JavaArt Chapter 2. Executing Pixels using Drag-and-Drop

This is the second part of a two-chapter discussion on how to convert a Java program into an 'executable' image: a picture that can be executed just like its original source code. An example image is shown in Figure 1 (enlarged by a factor of 4 to make it easier to see).

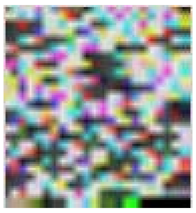


Figure 1. An Executable Image.

This chapter is also about how to make a drag-and-drop Java application (called ExecutePixels). A user can drop an executable image onto ExecutePixel's icon, and have the image's program spring into life. This is surprisingly difficult to implement in Java alone, and I look at how to get some extra help from batch files, VBScript, and executable wrapper builders (e.g. launch4j). Figure 2 illustrates what I want to do.

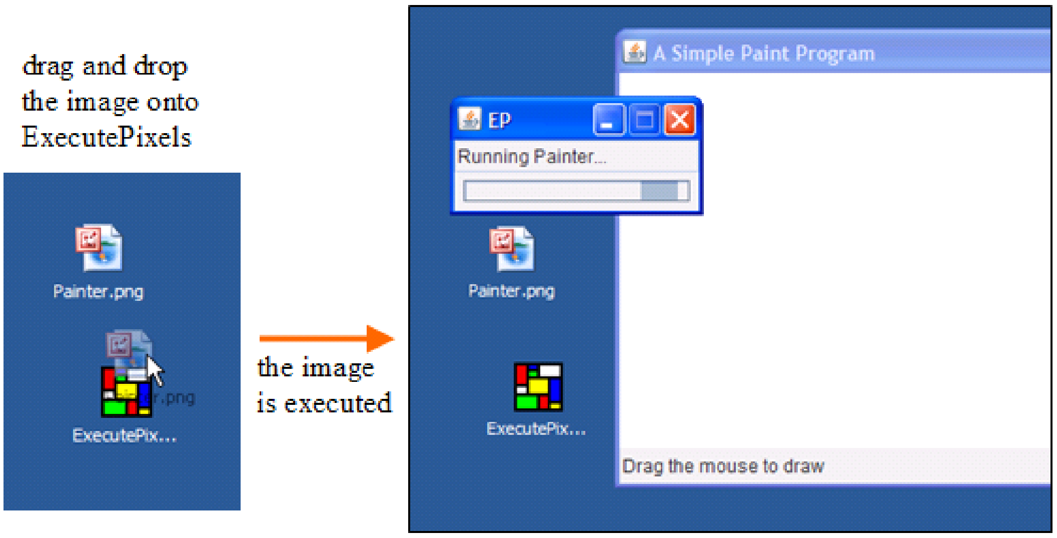


Figure 2. Executing an Image.

When the image is dropped onto ExecutePixels, it's dynamically translated back into Java, compiled to byte codes, and passed to the JVM for execution. The translation, compilation, and execution are carried out by ExecutePixels "on-the-fly", without creating any temporary files on the machine (i.e. no Java or class files are saved).

The first chapter of this two-parter looked at implementing on-the-fly dynamic compilation with Java 6's Compiler API, and with the Janino compiler API (http://www.janino.net/). Due to Janino's small size, its open source license, and its more direct support for dynamic compilation, I'll employ Janino here.

1. There and Back Again

The translation of Java text to an image, and of an image back to text is handled by ImageText.java. It offers two text-to-image static methods:

- BufferedImage textToImage(String fnm);
- boolean textToPng(String fnm);

and two image-to-text static methods:

- String pngToString(String fnm);
- boolean pngToText(String fnm);

I'll explain what these methods do in the next two sections.

1.1. From Text to Image

textToImage() reads ASCII text from a file, and uses it to create a square-ish BufferedImage. The other method, textToPng(), calls textToImage() and saves the resulting image into a PNG file.

textToImage() uses readCode() to copy the file's text into an ArrayList.

```
// global
private static final int BRIGHTNESS = 2;
     /* scale factor for increasing the brightness of the
        colours stored in the image pixels */

private static boolean readCode(String fnm,
                              ArrayList<Integer> pixColors)
{
  int numUnPrintables = 0;     // number of unprintable chars
  try {
    FileReader fr = new FileReader(fnm);
    int ch;
    while ((ch = fr.read()) != -1) {
      if (!isPrintable(ch)) {  // if character not printable
        ch = 32;    // replace with a space char
        numUnPrintables++;
      }
      pixColors.add(ch*BRIGHTNESS);    // increase brightness
    }
    fr.close();

    if (numUnPrintables > 0)
      System.out.println( numUnPrintables +
                  " chars unprintable; all set to spaces");
    System.out.println("Read in: " + fnm);
  }
  catch (IOException e)
  { System.out.println("Error reading: " + fnm);
    return false;
  }

  return true;
}  // end of readCode()
```

The range of acceptable characters is restricted to printable 7-bit ASCII, which allows each character's integer value to be multiplied by two to produce a number using no more than 8 bits. The multiplication is carried out to make the colours brighter in the final image.

The ArrayList is passed to createImage(), which repeatedly reads three values from the list. Each triple is combined into a single 32-bit integer, which becomes the colour of a pixel in the image. The idea is illustrated by Figure 3.

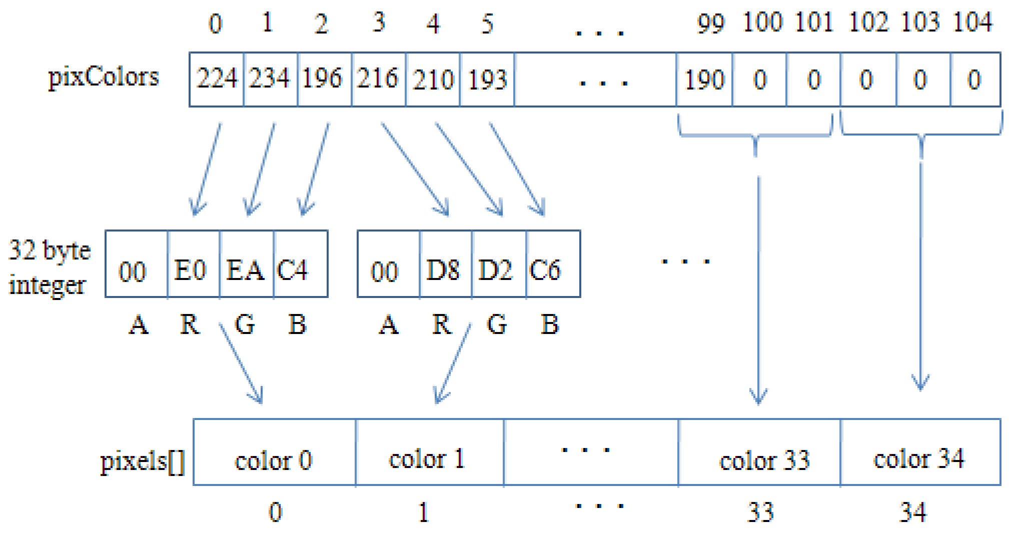


Figure 3. From Integer ArrayList to Pixel Array.

Each 32-bit integer is made up of four bytes representing the alpha (A), red (R), green (G), and blue(B) channels in the pixel. I decided not to use the alpha channel, so the pixels would all be opaque.

createImage() is defined as:

```
private static BufferedImage createImage(
                                  ArrayList<Integer> pixColors )
{
  Dimension imDim = calcImageDimensions(pixColors);

  // use an array to store the pixel values
  int len = imDim.width * imDim.height;
  int[] pixels = new int[len];

  int idx = 0;
  int redVal, greenVal, blueVal;
  for (int i=0; i < len; i++) {
    redVal = pixColors.get(idx);    // get values for the RGB channels
    greenVal = pixColors.get(idx+1);
    blueVal = pixColors.get(idx+2);

    // store the three values as a single pixel value
    pixels[i] = blueVal | (greenVal << 8) | (redVal << 16);
    idx += 3;
  }

  // create a bufferedImage for the pixels array
  BufferedImage im = new BufferedImage(imDim.width, imDim.height,
                                       BufferedImage.TYPE_INT_RGB);
  // fill the image with the pixels array data
  im.setRGB(0, 0, imDim.width, imDim.height, pixels, 0, imDim.width);
  if (im == null)
    System.out.println("No image created");
  return im;
}  // end of createImage()
```

At the end of createImage(), the pixels[] array is copied into a BufferedImage by a single call to BufferedImage.setRGB(). The array data is stored row-by-row, as illustrated in Figure 4.

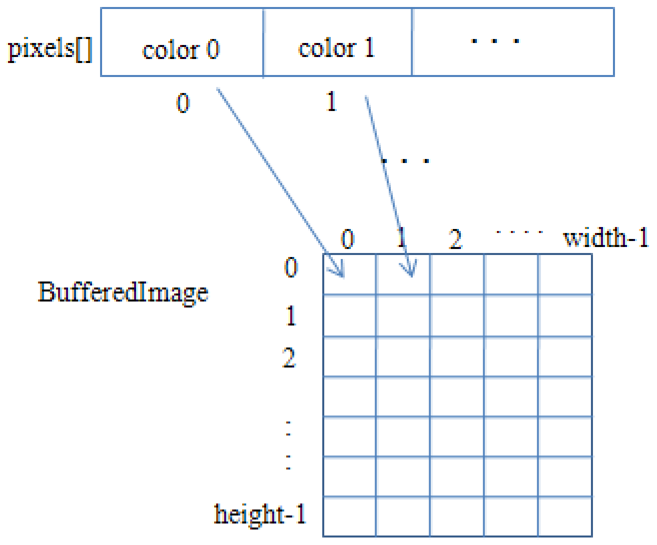


Figure 4. From Pixel Array to BufferedImage.

Calculating the Picture Size

A surprisingly tricky aspect of the text-to-image translation is calculating the dimensions of the BufferedImage. One issue is that the image should be roughly square (for aesthetic reasons), and also each pixel requires three numbers from the ArrayList.

Unfortunately, the ArrayList is unlikely to contain a multiple of three values, so it's necessary to pad it out with 'dummy' numbers (I chose '0' since it isn't a printable ASCII code). Even then, the chosen image dimensions (roughly square) will often require even more ArrayList filler.

For example, assume that the original ArrayList holds 100 values. This means that the list will need to be padded out with two '0's, to make it a multiple of three (i.e. 102%3 == 0). Therefore, the resulting image will use 102/3 == 34 pixels.

A simple way of calculating the image's square-ish dimensions is to start with the square root of the number of pixels, truncated to an integer. In my example, Math.sqrt(34) == 5, meaning that, as a first approximation, I should create an image 5x5 in size. This is too small for 34 pixels, so one of the dimensions is increased (e.g. the height) until the image is big enough. A dimension of 7x5 is enough to hold 35 pixels, two more than necessary.

35 pixels is equivalent to an ArrayList of size 35*3 == 105 values. As a consequence, the list must be padded out with another five dummy values to make it the right length. This is the situation shown in Figure 3, where the ArrayList has five '0's at its end as filler, so a 35 pixel array can be constructed.

All this fiddling is managed by calcImageDimensions(), which returns the dimensions for the BufferedImage (e.g. 7x5 in the example above), and also pads out the ArrayList with '0's.

```
private static Dimension calcImageDimensions(
                                ArrayList<Integer> pixColors)
{
  int imWidth = 0;   // starting values for image's dimensions
  int imHeight = 0;

  int numColors = pixColors.size();
  /* make numColors a multiple of three, since 3 colors are needed
     for each pixel (its R, G, and B values) */
  if (numColors%3 != 0) {
    int extras = 3 - numColors%3;
    numColors += extras;
  }

  int pixLen = numColors/3;    // no. of pixels in the image

  // calculate roughly square width and height values for image
  imWidth = (int) Math.sqrt(pixLen);
  imHeight = pixLen/imWidth;

  /* increase height value until width*height is greater
    (or equal) to the number of pixels in the image */
  while ((imWidth*imHeight) < pixLen)
    imHeight++;

  /* add 0's to the pixColors list as filler so the list size
     matches the image dimensions */
  int numFillers = (imWidth*imHeight*3) - pixColors.size();
  System.out.println("Added " + numFillers + " filler colors");
  for (int i=0; i < numFillers; i++)
    pixColors.add(0);

  Dimension imDim =  new Dimension(imWidth, imHeight);
  return imDim;
}  // end of calcImageDimensions()
```

1.2. From Image to Text

The two image-to-text conversion methods are:

- String pngToString(String fnm);

- boolean pngToText(String fnm);

pngToString() converts the pixels in the supplied PNG file into ASCII text. pngToText() calls pngToString(), and writes the resulting string into a file with a ".txt" extension.

The BufferedImage loaded from the image file is converted into text in stages like those outlined for createImage() above, except carried out in reverse. First the BufferedImage is pulled apart into a pixels array (see Figure 5), then each pixel is split into its red, green, and blue channels (Figure 6), and their values are converted into characters that are stored in a string (Figure 7).

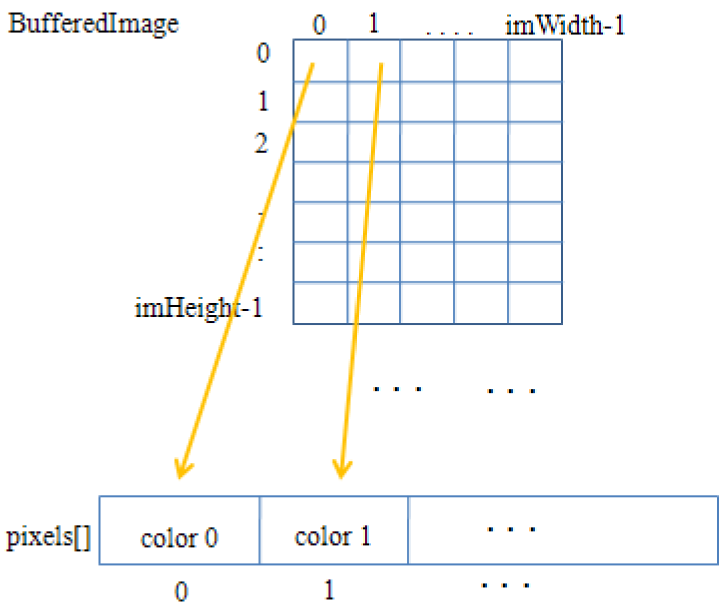


Figure 5. From BufferedImage to Pixel Array.

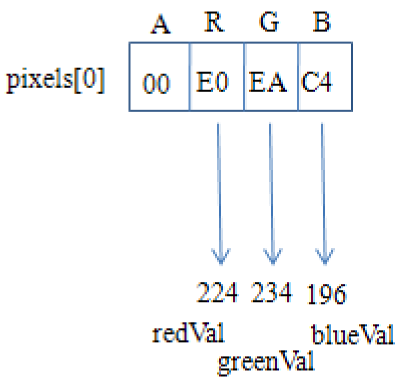


Figure 6. From Pixel to RGB Channel Values

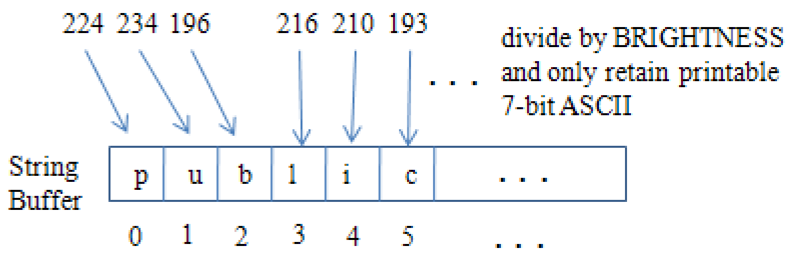


Figure 7. From Channel Values to String.

These transformations are implemented in convertImage():

```
private static String convertImage(BufferedImage im)
{
  int numUnPrintables = 0;    // number of unprintable chars found

  StringBuilder sb = new StringBuilder();

  // extract pixels from the image into an array
  int imWidth = im.getWidth();
  int imHeight = im.getHeight();
  int[] pixels = new int[imWidth * imHeight];
  im.getRGB(0, 0, imWidth, imHeight, pixels, 0, imWidth); //see Fig 4

  int redVal, greenVal, blueVal;
  for(int i=0; i < pixels.length; i++) {   //get RGB vals from pixels
    redVal = (pixels[i]>>16)&255;          // see Fig 5
    greenVal = (pixels[i]>>8)&255;
    blueVal = pixels[i]&255;

    // convert the RGB values to chars, and add them to sb (Fig 6)
    numUnPrintables += writeChar(sb, redVal);
    numUnPrintables += writeChar(sb, greenVal);
    numUnPrintables += writeChar(sb, blueVal);
  }

  if (numUnPrintables > 0)
    System.out.println(numUnPrintables +
         " unprintable characters were converted to spaces");

  return sb.toString();
}  // end of convertImage()
```

BufferedImage.getRGB() does the Figure 5 work of filling the pixel array. The Figure 6 task is carried out by a for-loop cycling through the array, pulling out channel values (redVal, greenVal, and blueVal) for each pixel. writeChar() does the Figure 7 job of adding the values to a StringBuffer object, sb.

writeChar() writes a value into the StringBuffer as a printable character. Part of this process is to remove the brightness factor, added when the text was changed to colours. Also, if the original character isn't printable, then a space is stored instead, and any dummy values ('0's) are ignored.

```
private static int writeChar(StringBuilder sb, int val)
```

```
{
  if (val == 0)  // don't store a filler value
    return 0;

  int ch = val/BRIGHTNESS;  // reduce colour brightness

  if (isPrintable(ch)) {
    sb.append( (char)ch );
    return 0;    // char is printable
  }
  else {   // save non-ASCII char as a space
    sb.append(' ');
    return 1;    // report that original char was unprintable
  }
}  // end of writeChar()
```

1.3. Testing the ImageText Class

The MakePNG class tests out ImageText.textToPng() by accepting a file name and generating a PNG file:

```
> java MakePNG Comparison.java
Read in: Comparison.java
Added 55 filler colors
Image written to PNG file: Comparison.png
Completed: true
```

The resulting Comparison.png image is shown in Figure 8, enlarged by a factor of four to make its pixels more visible.

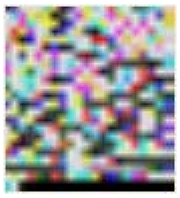


Figure 8. The Image Version of Comparison.java

The image had to be padded out with 55 ‘0’s, which are visible as a line of black pixels along the bottom row. The pixels are black because their RGB channels are all 0.

The MakePNG class does little more than call ImageText.textToPng():

```
public class MakePNG
{
  public static void main(String[] args)
  {
    if (args.length < 1)
      System.out.println("Usage: java MakePNG <file.java>");
    else {
      boolean completed = ImageText.textToPng(args[0]);
      System.out.println("Completed: " + completed);
    }
  }
}  // end of MakePNG class
```

The MakeTextFile class is a similar ‘test rig’ which uses ImageText.pngToText() to convert a PNG file into a text file.

```
> java MakeTextFile Comparison.png
Read Comparison.png
Code written to Comparison.txt
Completed: true
```

The result is a text file called Comparison.txt, which contains the Java code from the original Comparison.java file. After being renamed to Comparison.java, it can be compiled and executed in the same way as the original.

The MakeTextFile class is:

```
public class MakeTextFile
{
  public static void main(String[] args)
  {
    if (args.length < 1)
      System.out.println("Usage: java MakeTextFile <file.png>");
    else {
      boolean completed = ImageText.pngToText(args[0]);
      System.out.println("Completed: " + completed);
    }
  }
}  // end of MakeTextFile class
```

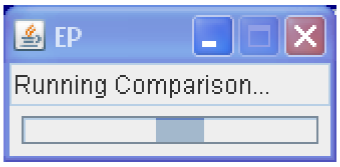
2. Executing an Image

The ExecutePixels class ‘executes’ the PNG file passed to it. First, the pixels in the file are converted into a string of ASCII characters using ImageText.pngToString(). ExecutePixels treats this string as the text of a complete Java application with a main() method requiring no input.

Janino's SimpleCompiler API is used to compile the Java text as a single "compilation unit" (i.e. treating the text as if it were the contents of a single ".java" file). Janino was described in the previous chapter, and can be downloaded from http://www.janino.net/.

The compiled code is executed by using reflection to start the application's main() method. Reflection was also explained in the previous chapter.

While the translation/compilation/execution phases are being carried out, ExecutePixels appears as a small GUI containing a status message and progress bar (see Figures 2 and 9). A timer is used to periodically update the status information.



Running Comparison...

Figure 9. The ExecutePixels Progress Window.

One of the design decisions that simplifies ExecutePixels is not to include an input/output window for the application, which means there's no way for it to read from stdin or write to stdout. The program must use dialog boxes, or other GUI means, to support IO interaction with the user.

ExecutePixels is started from the command line like so:

```
java -cp "janino.jar;." ExecutePixels Comparison.png
```

It's necessary to include the Janino JAR in the classpath, which is located in the same directory as the ExecutePixels class.

The command line approach is sufficient for testing, but it should be possible to simply drop a PNG file onto an ExecutePixels icon to start the image's execution (see Figure 2 for the general idea). I'll discuss several ways of achieving this later in the chapter, but for now I'll concentrate on the coding of ExecutePixels.java.

2.1. Initializing ExecutePixels

The ExecutePixels() constructor creates the GUI seen in Figure 9, starts the timer to report on how things are going, and tries to execute the PNG file.

```
public ExecutePixels(String[] args)
{
  super("EP");

  buildGUI();
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  pack();
  setResizable(false);
  setVisible(true);

  startReporting();
  execute(args);
  }  // end of ExecutePixels()
```

buildGUI() is conventional Swing code for creating a text field for status messages, and a indeterminate progress bar (a bar that keeps swinging backwards and forwards to indicate that ExecutePixels is doing something).

startReporting() creates the timer that periodically updates the text field. It also stops the progress bar swinging when an error occurs in the image's translation, compilation, or execution.

```
// globals
private JProgressBar progressBar;
private JTextField msgJTF;
private String progressMsg = "Starting...";

private boolean foundError = false;


private void startReporting()
// Fire a timer periodically to report on things
{
  javax.swing.Timer timer =    // call the listener every 500ms
     new javax.swing.Timer(500, new ActionListener() {
          public void actionPerformed(ActionEvent e)
          { if (foundError) {
              // stop the progress bar when there's an error
              progressBar.setIndeterminate(false);
              progressBar.setValue(20);
            }
            msgJTF.setText(progressMsg);  // update text field
          }
        });

  timer.start();
}  // end of startReporting()
```

2.2. Translating the Image

execute() calls ImageText.pngToString() to extract the image's code, which it passes to execCode() for compilation and execution.

```
private void execute(String[] args)
{
  if (args.length < 1)
    reportError("No image supplied");
  else {
    progressMsg = "Creating Code...";
    String codeStr = ImageText.pngToString(args[0]);
                       // translate image --> code
    if (codeStr == null)
      reportError("No code to execute");
    else
      execCode(codeStr);
  }
}  // end of execute()
```

2.3. Compiling and Executing the Image Code

execCode() compiles the code string using Janino's SimpleCompiler API, loads the class, and call its main() method with no inputs.

The details of how Janino works were explained in the previous chapter, so won't be discussed again. execCode() is very similar to the StringExecutor class example in section 7.2 of that chapter.

```
private void execCode(String codeStr)
{
  String className = getClassName(codeStr);
        // retrieve the public class name used in codeStr
  try {
    progressMsg = "Compiling " + className + "...";
    Class.forName( "org.codehaus.janino.SimpleCompiler" );
            // check if Janino's SimpleCompiler is present

    // compile the code in codeStr
    SimpleCompiler compiler = new SimpleCompiler();
    compiler.cook( new StringReader(codeStr) );

    // get the loaded class
    Class<?> cl = compiler.getClassLoader().loadClass(className);

    // get a reference to the class' main(String[]) method
    progressMsg = "Running " + className + "...";
    Method mainMeth = cl.getMethod("main",
                             new Class[] { String[].class });

    String[] args = new String[] { };     // no inputs for main()
    mainMeth.invoke(null, new Object[] { args} );   // call main()
  }
  catch(Exception ex) {
    if (ex instanceof InvocationTargetException) {
      Throwable t =
          ((InvocationTargetException) ex).getTargetException();
      reportError(t.toString());
    }
    else
      reportError(ex.toString());
  }
}  // end of execCode()
```

There are two differences between execCode() and StringExecutor: one major, the other minor. The major one is that it’s somewhat tricky for execCode() to obtain the class name required by ClassLoader.getClass(). The minor difference is that the call to main() in execCode() takes no arguments.

The class name is found using regular expressions (REs) to search for the identifier following the “public” and “class” tokens in the code string. The REs are defined and utilized in getClassName():

```
private String getClassName(String codeStr)
// return the identifier that appears after "public class"
{
  String className = null;

  // prepare regular expression
  String idPatternStr = "[a-zA-Z_$][\\w$]*";
     // RE string for a Java ID

  String codePatternStr = "public\\s*class\\s*(" + idPatternStr+ ")";
     // RE string for "public class (ID)" ; (...) is for grouping

  Pattern codePattern = Pattern.compile(codePatternStr);

  Matcher codeMatcher = codePattern.matcher(codeStr);  // search code

  if (codeMatcher.find() && (codeMatcher.groupCount() == 1))
    // found a single ID, which should be the class name
    className = codeMatcher.group(1);
  else
    reportError("Could not find class name in code");

  return className;
}  // end of getClassName()
```

The identifier is extracted using a RE capturing group, which is specified with parentheses in the codePatternStr RE. Capturing group 0 is the text matching the entire pattern, but I employ group 1 which stores the Java identifier found in the text.

3. Adding Drag and Drop Functionality

The application works, but only from the command line:

```
java -cp "janino.jar;." ExecutePixels Comparison.png
```

There's a few aspects of this user interface that I want to improve:

- ExecutePixels should be represented by a single icon, with 'confusing' features hidden away, such as the need to call java.exe and supply a classpath;
- a user should be able to execute an image by simply dragging and dropping it over the top of the ExecutePixels icon;
- the ExecutePixels icon should be a single, stand-alone executable, which can be moved to another machine and 'just work'.

To reduce my work in this regard, I'll restrict myself to being able to run ExecutePixels on different Windows machines.

A common misconception of most non-Java programmers is that these kinds of packaging requirements are impossible, or very complicated, in Java. In fact, there's a wealth of different techniques, and (free) software that can help.

I'll start by using a JAR file to hold the various parts of the ExecutePixels application, and utilize One-Jar to support JAR-inside-a-JAR functionality. Then I'll look at DOS batch files and VBScript programs to add drag-and-drop capabilities. Finally, I'll employ launch4j to produce a native executable version of ExecutePixels.

At the end, I'll discuss a few other approaches, such as Java Web Start and installer creation software, and point to sources of more information.

3.1. Using JARs

Classes and resources (e.g. images, sounds) that make up a Java application can be conveniently packaged together inside a JAR file; even a splash screen can be included.

The manifest for the ExecutePixels JAR (stored inside manifest.txt) is:

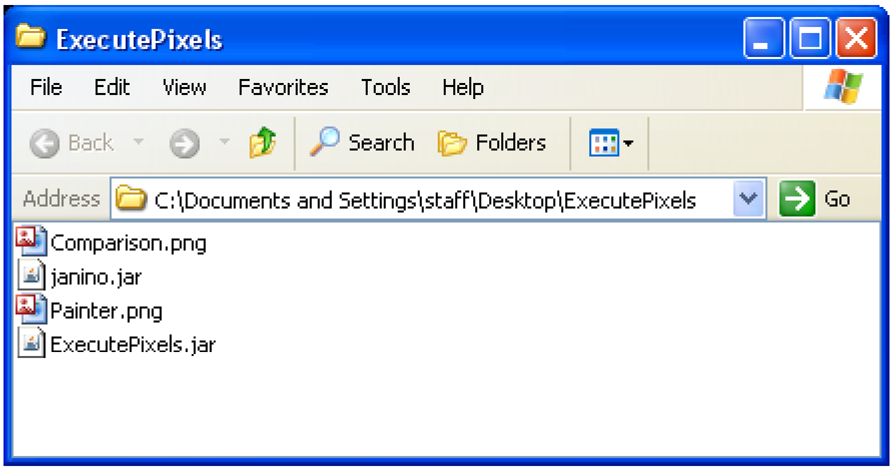
```
Main-class: ExecutePixels
Class-Path: janino.jar
SplashScreen-Image: epSplash.png
```

It includes the class path to janino.jar, which is located in the same directory as the resulting ExecutePixels.jar.

The code is then compiled and rolled into the JAR:

```
> javac -cp "janino.jar;." *.java
> jar cvmf manifest.txt ExecutePixels.jar *.class epSplash.png
```

The directory now holds two JAR files: ExecutePixels.jar and janino.jar (and two test PNG files), as shown in Figure 10.



Comparison.png
janino.jar
Painter.png
ExecutePixels.jar

Figure 10. ExecutePixels as a JAR file.

A PNG image is executed like so:

```
>  java -jar ExecutePixels.jar Painter.png
```

The two JAR files can be placed inside a single JAR by using P. Simon Tuffs' One-JAR (available at http://one-jar.sourceforge.net/). One-JAR is essentially a customized class loader which can deal with JARs and Windows Native Libraries (DLLs) stored inside a JAR.

The One-JAR site includes good documentation and examples. Also, a technical article on One-JAR's class loading techniques, called "Simplify your Application Delivery with One-JAR", can be found at http://www.ibm.com/developerworks/library/j-onejar/index.html?ca=drs-j4904

I downloaded the basic One-JAR tool, one-jar-boot-0.96.jar, from http://one-jar.sourceforge.net/. After unzipping it, a directory called OneJar is created. I added two directories: lib/ to hold janino.jar, and main/ for ExecutePixels.jar.

I also modified the One-JAR manifest file (called boot-manifest.mf) to refer to the top-level class in ExecutePixels.jar, and specified a splash screen. The resulting file is:

```
Manifest-Version: 1.0
Main-Class: com.simontuffs.onejar.Boot
One-Jar-Main-Class: ExecutePixels
SplashScreen-Image: epSplash.png
```

The modified OneJar directory structure is shown in Figure 11.

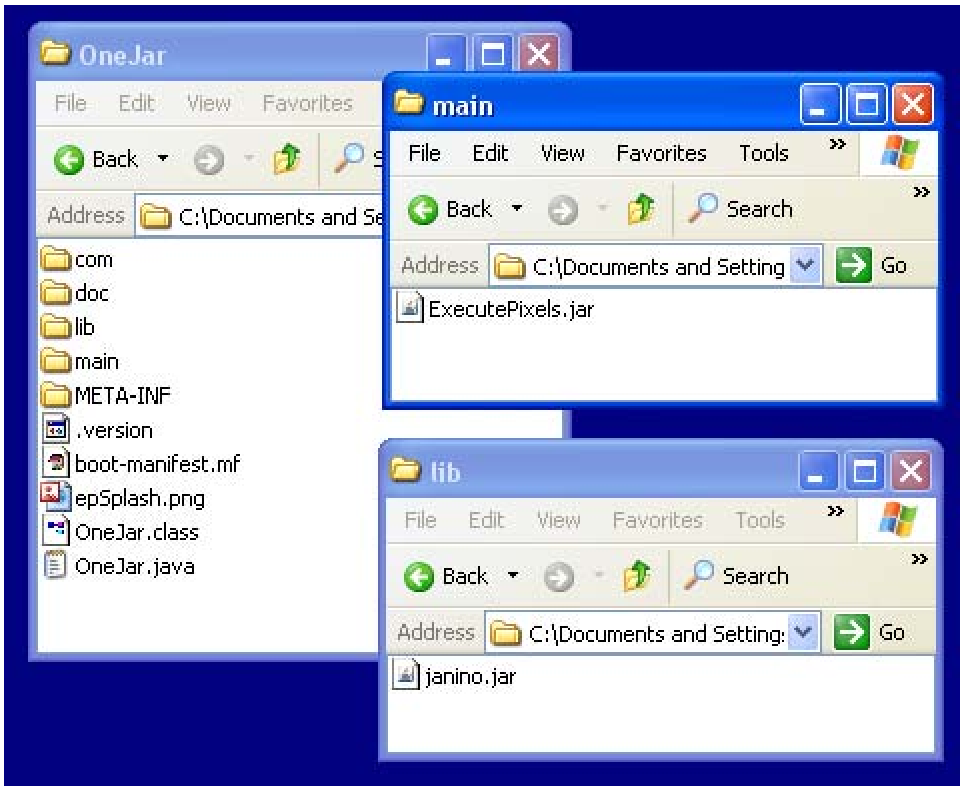


Figure 11. The OneJar Directory Structure.

The creation of the One-JAR version of ExecutePixels.jar is achieved with:

```
> jar cvfm ExecutePixels.jar boot-manifest.mf *
```

This JAR is executed in the normal manner:

```
> java -jar ExecutePixels.jar Painter.png
```

The difference lies inside the new ExecutePixels.jar, which contains janino.jar as part of its resources. The application now consists of only one JAR file.

The JAR mechanism, and the One-JAR tool, can hide the multitude of files and other resources that make up an application. However, JAR files do not support drag-and-drop functionality. This can only be remedied by turning to OS-specific solutions.

3.2. Scripting and ExecutePixels

MS Windows supports a wide range of scripting possibilities, but I'll only look at two: DOS Batch files and VBScript programs, both of which support drag-and-drop capabilities.

My plan is to drop a PNG file onto a script (batch or VBScript), and have the script execute the ExecutePixels JAR with the image file name as its parameter.

The batch file solution (ExecutePixels.bat) is short, but somewhat cryptic:

```
@echo off
cd /d %~dp0
```

```
start /b javaw -jar "%CD%"\ExecutePixels.jar %1
```

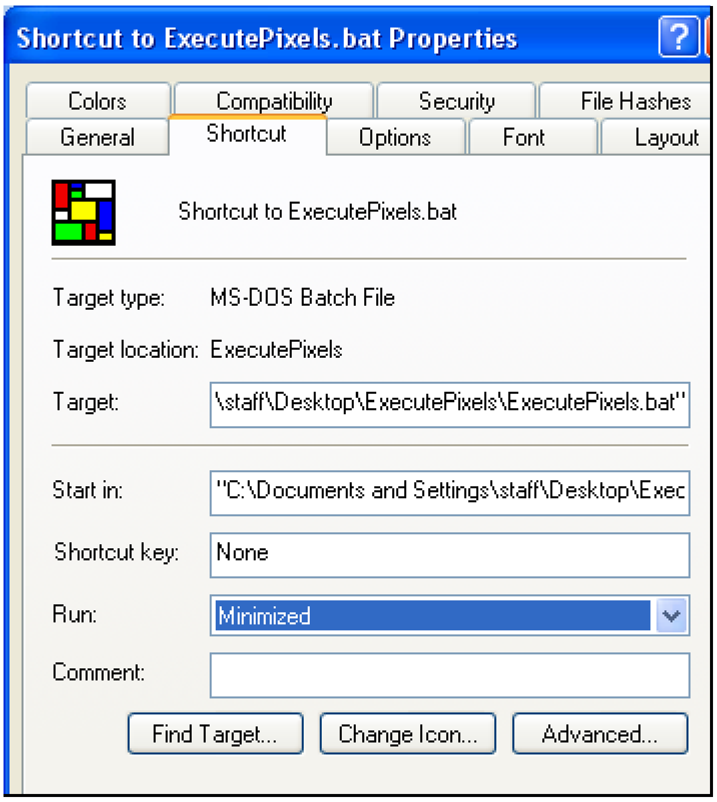
The cd command uses the batch file's name (%0) and the modifiers ~dp to change directory to the drive and path where the batch file is located. This folder also contains the ExecutePixels JAR, so I can use %CD% when I subsequently invoke the JAR.

Some batch file scripters may point out that I could use %CD% without the initial cd. However, %CD% expands to the drive letter and path of the *current* directory, which can be changed by the user before executing the batch file. I override any such changes by calling cd at the start of the file.

Of course this approach may still fail if the JAR file has been moved to a different folder without being accompanied by the batch file. This highlights the problem of using even just two files to implement an application.

Back in the batch file, start /b starts its command argument without creating a console window. Nevertheless, it's still necessary to use javaw.exe rather than java.exe to stop the JVM from creating its own command window.

Even with start and javaw, a DOS window still momentarily appears when the batch file is invoked. This can be 'fixed' by creating a shortcut for the batch file, which can have its properties changed. In particular, it's possible to start a shortcut in minimized form (i.e. only visible as an item in the taskbar), and its name and icon can also be changed (see Figure 12).



Shortcut to ExecutePixels.bat

Target type: MS-DOS Batch File

Target location: ExecutePixels

Target: \staff\Desktop\ExecutePixels\ExecutePixels.bat"

Start in: "C:\Documents and Settings\staff\Desktop\Exec

Shortcut key: None

Run: Minimized

Comment:

Figure 12. Altering the Properties of the Batch File Shortcut.

The resulting shortcut appears as in Figure 13.

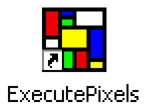


Figure 13. The Modified ExecutePixels Batch File Shortcut.

The drawback of shortcuts are that the “Target” and “Start in” directories are hardwired (as can be seen in Figure 12). So, if the original batch file is moved then the shortcut will stop working. Similarly, the choice of icon has a fixed path, so the shortcut can lose its pretty picture if the icon file is moved.

WSH and VBScript to the Rescue

There’s a slight air of “yesterday’s man” about batch files, which is a bit unfair. But it is true that Windows offers a range of more modern scripting solutions, based around Windows Scripting Host (WSH).

WSH provides an environment where users can execute scripts in a variety of languages, including VBScript, JScript, Perl, Rexx, and Python. Also, WSH supports COM (Component Object Model) objects, which are a standard way for applications (e.g. Word, Photoshop) or individual DLLs to present their capabilities as objects.

WSH can utilize objects and methods from WMI (Windows Management Instrumentation) and ADSI (Active Directory Service Interface) which makes it extremely easy for a script to interact with the Windows OS. For instance, WMI can retrieve service properties, start and stop services, and configure service settings.

WSH can also call existing command line tools, thereby allowing it to replicate old-style DOS functionality.

A good book on WSH and VBScript is “Windows: Scripting Secrets” by Tobias Weltner, Wiley, 2000.

Even WSH is thought of as being rather old-hat by some, since the introduction of Windows PowerShell (also known as Monad, Microsoft Shell, and MSH at different times). PowerShell is a combination command line shell and scripting language, based around 'cmdlets' (specialized .NET classes) and UNIX-style pipes to tie cmdlets and other commands together.

PowerShell 2.0 is currently under development, and will be part of Windows 7.

I’ve no need for the big guns of PowerShell because I only want to replicate the functionality of my simple batch file (i.e. support drag-and-drop and call ExecutePixels.jar), *and* avoid its irritating flicker at invocation time. VBScript has more than enough firepower for that.

Calling ExecutePixels with VBScript

The following VBScript program (ExecutePixels.vbs) mimics the actions of the batch file (somewhat verbosely).

```
' read script argument
set args = WScript.Arguments
if args.Count = 0 then
  MsgBox "No file supplied", vbCritical, "ExecutePixels"
  WScript.Quit
else
  fnm = args(0)   ' treat argument as a filename
end if


' get the folder where this script is stored
thisName = WScript.ScriptFullName
thisFolderPath = left(thisName, InstrRev(thisName, "\"))


' execute executePixels.jar on the supplied filename
set WshShell = CreateObject("WScript.Shell")
WshShell.run "cmd /k java -jar " & _
      chr(34) & thisFolderPath & "\" & chr(34) & _
      "ExecutePixels.jar " & chr(34) & fnm & chr(34), 0
         ' put double quotes round path and filename
set WshShell = Nothing
```

The first few lines check that the script was called with a file argument, which occurs when a file is dropped onto it. If the user double-clicks on the script without offering a file, then the program quits.

The path to the script (and ExecutePixels) is extracted from Wscript.ScriptFullName by employing substring matching on the last "\".

The CreateObject() call returns a reference to a WScript.Shell object which provides access to the OS, including to environment variables, the registry, and shortcuts. I use the WScript.Shell run method to start the ExecutePixels JAR:

```
  cmd /k java –jar "<full path>\"ExecutePixels.jar "<file name>"
```

cmd /k executes the command following it and its interpreter window remains in existence until the call has finished. Unlike in the batch file, I don't need to use start /b and javaw to suppress the console window; instead I utilize a "0" argument at the end of the WScript.Shell run call.

ExecutePixels.vbs replaces ExecutePixels.bat in the directory holding ExecutePixels.jar. The user executes an image by dropping it onto ExecutePixels.vbs, in the same way as the batch file, but without a console window appearing fleetingly as the script starts.

As with batch files, it's possible to create a shortcut to the VBScript file, thereby allowing it to be allocated a different name and icon. However, the problems with hardwired "Target", "Starts In" fields, and other parameters, are still a concern.

3.3. Using an Executable Wrapper

Scripting solutions work, but are susceptible to breaking if the script gets separated from the JAR file. What I really want is a *single* ExecutePixels executable, which supports drag-and-drop. Also, it would be nice to assign the executable a simple name and a pleasing icon.

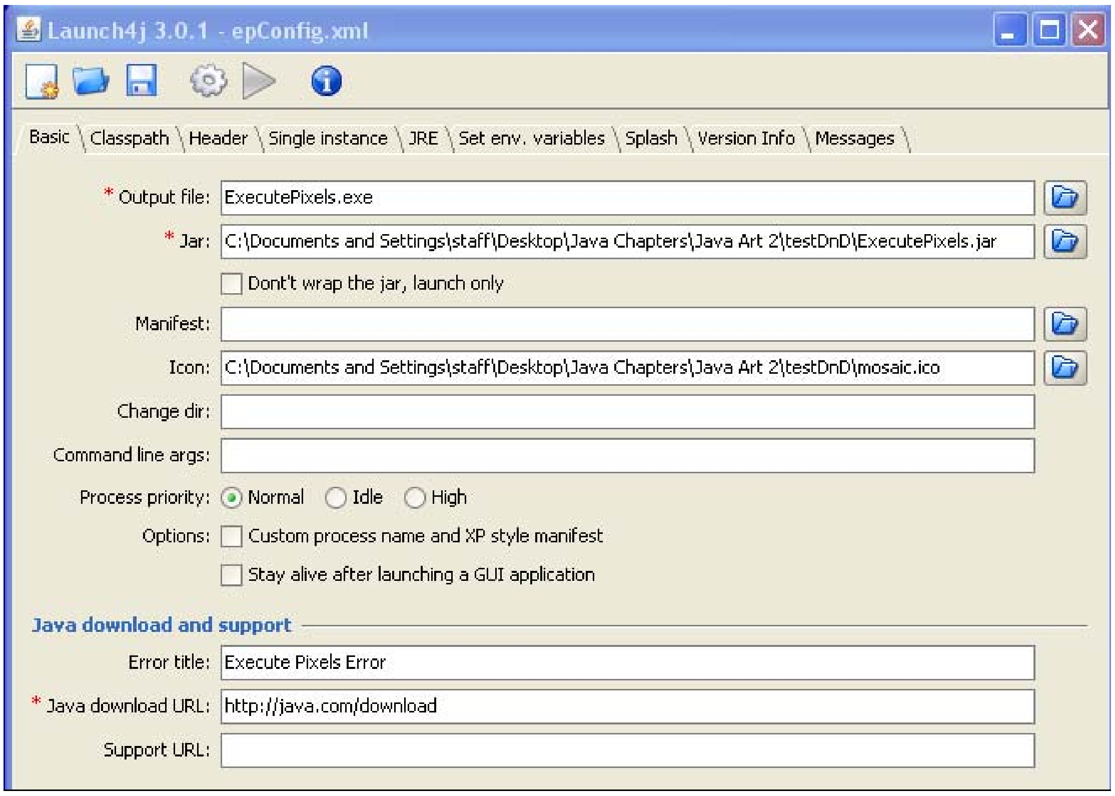
The solution is to employ an executable wrapper builder – software that can wrap a JAR file in a small layer of native code to turn it into an executable. This is such a common requirement of Java programmers that there's a wide choice of freeware and commercial solutions available. I chose launch4j (http://launch4j.sourceforge.net/) because it's free, offers a range of features via a simple GUI interface, is small (about 30 KB), and comes with examples and documentation.

launch4j's emphasis is on wrapping Java code to turn it into a Windows program. The original code can consist of multiple class files, JARs, and DLLs, and launch4j's output can be fine-tuned depending on if the Java source employs a GUI or the console. launch4j doesn't offer wrappers to create Linux or Mac OS X applications, although you can run launch4j on those platforms to create Windows program.

The executable can be configured to utilize a specified JRE version on the local machine, or use a bundled copy, or download one. It's possible to set runtime options, like the initial/max heap sizes, and modify Windows environment variables. The wrapper can include an application icon and a native splash screen which starts without waiting for the JRE to boot up.

I downloaded and installed the Windows version of launch4j, launch4j-3.0.1-win32.exe. When it starts, it offers a tabbed series of panes for setting up a configuration file for the Windows executable.

Figure 14 show a fragment of the first 'Basic' tabbed pane, which contains almost everything I need to specify the ExecutePixels executable.



* Output file:	ExecutePixels.exe
* Jar:	C:\Documents and Settings\staff\Desktop\Java Chapters\Java Art 2\testDnD\ExecutePixels.jar
	☐ Don't wrap the jar, launch only
Manifest:	
Icon:	C:\Documents and Settings\staff\Desktop\Java Chapters\Java Art 2\testDnD\mosaic.ico
Change dir:	
Command line args:	
Process priority:	[✓ Normal] Idle High
Options:	☐ Custom process name and XP style manifest
	☐ Stay alive after launching a GUI application
Error title:	Execute Pixels Error
* Java download URL:	http://java.com/download
Support URL:	

Figure 14. launch4j's Basic Configuration Pane.

The required information includes a name for the executable, the JAR that's to be wrapped, and an on-screen icon (an ICO file). On the other panes, I set a minimum JRE version and supplied a splash screen image.

A click on the 'cog wheel' icon in the menu bar generates ExecutePixels.exe. Its on-screen icon is shown in Figure 15.

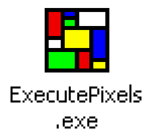


Figure 15. The ExecutePixels Executable.

I obtained the icon from a free site, http://www.iconarchive.com, but there are plenty of tools which can convert PNG images into ICO files (e.g. IcoFx from http://icofx.ro/).

The advantage of a single executable is that it can't be separated from other parts of the application, and can be moved to any machine and still work (assuming there's a suitable JRE present). Since it's a native executable, it is on equal terms with other Windows applications.

4. Time to Stop?

At last, I can rest easy since I finally have a program that does everything I want:

- ExecutePixels is represented by a single icon, with 'confusing' features hidden away, such as the java.exe call and the classpath;
- a user can execute a PNG file by simply dragging and dropping it over the top of the ExecutePixels icon;
- the ExecutePixels icon is a single, stand-alone executable, which can be moved to another machine and 'just work'.

There is perhaps one thing missing from ExecutePixels: a simple installer. I'll briefly make a few suggestions about how to create one in the next section.

4.1. The Next Step: Creating an Installer

There's plenty of demand for installer functionality tuned to Java, which happily means there are numerous available freeware and commercial installer-builders.

izPack (http://izpack.org/) is a popular freeware solution which can generate installers for Windows, Mac OS X, Solaris, Linux, and BSD. On Windows, the installer can set up shortcuts and interact with the registry. IzPack-created installers require a JVM on the host machine.

The generated installer presents a familiar series of panels leading the user through the installation choices. izPack comes with a large set of predefined panels, and custom ones can be added.

Although izPack generates GUI-based installers, izPack itself is a command-line utility, which requires the developer to grapple with a XML file. Fortunately, there's a GUI front-end for izPack called PackJacket (http://packjacket.sourceforge.net/) which hides most of the XML.

On the commercial side, I've used a trial version of install4j (http://www.ej-technologies.com/products/install4j/overview.html) with some success. Details can be found online at http://fivedots.coe.psu.ac.th/~ad/jg/app1/.

Of course, there's also Java Web Start (JWS), Sun's very own installer for Web-based Java applications (http://java.sun.com/javase/technologies/desktop/javawebstart/). Typically, the user points a browser at a page containing a link to a deployment file; the retrieval of that file triggers the execution of JWS on the client, which takes over from the browser. (This assumes that JWS is already present of the machine.)

JWS uses the information in the deployment file to download the various JAR files making up the application, together with installation icons, a splash screen, and other details. The application is stored in a local cache, and executed inside a JVM sandbox. Subsequent executions of the application utilize the cached copy, unless the original has been modified, in which case the changed JARs are downloaded again.

Details of my experiences with JWS can be found online at http://fivedots.coe.psu.ac.th/~ad/jg/app2/.

4.2. More Information on Native Executables

"Java to EXE - Why, When, When Not, and How" by Dmitry Leskov (http://www.javalobby.org/articles/java2exe/) is a good survey from 2005 of the various ways of turning a Java application into a native executable.

He examines the pros and cons of executable JARs, Java Web Start, custom Java launchers and wrappers, and ahead-of-time compilers, and includes links to lots of tools and resources. Almost as valuable are the Javalobby forum posts that his article provoked, which can be read at http://www.javalobby.org/java/forums/t19231.html.

Back in 2003-2004, Joshua Marinacci wrote a three-part article at java.net called "Make Your Swing App Go Native" (http://today.java.net/pub/a/today/2003/12/08/swing.html). In part 2, he looked at using JARs, shell scripts, and native executables (he suggested a trial version of JexePack, http://www.duckware.com/jexepack/).

In 2006, Kustaa Nyholm published "Java on Desktop Goes Native" at his website (http://www.sparetimelabs.com/javagoesnative/). He went with jStub for packaging his Java application as an executable (http://ovanttasks.sourceforge.net/rat/chapter-N10382.html).