# Java Art Chapter 4. Visualization with Whorld

The aim of this chapter is to create a visualizer that can turn any executing Java program into a pleasing animated kaleidoscope. The approach I've chosen is summarized by Figure 1.

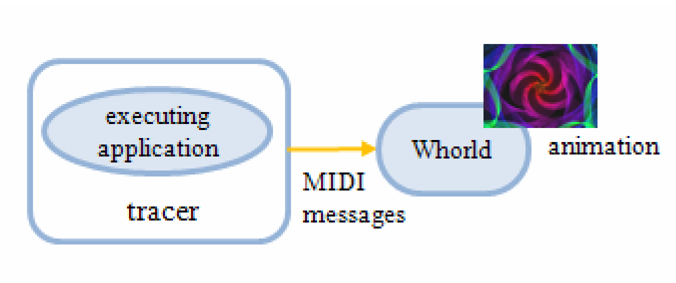


Figure 1. A Visualizer for a Java Application.

The application being visualized is monitored by a tracer implemented using the Java Platform Debugger Architecture (JPDA), specifically its Java Debug Interface (JDI) API. The JDI can be configured to watch for execution events such as the loading/unloading of classes, object state changes, method entry/exit, code execution, and JVM state changes. The details of such a tracer were explained in the last chapter.

This chapter describes a tracer called WhorldTracer, which watches for method entries and returns, and converts the details into MIDI messages sent to the Whorld visualizer (available from http://www.churchofeuthanasia.org/whorld).

MIDI (Musical Instrument Digital Interface) is a protocol that enables electronic musical instruments (e.g. synthesizers, sound cards, drum machines) to communicate and synchronize.

Whorld converts MIDI messages into animation parameters, which generate a myriad of swirling 'psychedelic' patterns. Figure 2, taken from the Whorld website, show some of the many possibilities.

Figure 2. Some Possible Whorld Animations.

Figure 3 is a Whorld animation generated by my WhorldTrace application. Each method call in the traced application is converted into a Whorld 'ring', which gradually grows, multiplies, and spreads out from the center of the screen until it disappears off the edges of the visualization window.

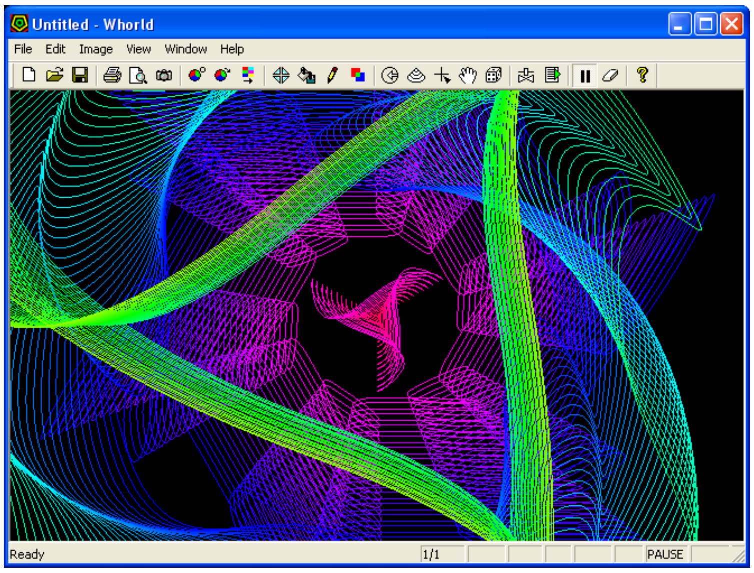


Figure 3. A WhorldTrace Animation.

I'll start by explaining Whorld's capabilities, then how WhorldTrace is connected to Whorld using the LoopBe1 internal MIDI driver (free for non-commercial use from http://www.nerds.de/en/loopbe1.html).

MIDI is only used as a message passing mechanism, and Whorld immediately maps the messages into animation parameters. As a consequence, we only need to employ (and understand) a tiny part of the extensive MIDI features offered by the Java Sound API.

## 1. Introduction to Whorld

A Whorld visualization is based on animated, growing rings. A ring is a closed polyline (i.e. a shape composed of lines, curves, or arcs, which have contiguous endpoints). A ring is 'born' in the centre of the Whorld visualization window, and gradually 'grows' outwards until it disappears off the edges of the window. Ring parameters specify a ring's growth behavior: how it rotates, changes color, and shape.

As a ring grows outwards, new copies of the ring appear in the center. Over time, the visualization window will be filled by multiple copies of the ring at different stages of growth.

For example, the visualization in Figure 4 began from a single pentagonal ring whose growth involves it slowly rotating counter-clockwise.

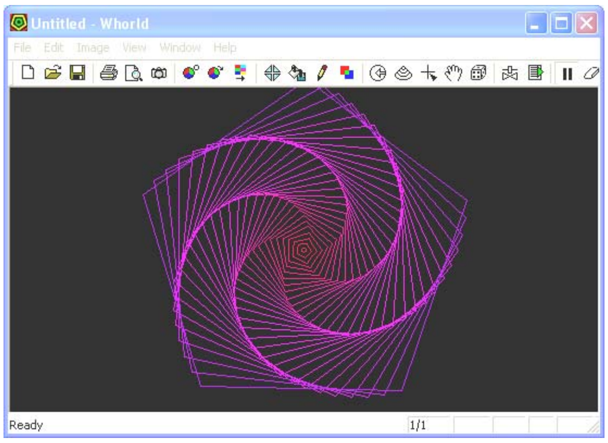
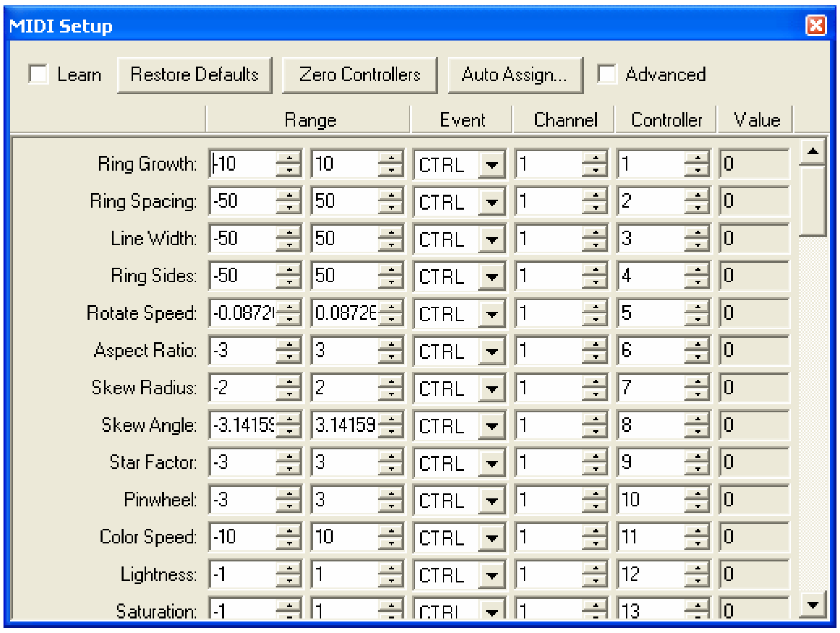


Figure 4. A Simple Series of Whorld Rings.

Parameters can be adjusted manually via a parameters dialog, or by MIDI messages (my approach), or with Whorld *oscillators*. An oscillator generates a repeating waveform which is used to change a parameter's value.

The Whorld MIDI setup dialog (see Figure 5) specifies the mapping between MIDI messages and Whorld's parameters.


MIDI Setup

☐ Learn | Restore Defaults | Zero Controllers | Auto Assign... | ☐ Advanced

| | Range | | Event | Channel | Controller | Value |
|---|---|---|---|---|---|---|
| Ring Growth: | -10 | 10 | CTRL | 1 | 1 | 0 |
| Ring Spacing: | -50 | 50 | CTRL | 1 | 2 | 0 |
| Line Width: | -50 | 50 | CTRL | 1 | 3 | 0 |
| Ring Sides: | -50 | 50 | CTRL | 1 | 4 | 0 |
| Rotate Speed: | -0.0872 | 0.0872 | CTRL | 1 | 5 | 0 |
| Aspect Ratio: | -3 | 3 | CTRL | 1 | 6 | 0 |
| Skew Radius: | -2 | 2 | CTRL | 1 | 7 | 0 |
| Skew Angle: | -3.1415 | 3.14159 | CTRL | 1 | 8 | 0 |
| Star Factor: | -3 | 3 | CTRL | 1 | 9 | 0 |
| Pinwheel: | -3 | 3 | CTRL | 1 | 10 | 0 |
| Color Speed: | -10 | 10 | CTRL | 1 | 11 | 0 |
| Lightness: | -1 | 1 | CTRL | 1 | 12 | 0 |
| Saturation: | -1 | 1 | CTRL | 1 | 13 | 0 |

Figure 5. The Mapping of MIDI Messages to Whorld Parameters.

Each row in the dialog displays information about the MIDI message that's assigned to a parameter, including the message's type (event), channel, and controller, and the range over which the parameter can be adjusted.

For example, the ring growth parameter (row no. 1) can be varied between -10 and 10 by sending a MIDI control message along channel 1 to controller 1. Ring spacing (row no. 2) is affected by a message sent to controller 2. I'll explain control messages, channels, and controllers in the "MIDI Overview" section below.

When a MIDI message arrives, its corresponding parameter is changed, and that change is applied to all the rings generated from then on.

A MIDI interface makes it possible to use Whorld to visualize music. Some (or all) of the music can be sent across to Whorld as MIDI messages to generate an animation.

Whorld includes a "Vee-Jay" mode for displaying an animation in optimized full-screen mode. Still frames can be captured as snapshots, or the visualization can be recorded as an AVI movie.

## 1.1. Whorld Parameters

Whorld supports over 20 basic parameters, a range of display modes, special effects, and master controls, which can be adjusted by over 70 different MIDI messages.

To keep things relatively simple, WhorldTracer only utilizes ten parameters, but that's still enough to generate a wide range of interesting animations.

Briefly, the parameters I'll be using are:

- ring sides: specify the number of sides in a ring;
- rotation speed: rotate a ring by a certain number of degrees per frame;
- star factor: 'fold out' or 'fold in' ring edges;
- pinwheel: shift 'star factor' vertices to the left or right;
- odd curve: adjust the curvature of 'star factor' vertices (called *odd vertices*);
- even curve: adjust the curvature of 'star factor' insets (called *even vertices*);
- odd shear: make the curvature of odd vertices asymmetrical;
- even shear: make the curvature of even vertices asymmetrical;
- skew radius: offset each ring's origin;
- skew angle: change the skew direction.

Most of these parameters are explained in more detail below.

### Rotation Speed

The rotation speed is the amount each ring rotates per frame, in degrees. A positive value causes counter-clockwise movement, negative means clockwise.

### Star Factor

The star factor parameter causes the sides of a ring to fold outwards, or inwards, forming star shapes as illustrated in Figure 6.

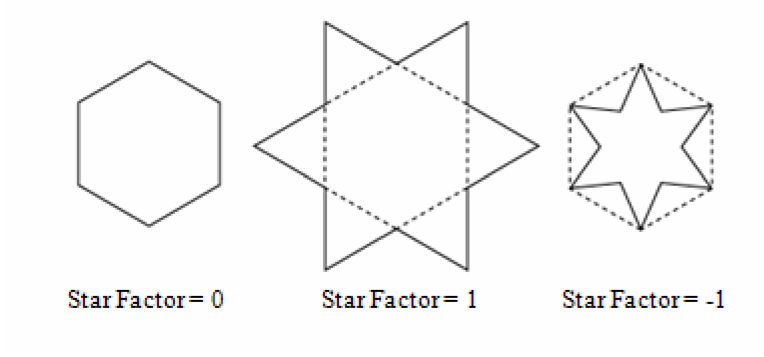


Figure 6. Three Star Factors.

A zero star factor has no effect on a ring. A star factor of 1 changes a hexagon into a Star of David, by making its sides fold outwards to form triangles (see Figure 6). A star factor of -1 makes the edges fold inwards.

A star factor of 1 applied to a square makes it becomes a diamond, while a triangle becomes a hexagon. A star factor of −1 makes a triangle become the Mercedes car symbol.

The star factor transforms a ring into a series of alternating even and odd vertices. The even vertices are the corners of the original ring. The odd vertices are the points of the star when the star factor is positive (see Figure 7), or the inner points when the star factor is negative.

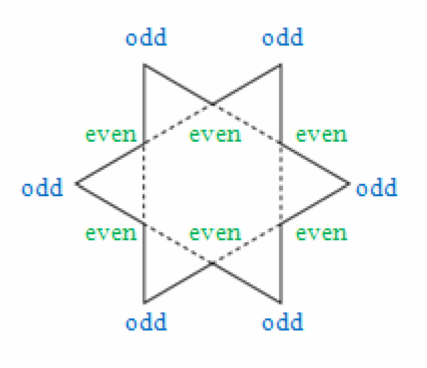


Figure 7. The Even and Odd Vertices of a Hexagon Star Ring.

### Pinwheel

The pinwheel parameter is typically used in combination with a star factor since it offsets the odd vertices of a ring, causing the shape to resemble a pinwheel or turbine, as illustrated by Figure 8.

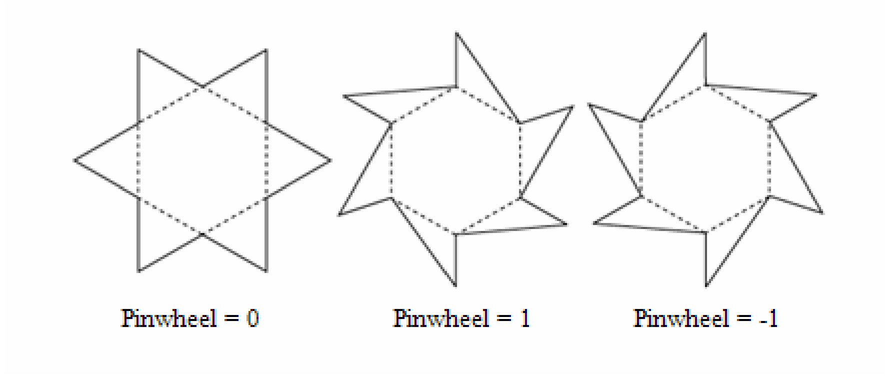


Figure 8. Pinwheel Applied to a Ring

A pinwheel of 0 has no effect. A pinwheel of 1 shifts the odd vertices counter-clockwise, while -1 shifts them clockwise. Larger values can make the ring's vertices overlap in interesting ways.

## Odd and Even Curves

The odd and even curve parameters apply curvature to the odd and even vertices of a ring. Positive values produce simple curves, while negative values produce loops or curls. Figure 9 shows the effects of positive and negative *odd* curvature applied to a hexagon star ring – only the odd vertices are affected.

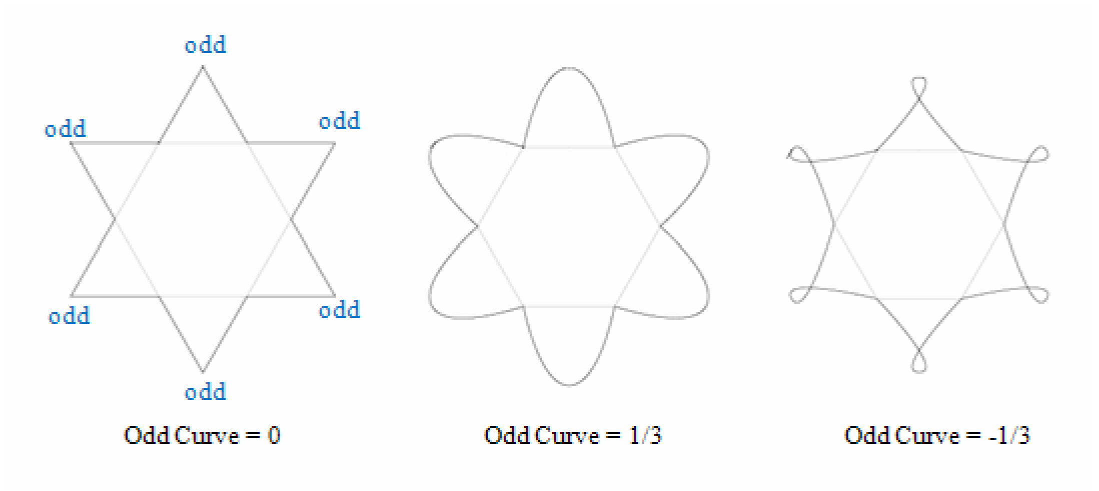


Figure 9. Odd Curvature Applied to a Ring.

## Odd and Even Shear

Odd and even shear makes odd and even curvatures asymmetrical.

The curvature at a vertex is determined by two invisible control points, which are normally equidistant from the vertex (the A and B points in the left hand ring of Figure 10). When odd or even shear is applied, the two distances will no longer be equal.

For odd shear, when its value is -1 (the middle diagram of Figure 10), the counter-clockwise point (A) coincides with the vertex, eliminating curvature on that side. As the shear value becomes more negative, the curve becomes more twisted in the counter-clockwise direction (see the right hand ring in Figure 10).

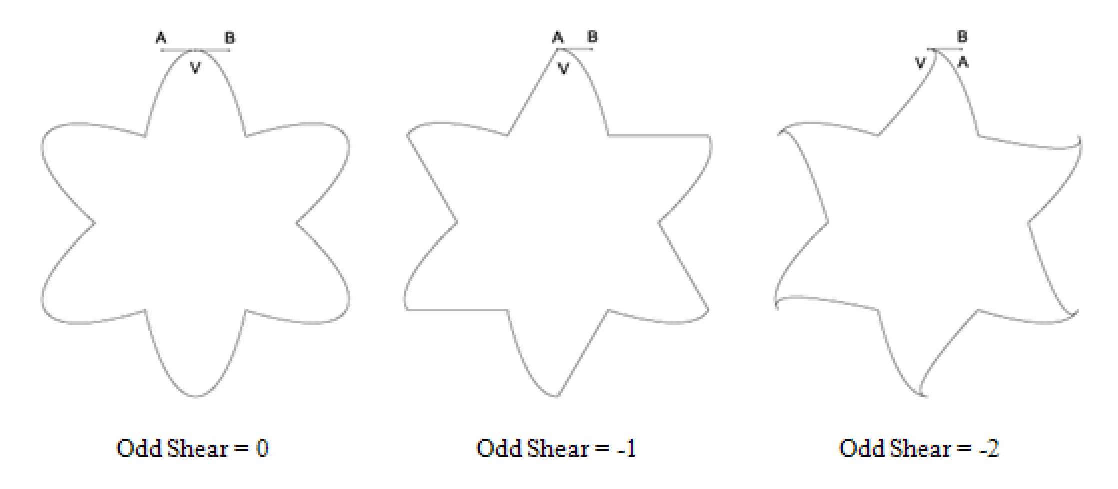


Figure 10. Odd Shear Applied to a Ring.

A positive odd shear as the opposite effect, making odd vertices twist in a clockwise direction.

Even shear as a similar effect, but on even vertices only.

## Skew Radius and Angle

Skew radius is the amount that each ring's origin shifts from the center as it grows. The direction of the shift is determined by the skew angle. At zero degrees, the skew direction is due south; at 90 degrees, it's due east, as shown in Figure 11.

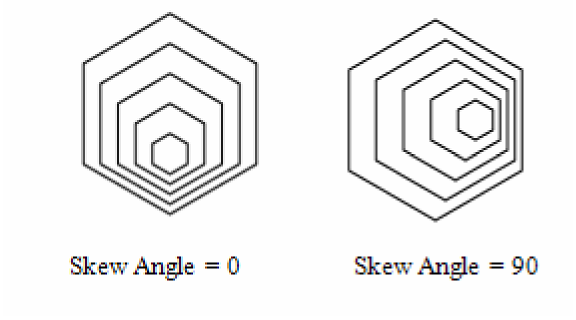


Figure 11. Different Skew Angles.

In small amounts, skew makes the visualization appear to have depth, or resemble a tunnel.

## 2. MIDI Overview

It's not necessary to understand all the intricacies of MIDI, since WhorldTracer only uses it for sending messages to Whorld, and Whorld immediately maps those messages into ring parameters. However, the message passing mechanism utilizes MIDI channels and controllers, which need some explanation.

A MIDI musical sequence stores 'instructions' for playing the music rather than the music itself, and  is converted into audio output using a sequencer and synthesizer. Their configuration is shown in greatly simplified form in Figure 12.

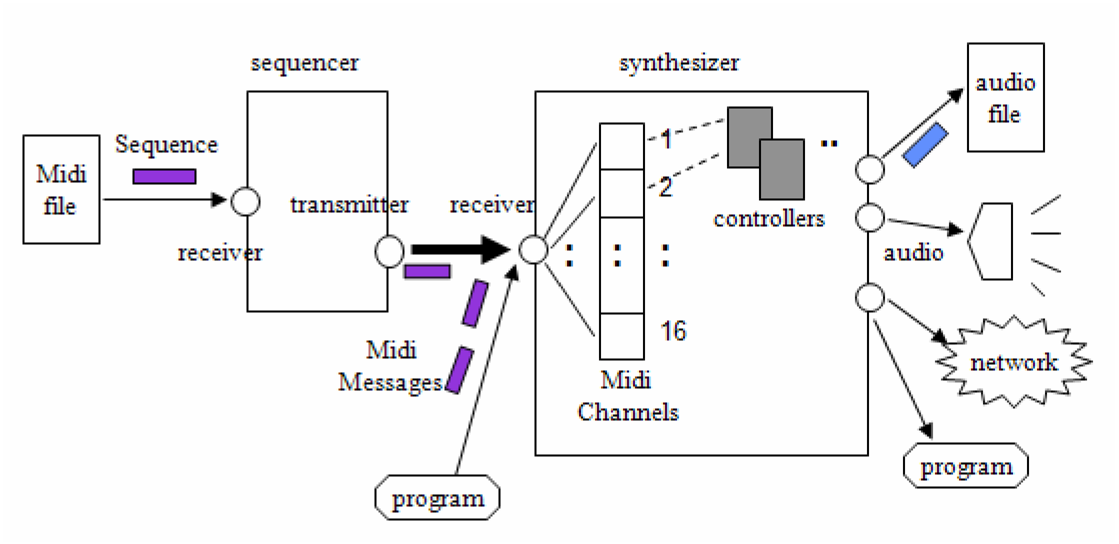


Figure 12. A MIDI Sequencer and Synthesizer.

A sequencer can be thought of as an orchestral conductor, while a synthesizer is the orchestra, made up of 16 *channels* (the musicians) playing different parts of the score. Each channel has a set of associated *controllers*, whose composition depends on the particular synthesizer, but generally includes controls for volume level, stereo balancing, panning, and so on. Each controller is identified by a unique ID, between 0 and 127.

It’s possible for a Java program to directly communicate with a synthesizer, sending it a stream of messages (as MidiMessage objects). Each message is routed to a channel based on its channel setting.

MIDI messages are encoded using three subclasses of MidiMessage: ShortMessage, SysexMessage, and MetaMessage. ShortMessage is the most important, including NOTE_ON and NOTE_OFF messages for starting and terminating note playing, and the CONTROL_CHANGE message for affecting specific controllers.

The MidiSystem class provides access to the MIDI resources installed on a machine, such as synthesizers and sequencers. A synthesizer is sent MIDI messages via a receiver port, which is obtained like so:

```
Synthesizer synthesizer = MidiSystem.getSynthesizer();
synthesizer.open();
Receiver receiver = synthesizer.getReceiver();
```

The ShortMessage class allows a CONTROL_CHANGE message to be directed towards a specific channel and controller via the receiver:

```
// Set controller no. 7 on channel 4 to 127
ShortMessage volMsg = new ShortMessage();
volMsg.setMessage(ShortMessage.CONTROL_CHANGE, 3, 7, 127);
receiver.send(volMsg, -1);
```

The second argument of ShortMessage.setMessage() is the channel ID (an index between 0 and 15, not 1 and 16, which explains the use of 3 to denote channel 4). The third argument is the channel controller ID, and the fourth is the message. The message value must fit into a single byte, so should be between 0 and 127.

A MIDI message has to be sent with a time-stamp: -1 means that the message should be processed immediately.

## 3. Linking WhorldTracer and Whorld

The previous section shows that it's relatively simple to communicate with a MIDI synthesizer, but WhorldTrace isn't using a synthesizer. As Figure 1 illustrates, WhorldTrace needs to send MIDI messages to Whorld, which is a Windows application. The solution requires a slight modification to Figure 1, adding the LoopBe1virtual MIDI Driver to the picture (see Figure 13).

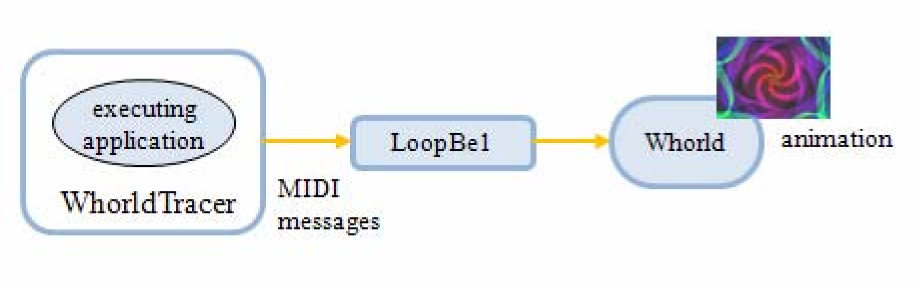
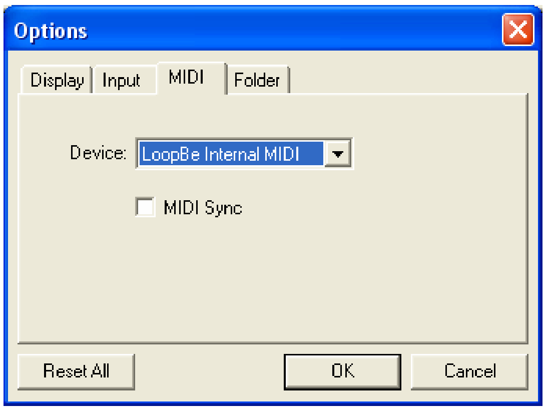


Figure 13. Linking WhorldTrace and Whorld with LoopBe1.

LoopBe1 is an internal MIDI device for transferring MIDI data between applications, which in this case are WhorldTrace and Whorld. LoopBe1 is free for non-commercial use, and available from http://www.nerds.de/en/loopbe1.html.

After LoopBe1 has been installed, Whorld can be configured to accept LoopBe1's input via a MIDI options dialog, as shown in Figure 14.



Device: LoopBe Internal MIDI

MIDI Sync

Figure 14. Configuring Whorld to use LoopBe1.

Java code requires a search function to find LoopBe1 amongst the other MIDI devices installed on a machine. Deciding on how to search for a particular device can be tricky, so it helps to first write a standalone application which lists all the MIDI devices, as shown below:

```
public class MIDIInfo
{
  public static void main(String[] args)
  {
    try {
      MidiDevice.Info[] devs = MidiSystem.getMidiDeviceInfo();
      for(int i=0; i < devs.length; i++){
        System.out.print("MIDI device "+i+": ");
        MidiDevice dev = MidiSystem.getMidiDevice(devs[i]);

        boolean bAllowsInput = (dev.getMaxTransmitters() != 0);
        boolean bAllowsOutput = (dev.getMaxReceivers() != 0);
        System.out.print( (bAllowsInput?"IN ":"   ") +
                          (bAllowsOutput?"OUT ":"    "));
        if(MidiSystem.getMidiDevice(devs[i]) instanceof Receiver)
          System.out.print("receiver\t");
        if(MidiSystem.getMidiDevice(devs[i]) instanceof Transmitter)
          System.out.print("transmitter\t");
        if(MidiSystem.getMidiDevice(devs[i]) instanceof Synthesizer)
          System.out.print("synthesizer\t");
        if(MidiSystem.getMidiDevice(devs[i]) instanceof Sequencer)
          System.out.print("sequencer");
        System.out.println();
        System.out.println("\t"+ devs[i].getName()+" ("+
                                 devs[i].getDescription()+")\t");
      }
    }
    catch (MidiUnavailableException e){
      System.out.println("No devices available");
      System.exit(0);
    }
  }  // end of main()
} // end of MIDIInfo class
```

The program calls MidiSystem.getMidiDeviceInfo() to return an array of all the devices, and then loops through them printing out their details via a call to

MidiSystem.getMidiDevice(). On one of my test machines (which has LoopBe1 installed), the following output is produced:

```
> java MIDIInfo
MIDI device 0: IN
   LoopBe Internal MIDI (No details available)
MIDI device 1:    OUT
   Microsoft MIDI Mapper (Windows MIDI_MAPPER)
MIDI device 2:    OUT
   Microsoft GS Wavetable SW Synth (Internal software synthesizer)
MIDI device 3:    OUT
   LoopBe Internal MIDI (External MIDI Port)
MIDI device 4: IN OUT sequencer
   Real Time Sequencer (Software sequencer)
MIDI device 5:    OUT synthesizer
   Java Sound Synthesizer (Software wavetable synthesizer & receiver)
```

LoopBe1 uses the name "LoopBe Internal MIDI", and appears twice in the devices list: once as a receiver (device 0) and once as a transmitter (device 3). Whorld is using LoopBe1's transmitter to receiver MIDI messages. My Java code must send messages to LoopBe1's receiver.

## 3.1. Exploring Whorld's MIDI Interface

Although Whorld's documentation is generally very good, it contains few details on how the data inside a MIDI message is mapped to a parameter. However by studying Whorld's MIDI setup dialog (shown in Figure 5), it's clear that ring spacing is affected by a message sent to controller 2 of channel 1. This can be coded as a Java CONTROL_CHANGE message :

```
// Set controller 2 on channel 1 to value
ShortMessage volMsg = new ShortMessage();
volMsg.setMessage(ShortMessage.CONTROL_CHANGE, 0, 2, value);
receiver.send(volMsg, -1);
```

Note that the channel ID is 0 which denotes channel 1.

There's also the issue of how the message's value (an integer between 0 and 127) is converted into a ring spacing value, which according to the dialog in Figure 5 can range between -50 and 50. This question also applies to the other parameters that I want WhorldTrace to manipulate.

For this reason, I developed a small application called SendParam.java to explore the Whorld MIDI interface without also having to contend with the complexities of tracing.

SendParam.java uses LoopBe1 to send a single user-supplied parameter value to Whorld as a MIDI message, and then terminates. The configuration is shown in Figure 15.

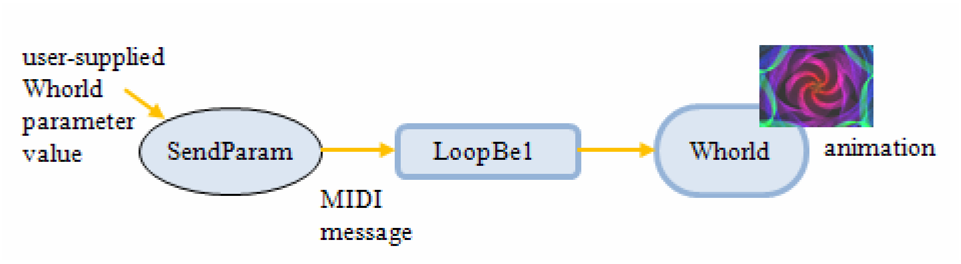


Figure 15. Linking SendParam and Whorld with LoopBe1.

Before SendParam can send a MIDI message it must first find the LoopBe1 MIDI receiver. It uses a similar approach to MIDIInfo.java shown above – it employs MidiSystem.getMidiDeviceInfo() to return an array of all the known devices, and examines each by calling MidiSystem.getMidiDevice(). It considers two things: the name of the device, and whether it's a receiver. This second condition is required to exclude the LoopBe1 transmitter.

The search code is contained in the findReceiver() method:

```
private static Receiver findReceiver(String name)
// find (and open) a MIDI receiver whose name starts with name
{
  System.out.println("Searching for device: " + name);
  try {
    MidiDevice.Info[] devices = MidiSystem.getMidiDeviceInfo();
    for(MidiDevice.Info devInfo : devices)
      if (devInfo.getName().startsWith(name)) {
        MidiDevice dev = MidiSystem.getMidiDevice(devInfo);
        if (dev.getMaxReceivers() != 0) {   // must be a receiver
          System.out.println("Found: " + devInfo.getDescription());
          dev.open();
          System.out.println("Opening receiver");
          return dev.getReceiver();
        }
      }
    System.out.println("Device not found");
    System.exit(0);
  }
  catch (MidiUnavailableException e){
    System.out.println(e);
    System.exit(0);
  }
  return null;
}  // end of findReceiver()
```

findReceiver() is called from main() in SendParam.java, and looks for a receiver whose name begins with "LoopBe":

```
// in main()
Receiver receiver = findReceiver("LoopBe");
sendMessage(receiver, controllerID, value);
receiver.close();
```

The receiver is passed to sendMessage() along with a controller ID and parameter value:

```
// global
// channel 1 chosen to receive messages
private static final int CHANNEL = 0;


private static void sendMessage(Receiver receiver,
                                   int controllerID, int value)
// send a CONTROL_CHANGE message to controller, setting it to value
{
  System.out.println("Sending " + value +
                          " to controller " + controllerID);
  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.CONTROL_CHANGE,
                                  CHANNEL, controllerID, value);
    receiver.send(message, -1);
  }
  catch (InvalidMidiDataException e) {
    System.out.println(e);
  }
} // end of sendMessage()
```

### 3.2. Using SendParam

SendParam.java is called from the command line and expects a Whorld parameter name and a value between 0 and 127. For example:

```
> java SendParam rotateSpeed 77
Searching for device: LoopBe
Found: External MIDI Port
Opening receiver
Sending 77 to controller 5
```

The parameter name ("rotateSpeed") is mapped to a controller ID by matching its position in an array of names with the corresponding position in an array of IDs:

```
// parameter names used in Whorld
private static final String[] paramNames =
  { "ringSides", "rotateSpeed", "starFactor", "pinWheel",
    "oddCurve", "evenCurve", "oddShear", "evenShear",
    "skewRadius", "skewAngle" };

// controller IDs used to modify the Whorld parameters
private static final int[] controllerIDs =
   { 4, 5, 9, 10, 18, 17, 20, 19, 7, 8 };
```

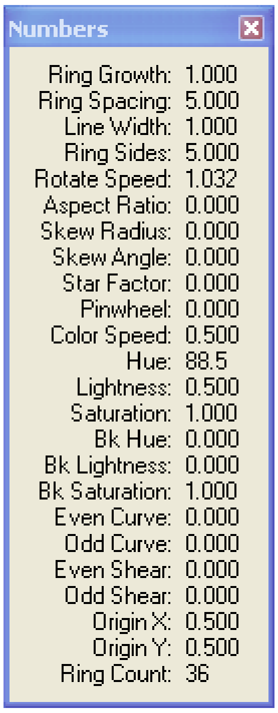
Since "rotateSpeed" is in index position 1 in paramNames, it corresponds to controller ID 5 in the controllerIDs array.

The controller IDs in the array were determined by looking at Whorld's MIDI Setup dialog (shown in Figure 5).

## 3.3. What Value Should be Sent?

In the example above, I sent 77 as the rotation speed value. How did I arrive at this 'magical' value, and what does it do in Whorld? The first clue is the MIDI Setup dialog which indicates that the rotation speed can range between -0.087 and 0.087. If this is mapped onto the byte range 0 to 127, then 64 is roughly 0, greater than 64 is positive which is a counter-clockwise rotation, and less than 64 is negative which makes a ring rotate clockwise.

In that case, 77 will induce a counter-clockwise rotation, but by how much? The easiest way of determining this is to look at Whorld's Numbers dialog (shown in Figure 16), which shows the current values for the basic parameters.



|  |  |
|---|---|
| Ring Growth: | 1.000 |
| Ring Spacing: | 5.000 |
| Line Width: | 1.000 |
| Ring Sides: | 5.000 |
| Rotate Speed: | 1.032 |
| Aspect Ratio: | 0.000 |
| Skew Radius: | 0.000 |
| Skew Angle: | 0.000 |
| Star Factor: | 0.000 |
| Pinwheel: | 0.000 |
| Color Speed: | 0.500 |
| Hue: | 88.5 |
| Lightness: | 0.500 |
| Saturation: | 1.000 |
| Bk Hue: | 0.000 |
| Bk Lightness: | 0.000 |
| Bk Saturation: | 1.000 |
| Even Curve: | 0.000 |
| Odd Curve: | 0.000 |
| Even Shear: | 0.000 |
| Odd Shear: | 0.000 |
| Origin X: | 0.500 |
| Origin Y: | 0.500 |
| Ring Count: | 36 |

Figure 16. Whorld's Numbers Dialog.

The rotation speed is shown in the fifth row: 1.032.

This indicates that 77 produces a counter-clockwise rotation of about 1 degree per frame. Since 77 is 13 more than 64, it's reasonable to suppose that sending a parameter value of 51 (64-13) will produce a clockwise rotation of about 1 degree:

```
> java SendParam rotateSpeed 51
Searching for device: LoopBe
Found: External MIDI Port
Opening receiver
Sending 51 to controller 5
```

A quick look at the Numbers dialog after executing this call confirms my guess: the rotation speed is now -1.016.

In general, it's also a good idea to look at the Whorld visualization window to see if the parameter change looks 'good'.

I used this approach (a combination of a bit of maths, guesswork, and testing) to determine suitable values for all the parameters used by WhorldTrace.

## 4. Tracing with Whorld

It's time to implement WorldTrace, which communicates with Whorld via the LoopBe1 MIDI Driver, as shown in Figure 17.

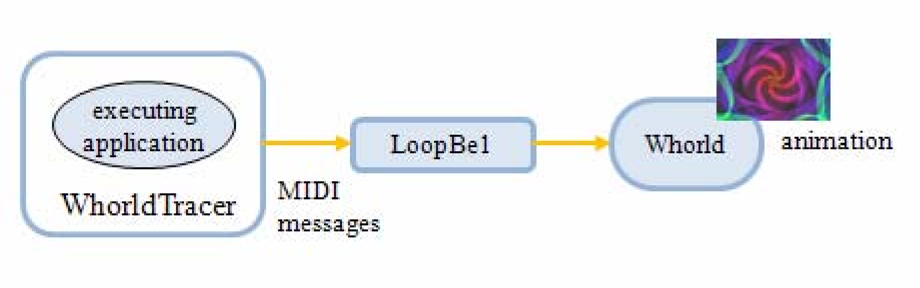


Figure 17. WhorldTracer and Whorld

My preliminary work makes the coding of WhorldTrace relatively straightforward – the tracing component is mostly a version of SimpleTrace described in the previous chapter, together with some MIDI-related methods, closely based on those used in SendParam.java.

The tracing part of WhorldTrace is actually simpler than SimpleTrace since it only monitors method entries and exits. However, the conversion of that information into suitable Whorld parameters requires some additional effort. This is reflected in the class diagrams for WhorldTrace shown in Figure 18 (only public methods are shown).

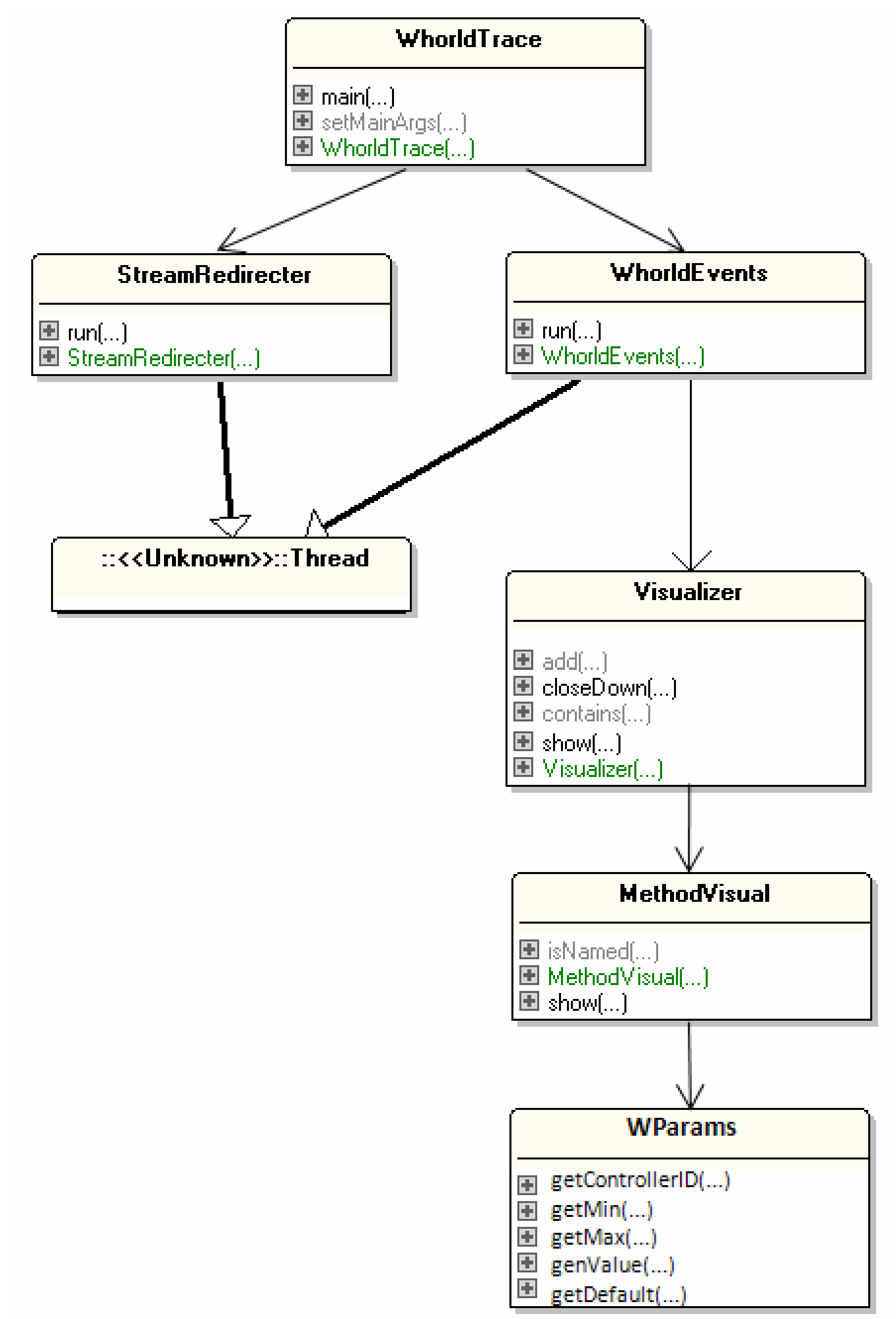


Figure 18. Class Diagrams for WhorldTrace.

The three classes at the top of Figure 18 (WhorldTrace, StreamRedirecter, and WhorldEvents) contain the tracer code.

WhorldTrace.java is little more than a renamed SimpleTrace.java. It sets up the command-line launching connection which starts the JVM and creates a local link with the JVM on the same machine. It passes the application's name and input arguments over to the JVM, and employs the StreamRedirecter class to redirect the JVM's output and error streams to stdout and stderr. StreamRedirecter is unchanged from the previous chapter.

WhorldEvents only monitors method entry/exit events, so is essentially a subset of JDIEventMonitor from the last chapter. I'll only describe the parts of WhorldEvents which differ significantly from JDIEventMonitor.

The Visualizer, MethodVisual, and WParams classes are concerned with MIDI processing and Whorld parameter creation and maintenance.

Visualizer sets up the MIDI receiver link to LoopBe1, and stores MethodVisual objects in an ArrayList. Each MethodVisual instance represents a method visualization as Whorld parameters, which are sent to Whorld when the method needs to be displayed. Parameter generation is carried out by methods in the WParams enumerated type.

## 5. Tracing Method Events

The WhorldEvents class creates and enables method entry and exit event requests in setEventRequests().

```
private void setEventRequests()
{
  EventRequestManager mgr = vm.eventRequestManager();

  MethodEntryRequest menr = mgr.createMethodEntryRequest();
  for (int i = 0; i < excludes.length; ++i)   //report method entries
    menr.addClassExclusionFilter(excludes[i]);
  menr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  menr.enable();

  MethodExitRequest mexr = mgr.createMethodExitRequest();
  for (int i = 0; i < excludes.length; ++i)   // report method exits
    mexr.addClassExclusionFilter(excludes[i]);
  mexr.setSuspendPolicy(EventRequest.SUSPEND_EVENT_THREAD);
  mexr.enable();
}  // end of setEventRequests()
```

This same method appears in the JDIEventMonitor class, but also requests class and thread events.

### Handling Method Events

As in JDIEventMonitor, WhorldEvents calls handleEvent() to process an incoming event:

```
private void handleEvent(Event event)
{
  // method events
  if (event instanceof MethodEntryEvent)
    methodEntryEvent((MethodEntryEvent) event);
  else if (event instanceof MethodExitEvent)
    methodExitEvent((MethodExitEvent) event);

  // VM events
  else if (event instanceof VMStartEvent)
```

```
    vmStartEvent((VMStartEvent) event);
  else if (event instanceof VMDeathEvent)
    vmDeathEvent((VMDeathEvent) event);
  else if (event instanceof VMDisconnectEvent)
    vmDisconnectEvent((VMDisconnectEvent) event);
  else
    throw new Error("Unexpected event type");
}  // end of handleEvent()
```

handleEvent() in JDIEventMonitor also deals with classes, threads, step events, and modified field events.

The JVM-related processing in vmStartEvent(), vmDeathEvent(), and vmDisconnectEvent() is unchanged from JDIEventMonitor .

## Entering a Method

Method entry triggers the creation of a method visualization (if one doesn't already exists), and the display of that visualization by Whorld.

```
// global
private Visualizer visualizer;


private void methodEntryEvent(MethodEntryEvent event)
// entered a method but no code executed yet
{
  Method meth = event.method();
  String className = meth.declaringType().name();
  String methodName = meth.name();
  int methodSize = meth.bytecodes().length;

  System.out.println("===>> " + className +  "." + methodName +"()" +
                          " method size (bytes) = " + methodSize);

  visualizer.add(className, methodName, methodSize);
  visualizer.show(className, methodName);
}  // end of methodEntryEvent()
```

To help determine the ‘complexity’ of the visualization, the size of the method in bytes is passed to Visualizer.add(), along with the class and method names.

## Leaving a Method

methodExitEvent() is called when all the code in a method has been executed, and the application is about to return to the calling function. Whorld must stop visualizing the method that's about to return, and resume the visualization of the caller.

The tricky part of implementing this behavior is that it requires the visualizer to know the class and method names of the calling function so it's visualization can be reinstated. This information is available, but not directly. The tracer needs to look 'beneath' the current frame on the call stack, at the frame representing the calling function.

For example if method a() called b(), then the current frame would contain information about b(), and the next frame down in the stack would be about a().

Unfortunately, the stack examination has to be a little bit more complicated to distinguish between application and system methods. For instance, it’s possible that method a() calls System.out.println() with a b() argument (perhaps we want to print b()'s result). The call stack would then have b() on the top, then a println() frame, *and* then a().

To deal with this situation, the call stack is searched until a frame is found whose method has a visualization, which means that it must be an application method.

```
private void methodExitEvent(MethodExitEvent event)
{
  ThreadReference thr = event.thread();
  try {
    int numFrames = thr.frameCount();
    if (numFrames == 1)    // there's no calling frame
      return;

    StackFrame returnFrame = null;
    String className, methodName;

    // search down through the call stack, starting at frame 1
    for (int i=1; i < numFrames; i++) {
      returnFrame = thr.frame(i);   // examine a frame
      if (returnFrame != null) {
        Location loc = returnFrame.location();
        Method meth = loc.method();
        className = meth.declaringType().name();
        methodName = meth.name();

        // if the function has a visualization, show it
        if (visualizer.contains(className, methodName)) {
          System.out.println("<<=== " +
                         className + "." + methodName +"()");
          visualizer.show(className, methodName);
          return;
        }
      }
    }
  }
  catch (IncompatibleThreadStateException e)
  {  System.out.println(e); }
}  // end of methodExitEvent()
```

## 6. Tracing Example

The tracing and visualization carried out by WhorldTrace can be demonstrated by considering the following Foo class:

```
public class Foo
{
  int a = 0, b = 1, c = 2;

  public static void main(String[] args)
  { int e = 4;
```

```
    int g = 5;
    Foo f = new Foo();
    for (int i = 0; i < 2; i++)
      f.foo();
  }

  public void foo()
  { int d = 3;
    System.out.println("Foo");
    d = 4;
  }
}  // end of Foo class
```

Foo contains three methods: main(), the Foo() constructor (not defined, but still present), and Foo.foo().

When Foo is traced, the following textual output is generated by WhorldTrace:

```
> java -cp "C:\Program Files\Java\jdk1.6.0_10\lib\tools.jar;."
                                                   WhorldTrace Foo
Searching for device: LoopBe
Found: External MIDI Port
Opening receiver
Visualizer Paused
Clearing the Visualizer
-- VM Started --
===>> Foo.main() method size (bytes) = 32
  Adding Foo.main()
Visualizer Resumed
  Showing Foo.main()
===>> Foo.<init>() method size (bytes) = 20
  Adding Foo.<init>()
  Showing Foo.<init>()
<<=== Foo.main()
  Showing Foo.main()
===>> Foo.foo() method size (bytes) = 13
  Adding Foo.foo()
  Showing Foo.foo()
<<=== Foo.main()
  Showing Foo.main()
===>> Foo.foo() method size (bytes) = 13
Method Foo.foo() already exists
  Showing Foo.foo()
<<=== Foo.main()
  Showing Foo.main()
-- The application has exited --
Foo
Foo
```

It shows that main() calls Foo(), and then Foo.foo() twice. There are three calls to Visualizer.add(), denoted by the “Adding…” text lines, with the second call to Foo.foo() not generating another visualization. Whorld is instructed to change its visualization several times:

main() → Foo() → main() → Foo.foo() → main() → Foo.foo() → main()

This is the correct, intended behavior, but the visualization displayed by Whorld is too simple (see Figure 19).

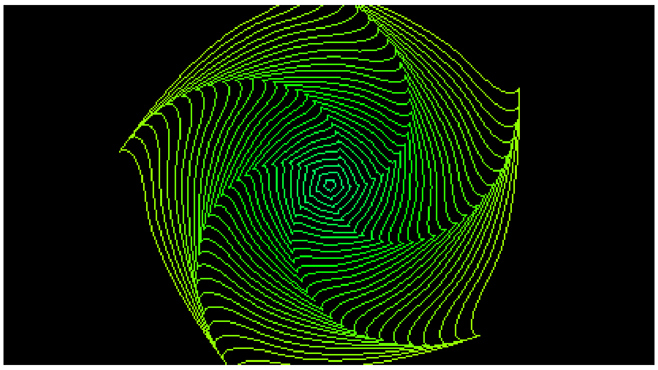


Figure 19. The Visualization of the Foo class.

The textual output of the trace indicates that there should be three different visualization rings, one each for main(), Foo(), and Foo.foo(), but usually Whorld only shows one type of ring (or sometimes two). Figure 19 only has one ring, which is growing and rotating because of its parameter settings.

Rings are missing because of timing issues – if a traced method executes and returns very quickly then Whorld doesn't have enough time to start displaying that method's ring before it's told to restore the old one. This is only a problem with short methods, of a few lines.

Another quirk of WhorldTrace is that Whorld will keep displaying the ring for main() even after the trace has finished. This behavior can easily be changed by sending a pause and/or clear message to Whorld before WhorldTrace exits. The details on how to implement these messages is explained below.

## 7. The Visualizer

The Visualizer class acts as an interface for the Whorld visualizer, which is accessed via the LoopBe 1 MIDI virtual input port, and controlled using MIDI CONTROL_CHANGE messages. The MIDI code is mostly borrowed, with minor changes, from SendParam.java described earlier in this chapter.

This class also stores an ArrayList of MethodVisual objects, one for each method visualization used by Whorld.

The Visualizer constructor uses a version of SendParam's findReceiver() method to open a link to Whorld via LoopBe1, and also initializes the ArrayList.

```
// globals
// (start of) name of MIDI device linked to Whorld
private static final String MIDI_NAME = "LoopBe";

private Receiver receiver;
private ArrayList<MethodVisual> visuals;
```

```
public Visualizer()
{
  receiver = findReceiver(MIDI_NAME);
  pause(true);
  clear();
  visuals = new ArrayList<MethodVisual>();
}
```

The pause() and clear() methods cause Whorld to pause and clear its visualization window.

```
// Whorld controller ID
private static final int PAUSE = 51;
private boolean isPaused = false;


private void pause(boolean b)
// pause/resume the Whorld Visualizer
{
  isPaused = b;
  if (isPaused)
    System.out.println("Visualizer Paused");
  else
    System.out.println("Visualizer Resumed");

  int val = (isPaused) ? 127 : 0;     // 127 == pause; 0 == resume
  sendMessage(receiver, PAUSE, val);
}  // end of pause()
```

pause() sends 127 or 0 to controller ID 51 to pause or resume ring generation. I found the ID by browsing through the MIDI Setup dialog in Whorld, which lists all the controllers and their Whorld functions (see Figure 5).

sendMessage() creates a CONTROL_CHANGE ShortMessage, and sends it to the specified controller. The code is almost identical to sendMessage() in SendParam.java.

```
// channel 1 chosen to receive messages
private static final int CHANNEL = 0;


private void sendMessage(Receiver receiver,
                              int controller, int value)
{ ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.CONTROL_CHANGE, CHANNEL,
                                         controller, value);
    receiver.send(message, -1);
  }
  catch (InvalidMidiDataException e)
  {  System.out.println(e);  }
} // end of sendMessage()
```

clear () affects Whorld by communicating with controller ID 52

```
// Whorld controller ID
```

```
private static final int CLEAR = 52;

private void clear()
// clear the Whorld visualizer window;
// must switch clear off then on for clearing to work
{
  System.out.println("Clearing the Visualizer");
  sendMessage(receiver, CLEAR, 0);    // 0 == off
  sendMessage(receiver, CLEAR, 127);    // 127 == on
}
```

### 7.1. Adding a Visualization

add() adds a method visualization to the ArrayList, labelled with its class and method name, but only if there's no visualization for that method already in the list.

```
// global
private ArrayList<MethodVisual> visuals;


public boolean add(String className, String methodName,
                                             int methodSize)
{ for(MethodVisual vis: visuals)
    if (vis.isNamed(className, methodName)) {
      System.out.println("Method " + className + "." + methodName +
                                         "() already exists");
      return false;
    }
  visuals.add( new MethodVisual(className, methodName, methodSize) );
  return true;
}  // end of add()
```

The new MethodVisual object also takes the method size as an argument, which it uses as a guide for selecting Whorld parameters.

### 7.2. Showing a Visualization

The display of a visualization involves searching the ArrayList for the specified class and method name combination, and then calling MethodVisual.show() on the retrieved MethodVisual object.

```
public void show(String className, String methodName)
{
  for(MethodVisual vis: visuals)
    if (vis.isNamed(className, methodName)) {
      if (isPaused)
        pause(false);
      vis.show(receiver);
      return;
   }
}  // end of show()
```

## 8. Visualizing a Method

The MethodVisual class generates and stores the Whorld parameters used to draw the visual for a particular method. It sends these parameters to Whorld when MethodVisual.show() is called from Visualizer.

Parameter generation uses the byte size of the method to decide how many of the parameters will have (simple) default values, but the numerical details for each parameter are hidden away in the WParams enumerated type.

The MethodVisual constructor initializes ten Whorld Parameters using either default or random generated values.

```
// globals
// byte sizes at which code 'complexity' changes
private static final int MAX_SIMPLE = 40;   // 0-39 is simple code
private static final int MAX_MEDIUM = 80;
                                 // 40-79 is medium; 80+ is complex

private String className, methodName;

private int ringSides, starFactor, pinWheel, oddCurve, evenCurve,
            oddShear, evenShear, rotateSpeed, skewAngle, skewRadius;



public MethodVisual(String classNm, String methodNm, int methodSize)
{
  className = classNm;
  methodName = methodNm;
  System.out.println("  Adding " + className + "." +
                                            methodName + "()");

  // ---- always generate these parameters ----

  ringSides = WParams.RING_SIDES.genValue();
  starFactor = WParams.STAR_FACTOR.genValue();
  pinWheel = WParams.PIN_WHEEL.genValue();
  oddCurve = WParams.ODD_CURVE.genValue();
  rotateSpeed = WParams.ROTATE_SPEED.genValue();

  // ---- medium complexity ----

  evenCurve = (methodSize >= MAX_SIMPLE)?
    WParams.EVEN_CURVE.genValue():WParams.EVEN_CURVE.getDefault();

  oddShear = (methodSize >= MAX_SIMPLE)?
    WParams.ODD_SHEAR.genValue():WParams.ODD_SHEAR.getDefault();

  evenShear = (methodSize >= MAX_SIMPLE)?
    WParams.EVEN_SHEAR.genValue():WParams.EVEN_SHEAR.getDefault();

  // ---- above medium (high) complexity ----

  skewAngle = (methodSize >= MAX_MEDIUM)?
    WParams.SKEW_ANGLE.genValue():WParams.SKEW_ANGLE.getDefault();

  skewRadius = (methodSize >= MAX_MEDIUM)?
    WParams.SKEW_RADIUS.genValue():WParams.SKEW_RADIUS.getDefault();
} // end of MethodVisual()
```

The WParams methods return integers in the range 0 to 127.

genValue() returns a random value, which means that different visuals will be created for a program each time that WhorldTrace is called. This behavior could be changed; for example, the choice of parameters could be derived from a method's code structure, which would make a program visualization the same every time.

The boundaries between simple, medium, and complicated code are demarcated with the constants MAX_SIMPLE and MAX_MEDIUM, which were chosen arbitrarily based on my tracing test programs.

## 8.1. Showing the Visualization

The MethodVisual.show() method sends each parameter over to Whorld.

```
public void show(Receiver receiver)
{
  System.out.println("  Showing " + className + "." +
                                          methodName + "()");

  sendParameter(receiver, WParams.RING_SIDES, ringSides);
  sendParameter(receiver, WParams.STAR_FACTOR, starFactor);
  sendParameter(receiver, WParams.PIN_WHEEL, pinWheel);
  sendParameter(receiver, WParams.ODD_CURVE, oddCurve);
  sendParameter(receiver, WParams.EVEN_CURVE, evenCurve);
  sendParameter(receiver, WParams.ODD_SHEAR, oddShear);
  sendParameter(receiver, WParams.EVEN_SHEAR, evenShear);
  sendParameter(receiver, WParams.ROTATE_SPEED, rotateSpeed);
  sendParameter(receiver, WParams.SKEW_ANGLE, skewAngle);
  sendParameter(receiver, WParams.SKEW_RADIUS, skewRadius);
}  // end of show()
```

sendParameter() is a variant of sendMessage() seen last in the Visualization class. It creates a CONTROL_CHANGE  ShortMessage, and sends it to the specified controller. The only difference from sendMessage() is that WParams supplies the controller ID for a given parameter.

```
private void sendParameter(Receiver receiver,
                                   WParams param, int value)
{
  int controller = param.getControllerID();
                     // get controller for this parameter
  ShortMessage message = new ShortMessage();
  try {
    message.setMessage(ShortMessage.CONTROL_CHANGE, CHANNEL,
                                            controller, value);
    receiver.send(message, -1);
  }
  catch (InvalidMidiDataException e)
  {  System.out.println(e);  }
}
```

## 9. The Parameters Enumerated Type

The main purpose of the WParams enumerated type is to hide the numbers used for generating parameter values.

Each parameter has three integers associated with it: its controller ID in Whorld, and its  minimum and maximum allowed values. Each min-max range is typically a subset of the full range (0-127), which I arrived at by using SendParam.java to see what numbers produced interesting rings.

```
public enum WParams
{
  RING_SIDES    (4,  62, 66),
  ROTATE_SPEED  (5,  51, 77),
  STAR_FACTOR   (9,  43, 85),
  PIN_WHEEL     (10, 43, 85),
  ODD_CURVE     (18, 71, 85),
  EVEN_CURVE    (17, 71, 85),
  ODD_SHEAR     (20, 85, 106),
  EVEN_SHEAR    (19, 85, 106),
  SKEW_RADIUS   (7,  53, 75),
  SKEW_ANGLE    (8,  0,  127);

  private int controllerID, min, max;


  WParams(int controllerID, int min, int max)
  {
    this.controllerID = controllerID;
    this.min = min;
    this.max = max;
  }

  // more methods...

} // end of WParam enum
```

For example, the ring rotation speed (WParams.ROTATE_SPEED) uses controller ID 5 and can vary between 51 and 77. Earlier in this chapter, I used SendParam to work out that the 51-77 range is equivalent to a rotation of at most 1 degree/frame either in the clockwise or counter-clockwise direction.

The genValue() method returns a random integer within the min-max range:

```
public int genValue()
{ return (int)(Math.random()*(max-min+1) + min);
                         // add 1 so max is possible
}
```

A better version of genValue() should probably base it's value on some measure of the method's structure, such as number and type of local variables, algorithmin complexity, or a software metric.

getDefault() returns a default value, which is always 64. For almost all of my selected parameters, this is equivalent to 0, or no change. The exception is the ring sides parameter, where 64 generates a ring with 5 sides.

```
public int getDefault()
{  return 64;  }
```

If WParams needed to use different defaults for each parameter, then they should be encoded as another enumerated type value, joining the controller ID and minimum and maximum values.