```
 1 import java.awt.*;
 2 import java.awt.event.*;
 3 import javax.swing.*;
 4 import java.awt.image.*;
 5 import javax.swing.border.*;
 6 import java.awt.image.BufferedImage;
 7 import java.io.*;
 8 import javax.swing.filechooser.FileFilter;
 9 import javax.imageio.ImageIO;
10 import java.net.URL;
11
12 /*************************************************************
13 MenuActionListener provides the link between the dropdown menu
14 items and the methods they actually call.
15
16 @author James Houghton
17 @version 06/05/2014
18 *************************************************************/
19 class MenuActionListener implements ActionListener
20 {
21    public void actionPerformed(ActionEvent evt)
22    {
23       if (evt.getActionCommand()=="Invert")
24          Tasks.task1();
25       if (evt.getActionCommand()=="Fade")
26          Tasks.task2();
27       if (evt.getActionCommand()=="Tint")
28          Tasks.task3();
29       if (evt.getActionCommand()=="Black/White")
30          Tasks.task4();
31       if (evt.getActionCommand()=="Remove color")
32          Tasks.task5();
33       if (evt.getActionCommand()=="Grayscale")
34          Tasks.task6();
35       if (evt.getActionCommand()=="Colorize")
36          Tasks.task7();
37       if (evt.getActionCommand()=="Clear effects")
38          Display.clearEffects();
39       if (evt.getActionCommand()=="Encode")
40          Tasks.encode();
41       if(evt.getActionCommand()=="Decode")
42          Tasks.decode();
43       if (evt.getActionCommand()=="Open...")
44          Tasks.open();
45       if (evt.getActionCommand()=="Save")
46          Tasks.save();
47       if (evt.getActionCommand()=="Save as...")
48          Tasks.saveAs();
49       if (evt.getActionCommand()=="Exit")
50          System.exit(0);
51         if (evt.getActionCommand()=="Help")
52          Tasks.help();
53         if (evt.getActionCommand()=="Info")
54          Tasks.info();
55       if (evt.getActionCommand()=="Copy to clipboard")
56          Tasks.copy();
57       if (evt.getActionCommand()=="Clear")
58          Display.clear();
59    }
```

```
 60 }
 61
 62 /**************************************************************
 63 Display contains methods for initializing and opening a GUI to
 64 run CRYPTICON.
 65 MenuActionListener contains the bindings between every button in
 66 the GUI and their respective actions.
 67
 68 @author James Houghton
 69 @version 06/05/2014
 70 **************************************************************/
 71 public class Display
 72 {
 73    /**************************************************************
 74    Main frame of CRYPTICON.
 75    **************************************************************/
 76    public static JFrame pictureFrame;
 77    /**************************************************************
 78    Side panel placed on the left of the main frame.
 79    **************************************************************/
 80    public static JPanel sidePanel;
 81
 82    /**************************************************************
 83    The menu bar of crypticon. Contains the three dropdown menus.
 84    **************************************************************/
 85    public static JMenuBar menuBar;
 86    /**************************************************************
 87    Effects dropdown menu. Contains all effects.
 88    **************************************************************/
 89    public static JMenu effects;
 90    public static JMenuItem effect1;
 91    public static JMenuItem effect2;
 92    public static JMenuItem effect3;
 93    public static JMenuItem effect4;
 94    public static JMenuItem effect5;
 95    public static JMenuItem effect6;
 96    public static JMenuItem effect7;
 97    public static JMenuItem cleareffects;
 98
 99    /**************************************************************
100    Main, image-containing panel.
101    **************************************************************/
102    public static JPanel picturePanel;
103    /**************************************************************
104    The label in which the loaded image is placed.
105    **************************************************************/
106    public static JLabel imageLabel;
107    /**************************************************************
108    The last instance of imageLabel.
109    **************************************************************/
110    public static JLabel prevImageLabel;
111
112    /**************************************************************
113    File dropdown menu. Contains File I/O options.
114    **************************************************************/
115    public static JMenu fileMenu;
116    /**************************************************************
117    Menu item used to open an image.
118    **************************************************************/
```

```
119    public static JMenuItem open;
120    /*************************************************************
121    Menu item used to save an image.
122    *************************************************************/
123    public static JMenuItem save;
124    /*************************************************************
125    Menu item used to save an image with a specific filename and location.
126    *************************************************************/
127    public static JMenuItem saveAs;
128    /*************************************************************
129    Menu item used to exit the program.
130    *************************************************************/
131    public static JMenuItem quit;
132
133    /*************************************************************
134    Dropdown menu containing help, information about the program, and its
135    developers.
136    *************************************************************/
137     public static JMenu resources;
138    /*************************************************************
139    Menu item used to assist the user in operating the program.
140    *************************************************************/
141     public static JMenuItem help;
142    /*************************************************************
143    Menu item used to show information about the program and its developers.
144    *************************************************************/
145     public static JMenuItem info;
146
147    /*************************************************************
148    Input text field. Where text to be encoded is placed.
149    *************************************************************/
150    public static JTextArea inputTextField;
151    /*************************************************************
152    Output text field. Where the decoded message is displayed.
153    *************************************************************/
154    public static JTextArea outputTextField;
155
156    /*************************************************************
157    Button that encodes the image with text.
158    *************************************************************/
159    public static JButton encode;
160    /*************************************************************
161    Button that displays the encoded message in the output text field.
162    *************************************************************/
163    public static JButton decode;
164    /*************************************************************
165    Button that allows the user to copy the output text to the system
166    clipboard.
167    *************************************************************/
168    public static JButton copy;
169    /*************************************************************
170    Button that clears and resets the output text field.
171    *************************************************************/
172    public static JButton clear;
173
174    /*************************************************************
175    Height of the loaded image.
176    *************************************************************/
177    public static int height;
```

```
178     /*************************************************************
179     Width of the loaded image.
180     *************************************************************/
181     public static int width;
182
183     /*************************************************************
184     The loaded image. Typically encoded with text.
185     *************************************************************/
186     public static BufferedImage loadedImage;
187     /*************************************************************
188     The loaded image that is used when effects are applied. Typically
189     not encoded with text.
190     *************************************************************/
191     public static BufferedImage imageToBeSent;
192     /*************************************************************
193     The image that is loaded when the user wishes to revert the
194     effects applied to it.
195     *************************************************************/
196     public static BufferedImage imageNoEffects;
197     /*************************************************************
198     When opening an image, filters visible files.
199     *************************************************************/
200     public static FileFilter fileFilter;
201
202     /*************************************************************
203     Previously opened file path.
204     *************************************************************/
205     public static String prevOpen;
206     /*************************************************************
207     Previously opened file.
208     *************************************************************/
209     public static String prevOpenFile;
210     /*************************************************************
211     Verified previously opened file path.
212     *************************************************************/
213     public static String prevOpenChecked;
214     /*************************************************************
215     Verified previously opened file.
216     *************************************************************/
217     public static String prevOpenFileChecked;
218     /*************************************************************
219     Text encoded in the loaded image.
220     *************************************************************/
221     public static String encodedText;
222
223     /*************************************************************
224     If true, no message was found in the loaded image.
225     *************************************************************/
226     public static boolean blankErrorOccurred;
227
228     /*************************************************************
229     Takes the defined JMenu entries in the beginning of the class
230     and creates the action listeners and binds them together.
231     *************************************************************/
232     public static void initMenu()
233     {
234        menuBar = new JMenuBar();
235        fileMenu = new JMenu("File");
236        effects = new JMenu("Effects");
```

```
237         resources = new JMenu("Resources");
238       effect1 = new JMenuItem("Invert");
239       effect2 = new JMenuItem("Fade");
240       effect3 = new JMenuItem("Tint");
241       effect4 = new JMenuItem("Black/White");
242       effect5 = new JMenuItem("Remove color");
243       effect6 = new JMenuItem("Grayscale");
244       effect7 = new JMenuItem("Colorize");
245       cleareffects = new JMenuItem("Clear effects");
246       open = new JMenuItem("Open...");
247       save = new JMenuItem("Save");
248       saveAs = new JMenuItem("Save as...");
249       quit = new JMenuItem("Exit");
250         help = new JMenuItem("Help");
251         info = new JMenuItem("Info");
252 
253       effect1.addActionListener(new MenuActionListener());
254       effect2.addActionListener(new MenuActionListener());
255       effect3.addActionListener(new MenuActionListener());
256       effect4.addActionListener(new MenuActionListener());
257       effect5.addActionListener(new MenuActionListener());
258       effect6.addActionListener(new MenuActionListener());
259       effect7.addActionListener(new MenuActionListener());
260       cleareffects.addActionListener(new MenuActionListener());
261       open.addActionListener(new MenuActionListener());
262       save.addActionListener(new MenuActionListener());
263       saveAs.addActionListener(new MenuActionListener());
264       quit.addActionListener(new MenuActionListener());
265         help.addActionListener(new MenuActionListener());
266         info.addActionListener(new MenuActionListener());
267 
268       effects.add(effect1);
269       effects.add(effect2);
270       effects.add(effect3);
271       effects.add(effect4);
272       effects.add(effect5);
273       effects.add(effect6);
274       effects.add(effect7);
275       effects.add(cleareffects);
276       fileMenu.add(open);
277       fileMenu.add(save);
278       fileMenu.add(saveAs);
279       fileMenu.add(quit);
280         resources.add(info);
281         resources.add(help);
282 
283       menuBar.add(fileMenu);
284       menuBar.add(effects);
285         menuBar.add(resources);
286 
287       pictureFrame.setJMenuBar(menuBar);
288    }
289    /**************************************************************
290    Performs basic tasks to initialize the GUI.
291    **************************************************************/
292    public static void initFrame()
293     {
294       pictureFrame = new JFrame();
295       pictureFrame.setResizable(false);
```

```
296       pictureFrame.getContentPane().setLayout(new BorderLayout());
297       pictureFrame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE);
298       pictureFrame.setTitle("CRYPTICON");
299       Dimension dimension = new Dimension(40, 400);
300       pictureFrame.setPreferredSize(dimension);
301    }
302    /************************************************************
303    Creates buttons to apply inputText to the image.
304    ************************************************************/
305    public static void initButtonsAndMenu()
306    {
307       encode = new JButton("Encode");
308       decode = new JButton("Decode");
309       clear = new JButton("Clear");
310       copy = new JButton("Copy to clipboard");
311       encode.addActionListener(new MenuActionListener());
312       decode.addActionListener(new MenuActionListener());
313       copy.addActionListener(new MenuActionListener());
314       clear.addActionListener(new MenuActionListener());
315
316       setDisabled();
317       open.setEnabled(true);
318       quit.setEnabled(true);
319
320       fileFilter = new FileFilter()
321       {
322          public boolean accept(File file)
323          {
324             if (file.isDirectory())
325             {
326                return true;
327             }
328             String name = file.getName();
329             return name.toLowerCase().endsWith(".png")||name.toLowerCase().endsWith(".j
pg");
330          }
331          public String getDescription()
332          {
333             return "Image files (*.png, *.jpg)";
334          }
335       };
336    }
337    /************************************************************
338    Disables all menu items that can be used when an image is loaded.
339    ************************************************************/
340    public static void setDisabled()
341    {
342       effect1.setEnabled(false);
343       effect2.setEnabled(false);
344       effect3.setEnabled(false);
345       effect4.setEnabled(false);
346       effect5.setEnabled(false);
347       effect6.setEnabled(false);
348       effect7.setEnabled(false);
349       cleareffects.setEnabled(false);
350       save.setEnabled(false);
351       saveAs.setEnabled(false);
352       encode.setEnabled(false);
353       decode.setEnabled(false);
```

```
354       copy.setEnabled(false);
355       clear.setEnabled(false);
356    }
357    /*************************************************************
358    Enables all menu items that can be used when an image is loaded.
359    *************************************************************/
360    public static void setEnabled()
361    {
362       effect1.setEnabled(true);
363       effect2.setEnabled(true);
364       effect3.setEnabled(true);
365       effect4.setEnabled(true);
366       effect5.setEnabled(true);
367       effect6.setEnabled(true);
368       effect7.setEnabled(true);
369       cleareffects.setEnabled(true);
370       save.setEnabled(true);
371       saveAs.setEnabled(true);
372       encode.setEnabled(true);
373       decode.setEnabled(true);
374    }
375    /*************************************************************
376    Creates and formats input and output text fields to be displayed.
377    *************************************************************/
378    public static void initSidePanelandTextField()
379    {
380       sidePanel = new JPanel();
381       sidePanel.setLayout(new FlowLayout());
382       sidePanel.setBackground(new Color(160,160,160));
383       sidePanel.setBorder(BorderFactory.createLineBorder(Color.BLACK));
384       inputTextField = new JTextArea(10,11);
385       inputTextField.setBounds(5,5,100,100);
386       inputTextField.setPreferredSize(new Dimension(150,200));
387       inputTextField.setLineWrap(true);
388       inputTextField.setWrapStyleWord(true);
389       inputTextField.setBorder(BorderFactory.createCompoundBorder(BorderFactory.createL
ineBorder(Color.BLACK,2),BorderFactory.createEmptyBorder(5,5,5,5)));
390       sidePanel.add(inputTextField);
391       outputTextField = new JTextArea(10,11);
392       outputTextField.setBounds(5,5,100,100);
393       outputTextField.setPreferredSize(new Dimension(150,200));
394       outputTextField.setLineWrap(true);
395       outputTextField.setWrapStyleWord(true);
396       outputTextField.setEditable(false);
397       outputTextField.setBorder(BorderFactory.createCompoundBorder(BorderFactory.create
LineBorder(Color.BLACK,2),BorderFactory.createEmptyBorder(5,5,5,5)));
398       outputTextField.setBackground(new Color(216,216,216));
399    }
400    /*************************************************************
401    Adds the output text field to the side panel.
402    *************************************************************/
403    public static void applyOutputTextField()
404    {
405       sidePanel.add(outputTextField);
406    }
407    /*************************************************************
408    Adds encode and decode buttons to the side panel.
409    *************************************************************/
410    public static void applyButtons()
```

```
411    {
412       sidePanel.add(encode);
413       sidePanel.add(decode);
414    }
415    /*************************************************************
416    Adds the side panel to the main frame of the program.
417    *************************************************************/
418    public static void applySidePanel()
419    {
420       pictureFrame.add(sidePanel);
421    }
422    /*************************************************************
423    Creates the frame which a scaled version of the loaded image
424    will be held inside of.
425    *************************************************************/
426    public static void initImagePanel()
427    {
428        picturePanel = new JPanel(new BorderLayout());
429    }
430    /*************************************************************
431    Converts an Image to a BufferedImage.
432    *************************************************************/
433    public static BufferedImage toBufferedInstance(Image image,int width,int height)
434    {
435       BufferedImage bi = new BufferedImage(width,height,BufferedImage.TYPE_INT_ARGB);
436       Graphics2D g2d = bi.createGraphics();
437       g2d.drawImage(image, 0, 0, null);
438       g2d.dispose();
439       return bi;
440    }
441    /*************************************************************
442    Returns a filled image label, with a blank or custom image.
443    *************************************************************/
444    public static JLabel initImageLabel(String filename,int mode)
445    {
446       JLabel imageLabel;
447       BufferedImage image=null;
448       if(mode==0)
449       {
450          boolean success=false;
451          while(success==false && filename!=null){
452             try
453             {
454                image = ImageIO.read(new File(filename));
455                prevOpenChecked=prevOpen;
456                prevOpenFileChecked=prevOpenFile;
457                success=true;
458             }
459             catch(IOException e)
460             {
461             boolean again=tryAgain();
462             if (again==true)
463             {
464                filename=fnPrompt();
465                if(filename!=null)
466                {
467                   if(filename.length()>4)
468                   {
469                      if(!filename.substring(filename.length()-4).toLowerCase().equals("
```

```
.png")||!filename.substring(filename.length()-4).toLowerCase().equals(".jpg"))
                     {
                        filename+=".png";
                        try
                        {
                           ImageIO.read(new File(filename));
                           setEnabled();
                        }
                        catch(IOException e1)
                        {
                           filename = filename.substring(0,filename.length()-4);
                           filename+=".jpg";
                           try
                           {
                              ImageIO.read(new File(filename));
                              setEnabled();
                           }
                           catch(IOException e2)
                           {
                              filename = filename.substring(0,filename.length()-4);
                           }
                        }
                     }
                     else
                        setEnabled();
                  }
               }
               else
                  return prevImageLabel;
            }
            else
               return prevImageLabel;
            }
         }
      }
      else{
         try
         {
            InputStream stream = Display.class.getResourceAsStream("resources/images/bl
ank.png");
            image = ImageIO.read(stream);
         }
         catch(IOException e){}
      }
      imageNoEffects=image;
      imageLabel = initImageLabel(image);
      setImageToBeSent(image);
      prevImageLabel = imageLabel;
      return imageLabel;
   }
   /*************************************************************
   Opens a dialog prompting the user to attempt to find a file
   again.  Is called until a valid filename is retrieved.
   *************************************************************/
   public static boolean tryAgain()
   {
      JOptionPane op = new JOptionPane();
      op.setMessageType(JOptionPane.ERROR_MESSAGE);
      int i=op.showConfirmDialog(null,"File does not exist or cannot be read!\nEnsure t
```

```
he selected file is a JPG or PNG image.\nTry again?","File cannot be read!",JOptionPane.YES
_NO_OPTION);
527       if(i==JOptionPane.YES_OPTION)
528          return true;
529       if(i==JOptionPane.NO_OPTION)
530          return false;
531       else
532          return false;
533    }
534    /*************************************************************
535    Creates and returns a filled label with a custom filename.
536    *************************************************************/
537    public static JLabel initImageLabel(String filename)
538    {
539       JLabel label = initImageLabel(filename,0);
540       return label;
541    }
542    /*************************************************************
543    Returns a filled label with a scaled BufferedImage, rather than
544    a file that must be read from a disk.
545    *************************************************************/
546    public static JLabel initImageLabel(BufferedImage image)
547    {
548       blankErrorOccurred=false;
549       JLabel imageLabel;
550       height = image.getHeight();
551       width = image.getWidth();
552       loadedImage = image;
553       double scaler = 600.0/width;
554       double dsWidth = width*scaler;
555       double dsHeight = height*scaler;
556       int sWidth = (int) dsWidth;
557       int sHeight = (int) dsHeight;
558       Image sImage=image.getScaledInstance(sWidth,sHeight,Image.SCALE_SMOOTH);
559       BufferedImage bsImage=toBufferedInstance(sImage,sWidth,sHeight);
560       imageLabel = new JLabel(new ImageIcon(bsImage));
561       return imageLabel;
562    }
563    /*************************************************************
564    Gets input from the input text field and converts it to a string.
565    This method can also retrieve the loaded image's text.
566    *************************************************************/
567    public static String getText(int mode)
568    {
569       if(mode==0)
570       {
571          return encodedText;
572       }
573       else
574          return inputTextField.getText();
575     }
576    /*************************************************************
577    Returns the encoded text in the image when no parameter is
578    passed.
579    *************************************************************/
580    public static String getText()
581    {
582       return getText(0);
583    }
```

```
584    /*************************************************************
585    Returns the text from the output text field.
586    *************************************************************/
587    public static String getOutput()
588    {
589       return outputTextField.getText();
590    }
591    /*************************************************************
592    Sets the text of the output text field to the decrypted text if a
593    message was successfully decrypted from the image.
594    *************************************************************/
595    public static void setText(String string)
596    {
597       if(blankErrorOccurred==false)
598       {
599          outputTextField.setBackground(Color.WHITE);
600          outputTextField.setText(string);
601          clear.setEnabled(true);
602          copy.setEnabled(true);
603       }
604    }
605    /*************************************************************
606    Sets BufferedImage references to the passed in BufferedImage.
607    Afterwards, the initialized label containing the scaled image is
608    displayed.
609    *************************************************************/
610    public static void loadBI(BufferedImage image, int mode)
611    {
612       if (mode==1)
613       {
614          imageToBeSent = image;
615          return;
616       }
617       else if (mode==2)
618       {
619
620       }
621       else if (mode==3)
622       {
623          imageToBeSent = image;
624       }
625       else
626          imageNoEffects = image;
627       imageLabel = initImageLabel(image);
628       Border paddingBorder = BorderFactory.createEmptyBorder(10,10,10,10);
629       Border border = BorderFactory.createLineBorder(Color.BLACK,5);
630       imageLabel.setBorder(BorderFactory.createCompoundBorder(border,paddingBorder));
631       picturePanel.add(BorderLayout.EAST,imageLabel);
632       pictureFrame.getContentPane().add(BorderLayout.EAST,picturePanel);
633       pictureFrame.validate();
634       pictureFrame.repaint();
635    }
636    /*************************************************************
637    Runs loadImage again.
638    *************************************************************/
639    public static void loadImage(String filename)
640    {
641       loadImage(filename,0);
642    }
```

```
643    /*************************************************************
644    Resets variables that would have changed upon image opening and
645    changing. Takes given String and checks if the extension is
646    present, and which extensions are valid. Then, it initializes and
647    displays the image specified by the given String filename.
648    *************************************************************/
649    public static void loadImage(String filename,int mode)
650    {
651       setEncodedText(null);
652       if(mode==0)
653         {
654           filename=fnPrompt();
655             if(filename!=null)
656             {
657             if(filename.length()>4)
658             {
659                if(!filename.substring(filename.length()-4).toLowerCase().equals(".png")
||!filename.substring(filename.length()-4).toLowerCase().equals(".jpg"))
660                {
661                   filename+=".png";
662                   try
663                   {
664                      ImageIO.read(new File(filename));
665                      setEnabled();
666                   }
667                   catch(IOException e1)
668                   {
669                      filename = filename.substring(0,filename.length()-4);
670                      filename+=".jpg";
671                      try
672                      {
673                         ImageIO.read(new File(filename));
674                         setEnabled();
675                      }
676                      catch(IOException e2)
677                      {
678                         filename = filename.substring(0,filename.length()-4);
679                      }
680                   }
681                }
682                else
683                   setEnabled();
684             }
685             else
686                filename+=".png";
687                try
688                {
689                   ImageIO.read(new File(filename));
690                   setEnabled();
691                }
692                catch(IOException e1)
693                {
694                   filename = filename.substring(0,filename.length()-4);
695                   filename+=".jpg";
696                   try
697                   {
698                      ImageIO.read(new File(filename));
699                      setEnabled();
700                   }
```

```
                catch(IOException e2)
                {
                   filename = filename.substring(0,filename.length()-4);
                }
             }
          unloadImage();
          clear();
          initImageLabelS2(filename);
          Tasks.setLoadedImage(filename);
        }
      }
    else
      {
       filename = "resources/images/blank.png";
       initImageLabelS2(filename,1);
      }
  }
  /*************************************************************
  Performs the post-creation steps to accurately display the
  new image label created by initImageLabel.
  *************************************************************/
  public static void initImageLabelS2(String filename,int mode)
  {
     imageLabel=initImageLabel(filename,mode);
     Border paddingBorder = BorderFactory.createEmptyBorder(10,10,10,10);
     Border border = BorderFactory.createLineBorder(Color.BLACK,5);
     imageLabel.setBorder(BorderFactory.createCompoundBorder(border,paddingBorder));
     picturePanel.add(BorderLayout.EAST,imageLabel);
     pictureFrame.getContentPane().add(BorderLayout.EAST,picturePanel);
     pictureFrame.validate();
     pictureFrame.repaint();
  }
  /*************************************************************
  When not passed a mode, performs the default initImageLabel
  initialization and finishes the post-creation phase to display
  the image label.
  *************************************************************/
  public static void initImageLabelS2(String filename)
  {
     imageLabel=initImageLabel(filename);
     Border paddingBorder = BorderFactory.createEmptyBorder(10,10,10,10);
     Border border = BorderFactory.createLineBorder(Color.BLACK,5);
     imageLabel.setBorder(BorderFactory.createCompoundBorder(border,paddingBorder));
     picturePanel.add(BorderLayout.EAST,imageLabel);
     pictureFrame.getContentPane().add(BorderLayout.EAST,picturePanel);
     pictureFrame.validate();
     pictureFrame.repaint();
  }
  /*************************************************************
  Prompts the user, by opening a JFileChooser, for a valid
  filename. This filename will be given back to the method it was
  called in.
  *************************************************************/
  public static String fnPrompt()
  {
     JFileChooser fileChooser = new JFileChooser(prevOpen);
     fileChooser.setFileFilter(fileFilter);
     Action details = fileChooser.getActionMap().get("viewTypeDetails");
     details.actionPerformed(null);
```

```
760       int returnval = fileChooser.showOpenDialog(pictureFrame);
761          if(returnval == JFileChooser.APPROVE_OPTION)
762          {
763             String prevOpenFull = fileChooser.getSelectedFile().getAbsolutePath();
764             prevOpen = prevOpenFull;
765             int length=prevOpen.length();
766             int i=length;
767             while(i>0 && !prevOpen.substring(i-1,i).equals("\\"))
768             {
769                i--;
770             }
771             prevOpen = prevOpen.substring(0,i);
772             prevOpenFile = prevOpenFull.substring(i,length);
773             if(prevOpenFile.length()>4)
774             {
775                if(prevOpenFile.substring(prevOpenFile.length()-4).toLowerCase().equals(
".jpg"))
776                {
777                   prevOpenFile=prevOpenFile.substring(0,prevOpenFile.length()-4);
778                   prevOpenFile+=".png";
779                }
780                if(!prevOpenFile.substring(prevOpenFile.length()-4).toLowerCase().equals
(".png"))
781                {
782                   prevOpenFile+=".png";
783                }
784             }
785             else prevOpenFile+=".png";
786             return fileChooser.getSelectedFile().getAbsolutePath();
787          }
788         else
789          return null;
790
791    }
792    /*************************************************************
793    Prompts the user for a location to save the loaded BufferedImage.
794    *************************************************************/
795    public static String fnSave()
796    {
797       JFileChooser fileChooser = new JFileChooser(prevOpenChecked);
798       fileChooser.setSelectedFile(new File(prevOpenFileChecked));
799       Action details = fileChooser.getActionMap().get("viewTypeDetails");
800       details.actionPerformed(null);
801       int returnval = fileChooser.showSaveDialog(pictureFrame);
802       if(returnval == JFileChooser.APPROVE_OPTION)
803          return fileChooser.getSelectedFile().getAbsolutePath();
804       else
805          return null;
806    }
807    /*************************************************************
808    Returns the height of the loaded BufferedImage.
809    *************************************************************/
810    public static int getHeight()
811    {
812       return height;
813    }
814    /*************************************************************
815    Returns the width of the loaded BufferedImage.
816    *************************************************************/
```

```
817    public static int getWidth()
818    {
819       return width;
820    }
821    /*****************************************************************
822    Unloads and nullifies the imageLabel containing the loaded
823    BufferedImage.
824    *****************************************************************/
825    public static void unloadImage()
826    {
827       if(imageLabel!=null)
828       {
829          picturePanel.remove(imageLabel);
830          imageLabel = null;
831       }
832    }
833    /*****************************************************************
834    Returns the unencrypted BufferedImage.
835    *****************************************************************/
836    public static BufferedImage getImage()
837    {
838       return getImage(0);
839    }
840    /*****************************************************************
841    Depending on the passed in mode, returns the unencrypted loaded
842    BufferedImage, or just the encrypted or not BufferedImage.
843    *****************************************************************/
844    public static BufferedImage getImage(int mode)
845    {
846       if(mode==1)
847          return loadedImage;
848       else
849          return imageToBeSent;
850    }
851    /*****************************************************************
852    Add the 'Copy to clipboard' button to the side panel.
853    *****************************************************************/
854    public static void applyCopy()
855    {
856       sidePanel.add(copy);
857    }
858    /*****************************************************************
859    Adds each element to the side panel in order.
860    *****************************************************************/
861    public static void apply()
862    {
863       applyButtons();
864       applyOutputTextField();
865       applyCopy();
866       applyClear();
867       applySidePanel();
868    }
869    /*****************************************************************
870    Centers CRYPTICON on the desktop.
871    *****************************************************************/
872    public static void centerFrame()
873    {
874       Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
875       Point middle = new Point(screenSize.width/2,screenSize.height/2);
```

```
876       Point middleCorrected = new Point(middle.x-(pictureFrame.getWidth()/2),middle.y-(
pictureFrame.getHeight()/2));
877       pictureFrame.setLocation(middleCorrected);
878    }
879    /*****************************************************************
880    Set the passed in BufferedImage to the unencrypted BufferedImage
881    reference in the Display class.
882    *****************************************************************/
883    public static void setImageToBeSent(BufferedImage image)
884    {
885       imageToBeSent=image;
886    }
887    /*****************************************************************
888    Add the 'clear' button to the side panel.
889    *****************************************************************/
890    public static void applyClear()
891    {
892       sidePanel.add(clear);
893    }
894    /*****************************************************************
895    Clear and reset the formatting of the output text field.
896    Disable buttons that would have no function when output text field
897    contains no text.
898    *****************************************************************/
899    public static void clear()
900    {
901       outputTextField.setText("");
902       outputTextField.setBackground(new Color(216,216,216));
903       copy.setEnabled(false);
904       clear.setEnabled(false);
905    }
906    /*****************************************************************
907    Loads the BufferedImage that was not used when effects were applied.
908    All effects applied to a loaded, unsaved image are reset.
909    *****************************************************************/
910    public static void clearEffects()
911    {
912       Display.unloadImage();
913       loadBI(imageNoEffects,1);
914       BufferedImage tmpimage = imageNoEffects;
915       try
916       {
917          imageNoEffects=Steg.encrypt(imageNoEffects);
918          loadBI(imageNoEffects,2);
919       }
920       catch(Exception e){loadBI(tmpimage,1);}
921    }
922    /*****************************************************************
923    Store text encoded in the image.
924    *****************************************************************/
925    public static void setEncodedText(String message)
926    {
927       encodedText = message;
928    }
929    /*****************************************************************
930    If a desired string to be encoded is too long, produce an error
931    and tell the user how many characters need to be removed.
932    *****************************************************************/
933    public static void lengthError(int over)
```

```
934    {
935       String s="s";
936       if(over==1)
937          s="";
938       JOptionPane.showMessageDialog(null,"Your message is "+over+" character"+s+" too l
ong.\nYou must shorten your message or choose a larger image.","Error!",JOptionPane.ERROR_M
ESSAGE);
939    }
940    /*************************************************************
941    If no message is found in the image, display an error.
942    *************************************************************/
943    public static void blankError()
944    {
945       JOptionPane.showMessageDialog(null,"No message was found in this image.\nEnsure t
hat it was encoded using CRYPTICON.","Error!",JOptionPane.ERROR_MESSAGE);
946       blankErrorOccurred=true;
947       clear();
948    }
949    /*************************************************************
950    Let the user know, if the encryption may take a long time, when
951    the process has started.
952    *************************************************************/
953    public static void eTimeErrorStart()
954    {
955       JOptionPane.showMessageDialog(null,"Encryption of this message may take a substan
tial amount of time,\ndepending on the processing power of your machine.\nAnother popup wil
l inform you when this process is finished.\nContinue?","Encrypting...",JOptionPane.ERROR_M
ESSAGE);
956    }
957    /*************************************************************
958    If the encryption was thought to take a long time, when the process
959    has finished, notify the user.
960    *************************************************************/
961    public static void eTimeErrorEnd()
962    {
963       JOptionPane.showMessageDialog(null,"The encryption process has been completed.\nY
ou are now free to save this image.","Process completed.",JOptionPane.INFORMATION_MESSAGE);
964    }
965    /*************************************************************
966    If the decryption of a certain image is thought to take a long time,
967    notifies the user of this, and asks them
968    if they want to proceed.
969    *************************************************************/
970    public static int timeError()
971    {
972       int choice = JOptionPane.showConfirmDialog(null,"Decryption of this message may t
ake a substantial amount of time due to its length,\ndepending on the processing power of y
our machine.\nHowever, the message should be able to be retrieved.\nContinue?","Decrypting.
..",JOptionPane.YES_NO_OPTION);
973       if(choice==JOptionPane.YES_OPTION)
974          return 1;
975       else
976          return 0;
977    }
978    /*************************************************************
979    Sets up the frames to hold the images and menus and loads them.
980    *************************************************************/
981    public static void init()
982    {
```

```
983       initFrame();
984       initMenu();
985       initSidePanelandTextField();
986       initButtonsAndMenu();
987       apply();
988       pictureFrame.pack();
989       initImagePanel();
990       loadImage("",1);
991       Dimension dim = new Dimension(800,600);
992       pictureFrame.setPreferredSize(dim);
993       pictureFrame.setSize(dim);
994       centerFrame();
995       blankErrorOccurred=false;
996       pictureFrame.setIconImage(new ImageIcon(Display.class.getResource("resources/imag
es/icon.ico")).getImage());
997       pictureFrame.setVisible(true);
998
999    }
000 }
```