Chapter 4. Distortion

In music, *distortion* is the sound effect that occurs when the amplitude peaks of a signal are compressed or cut off when the audio equipment is overloaded. Chapter 6 of volume 1 contains the mathematics behind the distortion effect described below.

There are various types of distortion. The signal peaks may be cut off, which is called a *clip* or a *hard clip*. The signal peaks may be compressed and then cut off, which is called a *soft clip*. The signal peaks may also be simply compressed. These are the three types of distortion implemented in this chapter.

4.1. Implementation of the distortion

The following is an implementation of the simple delay in the Orinj framework, assuming one channel of audio with the sampling resolution of 16 bits. The graphical user interface for this delay is described in the sections that follow.

Code 11. Distortion

```
package mycompany.com;

import java.io.*;
import javax.sound.sampled.AudioFormat;
import recordingblogs.com.oreffect.*;

public class Distortion extends Undo implements EffectInterface
{
   public static final String [] TYPES = {"Hard clip", "Soft clip", "No clip"};

   public static final int HARDCLIP = 0;
   public static final int SOFTCLIP = 1;
   public static final int NOCLIP = 2;

   public static final float MINTHRESHOLD = -50F;
   public static final float MAXTHRESHOLD = 0F;

   private int m_type;
   private float m_threshold;

   public Distortion()
   {
      m_type = HARDCLIP;
      m_threshold = -15F;
   }

   public int getType()
   {
```

```
    return m_type;
}

public void setType(int type)
{
    m_type = type;
}

public float getThreshold()
{
    return m_threshold;
}

public void setThreshold(float threshold)
{
    m_threshold = threshold;
}

public boolean allowsDryWetMix()
{
    return true;
}

public boolean allowsSideChaining()
{
    return false;
}

public boolean readObject(ReadInterface ar)
{
    try
    {
        m_type = ar.readInt();
        m_threshold = ar.readFloat();
    }
    catch (IOException e)
    {
        return false;
    }

    return true;
}

public boolean writeObject(WriteInterface ar)
{
    try
    {
        ar.writeInt(m_type);
```

```
            ar.writeFloat(m_threshold);
        }
        catch (IOException e)
        {
            return false;
        }

        return true;
    }

    public void setEqual(EffectInterface effect)
    {
        if (this == effect)
            return;
        if (effect.getClass() != this.getClass())
            return;
        Distortion e = (Distortion) effect;
        m_type = e.m_type;
        m_threshold = e.m_threshold;
    }

    public void startPlay()
    {
    }

    public void stopPlay()
    {
    }

    public boolean hasData()
    {
        return false;
    }

    public void setLanguage(String language)
    {
    }

    public void apply(byte [] dry, byte [] wet, byte [] control,
        AudioFormat format, double time)
    {
        // Initialize variables.  blockAlign is the size of the block align in the
        // audio (the number of bytes for one sample from all channels).  The
        // computation here is applicable to any sampling resolution or rate. sFrom
        // is the value of one sample in the incoming audio data.  sTo is the value
        // of one sample of the audio data produced by the effect.  sign is the
        // sign of the sample sFrom
```

```
int blockAlign = (format.getChannels() * format.getSampleSizeInBits()) / 8;
int channels = format.getChannels();
float sFrom = 0;
float sTo = 0;
int sign = 1;

// Convert the threshold from decibels to a multiple.  For example, a gain
// of about 6 decibels will be translated to a multiple of approximately 2
float threshold = (float) Math.pow(10, m_threshold / 20);

// Loop through the samples in the signal.  In a mono 16-bit signal,
// blockAlign is 2 bytes
for(int i = 0; i < dry.length; i += blockAlign)
{
   // Compute the value and sign of each incoming sample
   sFrom = (float) ((short) (((dry[i + 1] & 0xff) << 8) + (dry[i] & 0xff)))
     / Short.MAX_VALUE;
   sign = (sFrom >= 0) ? 1 : -1;

   // Compute the value of output at the sample.  It is better if this
   // check encompasses and repeats the 'for' loop, so that the check is
   // not performed at every sample, but this version is easier to read
   sTo = 0;
   if (m_type == HARDCLIP)
   {
      // Cut the peaks of the signal at the threshold
      sTo = sign * Math.min(Math.abs(sFrom), threshold);
   }
   else if (m_type == SOFTCLIP)
   {
      // Apply the cubic soft clipper
      if (Math.abs(sFrom) <= threshold)
         sTo = sFrom - (sFrom * sFrom * sFrom / 3);
      else
         sTo = sign * (threshold - (threshold * threshold * threshold
            / 3));
   }
   else if (m_type == NOCLIP)
   {
      // Condense the peaks above the threshold
      if (Math.abs(sFrom) <= threshold)
         sTo = sFrom;
      else
         sTo = sign * (threshold + (float) Math.atan((Math.abs(sFrom) –
            threshold) / threshold) * threshold);
   }

   // Ensure that the computed output is within the limits of the bit
```

```
            // resolution
            sTo = Math.min(Math.max(sTo * Short.MAX_VALUE, Short.MIN_VALUE),
               Short.MAX_VALUE);

            // Store the computed output sample in the output buffer
            wet[i] = (byte) ((int) sTo & 0xff);
            wet[i + 1] = (byte) ((int) sTo >>> 8 & 0xff);
         }
      }
   }
```

This class implements **`EffectInterface`**. With **`EffectInterface`**, Orinj can use many different effects without knowing exactly what they are.

`String [] TYPES` – These strings describe the types of distortion implemented here.

`int HARDCLIP` – This is a flag to use hard clip distortion.

`int SOFTCLIP` – This is a flag to use soft clip distortion.

`int NOCLIP` – This is a flag to use no clip distortion.

`float MINTHRESHOLD` – This is the minimum threshold in decibels that the user can choose, above which signal peaks will be clipped or condensed. The actual threshold chosen by the user is **`m_threshold`** below. **`MINTHREHSHOLD`** is used only to set up the graphical user interface and when checking user input for errors.

`float MAXTHRESHOLD` – This is the maximum threshold in decibels that the user can choose, above which signal peaks will be clipped or condensed.

`int m_type` – This is the type of distortion chosen by the user. It can be **`HARDCLIP`**, **`SOFTCLIP`**, or **`NOCLIP`**.

`float m_threshold` – This is the threshold, above which signal peaks will be clipped or condensed. It should be between **`MINTHRESHOLD`** and **`MAXTHRESHOLD`**.

`Distortion()` – This is the default and only constructor.

`int getType()` – This function returns the type of distortion **`m_type`**.

`void setType(int type)` – This function sets the type of distortion **`m_type`** to **`type`**.

`float getThreshold()` – This function returns the threshold **`m_threshold`**, above which peaks will be clipped or condensed.

`void setThreshold(float threshold)` – This function sets the threshold **`m_threshold`**, above which peaks will be clipped or condensed, to **`threshold`**.

`boolean allowsDryWetMix()` – This function is required by **`EffectInterface`**. It simply returns **`true`**, as separating the dry and wet signal in a simple delay is easy and doing so allows Orinj to

mix those two signals at different levels if the user chooses to do so. Of course, the **apply** function below must be implemented accordingly. In most cases, Orinj users will probably mix the wet signal at its full amplitude and zero out the dry signal. Nonetheless, allowing a dry wet mix means that the user can change the combination of the two signals and make the distortion less audible.

boolean allowsSideChaining() – This function is required by **EffectInterface**. This effect does not use information from another signal and therefore does not allow side chaining. This function simply returns **false**.

void startPlay() – This function is required by **EffectInterface**. This function must implement actions that should be taken at the beginning of playing. There is nothing that needs to be done in this class.

void stopPlay() – This function is required by **EffectInterface**. Most Orinj DSP effects will not make use of this function.

boolean hasData() – This function is required by **EffectInterface**. In this class, this function should simply return **false**. This function is discussed further in other effects below.

void setLanguage(String language) – This function is required by **EffectInterface**. This function allows Orinj to set the language for effect controls and tooltips to the language selected by the user in the Orinj preferences. **language** is the ISO 639-2 three-letter code for the chosen language. Effects do not have to implement this function. It is not implemented above.

boolean writeObject(WriteInterace ar) – This function is required by **EffectInterface**. This function stores the effect member data for serialization and for undo. It returns **true** if there are no errors and **false** if there are errors. As above, **ar** can be used approximately as an object of class **java.io.ObjectOutputStream**.

boolean readObject(ReadInterace ar) – This function is required by **EffectInterface**. This function reads the effect member data from serialization and for undo. It returns **true**, if there are no errors and **false**, if there are errors. **ar** can be used approximately as an object of class **java.io.ObjectInputStream**.

void setEqual(EffectInterace effect) – This function is required by **EffectInterface**. This function sets the member data of the current effect equal to the member data of **effect**. This function is not used in the current version of Orinj, but may be used in future versions.

void apply(byte [] dry, byte [] wet, byte [] control, AudioFormat format, double time) – This function is required by **EffectInterface**. It applies the effect to the audio data contained in **dry**. To allow Orinj to mix the dry and wet audio data, **dry** is left unchanged and the delayed and decayed repetition of the signal is placed in **wet**. **dry**, **wet**, and **control** are of the same length and with audio data in the format specified by **format**. This effect does not allow side chaining and **control** is therefore not used. **dry** does not contain all audio data that are sent to the effect. Since audio data can be large, they are sent to the effect in pieces.

As with all effects throughout this book, this implementation only works for the sampling resolution of 16 bits. Other sampling resolutions are briefly discussed below.

While this implementation is appropriate for both mono (single channel) and stereo (two channel) signals, other effects must work with each channel separately. Working with mono and stereo signals is further described below.

This effect allows dry / wet mix: adjustments in the relative levels of the dry and wet signal. Orinj will take both the dry and wet signals and will mix those according to the dry and wet levels specified by the user. If an effect does not allow dry / wet mix, Orinj will take only the wet data and ignore the dry data in further processing.

This effect does not allow side chaining. A side chained effect is one that uses other sound data to control the sound data that are changed. A side chained compressor, for example, changes the dynamics of one track based on the dynamics of a second track. The audio data contained in the second track would be sent to the effect in the parameter control. In an effect that does not allow side chaining, control may be null and should not be used.

4.2. Using audio buffers

Take a two-and-a-half-minute song, recorded in two channels with the sampling rate 44100 Hz and sampling resolution 16 bits (2 bytes per sample per channel). The ready mixed song will contain 2.5 * 60 * 44100 * 2 * 2 = 26,460,000 bytes ≈ 25 Mb. Now suppose that you are recording this song in at least four tracks. The audio data for this short and simple recording could easily exceed 100 Mb. Actual recording sessions may be even larger.

Buffers allow audio software applications to keep audio data on disk, rather than in memory, and to only access and process portions of these data. Of course, it does not make sense to access and process the data byte by byte. This would require too many disk access operations, which would be slow. Instead, data are read in buffers that are larger than a single byte, but smaller than the whole song.

Using audio buffers in the implementation of this distortion is easy, as the distortion is computed from information contained only in the current buffers. This is not so with other effects. All remaining effects in this book must use not only the current buffers, but also past incoming audio buffers. They must therefore store and manage past audio buffers. Examples of how this can be done are presented in the chapters that follow.

4.3. Other sampling rates and resolutions

There is no need for changes in the code to handle other sampling rates. Adjustments are needed for other sampling resolutions.

Note the following code snippets. These convert byte data into audio samples and audio samples into bytes. In the distortion effect, there is additional scaling by **`Short.MAX_VALUE`** to

ensure that the effect works with floating-point numbers between -1 and 1, but the important portions of the code are here below.

Code 12. Bytes to 16-bit samples and 16-bit samples to bytes

```
sFrom = (short) (((bufferFrom[curByte + 1] & 0xff) << 8)
   + (bufferFrom[curByte] & 0xff));
...
wet[i] = (byte)(sTo & 0xff);
wet[i + 1] = (byte)(sTo >>> 8 & 0xff);
```

Each sample in a 16-bit sampling resolution is stored in 16 / 8 = 2 bytes. Since WAVE files are little endian, the least significant bit is first and can be used as it is. The second byte is the most significant byte and must be up-shifted by 8 bits. The sum of the two is the audio sample. The bit masks (0xff) are there to remove the sign – otherwise each byte will be treated as a signed byte – and the typecasting to short introduces the sign into the two-byte value. The opposite steps are taken when converting the value **`sTo`** to bytes.

Note that some (probably most) audio applications convert byte data to floating-point data and, internally, work only with floating-point buffers. In such an application, the apply function may have floating-point buffer arguments. This may be beneficial, as it would mean that the code does not have to be adjusted to handle other sampling resolutions. The code above does need changes.

The corresponding code for 8-bit (unsigned) audio data is as follows. The use of 128 is here, since this would be the middle of a signal oscillating between 0 and 256 in unsigned data represented by one byte samples. In addition, 128 represents silence, unlike with 16, 24, and 32 bits, where silence is zero. Note also that the limits applied to **`sTo`** in 8-bit computations are **`Byte.MIN_VALUE`** and **`Byte.MAX_VALUE`**, rather than **`Short.MIN_VALUE`** and **`Short.MAX_VALUE`**.

Code 13. Bytes to 8-bit samples and 8-bit samples to bytes

```
sFrom = (bufferFrom[curByte] & 0xff) -128;
...
wet[i] = (byte)((sTo + 128) & 0xff);
```

The corresponding code for 24-bit audio data is as follows. The limits applied to 24-bit data are ±8388607. The upshifting by additional 8 bits than what might be expected and the downshifting by eight bits is to preserve the sign (i.e., up to a four-byte integer before downshifting to a three-byte integer).

Code 14. Bytes to 24-bit samples and 24-bit samples to bytes

```
sFrom = ((bufferFrom[curByte + 2] & 0xff) << 24)
   + ((bufferFrom[curByte + 1] & 0xff) << 16)
   + ((bufferFrom[curByte] & 0xff) << 8) >> 8;
...
wet[i] = (byte)(sTo & 0xff);
wet[i + 1] = (byte)(sTo >>> 8 & 0xff);
```

```
wet[i + 2] = (byte)(sTo >>> 16 & 0xff);
```

When working with 32-bit data, Orinj expects floating-point numbers. The byte-to-sample and back conversions are as follows (**`fFrom`** and **`fTo`** are floating-point numbers and **`iTo`** is an integer). The maximum and minimum values for the samples are ±1.

Code 15. Bytes to 32-bit samples and 32-bit samples to bytes

```
fFrom = Float.intBitsToFloat(
   ((bufferFrom[curByte + 3] & 0xff) << 24)
   + ((bufferFrom[curByte + 2] & 0xff) << 16)
   + ((bufferFrom[curByte + 1] & 0xff) << 8)
   + (bufferFrom[curByte] & 0xff));
...
iTo = Float.floatToIntBits(fTo);
wet[i] = (byte)(iTo & 0xff);
wet[i + 1] = (byte)(iTo >>> 8 & 0xff);
wet[i + 2] = (byte)(iTo >>> 16 & 0xff);
wet[i + 3] = (byte)(iTo >>> 24 & 0xff);
```

Subsequent effects are only implemented on 16-bit data and we do not discuss 8-, 24- or 32-bit data below. These conversions from byte arrays to integer or floating-point data can be used in each of the effects below, to create the appropriate bit resolution implementation.

4.4. Error checking

To simplify the code above, there is no checking for errors. Checks could be added at least to functions that set the values of member data of the distortion. For example:

Code 16. Example error checking in the simple delay

```
public void setThreshold(float threshold)
{
   if (threshold < MINTHRESHOLD || threshold > MAXTHRESHOLD)
   {
      // Show some error if needed
   }
   threshold = Math.min(MAXTHRESHOLD, Math.max(MINTHRESHOLD, threshold));
   m_threshold = threshold;
}
```

While it is beneficial to implement such checks in effects, for brevity, we do not discuss these types of error checks below.

4.5. Implementation of the distortion graphical user interface

The following is an implementation of the graphical user interface for the distortion in the Orinj framework.

Code 17. Distortion graphical user interface

```
package mycompany.com;

import java.awt.*;
import java.awt.event.*;
import java.util.Hashtable;
import javax.swing.*;
import javax.swing.event.*;
import javax.sound.sampled.*;
import recordingblogs.com.oreffect.*;

public class DistortionPanel extends JPanel implements EffectPanelInterface,
   ActionListener, ChangeListener, FocusListener
{
   private JComboBox<String> m_typeCombo;
   private JTextField m_thresholdField;
   private JSlider m_thresholdSlider;

   private Distortion m_distortion;

   public class SliderLabel extends JLabel
   {
      public SliderLabel(String string)
      {
         super(string);
         setFont(EffectFont.SMALLFONT);
      }
   }

   private class FocusPolicy extends
      ContainerOrderFocusTraversalPolicy
   {
      public Component getDefaultComponent(Container c)
      {
         return m_typeCombo;
      }

      public Component getLastComponent(Container c)
      {
         return m_thresholdSlider;
      }

      public Component getFirstComponent(Container c)
```

```
    {
       return m_typeCombo;
    }

    public Component getComponentBefore(Container c, Component a)
    {
       if (a == m_thresholdSlider)
          return m_thresholdField;
       if (a == m_thresholdField)
          return m_typeCombo;
       if (a == m_typeCombo)
          return m_thresholdSlider;
       return m_typeCombo;
    }

    public Component getComponentAfter(Container c, Component a)
    {
       if (a == m_typeCombo)
          return m_thresholdField;
       if (a == m_thresholdField)
          return m_thresholdSlider;
       if (a == m_thresholdSlider)
          return m_typeCombo;
       return m_typeCombo;
    }
}

public DistortionPanel(Distortion distortion, AudioFormat format)
{
   GridBagLayout layout = new GridBagLayout();
   GridBagConstraints constraints = new GridBagConstraints();
   setLayout(layout);

   JLabel typeLabel = new JLabel("Type:");
   typeLabel.setFont(EffectFont.LARGEFONT);
   constraints.gridx = 0;
   constraints.gridy = 0;
   constraints.weightx = 0;
   constraints.weighty = 1;
   constraints.anchor = GridBagConstraints.NORTHWEST;
   constraints.fill = GridBagConstraints.BOTH;
   layout.setConstraints(typeLabel, constraints);
   add(typeLabel);

   m_typeCombo = new JComboBox<String>(Distortion.TYPES);
   m_typeCombo.addActionListener(this);
   m_typeCombo.addFocusListener(this);
```

```
m_typeCombo.setToolTipText("Set the type of distortion");
constraints.gridx = 2;
constraints.weightx = 1;
constraints.insets.left = 5;
layout.setConstraints(m_typeCombo, constraints);
add(m_typeCombo);

JLabel thresholdLabel = new JLabel("Threshold (dB):");
thresholdLabel.setFont(EffectFont.LARGEFONT);
constraints.gridx = 0;
constraints.gridy = 1;
constraints.weightx = 0;
constraints.insets.top = 5;
constraints.anchor = GridBagConstraints.NORTHWEST;
constraints.fill = GridBagConstraints.HORIZONTAL;
layout.setConstraints(thresholdLabel, constraints);
add(thresholdLabel);

m_thresholdField = new JTextField(4);
m_thresholdField.setToolTipText("Set the threshold");
m_thresholdField.addActionListener(this);
m_thresholdField.addFocusListener(this);
constraints.gridx = 1;
constraints.insets.left = 5;
layout.setConstraints(m_thresholdField, constraints);
add(m_thresholdField);

Hashtable<Integer, SliderLabel> numbersThreshold =
   new Hashtable<Integer, SliderLabel>();
numbersThreshold.put(new Integer(-50), new SliderLabel("-50"));
numbersThreshold.put(new Integer(-40), new SliderLabel("-40"));
numbersThreshold.put(new Integer(-30), new SliderLabel("-30"));
numbersThreshold.put(new Integer(-20), new SliderLabel("-20"));
numbersThreshold.put(new Integer(-10), new SliderLabel("-10"));
numbersThreshold.put(new Integer(0), new SliderLabel("0"));

m_thresholdSlider = new JSlider(JSlider.HORIZONTAL,
   (int) Distortion.MINTHRESHOLD, (int) Distortion.MAXTHRESHOLD, 0);
m_thresholdSlider.setMajorTickSpacing(10);
m_thresholdSlider.setMinorTickSpacing(5);
m_thresholdSlider.setPaintTicks(true);
m_thresholdSlider.setPaintLabels(true);
m_thresholdSlider.setFont(EffectFont.SMALLFONT);
m_thresholdSlider.setLabelTable(numbersThreshold);
m_thresholdSlider.setToolTipText("Set the threshold");
m_thresholdSlider.addChangeListener(this);
m_thresholdSlider.addFocusListener(this);
constraints.gridx = 2;
```

```
      constraints.weightx = 1;
      layout.setConstraints(m_thresholdSlider, constraints);
      add(m_thresholdSlider);

      m_distortion = distortion;

      setFocusTraversalPolicy(new FocusPolicy());
      updateData();
   }

   public void focusGained(FocusEvent e)
   {
   }

   public void focusLost(FocusEvent e)
   {
      if (e.getSource() == m_thresholdField)
      {
         m_distortion.setThreshold(Float.parseFloat(m_thresholdField.getText()));
         updateData();
      }
      else if(e.getSource() == m_thresholdSlider)
      {
         m_distortion.setThreshold((float) m_thresholdSlider.getValue());
      }
      else if (e.getSource() == m_typeCombo)
      {
         m_distortion.setType(m_typeCombo.getSelectedIndex());
      }
   }

   public void stateChanged(ChangeEvent e)
   {
      if(e.getSource() == m_thresholdSlider)
      {
         m_distortion.setThreshold((float) m_thresholdSlider.getValue());
         updateData();
      }
   }

   public void actionPerformed(ActionEvent e)
   {
      if(e.getSource() == m_thresholdField)
      {
         m_distortion.setThreshold(Float.parseFloat(m_thresholdField.getText()));
         updateData();
      }
      else if(e.getSource() == m_typeCombo)
```

```
      {
         m_distortion.setType(m_typeCombo.getSelectedIndex(), false);
      }
   }

   public void updateData()
   {
      m_typeCombo.setSelectedIndex(m_distortion.getType());
      m_thresholdSlider.setValue((int) m_distortion.getThreshold());
      m_thresholdField.setText(Float.toString(m_distortion.getThreshold()));
   }
}
```

The member data and functions of this class are as follows.

`JComboBox<String> m_typeCombo` – This combo box allows the user to select the type of distortion – hard clip, soft clip, or no clip.

`JTextField m_thresholdField` – This field allows the user to adjust the threshold of the distortion.

`JSlider m_thresholdSlider` – This slider also allows the user to adjust the threshold of the distortion. Using a slider and a text field is a choice. The slider is easier to use, but not as precise. In this graphical user interface, as one of these two controls changes, the other one is adjusted accordingly.

`Distortion m_distortion` – This is the distortion effect.

`class SliderLabel` – This is a simple class to allow the threshold slider to use **`EffectFont`** fonts for its labels.

`class FocusPolicy` – This is the focus order of the user interface. Defining the focus policy is a good idea either way, but the focus policy of this user interface is used by Orinj in the dialog that encompasses this interface when the effect is created by the user.

`DistortionPanel(Distortion distortion, AudioFormat format)` – This is the default and only constructor. Note the call to **`updateData`** at the end of the constructor, which sets the controls in this interface to the values of the effect controls.

`void focusLost(FocusEvent e)` – This function captures the focus leaving and control and sets the corresponding control in the effect to the value of the interface control.

`void stateChanged(ChangeEvent e)` – This function captures changes in the threshold slider.

`void actionPerformed(ActionEvent e)` – This function captures changes in the combo box and pressing ENTER in the text field.

`void updateData()` – This function sets the interface controls to the values of the corresponding controls in the effect.

4.6. Error checking in the effect panel interface

The user can enter erroneous data in the text fields, such as "-4t" instead of "-45" for the threshold or "45" where the threshold can only be negative. Error checking can be implemented in **focusLost** as follows (the following is only a portion).

Code 18. Error checks in the graphical user interface – focusLost

```
public void focusLost(FocusEvent e)
{
   if (e.getSource() == m_thresholdField)
   {
      try
      {
         float value = Float.parseFloat(m_thresholdField.getText());
         if (value < Distortion.MINTHRESHOLD || value > Distortion.MAXTHRESHOLD)
         {
            JOptionPane.showMessageDialog(this, "Please enter a number between "
               + Distortion.MINTHRESHOLD + " and " + Distortion.MAXTHRESHOLD,
               "Error", JOptionPane.ERROR_MESSAGE);
            updateData();
            m_thresholdField.grabFocus();
            return;
         }
         m_distortion.setThreshold(value);
         updateData();
      }
      catch (NumberFormatException ex)
      {
         JOptionPane.showMessageDialog(this, "Please enter a valid number",
            "Error", JOptionPane.ERROR_MESSAGE);
         updateData();
         m_thresholdField.grabFocus();
      }
   }
   ...
}
```

This code will show the first error if the user enters a number outside of the minimum and maximum allowed threshold values. The code will show the second error if the user enters a non-number. Without this error checking, the effect – specifically the conversion of the text field text to a floating-point value – may throw and uncaught exception.

Checking for errors in text fields should also be performed in **actionPerformed**. The following is an example. Note that on errors **actionPerformed** simply changes the current control focus to another control, which means that the error is caught and a message is displayed by **focusLost**.

Code 19. Error checks in the graphical user interface – actionPerformed

```
public void actionPerformed(ActionEvent e)
{
   if(e.getSource() == m_thresholdField)
   {
      try
      {
         float value = Float.parseFloat(m_thresholdField.getText());
         if (value < Distortion.MINTHRESHOLD || value > Distortion.MAXTHRESHOLD)
         {
            m_thresholdSlider.grabFocus();
            return;
         }
         m_distortion.setThreshold(value, false);
         updateData();
      }
      catch (NumberFormatException ex)
      {
         m_thresholdSlider.grabFocus();
      }
   }
   ...
}
```

The code samples in the rest of the book do not include such error checks. This is done for brevity. These error checks are needed. They may be implemented as they are implemented here or differently.

4.7. Undo

To implement undo in the Orinj effect framework, when setting **`m_threshold`** and **`m_type`**, the effect class **`Distortion`** can fire an undo event as in the following example.

Code 20. Undo in the distortion when setting the threshold

```
public void setThreshold(float threshold, boolean storeUndo)
{
   if (threshold != m_threshold && storeUndo)
      fireEffectUndoEvent(new EffectUndoEvent(this,
         "Undo Distortion Threshold"));
   m_threshold = threshold;
}
```

This is possible, because **`Distortion`** extends **`Undo`**.

Note the addition of **`storeUndo`** in the function arguments. **`storeUndo`** is useful if we do not want to fire an undo event every time the threshold changes. In Orinj, for example, effect

member data change with every movement of a corresponding control slider. This allows the user to hear changes in the effect as the slider moves during playback. However, an undo event is created only when the focus moves away from the slider and not on every movement of the slider (i.e., when the user moves to a different control).

In other words, **stateChanged** in the implementation of **DistortionPanel** calls **setThreshold** with **storeUndo** equal to **false**, but **focusLost** calls **setThreshold** with **storeUndo** equal to **true**. Note also that without **storeUndo**, there is no need to implement **focusLost** on the threshold slider as the threshold is set to its appropriate value by **stateChanged**.

> Implementing undo is not required. It is the programmer's choice. The code samples in the sections that follow do not implement undo.

4.8. Implementation in effect.xml

The following is an example of the effect.xml file that should be included in the effect package. More effects can be added in the same XML file. The two languages are also examples. More (or less) languages can be used.

Code 21. effect.xml file for the distortion

```
<?xml version="1.0" encoding="UTF-8"?>
<effectpack xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://www.recordingblogs.com/sa/rbdocs/xml
orinjeffect.xsd">
   <effect>
      <effectclass>mycompany.com.Distortion</effectclass>
      <dialogclass>mycompany.com.DistortionPanel</dialogclass>
      <title>My Distortion</title>
      <version>3.0</version>
      <type>dynamics</type>
      <translations>
         <translation>
            <codeISO639-2>eng</codeISO639-2>
            <translatedtitle>My Distortion</translatedtitle>
         </translation>
         <translation>
            <codeISO639-2>bul</codeISO639-2>
            <translatedtitle>Мой дисторшън</translatedtitle>
         </translation>
      </translations>
   </effect>
</effectpack>
```

4.9. Packaging

The following builds a JAR file for an effect pack that contains a single effect – the distortion discussed in this chapter.

This code assumes that the path to **`javac`** has been set and that the directory **`class`** exists.

Code 22. Packaging the distortion

```
javac "Delay\src\mycompany\com\Distortion.java" -d "class" -classpath
   "class;Orange\oreffect.jar"
javac "Delay\src\mycompany\com\DistortionPanel.java" -d "class" -classpath
   "class;Orange\oreffect.jar"
copy src\effect.xml class\effect.xml
jar cf "exampledistortion.jar" -C "class" .
```

The resulting JAR file should be placed in the orange/effects folder of the installation for Orinj. Orinj will recognize it and use it automatically upon startup.

4.10. Obfuscating

If the effect code is obfuscated, the two classes – in this case **`Distortion`** and **`DistortionPanel`** – should be preserved (that is, not renamed). In **`Distortion`**, the obfuscation should also preserve the constructor **`Distortion`** and the function **`setLanguage`**. In **`DistortionPanel`**, the obfuscation should preserve the constructor **`DistortionPanel`**. The constructors are necessary so that Orinj can instantiate objects of these two classes. **`setLanguage`** is similarly necessary so that Orinj can set the language of the effect when the language in Orinj changes.