKitchens to get the full explanation again.

The GUI listeners. Exactly the same as the previous chapter's version.

```
 public class MyDownTempoListener implements ActionListener {
     public void actionPerformed(ActionEvent a) {
       float tempoFactor = sequencer.getTempoFactor();
       sequencer.setTempoFactor((float)(tempoFactor * .97));
     }
  }

  public class MySendListener implements ActionListener {
     public void actionPerformed(ActionEvent a) {
         // make an arraylist of just the STATE of the checkboxes
          boolean[] checkboxState = new boolean[256];
         for (int i = 0; i < 256; i++) {
            JCheckBox check = (JCheckBox) checkboxList.get(i);
            if (check.isSelected()) {
              checkboxState[i] = true;
            }
         } // close loop
         String messageToSend = null;
         try {
            out.writeObject(userName + nextNum++ + ": " + userMessage.getText());
            out.writeObject(checkboxState);
         } catch(Exception ex) {
             System.out.println("Sorry dude. Could not send it to the server.");
         }
         userMessage.setText("");
     } // close actionPerformed
  } // close inner class

  public class MyListSelectionListener implements ListSelectionListener {
      public void valueChanged(ListSelectionEvent le) {
        if (!le.getValueIsAdjusting()) {
          String selected = (String) incomingList.getSelectedValue();
          if (selected != null) {
              // now go to the map, and change the sequence
               boolean[] selectedState = (boolean[]) otherSeqsMap.get(selected);
               changeSequence(selectedState);
               sequencer.stop();
               buildTrackAndStart();
           }
        }
     } // close valueChanged
  } // close inner class
```

This is new... it's a lot like the SimpleChatClient, except instead of sending a String message, we serialize two objects (the String message and the beat pattern) and write those two objects to the socket output stream (to the server).

This is also new -- a ListSelectionListener that tells us when the user made a selection on the list of messages. When the user selects a message, we IMMEDIATELY load the associated beat pattern (it's in the HashMap called otherSeqsMap) and start playing it. There's some if tests because of little quirky things about getting ListSelectionEvents.

```
public class RemoteReader implements Runnable {
    boolean[] checkboxState = null;
    String nameToShow = null;
    Object obj = null;
    public void run() {
      try {
        while((obj=in.readObject()) != null) {
          System.out.println("got an object from server");
          System.out.println(obj.getClass());
          String nameToShow = (String) obj;
          checkboxState = (boolean[]) in.readObject();
          otherSeqsMap.put(nameToShow, checkboxState);
          listVector.add(nameToShow);
          incomingList.setListData(listVector);
        } // close while
      } catch(Exception ex) {ex.printStackTrace();}
    } // close run
} // close inner class
```

This is the thread job -- read in data from the server. In this code, 'data' will always be two serialized objects: the String message and the beat pattern (an ArrayList of checkbox state values)

When a message comes in, we read (deserialize) the two objects (the message and the ArrayList of Boolean checkbox state values) and add it to the JList component. Adding to a JList is a two-step thing: you keep a Vector of the lists data (Vector is an old-fashioned ArrayList), and then tell the JList to use that Vector as it's source for what to display in the list.

```
public class MyPlayMineListener implements ActionListener {
   public void actionPerformed(ActionEvent a) {
      if (mySequence != null) {
         sequence = mySequence;  // restore to my original
      }
   } // close actionPerformed
} // close inner class

public void changeSequence(boolean[] checkboxState) {
   for (int i = 0; i < 256; i++) {
      JCheckBox check = (JCheckBox) checkboxList.get(i);
       if (checkboxState[i]) {
           check.setSelected(true);
       } else {
            check.setSelected(false);
       }
   } // close loop
} // close changeSequence
```

This method is called when the user selects something from the list. We IMMEDIATELY change the pattern to the one they selected.

All the MIDI stuff is exactly the same as it was in the previous version.

```
public void makeTracks(ArrayList list) {
   Iterator it = list.iterator();
   for (int i = 0; i < 16; i++) {
      Integer num = (Integer) it.next();
      if (num != null) {
         int numKey = num.intValue();
         track.add(makeEvent(144,9,numKey, 100, i));
         track.add(makeEvent(128,9,numKey,100, i + 1));
       }
    } // close loop
 } // close makeTracks()
```

```
      public  MidiEvent makeEvent(int comd, int chan, int one, int two, int tick) {
          MidiEvent event = null;
          try {
            ShortMessage a = new ShortMessage();
            a.setMessage(comd, chan, one, two);
            event = new MidiEvent(a, tick);
          }catch(Exception e) { }
          return event;
      } // close makeEvent

} // close class
```

Nothing new. Just like the last version.

Sharpen your pencil

What are some of the ways you can improve this program?

Here are a few ideas to get you started:

1) Once you select a pattern, whatever current pattern was playing is blown away. If that was a new pattern you were working on (or a modification of another one), you're out of luck. You might want to pop up a dialog box that asks the user if he'd like to save the current pattern.

2) If you fail to type in a command-line argument, you just get an exception when you run it! Put something in the main method that checks to see if you've passed in a command-line argument. If the user doesn't supply one, either pick a default or print out a message that says they need to run it again, but this time with an argument for their screen name.

3) It might be nice to have a feature where you can click a button and it will generate a random pattern for you. You might hit on one you really like. Better yet, have another feature that lets you load in existing 'foundation' patterns, like one for jazz, rock, reggae, etc. that the user can add to.

You can find existing patterns on the Head First Java web start.

Final BeatBox server program

Most of this code is identical to the SimpleChatServer we made in the Networking and Threads chapter. The only difference, in fact, is that this server receives, and then re-sends, two serialized objects instead of a plain String (although one of the serialized objects happens to *be* a String).

```
import java.io.*;
import java.net.*;
import java.util.*;

public class MusicServer {

    ArrayList<ObjectOutputStream> clientOutputStreams;

    public static void main (String[] args) {
         new MusicServer().go();
    }

    public class ClientHandler implements Runnable {

         ObjectInputStream in;
         Socket clientSocket;

         public ClientHandler(Socket socket) {
           try {
             clientSocket = socket;
              in = new ObjectInputStream(clientSocket.getInputStream());

           } catch(Exception ex) {ex.printStackTrace();}
          } // close constructor


        public void run() {
             Object o2 = null;
             Object o1 = null;
           try {

             while ((o1 = in.readObject()) != null) {

                o2 = in.readObject();

                System.out.println("read two objects");
                tellEveryone(o1, o2);
              } // close while

           } catch(Exception ex) {ex.printStackTrace();}
       } // close run
   } // close inner class
```

```
 public void go() {
   clientOutputStreams = new ArrayList<ObjectOutputStream>();

      try {
        ServerSocket serverSock = new ServerSocket(4242);

        while(true) {
           Socket clientSocket = serverSock.accept();
           ObjectOutputStream out = new ObjectOutputStream(clientSocket.getOutputStream());
           clientOutputStreams.add(out);

           Thread t = new Thread(new ClientHandler(clientSocket));
           t.start();

           System.out.println("got a connection");
        }
      }catch(Exception ex) {
        ex.printStackTrace();
      }
 } // close go

 public void tellEveryone(Object one, Object two) {
    Iterator it = clientOutputStreams.iterator();
    while(it.hasNext()) {
      try {
         ObjectOutputStream out = (ObjectOutputStream) it.next();
         out.writeObject(one);
         out.writeObject(two);
       }catch(Exception ex) {ex.printStackTrace();}
      }
 } // close tellEveryone

} // close class
```