11. Applets, normal window applications, packaging and sharing your work

In this chapter

- Converting Full Screen experiments into normal window applications,
- Packaging and sharing applications
  - packaging in a self running jar archive
  - deploying as Java applets
  - deploying as Java Web Start application

Upto now all our examples were in Full Screen Exclusive mode. This, of course, was the appropriate choice for a psychophysics experiment, nevertheless one may need to test or implement a version of the actual psychophysics experiment in a normal window. This is particularly useful to convert the experiment into an applet and place in a web page. The first objective of this chapter is to show how to create normal window applications using core Java tools. I will do this by creating a new class called NormalWindow. With this new class, converting the FullScreen applications into normal window applications will be just a matter of changing a few lines in the experimental code.

The other objective of the current chapter is to introduce ways to share your work with colleagues and others. (Java platform is particularly attractive because it makes sharing possible. Ability to share the work enhances communication, boosts the productivity.) One mean of sharing or demonstrating your work is turning the application into a Java Applet and place it on a web page. This way visitors of your web page could run an Applet version of your experiments without installing any programs other then the Java plug-ins. Another method to share your work is to pack it in a self running, so called *jar* file. Depending on the OS and your configuration, those files can run by just a mouse click. Last, I will show how to use Java Web Start technology to share your work.

I will start describing the implementation of the NormalWindow class, after that I will show how to convert the HelloPsychophysicist example from Chapter 2 into a normal window application and into Java applets.

11.1. NormalWindow class

As we will see next, there is not so much difference between the FSEM application and a normal window application. Therefor I will present the NormalWindow class by highlighting its differences from the FullScreen class.

11.1.1. Constructor

First of all, NormalWindow extends JPanel instead of JFrame. *Because* .... The constructor of the NormalWindow class differs from the FullScreen class. We eliminate the method invocations related to FSEM: setUndecorated(true), setIgnoreRepaint(true), setResizable(false), gDevice.setFullScreenWindow(this). We also eliminate the method related to setting the BufferStrategy, setNBuffers(nBuffers) totally. Here is the constructor of NormalWindow

```
public class NormalWindow extends JPanel{
  public NormalWindow(){

    super();

    setFont(defaultFont);
    setForeground(fgColor);
    setBackground(bgColor);
    setFocusable(true);

    // ...
  }
```

11.1.2. storing the entire screen in a BufferedImage

We store the visible screen in a BufferedImage called screenImage. In displayImage(), displayText() and blankScreen() methods we manipulate the screenImage, in updateScreen we actually show it on the screen. Notice the similarity and distinction to the FSEM version. In FSEM version we were drawing the images or texts on the video buffer by using its Graphics2D contents. Here we use the Graphics2D contents of the screenImage. In FSEM, the show() method of BufferStrategy class was responsible of actually displaying whatever changes we had done so far, here we will manually update the screen using the screenImage.

11.1.3. Double Buffering and active/passive rendering in NormalWindow

As we eliminated the setNBuffers() method, we are now on our own to determine various double buffering strategies, and since we eliminated the setIgnoreRepaint(true) in the constructor, we should manually decide our active/passive rendering choices.

As I mentioned before, in active rendering you are responsible with all renderings, JVM does not intervene. This allows you to precisely adjust your stimulation. This suits well in a FSEM application. But in a Normal Window application there are good reasons why JVM should sometimes intervene: for instance, when you close your window, or your window is occluded by other applications, and later brought to front again you have to update the content. In passive rendering this is automatically done by the JVM. Active rendering was suitable for the FullScreen applications, because no other window could occlude our application window, or any other OS related menus or bars could actually appear. So we did not need JVM's intervention to repaint the screen. It was all up to us. But in a normal window mode such occlusions may happen. Therefore it is more suitable to use passive rendering in normal window mode. By default we will adopt the passive rendering strategy in NormalWindow class. We define a boolean parameter passiveRendering and set it to true

```
private boolean passiveRendering = true;
```

But if you decide that you don't want the JVM update the screen passively, you can set passiveRendering to false with setPassiveRendering() method

```
public void setPassiveRendering(boolean pr){

  passiveRendering = pr;
  setIgnoreRepaint(!passiveRendering);
}
```

This method invokes the setIgnoreRepaint method, thereby instructing the JVM to passively update or not the application's visible surface. isPassiveRendering() method returns whether the strategy is set to passive rendering or not

```
public boolean isPassiveRendering(){

  return passiveRendering;
}
```

How does passive rendering work? A JPanel has a method called paintComponent(), a class inheriting from JPanel should override that method and provide the rules to paint the stimulus in it. JVM uses this method to repaint the application window in case of passive rendering. However you never invoke paintComponent() directly. You invoke another method called repaint() and it indirectly takes care of invoking the paintComponent() method.

Here is the new updateScreen() method. If the passiveRendering variable is set to true, it invokes the repaint() method, if not it actively paints the screen

```
public void updateScreen(){

  if(screenImage == null)
    createScreenImage();

  if(passiveRendering)
    repaint();
  else {
    Graphics g = getGraphics();
    try {
      if(g!=null)
        g.drawImage(screenImage, 0,0, this);
      Toolkit.getDefaultToolkit().sync();
    }
    finally {
      g.dispose();
    }
  }
}
```

First we check wheter or not screenImage is null. Next, if we are in passive rendering mode we just invoke the repaint method. But if we are in active rendering we do the following: we first get the Graphics contents of the component, here NormalWindow, then we draw the screenImage on this component, and finally dispose the Graphics2D.

Next let's look at the paintComponent() method of NormalWindow. JVM invokes paintComponent() method under two conditions: One, if the application invokes repaint() method; Two, if the window needs to be updated for some reason, such as minimization/maximization. Normally, if you are using passive rendering strategy, what you have to place in your paintComponent() is invoking the JPanel's paintComponent() method with super.paintComponent(), then place your application specific lines. What if you are doing active rendering? Although we eliminated the repaint() invocation in updateScreen() above, JVM is still going to automatically invoke paintComponent() method if there is a need to re-draw the window. You have to make sure that JVM knows that you are in active rendering mode and it should not update the screen automatically. Here is our paintComponent() implementation:

```
protected void paintComponent(Graphics g){

  if(passiveRendering){
    super.paintComponent(g);
    if(screenImage != null)
      g.drawImage(screenImage,0,0,this);
  }
}
```

On the other hand, even though it is not FSEM, we still want to use double buffering. This is not too difficult, because we had already set up the FullScreen class in a way that makes it easy to accomplish this. Recall that we first draw the stimulus on an off-screen buffer and then display it on the screen after the rendering is completed, we will keep that same strategy here in NormalWindow:

```
public void displayImage(int x, int y, BufferedImage bi) {
   if(screenImage == null)
      createScreenImage();

    Graphics2D g = screenImage.createGraphics();
    try {
      if(g!=null && bi!=null)
        g.drawImage(bi, x, y, null);
      Toolkit.getDefaultToolkit().sync();
    }
    finally {
      g.dispose();
    }
  }
```

then, just as in the FSEM to actually display the stimulus the application should invoke the updateScreen() method.

Here is the complete NormalWindow.java code

```
/*
 * chapter 10: HPWindow.java
 *
 * displays the text "Hello Psychophysicist (Normal Window)" and two images
 * in a normal window
 *
```

```
 */
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import java.awt.image.BufferedImage;
import java.io.IOException;
// HPWindow extends NormalWindow, instead of FullScreen
public class HPWindow extends NormalWindow implements Runnable {
  static JFrame mainFrame;

  public static void main(String[] args) {
    HPWindow nw = new HPWindow();

    // The only addition/change is here:

    //nw.setPassiveRendering(false);
    mainFrame = new JFrame();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.add(nw);
    mainFrame.setBounds(100, 100, 612, 612);
    mainFrame.setResizable(false);
    mainFrame.setVisible(true);
    // up to here

    Thread hpnw = new Thread(nw);
    hpnw.start();
  }

  public void run(){
    try {
      displayText("Hello Psychophysicist (Normal Window)");
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(
          HPWindow.class.getResource("fechner.png"));
      displayImage(bi2);
      updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
```

```
    } catch (InterruptedException e) {}
    finally {
      // and replace this
      //fs.closeScreen();
      mainFrame.dispose();
    }
  }
}
```

11.2. HelloPsychophysicist, normal window

Here is the HelloPsychophysicist.java example from Chapter 2, modified to act as a normal window application, the changes are marked with **bold font**

```
/*
 * chapter 10: HPWindow.java
 *
 * displays the text "Hello Psychophysicist (Normal Window)" and two images
 * in a normal window
 *
 */
import javax.imageio.ImageIO;
import javax.swing.JFrame;
import java.awt.image.BufferedImage;
import java.io.IOException;
// HPWindow extends NormalWindow, instead of FullScreen
public class HPWindow extends NormalWindow implements Runnable {
  static JFrame mainFrame;

  public static void main(String[] args) {
    HPWindow nw = new HPWindow();
    //nw.setPassiveRendering(false);

    // The only addition/change is from here:
    mainFrame = new JFrame();
    mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    mainFrame.add(nw);
    mainFrame.setBounds(100, 100, 612, 612);
    mainFrame.setResizable(false);
    mainFrame.setVisible(true);
    // up to here

    Thread hpnw = new Thread(nw);
    hpnw.start();
  }

  public void run(){
```

```
        try {
          displayText("Hello Psychophysicist (Normal Window)");
          updateScreen();
          Thread.sleep(2000);
          blankScreen();
          hideCursor();
          BufferedImage bi1 = ImageIO.read(
              HPWindow.class.getResource("psychophysik.png"));
          displayImage(bi1);
          updateScreen();
          Thread.sleep(2000);
          blankScreen();
          BufferedImage bi2 = ImageIO.read(
              HPWindow.class.getResource("fechner.png"));
          displayImage(bi2);
          updateScreen();
          Thread.sleep(2000);
        } catch (IOException e) {
          System.err.println("File not found");
          e.printStackTrace();
        } catch (InterruptedException e) {}
        finally {
          // and replace this
          //fs.closeScreen();
          mainFrame.dispose();
        }
      }
    }
```

When compiled and executed, this program opens a normal window and displays the text "Hello Psychophysicist (Normal Window)" and two other images. NormalWindow extends JPanel. But JPanels can't be displayed on the screen directly, it has to be put inside a, so called *heavier* component. Here I put it inside a JFrame:

```
        mainFrame = new JFrame();
        mainFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        mainFrame.add(nw);
        mainFrame.setBounds(100, 100, 612, 612);
        mainFrame.setResizable(false);
        mainFrame.setVisible(true);
```

setDefaultCloseOperation() method tells the JFrame to exit the program in case the user closes the window. In the next line we add the NormalWindow object nw to the JFrame. Then we set the position and size of the JFrame with setBounds(100, 100, 612, 612) method. It will be placed at (100,100) relative to upper left corner of the screen, with width and height equal to 612. We also invoke setResizable(false) so that the user can not resize the window. We make the JFrame visible by invoking setVisible(true) method. In the end, inside finally{}, instead of closeScreen() method of the FullScreen, we use the dispose() method of JFrame

```
        finally {
```

```
        // and replace this
        //fs.closeScreen();
        mainFrame.dispose();
      }
```

Another subtle difference is the way we load the images

```
BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
```

We have to this in order to get the self running jar files use the resources properly. In this case the resource file is an image, but this consideration extends to other types of files, as well. I will show how to use other types of resource files below in Section 11.7.

11.3. Java Applets

Applets don't have main() methods. Instead you should provide init() method for initialization of the applet. This method is invoked by the browser or appletviewer to inform the applet that it is loaded. The other relevant methods are start() and stop(). start() is called after the init() method and everytime the web page is revisited. Initializations, such as creation of new Threads can be done inside the init() method. start() and stop() can be used, for example, pause and resume a Threaded animation. I will provide two different versions of HelloPsychophysicist as Java applets below.

11.4. HelloPsychophysicist, as an applet

```
/*
 * chapter 10: HPApplet.java
 *
 * displays the text "Hello Psychophysicist (Applet)" and two images on an
 * otherwise entirely blank window
 *
 */
// <applet code="HPApplet.class" width=612 height=612>
// </applet>
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.image.BufferedImage;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JPanel;
public class HPApplet extends JApplet {
  private AnimationPanel animationPanel;
  public void init() {
    animationPanel = new AnimationPanel();
    animationPanel.setPassiveRendering(true);
```

```
    animationPanel.initAnimation();
    add(animationPanel, "Center");

    final JButton playButton = new JButton("Play Again");
    JPanel buttonPanel = new JPanel();
    buttonPanel.add(playButton, "Center");
    add(buttonPanel, "South");
    setSize(612, 612);

    playButton.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        animationPanel.startAnimation();
      }
    });

  }
  public void start() {
    animationPanel.startAnimation();
  }

  public void stop() {

    animationPanel.stopAnimation();
  }

}
class AnimationPanel extends NormalWindow implements Runnable {
  private Thread animator;
  private boolean running = false;
  public void initAnimation() {
    if (animator == null || !animator.isAlive())
      animator = new Thread(this);
  }
  public void startAnimation() {
    if (!animator.isAlive())
      animator = new Thread(this);

    if (!running) {
      running = true;
      animator.start();
    }

  }

  public void stopAnimation(){

    running = false;
  }
```

```
  public void run() {
    try {
      blankScreen();
      displayText("Hello Psychophysicist (Applet)");
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      hideCursor();
      BufferedImage bi1 = ImageIO.read(HPApplet.class
          .getResource("psychophysik.png"));
      displayImage(bi1);
      updateScreen();
      Thread.sleep(2000);
      blankScreen();
      BufferedImage bi2 = ImageIO.read(HPApplet.class
          .getResource("fechner.png"));
      displayImage(bi2);
      updateScreen();
      Thread.sleep(2000);
    } catch (IOException e) {
      System.err.println("File not found");
      e.printStackTrace();
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    finally {
      running = false;
    }
  }
}
```

Different from the FSEM and Normal Window versions, I included a button to replay the stimulus sequence

```
final JButton playButton = new JButton("Play Again");
```

For the JButton to be useful for anything, I add an action listener using the anonymous inner class technique (see Chapter XXX)

```
playButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    animationPanel.startAnimation();
  }
});
```

When the observer presses the “Play Again” button, the applet invokes the startAnimation() method of AnimationPanel class. We add the JButton first in a JPanel of itself and then add that JPanel to the Applet

```
JPanel buttonPanel = new JPanel();
```

```
buttonPanel.add(playButton, "Center");
add(buttonPanel, "South");
```

run() method of AnimationPanel class is the same as the former versions of HelloPsycophysicist. But this time I included 3 new methods initAnimation(), startAnimation(), stopAnimation(), to initialize, start and stop the Animation thread respectively. When HPApplet is first loaded by the browser (or by applet viewer) its init() method is invoked. Inside init() we initialize the AnimationPanel by invoking its initAnimation() method. Next, the start() method of the HPApplet is invoked. Note that you don't need to invoke this method, nor the init() and stop() methods, yourself. Their invocation is taken care of automatically. The start() method is invoked by the browser everytime you visit the web page holding the applet. In the start() method we invoke the startAnimation() method of the AnimationPanel to start the animation. So the animation starts as soon as the browser loads the page. stop() method of the HPApplet is invoked when you leave that page. In that method we invoke the stopAnimation() method of the AnimationPanel.

11.5. HelloPsychophysicist, a normal window as a pop-up in applet

Here I will show another way to deploy normal window applications. This is a combination of an Applet and a Normal Window. The Applet part displays a "Press to set in/visible" button and allows the user to set the normal window visible or invisible. The Normal Window part is the same as the AnimationPanel we used in Section 11.4, in fact we use it without any change. We don't need to re-write it in our code because it is in the same directory and accessible by the other java programs.

```
/*
 * chapter 10: HPApplet2.java
 *
 * By pressing a JButton
 * opens a JFrame and displays the text "Hello Psychophysicist (Applet)"
 * and two images on an otherwise entirely blank screen
 *
 */
// <applet code="HPApplet2.class" width=250 height=50>
// </applet>
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JApplet;
import javax.swing.JButton;
import javax.swing.JFrame;
public class HPApplet2 extends JApplet{
  public void init() {
    final AnimationPanel animationPanel = new AnimationPanel();
    animationPanel.initAnimation();

    final JFrame fs = new JFrame();
    fs.setTitle("Hello Psychophysicist");
    fs.setSize(612,612);
    fs.add(animationPanel);

    JButton playButton = new JButton("Press to set in/visible");
```

```
    add(playButton);
    setSize(250,50);

    playButton.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
        fs.setVisible(!fs.isVisible());
        if(fs.isVisible())
          animationPanel.startAnimation();
      }
    });
  }
}
```

11.6. Deploying applets

Applets can be deployed in various ways. If you are using Eclipse, you can run the HApplet.java by right-clicking on the HApplet.java file and choosing Run As →Java Applet. Or you can deploy it from a terminal by typing

```
appletviewer HApplet.java
```

and pressing enter. The commented lines

```
// <applet code="HPApplet.class" width=612 height=612>
// </applet>
```

instructs the appletviewer to initiate the HPApplet.class, within a frame with a size of 612 by 612. Below I will show initiating applets from within web pages.

To run applets from a web page, you create an html file like this one:

```
<html>
<body>
  <span style="font-family: sans-serif; font-weight: bold;">
    Sharing your work is easy, here is one possibility using applets:</span>
  <br>
  <br>

  <applet code="HPApplet.class" height="612" width="612">
    </applet>
  </body>
</html>
```

Again, you can run this html file directly from within Eclipse by double-clicking on it. Or you can deploy it with your java-enabled browser. Or you can use appletviewer

```
appletviewer HApplet.html
```

11.6.1. Packaging resources for Applets

If there are a number resource files that your applet uses, then the client's browser will have to download each one separately. For each such request a new connection has to be established. This would cause an overhead. Instead, you can put all the necessary files in a single bundled .jar file. You can do that directly from within Eclipse. Right click on the file(s) and choose Export → jar file (multiple .java files can be chosen from the Package explorer by pressing and holding the ctrl key and then left clicking on each java file). Click next and select the export destination name (for example HPApplet2.jar) under the same directory. Also check the image files psychophysik.png and fechner.png. Eclipse will include the corresponding files in the exported jar file. You can also do the export manually. Open a terminal and type

```
jar -cvf HPApplet2.jar *.class *.png
```

and press enter. In any case, the line in your web page changes slightly:

```
<applet code="HPApplet2.class" archive="HPApplet2.jar" width="250" height="50">
</applet>
```

This html file can similarly be deployed either directly from within Eclipse or by using a web browser, or by using appletviewer.

When packaging resources you must pay attention to how your application loads those resources. For instance, as we saw before, to load images you must use something like this:

```
BufferedImage bi1 = ImageIO.read(
          HPWindow.class.getResource("psychophysik.png"));
```

For opening files to read, we used to use a Scanner object like this

```
Scanner in = new Scanner(new BufferedReader(new FileReader(filename)));
```

you can still use a Scanner object but with a slightly different construction

```
Scanner in = new Scanner(HPWindow.class.getResourceAsStream(filename));
```

11.7. Preparing self running .jar files

An effective way of sharing your applications with your collaborators is preparing self-running jar files. To prepare a self-running jar file, first crate a manifest file, here manifest.mf. This file in its simplest form contains only two lines

```
Manifest-Version: 1.0
Main-Class: HPWindow
```

then pack your application in a .jar file using this manifest file

```
jar -cvfm HPWindow.jar manifest.mf HPWindow.class NormalWindow.class *.png
```

and press enter. Eclipse can also create self running jar files. You should still create a manifest file. Then choose the .java file(s) from the Package explorer, and right click and choose export. As in previous section, check the image resources and this time click Next. Click Next again, until you reach the window where you are asked for manifest file. Here you can either use an existing manifest file, or let Eclipse create one for you by providing the name of the class which holds the main class.

Once the self-running jar file is created, you can deploy it either by double-clicking on the file on your hard rive (this may work even if the file is on a web page), or manually from a terminal by typing

```
java -jar HPWindow.jar
```

11.8. Applications deployed with Java Web Start (JWS)

Java Web Start is a newer technology. To deploy applications with JWS, first prepare a self running .jar file as described above (for example HPWindow.jar). Then prepare the HPWindow.jnlp launch file

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp
  spec="1.0+"
  codebase="file:./"
  href="HPWindow.jnlp">
  <information>
    <title>HelloPsychophysicist Window </title>
    <vendor>Huseyin Boyaci</vendor>
    <description>A simple test of JWS
    </description>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version="1.5+"/>
    <jar href="HPWindow.jar"/>
  </resources>
  <application-desc/>
</jnlp>
```

Now you can launch the JWS application either pointing your web browser to the location of this .jnlp file, by double-clicking on the file on your hard drive, or using the javaws program included in JRE. For the later option, open a terminal and type

```
javaws HPWindow.jnlp
```

then press enter. There a few advantages of JWS over an Applet. A Java applet runs in a “sandbox” for security reasons. Therefore, for example, an applet can not touch your hard drive, can not read or write. Also an applet can not open a FSEM application. But JWS can (in principle). Try it using the HelloPsychophysicist program from Chapter 2. *(Failed under FC4?!)*

Here is a .html file which demonstrates all possible ways of sharing the work as we saw above

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
```

```
<head>
  <meta content="text/html; charset=ISO-8859-1"
 http-equiv="content-type">
  <title>Sharing your work</title>
</head>
<body>
  <span style="font-family: sans-serif; font-weight: bold;">
    Sharing your work is easy, here are some possibilities</span>
  <br>
  <br>
  <span style="font-family: sans-serif;">
    1) package your application in a
    .jar file, place a link to the jar file in your web page and double
    click on the link
    <a href="HPWindow.jar">HPWindow.jar</a>
    (if the click doesn't work, your visitors can copy it to local
    disk and double click on it. Or do
    <span style="font-family: courier;">
      java -jar HPWindow.jar
    </span>
    from a terminal)
    <br>
    2) prepare it in an applet like
    this embedded in your web page
    <br>
    <applet code="HPApplet.class" height="612" width="612">
    </applet>
    <br>
    <br>
    3) Or click on this button to start the applet in a new window:
    <br>
    <applet code="HPApplet2.class" archive="HPApplet2.jar"
    height="40" width="250">
    </applet>
    <br>
     4) Prepare it as a Java Web Start Application and tell them to click
     <a href="HPWindow.jnlp">here</a>, or copy the content of that file
     and run from a terminal
     <span style="font-family: courier;">
        javaws HPWindow.jnlp
     </span>
  </body>
</html>
```

11.9. Summary

To convert from FullScreen (FSEM) to normal window:

- Make your class extend NormalWindow instead of FullScreen
- create a JFrame and place your NormalWindow JPanel in it
- Eliminate calls to those methods:
  - setNBuffers(); getNBuffers(); setDisplayMode(); getDisplayMode(); reportDisplayMode();
- Decide whether you want passive rendering or not, set your preference using setPassiveRendering(pr) where pr is a boolean either true or false
- instead of closeScreen(), invoke the dispose() method of JFrame to close your normal window application.

To prepare an applet of your FullScreen application:

- create a JApplet, provide init(), start() and stop() methods
- Create a class extending NormalWindow and place it in the applet
- if it is an animation, supply the necessary method invokations to the NormalWindow from the init() or start() methods of the JPanel