Chapter 13.7. Scratchable Java

Scratch is a hugely enjoyable visual programming language (http://scratch.mit.edu), that makes it easy for primary school kids to create interactive stories, games, and animations (and learn programming skills in the process). The IDE can be downloaded for free, and is shown in Figure 1.

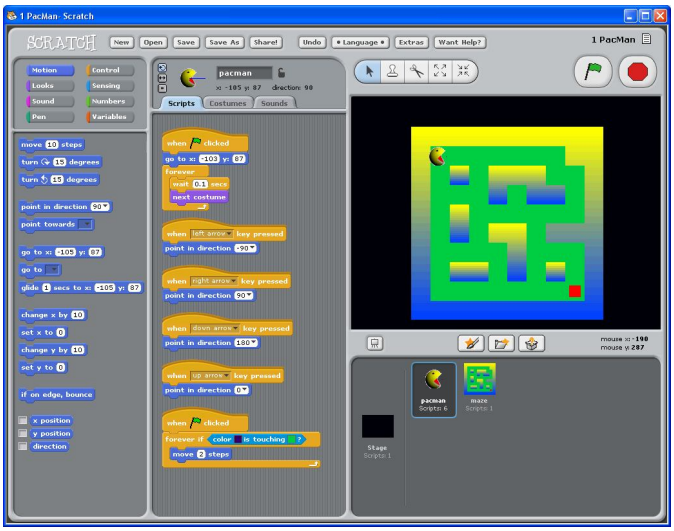

Figure 1. The Scratch IDE.

The aim of this chapter is to link Scratch and Java by utilizing an experimental feature of Scratch v1.3 which lets Scratch programs communicate with other software or hardware via a TCP socket and message passing.

I develop a small set of Java classes which hide the low-level details of socket creation and message construction and extraction, and use those classes to write several small-to-medium size examples showing how Scratch and Java can work together.

The examples include a Scratch program that utilizes Java as an external time source, and a Java GUI for a Scratch trampoline game. These larger examples show that writing a Scratch/Java application can be a little tricky because of Scratch's threaded nature, an issue that arises when interfacing any language to Scratch.

Another difficulty is the simple nature of the communication between Scratch and Java. Scratch's network protocol only supports messages involving strings and numbers, which makes it hard to implement richer forms of interaction. However, there are some coding techniques that can help matters, which are applicable to any language communicating with Scratch, not just Java.

This chapter doesn't explain how to program in Scratch, but the next section gives an overview of its features, and points to sources of more information.

1. An Overview of Scratch

Scratch programs are made up of sprites which can move about on a stage, play music, bounce off other sprites, and change their looks. A sprite is controlled by scripts built out of command blocks (blocks look a lot like jigsaw pieces) that can be snapped together in intuitive ways. The scripts act as independent threads of behavior at runtime, triggered by keyboard input, mouse movements, sprite clicking, external sensor activity, by broadcast messages, or due to changes in variables.

Command blocks are divided into eight categories: motion, looks, sounds, pen, control, sensing, numbers, and variables. The 'looks' category allows a sprite to display speech bubbles, thought bubbles, change its size, apply visual effects to itself such as adjusting its brightness, and change its costume. The 'pen' category offers the sort of capabilities familiar from Logo. 'Control' includes blocks for constructing loops and if-statements, and also event-based triggers. 'Sensing' offers blocks for monitoring the mouse, the keyboard, collision detection based on color and proximity, and the reading of sensors on a Picoboard (an external hardware board).

Variables are untyped, and treated as strings or numbers (similar to Java doubles). Variables may be local to a sprite, or be global to the application. The only data structure is the list.

Scratch was developed by the Lifelong Kindergarten Group at the MIT Media Lab, and so there's lots of support for teachers and children at the Scratch website (http://scratch.mit.edu). There are several forums (http://scratch.mit.edu/forums/), and a thriving project collection where people can uploading their work. Aside from the MIT website, other useful Scratch resources are http://resources.scratchr.org/pages/, http://cegsa.editme.com/ScratchSqueakEtoysResources, and the Scratch projects at Nebo Elementary School (http://nebomusic.net/scratch.html).

Scratch isn't a full-blown programming language for building air traffic control system and nuclear power plant software. It's a first language, deliberately lacking many features, such as arrays, procedures and functions, recursion, parameter passing and return values for scripts, file IO, classes, and inheritance.

Behind the scenes, Scratch is implemented using Squeak (http://www.squeak.org/), a variant of Smalltalk-80, which makes it fairly easy to extend the system. One interesting off-shoot is BYOB by Jens Mönig (http://www.chirp.scratchr.org/blog/?p=19) which addresses many of Scratch's shortcomings. BYOB supports the definition of new command blocks for procedures and functions, parameter passing to blocks, and recursion.

1.1. Remote Sensor Connections

Scratch v.1.3. added an experimental feature called *remote sensor connections* based on sending and receiving messages via a TCP socket. This makes it possible for Scratch to communicate with other languages, software, or hardware with network capabilities. The socket mechanism is hidden at the programmer level – instead scripts broadcast messages and utilize externally visible global variables.

The details were first explained in a forum post at http://scratch.mit.edu/forums/viewtopic.php?id=9458, which includes code snippets in Python and Processing. A first draft of the networking protocol was also posted to that forum thread. Subsequently, the Scratch Connections website

(http://scratchconnections.wik.is/) was set up as a central location for information on using remote sensors. Its projects page is particularly interesting, including examples that employ Twitter, the Ardunino hobbyist board (http://www.arduino.cc/), speech recognition and synthesis, and webcams. Unfortunately, from a Java programmer's point of view, the site is a bit disappointing, since most of the applications use Python, with a few written in Processing.

1.2. Messages Transmitted by Scratch

A script can send data out of Scratch using two different mechanisms. A broadcast command block can send a string to every script in the application, and also out through the remote sensors socket as a "broadcast" message. For instance, if a script broadcasts "hi", then the message:

{0, 0, 0, 14} broadcast "hi"

is transmitted through the socket. The message is preceded by four bytes which holds the length of the message (the "{", "}", and ","s are not part of the message).

The other way to output information is to change a global variable. This is converted into a "sensor-update" message. For example, if the "hits" global is changed to 12, then the following message would be transmitted through the socket:

{0, 0, 0, 31} sensor-update "Scratch-hits" 12

The length of the message is specified up-front as 4 bytes. The global variable name is prefixed by "Scratch-" and the new value is included at the end of the message.

No messages are issued when local variables inside a sprite are changed.

The network protocol allows a "sensor-update" message to contain more than one name/value pair, such as:

{0, 0, 0, 49} sensor-update "Scratch-hits" 12 "Scratch-foo" 0.1

However, my Scratch IO classes only support "sensor-update" messages with a single name/value pair.

1.3. Messages Sent to Scratch

"Broadcast" messages are sent to Scratch using the same message syntax as above. For example, "hi" can be delivered to Scratch through its socket as the message:

{0, 0, 0, 14} broadcast "hi"

The "hi" string will be broadcast to every script, and those scripts waiting for such a message will wake up and execute.

A "sensor-update" message sent into Scratch is handled somewhat differently. When a "sensor-update" message arrives, the same-named *virtual sensor* is updated. If no such virtual sensor currently exists, then one is created. For instance, if the following message is passed to Scratch:

{0, 0, 0, 30} sensor-update "Scratch-foo" 10

Then Scratch looks for the virtual sensor called "foo" and sets its value to 10. If no "foo" sensor exists then one is created.

A virtual sensor can be understood by comparing it to the seven 'real' sensors available in Scratch (called slider, light, sound, resistance-A, resistance-B, resistance-C, and resistance-D), which represent the seven sensors on a PicoBoard. When a PicoBoard is connected to Scratch, these sensor variables will store the hardware sensors' current values.

2. The Java ScratchIO Classes

I've written three Java classes for supporting Java/Scratch socket communication: ScratchIO, ScratchMessage, and StringToks. Their public methods are shown in the class diagrams in Figure 2.

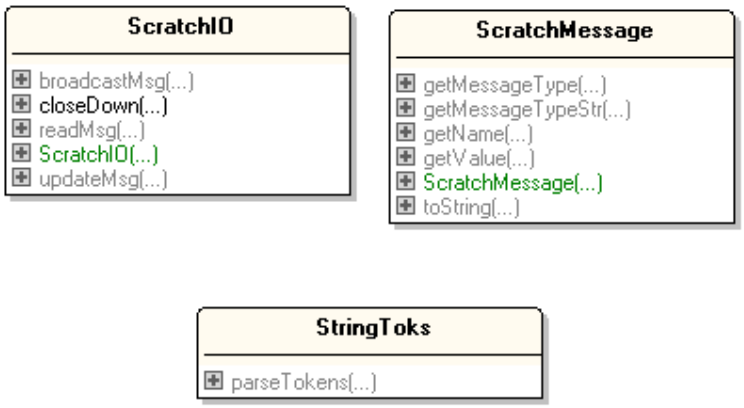

Figure 2. The Java/Scratch Classes.

ScratchIO attempts to connect to the TCP socket utilized by Scratch, and supports the sending of "broadcast" and "sensor-update" messages with its broadcastMsg() and updateMsg() methods. readMsg() is a blocking read which returns an incoming Scratch message as a ScratchMessage object. closeDown() closes the IO link with Scratch.

A ScratchMessage object can represent a "broadcast" message (in which case getMessageType() returns BROADCAST_MSG) or a "sensor-update" message getMessageType() returns SENSOR_UPDATE_MSG) or something unknown (getMessageType() returns UNKNOWN_MSG). If the object is a "sensor-update" message then getValue() returns its value as a string.

StringToks.parseTokens() is a variant of Java's built-in StringTokenizer which treats text inside double quotes as a single token, even if that text contains white space. It's utilized when parsing "broadcast" messages, which may be enclosed in quotes.

2.1. Linking to Scratch

The ScratchIO constructor tries to connect to a TCP socket at port 42001, which should already have been created by Scratch. Input and output streams are layered on top of the socket to support two-way communication between Java and Scratch.


```
// globals
private static final int SCRATCH_PORT = 42001;

private Socket scratchSocket;
private InputStream in = null;
private OutputStream out = null;

public ScratchIO()
{
  try {
    scratchSocket = new Socket("localhost", SCRATCH_PORT);
    in = scratchSocket.getInputStream();
    out = scratchSocket.getOutputStream();
  }
  catch (UnknownHostException e) {
    System.err.println("Scratch port (" + SCRATCH_PORT +
                                          ") not found");
    System.exit(1);
  }
  catch (IOException e) {
    System.err.println("Scratch IO link could not be created");
    System.exit(1);
  }
}  // end of ScratchIO()
```

The most common problem is to get a "Scratch IO link could not be created" error message, which indicates that Scratch hasn't created the socket. The socket must be initialized each time Scratch is started, by right clicking on the sensor command block in the 'sensing' blocks collection and selecting the "enable remote sensor connections" menu item (as shown in Figure 3).

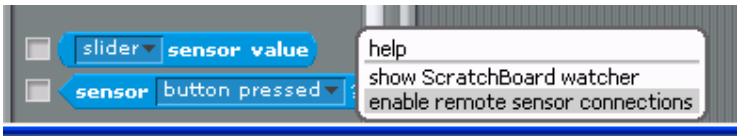

Figure 3. Enabling Remote Sensor Connections.

Another possible problem is that the 42001 port may be blocked by a firewall. The firewall settings need to be modified to allow Scratch and the JVM to use that port.

2.2. Receiving a Message

ScratchIO.readMsg() waits for a message to arrive on its input stream, and returns the text as a ScratchMessage object. The method has three distinct parts: first it extracts the message body length from the first four bytes of the message, then it reads the message body as text, and finally the text is converted into a ScratchMessage object.

```
public ScratchMessage readMsg()
{
  if (in == null) {
    System.err.println("Input stream error");
    return null;
```



```
  }

  ScratchMessage scratchMsg = null;
  int msgSize = readMsgSize();   // get message body length
  if (msgSize > 0) {
    try {
      byte[] buf = new byte[msgSize];
      in.read(buf, 0, msgSize);
      String msg = new String(buf);   // read the body as text
      scratchMsg = new ScratchMessage(msg);   // make an object
    }
    catch (IOException e) {
      System.err.println("Message read error: " + e);
      System.exit(0);
    }
  }
  return scratchMsg;
}  // end of readMsg()
```

readMsg() converts each message into a single object. In particular, it assumes that an incoming "sensor-update" message only contains a single name/value pair, although the Scratch networking protocol allows multiple name/value pairs.

readMsgSize() reads four bytes into an array, then converts it into an integer.

```
// globals
private static final int NUM_BYTES_SIZE = 4;

private int readMsgSize()
{
  int msgSize = -1;
  try {
    byte[] buf = new byte[NUM_BYTES_SIZE];
    in.read(buf, 0, NUM_BYTES_SIZE);
    msgSize = byteArrayToInt(buf);
  }
  catch (IOException e) {
    System.err.println("Header read error: " + e);
    System.exit(0);
  }
  return msgSize;
  }  // end of readMsgSize()
```


The array's bytes are shifted into the four bytes of a 32-bit integer, as shown in Figure 4.

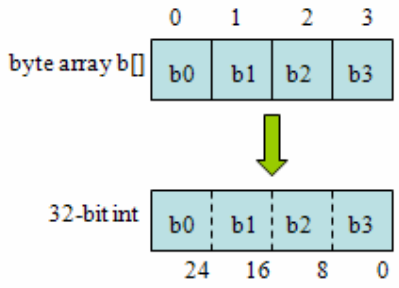

Figure 4. From Byte Array to Integer.

The code is concise but a tad opaque.

```
private static int byteArrayToInt(byte [] b)
// shift the first 4 elements of a byte array into an integer
{
  return (b[0] << 24) + ((b[1] & 0xFF) << 16) +
         ((b[2] & 0xFF) << 8) + (b[3] & 0xFF);
}
```

2.3. Sending a Message

The user can send "broadcast" or "sensor-update" messages over to Scratch by calling the ScratchIO methods broadcastMsg() or updateMsg(). These are wrappers for calls to the private sendMsg() method.

```
public boolean broadcastMsg(String msg)
{  return sendMsg("broadcast \"" + msg + "\"");  }

public boolean updateMsg(String name, String value)
{  return sendMsg("sensor-update " + name + " " + value);  }
```

broadcastMsg() places double quotes around its message argument, just in case it contains spaces. Scratch incorrectly processes unquoted strings containing white space.

The lack of quotes around the name and value arguments of updateMsg() is due to the assumption that 'name' will be a Java-style identifier and 'value' a numerical value, neither of which contain spaces.

sendMsg() writes the bytes of the message onto the output stream heading into Scratch. It prefixes the message with 4 bytes holding the size of that message.

```
private boolean sendMsg(String msg)
{
  if (out == null) {
    System.err.println("Output stream error");
    return false;
```



```
  }

  try {   // write message size, then the message
    byte[] sizeBytes = intToByteArray( msg.length() );
    for (int i = 0; i < NUM_BYTES_SIZE; i++)
      out.write(sizeBytes[i]);
    out.write(msg.getBytes());
  }
  catch (IOException e) {
    System.err.println("Couldn't send: " + msg);
    return false;
  }
  return true;
}  // end of sendMsg()
```

intToByteArray() does the opposite of byteArrayToInt(), using its bit operations to extract four bytes from the integer and store them in an array, as shown in Figure 5.

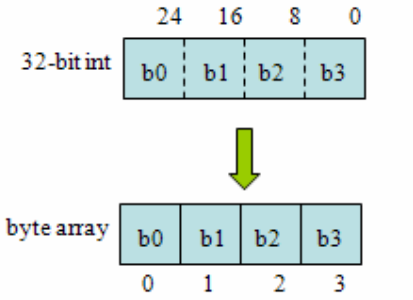

Figure 5. From Integer to Byte Array.

The code for intToByteArray():

```
private byte[] intToByteArray(int value)
{ return new byte[] {
          (byte)(value >>> 24), (byte)(value >> 16 & 0xff),
          (byte)(value >> 8 & 0xff), (byte)(value & 0xff) };
}
```

2.4. The Scratch Message Class

A ScratchMessage object is essentially a container for four pieces of data:

- the message type, which will be either BROADCAST_MSG, SENSOR_UPDATE_MSG, or UNKNOWN_MSG;
- the message type in string form (to make it easier to print);
- the message name. This does not include the "Scratch-" prefix added to the name in a "sensor-update" message;
- the "sensor-update" message value as a string. The value will be null for a "broadcast" message.

ScratchMessage includes get methods for accessing these fields: getMessageType(), getMessageTypeStr(), getName(), getValue(), and toString() for converting an object back into a string.

The only complexity in the class is the parsing of the original text message carried out in the constructor. The message is tokenized using my StringToks.parseTokens() method. StringToks is very similar to Java's built-in StringTokenizer class except that it treats text inside double quotes (e.g. "this is an example") as a single token.

Having introduced the Java Scratch classes, the rest of this chapter will be given over to examples showing how the classes are used. I'll begin with three short examples showing how Java interacts with Scratch's "broadcast" messages, global variables, and virtual sensors. Then I'll explain two slightly larger examples: an animated clock and a trampoline game, which show some of the coding issues when writing more realistically-sized applications.

3. Broadcasting Messages

The broadcasting of messages between Java and Scratch is illustrated by the example shown in Figure 6.

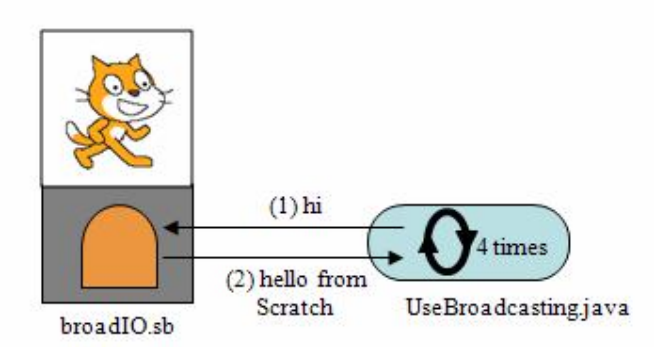

Figure 6. Broadcasting Messages.

The Scratch side of the application is a single script stored in broadIO.sb, and the Java code is in UseBroadcasting.java.

UseBroadcasting connects to Scratch's socket and sends a "hi" broadcast message. It waits for a reply and prints it out. This exciting handshaking is carried out four times.

```
public class UseBroadcasting
{
  public static void main(String argv[])
  {
    ScratchIO sio = new ScratchIO();

    for(int i=0; i < 4; i++) {
```



```
      System.out.println(i + ". sending \"hi\"");
      sio.broadcastMsg("hi");
      ScratchMessage msg = sio.readMsg();
      System.out.println( "  " + i + ". received: " + msg);
      try {
        Thread.sleep(1000);
      }
      catch (InterruptedException e) {}
    }
    sio.closeDown();
  }  // end of main()
}  // end of UseBroadcasting class
```

UseBroadcasting can easily fail:

```
> java UseBroadcasting
Scratch IO link could not be created
```

This occurs if Scratch's remote sensor connections haven't been initialized, which means that there's no socket waiting for the Java client to use. Inside Scratch, the user must right click on the sensor command block in the 'sensing' blocks collection and select the "enable remote sensor connections" menu item (as shown in Figure 3).

The script in broadIO.sb is shown in Figure 7.

Figure 7. The broadIO.sb Script.

The script is triggered when a "hi" broadcast message arrives, making Scratch the cat meow and display a "Hello!" speech bubble. After 2 seconds, "hello from Scratch" is broadcast, which will arrive at UseBroadcasting and be displayed.

The output from UseBroadcasting:

```
> java UseBroadcasting
0. sending "hi"
  0. received: broadcast hello from Scratch
1. sending "hi"
  1. received: broadcast hello from Scratch
2. sending "hi"
  2. received: broadcast hello from Scratch
3. sending "hi"
  3. received: broadcast hello from Scratch
```

4. Monitoring Global Variables

The use of Scratch global variables to send information to Java is illustrated by the example shown in Figure 8.

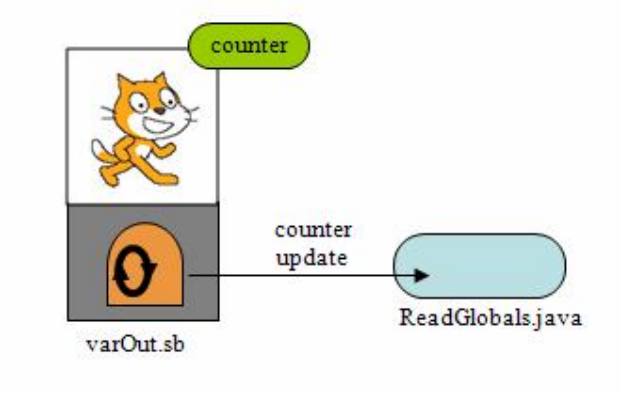

Figure 8. Monitoring Global Variables.

The Scratch side of the application is a single script stored in varOut.sb (see Figure 9), which uses a global variable called counter. The Java code is in ReadGlobals.java.

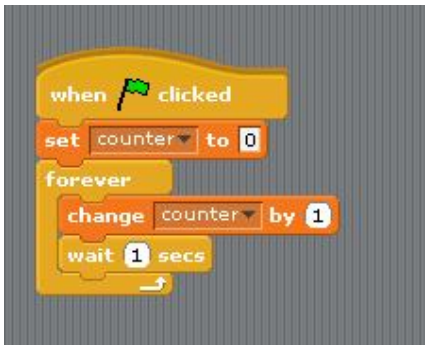

Figure 9. The varOut.sb Script.

After the script is started, counter is incremented every second.

ReadGlobals waits for a message and prints out it's details.

```
public class ReadGlobals
{
  public static void main(String argv[])
  {
    ScratchIO sio = new ScratchIO();
    ScratchMessage scratchMsg;
    while(true) {
      scratchMsg = sio.readMsg();
      if (scratchMsg != null) {
        System.out.println(scratchMsg);
        System.out.println("Message type: " +
                                  scratchMsg.getMessageTypeStr());
        System.out.println("        Name: " + scratchMsg.getName());
        System.out.println("       Value: " + scratchMsg.getValue());
      }
```



```
    }
  }  // end of main()
}  // end of ReadGlobals class
```

A "sensor-update" message is only transmitted from Scratch if remote sensor connections have been enabled, and only when a global variable changes.

Some sample output from ReadGlobals:

```
> java ReadGlobals
sensor-update counter 4
Message type: sensor-update
        Name: counter
       Value: 4
sensor-update counter 5
Message type: sensor-update
        Name: counter
       Value: 5
sensor-update counter 6
Message type: sensor-update
        Name: counter
       Value: 6
        :
```

During testing, it's useful to display the global variable on the Scratch stage. It's also possible to include a slider in the display, so the variable's value can be adjusted manually (see Figure 10). Whenever the value is changed, a "sensor-update" message will be output, and subsequently reported by ReadGlobals.

Figure 10. The Scratch Display of Counter.

5. Using Virtual Sensors

If Java wants to change a Scratch variable then it must be defined as a virtual sensor. However, it appears impossible to write a Scratch script using a virtual sensor, because there's no way to declare one inside Scratch

Although a virtual sensor can't be created from within Scratch, it is possible to write a Java program that creates one. Once a variable exists, it can be used in scripts.

MakeSensor.java contains code for creating a virtual sensor:

```
public class MakeSensor
{
  public static void main(String args[])
  {
    if (args.length != 2)
      System.out.println("Usage: MakeSensor <name> <value>");
    else {
```



```
      ScratchIO sio = new ScratchIO();
      sio.updateMsg(args[0], args[1]);
      sio.closeDown();
    }
  }  // end of main()
}  // end of MakeSensor class
```

The user calls MakeSensor with a name and initial value, and that virtual sensor will appear as a new sensor in Scratch. For example:

```
> java MakeSensor foobar 23
```

Scratch will now contain a foobar virtual sensor, as shown in Figure 11.

Figure 11. The foobar Virtual Sensor.

For a sensor to be displayed as in Figure 11, it must be selected from the 'slider' command block's pop down list (which is in the 'sensing' blocks collection), and it's 'view on stage' checkbox selected.

The following example uses all the remote sensor connection capabilities: a virtual sensor, a "broadcast" message, and a global variable. The application is shown in Figure 12.

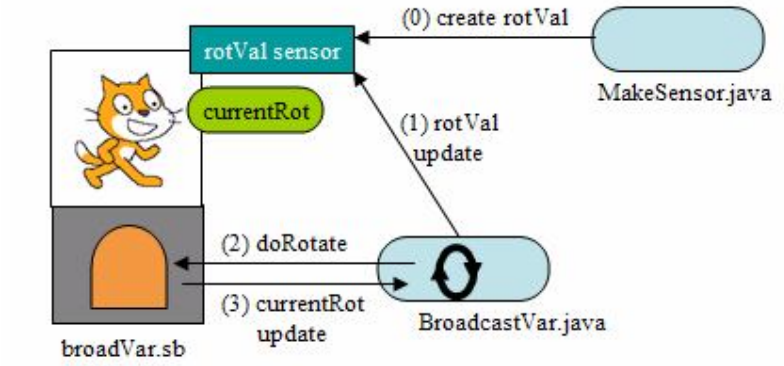

Figure 12. A Virtual Sensor Example.

Initially, I used MakeSensor.java to create a virtual sensor called rotVal in Scratch, and then I wrote the script (in broadVar.sb) that utilizes it.

BroadcastVar.java changes rotVal, then triggers the broadVar.sb script by broadcasting a "doRotate" message. The script uses rotVal as the number of degrees to rotate the cat sprite, and adds rotVal to a global variable called currentRot. The

change to currentRot is seen by BroadcastVar via a "sensor-update" message, which makes BroadcastVar start over and send another rotVal value to Scratch.

The three forms of communication used in BroadcastVar are labeled in Figure 12, and in code below:

```
public class BroadcastVar
{
  public static void main(String argv[])
  {
    int[] rotations = {15, 45, 90, 30, 30, 15, 45, 90};
    ScratchIO sio = new ScratchIO();

    for(int i=0; i < rotations.length; i++) {
      System.out.println("Rotating by " + rotations[i]);
      sio.updateMsg("rotVal", "" + rotations[i]);  // 1.update sensor
      sio.broadcastMsg("doRotate");          // 2. send trigger msg

      ScratchMessage msg = sio.readMsg();    // 3. read global
      System.out.println("Received: " + msg);
      try {
        Thread.sleep(1000);
      }
      catch (InterruptedException e) {}
    }
    sio.closeDown();
  }  // end of main()
}  // end of BroadcastVar class
```

The broadVar.sb script is shown in Figure 13.

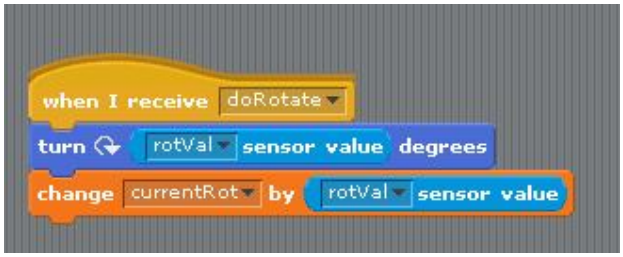

Figure 13. The broadVar.sb Script

It utilizes the rotVal virtual sensor and the currentRot global variable, both shown on Scratch's stage in Figure 14.

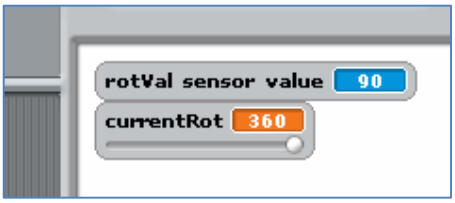

Figure 14. The Virtual Sensor and Global used in broadVar.sb.

The script makes the cat sprite rotate clockwise through 360 degrees over the course of a few seconds. The output from BroadcastVar.java is:

```
> java BroadcastVar
Rotating by 15
Received: sensor-update currentRot 15
Rotating by 45
Received: sensor-update currentRot 60
Rotating by 90
Received: sensor-update currentRot 150
Rotating by 30
Received: sensor-update currentRot 180
Rotating by 30
Received: sensor-update currentRot 210
Rotating by 15
Received: sensor-update currentRot 225
Rotating by 45
Received: sensor-update currentRot 270
Rotating by 90
Received: sensor-update currentRot 360
>
```

This example illustrates a useful coding technique for sending data from Java to Scratch – a virtual sensor is updated with the information by Java, then the script that needs it is woken up with a separate "broadcast" message.

6. Keeping a Scratch Clock On-time

The example in this section illustrates a common type of Scratch-to-Java link – when Scratch uses Java as an information supplier. For instance, Java can be employed to deliver data from the Web, from the OS, or act as an interface to non-standard peripherals.

I'll keep things simple here by having Scratch use Java as a time source to periodically update a clock. The example is shown in overview in Figure 15.

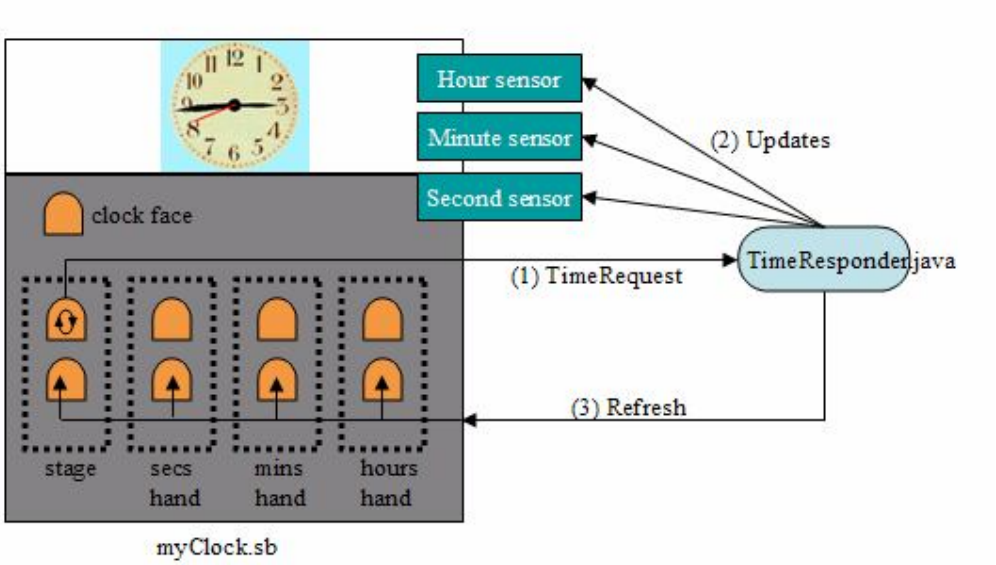

Figure 15. The Clock Example.

myClock.sb displays a clock with hours, minutes, and seconds hands, driven by Scratch's built-in timer. The application also periodically sends a time request to a Java TimeResponder object, and the returned information is utilized to correct the position of the clock hands.

A "TimeRequest" message is broadcast to TimeResponder, which responds by updating the Hour, Minute, and Second virtual sensors. These updates are followed by TimeResponder broadcasting a "Refresh" message into Scratch. This wakes up four scripts, three of which update the clock hands with the sensor values, and the other resets the Scratch timer.

Of note is the coding technique, first seen in the previous example – the use of a broadcast message to wake up scripts after virtual sensors have been changed.

This example is closely based on "System Clock" by Paddle2See at http://scratchconnections.wik.is/Projects. His version uses a Python timer, and the Python code drives the clock rather than the Scratch scripts. I've also taken the liberty of reusing his excellent clock face and hands graphics.

6.1. Starting the Clock

Each of the clock hands (hours, minutes, seconds) has a script that moves the hand around the face at the correct speed. These scripts are shown in Figures 16-18.

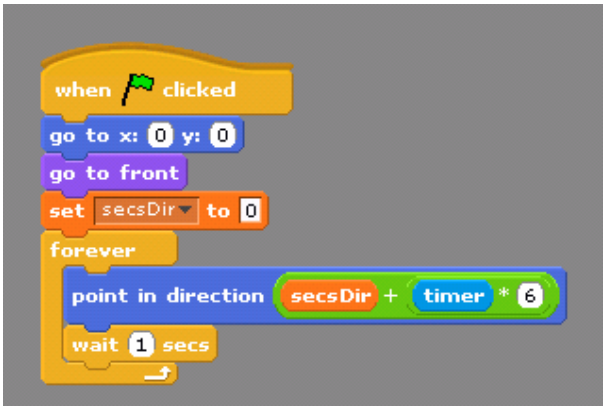

Figure 16. Moving the Seconds Hand.

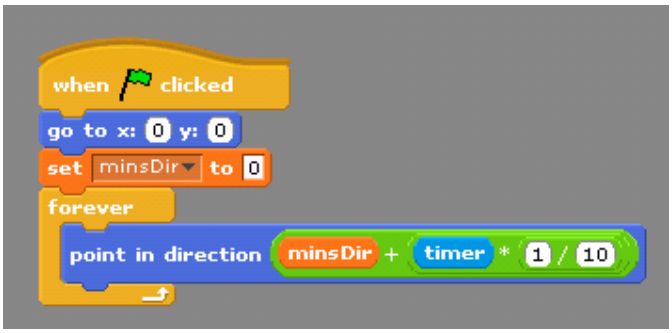

Figure 17. Moving the Minutes Hand.

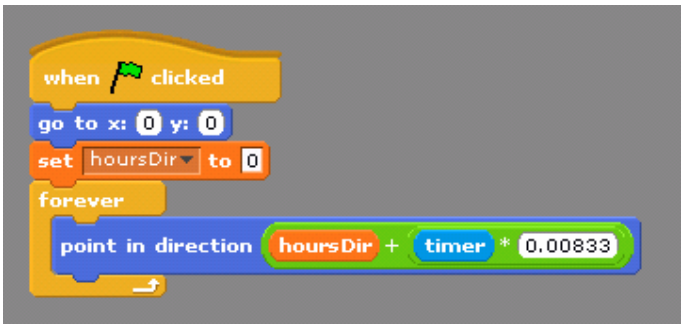

Figure 18. Moving the Hours Hand.

The direction of a hand is calculated using a combination of a sprite variable (secsDir for the seconds hand, minsDir for the minutes hand, and hoursDir for the hours hand), and the current timer value. Initially, secsDir, minsDir, and hoursDir are 0, but they are modified after each Java time update.

6.2. Requesting the Time

A script associated with the stage is used to reset the Scratch timer to 0, and to periodically broadcast time requests to Java (see Figure 19).

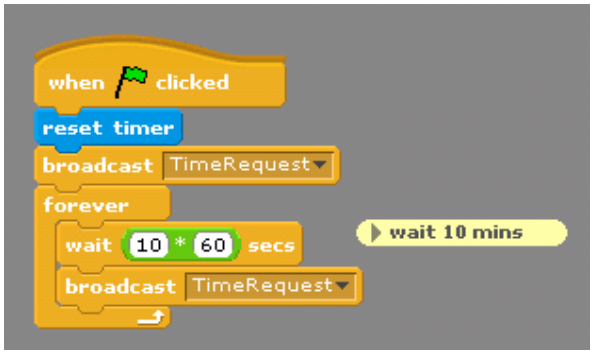

Figure 19. Requesting the Time.

The delay between requests (10 minutes) is arbitrary, and could be changed.

6.3. Responding to a Time Request

The TimeResponder Java code waits for a broadcast message to arrive, using it as a signal to get the current time (using a Calendar object). The component hours, minutes, and seconds are passed to Scratch by sending three "sensor-update" messages to the virtual sensors called Hour, Minute, and Second. These updates are followed by a "Refresh" broadcast that triggers four scripts over in Scratch.

```
public class TimeResponder
{
  public static void main(String[] args)
  {
    SimpleDateFormat fmt = new SimpleDateFormat("h:mm:ss a");
    ScratchIO sio = new ScratchIO();
    while (true) {
      System.out.println("Waiting...");
      ScratchMessage msg = sio.readMsg();  // wait for a msg
      System.out.println("Received: " + msg);
      sendRefresh(sio, fmt);
    }
  }   // end of main()


  private static void sendRefresh(ScratchIO sio,
                                  SimpleDateFormat fmt)
  // update the virtual time sensors; broadcast a refresh
  {
    Calendar c = Calendar.getInstance();
    System.out.println("Sending time: " + fmt.format( c.getTime() ));

    sio.updateMsg("Hour", ""+c.get(Calendar.HOUR));
    sio.updateMsg("Minute", ""+c.get(Calendar.MINUTE));
    sio.updateMsg("Second", ""+c.get(Calendar.SECOND));

    sio.broadcastMsg("Refresh");
  } // end of sendRefresh()

} // class TimeResponder
```

There's a subtle weakness in the code – it doesn't check the name of the incoming broadcast message to see if it really is "TimeRequest". This can be easily remedied with the addition of the line in bold:

```
    :
ScratchMessage msg = sio.readMsg();
System.out.println("Received: " + msg);
if (msg.getName().equals("TimeRequest")
  sendRefresh(sio, fmt);
   :
```

TimerResponder's correct functioning depends on the Scratch code having its remote sensor connection enabled. Also, the Scratch code will only begin executing when the user clicks on the IDE's green flag, thereby starting the stage and hands scripts shown in Figures 16-19. It's also necessary to have previously created the three virtual sensors (Hour, Minute, and Second) with MakeSensor.java.

6.4. Refreshing the Clock

As Figure 15 indicates, a "Refresh' message broadcast from the Java side, causes four Scratch scripts to wake up. The scripts associated with the clock hands read the virtual sensor relevant to them, and update their direction variable (see Figures 20-22).

Figure 20. Updating the Seconds Hand Direction.

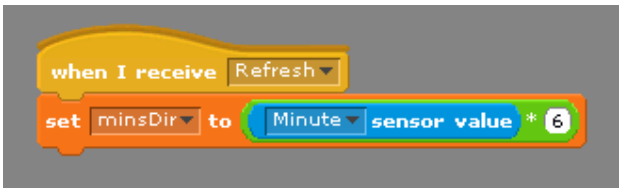

Figure 21. Updating the Minutes Hand Direction.

Figure 22. Updating the Hours Hand Direction.

The seconds hand script updates secsDir using the Second virtual sensor, the minutes hand updates minsDir using the Minute virtual sensor, and the hours hand script modifies hoursDir with the Hour and Minute sensors.

The scripts in Figures 20-22 can only be written once their virtual sensors have been created, which can be done with calls to MakeSensor.java:

```
> java MakeSensor Second 0
> java MakeSensor Minute 0
> java MakeSensor Hour 0
```

The values assigned to the sensors don't matter since they'll be set by TimeResponder when the Scratch application starts.

The sensors will only be created if remote sensor connections have been enabled inside Scratch. This can be confirmed by making the sensors visible on the stage, by selecting their 'view on stage' checkboxes (as shown in Figure 11 for sensor foobar).

A "Refresh" message also activates a script in the stage which resets the timer to 0 (Figure 23).

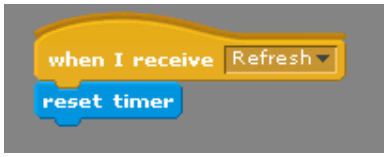

Figure 23. Resetting the Timer.

These modification to the three direction variables and the timer will be employed in future updates to the hands (see Figures 16-18).

7. A Java GUI for a Scratch Game

The previous example showed how Scratch could make use of Java as an information source. Scratch was in charge, and called Java when it needed assistance.

The example in this section deals with the other main way of linking Java and Scratch – Java is the 'boss', telling the Scratch application what to do and when, and receiving data from Scratch about its current status.

The application is a variant of the Trampoline game (see Figure 24) which is included as one of Scratch's example projects (in the Games sub-directory).

Figure 24. The Original Trampoline Game.

The user interface of the original game is based on four scripts which wake up in response to arrow key pressed by the user. My version replaces this interface with a Java GUI shown in Figure 25.

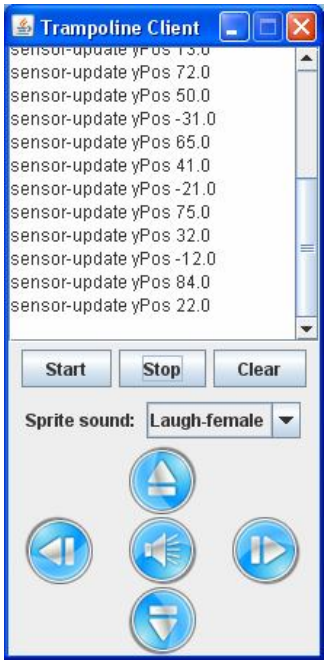

Figure 25. The Java GUI for the Modified Trampoline Game.

The "Start" and "Stop" buttons start and stop the girl bouncing up and down, the "up" button makes her body briefly twist, the "left" and "right" buttons make her rotate anti-clockwise and clockwise, and the "down" button makes her touch her toes. The "sound" button tells Scratch to play a sound, which depends on the selection in the GUI's combo box.

The ongoing vertical position of the girl is reported in the GUI's scrolling text area.

The Scratch code which carries out these tasks is substantially the same as in the original Trampoline game, except that the scripts are triggered by broadcast messages sent from the Java GUI rather than by key presses. An overview of the application is shown in Figure 26.

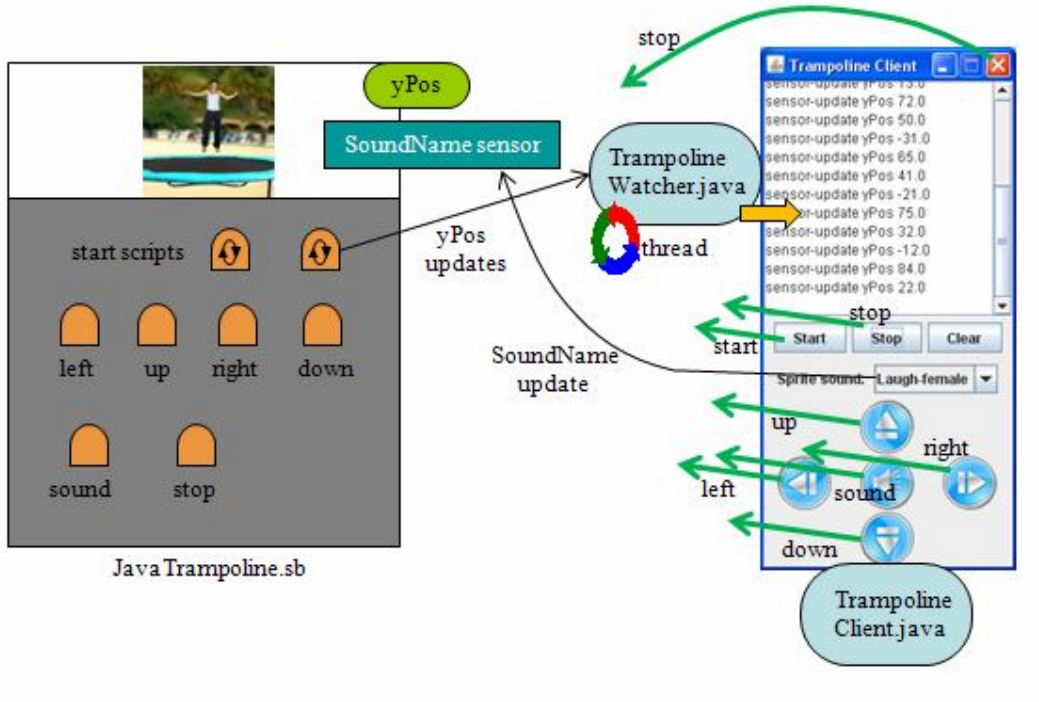

Figure 26. The Java Trampoline Game.

Almost all the GUI controls work by broadcasting messages to Scratch. For example, when the user presses the central "Sound" button, it broadcasts a "soundEvent" message.

The GUI sends seven different kinds of messages, which are processed over in Scratch by eight scripts (a"startEvent" message triggers two scripts).

The Scratch application also contains a yPos global variable and a soundName virtual sensor. yPos is used to transmit the girl's vertical position over to Java, while soundName is set by the GUI's combo box of sound names.

The processing of the yPos data is handled by a separate Java thread. This allows the GUI to execute without the need to periodically wait for messages coming from Scratch. This illustrates a general coding issue: since Scratch is threaded, a script can output messages at any time, and the best way of handling that externally is with some kind of watcher, running in its own thread or process. This suggests that the language being interfaced to Scratch should support threads (or processes).

7.1. The Trampoline Client

Most of the TrampolineClient class is given over to the creation of the GUI, which I won't describe here. The constructor also starts the watcher thread.

```
// globals
private ScratchIO scratchIO = null;
private TrampolineWatcher watcher;
```

```
public TrampolineClient()
{
   super("Trampoline Client");
   scratchIO = new ScratchIO();

   initializeGUI();
   watcher = new TrampolineWatcher(this, scratchIO);
   watcher.start();    // start watching for scratch msgs

   addWindowListener( new WindowAdapter() {
     public void windowClosing(WindowEvent e)
     { closeDown(); }
   });

   setSize(220,450);
   setResizable(false);
   positionOnRight();
   setVisible(true);
} // end of TrampolineClient();
```

The closeDown() method broadcasts a "stopEvent" message to Scratch, terminates the watcher, and closes the Scratch connection.

```
private void closeDown()
{
  watcher.finish();
  scratchIO.broadcastMsg("stopEvent");
  scratchIO.closeDown();
  System.exit(0);
}
```

The other broadcast messages are sent from actionPerformed(), which is called when the user interacts with the GUI.

```
public void actionPerformed(ActionEvent e)
/* A button has been pressed; one of:
       jbStart, jbStop, jbLeft, jbUp, jbRight, jbDown, jbSound
   or a sound names combo box item has been selected.
*/
{
  // start, stop, and clear buttons
  if (e.getSource() == jbStart)
    scratchIO.broadcastMsg("startEvent");
  else if (e.getSource() == jbStop)
    scratchIO.broadcastMsg("stopEvent");
  else if (e.getSource() == jbClear)
    jtaMesgs.setText(null);

  // image-based buttons
  else if (e.getSource() == jbLeft)
    scratchIO.broadcastMsg("leftEvent");
  else if (e.getSource() == jbUp)
    scratchIO.broadcastMsg("upEvent");
  else if (e.getSource() == jbRight)
    scratchIO.broadcastMsg("rightEvent");
  else if (e.getSource() == jbDown)
    scratchIO.broadcastMsg("downEvent");
  else if (e.getSource() == jbSound)
```

```
    scratchIO.broadcastMsg("soundEvent");

  // sound names combo box list
  else if (e.getSource() == soundsList) {
    String soundName = (String)soundsList.getSelectedItem();
    System.out.println("Selected: " + soundName);
    scratchIO.updateMsg("soundName", soundName);
  }
  else
    System.out.println("Unknown GUI action: " + e);
}  // end of actionPerformed()
```

Once the GUI component that triggered actionPerformed() is identified, then the corresponding message is broadcast to Scratch.

The only slightly unusual coding is for the selection of a sound name from the combo box. The soundName virtual sensor must be updated, and so a "sensor-update" message is sent out containing "soundName" and the name of the selected sound. The soundName value will be read whenever Scratch is requested to play a sound, which occurs when a "soundEvent" message is sent out after the user presses the GUI's sound button.

7.2. Watching Scratch

TrampolineWatcher is a simple threaded class which spends most of it time waiting to receive messages from Scratch.

```
// globals
private TrampolineClient client;  // link to client
private ScratchIO scratchIO;      // link to scratch
private volatile boolean isFinished;


public void run()
// Read a scratch message, display it in the GUI, repeat.
{
  ScratchMessage scratchMsg;
  try {
    while (!isFinished) {
      scratchMsg = scratchIO.readMsg();
      if (scratchMsg != null)
        client.showMsg(scratchMsg.toString() + "\n");
    }
  }
  catch(Exception e)   // socket closure causes termination of while
  {  JOptionPane.showMessageDialog( null, "Link to Scratch Lost!",
                  "Connection Closed", JOptionPane.ERROR_MESSAGE);
     System.exit(0);
  }
} // end of run()
```

It's the possibility that ScratchIO.readMsg() will block the watcher for long periods of time which motivated the movement of this code into its own thread. A watcher thread similar to this one will be needed in most applications that wait for messages from Scratch.

7.3. Starting and Stopping the Game

JavaTrampoline.sb utilizes eight scripts, which are triggered by various broadcast messages coming from Java.

When a "startEvent" message arrives (after the user presses the "Start" button), the two scripts shown in Figure 27 are activated.

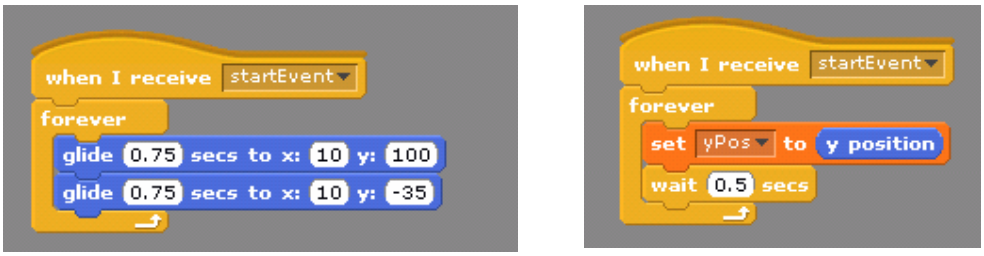

Figure 27. Scripts Triggered by "startEvent".

The left hand script moves the girl sprite up and down, ranging between a y axis position of 100 and -35. The right hand script updates the yPos global with the current y position every 0.5 seconds. This causes a series of "sensor-update" messages to be transmitted to Java where the TrampolineWatcher thread displays them in the GUI's text area.

A "stopEvent" message is sent to Scratch when the user presses the "Stop" button. This is handled by a script that simply stops all the scripts (see Figure 28).

Figure 28. The Script Triggered by "stopEvent".

7.4. Other Game Messages

The other five scripts deal with the messages sent from the image buttons in the GUI: "leftEvent", "upEvent", "rightEvent", "downEvent", and "soundEvent". For example, "rightEvent" causes the girl sprite to rotate in a clockwise direction, using the script in Figure 29.

Figure 29. The Script Triggered by "rightEvent".

The arrival of a "soundEvent" message makes the girl pull in her arms and legs, a sound is played, and she then returns to jumping (as implemented in Figure 30).

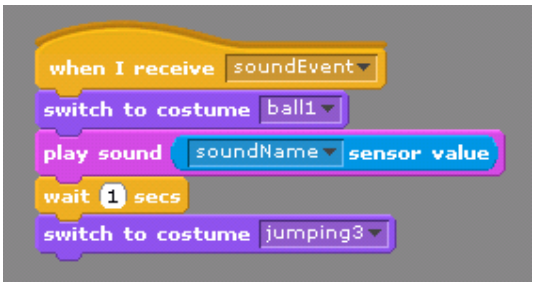

Figure 30. The Script Triggered by "soundEvent".

The choice of sound comes from reading the soundName virtual sensor, which is set by a "sensor-update" message broadcast by the GUI's combo box.

7.5. Debugging the Scratch Code

The Scratch half of the application can be tested and debugged fairly easily without involving the Java trampoline code.

The scripts triggered by broadcast messages need to be rewritten to wake up in response to key presses. The two scripts that are activated because of "startEvent" can be recoded to respond to the pressing of Scratch's green start flag.

The "soundEvent" script (Figure 30) cannot be tested until a soundName virtual sensor has been created, which can be done by calling MakeSensor:

```
> java MakeSensor soundName "Laugh-female"
```

The sound name should match one of the sounds associated with the girl sprite.

The call to MakeSensor will only work if remote sensor connections have been enabled inside Scratch. This can be confirmed by making the soundName sensor visible on the stage, by selecting its 'view on stage' checkbox (as shown in Figure 11 for sensor foobar).