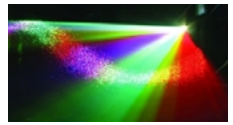
Chapter 11
Sampled Auditory Content

The Design and Implementation of Multimedia Software

David Bernstein

Jones and Bartlett Publishers

www.jbpub.com

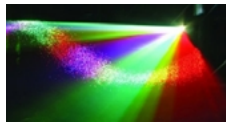


Temporal Sampling

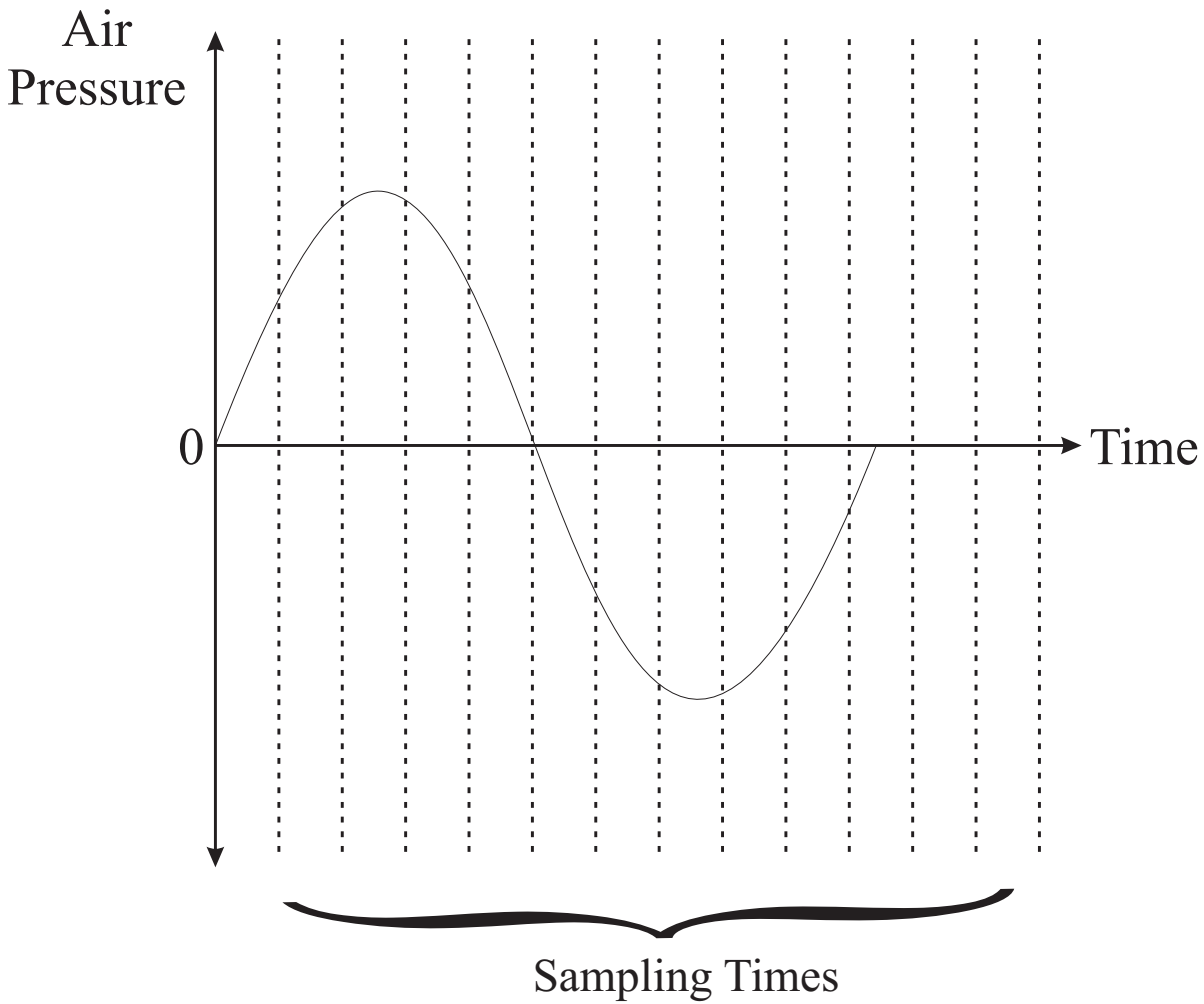
Definition

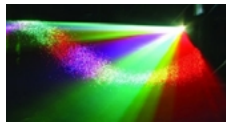
Temporal sampling involves measuring the wave at (usually regular) discrete points in time.

- CDs normally use a 44.1kHz sampling rate (i.e., contain 44,100 samples per second).
- DVD audio normally uses a 96kHz sampling rate (i.e., the the audio track contains 96,000 samples per second).



Temporal Sampling (cont.)



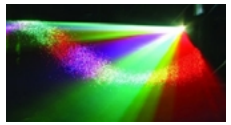


Quantization

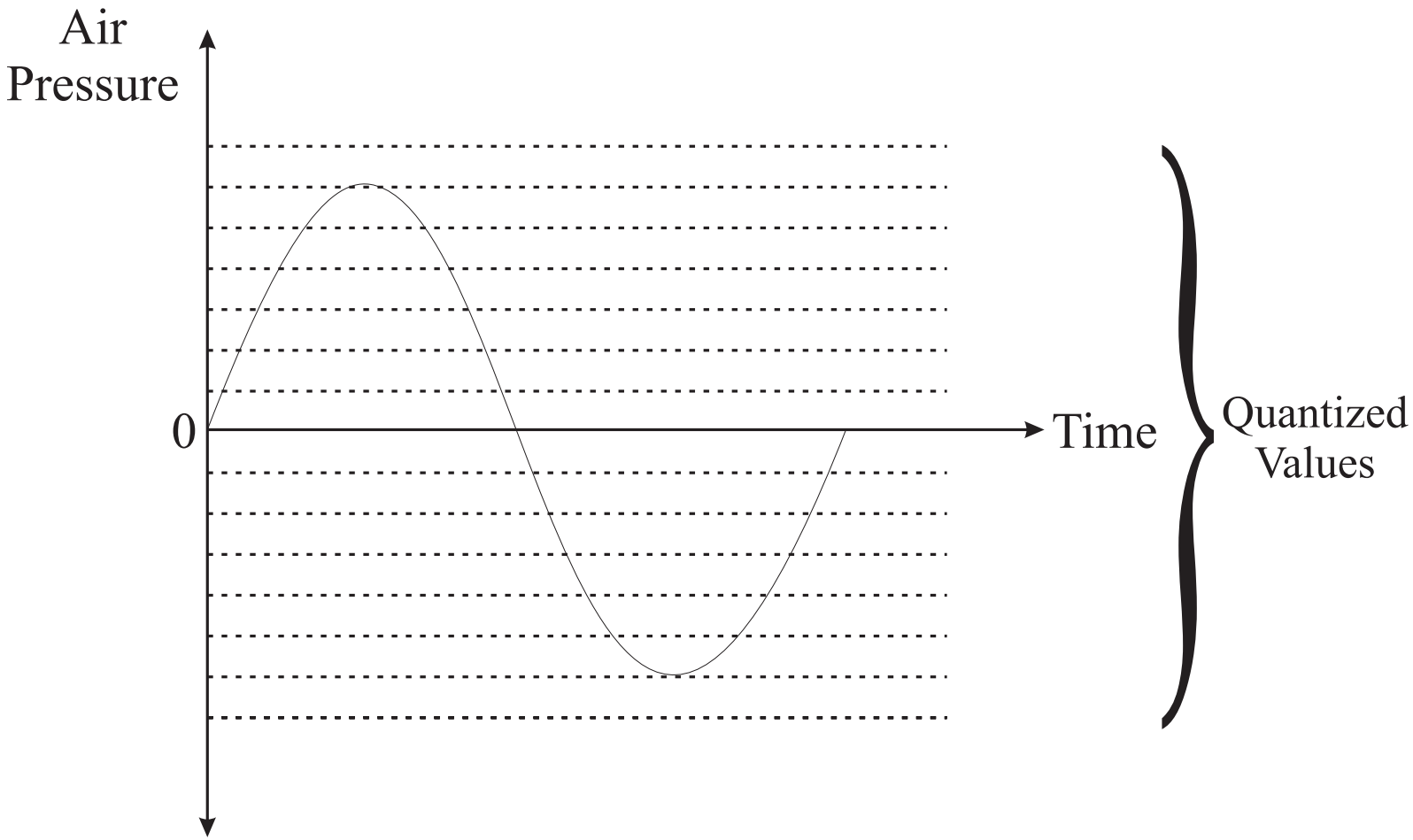
Definition

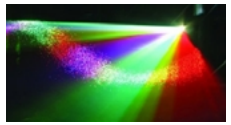
Quantization involves limiting the measured amplitudes to a discrete set of values.

For example, if 8 bits quantization is used there are 256 different amplitudes and the actual amplitude is rounded or truncated to one of these 256 values.



Quantization (cont.)

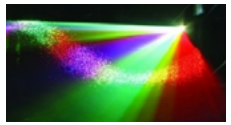




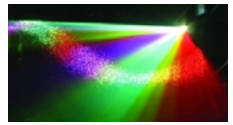
The `AudioFormat` Class

An `AudioFormat` object has, among others, the following attributes:

- The number of channels (e.g., mono, stereo).
- The sampling rate.
- The quantization (i.e., the number of bits per sample).
- The encoding technique (e.g., linear pulse code modulation, nonlinear mu-law).



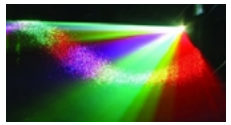
Presentation of Sampled Audio in Java



The `Clip` Class

A `Clip` object:

- Is a type of `Line` that contains data that can be loaded prior to presentation.
- Renders its sampled auditory content when its `start()` method is called.

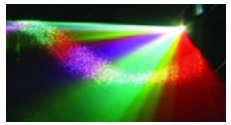


Creating a Clip Object

```
// Get the resource
finder = ResourceFinder.createInstance();
is     = finder.findInputStream("/"+args[0]);

// Create an AudioInputStream from the InputStream
stream = AudioSystem.getAudioInputStream(is);
```

```
// Create a Clip (i.e., a Line that can be pre-load
clip = AudioSystem.getClip();
```

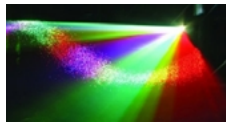
```
// Tell the Clip to acquire any required system
// resources and become operational
clip.open(stream);
```



Using a `Clip` Object

```
// Present the Clip (without blocking the
// thread of execution)
clip.start();
```

Note that `start()` method does not block the thread of execution.

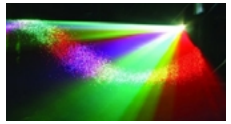


Requirements

F11.1 Encapsulate signals.

F11.2 Operate on signals.

F11.3 Present/render these signals.



An Overview

- Any Encapsulation Must Include:

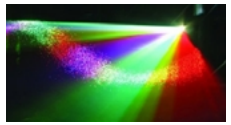
  The sample points for all of the signals (i.e., one signal for monophonic, two signals for stereophonic, etc).

  Information about the sampling process.

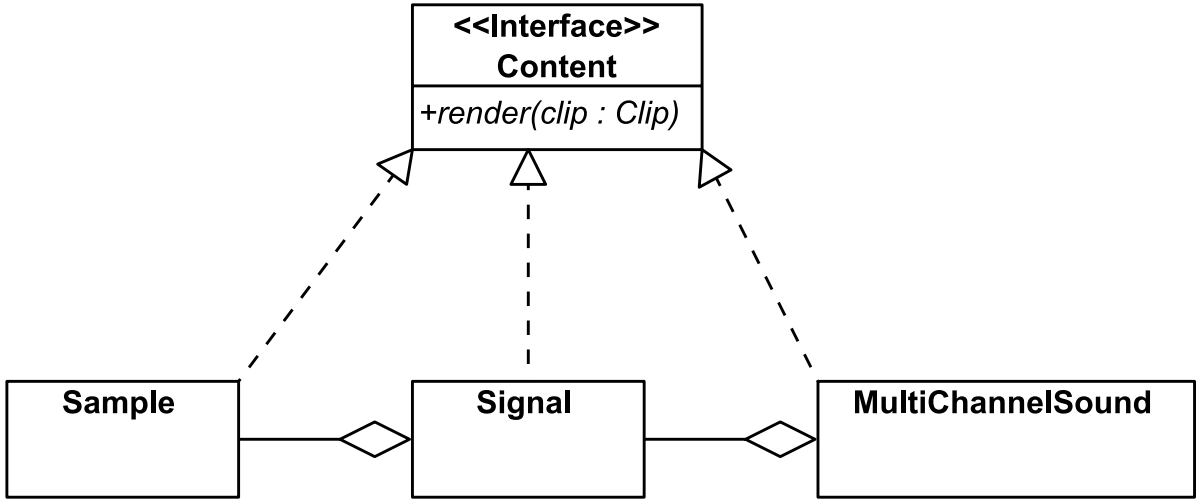
- Some Observations:

  Information about the sampling process can be stored in an `AudioFormat` object.

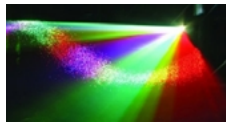
  All that remains is to consider ways to encapsulate samples and signals.



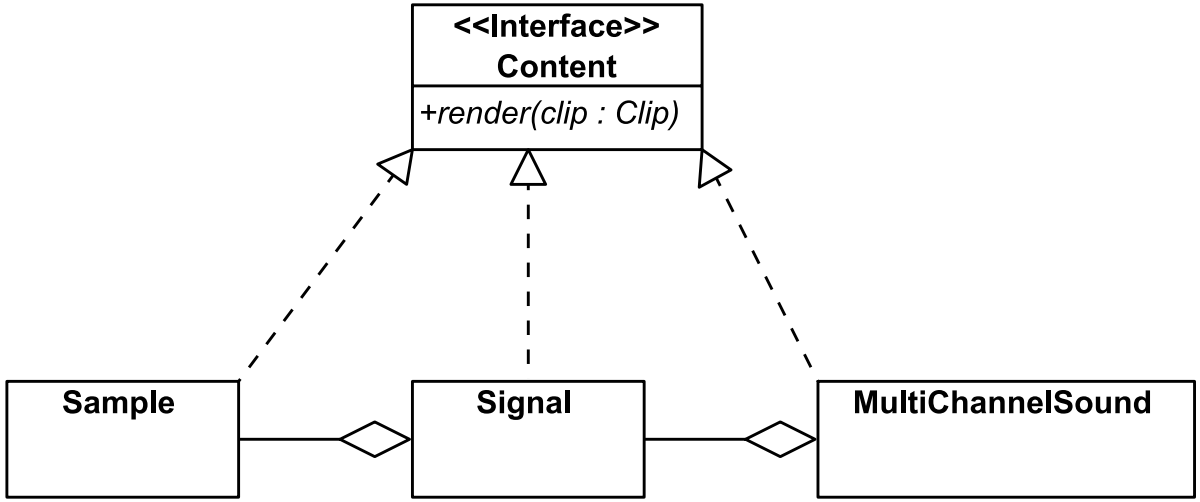
Alternative 1



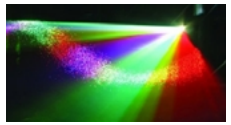
What are the shortcomings?



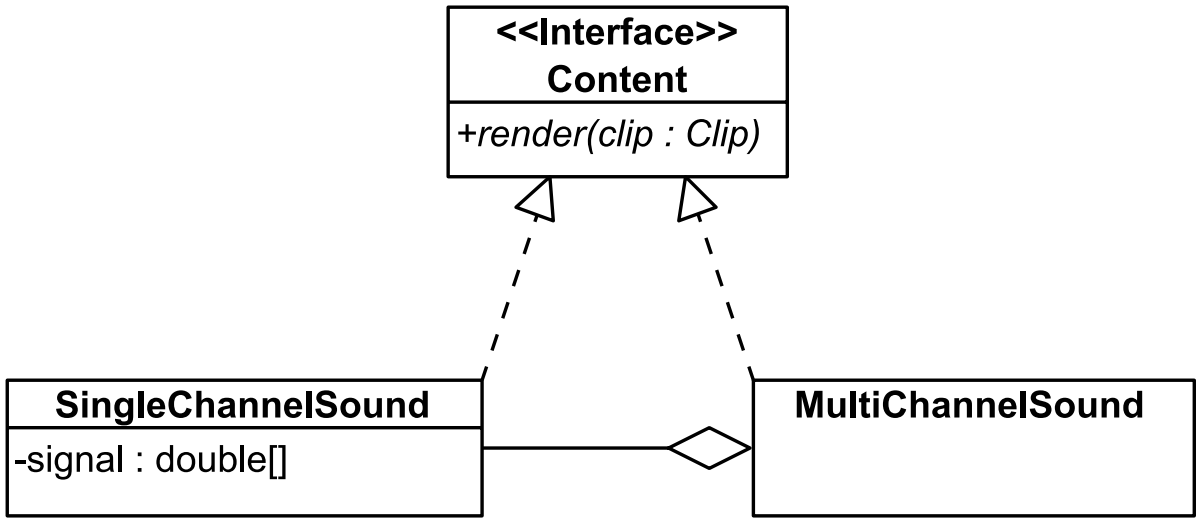
Alternative 1



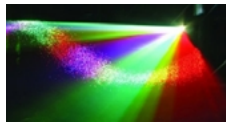
Since a sample is nothing but a numeric value, there is no reason to have a `Sample` class.



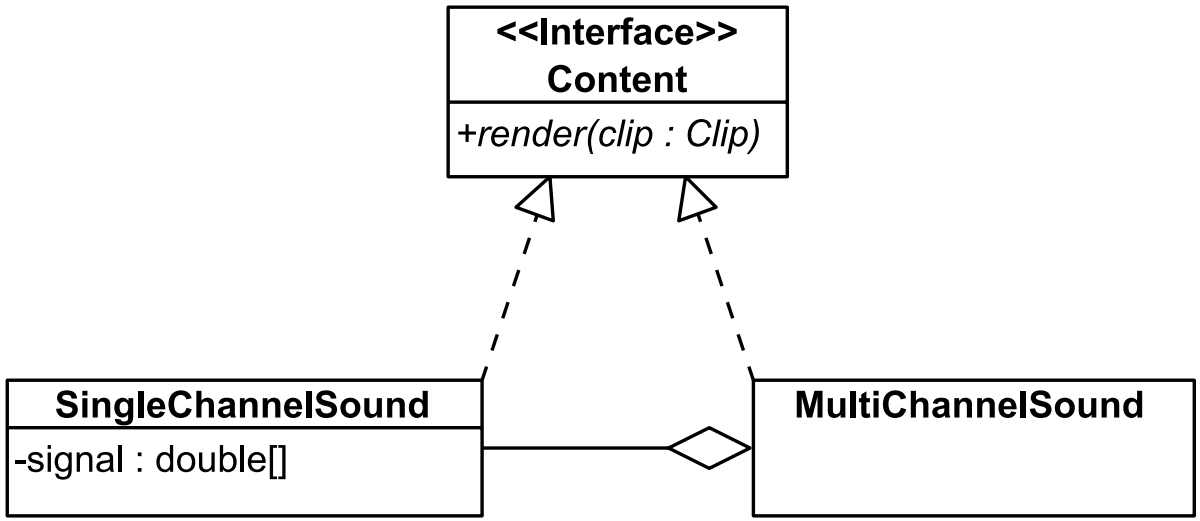
Alternative 2



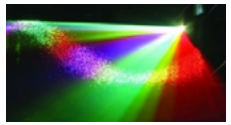
What are the shortcomings?



Alternative 2

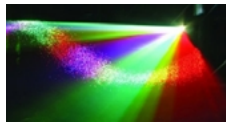


Many existing file formats make it difficult to independently/sequentially construct `SingleChannelSound` objects and then combine them into a `MultiChannelSound` object.



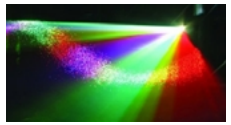
Alternative 3

BufferedSound
-signal : List<double[]>
+render(clip : Clip)



BufferedSound – Structure

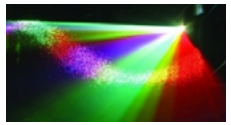
```
package auditory.sampled;

import java.util.*;
import javax.sound.sampled.*;

public class BufferedSound implements Content
{
    private ArrayList<double[]>  channels;
    private AudioFormat          format;
    private int                  numberOfSamples;

    private static final double MAX_AMPLITUDE       =  32767.0;
    private static final double MIN_AMPLITUDE       = -32767.0;
    private static final int    SAMPLE_SIZE_IN_BITS = 16;
    private static final int    BYTES_PER_CHANNEL   = SAMPLE_SIZE_IN_BITS/8;
}
```



BufferedSound – Constructor

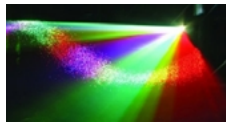
To simplify the discussion that follows, this class uses sampling processes that vary only in their sampling rates; all other aspects of the process are standardized. This is evident in the explicit value constructor of the class.

```
public BufferedSound(float sampleRate)
{
   format = new AudioFormat(
      AudioFormat.Encoding.PCM_SIGNED,
      sampleRate,          // Sample rate in Hz
      SAMPLE_SIZE_IN_BITS, // Sample size in bits
      0,                   // Number of channels
      0,                   // Frame size in bytes
      sampleRate,          // Frame rate in Hz
      true);               // Big-endian or not

   channels = new ArrayList<double[]>();
   numberOfSamples = 0;
}
```



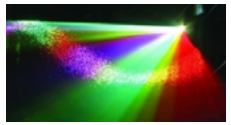
BufferedSound – addChannel()

```
public synchronized void addChannel(double[] signal)
{
   if (numberOfSamples == 0) numberOfSamples = signal.length;

   if (numberOfSamples == signal.length)
   {
      channels.add(signal);
      updateAudioFormat();
   }
}
```



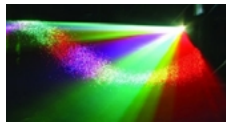
BufferedSound– updateAudioFormat()

```
private void updateAudioFormat()
{
   format = new AudioFormat(
      format.getEncoding(),                  // Encoding
      format.getSampleRate(),                // Sample rate in Hz
      format.getSampleSizeInBits(),          // Sample size in bits
      channels.size(),                       // Number of channels
      channels.size()*BYTES_PER_CHANNEL,     // Frame size in bytes
      format.getSampleRate(),                // Frame rate in Hz
      format.isBigEndian());                 // Big-endian or not
}
```



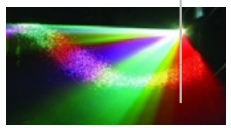
BufferedSound – matches()

```
public synchronized boolean matches(BufferedSound other)
{
   boolean      result;

   result = false;
   result = getAudioFormat().matches(other.getAudioFormat()) &&
            (getNumberOfSamples() == other.getNumberOfSamples());


   return result;
}
```



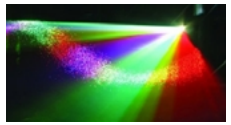
BufferedSound – append()

```
public synchronized void append(BufferedSound other)
{
   ArrayList<double[]>  temp;
   double[]             otherSignal, tempSignal, thisSignal;
   Iterator<double[]>   i, j;

   if (matches(other))
   {
      temp = new ArrayList<double[]>();

      i = channels.iterator();
      j = other.channels.iterator();
      while (i.hasNext())
      {
         thisSignal  = i.next();
         otherSignal = j.next();

         // Allocate space for the new signal
         tempSignal = new double[thisSignal.length +
                                 otherSignal.length];

         // Copy the current signal
         System.arraycopy(thisSignal, 0,
                          tempSignal, 0, thisSignal.length);

         // Append the other left signal
         System.arraycopy(otherSignal, 0,
                          tempSignal, thisSignal.length,
```



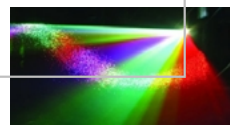
BufferedSound – append() (cont.)

```
                                otherSignal.length);

            // Save the longer signal
            temp.add(tempSignal);
        }
        channels = temp;
    }
}
```

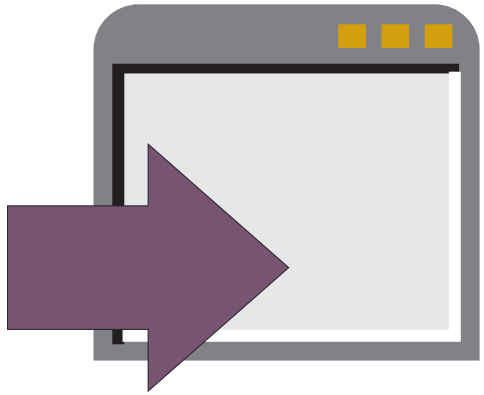


BufferedSoundFactory – Pure Tones

```
public BufferedSound createBufferedSound(double frequency,
                                         int    length,
                                         float  sampleRate,
                                         double amplitude)
{
   BufferedSound     sound;
   double            radians,radiansPerSample, rmsValue;
   double[]          signal;
   int n;

   //samples =      samples/sec * sec
   n         = (int)(sampleRate * (double)length/1000000.0);

   signal    = new double[n];
   //  rads/sample  = ( rads/cycle * cycles/sec)/ samples/sec
   radiansPerSample = (Math.PI*2.0 * frequency) / sampleRate;
   for (int i=0; i<signal.length; i++)
   {
      // rad  =   rad/sample     * sample
      radians = radiansPerSample * i;

      signal[i] = amplitude * Math.sin(radians);
   }
   sound = new BufferedSound(sampleRate);
   sound.addChannel(signal);
   return sound;
}
```



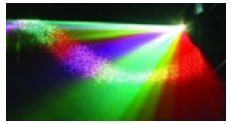
Pure Tones – Demonstration



In `extras`:

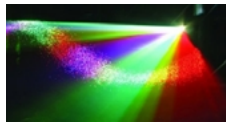
PureTone.html

```
java -cp BufferedSound.jar BufferedSoundApplication NONE 200
```



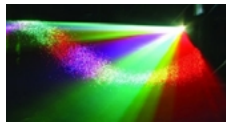
BufferedSoundFactory – Using an AudioInputStream

```
public BufferedSound createBufferedSound(AudioInputStream inStream)
                              throws IOException,
                                     UnsupportedAudioFileException
{
   AudioFormat         inFormat, pcmFormat;
   AudioInputStream    pcmStream;
   BufferedSound       sound;
   byte[]              rawBytes;
   double[]            leftSignal, monoSignal, rightSignal;
   int                 bufferSize, offset, n, sampleLength;
   int[]               signal;
}
```



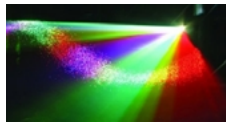
BufferedSoundFactory – Encoding

```
inFormat = inStream.getFormat();

 // Convert ULAW and ALAW to PCM
 if ((inFormat.getEncoding() == AudioFormat.Encoding.ULAW) ||
     (inFormat.getEncoding() == AudioFormat.Encoding.ALAW)   ) {

     pcmFormat = new AudioFormat(
                         AudioFormat.Encoding.PCM_SIGNED,
                         inFormat.getSampleRate(),
                         inFormat.getSampleSizeInBits()*2,
                         inFormat.getChannels(),
                         inFormat.getFrameSize()*2,
                         inFormat.getFrameRate(),
                         true);

     pcmStream = AudioSystem.getAudioInputStream(pcmFormat,
                                                 inStream);
 }
 else // It is PCM
 {
    pcmFormat = inFormat;
    pcmStream = inStream;
 }
```



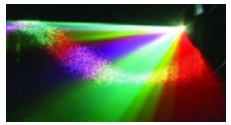
BufferedSoundFactory – Buffer

```
// Create a buffer and read the raw bytes
bufferSize = (int)(pcmStream.getFrameLength())
                   * pcmFormat.getFrameSize();

rawBytes = new byte[bufferSize];
offset   = 0;
n        = 0;
while (pcmStream.available() > 0)
{
   n  = pcmStream.read(rawBytes, offset, bufferSize);
   offset += n;
}
```



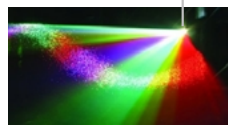
BufferedSoundFactory – Conversion

```
// Convert the raw bytes
if (pcmFormat.getSampleSizeInBits() == 8)
{
   signal = processEightBitQuantization(rawBytes, pcmFormat);
}
else
{
   signal = processSixteenBitQuantization(rawBytes, pcmFormat);
}
```



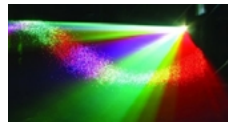
BufferedSoundFactory – Channels

```
sound = new BufferedSound(pcmFormat.getSampleRate());

// Process the individual channels
if (pcmFormat.getChannels() == 1)  // Mono
{
   sampleLength = signal.length;
   monoSignal   = new double[sampleLength];

   for (int i=0; i<sampleLength; i++)
   {
      monoSignal[i]  = signal[i]; // Convert to double
   }
   sound.addChannel(monoSignal);
}
else                                // Stereo
{
   sampleLength = signal.length/2;
   leftSignal   = new double[sampleLength];
   rightSignal  = new double[sampleLength];

   for (int i=0; i<sampleLength; i++)
   {
      leftSignal[i]  = signal[2*i];
      rightSignal[i] = signal[2*i+1];
   }
   sound.addChannel(leftSignal);
```



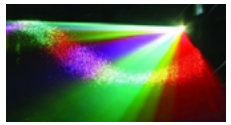
BufferedSoundFactory – Channels (cont.)

```
            sound.addChannel(rightSignal);
        }
```



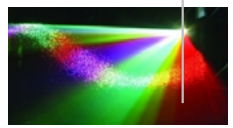
BufferedSoundFactory – 8-bit

```
private int[] processEightBitQuantization(
                                    byte[]      rawBytes,
                                    AudioFormat format)
{
   int          lsb, msb;
   int[]        signal;
   String       encoding;


   signal = new int[rawBytes.length];
   encoding = format.getEncoding().toString();

   if (encoding.startsWith("PCM_SIGN"))
   {
      for (int i=0; i<rawBytes.length; i++)
         signal[i] = rawBytes[i];
   }
   else
   {
      for (int i=0; i<rawBytes.length; i++)
         signal[i] = rawBytes[i]-128;
   }

   return signal;
}
```



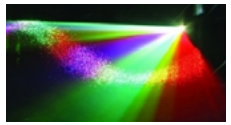
BufferedSoundFactory – 16-bit

```
private int[] processSixteenBitQuantization(
                                        byte[]      rawBytes,
                                        AudioFormat format)
{
   int         lsb, msb;
   int[]       signal;

   signal = new int[rawBytes.length / 2];
   if (format.isBigEndian())  // Big-endian
   {
      for (int i=0; i<signal.length; i++)
      {
         // First byte is high-order byte
         msb = (int) rawBytes[2*i];

         // Second byte is low-order byte
         lsb = (int) rawBytes[2*i+1];

         signal[i] = msb << 8 | (255 & lsb);
      }
   }
   else                          // Little-endian
   {
      for (int i=0; i<signal.length; i++)
      {
         // First byte is low-order byte
         lsb = (int) rawBytes[2*i];
```



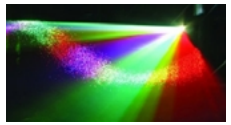
BufferedSoundFactory – 16-bit (cont.)

```
            // Second byte is high-order byte
            msb = (int) rawBytes[2*i+1];

            signal[i] = msb << 8 | (255 & lsb);
        }
    }
    return signal;
}
```

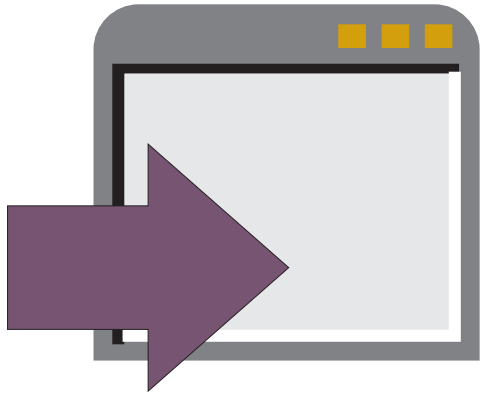


BufferedSoundFactory – Using a File

```
public BufferedSound createBufferedSound(String name)
                              throws IOException,
                                     UnsupportedAudioFileException
{
   AudioInputStream        stream;
   URL                     url;

   url    = finder.findURL(name);
   stream = AudioSystem.getAudioInputStream(url);

   return createBufferedSound(stream);
}
```



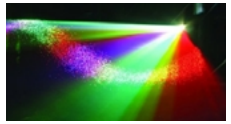
Using a File – Demonstration



In `extras`:

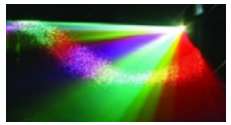
FilePlayer.html

```
java -cp FilePlayer.jar FilePlayerApplication preface.aif
```



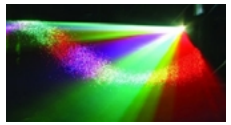
Unary Operations

```
package auditory.sampled;

public interface BufferedSoundUnaryOp
{
    public BufferedSound filter(BufferedSound src,
                                BufferedSound dest);
}
```

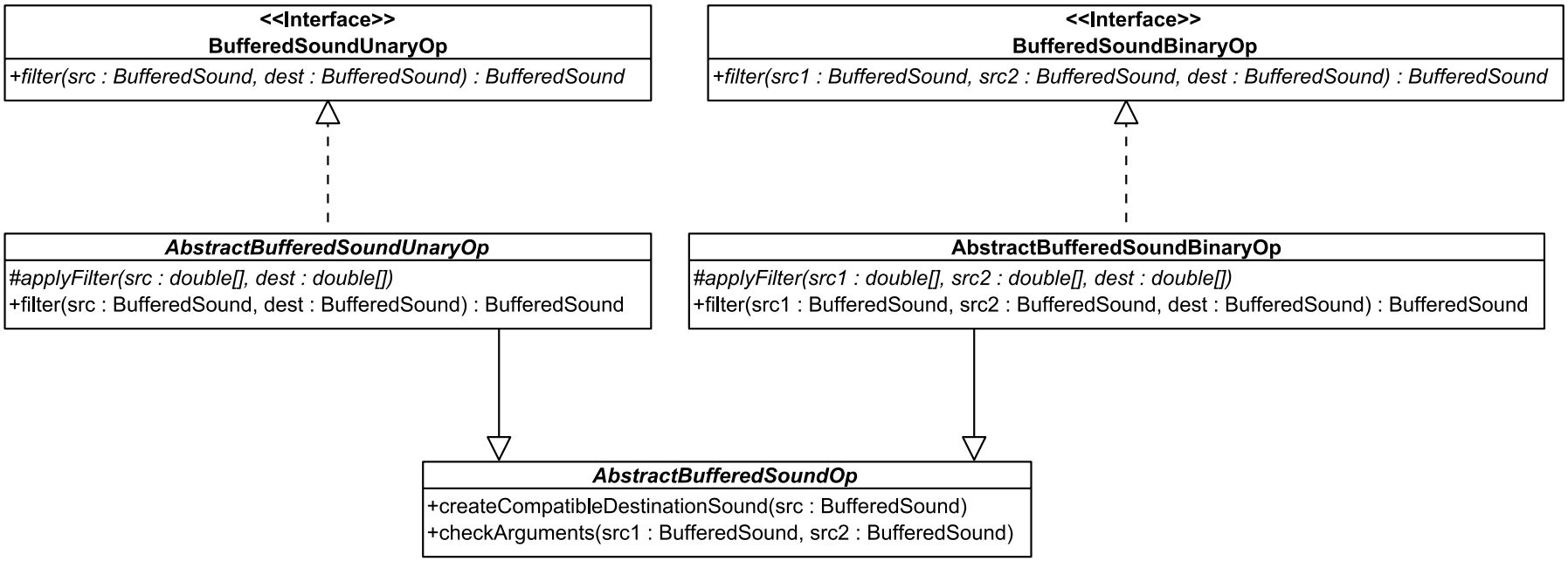


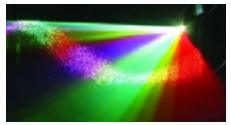
Binary Operations

```
package auditory.sampled;

public interface BufferedSoundBinaryOp
{
    public BufferedSound filter(BufferedSound src1, BufferedSound src2,
                                BufferedSound dest)
                         throws IllegalArgumentException;
}
```



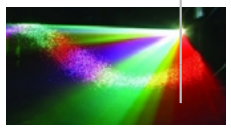
Operating on Sampled Auditory Content





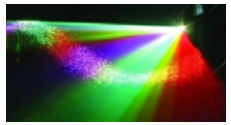
AbstractBufferedSoundOp

```
package auditory.sampled;

public abstract class AbstractBufferedSoundOp
{
    public BufferedSound createCompatibleDestinationSound(
                                          BufferedSound src)
    {
       BufferedSound        temp;
       float                sampleRate;
       int                  channels, length;

       channels   = src.getNumberOfChannels();
       length     = src.getNumberOfSamples();
       sampleRate = src.getSampleRate();

       temp = new BufferedSound(sampleRate);

       for (int i=0; i<channels; i++)
       {
          temp.addChannel(new double[length]);
       }

       return temp;
    }

    protected void checkArguments(BufferedSound a, BufferedSound b)
                                  throws IllegalArgumentException
    {
```



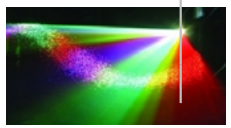
AbstractBufferedSoundOp (cont.)

```
        if (!a.matches(b))
            throw(new IllegalArgumentException("Argument Mismatch"));
    }
}
```



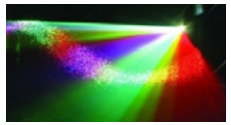
AbstractBufferedSoundUnaryOp

```
package auditory.sampled;

import java.util.*;

public abstract class      AbstractBufferedSoundUnaryOp
                extends    AbstractBufferedSoundOp
                implements BufferedSoundUnaryOp
{
    public abstract void applyFilter(double[] source,
                                     double[] destination);

    public void applyFilter(Iterator<double[]> source,
                            Iterator<double[]> destination)
    {
       while (source.hasNext())
       {
          applyFilter(source.next(), destination.next());
       }
    }

    public BufferedSound filter(BufferedSound src,
                                BufferedSound dest)
    {
       Iterator<double[]>    source, destination;

       // Construct the destination if necessary; otherwise check it
       if (dest == null)
          dest = createCompatibleDestinationSound(src);
```



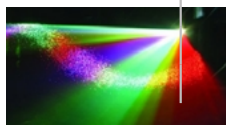
AbstractBufferedSoundUnaryOp (cont.)

```
        // Get the source channels
        source      = src.getSignals();

        // Get the destination channels
        destination = dest.getSignals();

        // Apply the filter
        applyFilter(source, destination);

        return dest;
    }
}
```



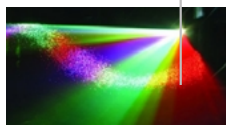
AbstractBufferedSoundBinaryOp

```
package auditory.sampled;

import java.util.*;

public abstract class      AbstractBufferedSoundBinaryOp
                extends    AbstractBufferedSoundOp
                implements BufferedSoundBinaryOp
{
    public abstract void applyFilter(double[] source1,
                                     double[] source2,
                                     double[] destination);

    public void applyFilter(Iterator<double[]> source1,
                            Iterator<double[]> source2,
                            Iterator<double[]> destination)
    {
       while (source1.hasNext())
       {
          applyFilter(source1.next(), source2.next(), destination.next());
       }
    }

    protected void checkArguments(BufferedSound a,
                                  BufferedSound b)
                                  throws IllegalArgumentException
    {
       if (!a.matches(b))
           throw(new IllegalArgumentException("Argument Mismatch"));
```



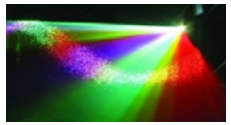
AbstractBufferedSoundBinaryOp (cont.)

```
}

public BufferedSound filter(BufferedSound src1,
                            BufferedSound src2,
                            BufferedSound dest)
                     throws IllegalArgumentException
{
   Iterator<double[]>   source1, source2, destination;


   // Check the properties of the two source sounds
   checkArguments(src1, src2);

   // Construct the destination if necessary; otherwise check it
   if (dest == null)
      dest = createCompatibleDestinationSound(src1);
   else
      checkArguments(src1, dest);


   // Get the source channels
   source1     = src1.getSignals();
   source2     = src2.getSignals();

   // Get the destination channels
   destination = dest.getSignals();
```



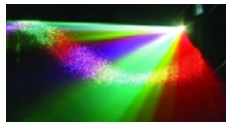
AbstractBufferedSoundBinaryOp (cont.)

```
        // Apply the filter
        applyFilter(source1, source2, destination);

        return dest;
    }
}
```

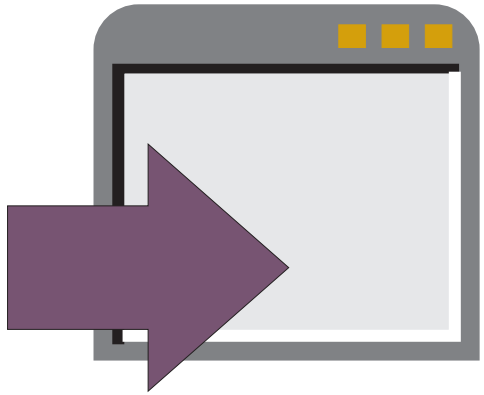


AddOp

```
package auditory.sampled;

public class AddOp extends    AbstractBufferedSoundBinaryOp
{
    public void applyFilter(double[] source1, double[] source2,
                            double[] destination)
    {
       for (int i=0; i<source1.length; i++)
       {
             destination[i] = source1[i] + source2[i];
       }
    }
}
```



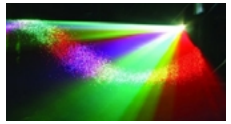
AddOp – Demonstration



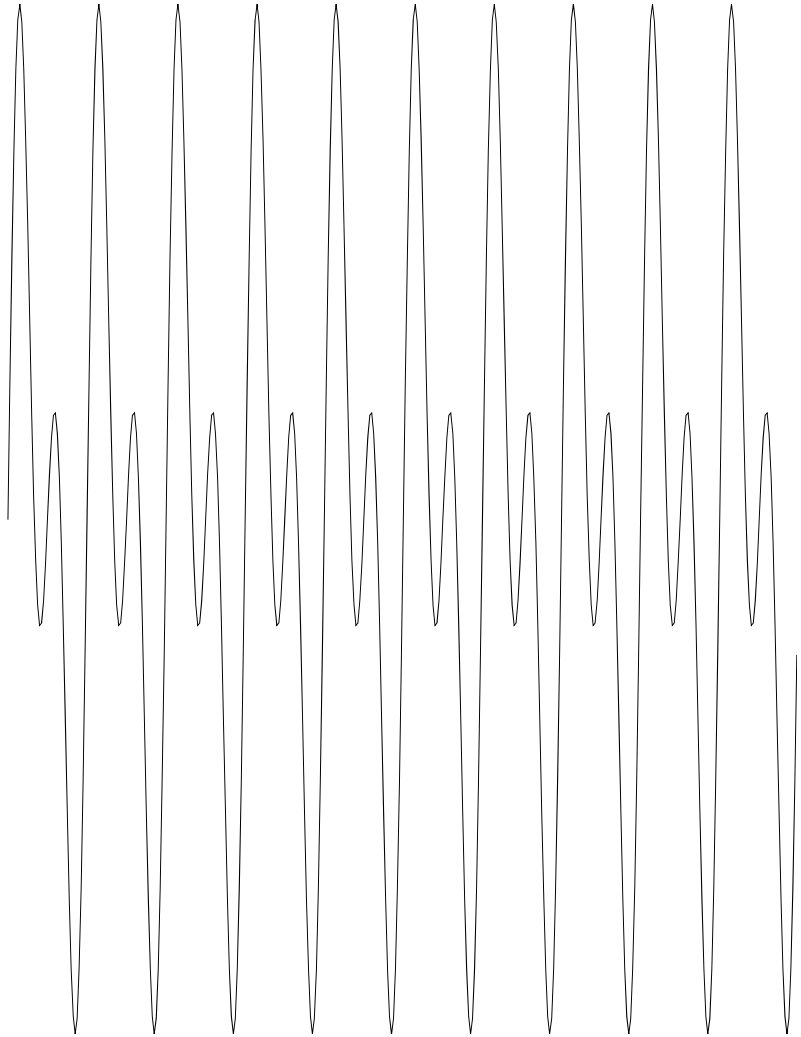
In extras:

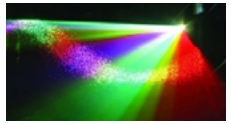
AddOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication BINARY 100 ADD 200
```

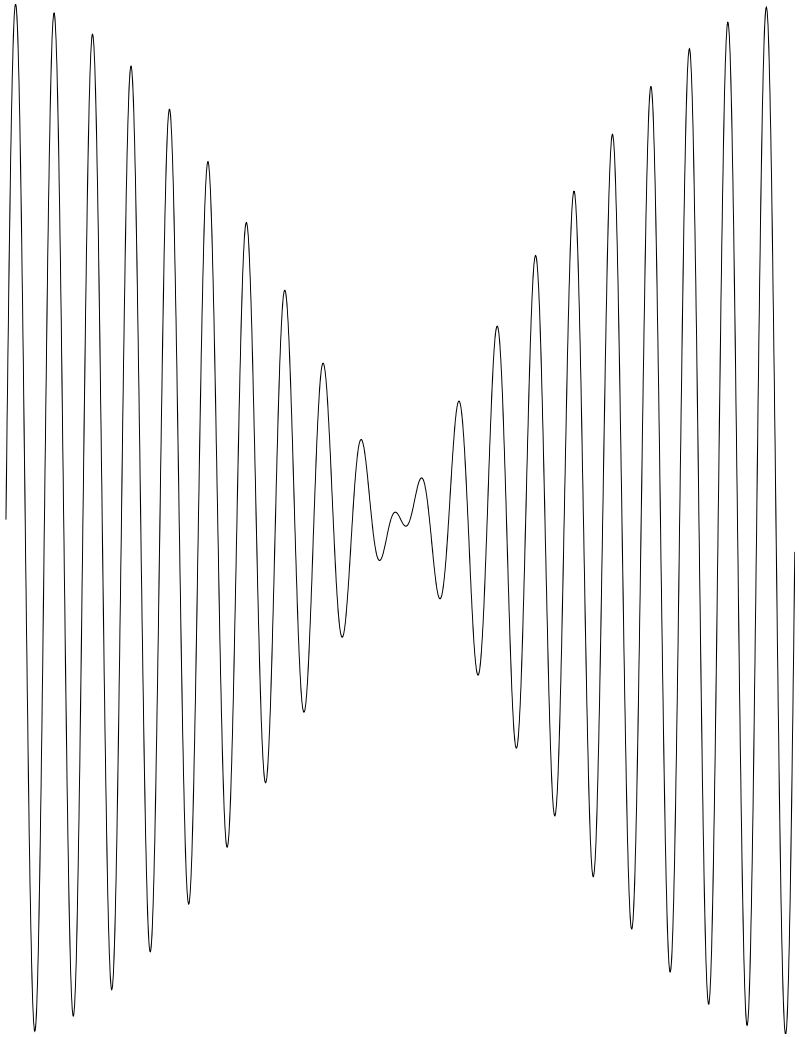


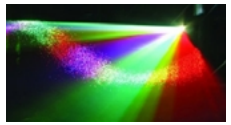
The Result of Adding 100Hz and 200Hz Sine Waves



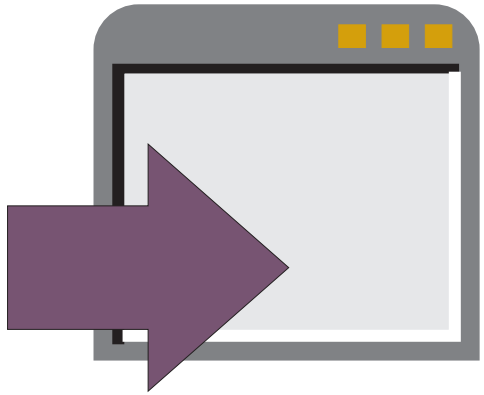


Adding 100Hz and 105Hz Sine Waves – Beating





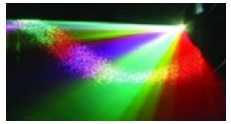
AddOp – Demonstration



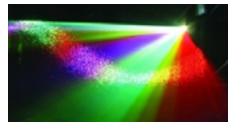
In extras:

Beating.html

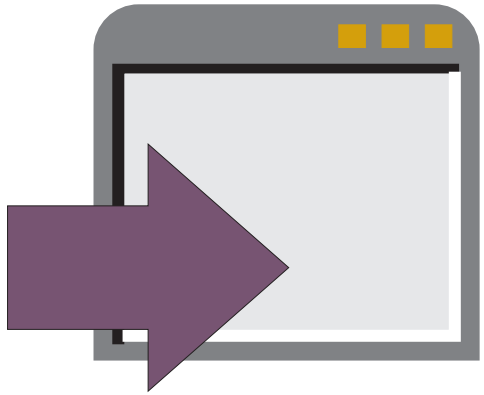
```
java -cp BufferedSound.jar BufferedSoundApplication BINARY 100 ADD 105
```



```
package auditory.sampled;

public class ReverseOp extends    AbstractBufferedSoundUnaryOp
{
    public void applyFilter(double[] source, double[] destination)
    {
       int       length;

       length    = source.length;

       for (int i=0; i<length; i++)
       {
          destination[i]  = source[length-1-i];
       }
    }
}
```



ReverseOp – Demonstration



In extras:

Original

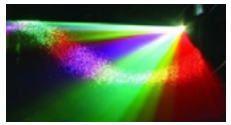
Number9.html

```
java -cp FilePlayer.jar FilePlayerApplication number9.aif
```

Reversed

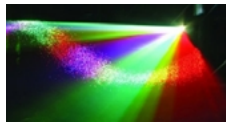
Reverse.html

```
java -cp BufferedSound.jar BufferedSoundApplication UNARY number9.aif REVERSE
```

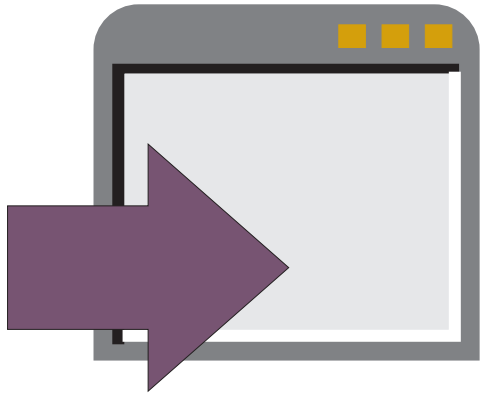


InvertOp

```
package auditory.sampled;

public class InvertOp extends    AbstractBufferedSoundUnaryOp
{
    public void applyFilter(double[] source, double[] destination)
    {
       int       length;

       length    = source.length;

       for (int i=0; i<length; i++)
       {
          destination[i]  = -source[i];
       }
    }
}
```



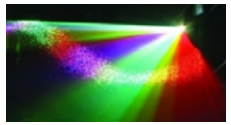
InvertOp – Demonstration



In extras:

InvertOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication UNARY preface.aif INVERT
```



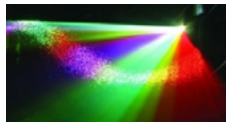
Categorizing Filters

- Causal:

  Use only sample points ‘before’ the current point.

- Non-Causal:

  Can use sample points ‘after’ the current point.



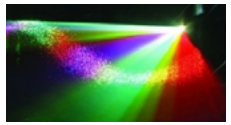
Categorizing Filters (cont.)

- Finite:

  Only use the source.

- Infinite:

  Use both the source and the destination.



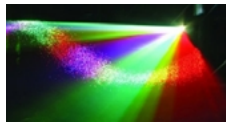
Categorizing Filters (cont.)

- Linear:

  Only combine the sample points using addition and multiplication by a constant.

- Non-Linear:

  Combine sample points in any fashion.



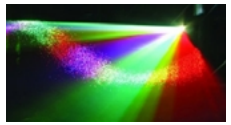
Categorizing Filters (cont.)

- Time-Invariant:

  Do not change over time.

- Adaptive:

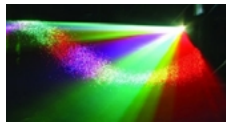
  Change over time.



Categorizing Filters (cont.)

Letting d denote the destination, s denote the source, and w and v denote weights, an *infinite, linear, causal filter* is a filter than can be expressed as follows:

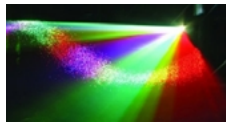
$$d_i = \sum_{k=0}^{n} s_{i-k} w_k + \sum_{j=0}^{m} d_{i-j} v_j \text{ for all } i$$



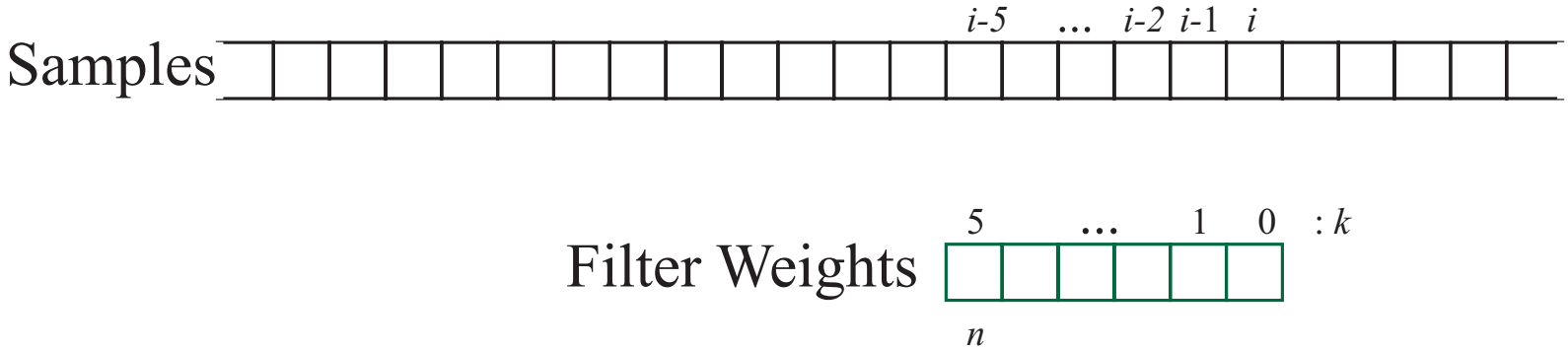
Categorizing Filters (cont.)

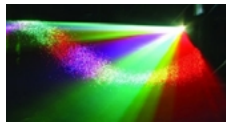
A *finite, linear causal filter* (which is also called a *finite impulse response* or FIR filter) is a filter that can be expressed as follows:

$$d_i = \sum_{k=0}^{n} s_{i-k} w_k \text{ for all } i$$



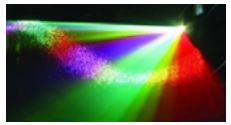
A FIR Filter





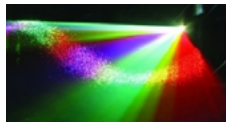
FIRFilter – Structure

```
package auditory.sampled;

public class FIRFilter
{
    private double[]     weights;

    public FIRFilter(double[] weights)
    {
       this.weights = new double[weights.length];
       System.arraycopy(weights, 0, this.weights, 0, weights.length);
    }
}
```



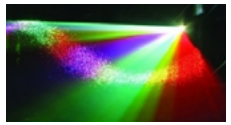
FIRFilter – Getters

```
public int getLength()
{
   int   length;

   length = 0;
   if (weights != null) length = weights.length;

   return length;
}
public double getWeight(int index)
{
   double   weight;

   weight = 0.0;
   if ((weights == null) && (index == weights.length-1))
   {
      weight = 1.0;
   }
   else if ((index >=0) && (index < weights.length-1))
   {
      weight = weights[index];
   }

   return weight;
}
```



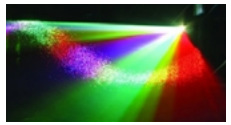
FIRFilterOp – Structure

```
package auditory.sampled;

public class FIRFilterOp extends AbstractBufferedSoundUnaryOp
{
    private FIRFilter          fir;

    public FIRFilterOp(FIRFilter fir)
    {
       this.fir = fir;
    }
}
```

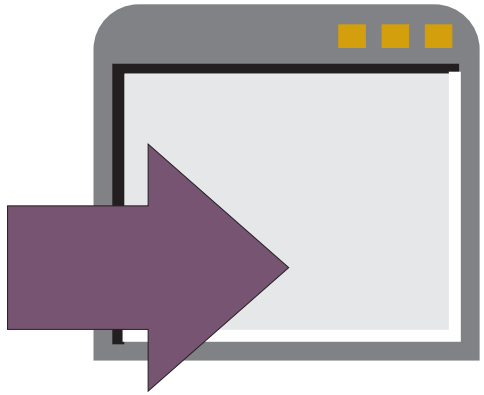


FIRFilterOp – applyFilter()

```
public void applyFilter(double[] source, double[] destination)
{
   double    weight;
   int       length, n;

   n         = fir.getLength();
   length    = source.length;

   // Copy the first n-2 samples
   for (int i=0; i<n-1; i++)
   {
      destination[i]  = source[i];
   }

   // Filter the remaining samples
   for (int i=n-1; i<length; i++)
   {
      for (int k=0; k<n; k++)
      {
         weight        = fir.getWeight(k);

         destination[i]  += source[i-k]  * weight;
      }
   }
}
```



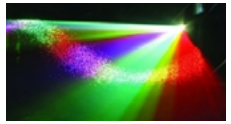
FIRFilterOp – Demonstration



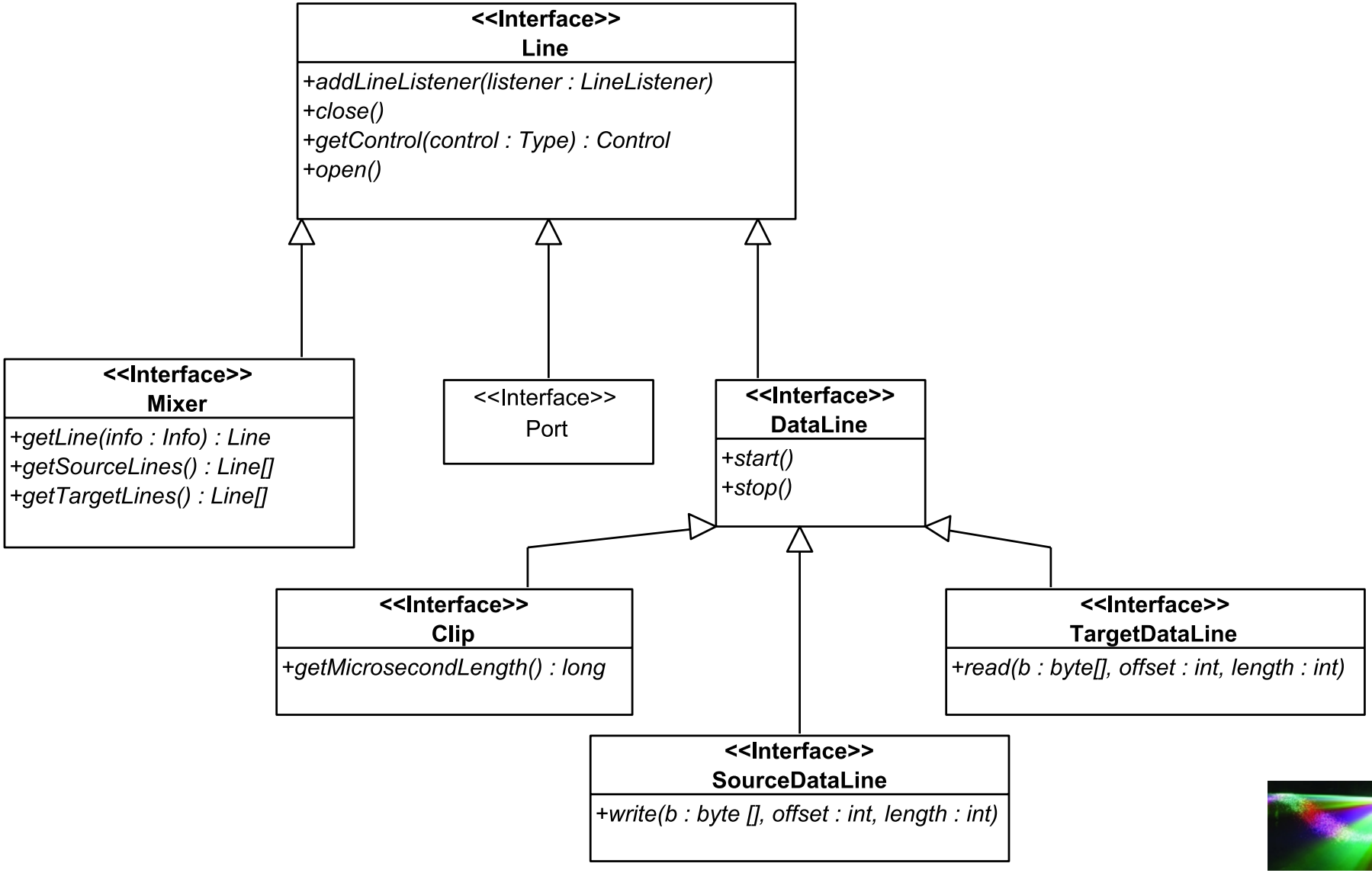
In extras:

FIRFilterOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication UNARY preface.aif FIR
```

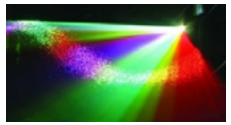


The Java Sound API



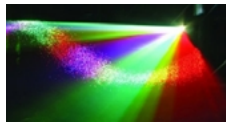
BoomBox – Structure

```
package auditory.sampled;

import java.util.*;
import javax.sound.sampled.*;

public class BoomBox implements LineListener
{
    private Content         content;
    private Clip            clip;
    private final Object    sync = new Object();

    public BoomBox(Content content)
    {
       this.content = content;
    }
}
```



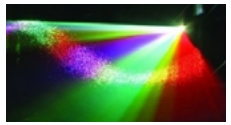
BoomBox – Listeners

```
private Vector<LineListener> listeners = new Vector<LineListener>();
```



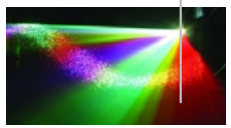
BoomBox – Managing Listeners

```
public void addLineListener(LineListener listener)
{
   listeners.add(listener);
}

public void removeLineListener(LineListener listener)
{
   listeners.remove(listener);
}
```



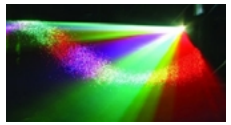
BoomBox – update()

```
public void update(LineEvent evt)
{
   Enumeration      e;
   LineEvent.Type   type;
   LineListener     listener;

   synchronized(sync)
   {
      // Forward the LineEvent to all LineListener objects
      e = listeners.elements();
      while (e.hasMoreElements())
      {
         listener = (LineListener)e.nextElement();
         listener.update(evt);
      }

      // Get the type of the event
      type = evt.getType();

      // Process STOP events
      if (type.equals(LineEvent.Type.STOP))
      {
         sync.notifyAll();
         clip.close();
         clip.removeLineListener(this);
         clip = null;
      }
   }
```



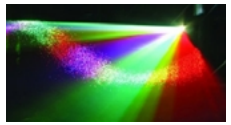
BoomBox – update() (cont.)

```
    }
}
```



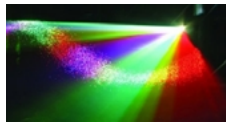
BoomBox – Rendering

```
public void start(boolean block)
            throws LineUnavailableException
{
   Clip                clip;

   clip = AudioSystem.getClip();
   clip.addLineListener(this); // So the calling thread can be informed
   content.render(clip);
   synchronized(sync)
   {
      // Wait until the Clip stops [and notifies us by
      // calling the update() method]
      if (block)
      {
         try
         {
            sync.wait();
         }
         catch (InterruptedException ie)
         {
            // Ignore
         }
      }
   }
}
```



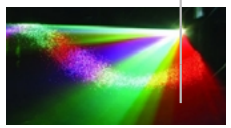
BufferedSound - Scaling

```
private short scaleSample(double sample)
{
   short     scaled;

   if      (sample > MAX_AMPLITUDE) scaled=(short)MAX_AMPLITUDE;
   else if (sample < MIN_AMPLITUDE) scaled=(short)MIN_AMPLITUDE;
   else                             scaled=(short)sample;

   return scaled;
}
```



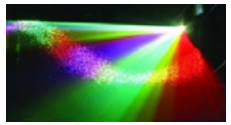
BufferedSound – Rendering

```
public synchronized void render(Clip clip)
                    throws LineUnavailableException
   size   = channels.size();
   length = getNumberOfSamples();
   frameSize  = format.getFrameSize();

   //  bytes           samples/channel *  bytes/channel     *  channels
   rawBytes = new byte[length          *  BYTES_PER_CHANNEL *     size];
   channel  = 0;
   iterator = channels.iterator();
   while (iterator.hasNext())
   {
      signal = iterator.next();
      offset = channel * BYTES_PER_CHANNEL;

      for (int i=0; i<length; i++)
      {
         scaled = scaleSample(signal[i]);

         // Big-endian
         rawBytes[frameSize*i+offset]   = (byte)(scaled >> 8);
         rawBytes[frameSize*i+offset+1] = (byte)(scaled & 0xff);

         // Little-endian
         // rawBytes[frameSize*i+offset+1] = (byte)(scaled >> 8);
         // rawBytes[frameSize*i+offset]   = (byte)(scaled & 0xff);
      }
      ++channel;
```



BufferedSound – Rendering (cont.)

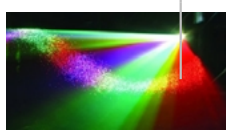
```
        }
        // Throws LineUnavailableException
        clip.open(format, rawBytes, 0, rawBytes.length);

        // Start the Clip
        clip.start();
```



NoiseOp

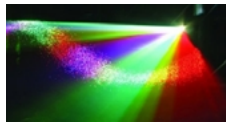
Recall that noise is a signal that is generated by a random process.

```
package auditory.sampled;

import java.util.Random;

public class NoiseOp extends    AbstractBufferedSoundUnaryOp
{
    private double        max;
    private Random        rng;

    public NoiseOp(double max)
    {
       this.max = max;
       rng = new Random(System.currentTimeMillis());

    }

    public void applyFilter(double[] source, double[] destination)
    {
       int       length;

       length    = source.length;

       for (int i=0; i<length; i++)
       {
          destination[i]  = source[i]  + (max - rng.nextDouble()*max*2.0);
```

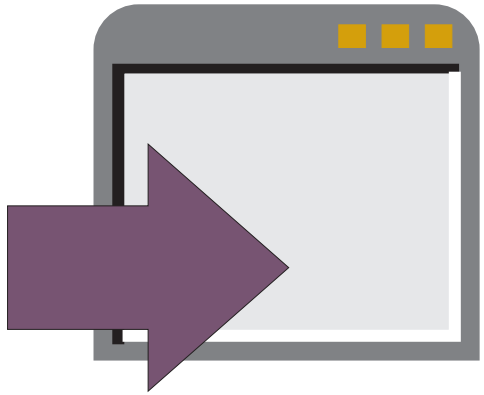


NoiseOp (cont.)

```
        }
    }
}
```



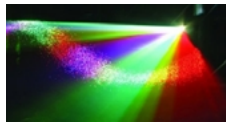
NoiseOp – Demonstration



In extras:

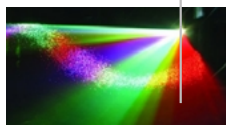
NoiseOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication UNARY preface.aif NOISE
```



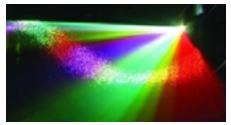
SpeedChangeOp

```
package auditory.sampled;

public class SpeedChangeOp extends AbstractBufferedSoundUnaryOp
{
    private double        multiplier;

    public SpeedChangeOp(double multiplier)
    {
       this.multiplier = multiplier;
    }

    public BufferedSound createCompatibleDestinationSound(BufferedSound src)
    {
       BufferedSound        temp;
       float                sampleRate;
       int                  channels, length;

       channels   = src.getNumberOfChannels();
       length     = src.getNumberOfSamples();
       sampleRate = src.getSampleRate() * (float)multiplier;

       temp = new BufferedSound(sampleRate);

       for (int i=0; i<channels; i++)
       {
          temp.addChannel(new double[length]);
       }
```

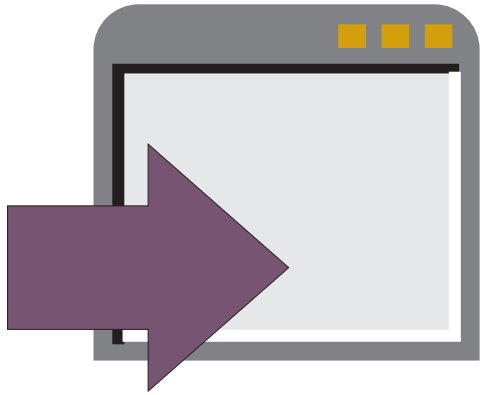


SpeedChangeOp (cont.)

```
        return temp;
    }

    public void applyFilter(double[] source, double[] destination)
    {
        int       length;

        length    = source.length;

        for (int i=0; i<length; i++)
        {
            destination[i]  = source[i];
        }
    }
}
```



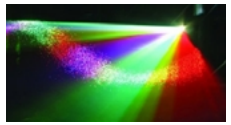
SpeedChangeOp – Demonstration



In extras:

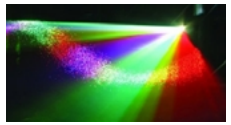
SpeedChangeOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication UNARY preface.aif SPEEDCHANGE
```

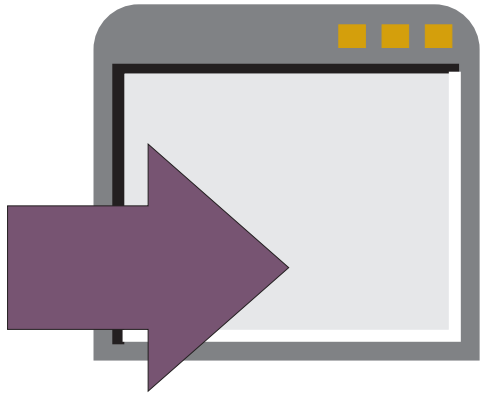


MultiplyOp

```
package auditory.sampled;

public class MultiplyOp extends AbstractBufferedSoundBinaryOp
{
    public void applyFilter(double[] source1, double[] source2,
                            double[] destination)
    {
       for (int i=0; i<source1.length; i++)
       {
             destination[i] = source1[i] * source2[i];
       }
    }
}
```



MultiplyOp – Demonstration



In extras:

MultiplyOp.html

```
java -cp BufferedSound.jar BufferedSoundApplication BINARY 100 MULTIPLY 200
```