```
/*
 * ========================================================================
 * DrawingPanel.java
 * Simplified Java drawing window class
 * to accompany Building Java Programs textbook and associated materials
 *
 * authors: Stuart Reges, University of Washington
 *          Marty Stepp
 * version: 4.07, 2022/04/07 (BJP 5th edition)
 * (make sure to also update version string in Javadoc header below!)
 * ========================================================================
 *
 * COMPATIBILITY NOTE: This version of DrawingPanel requires Java 8 or higher.
 * If you need a version that works on Java 7 or lower, please see our
 * web site at http://www.buildingjavaprograms.com/ .
 * To make this file work on Java 7 and lower, you must make two small
 * modifications to its source code.
 * Search for the two occurrences of the annotation @FunctionalInterface
 * and comment them out or remove those lines.
 * Then the file should compile and run properly on older versions of Java.
 *
 * ========================================================================
 *
 * The DrawingPanel class provides a simple interface for drawing persistent
 * images using a Graphics object.  An internal BufferedImage object is used
 * to keep track of what has been drawn.  A client of the class simply
 * constructs a DrawingPanel of a particular size and then draws on it with
 * the Graphics object, setting the background color if they so choose.
 * See JavaDoc comments below for more information.
 */

import java.awt.FontMetrics;
import java.awt.Rectangle;
import java.awt.Shape;
import java.awt.image.ImageObserver;
import java.text.AttributedCharacterIterator;
import java.util.Collections;
import java.awt.AlphaComposite;
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Composite;
import java.awt.Container;
import java.awt.Dimension;
import java.awt.EventQueue;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.Frame;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.GridLayout;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.Point;
import java.awt.RenderingHints;
import java.awt.Toolkit;
import java.awt.Window;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.awt.event.WindowListener;
import java.awt.image.BufferedImage;
import java.awt.image.PixelGrabber;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintStream;
import java.lang.Exception;
import java.lang.Integer;
import java.lang.InterruptedException;
import java.lang.Math;
import java.lang.Object;
import java.lang.OutOfMemoryError;
import java.lang.SecurityException;
import java.lang.String;
import java.lang.System;
import java.lang.Thread;
import java.net.URL;
import java.net.NoRouteToHostException;
```

```
import java.net.SocketException;
import java.net.UnknownHostException;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.TreeMap;
import java.util.Vector;
import javax.imageio.ImageIO;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JColorChooser;
import javax.swing.JDialog;
import javax.swing.JFileChooser;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JSlider;
import javax.swing.KeyStroke;
import javax.swing.Timer;
import javax.swing.UIManager;
import javax.swing.event.ChangeEvent;
import javax.swing.event.ChangeListener;
import javax.swing.event.MouseInputAdapter;
import javax.swing.event.MouseInputListener;
import javax.swing.filechooser.FileFilter;

/**
 *
 * DrawingPanel is a simplified Java drawing window class to accompany
 * Building Java Programs textbook and associated materials.
 *
 * <p>
 * Authors: Stuart Reges (University of Washington) and Marty Stepp.
 *
 * <p>
 * Version: 4.07, 2022/04/07 (to accompany BJP 5th edition).
 *
 * <p>
 * You can always download the latest {@code DrawingPanel} from
 * <a target="_blank" href="http://www.buildingjavaprograms.com/drawingpanel/DrawingPanel.java">
 * http://www.buildingjavaprograms.com/drawingpanel/DrawingPanel.java</a> .
 *
 * <p>
 * For more information and related materials, please visit
 * <a target="_blank" href="http://www.buildingjavaprograms.com">
 * www.buildingjavaprograms.com</a> .
 *
 * <p>
 * COMPATIBILITY NOTE: This version of DrawingPanel requires Java 8 or higher.
 * To make this file work on Java 7 and lower, you must make two small
 * modifications to its source code.
 * Search for the two occurrences of the annotation @FunctionalInterface
 * and comment them out or remove those lines.
 * Then the file should compile and run properly on older versions of Java.
 *
 * <h3>Description:</h3>
 *
 * <p>
 * The {@code DrawingPanel} class provides a simple interface for drawing persistent
 * images using a {@code Graphics} object.  An internal {@code BufferedImage} object is used
 * to keep track of what has been drawn.  A client of the class simply
 * constructs a {@code DrawingPanel} of a particular size and then draws on it with
 * the {@code Graphics} object, setting the background color if they so choose.
 * </p>
 *
 * <p>
 * The intention is that this custom library will mostly "stay out of the way"
 * so that the client mostly interacts with a standard Java {@code java.awt.Graphics}
 * object, and therefore most of the experience gained while using this library
 * will transfer to Java graphics programming in other contexts.
 * {@code DrawingPanel} is not intended to be a full rich graphical library for things
 * like object-oriented drawing of shapes, animations, creating games, etc.
 * </p>
 *
 * <h3>Example basic usage:</h3>
 *
 * <p>
 * Here is a canonical example of creating a {@code DrawingPanel} of a given size and
```

```
* using it to draw a few shapes.
* </p>
*
* <pre>
* // basic usage example
* DrawingPanel panel = new DrawingPanel(600, 400);
* Graphics g = panel.getGraphics();
* g.setColor(Color.RED);
* g.fillRect(17, 45, 139, 241);
* g.drawOval(234, 77, 100, 100);
* ...
* </pre>
*
* <p>
* To ensure that the image is always displayed, a timer calls repaint at
* regular intervals.
* </p>
*
* <h3>Pixel processing (new in BJP 4th edition):</h3>
*
* <p>
* This version of {@code DrawingPanel} allows you to loop over the pixels of an image.
* You can process each pixel as a {@code Color} object (easier OO interface, but takes
* more CPU and memory to run) or as a 32-bit RGB integer (clunkier to use, but
* much more efficient in runtime and memory usage).
* Look at the methods get/setPixel(s) to get a better idea.
*
* <pre>
* // example of horizontally flipping an image
* public static void flipHorizontal(DrawingPanel panel) {
*     int width  = panel.getWidth();
*     int height = panel.getHeight();
*     int[][] pixels = panel.getPixelsRGB();
*     for (int row = 0; row &lt; height; row++) {
*         for (int col = 0; col &lt; width / 2; col++) {
*             // swap this pixel with the one opposite it
*             int col2 = width - 1 - col;
*             int temp = pixels[row][col];
*             pixels[row][col] = pixels[row][col2];
*             pixels[row][col2] = temp;
*         }
*     }
*     panel.setPixels(pixels);
* }
* </pre>
*
* <h3>Event listeners and lambdas (new in BJP 4th edition):</h3>
*
* <p>
* With Java 8, you can now attach event handlers to listen to keyboard and mouse
* events that occur in a {@code DrawingPanel} using a lambda function.  For example:
*
* <pre>
* // example of attaching a mouse click handler using Java 8
* panel.onClick( (x, y) -&gt; System.out.println(x + " " + y) );
* </pre>

* <h3>Debugging facilities (new in BJP 4th edition):</h3>
*
* <p>
* This version now includes an inner class named {@code DebuggingGraphics}
* that keeps track of how many times various drawing methods are called.
* It includes a {@code showCounts} method for the {@code DrawingPanel} itself
* that allows a client to examine this.  The panel will record basic drawing
* methods performed by a version of the {@code Graphics} class obtained by
* calling {@code getDebuggingGraphics} :
*
* <pre>
* // example of debugging counts of graphics method calls
* Graphics g = panel.getDebuggingGraphics();
* </pre>
*
* <p>
* Novices will be encouraged to simply print it at the end of {@code main}, as in:
*
* <pre>
* System.out.println(panel.getCounts());
* </pre>
*
* <h3>History and recent changes:</h3>
*
* 2022/04/07
* - Minor update to remove a security manager-related compiler warning in JDK 17+.
*
* 2016/07/25
* - Added and cleaned up BJP4 features, static anti-alias settings, bug fixes.
* <p>
```

```
 *
 * 2016/03/07
 * - Code cleanup and improvements to JavaDoc comments for BJP4 release.
 * <p>
 *
 * 2015/09/04
 * - Now includes methods for get/setting individual pixels and all pixels on the
 *   drawing panel.  This helps facilitate 2D array-based pixel-processing
 *   exercises and problems for Building Java Programs, 4th edition.
 * - Code cleanup and reorganization.
 *   Now better alphabetization/formatting of members and encapsulation.
 *   Commenting also augmented throughout code.
 * <p>
 *
 * 2015/04/09
 * - Now includes a DebuggingGraphics inner class that keeps track of how many
 *   times various drawing methods are called.
 *   All additions are commented (search for "DebuggingGraphics")
 * <p>
 *
 * 2011/10/25
 * - save zoomed images (2011/10/25)
 * <p>
 *
 * 2011/10/21
 * - window no longer moves when zoom changes
 * - grid lines
 *
 * @author Stuart Reges (University of Washington) and Marty Stepp
 * @version 4.07, 2022/04/07 (BJP 5th edition)
 */
public final class DrawingPanel implements ImageObserver {
        // class constants
        private static final Color GRID_LINE_COLOR      = new Color(64, 64, 64, 128);   // color of grid lines on panel
        private static final Object LOCK                = new Object();                 // object used for concurrency locking

        private static final boolean SAVE_SCALED_IMAGES = true;    // if true, when panel is zoomed, saves images at that zoom factor
        private static final int DELAY                  = 100;     // delay between repaints in millis
        private static final int MAX_FRAMES             = 100;     // max animation frames
        private static final int MAX_SIZE               = 10000;   // max width/height
        private static final int GRID_LINES_PX_GAP_DEFAULT = 10;   // default px between grid lines

        private static final String VERSION             = "4.07 (2022/04/07)";
        private static final String ABOUT_MESSAGE       = "DrawingPanel\n"
                        + "Graphical library class to support Building Java Programs textbook\n"
                        + "written by Stuart Reges, University of Washington\n"
                        + "and Marty Stepp\n\n"
                        + "Version: " + VERSION + "\n\n"
                        + "please visit our web site at:\n"
                        + "http://www.buildingjavaprograms.com/";
        private static final String ABOUT_MESSAGE_TITLE = "About DrawingPanel";
        private static final String COURSE_WEB_SITE     = "https://courses.cs.washington.edu/courses/cse142/CurrentQtr/drawingpanel.txt";
        private static final String TITLE               = "Drawing Panel";

        /** An RGB integer representing alpha at 100% opacity (0xff000000). */
        public static final int PIXEL_ALPHA             = 0xff000000;   // rgb integer for alpha 100% opacity

        /** An RGB integer representing 100% blue (0x000000ff). */
        public static final int PIXEL_BLUE              = 0x000000ff;   // rgb integer for 100% blue

        /** An RGB integer representing 100% green (0x0000ff00). */
        public static final int PIXEL_GREEN             = 0x0000ff00;   // rgb integer for 100% green

        /** An RGB integer representing 100% red (0x00ff0000). */
        public static final int PIXEL_RED               = 0x00ff0000;   // rgb integer for 100% red

        /**
         * The default width of a DrawingPanel in pixels, if none is supplied at construction (500 pixels).
         */
        public static final int DEFAULT_WIDTH           = 500;

        /**
         * The default height of a DrawingPanel in pixels, if none is supplied at construction (400 pixels).
         */
        public static final int DEFAULT_HEIGHT          = 400;

        /** An internal constant for setting system properties; clients should not use this. */
        public static final String ANIMATED_PROPERTY    = "drawingpanel.animated";

        /** An internal constant for setting system properties; clients should not use this. */
        public static final String ANIMATION_FILE_NAME  = "_drawingpanel_animation_save.txt";

        /** An internal constant for setting system properties; clients should not use this. */
        public static final String ANTIALIAS_PROPERTY    = "drawingpanel.antialias";

        /** An internal constant for setting system properties; clients should not use this. */
        public static final String AUTO_ENABLE_ANIMATION_ON_SLEEP_PROPERTY = "drawingpanel.animateonsleep";
```

```
/** An internal constant for setting system properties; clients should not use this. */
public static final String DIFF_PROPERTY        = "drawingpanel.diff";

/** An internal constant for setting system properties; clients should not use this. */
public static final String HEADLESS_PROPERTY    = "drawingpanel.headless";
private static final String AWT_HEADLESS_PROPERTY = "java.awt.headless";

/** An internal constant for setting system properties; clients should not use this. */
public static final String MULTIPLE_PROPERTY    = "drawingpanel.multiple";

/** An internal constant for setting system properties; clients should not use this. */
public static final String SAVE_PROPERTY        = "drawingpanel.save";

/* a list of all DrawingPanel instances ever created; used for saving graphical output */
private static final List<DrawingPanel> INSTANCES = new ArrayList<DrawingPanel>();

// static variables
private static boolean DEBUG = false;
private static int instances = 0;
private static String saveFileName = null;
private static Boolean headless = null;
private static Boolean antiAliasDefault = true;
private static Thread shutdownThread = null;

// static class initializer - sets up thread to close program if
// last DrawingPanel is closed
static {
        try {
                String debugProp = String.valueOf(System.getProperty("drawingpanel.debug")).toLowerCase();
                DEBUG = DEBUG || "true".equalsIgnoreCase(debugProp)
                              || "on".equalsIgnoreCase(debugProp)
                              || "yes".equalsIgnoreCase(debugProp)
                              || "1".equals(debugProp);
        } catch (Throwable t) {
                // empty
        }
}

/*
 * Called when DrawingPanel class loads up.
 * Checks whether the user wants to save an animation to a file.
 */
private static void checkAnimationSettings() {
        try {
                File settingsFile = new File(ANIMATION_FILE_NAME);
                if (settingsFile.exists()) {
                        Scanner input = new Scanner(settingsFile);
                        String animationSaveFileName = input.nextLine();
                        input.close();
                        System.out.println("***");
                        System.out.println("*** DrawingPanel saving animated GIF: " +
                                        new File(animationSaveFileName).getName());
                        System.out.println("***");
                        settingsFile.delete();

                        System.setProperty(ANIMATED_PROPERTY, "1");
                        System.setProperty(SAVE_PROPERTY, animationSaveFileName);
                }
        } catch (Exception e) {
                if (DEBUG) {
                        System.out.println("error checking animation settings: " + e);
                }
        }
}

/*
 * Helper that throws an IllegalArgumentException if the given integer
 * is not between the given min-max inclusive
 */
private static void ensureInRange(String name, int value, int min, int max) {
        if (value < min || value > max) {
                throw new IllegalArgumentException(name + " must be between " + min
                                + " and " + max + ", but saw " + value);
        }
}

/*
 * Helper that throws a NullPointerException if the given value is null
 */
private static void ensureNotNull(String name, Object value) {
        if (value == null) {
                throw new NullPointerException("null value was passed for " + name);
        }
}

/**
```

```
	 * Returns the alpha (opacity) component of the given RGB pixel from 0-255.
	 * Often used in conjunction with the methods getPixelRGB, setPixelRGB, etc.
	 * @param rgb RGB integer with alpha in bits 0-7, red in bits 8-15, green in
	 * bits 16-23, and blue in bits 24-31
	 * @return alpha component from 0-255
	 */
	public static int getAlpha(int rgb) {
		return (rgb & 0xff000000) >> 24;
	}

	/**
	 * Returns the blue component of the given RGB pixel from 0-255.
	 * Often used in conjunction with the methods getPixelRGB, setPixelRGB, etc.
	 * @param rgb RGB integer with alpha in bits 0-7, red in bits 8-15, green in
	 * bits 16-23, and blue in bits 24-31
	 * @return blue component from 0-255
	 */
	public static int getBlue(int rgb) {
		return (rgb & 0x000000ff);
	}

	/**
	 * Returns the green component of the given RGB pixel from 0-255.
	 * Often used in conjunction with the methods getPixelRGB, setPixelRGB, etc.
	 * @param rgb RGB integer with alpha in bits 0-7, red in bits 8-15, green in
	 * bits 16-23, and blue in bits 24-31
	 * @return green component from 0-255
	 */
	public static int getGreen(int rgb) {
		return (rgb & 0x0000ff00) >> 8;
	}

	/**
	 * Returns the red component of the given RGB pixel from 0-255.
	 * Often used in conjunction with the methods getPixelRGB, setPixelRGB, etc.
	 * @param rgb RGB integer with alpha in bits 0-7, red in bits 8-15, green in
	 * bits 16-23, and blue in bits 24-31
	 * @return red component from 0-255
	 */
	public static int getRed(int rgb) {
		return (rgb & 0x00ff0000) >> 16;
	}

	/*
	 * Returns the given Java system property as a Boolean.
	 * Note uppercase-B meaning that if the property isn't set, this will return null.
	 * That also means that if you call it and try to store as lowercase-B boolean and
	 * it's null, you will crash the program.  You have been warned.
	 */
	private static Boolean getPropertyBoolean(String name) {
		try {
			String prop = System.getProperty(name);
			if (prop == null) {
				return null;
			} else {
				return name.equalsIgnoreCase("true")
						|| name.equals("1")
						|| name.equalsIgnoreCase("on")
						|| name.equalsIgnoreCase("yes");
			}
		} catch (SecurityException e) {
			if (DEBUG) System.out.println("Security exception when trying to read " + name);
			return null;
		}
	}

	/**
	 * Returns the file name used for saving all DrawingPanel instances.
	 * By default this is null, but it can be set using setSaveFileName
	 * or by setting the SAVE_PROPERTY env variable.
	 * @return the shared save file name
	 */
	public static String getSaveFileName() {
		if (saveFileName == null) {
			try {
				saveFileName = System.getProperty(SAVE_PROPERTY);
			} catch (SecurityException e) {
				// empty
			}
		}
		return saveFileName;
	}

	/*
	 * Returns whether the given Java system property has been set.
	 */
	private static boolean hasProperty(String name) {
```

```
                try {
                        return System.getProperty(name) != null;
                } catch (SecurityException e) {
                        if (DEBUG) System.out.println("Security exception when trying to read " + name);
                        return false;
                }
        }

        /**
         * Returns true if DrawingPanel instances should anti-alias (smooth) their graphics.
         * By default this is true, but it can be set to false using the ANTIALIAS_PROPERTY.
         * @return true if anti-aliasing is enabled (default true)
         */
        public static boolean isAntiAliasDefault() {
                if (antiAliasDefault != null) {
                        return antiAliasDefault;
                } else if (hasProperty(ANTIALIAS_PROPERTY)) {
                        return getPropertyBoolean(ANTIALIAS_PROPERTY);
                } else {
                        return true;    // default
                }
        }

        /**
         * Returns true if the class is in "headless" mode, meaning that it is running on
         * a server without a graphical user interface.
         * @return true if we are in headless mode (default false)
         */
        public static boolean isHeadless() {
                if (headless != null) {
                        return headless;
                } else {
                        return hasProperty(HEADLESS_PROPERTY) && getPropertyBoolean(HEADLESS_PROPERTY);
                }
        }

        /**
         * Internal method; returns whether the 'main' thread is still running.
         * Used to determine whether to exit the program when the drawing panel
         * is closed by the user.
         * This is an internal method not meant to be called by clients.
         * @return true if main thread is still running
         */
        public static boolean mainIsActive() {
                ThreadGroup group = Thread.currentThread().getThreadGroup();
                int activeCount = group.activeCount();

                // look for the main thread in the current thread group
                Thread[] threads = new Thread[activeCount];
                group.enumerate(threads);
                for (int i = 0; i < threads.length; i++) {
                        Thread thread = threads[i];
                        String name = String.valueOf(thread.getName()).toLowerCase();
                        if (DEBUG) System.out.println("    DrawingPanel.mainIsActive(): " + thread.getName() + ", priority=" + thread.
getPriority() + ", alive=" + thread.isAlive() + ", stack=" + java.util.Arrays.toString(thread.getStackTrace()));
                        if (name.indexOf("main") >= 0 ||
                                name.indexOf("testrunner-assignmentrunner") >= 0) {
                                // found main thread!
                                // (TestRunnerApplet's main runner also counts as "main" thread)
                                return thread.isAlive();
                        }
                }

                // didn't find a running main thread; guess that main is done running
                return false;
        }

        /*
         * Returns whether the given Java system property has been set to a
         * "truthy" value such as "yes" or "true" or "1".
         */
        private static boolean propertyIsTrue(String name) {
                try {
                        String prop = System.getProperty(name);
                        return prop != null && (prop.equalsIgnoreCase("true")
                                        || prop.equalsIgnoreCase("yes")
                                        || prop.equalsIgnoreCase("1"));
                } catch (SecurityException e) {
                        if (DEBUG) System.out.println("Security exception when trying to read " + name);
                        return false;
                }
        }

        /**
         * Saves every DrawingPanel instance that is active.
         * @throws IOException if unable to save any of the files.
         */
```

```
	public static void saveAll() throws IOException {
		for (DrawingPanel panel : INSTANCES) {
			if (!panel.hasBeenSaved) {
				panel.save(getSaveFileName());
			}
		}
	}

	/**
	 * Sets whether DrawingPanel instances should anti-alias (smooth) their pixels by default.
	 * Default true.  You can set this on a given DrawingPanel instance with setAntialias(boolean).
	 * @param value whether to enable anti-aliasing (default true)
	 */
	public static void setAntiAliasDefault(Boolean value) {
		antiAliasDefault = value;
	}

	/**
	 * Sets the class to run in "headless" mode, with no graphical output on screen.
	 * @param value whether to enable headless mode (default false)
	 */
	public static void setHeadless(Boolean value) {
		headless = value;
		if (headless != null) {
			if (headless) {
				// Set up Java AWT graphics configuration so that it can draw in 'headless' mode
				// (without popping up actual graphical windows on the server's monitor)
				// creating the buffered image below will prep the classloader so that Image
				// classes are available later to the JVM
				System.setProperty(AWT_HEADLESS_PROPERTY, "true");
				System.setProperty(HEADLESS_PROPERTY, "true");
				java.awt.image.BufferedImage img = new java.awt.image.BufferedImage(100, 100, java.awt.image.BufferedImage.
TYPE_INT_RGB);
				img.getGraphics().drawRect(10, 20, 30, 40);
			} else {
				System.setProperty(AWT_HEADLESS_PROPERTY, "false");
				System.setProperty(HEADLESS_PROPERTY, "false");
			}
		}
	}

	/**
	 * Sets the file to be used when saving graphical output for all DrawingPanels.
	 * @param file the file to use as default save file
	 */
	public static void setSaveFile(File file) {
		setSaveFileName(file.toString());
	}

	/**
	 * Sets the filename to be used when saving graphical output for all DrawingPanels.
	 * @param filename the name/path of the file to use as default save file
	 */
	public static void setSaveFileName(String filename) {
		try {
			System.setProperty(SAVE_PROPERTY, filename);
		} catch (SecurityException e) {
			// empty
		}
		saveFileName = filename;
	}

	/**
	 * Returns an RGB integer made from the given red, green, and blue components
	 * from 0-255.  The returned integer is suitable for use with various RGB
	 * integer methods in this class such as setPixel.
	 * @param r red component from 0-255 (bits 8-15)
	 * @param g green component from 0-255 (bits 16-23)
	 * @param b blue component from 0-255 (bits 24-31)
	 * @return RGB integer with full 255 for alpha and r-g-b in bits 8-31
	 * @throws IllegalArgumentException if r, g, or b is not in 0-255 range
	 */
	public static int toRgbInteger(int r, int g, int b) {
		return toRgbInteger(/* alpha */ 255, r, g, b);
	}


	/**
	 * Returns an RGB integer made from the given alpha, red, green, and blue components
	 * from 0-255.  The returned integer is suitable for use with various RGB
	 * integer methods in this class such as setPixel.
	 * @param alpha alpha (transparency) component from 0-255 (bits 0-7)
	 * @param r red component from 0-255 (bits 8-15)
	 * @param g green component from 0-255 (bits 16-23)
	 * @param b blue component from 0-255 (bits 24-31)
	 * @return RGB integer with the given four components
	 * @throws IllegalArgumentException if alpha, r, g, or b is not in 0-255 range
```

```
     */
    public static int toRgbInteger(int alpha, int r, int g, int b) {
            ensureInRange("alpha", alpha, 0, 255);
            ensureInRange("red", r, 0, 255);
            ensureInRange("green", g, 0, 255);
            ensureInRange("blue", b, 0, 255);
            return ((alpha & 0x000000ff) << 24)
                         | ((r & 0x000000ff) << 16)
                         | ((g & 0x000000ff) << 8)
                         | ((b & 0x000000ff));
    }

    /*
     * Returns whether the current program is running in the DrJava editor.
     * This was needed in the past because DrJava messed with some settings.
     */
    private static boolean usingDrJava() {
            try {
                    return System.getProperty("drjava.debug.port") != null ||
                            System.getProperty("java.class.path").toLowerCase().indexOf("drjava") >= 0;
            } catch (SecurityException e) {
                    // running as an applet, or something
                    return false;
            }
    }

    // fields
    private ActionListener actionListener;
    private List<ImageFrame> frames;        // stores frames of animation to save
    private boolean animated = false;       // changes to true if sleep() is called
    private boolean antialias = isAntiAliasDefault();   // true to smooth corners of shapes
    private boolean gridLines = false;      // grid lines every 10px on screen
    private boolean hasBeenSaved = false;  // set true once saved to file (to avoid re-saving same panel)
    private BufferedImage image;            // remembers drawing commands
    private Color backgroundColor = Color.WHITE;
    private Gif89Encoder encoder;           // for saving animations
    private Graphics g3;                    // new field to support DebuggingGraphics
    private Graphics2D g2;                  // graphics context for painting
    private ImagePanel imagePanel;          // real drawing surface
    private int currentZoom = 1;            // panel's zoom factor for drawing
    private int gridLinesPxGap = GRID_LINES_PX_GAP_DEFAULT;    // px between grid lines
    private int initialPixel;               // initial value in each pixel, for clear()
    private int instanceNumber;             // every DPanel has a unique number
    private int width;                      // dimensions of window frame
    private int height;                     // dimensions of window frame
    private JFileChooser chooser;           // file chooser to save files
    private JFrame frame;                   // overall window frame
    private JLabel statusBar;               // status bar showing mouse position
    private JPanel panel;                   // overall drawing surface
    private long createTime;                // time at which DrawingPanel was constructed
    private Map<String, Integer> counts;    // new field to support DebuggingGraphics
    private MouseInputListener mouseListener;
    private String callingClassName;        // name of class that constructed this panel
    private Timer timer;                    // animation timer
    private WindowListener windowListener;

    /**
     * Constructs a drawing panel with a default width and height enclosed in a window.
     * Uses DEFAULT_WIDTH and DEFAULT_HEIGHT for the panel's size.
     */
    public DrawingPanel() {
            this(DEFAULT_WIDTH, DEFAULT_HEIGHT);
    }

    /**
     * Constructs a drawing panel of given width and height enclosed in a window.
     * @param width panel's width in pixels
     * @param height panel's height in pixels
     */
    public DrawingPanel(int width, int height) {
            ensureInRange("width", width, 0, MAX_SIZE);
            ensureInRange("height", height, 0, MAX_SIZE);

            checkAnimationSettings();

            if (DEBUG) System.out.println("DrawingPanel(): going to grab lock");
            synchronized (LOCK) {
                    instances++;
                    instanceNumber = instances;  // each DrawingPanel stores its own int number
                    INSTANCES.add(this);

                    if (shutdownThread == null && !usingDrJava()) {
                            if (DEBUG) System.out.println("DrawingPanel(): starting idle thread");
                            shutdownThread = new Thread(new Runnable() {
                                    // Runnable implementation; used for shutdown thread.
                                    public void run() {
                                            boolean save = shouldSave();
```

```
                                        try {
                                                while (true) {
                                                        // maybe shut down the program, if no more DrawingPanels are onscreen
                                                        // and main has finished executing
                                                        save |= shouldSave();
                                                        if (DEBUG) System.out.println("DrawingPanel idle thread: instances=" +
instances + ", save=" + save + ", main active=" + mainIsActive());
                                                        if ((instances == 0 || save) && !mainIsActive()) {
                                                                try {
                                                                        System.exit(0);
                                                                } catch (SecurityException sex) {
                                                                        if (DEBUG) System.out.println("DrawingPanel idle thread:
unable to exit program: " + sex);
                                                                }
                                                        }

                                                        Thread.sleep(250);
                                                }
                                        } catch (Exception e) {
                                                if (DEBUG) System.out.println("DrawingPanel idle thread: exception caught: " + e);
                                        }
                                }
                        });
                        // shutdownThread.setPriority(Thread.MIN_PRIORITY);
                        shutdownThread.setName("DrawingPanel-shutdown");
                        shutdownThread.start();
                }
        }

        this.width = width;
        this.height = height;

        if (DEBUG) System.out.println("DrawingPanel(w=" + width + ",h=" + height + ",anim=" + isAnimated() + ",graph=" + isGraphical()
+ ",save=" + shouldSave());

        if (isAnimated() && shouldSave()) {
                // image must be no more than 256 colors
                image = new BufferedImage(width, height, BufferedImage.TYPE_BYTE_INDEXED);
                // image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
                antialias = false;   // turn off anti-aliasing to save palette colors

                // initially fill the entire frame with the background color,
                // because it won't show through via transparency like with a full ARGB image
                Graphics g = image.getGraphics();
                g.setColor(backgroundColor);
                g.fillRect(0, 0, width + 1, height + 1);
        } else {
                image = new BufferedImage(width, height, BufferedImage.TYPE_INT_ARGB);
        }
        initialPixel = image.getRGB(0, 0);

        g2 = (Graphics2D) image.getGraphics();
        // new field assignments for DebuggingGraphics
        g3 = new DebuggingGraphics();
        counts = new TreeMap<String, Integer>();
        g2.setColor(Color.BLACK);
        if (antialias) {
                g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
        }

        if (isAnimated()) {
                initializeAnimation();
        }

        if (isGraphical()) {
                try {
                        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
                } catch (Exception e) {
                        // empty
                }

                statusBar = new JLabel(" ");
                statusBar.setBorder(BorderFactory.createLineBorder(Color.BLACK));

                panel = new JPanel(new FlowLayout(FlowLayout.LEFT, 0, 0));
                panel.setBackground(backgroundColor);
                panel.setPreferredSize(new Dimension(width, height));
                imagePanel = new ImagePanel(image);
                imagePanel.setBackground(backgroundColor);
                panel.add(imagePanel);

                // listen to mouse movement
                mouseListener = new DPMouseListener();
                panel.addMouseMotionListener(mouseListener);

                // main window frame
                frame = new JFrame(TITLE);
```

```
                // frame.setResizable(false);
                windowListener = new DPWindowListener();
                frame.addWindowListener(windowListener);
                // JPanel center = new JPanel(new FlowLayout(FlowLayout.CENTER, 0, 0));
                JScrollPane center = new JScrollPane(panel);
                // center.add(panel);
                frame.getContentPane().add(center);
                frame.getContentPane().add(statusBar, "South");
                frame.setBackground(Color.DARK_GRAY);

                // menu bar
                actionListener = new DPActionListener();
                setupMenuBar();

                frame.pack();
                center(frame);
                frame.setVisible(true);
                if (!shouldSave()) {
                        toFront(frame);
                }

                // repaint timer so that the screen will update
                createTime = System.currentTimeMillis();
                timer = new Timer(DELAY, actionListener);
                timer.start();
        } else if (shouldSave()) {
                // headless mode; just set a hook on shutdown to save the image
                callingClassName = getCallingClassName();
                try {
                        Runtime.getRuntime().addShutdownHook(new Thread(new Runnable() {
                                // run on shutdown to save the image
                                public void run() {
                                        if (DEBUG) System.out.println("DrawingPanel.run(): Running shutdown hook");
                                        if (DEBUG) System.out.println("DrawingPanel shutdown hook: instances=" + instances);
                                        try {
                                                String filename = System.getProperty(SAVE_PROPERTY);
                                                if (filename == null) {
                                                        filename = callingClassName + ".png";
                                                }

                                                if (isAnimated()) {
                                                        saveAnimated(filename);
                                                } else {
                                                        save(filename);
                                                }
                                        } catch (SecurityException e) {
                                                System.err.println("Security error while saving image: " + e);
                                        } catch (IOException e) {
                                                System.err.println("Error saving image: " + e);
                                        }
                                }
                        }));
                } catch (Exception e) {
                        if (DEBUG) System.out.println("DrawingPanel(): unable to add shutdown hook: " + e);
                }
        }
}

/**
 * Constructs a drawing panel that displays the image from the given file enclosed in a window.
 * The panel will be sized exactly to fit the image inside it.
 * @param imageFile the image file to load
 * @throws RuntimeException if the image file is not found
 */
public DrawingPanel(File imageFile) {
        this(imageFile.toString());
}

/**
 * Constructs a drawing panel that displays the image from the given file name enclosed in a window.
 * The panel will be sized exactly to fit the image inside it.
 * @param imageFileName the file name/path of the image file to load
 * @throws RuntimeException if the image file is not found
 */
public DrawingPanel(String imageFileName) {
        this();
        Image image = loadImage(imageFileName);
        setSize(image.getWidth(this), image.getHeight(this));
        getGraphics().drawImage(image, 0, 0, this);
}

/**
 * Adds the given event listener to respond to key events on this panel.
 * @param listener the key event listener to attach
 */
public void addKeyListener(KeyListener listener) {
        ensureNotNull("listener", listener);
```

```
                frame.addKeyListener(listener);
                panel.setFocusable(false);
                frame.requestFocusInWindow();
                frame.requestFocus();
        }

        /**
         * Adds the given event listener to respond to mouse events on this panel.
         * @param listener the mouse event listener to attach
         */
        public void addMouseListener(MouseListener listener) {
                ensureNotNull("listener", listener);
                panel.addMouseListener(listener);
                if (listener instanceof MouseMotionListener) {
                        panel.addMouseMotionListener((MouseMotionListener) listener);
                }
        }

        /**
         * Adds the given event listener to respond to mouse events on this panel.
         */
//      public void addMouseListener(MouseMotionListener listener) {
//          panel.addMouseMotionListener(listener);
//          if (listener instanceof MouseListener) {
//              panel.addMouseListener((MouseListener) listener);
//          }
//      }

//      /**
//       * Adds the given event listener to respond to mouse events on this panel.
//       */
//      public void addMouseListener(MouseInputListener listener) {
//          addMouseListener((MouseListener) listener);
//      }

        /*
         * Whether the panel should automatically switch to animated mode
         * if it calls the sleep method.
         */
        private boolean autoEnableAnimationOnSleep() {
                return propertyIsTrue(AUTO_ENABLE_ANIMATION_ON_SLEEP_PROPERTY);
        }

        /*
         * Moves the given JFrame to the center of the screen.
         */
        private void center(Window frame) {
                Toolkit tk = Toolkit.getDefaultToolkit();
                Dimension screen = tk.getScreenSize();
                int x = Math.max(0, (screen.width - frame.getWidth()) / 2);
                int y = Math.max(0, (screen.height - frame.getHeight()) / 2);
                frame.setLocation(x, y);
        }

        /*
         * Constructs and initializes our JFileChooser field if necessary.
         */
        private void checkChooser() {
                if (chooser == null) {
                        chooser = new JFileChooser();
                        try {
                                chooser.setCurrentDirectory(new File(System.getProperty("user.dir")));
                        } catch (Exception e) {
                                // empty
                        }
                        chooser.setMultiSelectionEnabled(false);
                        chooser.setFileFilter(new DPFileFilter());
                }
        }

        /**
         * Erases all drawn shapes/lines/colors from the panel.
         */
        public void clear() {
                int[] pixels = new int[width * height];
                for (int i = 0; i < pixels.length; i++) {
                        pixels[i] = initialPixel;
                }
                image.setRGB(0, 0, width, height, pixels, 0, 1);
        }

        /*
         * Compares the current DrawingPanel image to an image file on disk.
         */
        private void compareToFile() {
                // save current image to a temp file
                try {
```

```
                String tempFile = saveToTempFile();

                // use file chooser dialog to find image to compare against
                checkChooser();
                if (chooser.showOpenDialog(frame) != JFileChooser.APPROVE_OPTION) {
                        return;
                }

                // user chose a file; let's diff it
                new DiffImage(chooser.getSelectedFile().toString(), tempFile);
        } catch (IOException ioe) {
                JOptionPane.showMessageDialog(frame,
                                                                "Unable to compare images: \n" + ioe);
        }
}

/*
 * Compares the current DrawingPanel image to an image file on the web.
 */
private void compareToURL() {
        // save current image to a temp file
        try {
                String tempFile = saveToTempFile();

                // get list of images to compare against from web site
                URL url = new URL(COURSE_WEB_SITE);
                Scanner input = new Scanner(url.openStream());
                List<String> lines = new ArrayList<String>();
                List<String> filenames = new ArrayList<String>();
                while (input.hasNextLine()) {
                        String line = input.nextLine().trim();
                        if (line.length() == 0) { continue; }

                        if (line.startsWith("#")) {
                                // a comment
                                if (line.endsWith(":")) {
                                        // category label
                                        lines.add(line);
                                        line = line.replaceAll("#\\s*", "");
                                        filenames.add(line);
                                }
                        } else {
                                lines.add(line);

                                // get filename
                                int lastSlash = line.lastIndexOf('/');
                                if (lastSlash >= 0) {
                                        line = line.substring(lastSlash + 1);
                                }

                                // remove extension
                                int dot = line.lastIndexOf('.');
                                if (dot >= 0) {
                                        line = line.substring(0, dot);
                                }

                                filenames.add(line);
                        }
                }
                input.close();

                if (filenames.isEmpty()) {
                        JOptionPane.showMessageDialog(frame,
                                "No valid web files found to compare against.",
                                "Error: no web files found",
                                JOptionPane.ERROR_MESSAGE);
                        return;
                } else {
                        String fileURL = null;
                        if (filenames.size() == 1) {
                                // only one choice; take it
                                fileURL = lines.get(0);
                        } else {
                                // user chooses file to compare against
                                int choice = showOptionDialog(frame, "File to compare against?",
                                                "Choose File", filenames.toArray(new String[0]));
                                if (choice < 0) {
                                        return;
                                }

                                // user chose a file; let's diff it
                                fileURL = lines.get(choice);
                        }
                        new DiffImage(fileURL, tempFile);
                }
        } catch (NoRouteToHostException nrthe) {
                JOptionPane.showMessageDialog(frame, "You do not appear to have a working internet connection.\nPlease check your
```

```
internet settings and try again.\n\n" + nrthe);
                } catch (UnknownHostException uhe) {
                        JOptionPane.showMessageDialog(frame, "Internet connection error: \n" + uhe);
                } catch (SocketException se) {
                        JOptionPane.showMessageDialog(frame, "Internet connection error: \n" + se);
                } catch (IOException ioe) {
                        JOptionPane.showMessageDialog(frame, "Unable to compare images: \n" + ioe);
                }
        }

        /*
         * Closes the DrawingPanel and exits the program.
         */
        private void exit() {
                if (isGraphical()) {
                        frame.setVisible(false);
                        frame.dispose();
                }
                try {
                        System.exit(0);
                } catch (SecurityException e) {
                        // if we're running in an applet or something, can't do System.exit
                }
        }

        /*
         * Returns a best guess about the name of the class that constructed this panel.
         */
        private String getCallingClassName() {
                StackTraceElement[] stack = new RuntimeException().getStackTrace();
                String className = this.getClass().getName();
                for (StackTraceElement element : stack) {
                        String cl = element.getClassName();
                        if (!className.equals(cl)) {
                                className = cl;
                                break;
                        }
                }

                return className;
        }

        /**
         * Returns a map of counts of occurrences of calls of various drawing methods.
         * You can print this map to see how many times your graphics methods have
         * been called to aid in debugging.
         * @return map of {method name, count} pairs
         */
        public Map<String, Integer> getCounts() {
                return Collections.unmodifiableMap(counts);
        }

        /**
         * A variation of getGraphics that returns an object that records
         * a count for various drawing methods.
         * See also: getCounts
         * @return debug Graphics object
         */
        public Graphics getDebuggingGraphics() {
                if (g3 == null) {
                        g3 = new DebuggingGraphics();
                }
                return g3;
        }

        /**
         * Obtain the Graphics object to draw on the panel.
         * @return panel's Graphics object
         */
        public Graphics2D getGraphics() {
                return g2;
        }

        /*
         * Creates the buffered image for drawing on this panel.
         */
        private BufferedImage getImage() {
                // create second image so we get the background color
                BufferedImage image2;
                if (isAnimated()) {
                        image2 = new BufferedImage(width, height, BufferedImage.TYPE_BYTE_INDEXED);
                } else {
                        image2 = new BufferedImage(width, height, image.getType());
                }
                Graphics g = image2.getGraphics();
                // if (DEBUG) System.out.println("DrawingPanel getImage setting background to " + backgroundColor);
                g.setColor(backgroundColor);
```

```
        g.fillRect(0, 0, width, height);
        g.drawImage(image, 0, 0, panel);
        return image2;
}

/**
 * Returns the drawing panel's height in pixels.
 * @return drawing panel's height in pixels
 */
public int getHeight() {
        return height;
}

/**
 * Returns the color of the pixel at the given x/y coordinate as a Color object.
 * If nothing has been explicitly drawn on this particular pixel, the panel's
 * background color is returned.
 * @param x x-coordinate of pixel to retrieve
 * @param y y-coordinate of pixel to retrieve
 * @return pixel (x, y) color as a Color object
 * @throws IllegalArgumentException if (x, y) is out of range
 */
public Color getPixel(int x, int y) {
        int rgb = getPixelRGB(x, y);
        if (getAlpha(rgb) == 0) {
                return backgroundColor;
        } else {
                return new Color(rgb, /* hasAlpha */ true);
        }
}

/**
 * Returns the color of the pixel at the given x/y coordinate as an RGB integer.
 * The individual red, green, and blue components of the RGB integer can be
 * extracted from this by calling DrawingPanel.getRed, getGreen, and getBlue.
 * If nothing has been explicitly drawn on this particular pixel, the panel's
 * background color is returned.
 * See also: getPixel.
 * @param x x-coordinate of pixel to retrieve
 * @param y y-coordinate of pixel to retrieve
 * @return pixel (x, y) color as an RGB integer
 * @throws IllegalArgumentException if (x, y) is out of range
 */
public int getPixelRGB(int x, int y) {
        ensureInRange("x", x, 0, getWidth() - 1);
        ensureInRange("y", y, 0, getHeight() - 1);
        int rgb = image.getRGB(x, y);
        if (getAlpha(rgb) == 0) {
                return backgroundColor.getRGB();
        } else {
                return rgb;
        }
}

/**
 * Returns the colors of all pixels in this DrawingPanel as a 2-D array
 * of Color objects.
 * The first index of the array is the y-coordinate, and the second index
 * is the x-coordinate.  So, for example, index [r][c] represents the RGB
 * pixel data for the pixel at position (x=c, y=r).
 * @return 2D array of colors (row-major)
 */
public Color[][] getPixels() {
        Color[][] pixels = new Color[getHeight()][getWidth()];
        for (int row = 0; row < pixels.length; row++) {
                for (int col = 0; col < pixels[0].length; col++) {
                        // note axis inversion; x/y => col/row
                        pixels[row][col] = getPixel(col, row);
                }
        }
        return pixels;
}

/**
 * Returns the colors of all pixels in this DrawingPanel as a 2-D array
 * of RGB integers.
 * The first index of the array is the y-coordinate, and the second index
 * is the x-coordinate.  So, for example, index [r][c] represents the RGB
 * pixel data for the pixel at position (x=c, y=r).
 * The individual red, green, and blue components of each RGB integer can be
 * extracted from this by calling DrawingPanel.getRed, getGreen, and getBlue.
 * @return 2D array of RGB integers (row-major)
 */
public int[][] getPixelsRGB() {
        int[][] pixels = new int[getHeight()][getWidth()];
        int backgroundRGB = backgroundColor.getRGB();
        for (int row = 0; row < pixels.length; row++) {
```

```
                for (int col = 0; col < pixels[0].length; col++) {
                        // note axis inversion; x/y => col/row
                        int px = image.getRGB(col, row);
                        if (getAlpha(px) == 0) {
                                pixels[row][col] = backgroundRGB;
                        } else {
                                pixels[row][col] = px;
                        }
                }
        }
        return pixels;
}

/**
 * Returns the drawing panel's pixel size (width, height) as a Dimension object.
 * @return panel's size
 */
public Dimension getSize() {
        return new Dimension(width, height);
}

/**
 * Returns the drawing panel's width in pixels.
 * @return panel's width
 */
public int getWidth() {
        return width;
}

/**
 * Returns the drawing panel's x-coordinate on the screen.
 * @return panel's x-coordinate
 */
public int getX() {
        if (isGraphical()) {
                return frame.getX();
        } else {
                return 0;
        }
}

/**
 * Returns the drawing panel's y-coordinate on the screen.
 * @return panel's y-coordinate
 */
public int getY() {
        if (isGraphical()) {
                return frame.getY();
        } else {
                return 0;
        }
}

/**
 * Returns the drawing panel's current zoom factor.
 * Initially this is 1 to indicate 100% zoom, the original size.
 * A factor of 2 would indicate 200% zoom, and so on.
 * @return zoom factor (default 1)
 */
public int getZoom() {
        return currentZoom;
}


/**
 * Internal method;
 * notifies the panel when images are loaded and updated.
 * This is a required method of ImageObserver interface.
 * This is an internal method not meant to be called by clients.
 * @param img internal method; do not call
 * @param infoflags internal method; do not call
 * @param x internal method; do not call
 * @param y internal method; do not call
 * @param width internal method; do not call
 * @param height internal method; do not call
 */
@Override
public boolean imageUpdate(Image img, int infoflags, int x, int y, int width, int height) {
        if (imagePanel != null) {
                imagePanel.imageUpdate(img, infoflags, x, y, width, height);
        }
        return false;
}

/*
 * Sets up state for drawing and saving frames of animation to a GIF image.
 */
```

```
private void initializeAnimation() {
        frames = new ArrayList<ImageFrame>();
        encoder = new Gif89Encoder();
        /*
        try {
                if (hasProperty(SAVE_PROPERTY)) {
                        stream = new FileOutputStream(System.getProperty(SAVE_PROPERTY));
                }
                // encoder.startEncoding(stream);
        } catch (IOException e) {
                System.out.println(e);
        }
        */
}

/*
 * Returns whether this drawing panel is in animation mode.
 */
private boolean isAnimated() {
        return animated || propertyIsTrue(ANIMATED_PROPERTY);
}

/*
 * Returns whether this drawing panel is going to be displayed on screen.
 * This is almost always true except in some server environments where
 * the DrawingPanel is run 'headless' without a GUI, often for scripting
 * and automation purposes.
 */
private boolean isGraphical() {
        return !hasProperty(SAVE_PROPERTY) && !isHeadless();
}

/*
 * Returns true if the drawing panel class is in multiple mode.
 * This would be true if the current program pops up several drawing panels
 * and we want to save the state of each of them to a different file.
 */
private boolean isMultiple() {
        return propertyIsTrue(MULTIPLE_PROPERTY);
}

/**
 * Loads an image from the given file on disk and returns it
 * as an Image object.
 * @param file the file to load
 * @return loaded image object
 * @throws NullPointerException if filename is null
 * @throws RuntimeException if the given file is not found
 */
public Image loadImage(File file) {
        ensureNotNull("file", file);
        return loadImage(file.toString());
}

/**
 * Loads an image from the given file on disk and returns it
 * as an Image object.
 * @param filename name/path of the file to load
 * @return loaded image object
 * @throws NullPointerException if filename is null
 * @throws RuntimeException if the given file is not found
 */
public Image loadImage(String filename) {
        ensureNotNull("filename", filename);
        if (!(new File(filename)).exists()) {
                throw new RuntimeException("DrawingPanel.loadImage: File not found: " + filename);
        }
        Image img = Toolkit.getDefaultToolkit().getImage(filename);
        MediaTracker mt = new MediaTracker(imagePanel == null ? new JPanel() : imagePanel);
        mt.addImage(img, 0);
        try {
                mt.waitForID(0);
        } catch (InterruptedException ie) {
                // empty
        }
        return img;
}

/**
 * Adds an event handler for mouse clicks.
 * You can pass a lambda function here to be called when a mouse click event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onClick(DPMouseEventHandler e) {
        onMouseClick(e);
}
```

```java
/**
 * Adds an event handler for mouse drags.
 * You can pass a lambda function here to be called when a mouse drag event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onDrag(DPMouseEventHandler e) {
        onMouseDrag(e);
}

/**
 * Adds an event handler for mouse enters.
 * You can pass a lambda function here to be called when a mouse enter event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onEnter(DPMouseEventHandler e) {
        onMouseEnter(e);
}

/**
 * Adds an event handler for mouse exits.
 * You can pass a lambda function here to be called when a mouse exit event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onExit(DPMouseEventHandler e) {
        onMouseExit(e);
}

/**
 * Adds an event handler for key presses.
 * You can pass a lambda function here to be called when a key press event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onKeyDown(DPKeyEventHandler e) {
        ensureNotNull("event handler", e);
        DPKeyEventHandlerAdapter adapter = new DPKeyEventHandlerAdapter(e, "press");
        addKeyListener(adapter);
}

/**
 * Adds an event handler for key releases.
 * You can pass a lambda function here to be called when a key release event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onKeyUp(DPKeyEventHandler e) {
        ensureNotNull("event handler", e);
        DPKeyEventHandlerAdapter adapter = new DPKeyEventHandlerAdapter(e, "release");
        addKeyListener(adapter);
}

/**
 * Adds an event handler for mouse clicks.
 * You can pass a lambda function here to be called when a mouse click event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onMouseClick(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "click");
        addMouseListener((MouseListener) adapter);
}

/**
 * Adds an event handler for mouse button down events.
 * You can pass a lambda function here to be called when a mouse button down event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onMouseDown(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "press");
        addMouseListener((MouseListener) adapter);
}

/**
 * Adds an event handler for mouse drags.
 * You can pass a lambda function here to be called when a mouse drag event occurs.
 * @param e event handler function to call
 * @throws NullPointerException if event handler is null
 */
public void onMouseDrag(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
```

```
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "drag");
        addMouseListener((MouseListener) adapter);
    }

    /**
     * Adds an event handler for mouse enters.
     * You can pass a lambda function here to be called when a mouse enter event occurs.
     * @param e event handler function to call
     * @throws NullPointerException if event handler is null
     */
    public void onMouseEnter(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "enter");
        addMouseListener((MouseListener) adapter);
    }

    /**
     * Adds an event handler for mouse exits.
     * You can pass a lambda function here to be called when a mouse exit event occurs.
     * @param e event handler function to call
     * @throws NullPointerException if event handler is null
     */
    public void onMouseExit(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "exit");
        addMouseListener((MouseListener) adapter);
    }

    /**
     * Adds an event handler for mouse movement.
     * You can pass a lambda function here to be called when a mouse move event occurs.
     * @param e event handler function to call
     * @throws NullPointerException if event handler is null
     */
    public void onMouseMove(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "move");
        addMouseListener((MouseListener) adapter);
    }

    /**
     * Adds an event handler for mouse button up events.
     * You can pass a lambda function here to be called when a mouse button up event occurs.
     * @param e event handler function to call
     * @throws NullPointerException if event handler is null
     */
    public void onMouseUp(DPMouseEventHandler e) {
        ensureNotNull("event handler", e);
        DPMouseEventHandlerAdapter adapter = new DPMouseEventHandlerAdapter(e, "release");
        addMouseListener((MouseListener) adapter);
    }

    /**
     * Adds an event handler for mouse movement.
     * You can pass a lambda function here to be called when a mouse move event occurs.
     * @param e event handler function to call
     * @throws NullPointerException if event handler is null
     */
    public void onMove(DPMouseEventHandler e) {
        onMouseMove(e);
    }

    /*
     * Returns whether the drawing panel should be closed and the program
     * should be shut down.
     */
    private boolean readyToClose() {
/*
        if (isAnimated()) {
            // wait a little longer, in case animation is sleeping
            return System.currentTimeMillis() > createTime + 5 * DELAY;
        } else {
            return System.currentTimeMillis() > createTime + 4 * DELAY;
        }
*/
        return (instances == 0 || shouldSave()) && !mainIsActive();
    }

    /*
     * Replaces all occurrences of the given old color with the given new color.
     */
    private void replaceColor(BufferedImage image, Color oldColor, Color newColor) {
        int oldRGB = oldColor.getRGB();
        int newRGB = newColor.getRGB();
        for (int y = 0; y < image.getHeight(); y++) {
            for (int x = 0; x < image.getWidth(); x++) {
                if (image.getRGB(x, y) == oldRGB) {
```

```
                                image.setRGB(x, y, newRGB);
                        }
                }
        }
}

/**
 * Takes the current contents of the drawing panel and writes them to
 * the given file.
 * @param file the file to save
 * @throws NullPointerException if filename is null
 * @throws IOException if the given file cannot be written
 */
public void save(File file) throws IOException {
        ensureNotNull("file", file);
        save(file.toString());
}


/**
 * Takes the current contents of the drawing panel and writes them to
 * the given file.
 * @param filename name/path of the file to save
 * @throws NullPointerException if filename is null
 * @throws IOException if the given file cannot be written
 */
public void save(String filename) throws IOException {
        ensureNotNull("filename", filename);
        BufferedImage image2 = getImage();

        // if zoomed, scale image before saving it
        if (SAVE_SCALED_IMAGES && currentZoom != 1) {
                BufferedImage zoomedImage = new BufferedImage(width * currentZoom, height * currentZoom, image.getType());
                Graphics2D g = (Graphics2D) zoomedImage.getGraphics();
                g.setColor(Color.BLACK);
                if (antialias) {
                        g.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
                }
                g.scale(currentZoom, currentZoom);
                g.drawImage(image2, 0, 0, imagePanel);
                image2 = zoomedImage;
        }

        // if saving multiple panels, append number
        // (e.g. output_*.png becomes output_1.png, output_2.png, etc.)
        if (isMultiple()) {
                filename = filename.replaceAll("\\*", String.valueOf(instanceNumber));
        }

        int lastDot = filename.lastIndexOf(".");
        String extension = filename.substring(lastDot + 1);

        // write file
        // (for some reason, NPEs throw sometimes for no reason; just squish them)
        try {
                ImageIO.write(image2, extension, new File(filename));
        } catch (NullPointerException npe) {
                // empty
        } catch (FileNotFoundException fnfe) {
                // this is a dumb file overwrite issue related to file locking; ignore
        }

        hasBeenSaved = true;
}

/**
 * Takes the current contents of the drawing panel and writes them to
 * the given file.
 * @param file the file to save
 * @throws NullPointerException if filename is null
 * @throws IOException if the given file cannot be written
 */
public void saveAnimated(File file) throws IOException {
        ensureNotNull("file", file);
        saveAnimated(file.toString());
}

/**
 * Takes the current contents of the drawing panel and writes them to
 * the given file.
 * @param filename name/path of the file to save
 * @throws NullPointerException if filename is null
 * @throws IOException if the given file cannot be written
 */
public void saveAnimated(String filename) throws IOException {
        ensureNotNull("filename", filename);
```

```
        // add one more final frame
        if (DEBUG) System.out.println("DrawingPanel.saveAnimated(" + filename + ")");
        frames.add(new ImageFrame(getImage(), 5000));
        // encoder.continueEncoding(stream, getImage(), 5000);

        // Gif89Encoder gifenc = new Gif89Encoder();

        // add each frame of animation to the encoder
        try {
                for (int i = 0; i < frames.size(); i++) {
                        ImageFrame imageFrame = frames.get(i);
                        encoder.addFrame(imageFrame.image);
                        encoder.getFrameAt(i).setDelay(imageFrame.delay);
                        imageFrame.image.flush();
                        frames.set(i, null);
                }
        } catch (OutOfMemoryError e) {
                System.out.println("Out of memory when saving");
        }

        // gifenc.setComments(annotation);
        // gifenc.setUniformDelay((int) Math.round(100 / frames_per_second));
        // gifenc.setUniformDelay(DELAY);
        // encoder.setBackground(backgroundColor);
        encoder.setLoopCount(0);
        encoder.encode(new FileOutputStream(filename));
}

/*
 * Called when the user presses the "Save As" menu item.
 * Pops up a file chooser prompting the user to save their panel to an image.
 */
private void saveAs() {
        String filename = saveAsHelper("png");
        if (filename != null) {
                try {
                        save(filename);  // save the file
                } catch (IOException ex) {
                        JOptionPane.showMessageDialog(frame, "Unable to save image:\n" + ex);
                }
        }
}

/*
 * Called when the user presses the "Save As" menu item on an animated panel.
 * Pops up a file chooser prompting the user to save their panel to an image.
 */
private void saveAsAnimated() {
        String filename = saveAsHelper("gif");
        if (filename != null) {
                try {
                        // record that the file should be saved next time
                        PrintStream out = new PrintStream(new File(ANIMATION_FILE_NAME));
                        out.println(filename);
                        out.close();

                        JOptionPane.showMessageDialog(frame,
                                "Due to constraints about how DrawingPanel works, you'll need to\n" +
                                "re-run your program.  When you run it the next time, DrawingPanel will \n" +
                                "automatically save your animated image as: " + new File(filename).getName()
                        );
                } catch (IOException ex) {
                        JOptionPane.showMessageDialog(frame, "Unable to store animation settings:\n" + ex);
                }
        }
}

/*
 * A helper method to facilitate the Save As action for both animated
 * and non-animated images.
 */
private String saveAsHelper(String extension) {
        // use file chooser dialog to get filename to save into
        checkChooser();
        if (chooser.showSaveDialog(frame) != JFileChooser.APPROVE_OPTION) {
                return null;
        }

        File selectedFile = chooser.getSelectedFile();
        String filename = selectedFile.toString();
        if (!filename.toLowerCase().endsWith(extension)) {
                // Windows is dumb about extensions with file choosers
                filename += "." + extension;
        }

        // confirm overwrite of file
        if (new File(filename).exists() && JOptionPane.showConfirmDialog(
```

```
                        frame, "File exists.  Overwrite?", "Overwrite?",
                        JOptionPane.YES_NO_OPTION) != JOptionPane.YES_OPTION) {
                return null;
        }

        return filename;
}

/*
 * Saves the drawing panel's image to a temporary file and returns
 * that file's name.
 */
private String saveToTempFile() throws IOException {
        File currentImageFile = File.createTempFile("current_image", ".png");
        save(currentImageFile.toString());
        return currentImageFile.toString();
}

/**
 * Sets whether the panel will always cover other windows (default false).
 * @param alwaysOnTop true if the panel should always cover other windows
 */
public void setAlwaysOnTop(boolean alwaysOnTop) {
        if (frame != null) {
                frame.setAlwaysOnTop(alwaysOnTop);
        }
}

/**
 * Sets whether the panel should use anti-aliased / smoothed graphics (default true).
 * @param antiAlias true if the panel should be smoothed
 */
public void setAntiAlias(boolean antiAlias) {
        this.antialias = antiAlias;
        Object value = antiAlias ? RenderingHints.VALUE_ANTIALIAS_ON : RenderingHints.VALUE_ANTIALIAS_OFF;
        if (g2 != null) {
                g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, value);
        }
        if (imagePanel != null) {
                imagePanel.repaint();
        }
}

/**
 * Sets the background color of the drawing panel to be the given color.
 * @param c color to use as background
 * @throws NullPointerException if color is null
 */
public void setBackground(Color c) {
        ensureNotNull("color", c);
        Color oldBackgroundColor = backgroundColor;
        backgroundColor = c;
        if (isGraphical()) {
                panel.setBackground(c);
                imagePanel.setBackground(c);
        }

        // with animated images, need to palette-swap the old bg color for the new
        // because there's no notion of transparency in a palettized 8-bit image
        if (isAnimated()) {
                replaceColor(image, oldBackgroundColor, c);
        }
}

/**
 * Sets the background color of the drawing panel to be the color
 * represented by the given RGB integer.
 * @param rgb RGB integer to use as background color (full alpha assumed/applied)
 */
public void setBackground(int rgb) {
        setBackground(new Color(rgb & 0xff000000, /* hasAlpha */ true));
}

/**
 * Enables or disables the drawing of grid lines on top of the image to help
 * with debugging sizes and coordinates.
 * By default the grid lines will be shown every 10 pixels in each dimension.
 * @param gridLines whether to show grid lines (true) or not (false)
 */
public void setGridLines(boolean gridLines) {
        setGridLines(gridLines, GRID_LINES_PX_GAP_DEFAULT);
}

/**
 * Enables or disables the drawing of grid lines on top of the image to help
 * with debugging sizes and coordinates.
 * The grid lines will be shown every pxGap pixels in each dimension.
```

```
     * @param gridLines whether to show grid lines (true) or not (false)
     * @param pxGap number of pixels between grid lines
     */
    public void setGridLines(boolean gridLines, int pxGap) {
            this.gridLines = gridLines;
            this.gridLinesPxGap = pxGap;
            if (imagePanel != null) {
                    imagePanel.repaint();
            }
    }

    /**
     * Sets the drawing panel's height in pixels to the given value.
     * After calling this method, the client must call getGraphics() again
     * to get the new graphics context of the newly enlarged image buffer.
     * @param height height, in pixels
     * @throws IllegalArgumentException if height is negative or exceeds MAX_SIZE
     */
    public void setHeight(int height) {
            setSize(getWidth(), height);
    }

    /**
     * Sets the color of the pixel at the given x/y coordinate to be the given color.
     * If the color is null, the call has no effect.
     * @param x x-coordinate of pixel to set
     * @param y y-coordinate of pixel to set
     * @param color Color to set the pixel to use
     * @throws IllegalArgumentException if x or y is out of bounds
     * @throws NullPointerException if color is null
     */
    public void setPixel(int x, int y, Color color) {
            ensureInRange("x", x, 0, getWidth() - 1);
            ensureInRange("y", y, 0, getHeight() - 1);
            ensureNotNull("color", color);
            image.setRGB(x, y, color.getRGB());
    }

    /**
     * Sets the color of the pixel at the given x/y coordinate to be the color
     * represented by the given RGB integer.
     * The passed RGB integer's alpha value is ignored and a full alpha of 255
     * is always used here, to avoid common bugs with using a 0 value for alpha.
     * See also: setPixel.
     * See also: setPixelRGB.
     * @param x x-coordinate of pixel to set
     * @param y y-coordinate of pixel to set
     * @param rgb RGB integer representing the color to set the pixel to use
     * @throws IllegalArgumentException if x or y is out of bounds
     */
    public void setPixel(int x, int y, int rgb) {
            setPixelRGB(x, y, rgb);
    }

    /**
     * Sets the color of the pixel at the given x/y coordinate to be the color
     * represented by the given RGB integer.
     * The passed RGB integer's alpha value is ignored and a full alpha of 255
     * is always used here, to avoid common bugs with using a 0 value for alpha.
     * See also: setPixel.
     * @param x x-coordinate of pixel to set
     * @param y y-coordinate of pixel to set
     * @param rgb RGB integer representing the color to set the pixel to use
     * @throws IllegalArgumentException if x or y is out of bounds
     */
    public void setPixelRGB(int x, int y, int rgb) {
            ensureInRange("x", x, 0, getWidth() - 1);
            ensureInRange("y", y, 0, getHeight() - 1);
            image.setRGB(x, y, rgb | PIXEL_ALPHA);
    }

    /**
     * Sets the colors of all pixels in this DrawingPanel to the colors
     * in the given 2-D array of Color objects.
     * The first index of the array is the y-coordinate, and the second index
     * is the x-coordinate.  So, for example, index [r][c] represents the RGB
     * pixel data for the pixel at position (x=c, y=r).
     * If the given array's dimensions do not match the width/height of the
     * drawing panel, the panel is resized to match the array.
     * If the pixel array is null or size 0, the call has no effect.
     * If any rows or colors in the array are null, those pixels will be ignored.
     * The 2-D array passed is assumed to be rectangular in length (not jagged).
     * @param pixels 2D array of pixels (row-major)
     * @throws NullPointerException if pixels array is null
     */
    public void setPixels(Color[][] pixels) {
            ensureNotNull("pixels", pixels);
```

```
        if (pixels != null && pixels.length > 0 && pixels[0] != null) {
                if (width != pixels[0].length || height != pixels.length) {
                        setSize(pixels[0].length, pixels.length);
                }
                for (int row = 0; row < height; row++) {
                        if (pixels[row] != null) {
                                for (int col = 0; col < width; col++) {
                                        if (pixels[row][col] != null) {
                                                int rgb = pixels[row][col].getRGB();
                                                image.setRGB(col, row, rgb);
                                        }
                                }
                        }
                }
        }
}

/**
 * Sets the colors of all pixels in this DrawingPanel to the colors
 * represented by the given 2-D array of RGB integers.
 * The first index of the array is the y-coordinate, and the second index
 * is the x-coordinate.  So, for example, index [r][c] represents the RGB
 * pixel data for the pixel at position (x=c, y=r).
 * If the given array's dimensions do not match the width/height of the
 * drawing panel, the panel is resized to match the array.
 * If the pixel array is null or size 0, the call has no effect.
 * The 2-D array passed is assumed to be rectangular in length (not jagged).
 * @param pixels 2D array of pixels (row-major)
 * @throws NullPointerException if pixels array is null
 */
public void setPixels(int[][] pixels) {
        setPixelsRGB(pixels);
}

/**
 * Sets the colors of all pixels in this DrawingPanel to the colors
 * represented by the given 2-D array of RGB integers.
 * The first index of the array is the y-coordinate, and the second index
 * is the x-coordinate.  So, for example, index [r][c] represents the RGB
 * pixel data for the pixel at position (x=c, y=r).
 * If the given array's dimensions do not match the width/height of the
 * drawing panel, the panel is resized to match the array.
 * If the pixel array is null or size 0, the call has no effect.
 * The 2-D array passed is assumed to be rectangular in length (not jagged).
 * @param pixels 2D array of pixels (row-major)
 * @throws NullPointerException if pixels array is null
 */
public void setPixelsRGB(int[][] pixels) {
        ensureNotNull("pixels", pixels);
        if (pixels != null && pixels.length > 0 && pixels[0] != null) {
                if (width != pixels[0].length || height != pixels.length) {
                        setSize(pixels[0].length, pixels.length);
                }
                for (int row = 0; row < height; row++) {
                        if (pixels[row] != null) {
                                for (int col = 0; col < width; col++) {
                                        // note axis inversion, row/col => y/x
                                        image.setRGB(col, row, pixels[row][col] | PIXEL_ALPHA);
                                }
                        }
                }
        }
}

/**
 * Sets the drawing panel's pixel size (width, height) to the given values.
 * After calling this method, the client must call getGraphics() again
 * to get the new graphics context of the newly enlarged image buffer.
 * @param width width, in pixels
 * @param height height, in pixels
 * @throws IllegalArgumentException if width/height is negative or exceeds MAX_SIZE
 */
public void setSize(int width, int height) {
        ensureInRange("width", width, 0, MAX_SIZE);
        ensureInRange("height", height, 0, MAX_SIZE);

        // replace the image buffer for drawing
        BufferedImage newImage = new BufferedImage(width, height, image.getType());
        if (imagePanel != null) {
                imagePanel.setImage(newImage);
        }
        newImage.getGraphics().drawImage(image, 0, 0, imagePanel == null ? new JPanel() : imagePanel);

        this.width = width;
        this.height = height;
        image = newImage;
        g2 = (Graphics2D) newImage.getGraphics();
```

```
        g2.setColor(Color.BLACK);
        if (antialias) {
                g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING, RenderingHints.VALUE_ANTIALIAS_ON);
        }
        zoom(currentZoom);
        if (isGraphical()) {
                frame.pack();
        }
}

/*
 * Sets the text that will appear in the drawing panel's bottom status bar.
 */
private void setStatusBarText(String text) {
        if (currentZoom != 1) {
                text += " (current zoom: " + currentZoom + "x" + ")";
        }
        statusBar.setText(text);
}

/*
 * Initializes the drawing panel's menu bar items.
 */
private void setupMenuBar() {
        // abort compare if we're running as an applet or in a secure environment
        // boolean secure = (System.getSecurityManager() != null);

    // for now, assume non-secure mode since DrawingPanel applet usage is minimal
    final boolean secure = false;

        JMenuItem saveAs = new JMenuItem("Save As...", 'A');
        saveAs.addActionListener(actionListener);
        saveAs.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
        saveAs.setEnabled(!secure);

        JMenuItem saveAnimated = new JMenuItem("Save Animated GIF...", 'G');
        saveAnimated.addActionListener(actionListener);
        saveAnimated.setAccelerator(KeyStroke.getKeyStroke("ctrl A"));
        saveAnimated.setEnabled(!secure);

        JMenuItem compare = new JMenuItem("Compare to File...", 'C');
        compare.addActionListener(actionListener);
        compare.setEnabled(!secure);

        JMenuItem compareURL = new JMenuItem("Compare to Web File...", 'U');
        compareURL.addActionListener(actionListener);
        compareURL.setAccelerator(KeyStroke.getKeyStroke("ctrl U"));
        compareURL.setEnabled(!secure);

        JMenuItem zoomIn = new JMenuItem("Zoom In", 'I');
        zoomIn.addActionListener(actionListener);
        zoomIn.setAccelerator(KeyStroke.getKeyStroke("ctrl EQUALS"));

        JMenuItem zoomOut = new JMenuItem("Zoom Out", 'O');
        zoomOut.addActionListener(actionListener);
        zoomOut.setAccelerator(KeyStroke.getKeyStroke("ctrl MINUS"));

        JMenuItem zoomNormal = new JMenuItem("Zoom Normal (100%)", 'N');
        zoomNormal.addActionListener(actionListener);
        zoomNormal.setAccelerator(KeyStroke.getKeyStroke("ctrl 0"));

        JCheckBoxMenuItem gridLinesItem = new JCheckBoxMenuItem("Grid Lines");
        gridLinesItem.setMnemonic('G');
        gridLinesItem.setSelected(gridLines);
        gridLinesItem.addActionListener(actionListener);
        gridLinesItem.setAccelerator(KeyStroke.getKeyStroke("ctrl G"));

        JMenuItem exit = new JMenuItem("Exit", 'x');
        exit.addActionListener(actionListener);

        JMenuItem about = new JMenuItem("About...", 'A');
        about.addActionListener(actionListener);

        JMenu file = new JMenu("File");
        file.setMnemonic('F');
        file.add(compareURL);
        file.add(compare);
        file.addSeparator();
        file.add(saveAs);
        file.add(saveAnimated);
        file.addSeparator();
        file.add(exit);

        JMenu view = new JMenu("View");
        view.setMnemonic('V');
        view.add(zoomIn);
        view.add(zoomOut);
```

```
        view.add(zoomNormal);
        view.addSeparator();
        view.add(gridLinesItem);

        JMenu help = new JMenu("Help");
        help.setMnemonic('H');
        help.add(about);

        JMenuBar bar = new JMenuBar();
        bar.add(file);
        bar.add(view);
        bar.add(help);
        frame.setJMenuBar(bar);
}

/**
 * Show or hide the drawing panel on the screen.
 * @param visible true to show, false to hide
 */
public void setVisible(boolean visible) {
        if (isGraphical()) {
                frame.setVisible(visible);
        }
}

/**
 * Sets the drawing panel's width in pixels to the given value.
 * After calling this method, the client must call getGraphics() again
 * to get the new graphics context of the newly enlarged image buffer.
 * @param width width, in pixels
 * @throws IllegalArgumentException if height is negative or exceeds MAX_SIZE
 */
public void setWidth(int width) {
        ensureInRange("width", width, 0, MAX_SIZE);
        setSize(width, getHeight());
}

/*
 * Returns whether the user wants to perform a 'diff' comparison of their
 * drawing panel with a given expected output image.
 */
private boolean shouldDiff() {
        return hasProperty(DIFF_PROPERTY);
}

/*
 * Returns whether the user wants to save the drawing panel contents to
 * a file automatically.
 */
private boolean shouldSave() {
        return hasProperty(SAVE_PROPERTY);
}

/*
 * Shows a dialog box with the given choices;
 * returns the index chosen (-1 == canceled).
 */
private int showOptionDialog(Frame parent, String title,
                String message, final String[] names) {
        final JDialog dialog = new JDialog(parent, title, true);
        JPanel center = new JPanel(new GridLayout(0, 1));

        // just a hack to make the return value a mutable reference to an int
        final int[] hack = {-1};

        for (int i = 0; i < names.length; i++) {
                if (names[i].endsWith(":")) {
                        center.add(new JLabel("<html><b>" + names[i] + "</b></html>"));
                } else {
                        final JButton button = new JButton(names[i]);
                        button.setActionCommand(String.valueOf(i));
                        button.addActionListener(new ActionListener() {
                                public void actionPerformed(ActionEvent e) {
                                        hack[0] = Integer.parseInt(button.getActionCommand());
                                        dialog.setVisible(false);
                                }
                        });
                        center.add(button);
                }
        }

        JPanel south = new JPanel();
        JButton cancel = new JButton("Cancel");
        cancel.setMnemonic('C');
        cancel.requestFocus();
        cancel.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
```

```
                        dialog.setVisible(false);
                }
        });
        south.add(cancel);

        dialog.setDefaultCloseOperation(JDialog.DO_NOTHING_ON_CLOSE);
        dialog.getContentPane().setLayout(new BorderLayout(10, 5));

        if (message != null) {
                JLabel messageLabel = new JLabel(message);
                dialog.add(messageLabel, BorderLayout.NORTH);
        }
        dialog.add(center);
        dialog.add(south, BorderLayout.SOUTH);
        dialog.pack();
        dialog.setResizable(false);
        center(dialog);
        cancel.requestFocus();
        dialog.setVisible(true);
        cancel.requestFocus();

        return hack[0];
}

/**
 * Causes the program to pause for the given amount of time in milliseconds.
 * This allows for animation by calling pause in a loop.
 * If the DrawingPanel is not showing on the screen, has no effect.
 * @param millis number of milliseconds to sleep
 * @throws IllegalArgumentException if a negative number of ms is passed
 */
public void sleep(int millis) {
        ensureInRange("millis", millis, 0, Integer.MAX_VALUE);
        if (isGraphical() && frame.isVisible()) {
                // if not even displaying, we don't actually need to sleep
                if (millis > 0) {
                        try {
                                Thread.sleep(millis);
                                panel.repaint();
                                // toFront(frame);
                        } catch (Exception e) {
                                // empty
                        }
                }
        }

        // manually enable animation if necessary
        if (!isAnimated() && !isMultiple() && autoEnableAnimationOnSleep()) {
                animated = true;
                initializeAnimation();
        }

        // capture a frame of animation
        if (isAnimated() && shouldSave() && !isMultiple()) {
                try {
                        if (frames.size() < MAX_FRAMES) {
                                frames.add(new ImageFrame(getImage(), millis));
                        }

                        // reset creation timer so that we won't save/close just yet
                        createTime = System.currentTimeMillis();
                } catch (OutOfMemoryError e) {
                        System.out.println("Out of memory after capturing " + frames.size() + " frames");
                }
        }
}

/**
 * Moves the drawing panel window on top of other windows so it can be seen.
 */
public void toFront() {
        toFront(frame);
}

/*
 * Brings the given window to the front of the Z-ordering.
 */
private void toFront(final Window window) {
        // TODO: remove anonymous inner class
        EventQueue.invokeLater(new Runnable() {
                public void run() {
                        if (window != null) {
                                window.toFront();
                                window.repaint();
                        }
                }
        });
```

```
	}

	/**
	 * Zooms the drawing panel in/out to the given factor.
	 * A zoom factor of 1, the default, indicates normal size.
	 * A zoom factor of 2 would indicate 200% size, and so on.
	 * The factor value passed should be at least 1; if not, 1 will be used.
	 * @param zoomFactor the zoom factor to use (1 or greater)
	 */
	public void zoom(int zoomFactor) {
		currentZoom = Math.max(1, zoomFactor);
		if (isGraphical()) {
			Dimension size = new Dimension(width * currentZoom, height * currentZoom);
			imagePanel.setPreferredSize(size);
			panel.setPreferredSize(size);
			imagePanel.validate();
			imagePanel.revalidate();
			panel.validate();
			panel.revalidate();
			// imagePanel.setSize(size);
			frame.getContentPane().validate();
			imagePanel.repaint();
			setStatusBarText(" ");

			// resize frame if any more space for it exists or it's the wrong size
			Dimension screen = Toolkit.getDefaultToolkit().getScreenSize();
			if (size.width <= screen.width || size.height <= screen.height) {
				frame.pack();
			}

			if (currentZoom != 1) {
				frame.setTitle(TITLE + " (" + currentZoom + "x zoom)");
			} else {
				frame.setTitle(TITLE);
			}
		}
	}

	// INNER/NESTED CLASSES

	/*
	 * Internal action listener for handling events on buttons and GUI components.
	 */
	private class DPActionListener implements ActionListener {
		// used for an internal timer that keeps repainting
		public void actionPerformed(ActionEvent e) {
			if (e.getSource() instanceof Timer) {
				// redraw the screen at regular intervals to catch all paint operations
				panel.repaint();
				if (shouldDiff() &&
					System.currentTimeMillis() > createTime + 4 * DELAY) {
					String expected = System.getProperty(DIFF_PROPERTY);
					try {
						String actual = saveToTempFile();
						DiffImage diff = new DiffImage(expected, actual);
						diff.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
					} catch (IOException ioe) {
						System.err.println("Error diffing image: " + ioe);
					}
					timer.stop();
				} else if (shouldSave() && readyToClose()) {
					// auto-save-and-close if desired
					try {
						if (isAnimated()) {
							saveAnimated(System.getProperty(SAVE_PROPERTY));
						} else {
							save(System.getProperty(SAVE_PROPERTY));
						}
					} catch (IOException ioe) {
						System.err.println("Error saving image: " + ioe);
					}
					exit();
				}
			} else if (e.getActionCommand().equals("Exit")) {
				exit();
			} else if (e.getActionCommand().equals("Compare to File...")) {
				compareToFile();
			} else if (e.getActionCommand().equals("Compare to Web File...")) {
				new Thread(new Runnable() {
					public void run() {
						compareToURL();
					}
				}).start();
			} else if (e.getActionCommand().equals("Save As...")) {
				saveAs();
			} else if (e.getActionCommand().equals("Save Animated GIF...")) {
				saveAsAnimated();
```

```
                } else if (e.getActionCommand().equals("Zoom In")) {
                        zoom(currentZoom + 1);
                } else if (e.getActionCommand().equals("Zoom Out")) {
                        zoom(currentZoom - 1);
                } else if (e.getActionCommand().equals("Zoom Normal (100%)")) {
                        zoom(1);
                } else if (e.getActionCommand().equals("Grid Lines")) {
                        setGridLines(((JCheckBoxMenuItem) e.getSource()).isSelected());
                } else if (e.getActionCommand().equals("About...")) {
                        JOptionPane.showMessageDialog(frame,
                                        ABOUT_MESSAGE,
                                        ABOUT_MESSAGE_TITLE,
                                        JOptionPane.INFORMATION_MESSAGE);
                }
        }
}

/*
 * Internal file filter class for showing image files in JFileChooser.
 */
private class DPFileFilter extends FileFilter {
        public boolean accept(File file) {
                return file.isDirectory() ||
                        (file.getName().toLowerCase().endsWith(".png") ||
                         file.getName().toLowerCase().endsWith(".gif"));
        }

        public String getDescription() {
                return "Image files (*.png; *.gif)";
        }
}

// BEGIN EVENT ADAPTER CODE FOR JAVA 8 FUNCTIONAL INTERFACE CLIENTS
// EXAMPLE:
// panel.onClick( (x, y) -> System.out.println(x + " " + y) );

/**
 * This functional interface is provided to allow Java 8 clients to write
 * lambda functions to handle mouse events that occur in a DrawingPanel.
 */
@FunctionalInterface
public static interface DPMouseEventHandler {
        /**
         * Called when a mouse event occurs at the given (x, y) position
         * in the drawing panel window.
         * @param x x-coordinate at which the event occurred
         * @param y y-coordinate at which the event occurred
         */
        public void onMouseEvent(int x, int y);
}

/**
 * This functional interface is provided to allow Java 8 clients to write
 * lambda functions to handle key events that occur in a DrawingPanel.
 */
@FunctionalInterface
public static interface DPKeyEventHandler {
        /**
         * Called when a key event occurs involving the given key character
         * in the drawing panel window.
         * @param keyCode char value that was typed
         */
        public void onKeyEvent(char keyCode);
}

// internal class to implement DPKeyEventHandler behavior.
private class DPKeyEventHandlerAdapter implements KeyListener {
        private DPKeyEventHandler handler;
        private String eventType;

        /**
         * Constructs a new key handler adapter.
         * @param handler event handler function
         * @param eventType type of event to print
         */
        public DPKeyEventHandlerAdapter(DPKeyEventHandler handler, String eventType) {
                this.handler = handler;
                this.eventType = eventType.intern();
        }

        /**
         * Called when a key press occurs.
         * @param e event that occurred
         */
        @Override
        public void keyPressed(KeyEvent e) {
                // empty; see keyTyped
```

```
        }

        /**
         * Called when a key release occurs.
         * @param e event that occurred
         */
        @Override
        public void keyReleased(KeyEvent e) {
                if (eventType == "release") {
                        int keyCode = e.getKeyCode();
                        if (keyCode < ' ') {
                                return;
                        }
                        handler.onKeyEvent(e.getKeyChar());
                }
        }

        /**
         * Called when a key type event occurs.
         * @param e event that occurred
         */
        @Override
        public void keyTyped(KeyEvent e) {
                if (eventType == "press") {
                        handler.onKeyEvent(e.getKeyChar());
                }
        }
}

// internal class to implement DPMouseEventHandler behavior.
private class DPMouseEventHandlerAdapter implements MouseInputListener {
        private DPMouseEventHandler handler;
        private String eventType;

        /**
         * Constructs a new mouse handler adapter.
         * @param handler event handler function
         * @param eventType type of event to print
         */
        public DPMouseEventHandlerAdapter(DPMouseEventHandler handler, String eventType) {
                this.handler = handler;
                this.eventType = eventType.intern();
        }

        /**
         * Called when a mouse press occurs.
         * @param e event that occurred
         */
        @Override
        public void mousePressed(MouseEvent e) {
                if (eventType == "press") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }

        /**
         * Called when a mouse release occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseReleased(MouseEvent e) {
                if (eventType == "release") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }

        /**
         * Called when a mouse click occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseClicked(MouseEvent e) {
                if (eventType == "click") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }

        /**
         * Called when a mouse enter occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseEntered(MouseEvent e) {
                if (eventType == "enter") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }
```

```
        /**
         * Called when a mouse exit occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseExited(MouseEvent e) {
                if (eventType == "exit") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }

        /**
         * Called when a mouse movement occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseMoved(MouseEvent e) {
                if (eventType == "move") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }

        /**
         * Called when a mouse drag occurs.
         * @param e event that occurred
         */
        @Override
        public void mouseDragged(MouseEvent e) {
                if (eventType == "drag") {
                        handler.onMouseEvent(e.getX(), e.getY());
                }
        }
}

// END EVENT ADAPTER CODE FOR JAVA 8 FUNCTIONAL INTERFACE CLIENTS

// Internal MouseListener class for handling mouse events in the panel.
private class DPMouseListener extends MouseInputAdapter {
        // listens to mouse movement
        public void mouseMoved(MouseEvent e) {
                int x = e.getX() / currentZoom;
                int y = e.getY() / currentZoom;
                String status = "(x=" + x + ", y=" + y + ")";
                if (x >= 0 && x < width && y >= 0 && y < height) {
                        int rgb = getPixelRGB(x, y);
                        int r = getRed(rgb);
                        int g = getGreen(rgb);
                        int b = getBlue(rgb);
                        status += ", r=" + r + " g=" + g + " b=" + b;
                }
                setStatusBarText(status);
        }
}

// Internal WindowListener class for handling window events in the panel.
private class DPWindowListener extends WindowAdapter {
        // called when DrawingPanel closes, to potentially exit the program
        public void windowClosing(WindowEvent event) {
                frame.setVisible(false);
                synchronized (LOCK) {
                        instances--;
                }
                frame.dispose();
        }
}

/*
 * This inner class passes through calls to the panel's Graphics object g2
 * but also records a count of how many times various basic drawing methods
 * are called. This is used for debugging purposes, so that a client can
 * compare their counts of various graphical method calls to those from an
 * expected output as a "sanity check" on their program's behavior.
 * Notice that it extends Graphics and not Graphics2D, so it is more limited
 * than g2.
 * @author Stuart Reges
 */
private class DebuggingGraphics extends Graphics {
        public Graphics create() {
                return g2.create();
        }

        public void translate(int x, int y) {
                g2.translate(x, y);
        }

        public Color getColor() {
```

```
        return g2.getColor();
}

public void setPaintMode() {
        g2.setPaintMode();
}

public void setXORMode(Color c1) {
        g2.setXORMode(c1);
}

public Font getFont() {
        return g2.getFont();
}

public void setFont(Font font) {
        g2.setFont(font);
}

public FontMetrics getFontMetrics(Font f) {
        return g2.getFontMetrics();
}

public Rectangle getClipBounds() {
        return g2.getClipBounds();
}

public void clipRect(int x, int y, int width, int height) {
        g2.clipRect(x, y, width, height);
}

public void setClip(int x, int y, int width, int height) {
        g2.setClip(x, y, width, height);
}

public Shape getClip() {
        return g2.getClip();
}

public void setClip(Shape clip) {
        g2.setClip(clip);
}

public void copyArea(int x, int y, int width, int height, int dx, int dy) {
        g2.copyArea(x, y, width, height, dx, dy);
}

public void clearRect(int x, int y, int width, int height) {
        g2.clearRect(x, y, width, height);
}

public void drawRoundRect(int x, int y, int width, int height,
                int arcWidth, int arcHeight) {
        g2.drawRoundRect(x, y, width, height, arcWidth, arcHeight);
}

public void fillRoundRect(int x, int y, int width, int height,
                int arcWidth, int arcHeight) {
        g2.fillRoundRect(x, y, width, height, arcWidth, arcHeight);
}

public void drawArc(int x, int y, int width, int height,
                int startAngle, int arcAngle) {
        g2.drawArc(x, y, width, height, startAngle, arcAngle);
}

public void fillArc(int x, int y, int width, int height,
                int startAngle, int arcAngle) {
        g2.fillArc(x, y, width, height, startAngle, arcAngle);
}

public void drawPolyline(int xPoints[], int yPoints[], int nPoints) {
        g2.drawPolyline(xPoints, yPoints, nPoints);
}

public void drawPolygon(int xPoints[], int yPoints[], int nPoints) {
        g2.drawPolygon(xPoints, yPoints, nPoints);
}

public void fillPolygon(int xPoints[], int yPoints[], int nPoints) {
        g2.fillPolygon(xPoints, yPoints, nPoints);
}

public void drawString(AttributedCharacterIterator iterator, int x,
                int y) {
        g2.drawString(iterator, x, y);
}
```

```
        public boolean drawImage(Image img, int x, int y, ImageObserver observer) {
                return g2.drawImage(img, x, y, observer);
        };

        public boolean drawImage(Image img, int x, int y, int width,
                        int height, ImageObserver observer) {
                return g2.drawImage(img, x, y, width, height, observer);
        };

        public boolean drawImage(Image img, int x, int y, Color bgcolor,
                        ImageObserver observer) {
                return g2.drawImage(img, x, y, bgcolor, observer);
        };

        public boolean drawImage(Image img, int x, int y, int width,
                        int height, Color bgcolor, ImageObserver observer) {
                return g2.drawImage(img, x, y, width, height, bgcolor, observer);
        }

        public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2,
                        int sx1, int sy1, int sx2, int sy2, ImageObserver observer) {
                return g2.drawImage(img, dx1, dy1, dx2, dy2, sx1, dy1, dx2, sy2,
                                observer);
        }

        public boolean drawImage(Image img, int dx1, int dy1, int dx2, int dy2,
                        int sx1, int sy1, int sx2, int sy2, Color bgcolor,
                        ImageObserver observer) {
                return g2.drawImage(img, dx1, dy1, dx2, dy2, sx1, dy1, sx2, sy2,
                                bgcolor, observer);
        }

        public void dispose() {
                g2.dispose();
        }

        public void drawOval(int x, int y, int width, int height) {
                g2.drawOval(x, y, width, height);
                recordString("drawOval");
        }

        public void fillOval(int x, int y, int width, int height) {
                g2.fillOval(x, y, width, height);
                recordString("fillOval");
        }

        public void drawString(String str, int x, int y) {
                g2.drawString(str, x, y);
                recordString("drawString");
        }

        public void drawLine(int x1, int y1, int x2, int y2) {
                g2.drawLine(x1, y1, x2, y2);
                recordString("drawLine");
        }

        public void fillRect(int x, int y, int width, int height) {
                g2.fillRect(x, y, width, height);
                recordString("fillRect");
        }

        public void drawRect(int x, int y, int width, int height) {
                g2.drawRect(x, y, width, height);
                recordString("drawRect");
        }

        public void setColor(Color c) {
                g2.setColor(c);
                // recordString("setColor");
        }

        public void recordString(String s) {
                if (!counts.containsKey(s)) {
                        counts.put(s, 1);
                } else {
                        counts.put(s, counts.get(s) + 1);
                }
        }
} // end class DebuggingGraphics

/*
 * This internal class represents a graphical panel that can pop up on the
 * screen to report the differences between two images.
 * It is used to allow the client to compare their program's output against
 * a known correct output and view which pixels differ between the two.
 */
```

```
	private class DiffImage extends JPanel
			implements ActionListener, ChangeListener {
		private static final long serialVersionUID = 0;

		private BufferedImage image1;
		private BufferedImage image2;
		private String image1name;
		private int numDiffPixels;
		private int opacity = 50;
		private String label1Text = "Expected";
		private String label2Text = "Actual";
		private boolean highlightDiffs = false;

		private Color highlightColor = new Color(224, 0, 224);
		private JLabel image1Label;
		private JLabel image2Label;
		private JLabel diffPixelsLabel;
		private JSlider slider;
		private JCheckBox box;
		private JMenuItem saveAsItem;
		private JMenuItem setImage1Item;
		private JMenuItem setImage2Item;
		private JFrame frame;
		private JButton colorButton;

		public DiffImage(String file1, String file2) throws IOException {
			setImage1(file1);
			setImage2(file2);
			display();
		}

		public void actionPerformed(ActionEvent e) {
			Object source = e.getSource();
			if (source == box) {
				highlightDiffs = box.isSelected();
				repaint();
			} else if (source == colorButton) {
				Color color = JColorChooser.showDialog(frame,
												"Choose highlight color",
highlightColor);
				if (color != null) {
					highlightColor = color;
					colorButton.setBackground(color);
					colorButton.setForeground(color);
					repaint();
				}
			} else if (source == saveAsItem) {
				saveAs();
			} else if (source == setImage1Item) {
				setImage1();
			} else if (source == setImage2Item) {
				setImage2();
			}
		}

		// Counts number of pixels that differ between the two images.
		public void countDiffPixels() {
			if (image1 == null || image2 == null) {
				return;
			}

			int w1 = image1.getWidth();
			int h1 = image1.getHeight();
			int w2 = image2.getWidth();
			int h2 = image2.getHeight();
			int wmax = Math.max(w1, w2);
			int hmax = Math.max(h1, h2);

			// check each pair of pixels
			numDiffPixels = 0;
			for (int y = 0; y < hmax; y++) {
				for (int x = 0; x < wmax; x++) {
					int pixel1 = (x < w1 && y < h1) ? image1.getRGB(x, y) : 0;
					int pixel2 = (x < w2 && y < h2) ? image2.getRGB(x, y) : 0;
					if (pixel1 != pixel2) {
						numDiffPixels++;
					}
				}
			}
		}

		// initializes diffimage panel
		public void display() {
			countDiffPixels();

			setupComponents();
			setupEvents();
```

```
                setupLayout();

                frame.pack();
                center(frame);

                frame.setVisible(true);
                toFront(frame);
        }

        // draws the given image onto the given graphics context
        public void drawImageFull(Graphics2D g2, BufferedImage image) {
                int iw = image.getWidth();
                int ih = image.getHeight();
                int w = getWidth();
                int h = getHeight();
                int dw = w - iw;
                int dh = h - ih;

                if (dw > 0) {
                        g2.fillRect(iw, 0, dw, ih);
                }
                if (dh > 0) {
                        g2.fillRect(0, ih, iw, dh);
                }
                if (dw > 0 && dh > 0) {
                        g2.fillRect(iw, ih, dw, dh);
                }
                g2.drawImage(image, 0, 0, this);
        }

        // paints the DiffImage panel
        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2 = (Graphics2D) g;

                // draw the expected output (image 1)
                if (image1 != null) {
                        drawImageFull(g2, image1);
                }

                // draw the actual output (image 2)
                if (image2 != null) {
                        Composite oldComposite = g2.getComposite();
                        g2.setComposite(AlphaComposite.getInstance(AlphaComposite.SRC_ATOP, ((float) opacity) / 100));
                        drawImageFull(g2, image2);
                        g2.setComposite(oldComposite);
                }
                g2.setColor(Color.BLACK);

                // draw the highlighted diffs (if so desired)
                if (highlightDiffs && image1 != null && image2 != null) {
                        int w1 = image1.getWidth();
                        int h1 = image1.getHeight();
                        int w2 = image2.getWidth();
                        int h2 = image2.getHeight();

                        int wmax = Math.max(w1, w2);
                        int hmax = Math.max(h1, h2);

                        // check each pair of pixels
                        g2.setColor(highlightColor);
                        for (int y = 0; y < hmax; y++) {
                                for (int x = 0; x < wmax; x++) {
                                        int pixel1 = (x < w1 && y < h1) ? image1.getRGB(x, y) : 0;
                                        int pixel2 = (x < w2 && y < h2) ? image2.getRGB(x, y) : 0;
                                        if (pixel1 != pixel2) {
                                                g2.fillRect(x, y, 1, 1);
                                        }
                                }
                        }
                }
        }

        public void save(File file) throws IOException {
                // String extension = filename.substring(filename.lastIndexOf(".") + 1);
                // ImageIO.write(diffImage, extension, new File(filename));
                String filename = file.getName();
                String extension = filename.substring(filename.lastIndexOf(".") + 1);
                BufferedImage img = new BufferedImage(getPreferredSize().width, getPreferredSize().height, BufferedImage.TYPE_INT_ARGB
);
                img.getGraphics().setColor(getBackground());
                img.getGraphics().fillRect(0, 0, img.getWidth(), img.getHeight());
                paintComponent(img.getGraphics());
                ImageIO.write(img, extension, file);
        }

        public void save(String filename) throws IOException {
```

```
        save(new File(filename));
}

// Called when "Save As" menu item is clicked
public void saveAs() {
        checkChooser();
        if (chooser.showSaveDialog(frame) != JFileChooser.APPROVE_OPTION) {
                return;
        }

        File selectedFile = chooser.getSelectedFile();
        try {
                save(selectedFile.toString());
        } catch (IOException ex) {
                JOptionPane.showMessageDialog(frame, "Unable to save image:\n" + ex);
        }
}

// called when "Set Image 1" menu item is clicked
public void setImage1() {
        checkChooser();
        if (chooser.showSaveDialog(frame) != JFileChooser.APPROVE_OPTION) {
                return;
        }

        File selectedFile = chooser.getSelectedFile();
        try {
                setImage1(selectedFile.toString());
                countDiffPixels();
                diffPixelsLabel.setText("(" + numDiffPixels + " pixels differ)");
                image1Label.setText(selectedFile.getName());
                frame.pack();
        } catch (IOException ex) {
                JOptionPane.showMessageDialog(frame, "Unable to set image 1:\n" + ex);
        }
}

// sets image 1 to be the given image
public void setImage1(BufferedImage image) {
        if (image == null) {
                throw new NullPointerException();
        }

        image1 = image;
        setPreferredSize(new Dimension(
                                                                Math.max(getPreferredSize().width, image.getWidth()),
                                                                Math.max(getPreferredSize().height, image.getHeight()))
                                                );
        if (frame != null) {
                frame.pack();
        }
        repaint();
}

// loads image 1 from the given filename or URL
public void setImage1(String filename) throws IOException {
        image1name = new File(filename).getName();
        if (filename.startsWith("http")) {
                setImage1(ImageIO.read(new URL(filename)));
        } else {
                setImage1(ImageIO.read(new File(filename)));
        }
}

// called when "Set Image 2" menu item is clicked
public void setImage2() {
        checkChooser();
        if (chooser.showSaveDialog(frame) != JFileChooser.APPROVE_OPTION) {
                return;
        }

        File selectedFile = chooser.getSelectedFile();
        try {
                setImage2(selectedFile.toString());
                countDiffPixels();
                diffPixelsLabel.setText("(" + numDiffPixels + " pixels differ)");
                image2Label.setText(selectedFile.getName());
                frame.pack();
        } catch (IOException ex) {
                JOptionPane.showMessageDialog(frame, "Unable to set image 2:\n" + ex);
        }
}

// sets image 2 to be the given image
public void setImage2(BufferedImage image) {
        if (image == null) {
                throw new NullPointerException();
```

```
        }

        image2 = image;
        setPreferredSize(new Dimension(
                                                                Math.max(getPreferredSize().width, image.getWidth()),
                                                                Math.max(getPreferredSize().height, image.getHeight()))
                                                );
        if (frame != null) {
                frame.pack();
        }
        repaint();
}

// loads image 2 from the given filename
public void setImage2(String filename) throws IOException {
        if (filename.startsWith("http")) {
                setImage2(ImageIO.read(new URL(filename)));
        } else {
                setImage2(ImageIO.read(new File(filename)));
        }

}

private void setupComponents() {
        String title = "DiffImage";
        if (image1name != null) {
                title = "Compare to " + image1name;
        }
        frame = new JFrame(title);
        frame.setResizable(false);
        // frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        slider = new JSlider();
        slider.setPaintLabels(false);
        slider.setPaintTicks(true);
        slider.setSnapToTicks(true);
        slider.setMajorTickSpacing(25);
        slider.setMinorTickSpacing(5);

        box = new JCheckBox("Highlight diffs in color: ", highlightDiffs);

        colorButton = new JButton();
        colorButton.setBackground(highlightColor);
        colorButton.setForeground(highlightColor);
        colorButton.setPreferredSize(new Dimension(24, 24));

        diffPixelsLabel = new JLabel("(" + numDiffPixels + " pixels differ)");
        diffPixelsLabel.setFont(diffPixelsLabel.getFont().deriveFont(Font.BOLD));
        image1Label = new JLabel(label1Text);
        image2Label = new JLabel(label2Text);

        setupMenuBar();
}

// initializes layout of components
private void setupLayout() {
        JPanel southPanel1 = new JPanel();
        southPanel1.setBorder(BorderFactory.createLineBorder(Color.DARK_GRAY));
        southPanel1.add(image1Label);
        southPanel1.add(slider);
        southPanel1.add(image2Label);
        southPanel1.add(Box.createHorizontalStrut(20));

        JPanel southPanel2 = new JPanel();
        southPanel2.setBorder(BorderFactory.createLineBorder(Color.DARK_GRAY));
        southPanel2.add(diffPixelsLabel);
        southPanel2.add(Box.createHorizontalStrut(20));
        southPanel2.add(box);
        southPanel2.add(colorButton);

        Container southPanel = javax.swing.Box.createVerticalBox();
        southPanel.add(southPanel1);
        southPanel.add(southPanel2);

        frame.add(this, BorderLayout.CENTER);
        frame.add(southPanel, BorderLayout.SOUTH);
}

// initializes main menu bar
private void setupMenuBar() {
        saveAsItem = new JMenuItem("Save As...", 'A');
        saveAsItem.setAccelerator(KeyStroke.getKeyStroke("ctrl S"));
        setImage1Item = new JMenuItem("Set Image 1...", '1');
        setImage1Item.setAccelerator(KeyStroke.getKeyStroke("ctrl 1"));
        setImage2Item = new JMenuItem("Set Image 2...", '2');
        setImage2Item.setAccelerator(KeyStroke.getKeyStroke("ctrl 2"));
```

```
                JMenu file = new JMenu("File");
                file.setMnemonic('F');
                file.add(setImage1Item);
                file.add(setImage2Item);
                file.addSeparator();
                file.add(saveAsItem);

                JMenuBar bar = new JMenuBar();
                bar.add(file);

                // disabling menu bar to simplify code
                // frame.setJMenuBar(bar);
        }

        // method of ChangeListener interface
        public void stateChanged(ChangeEvent e) {
                opacity = slider.getValue();
                repaint();
        }

        // adds event listeners to various components
        private void setupEvents() {
                slider.addChangeListener(this);
                box.addActionListener(this);
                colorButton.addActionListener(this);
                saveAsItem.addActionListener(this);
                this.setImage1Item.addActionListener(this);
                this.setImage2Item.addActionListener(this);
        }
}

// inner class to represent one frame of an animated GIF
private static class ImageFrame {
        public Image image;
        public int delay;

        public ImageFrame(Image image, int delay) {
                this.image = image;
                this.delay = delay / 10;   // strangely, gif stores delay as sec/100
        }
}

// inner class to do the actual drawing onto the DrawingPanel
private class ImagePanel extends JPanel {
        private static final long serialVersionUID = 0;
        private Image image;

        // constructs the image panel
        public ImagePanel(Image image) {
                super(/* isDoubleBuffered */ true);
                setImage(image);
                setBackground(Color.WHITE);
                setPreferredSize(new Dimension(image.getWidth(this), image.getHeight(this)));
                setAlignmentX(0.0f);
        }

        // draws everything onto the panel
        public void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2 = (Graphics2D) g;
                if (currentZoom != 1) {
                        g2.scale(currentZoom, currentZoom);
                }
                g2.drawImage(image, 0, 0, this);

                // possibly draw grid lines for debugging
                if (gridLines) {
                        g2.setPaint(GRID_LINE_COLOR);
                        for (int row = 1; row <= getHeight() / gridLinesPxGap; row++) {
                                g2.drawLine(0, row * gridLinesPxGap, getWidth(), row * gridLinesPxGap);
                        }
                        for (int col = 1; col <= getWidth() / gridLinesPxGap; col++) {
                                g2.drawLine(col * gridLinesPxGap, 0, col * gridLinesPxGap, getHeight());
                        }
                }
        }

        public void setImage(Image image) {
                this.image = image;
                repaint();
        }
}

// BEGIN GIF ENCODING CLASSES

//**************************************************************************
// DirectGif89Frame.java
```

```
//////////////////////////////////////////////////////////////////////////////

// ===========================================================================
/**
 * Instances of this Gif89Frame subclass are constructed from RGB image
 * info, either in the form of an Image object or a pixel array.
 * <p>
 * There is an important restriction to note. It is only permissible to add
 * DirectGif89Frame objects to a Gif89Encoder constructed without an
 * explicit color map. The GIF color table will be automatically generated
 * from pixel information.
 *
 * @version 0.90 beta (15-Jul-2000)
 * @author J. M. G. Elliott (tep@jmge.net)
 * @see Gif89Encoder
 * @see Gif89Frame
 * @see IndexGif89Frame
 */
class DirectGif89Frame extends Gif89Frame {

        private int[] argbPixels;

        // -----------------------------------------------------------------------
        /**
         * Construct an DirectGif89Frame from a Java image.
         *
         * @param img
         *            A java.awt.Image object that supports pixel-grabbing.
         * @exception IOException
         *                If the image is unencodable due to failure of
         *                pixel-grabbing.
         */
        public DirectGif89Frame(Image img) throws IOException {
                PixelGrabber pg = new PixelGrabber(img, 0, 0, -1, -1, true);

                String errmsg = null;
                try {
                        if (!pg.grabPixels())
                                errmsg = "can't grab pixels from image";
                } catch (InterruptedException e) {
                        errmsg = "interrupted grabbing pixels from image";
                }

                if (errmsg != null)
                        throw new IOException(errmsg + " (" + getClass().getName()
                                        + ")");

                theWidth = pg.getWidth();
                theHeight = pg.getHeight();
                argbPixels = (int[]) pg.getPixels();
                ciPixels = new byte[argbPixels.length];

                // flush to conserve resources
                img.flush();
        }

        // -----------------------------------------------------------------------
        /**
         * Construct an DirectGif89Frame from ARGB pixel data.
         *
         * @param width
         *            Width of the bitmap.
         * @param height
         *            Height of the bitmap.
         * @param argb_pixels
         *            Array containing at least width*height pixels in the
         *            format returned by java.awt.Color.getRGB().
         */
        public DirectGif89Frame(int width, int height, int argb_pixels[]) {
                theWidth = width;
                theHeight = height;
                argbPixels = new int[theWidth * theHeight];
                System.arraycopy(argb_pixels, 0, argbPixels, 0, argbPixels.length);
                ciPixels = new byte[argbPixels.length];
        }

        // -----------------------------------------------------------------------
        Object getPixelSource() {
                return argbPixels;
        }
}

// ***************************************************************************
// Gif89Encoder.java
// ***************************************************************************

// ===========================================================================
```

```
/**
 * This is the central class of a JDK 1.1 compatible GIF encoder that,
 * AFAIK, supports more features of the extended GIF spec than any other
 * Java open source encoder. Some sections of the source are lifted or
 * adapted from Jef Poskanzer's <cite>Acme GifEncoder</cite> (so please see
 * the <a href="../readme.txt">readme</a> containing his notice), but much
 * of it, including nearly all of the present class, is original code. My
 * main motivation for writing a new encoder was to support animated GIFs,
 * but the package also adds support for embedded textual comments.
 * <p>
 * There are still some limitations. For instance, animations are limited to
 * a single global color table. But that is usually what you want anyway, so
 * as to avoid irregularities on some displays. (So this is not really a
 * limitation, but a "disciplinary feature" :) Another rather more serious
 * restriction is that the total number of RGB colors in a given input-batch
 * mustn't exceed 256. Obviously, there is an opening here for someone who
 * would like to add a color-reducing preprocessor.
 * <p>
 * The encoder, though very usable in its present form, is at bottom only a
 * partial implementation skewed toward my own particular needs. Hence a
 * couple of caveats are in order. (1) During development it was in the back
 * of my mind that an encoder object should be reusable - i.e., you should
 * be able to make multiple calls to encode() on the same object, with or
 * without intervening frame additions or changes to options. But I haven't
 * reviewed the code with such usage in mind, much less tested it, so it's
 * likely I overlooked something. (2) The encoder classes aren't thread
 * safe, so use caution in a context where access is shared by multiple
 * threads. (Better yet, finish the library and re-release it :)
 * <p>
 * There follow a couple of simple examples illustrating the most common way
 * to use the encoder, i.e., to encode AWT Image objects created elsewhere
 * in the program. Use of some of the most popular format options is also
 * shown, though you will want to peruse the API for additional features.
 *
 * <p>
 * <strong>Animated GIF Example</strong>
 *
 * <pre>
 *  import net.jmge.gif.Gif89Encoder;
 *  // ...
 *  void writeAnimatedGIF(Image[] still_images,
 *                        String annotation,
 *                        boolean looped,
 *                        double frames_per_second,
 *                        OutputStream out) throws IOException
 *  {
 *  Gif89Encoder gifenc = new Gif89Encoder();
 *  for (int i = 0; i < still_images.length; ++i)
 *    gifenc.addFrame(still_images[i]);
 *  gifenc.setComments(annotation);
 *  gifenc.setLoopCount(looped ? 0 : 1);
 *  gifenc.setUniformDelay((int) Math.round(100 / frames_per_second));
 *  gifenc.encode(out);
 *  }
 * </pre>
 *
 * <strong>Static GIF Example</strong>
 *
 * <pre>
 *  import net.jmge.gif.Gif89Encoder;
 *  // ...
 *  void writeNormalGIF(Image img,
 *                      String annotation,
 *                      int transparent_index,  // pass -1 for none
 *                      boolean interlaced,
 *                      OutputStream out) throws IOException
 *  {
 *  Gif89Encoder gifenc = new Gif89Encoder(img);
 *  gifenc.setComments(annotation);
 *  gifenc.setTransparentIndex(transparent_index);
 *  gifenc.getFrameAt(0).setInterlaced(interlaced);
 *  gifenc.encode(out);
 *  }
 * </pre>
 *
 * @version 0.90 beta (15-Jul-2000)
 * @author J. M. G. Elliott (tep@jmge.net)
 * @see Gif89Frame
 * @see DirectGif89Frame
 * @see IndexGif89Frame
 */
class Gif89Encoder {
        private static final boolean DEBUG = false;
        private Dimension dispDim = new Dimension(0, 0);
        private GifColorTable colorTable;
        private int bgIndex = 0;
        private int loopCount = 1;
```

```
private String theComments;
private Vector<Gif89Frame> vFrames = new Vector<Gif89Frame>();

// ------------------------------------------------------------------------
/**
 * Use this default constructor if you'll be adding multiple frames
 * constructed from RGB data (i.e., AWT Image objects or ARGB-pixel
 * arrays).
 */
public Gif89Encoder() {
        // empty color table puts us into "palette autodetect" mode
        colorTable = new GifColorTable();
}

// ------------------------------------------------------------------------
/**
 * Like the default except that it also adds a single frame, for
 * conveniently encoding a static GIF from an image.
 *
 * @param static_image
 *            Any Image object that supports pixel-grabbing.
 * @exception IOException
 *                See the addFrame() methods.
 */
public Gif89Encoder(Image static_image) throws IOException {
        this();
        addFrame(static_image);
}

// ------------------------------------------------------------------------
/**
 * This constructor installs a user color table, overriding the
 * detection of of a palette from ARBG pixels.
 *
 * Use of this constructor imposes a couple of restrictions: (1) Frame
 * objects can't be of type DirectGif89Frame (2) Transparency, if
 * desired, must be set explicitly.
 *
 * @param colors
 *            Array of color values; no more than 256 colors will be
 *            read, since that's the limit for a GIF.
 */
public Gif89Encoder(Color[] colors) {
        colorTable = new GifColorTable(colors);
}

// ------------------------------------------------------------------------
/**
 * Convenience constructor for encoding a static GIF from index-model
 * data. Adds a single frame as specified.
 *
 * @param colors
 *            Array of color values; no more than 256 colors will be
 *            read, since that's the limit for a GIF.
 * @param width
 *            Width of the GIF bitmap.
 * @param height
 *            Height of same.
 * @param ci_pixels
 *            Array of color-index pixels no less than width * height in
 *            length.
 * @exception IOException
 *                See the addFrame() methods.
 */
public Gif89Encoder(Color[] colors, int width, int height,
                byte ci_pixels[]) throws IOException {
        this(colors);
        addFrame(width, height, ci_pixels);
}

// ------------------------------------------------------------------------
/**
 * Get the number of frames that have been added so far.
 *
 * @return Number of frame items.
 */
public int getFrameCount() {
        return vFrames.size();
}

// ------------------------------------------------------------------------
/**
 * Get a reference back to a Gif89Frame object by position.
 *
 * @param index
 *            Zero-based index of the frame in the sequence.
 * @return Gif89Frame object at the specified position (or null if no
```

```
     *         such frame).
     */
    public Gif89Frame getFrameAt(int index) {
            return isOk(index) ? vFrames.elementAt(index) : null;
    }

    // ------------------------------------------------------------------------
    /**
     * Add a Gif89Frame frame to the end of the internal sequence. Note that
     * there are restrictions on the Gif89Frame type: if the encoder object
     * was constructed with an explicit color table, an attempt to add a
     * DirectGif89Frame will throw an exception.
     *
     * @param gf
     *            An externally constructed Gif89Frame.
     * @exception IOException
     *                If Gif89Frame can't be accommodated. This could happen
     *                if either (1) the aggregate cross-frame RGB color
     *                count exceeds 256, or (2) the Gif89Frame subclass is
     *                incompatible with the present encoder object.
     */
    public void addFrame(Gif89Frame gf) throws IOException {
            accommodateFrame(gf);
            vFrames.addElement(gf);
    }

    // ------------------------------------------------------------------------
    /**
     * Convenience version of addFrame() that takes a Java Image, internally
     * constructing the requisite DirectGif89Frame.
     *
     * @param image
     *            Any Image object that supports pixel-grabbing.
     * @exception IOException
     *                If either (1) pixel-grabbing fails, (2) the aggregate
     *                cross-frame RGB color count exceeds 256, or (3) this
     *                encoder object was constructed with an explicit color
     *                table.
     */
    public void addFrame(Image image) throws IOException {
            DirectGif89Frame frame = new DirectGif89Frame(image);
            addFrame(frame);
    }

    // ------------------------------------------------------------------------
    /**
     * The index-model convenience version of addFrame().
     *
     * @param width
     *            Width of the GIF bitmap.
     * @param height
     *            Height of same.
     * @param ci_pixels
     *            Array of color-index pixels no less than width * height in
     *            length.
     * @exception IOException
     *                Actually, in the present implementation, there aren't
     *                any unchecked exceptions that can be thrown when
     *                adding an IndexGif89Frame <i>per se</i>. But I might
     *                add some pedantic check later, to justify the
     *                generality :)
     */
    public void addFrame(int width, int height, byte ci_pixels[])
                    throws IOException {
            addFrame(new IndexGif89Frame(width, height, ci_pixels));
    }

    // ------------------------------------------------------------------------
    /**
     * Like addFrame() except that the frame is inserted at a specific point
     * in the sequence rather than appended.
     *
     * @param index
     *            Zero-based index at which to insert frame.
     * @param gf
     *            An externally constructed Gif89Frame.
     * @exception IOException
     *                If Gif89Frame can't be accommodated. This could happen
     *                if either (1) the aggregate cross-frame RGB color
     *                count exceeds 256, or (2) the Gif89Frame subclass is
     *                incompatible with the present encoder object.
     */
    public void insertFrame(int index, Gif89Frame gf) throws IOException {
            accommodateFrame(gf);
            vFrames.insertElementAt(gf, index);
    }
```

```
// --------------------------------------------------------------------------
/**
 * Set the color table index for the transparent color, if any.
 *
 * @param index
 *            Index of the color that should be rendered as transparent,
 *            if any. A value of -1 turns off transparency. (Default:
 *            -1)
 */
public void setTransparentIndex(int index) {
        colorTable.setTransparent(index);
}

// --------------------------------------------------------------------------
/**
 * Sets attributes of the multi-image display area, if applicable.
 *
 * @param dim
 *            Width/height of display. (Default: largest detected frame
 *            size)
 * @param background
 *            Color table index of background color. (Default: 0)
 * @see Gif89Frame#setPosition
 */
public void setLogicalDisplay(Dimension dim, int background) {
        dispDim = new Dimension(dim);
        bgIndex = background;
}

// --------------------------------------------------------------------------
/**
 * Set animation looping parameter, if applicable.
 *
 * @param count
 *            Number of times to play sequence. Special value of 0
 *            specifies indefinite looping. (Default: 1)
 */
public void setLoopCount(int count) {
        loopCount = count;
}

// --------------------------------------------------------------------------
/**
 * Specify some textual comments to be embedded in GIF.
 *
 * @param comments
 *            String containing ASCII comments.
 */
public void setComments(String comments) {
        theComments = comments;
}

// --------------------------------------------------------------------------
/**
 * A convenience method for setting the "animation speed". It simply
 * sets the delay parameter for each frame in the sequence to the
 * supplied value. Since this is actually frame-level rather than
 * animation-level data, take care to add your frames before calling
 * this method.
 *
 * @param interval
 *            Interframe interval in centiseconds.
 */
public void setUniformDelay(int interval) {
        for (int i = 0; i < vFrames.size(); ++i)
                vFrames.elementAt(i).setDelay(interval);
}

// --------------------------------------------------------------------------
/**
 * After adding your frame(s) and setting your options, simply call this
 * method to write the GIF to the passed stream. Multiple calls are
 * permissible if for some reason that is useful to your application.
 * (The method simply encodes the current state of the object with no
 * thought to previous calls.)
 *
 * @param out
 *            The stream you want the GIF written to.
 * @exception IOException
 *                If a write error is encountered.
 */
public void encode(OutputStream out) throws IOException {
        int nframes = getFrameCount();
        boolean is_sequence = nframes > 1;

        // N.B. must be called before writing screen descriptor
        colorTable.closePixelProcessing();
```

```
        // write GIF HEADER
        putAscii("GIF89a", out);

        // write global blocks
        writeLogicalScreenDescriptor(out);
        colorTable.encode(out);
        if (is_sequence && loopCount != 1)
                writeNetscapeExtension(out);
        if (theComments != null && theComments.length() > 0)
                writeCommentExtension(out);

        // write out the control and rendering data for each frame
        for (int i = 0; i < nframes; ++i) {
                DirectGif89Frame frame = (DirectGif89Frame) vFrames
                                .elementAt(i);
                frame.encode(out, is_sequence, colorTable.getDepth(),
                                colorTable.getTransparent());
                vFrames.set(i, null); // for GC's sake
                System.gc();
        }

        // write GIF TRAILER
        out.write((int) ';');

        out.flush();
}

public boolean hasStarted = false;

// ------------------------------------------------------------------------
/**
 * After adding your frame(s) and setting your options, simply call this
 * method to write the GIF to the passed stream. Multiple calls are
 * permissible if for some reason that is useful to your application.
 * (The method simply encodes the current state of the object with no
 * thought to previous calls.)
 *
 * @param out
 *            The stream you want the GIF written to.
 * @exception IOException
 *                If a write error is encountered.
 */
public void startEncoding(OutputStream out, Image image, int delay)
                throws IOException {
        hasStarted = true;
        boolean is_sequence = true;
        Gif89Frame gf = new DirectGif89Frame(image);
        accommodateFrame(gf);

        // N.B. must be called before writing screen descriptor
        colorTable.closePixelProcessing();

        // write GIF HEADER
        putAscii("GIF89a", out);

        // write global blocks
        writeLogicalScreenDescriptor(out);
        colorTable.encode(out);
        if (is_sequence && loopCount != 1)
                writeNetscapeExtension(out);
        if (theComments != null && theComments.length() > 0)
                writeCommentExtension(out);
}

public void continueEncoding(OutputStream out, Image image, int delay)
                throws IOException {
        // write out the control and rendering data for each frame
        Gif89Frame gf = new DirectGif89Frame(image);
        accommodateFrame(gf);
        gf.encode(out, true, colorTable.getDepth(),
                        colorTable.getTransparent());
        out.flush();
        image.flush();
}

public void endEncoding(OutputStream out) throws IOException {
        // write GIF TRAILER
        out.write((int) ';');

        out.flush();
}

public void setBackground(Color color) {
        bgIndex = colorTable.indexOf(color);
        if (bgIndex < 0) {
                try {
```

```
                        BufferedImage img = new BufferedImage(1, 1,
                                        BufferedImage.TYPE_BYTE_INDEXED);
                        Graphics g = img.getGraphics();
                        g.setColor(color);
                        g.fillRect(0, 0, 2, 2);
                        DirectGif89Frame frame = new DirectGif89Frame(img);
                        accommodateFrame(frame);
                        bgIndex = colorTable.indexOf(color);
                } catch (IOException e) {
                        if (DEBUG)
                                System.out
                                                .println("Error while setting background color: "
                                                                + e);
                }
        }
        if (DEBUG)
                System.out.println("Setting bg index to " + bgIndex);
}

// ------------------------------------------------------------------------
private void accommodateFrame(Gif89Frame gf) throws IOException {
        dispDim.width = Math.max(dispDim.width, gf.getWidth());
        dispDim.height = Math.max(dispDim.height, gf.getHeight());
        colorTable.processPixels(gf);
}

// ------------------------------------------------------------------------
private void writeLogicalScreenDescriptor(OutputStream os)
                throws IOException {
        putShort(dispDim.width, os);
        putShort(dispDim.height, os);

        // write 4 fields, packed into a byte (bitfieldsize:value)
        // global color map present? (1:1)
        // bits per primary color less 1 (3:7)
        // sorted color table? (1:0)
        // bits per pixel less 1 (3:varies)
        os.write(0xf0 | colorTable.getDepth() - 1);

        // write background color index
        os.write(bgIndex);

        // Jef Poskanzer's notes on the next field, for our possible
        // edification:
        // Pixel aspect ratio - 1:1.
        // Putbyte( (byte) 49, outs );
        // Java's GIF reader currently has a bug, if the aspect ratio byte
        // is
        // not zero it throws an ImageFormatException. It doesn't know that
        // 49 means a 1:1 aspect ratio. Well, whatever, zero works with all
        // the other decoders I've tried so it probably doesn't hurt.

        // OK, if it's good enough for Jef, it's definitely good enough for
        // us:
        os.write(0);
}

// ------------------------------------------------------------------------
private void writeNetscapeExtension(OutputStream os) throws IOException {
        // n.b. most software seems to interpret the count as a repeat count
        // (i.e., interations beyond 1) rather than as an iteration count
        // (thus, to avoid repeating we have to omit the whole extension)

        os.write((int) '!'); // GIF Extension Introducer
        os.write(0xff); // Application Extension Label

        os.write(11); // application ID block size
        putAscii("NETSCAPE2.0", os); // application ID data

        os.write(3); // data sub-block size
        os.write(1); // a looping flag? dunno

        // we finally write the relevent data
        putShort(loopCount > 1 ? loopCount - 1 : 0, os);

        os.write(0); // block terminator
}

// ------------------------------------------------------------------------
private void writeCommentExtension(OutputStream os) throws IOException {
        os.write((int) '!'); // GIF Extension Introducer
        os.write(0xfe); // Comment Extension Label

        int remainder = theComments.length() % 255;
        int nsubblocks_full = theComments.length() / 255;
        int nsubblocks = nsubblocks_full + (remainder > 0 ? 1 : 0);
        int ibyte = 0;
```

```
                for (int isb = 0; isb < nsubblocks; ++isb) {
                        int size = isb < nsubblocks_full ? 255 : remainder;

                        os.write(size);
                        putAscii(theComments.substring(ibyte, ibyte + size), os);
                        ibyte += size;
                }

                os.write(0); // block terminator
        }

        // ------------------------------------------------------------------------
        private boolean isOk(int frame_index) {
                return frame_index >= 0 && frame_index < vFrames.size();
        }
}

// ============================================================================
class GifColorTable {

        // the palette of ARGB colors, packed as returned by Color.getRGB()
        private int[] theColors = new int[256];

        // other basic attributes
        private int colorDepth;
        private int transparentIndex = -1;

        // these fields track color-index info across frames
        private int ciCount = 0; // count of distinct color indices
        private ReverseColorMap ciLookup; // cumulative rgb-to-ci lookup table

        // ------------------------------------------------------------------------
        GifColorTable() {
                ciLookup = new ReverseColorMap(); // puts us into "auto-detect mode"
        }

        // ------------------------------------------------------------------------
        GifColorTable(Color[] colors) {
                int n2copy = Math.min(theColors.length, colors.length);
                for (int i = 0; i < n2copy; ++i)
                        theColors[i] = colors[i].getRGB();
        }

        int indexOf(Color color) {
                int rgb = color.getRGB();
                for (int i = 0; i < theColors.length; i++) {
                        if (rgb == theColors[i]) {
                                return i;
                        }
                }
                return -1;
        }

        // ------------------------------------------------------------------------
        int getDepth() {
                return colorDepth;
        }

        // ------------------------------------------------------------------------
        int getTransparent() {
                return transparentIndex;
        }

        // ------------------------------------------------------------------------
        // default: -1 (no transparency)
        void setTransparent(int color_index) {
                transparentIndex = color_index;
        }

        // ------------------------------------------------------------------------
        void processPixels(Gif89Frame gf) throws IOException {
                if (gf instanceof DirectGif89Frame)
                        filterPixels((DirectGif89Frame) gf);
                else
                        trackPixelUsage((IndexGif89Frame) gf);
        }

        // ------------------------------------------------------------------------
        void closePixelProcessing() // must be called before encode()
        {
                colorDepth = computeColorDepth(ciCount);
        }

        // ------------------------------------------------------------------------
        void encode(OutputStream os) throws IOException {
                // size of palette written is the smallest power of 2 that can
                // accomdate
```

```
                // the number of RGB colors detected (or largest color index, in
                // case of
                // index pixels)
                int palette_size = 1 << colorDepth;
                for (int i = 0; i < palette_size; ++i) {
                        os.write(theColors[i] >> 16 & 0xff);
                        os.write(theColors[i] >> 8 & 0xff);
                        os.write(theColors[i] & 0xff);
                }
        }

        // ------------------------------------------------------------------------
        // This method accomplishes three things:
        // (1) converts the passed rgb pixels to indexes into our rgb lookup
        // table
        // (2) fills the rgb table as new colors are encountered
        // (3) looks for transparent pixels so as to set the transparent index
        // The information is cumulative across multiple calls.
        //
        // (Note: some of the logic is borrowed from Jef Poskanzer's code.)
        // ------------------------------------------------------------------------
        private void filterPixels(DirectGif89Frame dgf) throws IOException {
                if (ciLookup == null)
                        throw new IOException(
                                        "RGB frames require palette autodetection");

                int[] argb_pixels = (int[]) dgf.getPixelSource();
                byte[] ci_pixels = dgf.getPixelSink();
                int npixels = argb_pixels.length;
                for (int i = 0; i < npixels; ++i) {
                        int argb = argb_pixels[i];

                        // handle transparency
                        if ((argb >>> 24) < 0x80) // transparent pixel?
                                if (transparentIndex == -1) // first transparent color
                                                                                        // encountered?
                                        transparentIndex = ciCount; // record its index
                                else if (argb != theColors[transparentIndex]) // different
                                                                                                                                        //
pixel
                                                                                                                                        //
value?
                                {
                                        // collapse all transparent pixels into one color index
                                        ci_pixels[i] = (byte) transparentIndex;
                                        continue; // CONTINUE - index already in table
                                }

                        // try to look up the index in our "reverse" color table
                        int color_index = ciLookup.getPaletteIndex(argb & 0xffffff);

                        if (color_index == -1) // if it isn't in there yet
                        {
                                if (ciCount == 256)
                                        throw new IOException(
                                                        "can't encode as GIF (> 256 colors)");

                                // store color in our accumulating palette
                                theColors[ciCount] = argb;

                                // store index in reverse color table
                                ciLookup.put(argb & 0xffffff, ciCount);

                                // send color index to our output array
                                ci_pixels[i] = (byte) ciCount;

                                // increment count of distinct color indices
                                ++ciCount;
                        } else
                                // we've already snagged color into our palette
                                ci_pixels[i] = (byte) color_index; // just send filtered
                                                                                                                                // pixel
                }
        }

        // ------------------------------------------------------------------------
        private void trackPixelUsage(IndexGif89Frame igf) throws IOException {
                byte[] ci_pixels = (byte[]) igf.getPixelSource();
                int npixels = ci_pixels.length;
                for (int i = 0; i < npixels; ++i)
                        if (ci_pixels[i] >= ciCount)
                                ciCount = ci_pixels[i] + 1;
        }

        // ------------------------------------------------------------------------
        private int computeColorDepth(int colorcount) {
                // color depth = log-base-2 of maximum number of simultaneous
```

```
                // colors, i.e.
                // bits per color-index pixel
                if (colorcount <= 2)
                        return 1;
                if (colorcount <= 4)
                        return 2;
                if (colorcount <= 16)
                        return 4;
                return 8;
        }
}

// ============================================================================
// We're doing a very simple linear hashing thing here, which seems
// sufficient
// for our needs. I make no claims for this approach other than that it
// seems
// an improvement over doing a brute linear search for each pixel on the one
// hand, and creating a Java object for each pixel (if we were to use a Java
// Hashtable) on the other. Doubtless my little hash could be improved by
// tuning the capacity (at the very least). Suggestions are welcome.
// ============================================================================
class ReverseColorMap {

        private class ColorRecord {
                int rgb;
                int ipalette;

                ColorRecord(int rgb, int ipalette) {
                        this.rgb = rgb;
                        this.ipalette = ipalette;
                }
        }

        // I wouldn't really know what a good hashing capacity is, having missed
        // out
        // on data structures and algorithms class :) Alls I know is, we've got
        // a lot
        // more space than we have time. So let's try a sparse table with a
        // maximum
        // load of about 1/8 capacity.
        private static final int HCAPACITY = 2053; // a nice prime number

        // our hash table proper
        private ColorRecord[] hTable = new ColorRecord[HCAPACITY];

        // ----------------------------------------------------------------------
        // Assert: rgb is not negative (which is the same as saying, be sure the
        // alpha transparency byte - i.e., the high byte - has been masked out).
        // ----------------------------------------------------------------------
        int getPaletteIndex(int rgb) {
                ColorRecord rec;

                for (int itable = rgb % hTable.length; (rec = hTable[itable]) != null
                                && rec.rgb != rgb; itable = ++itable % hTable.length)
                        ;

                if (rec != null)
                        return rec.ipalette;

                return -1;
        }

        // ----------------------------------------------------------------------
        // Assert: (1) same as above; (2) rgb key not already present
        // ----------------------------------------------------------------------
        void put(int rgb, int ipalette) {
                int itable;

                for (itable = rgb % hTable.length; hTable[itable] != null; itable = ++itable
                                % hTable.length)
                        ;

                hTable[itable] = new ColorRecord(rgb, ipalette);
        }
}

// ****************************************************************************
// Gif89Frame.java
// ****************************************************************************

// ============================================================================
/**
 * First off, just to dispel any doubt, this class and its subclasses have
 * nothing to do with GUI "frames" such as java.awt.Frame. We merely use the
 * term in its very common sense of a still picture in an animation
 * sequence. It's hoped that the restricted context will prevent any
```

```
 * confusion.
 * <p>
 * An instance of this class is used in conjunction with a Gif89Encoder
 * object to represent and encode a single static image and its associated
 * "control" data. A Gif89Frame doesn't know or care whether it is encoding
 * one of the many animation frames in a GIF movie, or the single bitmap in
 * a "normal" GIF. (FYI, this design mirrors the encoded GIF structure.)
 * <p>
 * Since Gif89Frame is an abstract class we don't instantiate it directly,
 * but instead create instances of its concrete subclasses, IndexGif89Frame
 * and DirectGif89Frame. From the API standpoint, these subclasses differ
 * only in the sort of data their instances are constructed from. Most folks
 * will probably work with DirectGif89Frame, since it can be constructed
 * from a java.awt.Image object, but the lower-level IndexGif89Frame class
 * offers advantages in specialized circumstances. (Of course, in routine
 * situations you might not explicitly instantiate any frames at all,
 * instead letting Gif89Encoder's convenience methods do the honors.)
 * <p>
 * As far as the public API is concerned, objects in the Gif89Frame
 * hierarchy interact with a Gif89Encoder only via the latter's methods for
 * adding and querying frames. (As a side note, you should know that while
 * Gif89Encoder objects are permanently modified by the addition of
 * Gif89Frames, the reverse is NOT true. That is, even though the ultimate
 * encoding of a Gif89Frame may be affected by the context its parent
 * encoder object provides, it retains its original condition and can be
 * reused in a different context.)
 * <p>
 * The core pixel-encoding code in this class was essentially lifted from
 * Jef Poskanzer's well-known <cite>Acme GifEncoder</cite>, so please see
 * the <a href="../readme.txt">readme</a> containing his notice.
 *
 * @version 0.90 beta (15-Jul-2000)
 * @author J. M. G. Elliott (tep@jmge.net)
 * @see Gif89Encoder
 * @see DirectGif89Frame
 * @see IndexGif89Frame
 */
abstract class Gif89Frame {

        // // Public "Disposal Mode" constants ////

        /**
         * The animated GIF renderer shall decide how to dispose of this
         * Gif89Frame's display area.
         *
         * @see Gif89Frame#setDisposalMode
         */
        public static final int DM_UNDEFINED = 0;

        /**
         * The animated GIF renderer shall take no display-disposal action.
         *
         * @see Gif89Frame#setDisposalMode
         */
        public static final int DM_LEAVE = 1;

        /**
         * The animated GIF renderer shall replace this Gif89Frame's area with
         * the background color.
         *
         * @see Gif89Frame#setDisposalMode
         */
        public static final int DM_BGCOLOR = 2;

        /**
         * The animated GIF renderer shall replace this Gif89Frame's area with
         * the previous frame's bitmap.
         *
         * @see Gif89Frame#setDisposalMode
         */
        public static final int DM_REVERT = 3;

        // // Bitmap variables set in package subclass constructors ////
        int theWidth = -1;
        int theHeight = -1;
        byte[] ciPixels;

        // // GIF graphic frame control options ////
        private Point thePosition = new Point(0, 0);
        private boolean isInterlaced;
        private int csecsDelay;
        private int disposalCode = DM_LEAVE;

        // ----------------------------------------------------------------------
        /**
         * Set the position of this frame within a larger animation display
         * space.
```

```
     *
     * @param p
     *            Coordinates of the frame's upper left corner in the
     *            display space. (Default: The logical display's origin [0,
     *            0])
     * @see Gif89Encoder#setLogicalDisplay
     */
    public void setPosition(Point p) {
        thePosition = new Point(p);
    }

    // --------------------------------------------------------------------------
    /**
     * Set or clear the interlace flag.
     *
     * @param b
     *            true if you want interlacing. (Default: false)
     */
    public void setInterlaced(boolean b) {
        isInterlaced = b;
    }

    // --------------------------------------------------------------------------
    /**
     * Set the between-frame interval.
     *
     * @param interval
     *            Centiseconds to wait before displaying the subsequent
     *            frame. (Default: 0)
     */
    public void setDelay(int interval) {
        csecsDelay = interval;
    }

    // --------------------------------------------------------------------------
    /**
     * Setting this option determines (in a cooperative GIF-viewer) what
     * will be done with this frame's display area before the subsequent
     * frame is displayed. For instance, a setting of DM_BGCOLOR can be used
     * for erasure when redrawing with displacement.
     *
     * @param code
     *            One of the four int constants of the Gif89Frame.DM_*
     *            series. (Default: DM_LEAVE)
     */
    public void setDisposalMode(int code) {
        disposalCode = code;
    }

    // --------------------------------------------------------------------------
    Gif89Frame() {
    } // package-visible default constructor

    // --------------------------------------------------------------------------
    abstract Object getPixelSource();

    // --------------------------------------------------------------------------
    int getWidth() {
        return theWidth;
    }

    // --------------------------------------------------------------------------
    int getHeight() {
        return theHeight;
    }

    // --------------------------------------------------------------------------
    byte[] getPixelSink() {
        return ciPixels;
    }

    // --------------------------------------------------------------------------
    void encode(OutputStream os, boolean epluribus, int color_depth,
                int transparent_index) throws IOException {
        writeGraphicControlExtension(os, epluribus, transparent_index);
        writeImageDescriptor(os);
        new GifPixelsEncoder(theWidth, theHeight, ciPixels, isInterlaced,
                        color_depth).encode(os);
    }

    // --------------------------------------------------------------------------
    private void writeGraphicControlExtension(OutputStream os,
                boolean epluribus, int itransparent) throws IOException {
        int transflag = itransparent == -1 ? 0 : 1;
        if (transflag == 1 || epluribus) // using transparency or animating
                                                                                // ?
        {
```

```
                os.write((int) '!'); // GIF Extension Introducer
                os.write(0xf9); // Graphic Control Label
                os.write(4); // subsequent data block size
                os.write((disposalCode << 2) | transflag); // packed fields (1
                                                                                                                        // byte)
                putShort(csecsDelay, os); // delay field (2 bytes)
                os.write(itransparent); // transparent index field
                os.write(0); // block terminator
            }
        }

        // ------------------------------------------------------------------------
        private void writeImageDescriptor(OutputStream os) throws IOException {
            os.write((int) ','); // Image Separator
            putShort(thePosition.x, os);
            putShort(thePosition.y, os);
            putShort(theWidth, os);
            putShort(theHeight, os);
            os.write(isInterlaced ? 0x40 : 0); // packed fields (1 byte)
        }
}

// ============================================================================
class GifPixelsEncoder {

        private static final int EOF = -1;

        private int imgW, imgH;
        private byte[] pixAry;
        private boolean wantInterlaced;
        private int initCodeSize;

        // raster data navigators
        private int countDown;
        private int xCur, yCur;
        private int curPass;

        // ------------------------------------------------------------------------
        GifPixelsEncoder(int width, int height, byte[] pixels,
                         boolean interlaced, int color_depth) {
            imgW = width;
            imgH = height;
            pixAry = pixels;
            wantInterlaced = interlaced;
            initCodeSize = Math.max(2, color_depth);
        }

        // ------------------------------------------------------------------------
        void encode(OutputStream os) throws IOException {
            os.write(initCodeSize); // write "initial code size" byte

            countDown = imgW * imgH; // reset navigation variables
            xCur = yCur = curPass = 0;

            compress(initCodeSize + 1, os); // compress and write the pixel data

            os.write(0); // write block terminator
        }

        // ************************************************************************
        // (J.E.) The logic of the next two methods is largely intact from
        // Jef Poskanzer. Some stylistic changes were made for consistency sake,
        // plus the second method accesses the pixel value from a prefiltered
        // linear
        // array. That's about it.
        // ************************************************************************

        // ------------------------------------------------------------------------
        // Bump the 'xCur' and 'yCur' to point to the next pixel.
        // ------------------------------------------------------------------------
        private void bumpPosition() {
            // Bump the current X position
            ++xCur;

            // If we are at the end of a scan line, set xCur back to the
            // beginning
            // If we are interlaced, bump the yCur to the appropriate spot,
            // otherwise, just increment it.
            if (xCur == imgW) {
                xCur = 0;

                if (!wantInterlaced)
                    ++yCur;
                else
                    switch (curPass) {
                    case 0:
                        yCur += 8;
```

```
                        if (yCur >= imgH) {
                                ++curPass;
                                yCur = 4;
                        }
                        break;
                case 1:
                        yCur += 8;
                        if (yCur >= imgH) {
                                ++curPass;
                                yCur = 2;
                        }
                        break;
                case 2:
                        yCur += 4;
                        if (yCur >= imgH) {
                                ++curPass;
                                yCur = 1;
                        }
                        break;
                case 3:
                        yCur += 2;
                        break;
                }
        }
}

// ------------------------------------------------------------------------
// Return the next pixel from the image
// ------------------------------------------------------------------------
private int nextPixel() {
        if (countDown == 0)
                return EOF;

        --countDown;

        byte pix = pixAry[yCur * imgW + xCur];

        bumpPosition();

        return pix & 0xff;
}

// ****************************************************************************
// (J.E.) I didn't touch Jef Poskanzer's code from this point on. (Well,
// OK,
// I changed the name of the sole outside method it accesses.) I figure
// if I have no idea how something works, I shouldn't play with it :)
//
// Despite its unencapsulated structure, this section is actually highly
// self-contained. The calling code merely calls compress(), and the
// present
// code calls nextPixel() in the caller. That's the sum total of their
// communication. I could have dumped it in a separate class with a
// callback
// via an interface, but it didn't seem worth messing with.
// ****************************************************************************

// GIFCOMPR.C - GIF Image compression routines
//
// Lempel-Ziv compression based on 'compress'. GIF modifications by
// David Rowley (mgardi@watdcsu.waterloo.edu)

// General DEFINEs

static final int BITS = 12;

static final int HSIZE = 5003; // 80% occupancy

// GIF Image compression - modified 'compress'
//
// Based on: compress.c - File compression ala IEEE Computer, June 1984.
//
// By Authors: Spencer W. Thomas (decvax!harpo!utah-cs!utah-gr!thomas)
// Jim McKie (decvax!mcvax!jim)
// Steve Davies (decvax!vax135!petsd!peora!srd)
// Ken Turkowski (decvax!decwrl!turtlevax!ken)
// James A. Woods (decvax!ihnp4!ames!jaw)
// Joe Orost (decvax!vax135!petsd!joe)

int n_bits; // number of bits/code
int maxbits = BITS; // user settable max # bits/code
int maxcode; // maximum code, given n_bits
int maxmaxcode = 1 << BITS; // should NEVER generate this code

final int MAXCODE(int n_bits) {
        return (1 << n_bits) - 1;
}
```

```
int[] htab = new int[HSIZE];
int[] codetab = new int[HSIZE];

int hsize = HSIZE; // for dynamic table sizing

int free_ent = 0; // first unused entry

// block compression parameters -- after all codes are used up,
// and compression rate changes, start over.
boolean clear_flg = false;

// Algorithm: use open addressing double hashing (no chaining) on the
// prefix code / next character combination. We do a variant of Knuth's
// algorithm D (vol. 3, sec. 6.4) along with G. Knott's relatively-prime
// secondary probe. Here, the modular division first probe is gives way
// to a faster exclusive-or manipulation. Also do block compression with
// an adaptive reset, whereby the code table is cleared when the
// compression
// ratio decreases, but after the table fills. The variable-length
// output
// codes are re-sized at this point, and a special CLEAR code is
// generated
// for the decompressor. Late addition: construct the table according to
// file size for noticeable speed improvement on small files. Please
// direct
// questions about this implementation to ames!jaw.

int g_init_bits;

int ClearCode;
int EOFCode;

void compress(int init_bits, OutputStream outs) throws IOException {
        int fcode;
        int i /* = 0 */;
        int c;
        int ent;
        int disp;
        int hsize_reg;
        int hshift;

        // Set up the globals: g_init_bits - initial number of bits
        g_init_bits = init_bits;

        // Set up the necessary values
        clear_flg = false;
        n_bits = g_init_bits;
        maxcode = MAXCODE(n_bits);

        ClearCode = 1 << (init_bits - 1);
        EOFCode = ClearCode + 1;
        free_ent = ClearCode + 2;

        char_init();

        ent = nextPixel();

        hshift = 0;
        for (fcode = hsize; fcode < 65536; fcode *= 2)
                ++hshift;
        hshift = 8 - hshift; // set hash code range bound

        hsize_reg = hsize;
        cl_hash(hsize_reg); // clear hash table

        output(ClearCode, outs);

        outer_loop: while ((c = nextPixel()) != EOF) {
                fcode = (c << maxbits) + ent;
                i = (c << hshift) ^ ent; // xor hashing

                if (htab[i] == fcode) {
                        ent = codetab[i];
                        continue;
                } else if (htab[i] >= 0) // non-empty slot
                {
                        disp = hsize_reg - i; // secondary hash (after G. Knott)
                        if (i == 0)
                                disp = 1;
                        do {
                                if ((i -= disp) < 0)
                                        i += hsize_reg;

                                if (htab[i] == fcode) {
                                        ent = codetab[i];
                                        continue outer_loop;
```

```
                                }
                        } while (htab[i] >= 0);
                }
                output(ent, outs);
                ent = c;
                if (free_ent < maxmaxcode) {
                        codetab[i] = free_ent++; // code -> hashtable
                        htab[i] = fcode;
                } else
                        cl_block(outs);
        }
        // Put out the final code.
        output(ent, outs);
        output(EOFCode, outs);
}

// output
//
// Output the given code.
// Inputs:
// code: A n_bits-bit integer. If == -1, then EOF. This assumes
// that n_bits =< wordsize - 1.
// Outputs:
// Outputs code to the file.
// Assumptions:
// Chars are 8 bits long.
// Algorithm:
// Maintain a BITS character long buffer (so that 8 codes will
// fit in it exactly). Use the VAX insv instruction to insert each
// code in turn. When the buffer fills up empty it and start over.

int cur_accum = 0;
int cur_bits = 0;

int masks[] = { 0x0000, 0x0001, 0x0003, 0x0007, 0x000F, 0x001F, 0x003F,
                0x007F, 0x00FF, 0x01FF, 0x03FF, 0x07FF, 0x0FFF, 0x1FFF, 0x3FFF,
                0x7FFF, 0xFFFF };

void output(int code, OutputStream outs) throws IOException {
        cur_accum &= masks[cur_bits];

        if (cur_bits > 0)
                cur_accum |= (code << cur_bits);
        else
                cur_accum = code;

        cur_bits += n_bits;

        while (cur_bits >= 8) {
                char_out((byte) (cur_accum & 0xff), outs);
                cur_accum >>= 8;
                cur_bits -= 8;
        }

        // If the next entry is going to be too big for the code size,
        // then increase it, if possible.
        if (free_ent > maxcode || clear_flg) {
                if (clear_flg) {
                        maxcode = MAXCODE(n_bits = g_init_bits);
                        clear_flg = false;
                } else {
                        ++n_bits;
                        if (n_bits == maxbits)
                                maxcode = maxmaxcode;
                        else
                                maxcode = MAXCODE(n_bits);
                }
        }

        if (code == EOFCode) {
                // At EOF, write the rest of the buffer.
                while (cur_bits > 0) {
                        char_out((byte) (cur_accum & 0xff), outs);
                        cur_accum >>= 8;
                        cur_bits -= 8;
                }

                flush_char(outs);
        }
}

// Clear out the hash table

// table clear for block compress
void cl_block(OutputStream outs) throws IOException {
        cl_hash(hsize);
        free_ent = ClearCode + 2;
```

```
                clear_flg = true;

                output(ClearCode, outs);
        }

        // reset code table
        void cl_hash(int hsize) {
                for (int i = 0; i < hsize; ++i)
                        htab[i] = -1;
        }

        // GIF Specific routines

        // Number of characters so far in this 'packet'
        int a_count;

        // Set up the 'byte output' routine
        void char_init() {
                a_count = 0;
        }

        // Define the storage for the packet accumulator
        byte[] accum = new byte[256];

        // Add a character to the end of the current packet, and if it is 254
        // characters, flush the packet to disk.
        void char_out(byte c, OutputStream outs) throws IOException {
                accum[a_count++] = c;
                if (a_count >= 254)
                        flush_char(outs);
        }

        // Flush the packet to disk, and reset the accumulator
        void flush_char(OutputStream outs) throws IOException {
                if (a_count > 0) {
                        outs.write(a_count);
                        outs.write(accum, 0, a_count);
                        a_count = 0;
                }
        }
}

// ****************************************************************************
// IndexGif89Frame.java
// ****************************************************************************

// ============================================================================
/**
 * Instances of this Gif89Frame subclass are constructed from bitmaps in the
 * form of color-index pixels, which accords with a GIF's native palettized
 * color model. The class is useful when complete control over a GIF's color
 * palette is desired. It is also much more efficient when one is using an
 * algorithmic frame generator that isn't interested in RGB values (such as
 * a cellular automaton).
 * <p>
 * Objects of this class are normally added to a Gif89Encoder object that
 * has been provided with an explicit color table at construction. While you
 * may also add them to "auto-map" encoders without an exception being
 * thrown, there obviously must be at least one DirectGif89Frame object in
 * the sequence so that a color table may be detected.
 *
 * @version 0.90 beta (15-Jul-2000)
 * @author J. M. G. Elliott (tep@jmge.net)
 * @see Gif89Encoder
 * @see Gif89Frame
 * @see DirectGif89Frame
 */
class IndexGif89Frame extends Gif89Frame {

        // ------------------------------------------------------------------------
        /**
         * Construct a IndexGif89Frame from color-index pixel data.
         *
         * @param width
         *            Width of the bitmap.
         * @param height
         *            Height of the bitmap.
         * @param ci_pixels
         *            Array containing at least width*height color-index pixels.
         */
        public IndexGif89Frame(int width, int height, byte ci_pixels[]) {
                theWidth = width;
                theHeight = height;
                ciPixels = new byte[theWidth * theHeight];
                System.arraycopy(ci_pixels, 0, ciPixels, 0, ciPixels.length);
        }
```

```
                // --------------------------------------------------------------------------
                Object getPixelSource() {
                        return ciPixels;
                }
        }

        // --------------------------------------------------------------------------
        /**
         * Internal method;
         * write just the low bytes of a String. (This sucks, but the concept of an
         * encoding seems inapplicable to a binary file ID string. I would think
         * flexibility is just what we don't want - but then again, maybe I'm slow.)
         * This is an internal method not meant to be called by clients.
         */
        private static void putAscii(String s, OutputStream os) throws IOException {
                byte[] bytes = new byte[s.length()];
                for (int i = 0; i < bytes.length; ++i) {
                        bytes[i] = (byte) s.charAt(i); // discard the high byte
                }
                os.write(bytes);
        }

        // --------------------------------------------------------------------------
        /**
         * Internal method;
         * write a 16-bit integer in little endian byte order.
         * This is an internal method not meant to be called by clients.
         */
        private static void putShort(int i16, OutputStream os) throws IOException {
                os.write(i16 & 0xff);
                os.write(i16 >> 8 & 0xff);
        }
}
```