Java Music Systems: JMSL and JSyn
Nick Didkovsky (nick@didkovsky.com)
Class #1 — Java Overview & Syntona Quick Start

Some of these comments taken from Robert Rowe's lecture notes, with permission.

• A compiler changes the code of a high-level language (such as C or Pascal) into low-level machine codes for a particular platform. Therefore a single C program can be compiled to run on Macintosh, IBM-compatible, or UNIX-based computers. Programmers must use great care, however, not to let OS-specific commands (such as system calls) proliferate in their code because this will have to be rewritten each time the program is moved to a different operating system.

• Java is a language specification that was developed by Sun Microsystems for use in distributed systems. It is designed to be simple, object-oriented, small, portable, and secure. Originally Java was developed to run on embedded systems — small computers that run in devices devoted to other things, like your refrigerator. However it has scaled up gracefully to be a very powerful and elegant programming language that runs efficiently on a number of platforms.

• Java is designed to be simpler than languages such as C, C++, or Lisp. It is, however, a full blown programming language. Inherent in Java's design is the goal of minimizing the kinds of problems that slow down the programmer's productivity. Typical programming problems that are exceedingly time consuming to track down, such as memory leaks, lost pointers, and machine-level crashes that occur when developing in other languages such as C are eliminated by the fundamental design of Java.

• Interpreted languages (such as Lisp) are changed into machine instructions as they are run. This eliminates the compilation step and encourages a more interactive form of programming. Java is somewhat in between compiled and interpreted languages, in that Java programs are compiled into "bytecode", that then is interpreted on the host platform. This means that any computers running Java programs must have implemented a virtual machine that is able to take Java bytecodes and interpret them as low-level instructions on the host computer. Because of this approach Java programs will run with no modification on any computer that is Java-ready. In fact, because Java is now becoming so widespread, a number of compilers are available that translate Java directly into machine code, rather than bytecode for the virtual machine. While this makes the resulting applications more efficient, they are no longer portable.

• Object orientation is a concept in computer science that refers to the encapsulation of data and methods. An object contains within itself the data it needs to function and the methods that can operate on that data. Both methods and data can be either *private*, in which case only the object itself can use them, or *public*, meaning that other objects can access the data or execute the methods. Inheritance allows some classes to inherit data and methods from other classes. Thus programmers can use most of a generic object and specialize it in a subclass adding methods and data unique to their project.

• An object is an instantiated class, that is, an object is one concrete instance of the generic class definition. One class can generate many objects in a single application. Each of these objects maintains its own *state*, that is, its own set of values for the variables defined in the class.

• The traditional starting point for learning programming languages (at least since the C programming language gained popularity) is to make a program that prints out "Hello , World" on the computer screen. The Java program below declares a *class* that has one main function, in

which is an instruction that will cause "Hello, World" to be printed. As mentioned above, a class is a combination of methods and data. This encourages good program design, but also (importantly for Java) makes it easier to guarantee the security of the application.

• The *public* modifier of the class and main function indicates that they can be accessed from any other class in the program. If the class and that function had been identified as private members they could only be accessed by objects of the class *TrivialApplication*. The statement System.out.println calls a method "println()" that belongs to the class variable "out" of the class "System". Concatenating names like this is how the language identifies data and methods nested within other language constructs. The code below is a complete program — it can be compiled by a Java compiler and executed on any computer running a Java interpreter.

```
public class TrivialApplication {

        public static void main(String args[]) {
                System.out.println( "Hello World!" );
        }

}
```

• A Java *applet* is a program that can be added to a Web page and run by a web browser that is Java-enabled. A Java *application* is a standalone program that can be run in a Java virtual machine. Such applications do not require an operating system and so are designed for the network computer approach to multiuser systems.

• The HTML code to implement a web page containing the user's applet is shown below:

```
<HTML>
<HEAD>
<TITLE>My Java HTML page</TITLE>
</HEAD>
<BODY>
<applet code="mypackage.Calculate.class"
        codebase ="classes"
        width=600
        height=400 >
You need a Java-enabled browser to view this applet.
</applet>
</BODY>
</HTML>
```

Notice the polite message between the <applet> and </applet> tags. Browsers that do **not** understand Java will ignore the applet tags and print the "You need a…" message between them. Broswers that **do** understand Java are designed to ignore any text between the tags.

•A number of Java IDE's (Integrated Development Environments) are on the market, including Eclipse and IDEA by IntelliJ.

Eclipse and IDEA are very powerful packages which offer such modern tools as refactoring and continuous error checking. "Refactoring" includes such powerful operations as method extraction (highlight a chunk of inner code and extract it into its own method), smart renaming of variables, package names, and class names: as opposed to a global text search and replace, renaming an identifier in these packages knows the Java context of the identifier and only renames those which match. Refactoring is a hot topic in object oriented programming these days, and having some automated refactoring tools available in an IDE is an extraordinarily powerful resource.

Development Environment recommendations:
Mac OS X and Windows, use Eclipse. It is freely downloadable from
http://www.eclipse.org
Click Downloads tab and choose "Eclipse IDE for Java Developers"

Java's Swing Jump Start

Java has two packages for creating user interfaces and graphics: Java's Abstract Windowing Toolkit or "AWT", is a Java package that provides the programmer with GUI components, windows, layout, and event handling (by event handling we mean software response to user actions, like clicking a button or moving a slider). The more lightweight, modern, and preferred package that does all this and more is called "Swing." So let's start with programming in Swing.

Layouts

A Layout is a Java class that handles how components are arranged in a Container such as a Panel or a Frame. Simple layouts include:

FlowLayout, which adds components left to right, wrapping to the next row

BorderLayout, which divides the container into North, South, East, West, and Center

GridLayout which adds components in evenly spaced rows and columns. You can use ad hoc Panels liberally within a layout, to add components in a subarrangement (for example, a scrollbar and a label to display the scrollbar's value). Components like buttons, labels, etc are add()'ed to layouts.

Use the `setLayout()` method on a container to govern the layout style. For example:

```
JPanel p = new JPanel();
p.setLayout(new BorderLayout());
p.add(BorderLayout.North, new Label("Ollie");
p.add(BorderLayout.South, new Label("Parrot Jungle, 5 miles ahead");
p.add(new Label("egocentric")); // leave out a location and
                               // BorderLayout assumes "Center"

add(p); // add this panel to the Layout of this class
```

JLabel

A javax.swing.JLabel is simply a rectangular component that displays some text (String). Its text can be changed with its setText(String s) method. You can create a JLabel and add it to a layout like so:

```
JLabel myLabel;
…
myLabel = new JLabel("This will be displayed");
add(myLabel);
…
myLabel.setText("This is my new message");
```

Note that a Label's text can be updated at runtime with setText(). You can change it anytime you like - it is not permanent. So it can serve as a way to make messages appear in your GUI or show progress of some process.

`JButton`

A javax.swing.JButton is a component that can be clicked and fires a response. When clicked, it posts an ActionEvent to any ActionListener who cares to listen. The event is caught and handled by the method:

```
public void actionPerformed(ActionEvent e);
```

…which must be defined by any class which has registered itself as an ActionListener and adds itself to the button's list of listeners.

Below is a typical implementation of actionPerformed() which first checks which component caused the event, then calls the appropriate user-defined method:

```
public void actionPerformed(ActionEvent e) {
      Object source = e.getSource();
      if (source == myButton) handleMyButton;
      if (source == otherButton) handleOtherButton;
}
```

Below is a complete working example of an Applet that contains a button and changes its color when the button is clicked.

```
package didkovsky.javamusic;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;

/** Switch between two colors every time a button is pressed */

public class ColorChangingApplet extends JApplet implements ActionListener {

    JButton myButton;
    boolean colorSwitch;

    public void init() {
        myButton = new JButton("Color Change");
        JPanel p = new JPanel();
        p.add(myButton);
        add(p);
        colorSwitch = false;
        myButton.addActionListener(this);
    }

    void handleColorChange() {
        colorSwitch = !colorSwitch; // switch state from true to false
                                    // or from false to true
        if (colorSwitch) {
            myButton.setBackground(Color.PINK);
            System.out.println("changed button color to pink");
        } else {
            myButton.setBackground(Color.WHITE);
            System.out.println("changed button color to white");
        }
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == myButton) handleColorChange() ;
    }
}
```

Below is the HTML document that runs it (note the dot reference to the package `com.didkovsky.javamusic` and the specification of where the classes are located):

```
<HTML>
<HEAD>
<TITLE>ColorChangingApplet</TITLE>
</HEAD>
<BODY>
<applet code="com.didkovsky.javamusic.ColorChangingApplet.class"
       codebase ="classes"
       width=320
       height=200>
You need a Java-enabled browser to view this applet.
</applet>
</BODY>
</HTML>
```

Another Swing Example

Copy and paste this into a new class, study it, run it and see how it works. It shows how to use a variety of Swing components, adding a few things to the JApplet above. Nothing fancy but perhaps useful…

```
/*
 * Created on Sep 17, 2013 by Nick Didkovsky
 */
package didkovsky.javamusic;

import java.awt.Color;
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;

public class ColorChangingApplet2 extends JApplet implements
ActionListener {

    JButton myButton;
    boolean colorSwitch;

    JTextArea textArea;
    JTextField textField;

    public void init() {
        setLayout(new FlowLayout());
        myButton = new JButton("Color Change");
        JPanel p = new JPanel();
        p.add(myButton);
        add(p);
        colorSwitch = false;
        myButton.addActionListener(this);

        add(textArea = new JTextArea(3, 10));
        add(textField = new JTextField(6));
    }

    void handleColorChange() {
        colorSwitch = !colorSwitch; // switch state from true to false
        // or from false to true
        if (colorSwitch) {
```

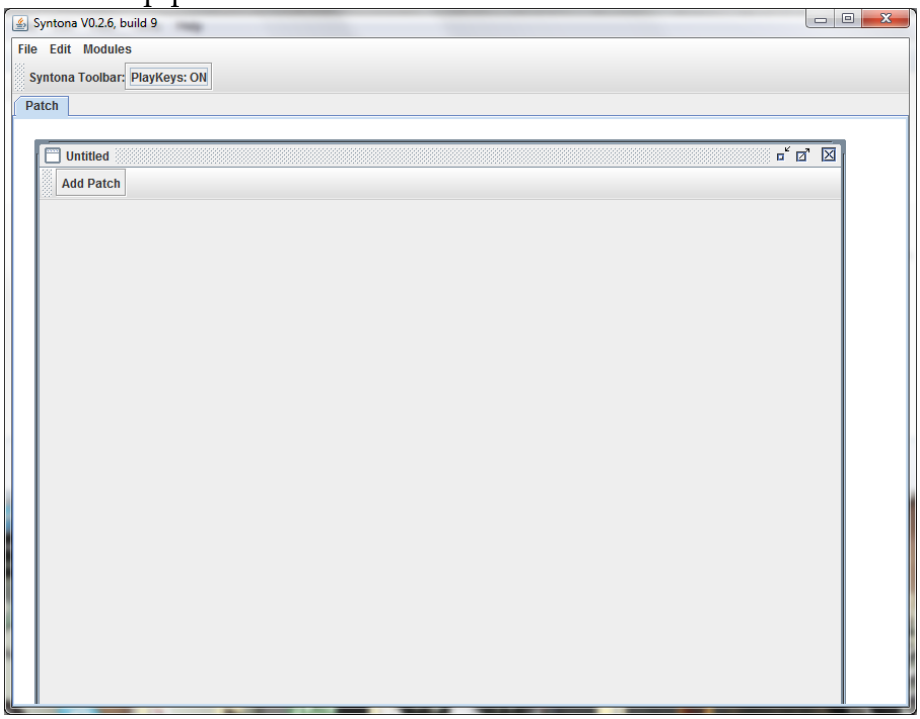
```
            myButton.setBackground(Color.PINK);
            System.out.println("changed button color to pink");
            textArea.setText("pink " + textField.getText());
        } else {
            myButton.setBackground(Color.WHITE);
            System.out.println("changed button color to white");
            textArea.setText("white " + textField.getText());
        }
    }

    public void actionPerformed(ActionEvent e) {
        Object source = e.getSource();
        if (source == myButton)
            handleColorChange();
    }
}
```

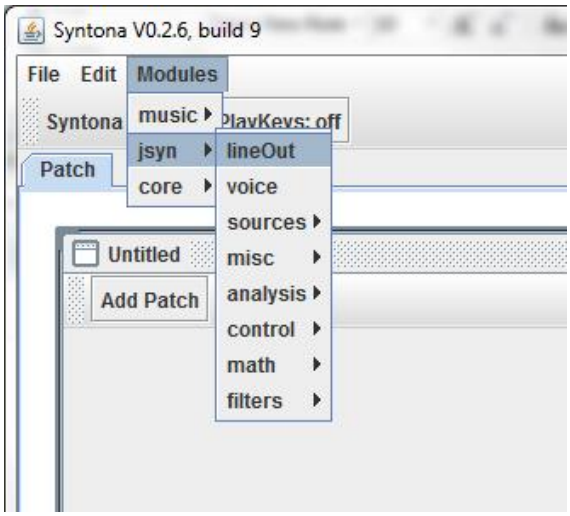
Syntona Quick Start: Creating a new Voice with Syntona

Syntona is a graphic patch editor created by JSyn author Phil Burk. It lets the user create and test JSyn circuits in a graphic user interface similar, for example to PD or Max/MSP. You can hear your sounds directly in Syntona, save and load patches, and most spectacularly, export Java source code (!). Download Syntona at http://www.softsynth.com/jsyn

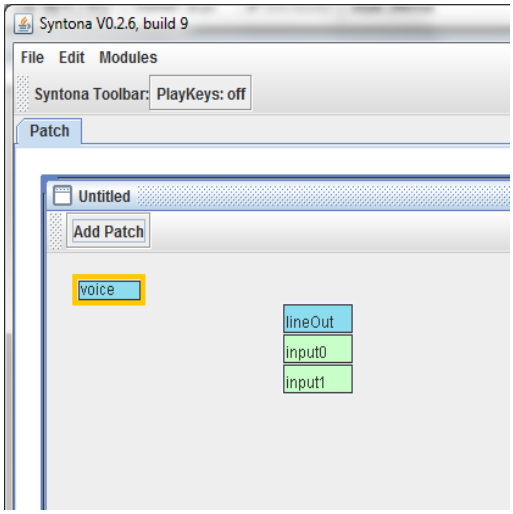
Double click Syntona.jar. An empty patch editor window opens. We refer to this as our "top patch"



1) Add a LineOut to the top patch. Select Modules -> JSyn -> lineOut as shown below

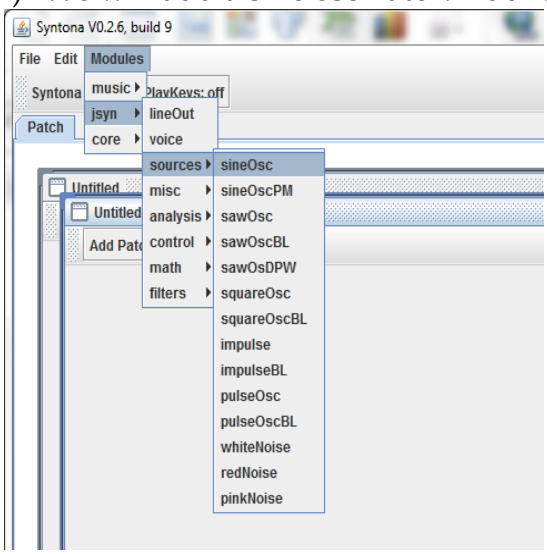


2) Similarly, add Modules -> JSyn -> voice. ***This voice module will be the patch we design and eventually export to JSyn.*** Your Syntona window should look like this:

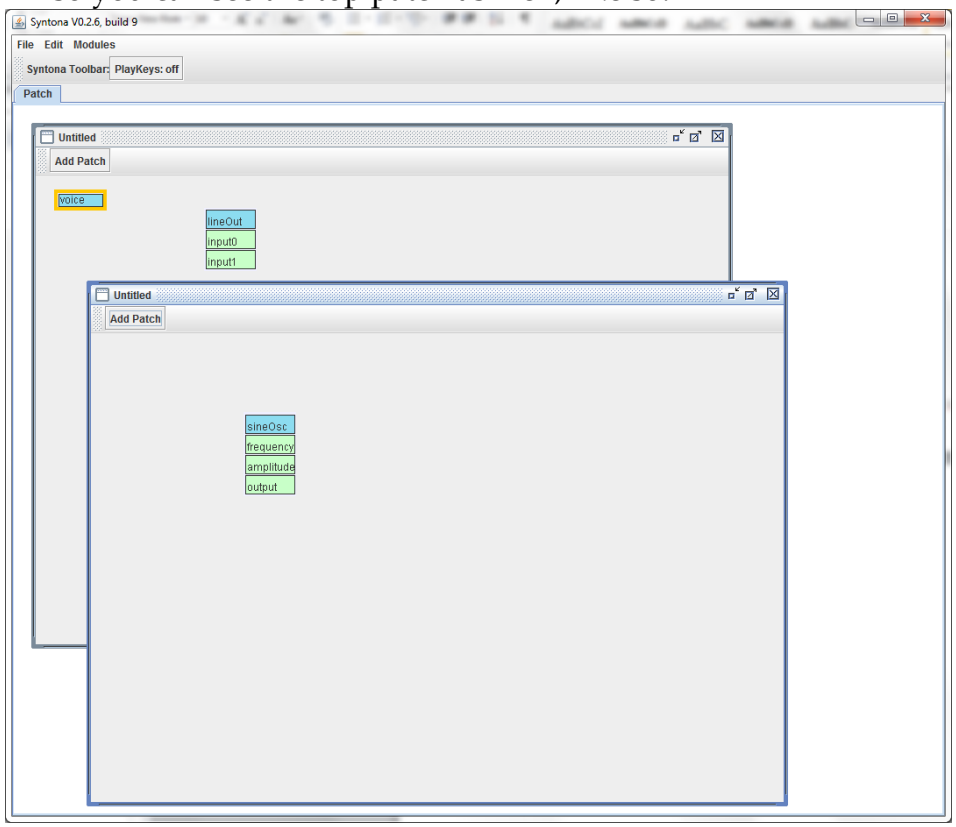


3) Double click "voice" module to open a new window for editing this voice.

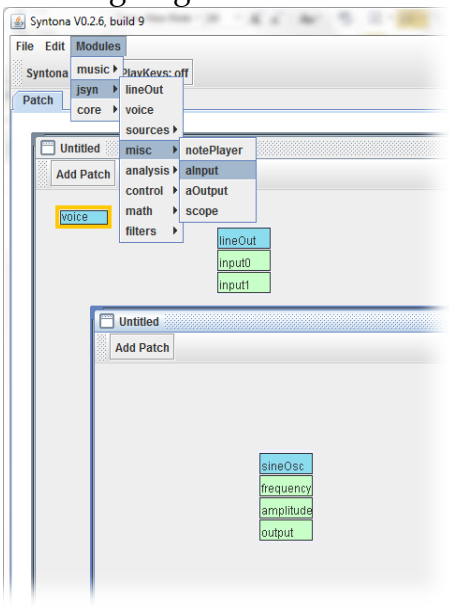
4) We will add a sine oscillator. Add Modules -> JSyn -> sources -> sineOsc



5) Move the sineOsc module a little to the right and reposition the window so you can see the top patch as well, like so:

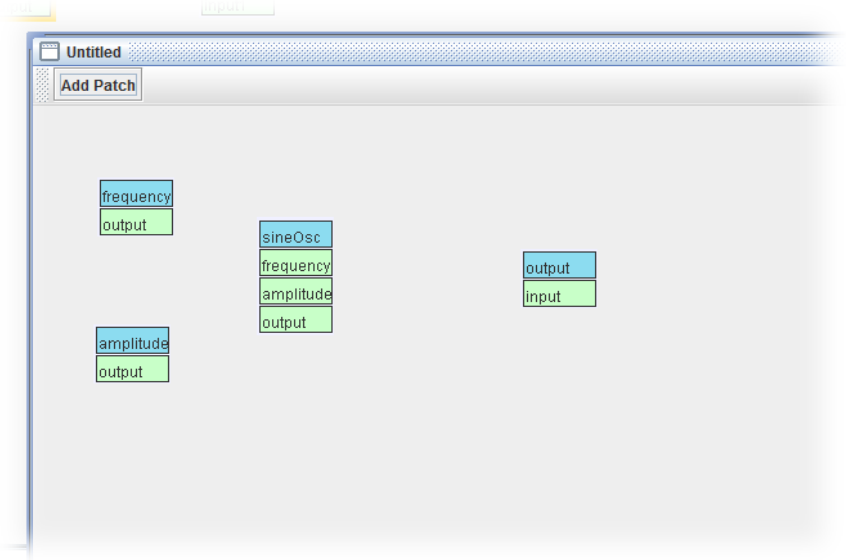


6) Now we add input and output ports to this voice so that the top patch (and later after we export the Java source, the rest of your Java environment), can access the inner workings of the voice we are designing. Select Modules -> JSyn -> misc -> aInput

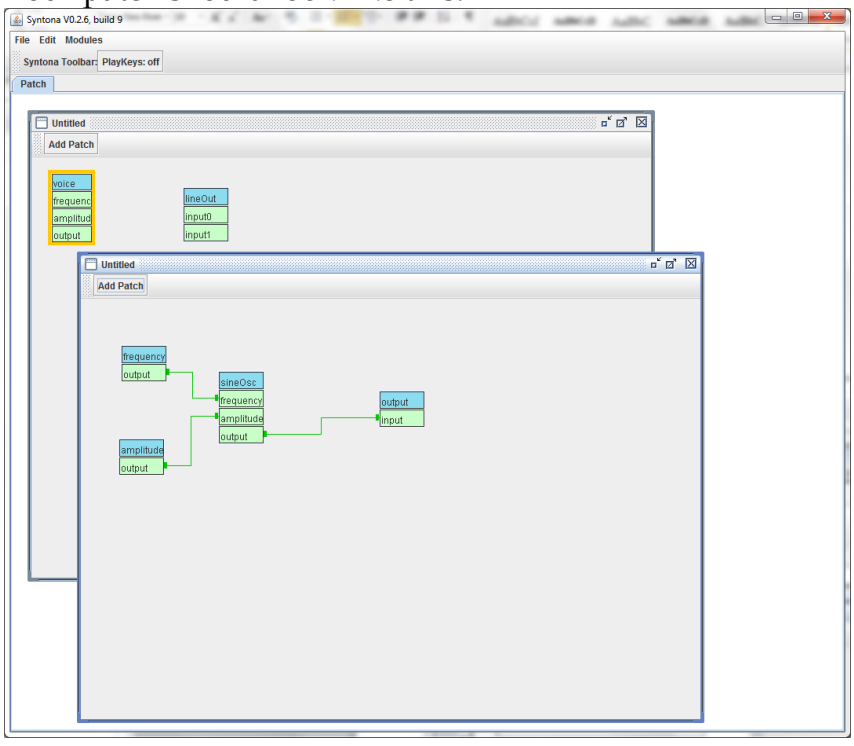


7) After the aInput appears, right click on it and rename it "frequency". You must hit Enter after making a change in order for Syntona to accept the change. Notice that the input port appears immediately in the top patch.
8) Add another aInput and rename it "amplitude"
9) Add an aOutput and rename it "output"

10) Your editor should look like this:



11) Now we connect the frequency input port **of the voice** to the frequency input port **of the oscillator**. Click on the frequency input's "output" port and drag to the frequency input port of the sineOsc. Similarly, connect amplitude to amplitude and sineOsc output to the output of the voice. Your patch should look like this:



12) Now activate your top patch and connect the output of your voice to input0 and input1 of lineOut. You will hear a sine tone begin immediately and sustain. Click the PlayKeys button and while it shows "PlayKeys:

ON", hit a few keyboard keys and listen to various diatonic pitches peformed by your voice. Be sure to save your patch as an XML file that can be reloaded later, with Save As.

13) Now we will export as Java Source (which is different than saving the Syntona patch in the previous step). Activate your voice's editor window again, and from the menu choose File -> Export Java Source as… and use a meaningful name like "SineVoice.java". Examine the exported source code. It should look something like this:

```
package patches;
import java.io.*;
import com.jsyn.*;
import com.jsyn.unitgen.*;
import com.jsyn.data.*;
import com.jsyn.ports.*;
import com.softsynth.shared.time.TimeStamp;
/**************
** WARNING - this code automatically generated by Syntona.
** The real source is probably a Syntona patch.
** Do NOT edit this file unless you copy it to another directory and change the name.
** Otherwise it is likely to get clobbered the next time you
** export Java source code from Syntona.
**
** Syntona is available from: http://www.softsynth.com/syntona/
*/
public class SineVoice extends Circuit implements UnitVoice
{
    // Declare units and ports.
    com.jsyn.unitgen.SineOscillator sineOsc;
    com.jsyn.unitgen.PassThrough frequencyPassThrough;
    public UnitInputPort frequency;
    com.jsyn.unitgen.PassThrough amplitudePassThrough;
    public UnitInputPort amplitude;
    com.jsyn.unitgen.PassThrough outputPassThrough;
    public UnitOutputPort output;

    public SineVoice()
    {
        // Create unit generators.
        add( sineOsc = new com.jsyn.unitgen.SineOscillator() );
        add( frequencyPassThrough = new com.jsyn.unitgen.PassThrough() );
        addPort( frequency = frequencyPassThrough.input, "frequency");
        add( amplitudePassThrough = new com.jsyn.unitgen.PassThrough() );
        addPort( amplitude = amplitudePassThrough.input, "amplitude");
        add( outputPassThrough = new com.jsyn.unitgen.PassThrough() );
        addPort( output = outputPassThrough.output, "output");
        // Connect units and ports.
        frequencyPassThrough.output.connect(sineOsc.frequency);
        amplitudePassThrough.output.connect(sineOsc.amplitude);
        // Setup
        sineOsc.frequency.setup(40.0, 587.3312228939296, 8000.0);
        sineOsc.amplitude.setup(0.0, 0.5, 1.0);
        frequency.setup(0.0, 587.3312228939296, 100.0);
        amplitude.setup(0.0, 0.5, 100.0);
        outputPassThrough.input.setup(0.0, 0.0, 100.0);
    }

    @Override
    public void noteOn( double frequency, double amplitude, TimeStamp timeStamp )
    {
        this.frequency.set( frequency, timeStamp );
        this.amplitude.set( amplitude, timeStamp );
    }

    @Override
```

```
    public void noteOff( TimeStamp timeStamp )
    {
    }

    @Override
    public UnitOutputPort getOutput()
    {
        return output;
    }
}
```

14) You can drag this into Eclipse, and after renaming the package appropriately, use it in your SoundTesterApplet.

Homework Assignment:

Standard Track:

Create five Syntona patches that are significantly different from each other and conform to the "voice" model shown above. The top patch of at least one should have some sort of sequencing and compositional units driving the voice. Upload these to DropBox.

Java Track (extra credit):

Create a JApplet with a background colored anything but white, that includes the following Swing components and interactions:

- A JLabel that says "Hello I am <your names>'s first Applet"
- A JTextArea that initially displays the text "This is a TextArea."
- A JButton which, when clicked, clears the JTextArea and prints the current time in milliseconds (see Java's `System` class documentation).
- A JTextField into which the user can type a String.
- A JButton which, when clicked, pulls the String from the above JTextField, converts it to an integer, clears the JTextArea, and prints both the integer and the integer multiplied by 2 in the JTextArea. An algorithm for converting a String to an integer follows:

```
int  value = (new Integer(someString)).intValue();
```

Of course someString has to be a String consisting of digits. Don't expect to convert the String "Hello" to an int with this algorithm!!!

Make this applet part of a package that includes your name (for example, johnsmith.javamusic), and post it on your website. Email me the URL of the HTML page that runs it (nick@didkovsky.com). I suggest your website have a directory called `JavaMusicSystems` which contains the HTML work for this class, which would include HTML pages which run your Java classes and other media like soundfiles, etc. It should contain a "classes" directory into which all your .class files go. If your website does not permit Java Applets, upload your source code to Dropbox

MORE EXTRA CREDIT:

Create an applet containing a JTextField into which you can type a base frequency, another JTextField into which you can type the number of intervals in an octave, and another that takes the pitch you want to calculate. Add a button which, when clicked, prints the frequency of this step in a label.

For example, for 12 tone equal tempered music (Western European), type in 27.50 as the base frequency (this is A0, the lowest note on a piano, see Dodge/Jerse p. 37), 12 as the number of intervals per octave, and 2 to calculate the frequency of B0 (that is 2 steps above the base frequency). Your applet should calculate and display 30.87 Hz as the resulting frequency

The formula for calculating frequency from a base pitch and an interval number is:

$\text{Pitch} = \text{basefreq} * 2^{i/n}$

Where i = step and n = intervals/octave

Example: basefreq of A0 is 27.50

To calculate A#, which is one semitone higher, the formula would be:

$\text{Pitch} = 27.50 * 2^{1/12} = 29.135$