SimpleEditor Tutorial

Abstract: This tutorial will walk you through building a text editor similar to Windows Notepad in the Java programming language using the Eclipse IDE.

Step 1 – Create a New Project

Run Eclipse and create a new Java project named 'SimpleEditor'.

I highly recommend selecting the 'Create separate source and output folders' so the Java source code will be stored in a different directory from the compiler generated '.class' files.

Step 2 – Create the Main Window class

When our program is run, one of the first things we need to do is to create the top window, also called a frame window.

Add a new class to your project by right-clicking on the 'SimpleEditor' project in the Package Explorer window and selecting **New > Class** from the menu.

Enter the following values into the New Java Class dialog box fields:

Package: edu.uah.coned
Name: EditorFrame
Superclass: javax.swing.JFrame (can use Browser button)

The only option that should be checked in the bottom half of the dialog box is the 'Inherited abstract methods' checkbox.

We will be modifying this class a lot before our program is completed.

Step 3 – EditorFrame constructor

We need to add a constructor to our new class which will create the window when we create a new object. Let's also use the Singleton design pattern for this class so only one copy of it can be created.

Modify the class to read as follows:

```
public class EditorFrame extends JFrame {
```

```
	static EditorFrame _instance;

	private EditorFrame() {
		super("Simple Editor");
	}

	public static EditorFrame getInstance() {
		if( _instance == null ) {
			_instance = new EditorFrame();
		}
		return _instance;
	}
}
```

Step 4 – The Main() method

Since all Java programs require a method named Main(), let's create that next.

Add another class to your project named 'SimpleEditor'. Once again, let's use edu.uah.coned for the package name and make sure the checkbox to create the stub Main() method is enabled.

Modify the new class to read as follows:

```
	public static void main(String[] args) {
		EditorFrame mainWindow;
		mainWindow = EditorFrame.getInstance();
		mainWindow.setVisible(true);
	}
```

Step 5 – Test the program

Let's compile and run this basic beginning.

Select the 'Run...' option on the 'Run' menu.

Highlight 'Java Application' in the list on the left and click the 'New' button.

This should create a new entry named 'SimpleEditor' and fill in the Project name with 'SimpleEditor' and the Main class should be set to 'edu.uah.coned.SimpleEditor'.

If this is correct, click the 'Run' button to make Eclipse compile and execute our program.

You should see a new Window appear. If you expand the window, it should have the title 'Simple Editor'.

However, there is still one problem. While you can click the X in the corner to close the window, the program continues to run even after the window is closed.

Step 6 – Make application exit when closed

Some applications make not wish to exit when their main window is closed, but in our case we want our program to terminate automatically when our frame window is closed.

To do this we need to call the setDefaultCloseOperation() method and pass in an option to set the behavior we want. You can pass in one of the following values:

DO_NOTHING_ON_CLOSE

> Don't do anything; require the program to handle the operation in the windowClosing method of a registered WindowListener object.

HIDE_ON_CLOSE

> Automatically hide the frame after invoking any registered WindowListener objects. (Default behavior)

DISPOSE_ON_CLOSE

> Automatically hide and dispose the frame after invoking any registered WindowListener objects.

EXIT_ON_CLOSE

> Exit the application using the System exit method. Use this only in applications.

In this case, either EXIT_ON_CLOSE or DISPOSE_ON_CLOSE should work, although the EXIT_ON_CLOSE option forces the Java VM to terminate, while the other does not. The difference is that the EXIT option forces Java to terminate, while the DISPOSE option does not.

Modify the EditorFrame constructor to read like this:

```
private EditorFrame() {
      super("Simple Editor");
      setDefaultCloseOperation( EXIT_ON_CLOSE );
}
```

Aftering making these changes, run the program again and verify the program terminates when you close the main window.

There is an alternate way to do this same thing. Instead of calling the setDefaultCloseOperation() method, we could instead listen for the event that all windows send when they are closed. This requires that we create a new class that can listen for this event and calls the System.exit() method to terminate our program.

Modify the EditorFrame class as follows:

```
private EditorFrame() {
      super("Simple Editor");
      // setDefaultCloseOperation( EXIT_ON_CLOSE );
      addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                  System.exit(0);
            }
      });
}
```

You will also need to add the following import statements to the top of the file:

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
```

Step 7 – Create a Menu

Now lets add a menu to our frame window with the various options we will support.

First, add the following method to our EditorFrame class:

```
public void createMenu() {
      JMenuBar menuBar = new JMenuBar();

      JMenu fileMenu = new JMenu("File");
      JMenu editMenu = new JMenu("Edit");

      fileMenu.setMnemonic('F');
      editMenu.setMnemonic('E');

      menuBar.add( fileMenu );
```

```
            menuBar.add( editMenu );

            getContentPane().add( menuBar, BorderLayout.NORTH );
      }
```

You will also need to add the following import statements to the top of the file:

```
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import java.awt.BorderLayout;
```

Finally, modify our Main() method so it calls the new createMenu() method like this:

```
      public static void main(String[] args) {
            EditorFrame mainWindow;
            mainWindow = EditorFrame.getInstance();
            mainWindow.createMenu();
            mainWindow.setVisible(true);
      }
```

If you save and run this version, you will see our program now has File and Edit menu options, although they are still empty. We will fix that shortly.

Step 8 – Add options to the menu

We now have a menu for our program, but they are currently empty. Let's enhance the menu by adding some choices under File and Edit.

Modify the createMenu() method as follows:

```
      public void createMenu() {
            JMenuBar menuBar = new JMenuBar();

            JMenu fileMenu = new JMenu("File");
            JMenu editMenu = new JMenu("Edit");

            fileMenu.setMnemonic('F');
            editMenu.setMnemonic('E');

            menuBar.add( fileMenu );
            menuBar.add( editMenu );

            JMenuItem fileNewItem = new JMenuItem( "New", 'N' );
            JMenuItem fileOpenItem = new JMenuItem( "Open", 'O' );
            JMenuItem fileSaveItem = new JMenuItem( "Save", 'S' );
            JMenuItem fileSaveAsItem = new JMenuItem( "Save As", 'A' );
            JMenuItem fileCloseItem = new JMenuItem( "Close", 'C' );
```

```
        JMenuItem filePrintItem = new JMenuItem( "Print", 'P' );
        JMenuItem fileExitItem = new JMenuItem( "Exit", 'x' );

        fileMenu.add(fileNewItem);
        fileMenu.add(fileOpenItem);
        fileMenu.add(fileSaveItem);
        fileMenu.add(fileSaveAsItem);
        fileMenu.add(fileCloseItem);
        fileMenu.addSeparator();
        fileMenu.add(filePrintItem);
        fileMenu.addSeparator();
        fileMenu.add(fileExitItem);

        JMenuItem editCopyItem = new JMenuItem( "Copy", 'C' );
        JMenuItem editCutItem = new JMenuItem( "Cut", 't' );
        JMenuItem editPasteItem = new JMenuItem( "Paste", 'P' );

        editMenu.add(editCopyItem);
        editMenu.add(editCutItem);
        editMenu.add(editPasteItem);

        getContentPane().add( menuBar, BorderLayout.NORTH );
    }
```

You will also need to add the following import statement to the top of the file:

```
import javax.swing.JMenuItem;
```

Save and run the new code. The menu should display the new items, although they still do not work.

Step 9 – Handle menu options

We need to run some code whenever the user selects one of the menu options. For example, we need to close/exit if the user chooses **File > Exit**.

This requires that we listen for an event, in much the same way we listened for the window close event earlier. In this case, we need to listen for an 'actionPerformed' event from the menu items, not a window event however.

Modify the createMenu() method as shown below:

```
        ... Same as before ...

        JMenuItem fileExitItem = new JMenuItem( "Exit", 'x' );

        fileExitItem.addActionListener(new ActionListener() {
```

```
            public void actionPerformed(ActionEvent e) {
                  System.exit(0);
            }
      });

      fileMenu.add(fileNewItem);
```

... Same As before ...

You will need to add these import statements:

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Try running the code to see if this works. You should be able to exit using **File > Exit**.

Step 10 – Adding a toolbar

Even though our menu is not yet fully functional, let's put that aside for a moment. In today's modern computing systems, no respectable application would ship without a toolbar, so let's add one.

We are going to have a problem however. We want our toolbar to also be placed at the top of our program, preferably right below the menu bar we just added.

We are also going to need some graphics for the toolbar buttons. Java supports using JPG, PNG or GIF files as graphics for toolbar buttons. Naturally you can search the Internet and download prebuilt graphics, or you can use a default set of graphics provided by Sun.

The Sun graphics are recommended so Java applications have a consistent look and feel. You can download them by visiting http://java.sun.com and searching for 'Java Look and Feel Graphics'.

The file you download will be named jlfgr-1.0.zip. We need to extract all the graphics from this file using the following commands:

```
$ cd ${WORKSPACE}/SimpleEditor
$ mkdir images
$ cd images
$ unzip ${DOWNLOAD}/jlfgr-1.0.zip
$ jar -xf jlfgr-1.0.jar
```

Now we have a repository of graphics to use for our toolbar.

However, we still have a management problem. We are currently creating many different JMenuItem objects and adding them to the menu bar. A toolbar cannot hold these types of objects. Instead, it wants you to create and add Action objects to the toolbar.

If you examine the Java API documentation, you will find that Action is not a class, but an interface. This means we need to build a class that implements the Action interface, then create objects from that class that can then be added to a JToolBar.

If you examine the JMenuBar class, you will find that it can also use Action objects. Therefore, before we actually build the toolbar, let's create some Action classes and use them to build up the menu and toolbar at the same time.

It takes quite a bit of work to implement the Action interface yourself, so the Java designers have created a class to make this a bit easier. The AbstractAction class can be used via inheritance to create your own action classes. When you do this, the AbstractAction class will take care of many of the details required to implement the interface. However, you do still need to do 2 things to make a concrete Action class of your own.

First, you must add a constructor to your class. It is required that your constructor call the AbstractAction constructor using the super() method. You should supply at least the text for the action (used on the menu and on the toolbar) and optionally you should also provide an icon for the action (also used by both menus and toolbars).

We also want to define a few other things, like tooltips and shortcut keys for our actions, so let's first create our own abstract class from AbstractAction and then create a bunch of small action classes, one for each action we want to support in our project.

Start by adding a new class to our project named 'ActionItem'. Modify the class to read as follows:

```
package edu.uah.coned;

import javax.swing.AbstractAction;
import javax.swing.ImageIcon;

public abstract class ActionItem extends AbstractAction {
```

```
	private int _mnemonic;
	private String _tooltip;
	private int _key;
	private int _modifier;

	public ActionItem( String text, int mnemonic, int key, int
modifier, String tooltip, String icon ) {
		super( text, new ImageIcon(icon) );
		_mnemonic = mnemonic;
		_key = key;
		_modifier = modifier;
		_tooltip = tooltip;
	}

	public ActionItem( String text, int mnemonic, int key, int
modifier, String tooltip ) {
		super( text );
		_mnemonic = mnemonic;
		_key = key;
		_modifier = modifier;
		_tooltip = tooltip;
	}

	public int getMnemonic() {
		return _mnemonic;
	}

	public String getTooltip() {
		return _tooltip;
	}

	public int getKey() {
		return _key;
	}

	public int getModifier() {
		return _modifier;
	}
}
```

Next, let's add another class, this time a concrete class for our **File > New** action. Name it 'FileNewAction' and make sure it inherits from the 'ActionItem' class we just created.

Modify the class as follows:

```
package edu.uah.coned;

import java.awt.Event;
import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;

public class FileNewAction extends ActionItem {
```

```
	public FileNewAction() {
		super("File", 'F', KeyEvent.VK_N, Event.CTRL_MASK, "Create a
new empty file", "images/toolbarButtonGraphics/general/New16.gif" );
	}

	public void actionPerformed(ActionEvent e) {
		EditorFrame.getInstance().doNewAction();
	}

}
```

We must also create the doNewAction() method in the EditorFrame class as shown below:

```
	public void doNewAction() {
		// TODO Auto-generated method stub
	}
```

We will be adding some code to this method later. For the moment, leave the body of the method blank.

Repeat and create new classes named 'FileOpenAction', 'FileSaveAction', 'FileSaveAsAction', 'FileCloseAction', 'FileExitAction', 'EditCopyAction', 'EditCutAction' and 'EditPasteAction'. Customize the constructor and actionPerformed() methods for each class as appropriate.

Step 11 – Use the new Action classes

Now that we have created all the action classes, we need to revise the createMenu() method to use them.

We still have one small problem however. We want both a JMenuBar and a JToolBar to be added to the north pane of our main content window. However, panes can only hold one thing at a time. The solution for this is to use another container object that can hold other items.

The easiest way to do this is by creating a JPanel object, with its own layout manager to hold the menu and toolbar. Then we can add the panel to the north pane of the main frame window.

Modify the createMenu() method in EditorFrame as follows:

```
	public void createMenu() {
		JMenu fileMenu = new JMenu( "File" );
		fileMenu.setMnemonic( 'F' );
```

```
        JMenu editMenu = new JMenu( "Edit" );
        editMenu.setMnemonic( 'E' );

        JToolBar toolBar = new JToolBar();

        FileNewAction fileNew = new FileNewAction();
        FileOpenAction fileOpen = new FileOpenAction();
        FileSaveAction fileSave = new FileSaveAction();
        FileSaveAsAction fileSaveAs = new FileSaveAsAction();
        FileCloseAction fileClose = new FileCloseAction();
        FilePrintAction filePrint = new FilePrintAction();
        FileExitAction fileExit = new FileExitAction();

        EditCopyAction editCopy = new EditCopyAction();
        EditCutAction editCut = new EditCutAction();
        EditPasteAction editPaste = new EditPasteAction();

        fileMenu.add( fileNew );
        fileMenu.addSeparator();
        fileMenu.add( fileOpen );
        fileMenu.add( fileSave );
        fileMenu.add( fileSaveAs );
        fileMenu.addSeparator();
        fileMenu.add( fileClose );
        fileMenu.addSeparator();
        fileMenu.add( filePrint );
        fileMenu.addSeparator();
        fileMenu.add( fileExit );

        editMenu.add( editCopy );
        editMenu.add( editCut );
        editMenu.add( editPaste );

        toolBar.add( fileNew );
        toolBar.add( fileOpen );
        toolBar.add( fileSave );
        toolBar.addSeparator();
        toolBar.add( filePrint );
        toolBar.addSeparator();
        toolBar.add( editCopy );
        toolBar.add( editCut );
        toolBar.add( editPaste );

        JMenuBar menuBar = new JMenuBar();

        menuBar.add( fileMenu );
        menuBar.add( editMenu );

        JPanel panel = new JPanel( new GridLayout(2,1), false );
        panel.add( menuBar );
        panel.add( toolBar );

        getContentPane().add( panel, BorderLayout.NORTH );
    }
```

If you run this version of the program, you should see both the menu and the toolbar right below it.

This works, however the shortcut keys such as Ctrl+O and Ctrl+Q do not yet work. In addition, the toolbar buttons do not have any tooltips, which all modern GUI programs should have.

In addition, the menu is also displaying an icon next to each option. While this is not a problem, it is not very common.

Step 12 – Improve menu and toolbar

In order to fix these issues, we need to call additional methods to customize the buttons and items in the toolbars and menus. Instead of adding a lot of code to our createMenu() method, let's make the ActionItem class responsible for this.

Add the following methods to the ActionItem class:

```
	public ActionItem addToMenu( JMenu menu ) {
		JMenuItem item = menu.add( this );

		// Turn off icon on the menu
		item.setIcon( null );

		if( _mnemonic != 0 ) {
			item.setMnemonic( _mnemonic );
		}

		if( _key != 0 ) {
			item.setAccelerator( KeyStroke.getKeyStroke( _key,
_modifier) );
		}
		return this;
	}

	public ActionItem addToToolbar( JToolBar toolbar ) {
		JButton button = toolbar.add( this );

		if( button.getIcon() == null ) {
			toolbar.remove( button );
		} else {
			button.setText( "" );
			if( _tooltip != null ) {
				button.setToolTipText( _tooltip );
			}
		}

		return this;
	}
```

Next, let's adjust the EditorFrame class to use the new methods. At some point later, we may need to dynamically enable/disable some of these actions (for example you cannot copy or cut text if no characters are selected).

That means we may need to our ActionItem objects after createMenu() has run. This requires use to add some additional fields to the EditorFrame class as shown below:

```
static EditorFrame _instance;

private ActionItem _fileNew;
private ActionItem _fileOpen;
private ActionItem _fileSave;
private ActionItem _fileSaveAs;
private ActionItem _fileClose;
private ActionItem _filePrint;
private ActionItem _fileExit;
private ActionItem _editCopy;
private ActionItem _editCut;
private ActionItem _editPaste;
```

Now, we can finally write the final version of the createMenu() method. Here is it:

```
public void createMenu() {
      JMenu fileMenu = new JMenu( "File" );
      fileMenu.setMnemonic( 'F' );

      JMenu editMenu = new JMenu( "Edit" );
      editMenu.setMnemonic( 'E' );

      JToolBar toolBar = new JToolBar();

      _fileNew = new FileNewAction();
      _fileOpen = new FileOpenAction();
      _fileSave = new FileSaveAction();
      _fileSaveAs = new FileSaveAsAction();
      _fileClose = new FileCloseAction();
      _filePrint = new FilePrintAction();
      _fileExit = new FileExitAction();

      _editCopy = new EditCopyAction();
      _editCut = new EditCutAction();
      _editPaste = new EditPasteAction();

      _fileNew.addToMenu( fileMenu ).addToToolbar( toolBar );
      fileMenu.addSeparator();
      _fileOpen.addToMenu( fileMenu ).addToToolbar( toolBar );
      _fileSave.addToMenu( fileMenu ).addToToolbar( toolBar );
```

```
        _fileSaveAs.addToMenu( fileMenu );
        fileMenu.addSeparator();
        _fileClose.addToMenu( fileMenu );
        fileMenu.addSeparator();
        toolBar.addSeparator();
        _filePrint.addToMenu( fileMenu ).addToToolbar( toolBar );
        fileMenu.addSeparator();
        _fileExit.addToMenu( fileMenu );

        toolBar.addSeparator();

        _editCopy.addToMenu( editMenu ).addToToolbar( toolBar );
        _editCut.addToMenu( editMenu ).addToToolbar( toolBar );
        _editPaste.addToMenu( editMenu ).addToToolbar( toolBar );

        JMenuBar menuBar = new JMenuBar();

        menuBar.add( fileMenu );
        menuBar.add( editMenu );

        JPanel panel = new JPanel( new GridLayout(2,1), false );
        panel.add( menuBar );
        panel.add( toolBar );

        getContentPane().add( panel, BorderLayout.NORTH );
    }
```

Step 13 – Add a status bar

Now that we have a menu and toolbar in place, let's move our attention to adding a status bar to the bottom of the main window.

Since Java does not have any kind of built-in support for a status bar, I have created a reusable class that can be used for this.

See File: *JStatusBar.java*

Import the JStatusBar.java code into your project and then add the following method to the EditorFrame class:

```
    public void createStatusBar() {
        _statusBar = new JStatusBar( StatusBar.ALL_PANES );
        getContentPane().add( _statusBar, BorderLayout.SOUTH );
    }
```

You will of course need to create a variable named _statusBar. It should look like this:

```
    private StatusBar _statusBar;
```

You also need to modify the Main() method so it calls the createStatusBar() method. Like this:

```
public static void main(String[] args) {
      EditorFrame mainWindow;

      mainWindow = EditorFrame.getInstance();
      mainWindow.createMenu();
      mainWindow.createStatusBar();
      mainWindow.setSize( 400, 300 );
      mainWindow.setVisible(true);
}
```

Step 14 – Setup the text area.

Now that our menu, toolbar and status bar are working, let's start working on the code that let's users edit text. We are going to be using the JTextArea class for this. All we really need to do is add a variable and then create the new text area and add it to our context pane. Probably the best place for this code is in our constructor.

```
private EditorFrame() {
      super("Simple Editor");
      // setDefaultCloseOperation( EXIT_ON_CLOSE );
      addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                  System.exit(0);
            }
      });

      _textArea = new JTextArea();
      getContentPane().add( _textArea, BorderLayout.CENTER );
}
```

This works, but if you run the program you should find a problem. Trying typing in enough lines to make the text scroll. It doesn't!!!

Step 15 – Adding scroll bars

The JScrollPane class is designed to add scroll bars to any other pane. It can add both a vertical or horizontal scroll bars to the pane. It acts as a container and monitors the desired size of the inner component and displays the scroll bars as needed. The scroll bars can also be turned on or off permanently using the setHorizontalScrollBarPolicy() and setVeritcalScrollBarPolicy() methods. It also can hold other components called the column and row header areas. In addition, you can also add your own components to the corners of the JScrollBar main window.

It supports scrolling using two different techniques. If the component added to the main viewport area supports the Scrollable interface, then the Scrollable methods are called directly. If the component does not support the Scrollable interface, then the JScrollBar will monitor the embedded components preferred size and control the scroll bars using the component's desired size.

To make use of the JScrollBar component, add the following code to the EditorFrame's constructor:

```
private EditorFrame() {
      super("Simple Editor");
      // setDefaultCloseOperation( EXIT_ON_CLOSE );
      addWindowListener( new WindowAdapter() {
            public void windowClosing(WindowEvent event) {
                  System.exit(0);
            }
      });

      _textArea = new JTextArea();
      JScrollPane textPane = new JscrollPane();
      textPane.setViewportView( _textArea );
      getContentPane().add( _textArea, BorderLayout.CENTER );
}
```

If you run the program and enter enough text, the scroll bars should now appear automatically.

Step 16 – Insert and Overwrite modes

By default all objects and classes derived from JTextComponent only allow inserting new text into the text region. However, most modern text editors support either inserting or overwriting text, which can be toggled by pressing the INSERT key. The first thing we need to do is to listen for a press of the INSERT key so we can monitor and toggle the current mode of operation.

We could add this support directly to our EditorFrame class, however this would not allow us to easily reuse the code again in future project, except via cut 'n paste. In addition, the text component itself should be responsible for monitoring its own state.

Let's create a class that extends the JTextPane class and monitors the INSERT key for us. Here is the full class:

```
package edu.uah.coned;
```

```
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;

import javax.swing.JTextPane;
import javax.swing.text.Document;
import javax.swing.text.DocumentFilter;
import javax.swing.text.StyledDocument;


/**
 * A small class that add the ability to monitor the
 * keyboard's INSERT key and can switch between insert
 * and overwrite modes.
 * <p>
 * This requires that you add an OverwriteFilter using
 * the getDocument()->setDocumentFilter() method.
 * <p>
 * @author Randy L. Pearson
 * @see Document
 * @see JTextPane
 * @see DocumentFilter
 * @see OverwriteFilter
 */

public class JOverwritableTextPane extends JTextPane {

	/**
	 *
	 */
	private static final long serialVersionUID = 9092605442714879799L;

	/**
	 * Constants for the typing mode
	 *
	 */
	public static enum TypingMode {
		INSERT,
		OVERWRITE,
	}

	/**
	 * Default constructor.
	 *
	 */
	public JOverwritableTextPane() {
		super();
		initialize();
	}

	/**
	 * Overloaded constructor.
	 * @param doc The document to place inside the text pane.
	 * @see StyledDocument
	 */
	public JOverwritableTextPane(StyledDocument doc) {
		super(doc);
```

```
            initialize();
      }

      /**
       * The bound property name for the change notification.
       */
      public static final String TYPING_MODE_CHANGED_PROPERTY =
"TypingModeChanged";

      /**
       * The current typing mode.
       */
      protected TypingMode mode = TypingMode.INSERT;

      /**
       * Sets the active typing mode.
       * @param newMode The desired typing mode.
       * @see TypingMode
       */
      public void setTypingMode(TypingMode newMode) {
            TypingMode tmp = mode;
            mode = newMode;

            firePropertyChange(TYPING_MODE_CHANGED_PROPERTY, tmp,
mode );
      }

      /**
       * @return The current typing mode
       * @see TypingMode
       */
      public TypingMode getTypingMode() {
            return mode;
      }

      /**
       * Registers this object to listen for KeyEvents
       * so we can detect when the INSERT key is pressed.
       */
      private void initialize() {
            addKeyListener(new KeyAdapter() {

                  public void keyPressed(KeyEvent e) {
                        int key = e.getKeyCode();
                        if( key == KeyEvent.VK_INSERT) {
                              toggleInsertMode();
                        }
                  }
            });
      }

      /**
       * Toggles the current typing mode.
       */
      private void toggleInsertMode() {
            if( mode == TypingMode.INSERT )
```

```
                setTypingMode(TypingMode.OVERWRITE);
          else
                setTypingMode(TypingMode.INSERT);
    }
}
```

Now, adjust the EditorFrame class to use the new JOverwriteTextPane class by changing the constructor to read like this:

```
    private EditorFrame() {
          super("Simple Editor");
          // setDefaultCloseOperation( EXIT_ON_CLOSE );
          addWindowListener( new WindowAdapter() {
                public void windowClosing(WindowEvent event) {
                      System.exit(0);
                }
          });

          _textArea = new JOverwritableTextPane();
          JScrollPane textPane = new JScrollPane();
          textPane.setViewportView(_textArea);
          getContentPane().add( textPane, BorderLayout.CENTER );
    }
```

You must also change the _textArea declaration from the JTextArea class to the new JOverwriteTextPane class instead.

However, if you run the program, you will find overwrite mode still does not work. We've got more work to do still.

The problem is that we are not yet filtering and processing the keys while overwrite mode is enabled. It is possible to do this in the JOverwriteTextPane class by adjusting the code in the KeyPress handler, but Java supports an second (easier) way to do this.

We can create a DocumentFilter that acts as a kind of manager for the text document we are working on. The DocumentFilter is called before every insert, replace or remove operation and it is free to allow, deny or modify exactly what happens for each of these events.

Let's add the following DocumentFilter derived class to our project to demonstrate how this works.

```
package edu.uah.coned;

import javax.swing.text.AttributeSet;
import javax.swing.text.BadLocationException;
import javax.swing.text.DocumentFilter;
```

```
import edu.uah.coned.JOverwritableTextPane.TypingMode;


/**
 * This class implements a filter that supports an overwrite
 * mode for JOverwriteableTextPane documents.
 * <p>
 * <p>
 * This code only overwrites one character at a time, so normal
 * actions such as pasting text work as expected.  It also does
 * not remove end of line markers to avoid the strange effect of
 * lines getting automatically joined to each other.
 * <p>
 * @author Randy
 * @see JOverwritableTextPane
 * @see DocumentFilter
 */
public class OverwriteFilter extends DocumentFilter {

      private JOverwritableTextPane _pane = null;

      /**
       * Constructor.
       * @param pane The pane to which overwrite support is added.
       */
      public OverwriteFilter(JOverwritableTextPane pane) {
            _pane = pane;
      }

      /**
       * Called everytime a character/string is replaced in the
document.
       * If the pane's OVERWRITE mode is active, the next character in
       * the document is removed before the typed character is inserted.
       */
      public void replace(FilterBypass fb, int offset, int length,
String text, AttributeSet attrs) throws BadLocationException {
            if( _pane.getTypingMode() == TypingMode.OVERWRITE ) {
                  if( offset < fb.getDocument().getLength() ) {
                        String nextChars = fb.getDocument().getText
(offset,2);
                        // This prevents joining one line to the next in
overwrite mode.
                        if( nextChars.charAt(0) != 13 &&
nextChars.charAt(0) != 10 ) {
                              fb.remove(offset,1);
                        }
                  }
            }
            // Let base filter do its thing
            super.replace(fb, offset, length, text, attrs);
    }
}
```

Notice this class overrides the default 'replace' action and when in

overwrite mode, it first removes the character at the current offset within the document prior to the insertion of the just typed character to achieve the overwrite effect.

Now all we have to do is to force our text area to use the new filter class by changing the EditorFrame constructor to read as follows:

```
      private EditorFrame() {
            super("Simple Editor");
            // setDefaultCloseOperation( EXIT_ON_CLOSE );
            addWindowListener( new WindowAdapter() {
                  public void windowClosing(WindowEvent event) {
                        System.exit(0);
                  }
            });

            _textArea = new JOverwritableTextPane();
            _document = (AbstractDocument) _textArea.getDocument();
            _document.setDocumentFilter( new OverwriteFilter
(_textArea) );
            JScrollPane textPane = new JScrollPane();
            textPane.setViewportView(_textArea);
            getContentPane().add( textPane, BorderLayout.CENTER );
      }
```

If you run the program again, you should see that overwrite mode now works.

NOTE: This class could probably be improved slightly by removing the reference to the JOverwriteTextPane class and adding a new method to the filter called setTypingMode(). That is left as an exercise for the student.

Step 17 – Fix focus

I am sure many of you may have noticed that when we run the program, the cursor is not placed inside the text area initially. Instead the first toolbar button has a dotted focus box drawn around it and by pressing the TAB key, you can move from button to button until finally the text area gets the focus. Once the text area gets the focus, it keeps it.

Let's fix this little annoyance by changing our ActionItem's addToToolbar () method as shown below:

```
      public ActionItem addToToolbar( JToolBar toolbar ) {
            JButton button = toolbar.add( this );
```

```
            button.setFocusable(false);

            if( button.getIcon() == null ) {
                  toolbar.remove( button );
            } else {
                  button.setText( "" );
                  if( _tooltip != null ) {
                        button.setToolTipText( _tooltip );
                  }
            }
      }
```

NOTE: Also discuss Java's new Focus management system including the FocusEvent, KeyboardFocusManager, the abstract FocusTraversalPolicy class, and the concrete classes ContainerOrderFocusTraversalPolicy, DefaultFocusTraversalPolicy (AWT default), InternalFrameFocusTraversalPolicy, LayoutFocusTraversalPolicy (Swing/mixed default) and SortingFocusTraversalPolicy.

Step 18 – Open and save operations

Now that our editor is working fairly well, let's turn our attention to getting operations like **File > Open** and **File > Save** working.

First, let's implement the doFileNew() and doFileOpen() methods as shown below:

```
      public void doNewAction() {
            _textArea.setText( "" );
      }

      public void doOpenAction() {
            JFileChooser chooser = new JFileChooser();
            chooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
            if( chooser.showOpenDialog(this) ==
JFileChooser.APPROVE_OPTION ) {
                  openFile( chooser.getSelectedFile().toString() );
            }
      }
```

Next, add and enter the openFile() method as shown below:

```
      private boolean openFile(String filename) {
            try {
                  BufferedReader reader = new BufferedReader( new
FileReader(filename) );

                  String bigString = "";
                  String littleString = "";
                  String eol =  System.getProperty( "line.separator" );
```

```
                do {
                      littleString = reader.readLine();
                      if( littleString != null ) {
                            littleString += eol;
                            bigString += littleString;
                      }
                }
                while( littleString != null );

                reader.close();

                _textArea.setText( bigString );
                _statusBar.setStatus( "Loaded " + filename );
                return true;
          } catch( IOException e ) {
                JOptionPane.showMessageDialog( this, "Error: " +
e.getMessage() );
          }
          _statusBar.setStatus( "Failed to read " + filename );
          return false;
     }
```

You should be able to run the program and use the **File > New** and **File > Open** options now.

This seems to work pretty well, however, if you try to open a file that is very large (> ~5,000 lines) the program will appear to hang. If you wait long enough, the program will finally open and display the file, but we obviously have a performance problem we should try to solve.

Step 19 – Improving file reading performance

Here is a more efficient version of the openFile() method:

```
private boolean openFile(String filename) {
     try {
          File file = new File( filename );

          if( !file.exists() )
                throw new IOException(filename + " does not exist!");

          if( file.length() > Integer.MAX_VALUE )
                throw new IOException(filename + " Oops...  Too
large!");

          char[] buffer = new char[(int) file.length()];

          FileReader reader = new FileReader(filename);

          reader.read(buffer);

          _textArea.setText( String.copyValueOf(buffer) );
```

```
            reader.close();
            _statusBar.setStatus( "Loaded " + filename );

            return true;
      } catch( IOException e ) {
            JOptionPane.showMessageDialog( this, "Error: " +
e.getMessage() );
      }
      _statusBar.setStatus( "Failed to read " + filename );
      return false;
}
```

Notice how this version reads the entire file into a character array and then converts that into a string. This is much faster than parsing the file one line at a time. Naturally, your programs may need to read many different kinds of data, perhaps one line at a time.

Step 20 – Saving files

Let's write a function that can save our text to a file. This will be used by both the doFileSave() and doFileSaveAs() methods. Add a new method to the EditorFrame called saveFile() as shown below:

```
private boolean saveFile( String filename ) {
      try {
            BufferedWriter writer = new BufferedWriter( new FileWriter
(filename) );
            writer.write( _textArea.getText() );
            writer.close();
            return true;
      } catch( IOException e ) {
            JOptionPane.showMessageDialog( this, "Error: " +
e.getMessage() );
      }
      return false;
}
```

Now, we can complete the doFileSaveAs() method. Here it is:

```
public void doFileSaveAs() {
      JFileChooser chooser = new JFileChooser();
      chooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
      int option = chooser.showSaveDialog(this);
      if( option == chooser.showSaveDialog(this) ) {
            File theFile = chooser.getSelectedFile();
            if( theFile.exists() ) {
                  option = JOptionPane.showConfirmDialog(
                        this,
                        theFile.toString() + " already exists.
Overwrite?",
                        "File Exists",
```

```
                        JOptionPane.YES_NO_OPTION,
                        JOptionPane.QUESTION_MESSAGE );
                  if( option != JOptionPane.YES_OPTION )
                        return;
            }
            saveFile( theFile.toString() );
      }
}
```

Run the program and test that it works as expected.

Step 21 – Implementing the doFileSave() method

Now that we have the code in place to save and open files, we can turn our attention to the other methods.

First, in order to support a normal save (as opposed to save as), we need to remember the name of the file we lasted opened or saved.

First, add a new instance variable to the EditorFrame class that looks like this:

```
      private String currentFile = null;
```

Now, modify the saveFile() method so it updates the variable after the file is written. It should now look like this:

```
private boolean saveFile( String filename ) {
      try {
            BufferedWriter writer = new BufferedWriter( new FileWriter
(filename) );
            writer.write( _textArea.getText() );
            writer.close();
            currentFile = filename;
            return true;
      } catch( IOException e ) {
            JOptionPane.showMessageDialog( this, "Error: " +
e.getMessage() );
      }
      return false;
}
```

You should also modify the doFileNew() method to clear this (to avoid nasty surprises) as shown below:

```
public void doNewAction() {
      _textArea.setText( "" );
      _currentFile = null;
}
```

And also edit the doFileOpen() method to update the filename after a successful open as shown here:

```
public void doOpenAction() {
      JFileChooser chooser = new JFileChooser();
      chooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
      if( chooser.showOpenDialog(this) == JFileChooser.APPROVE_OPTION )
{
            if( openFile( chooser.getSelectedFile().toString() ) )
                  _currentFile = chooser.getSelectedFile().toString();
      }
}
```

Now we can finally implement doFileSave() to work like this:

```
public void doFileSave() {
      if( _currentFile != null )
            saveFile( _currentFile );
      else
            doFileSaveAs();
}
```

You may also want to display the current file name in the title of the program as a reminder for the user. You can do so using the setTitle() method.

Step 22 – Prompt to save changes

Most modern programs will ask the user if they want to save their changes before the program exits, opens a different file or closes the current file. In order to add that support to our program, we need to query our text pane and find out if the text was modified and then prompt the user at the appropriate times.

It would be very handy if the text area tracked this for us, but unfortunately, it does not. We have to add this support ourselves.

We need to detect when the user inserts or deletes any characters in the text document. One method for doing this is by registering a listener for all keyboard events with the text pane object.

You could modify the initialize() method as shown below:

```
      // ... Same as before ...

      _statusBar = new JStatusBar( JStatusBar.ALL_PANES );
      getContentPane().add( _statusBar, BorderLayout.SOUTH );
```

```
	_textArea.addKeyListener( new KeyAdapter() {
		public void keyTyped( KeyEvent e ) {
			switch( e.getKeyChar() )
			{
				case KeyEvent.VK_LEFT:
				case KeyEvent.VK_RIGHT:
				case KeyEvent.VK_DOWN:
				case KeyEvent.VK_UP:
				case KeyEvent.VK_PAGE_UP:
				case KeyEvent.VK_PAGE_DOWN:
					break;
				default:
					_textModified = true;
			}
		}
	});
```

The technique shown above works, but sometimes get some false hits. A better way to do this is using a DocumentFilter. Since we already have a custom DocumentFilter, let's make a smarter version by building a new DocumentFilter that extends the OverwriteDocumentFilter class.

Create a new class named ModifiedFilter and enter the following code:

```
package edu.uah.coned;

import javax.swing.text.AttributeSet;
import javax.swing.text.BadLocationException;
import javax.swing.text.DocumentFilter;

public class ModifiedFilter extends OverwriteFilter {

	private boolean _textModified = false;

	public ModifiedFilter(JOverwritableTextPane pane) {
		super(pane);
	}

	public boolean isTextModified() {
		return _textModified;
	}

	public void clearTextModified() {
		_textModified = false;
	}

	public void insertString(DocumentFilter.FilterBypass fb, int
offset, String string, AttributeSet attr )  throws BadLocationException
{
		super.insertString( fb, offset, string, attr );
		_textModified = true;
	}

	public void remove(DocumentFilter.FilterBypass fb, int offset, int
```

```
length )  throws BadLocationException {
            super.remove( fb, offset, length );
            _textModified = true;
      }

      public void replace(DocumentFilter.FilterBypass fb, int offset,
int length, String text, AttributeSet attr )  throws
BadLocationException {
            super.replace( fb, offset, length, text, attr );
            _textModified = true;
      }
}
```

Now, add the following variable to the EditorFrame class:

```
      ModifiedFilter _filter = null;
```

Change the initialize() method to create a ModifiedFilter instead of a OverwriteFilter (because of inheritance, a ModifiedFilter is also an OverwriteFilter). Here is the modified code:

```
      _textArea = new JOverwritableTextPane();
      _filter = new ModifiedFilter( _textArea );
      _document.setDocumentFilter( _filter );
      JScrollPane textPane = new JScrollPane();
      textPane.setViewportView(_textArea);
      getContentPane().add( textPane, BorderLayout.CENTER );
```

Finally, add calls to _filter.clearTextModifed() as appropriate. This should be called in following methods:

```
      doNewAction()
      doOpenAction()
      saveFile()
```

Now, we add a new method that checks to see if the text was modified and prompts the user to save their changes. Call this method checkAndConfirmSave() and edit it to read as follows:

```
private boolean checkAndConfirmSave() {
      if( _filter.isTextModified() ) {
            int option;

            option = JOptionPane.showConfirmDialog( this,
                        "Text has been modified.  Save changes?",
                        "Save Modifications",
                        JOptionPane.YES_NO_CANCEL_OPTION,
                        JOptionPane.QUESTION_MESSAGE);

            if( option == JOptionPane.NO_OPTION )
                  return true;
```

```
            else if( option == JOptionPane.CANCEL_OPTION )
                  return false;

            return doFileSave();
      }
      return true;
}
```

The function requires that doFileSave() returns a boolean also, so modify that function to read like this:

```
      public boolean doFileSave() {
            if( _currentFile != null )
                  return saveFile( _currentFile );
            else
                  return doFileSaveAs();
      }
```

That means doFileSaveAs() must also be modified to return a boolean too, like this:

```
public boolean doFileSaveAs() {
      JFileChooser chooser = new JFileChooser();
      chooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
      int option = chooser.showSaveDialog(this) );
      if( JFileChooser.APPROVE_OPTION == option ) {
            File theFile = chooser.getSelectedFile();
            if( theFile.exists() ) {
                  int option = JOptionPane.showConfirmDialog(
                              this,
                              theFile.toString() + " already exists.
Overwrite?",
                              "File Exists",
                              JOptionPane.YES_NO_OPTION,
                              JOptionPane.QUESTION_MESSAGE );
                  if( option != JOptionPane.YES_OPTION )
                        return false;
            }
            return saveFile( theFile.toString() );
      }
      return false;
}
```

Whew, one small change does sometimes cause these kinds of ripples.

Now, we are finally ready to make use of the new text modified option to improve our program. We want to call the checkAndConfirmSave() method before creating a new empty file, opening a file or exiting the program.

Start by modifying doFileNew() so it reads like this:

```
public void doFileNew() {
	if( !checkAndConfirmSave() )
		return;

	_textArea.setText( "" );
	_currentFile = null;
	_filter.clearTextModified();
	setTitle( "Simple Editor" );
}
```

Next, modify doFileOpen() to read as follows:

```
public void doFileOpen() {
	if( !checkAndConfirmSave() )
		return;

	JFileChooser chooser = new JFileChooser();
	chooser.setFileSelectionMode( JFileChooser.FILES_ONLY );
	int option = chooser.showOpenDialog(this);
	if( option == JFileChooser.APPROVE_OPTION ) {
		if( openFile( chooser.getSelectedFile().toString() ) ) {
			_currentFile = chooser.getSelectedFile().toString();
			_filter.clearTextModified();
			setTitle( "Simple Editor - " + _currentFile );
		}
	}
}
```

Also modify the doFileExit() to read like this:

```
public void doFileExit() {
	if( !checkAndConfirmSave() )
		return;
	System.exit(0);
}
```

And finally, modify the EditorFrame constructor so we also do this when the user tries to close the main window.

```
private EditorFrame() {
	super("Simple Editor");
	setDefaultCloseOperation( DO_NOTHING_ON_CLOSE );
	addWindowListener( new WindowAdapter() {
		public void windowClosing(WindowEvent event) {
			doFileExit();
		}
	});
}
```

Hopefully you noticed that if don't process the windowClosing() event ourselves, we could not check and save changes to the document.

Try it out and make sure the program is working as expected.

Step 23 – Supporting Copy, Cut and Paste operations

This should be easy since text areas pretty much handle this themselves. Start by revising our doEditCopy(), doEditCut() and doEditPaste() functions as shown below:

```
public void doEditCopy() {
      _textArea.copy();
}

public void doEditCut() {
      _textArea.cut();
}

public void doEditPaste() {
      _textArea.paste();
}
```

If you run the program, you will find that it works just fine. However, we really should try to disable edit, copy and paste on the menu when there is nothing available to be copied or pasted. While not critical, it would give our program a more professional feel to users.

Step 24 – Disabling and enabling Copy, Cut and Paste

The copy and cut operations should only be enabled when the user selects some text within the window. This means we need to be informed when the user does this. Java's answer to this is called a CaretListener.

Modify the initialize() method and register a caret listener:

```
      _textArea.addCaretListener( new CaretListener() {

            public void caretUpdate(CaretEvent e) {
                  boolean isSelected = false;
                  int dot = e.getDot();
                  int mark = e.getMark();

                  if( dot != mark )
                        isSelected = true;

                  _editCopy.setEnabled(isSelected);
                  _editCut.setEnabled(isSelected);
            }
      });
```

In order to control our Paste action, we need to monitor the system clipboard. Add the following code also:

```
            Clipboard clipboard = getToolkit().getSystemClipboard();

            clipboard.addFlavorListener(new FlavorListener() {

                  public void flavorsChanged(FlavorEvent e) {
                        Clipboard clipboard = (Clipboard) e.getSource();
                        boolean canPaste =
clipboard.isDataFlavorAvailable( DataFlavor.plainTextFlavor );

                        _editPaste.setEnabled(canPaste);
                  }

            });
```

Try it out.

Step 25 – Printing our documents

See Pages: 539-575

Printing support has improved drastically since the early days of Java. The first versions of Java did not have any kind of native printing support at all. Instead all you could really do was grab a copy of the display and send it to a printer.

Unfortunately, even the best graphics displays available today do not have very many dots per inch when compared to even low quality laser printers. Typically monitors operate at less than 100 dpi, while most laser can print 300 dpi or more. The results were not pretty to say the least, especially when the graphics are stretched to maintain the correct relative size on the paper.

Printing from Java programs requires using several different classes. The following describes the more important classes:

Printable interface

The class that does the actual printing must provide a method like this:

```
int print( Graphics g, PageFormat pf, int pageNum );
```

The Graphics object is a standard class used for all drawing within Java. It provides many drawing methods such as drawArc(), drawChars(),

drawImage(), drawLine(), drawRect(), drawString() and many others.

The PageFormat object describes the properties of a single page such as its width, height and any margins that are in effect.

The final parameter is the page currently being printed, with the first page as page 0.

Once the printing actually starts, Java will repeatedly call this method over and over. The method must return either NO_SUCH_PAGE or PAGE_EXISTS. Returning NO_SUCH_PAGE terminates the printing process.

PrinterJob class

This class is used to initiate the printing process. Most often, you will use it like this:

```
Printable doc = { ... };
PrinterJob job = PrinterJob.getPrinterJob();
job.setPrintable( doc );

HashPrintRequestAttributeSet attrs = new HashPrintRequestAttributeSet();
if( job.printDialog(attrs) )
{
      try
      {
            job.print( attrs );
      }
      catch( PrinterException ex )
      {
            ...
      }
}
```

That will create a new job that is associated with the default printer installed on the system, then displays a printer selection dialog box by calling the printDialog() method. That will return true if the user clicks on OK, so now we can finally print the document by calling the print() method, which will in turn repeatedly call the print() method of the Printable object.

However, since Java 1.4 has made it even easier to print common types of document by adding direct printing support for many different flavors of documents. Collectively the built-in printing support is accessed via the PrinterServiceLookup class.

Let's add printing support to our project using the new services.

Implement the doPrint() method as shown below:

```
public void doPrint() {
      try {
            DocFlavor flavor = DocFlavor.STRING.TEXT_PLAIN;
            PrintService[] services =
PrintServiceLookup.lookupPrintServices( flavor, null );
            int printerToUse = 0;

            if( services.length == 0 ) {
                  JOptionPane.showMessageDialog( this, "No printers are
defined!", "No Printers Available", JOptionPane.ERROR_MESSAGE );
                  return;
            }

            String[] printers = new String[services.length];
            for (int i = 0; i < services.length; i++) {
                  printers[i] = services[i].getName();
            }

            if( services.length > 1 ) {
                  String printer = (String) JoptionPane.showInputDialog(
                              this,
                              "Which printer should I print to",
                              "Select printer",
                              JOptionPane.QUESTION_MESSAGE,
                              null,
                              printers,
                              printers[0]);

                  if( printer == null ) {
                        return;
                  }

                  for (int i = 0; i < printers.length; i++) {
                        if( printers[i] == printer ) {
                              printerToUse = i;
                              break;
                        }
                  }
            }

            DocPrintJob job = services[printerToUse].createPrintJob();
            Doc doc = new SimpleDoc( _textArea.getText(), flavor,
null );
            job.print( doc, null );
      } catch( Exception ex ) {
            JOptionPane.showMessageDialog( this, "Error: " +
ex.getLocalizedMessage() );
      }
}
```

As you can see, using the built-in support is very easy. Most of our work involved getting the list of available printers and displaying them to the user.

Step 26 – Adding Search and Replace support (Time Permitting)

No respectable editor is done until it supports searching and replacing for text. We will need to first add some new actions to our menu and toolbar.

Create 3 new action classes named EditSearchAction, EditSearchAgainAction and EditReplaceAction. Complete the classes are shown below:

```
package edu.uah.coned;

import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;

@SuppressWarnings("serial")
public class EditFindAction extends ActionItem {

	public EditFindAction() {
		super("Find", 'F', KeyEvent.CTRL_MASK, KeyEvent.VK_F, "Find
text", "images/Find16.gif");
	}

	public void actionPerformed(ActionEvent e) {
		EditorFrame.getInstance().doEditFind();

	}

}

package edu.uah.coned;

import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;

@SuppressWarnings("serial")
public class EditFindAgainAction extends ActionItem {

	public EditFindAgainAction() {
		super( "Find Next", 'N', 0, KeyEvent.VK_F3, "Find next",
"images/FindAgain16.gif");
	}

	public void actionPerformed(ActionEvent e) {
		EditorFrame.getInstance().doEditFindAgain();
	}

}
```

```
package edu.uah.coned;

import java.awt.event.ActionEvent;
import java.awt.event.KeyEvent;

@SuppressWarnings("serial")
public class EditReplaceAction extends ActionItem {

      public EditReplaceAction() {
            super("Replace", 'R', KeyEvent.CTRL_MASK, KeyEvent.VK_H,
"Search and replace", "images/Replace16.gif");
      }

      public void actionPerformed(ActionEvent e) {
            EditorFrame.getInstance().doEditReplace();

      }

}
```

Of course, you must also add the doEditFind(), doEditFindAgain() and doEditReplace() methods to the EditorFrame. We will code those methods up soon.

First, we have to make a decision. The find action could use the JOptionPane class (see the showInputDialog() method) to request the string to be found from the user. However, if we want to allow the user to enter various options such as a checkbox for case-sensitivity, regular expression support and so on, then we need to build our own input dialog. Since we have to build a custom dialog for the replace action anyway, we might as well do one for the standard find action also.

Start by creating a new class named FindDialog that derives from the JDialog class and also implements the ActionListener interface.

Implement the class as shown below:

```
package edu.uah.coned;

import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Box;
import javax.swing.JButton;
import javax.swing.JCheckBox;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JTextField;
```

```
@SuppressWarnings("serial")
public class ReplaceDialog extends JDialog implements ActionListener {

	private JTextField _textFind;
	private JTextField _textReplace;
	private JCheckBox _chkIgnoreCase;
	private JCheckBox _chkReplaceAll;
	private boolean _okClicked = false;

	private ReplaceDialog( Frame frame ) {
		super(frame, "Replace Text", true);

		_textFind = new JTextField();
		_textReplace = new JTextField();
		_chkIgnoreCase = new JCheckBox("Ignore Case");
		_chkReplaceAll = new JCheckBox("Replace All");

		Box vBox = Box.createVerticalBox();

		vBox.add( Box.createVerticalGlue());
		vBox.add(Box.createVerticalStrut(5));
		add(vBox);

		Box box = Box.createHorizontalBox();

		box.add(Box.createHorizontalGlue());
		box.add(new JLabel("Find:"));
		box.add(Box.createHorizontalStrut(5));
		box.add(_textFind);
		box.add(Box.createHorizontalGlue());

		vBox.add(box);
		vBox.add( Box.createVerticalStrut(5));

		box = Box.createHorizontalBox();
		box.add(Box.createHorizontalGlue());
		box.add(new JLabel("Replace:"));
		box.add(Box.createHorizontalStrut(5));
		box.add(_textReplace);
		box.add(Box.createHorizontalGlue());

		vBox.add(box);
		vBox.add(Box.createVerticalStrut(5));

		box = Box.createHorizontalBox();
		box.add(Box.createHorizontalGlue());
		box.add(_chkIgnoreCase);
		box.add(Box.createHorizontalStrut(5));
		box.add(_chkReplaceAll);
		box.add(Box.createHorizontalGlue());

		vBox.add(box);
		vBox.add(Box.createVerticalStrut(5));

		JButton okButton = new JButton("OK");
		okButton.setActionCommand("OK");
```

```
        okButton.addActionListener(this);

        JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(this);

        okButton.setPreferredSize(cancelButton.getPreferredSize());

        box = Box.createHorizontalBox();
        box.add(Box.createHorizontalGlue());
        box.add(Box.createHorizontalStrut(20));
        box.add(okButton);
        box.add(Box.createHorizontalStrut(20));
        box.add(cancelButton);
        box.add(Box.createHorizontalStrut(20));
        box.add(Box.createHorizontalGlue());

        vBox.add(box);
        vBox.add(Box.createVerticalStrut(5));
        vBox.add(Box.createVerticalGlue());

        getRootPane().setDefaultButton(okButton);

        pack();
        setResizable(false);
        setLocationRelativeTo(frame);
    }

    public static ReplaceDialog createDialog( Frame owner ) {
        ReplaceDialog dialog = new ReplaceDialog( owner );

        return dialog;
    }

    public static ReplaceDialog createDialog( Frame owner,
                        String findText,
                        String replaceText,
                        boolean ignoreCase,
                        boolean replaceAll ) {

        ReplaceDialog dialog = new ReplaceDialog( owner );

        dialog.setFindText( findText );
        dialog.setReplaceText( replaceText );
        dialog.setIgnoreCase( ignoreCase );
        dialog.setReplaceAll( replaceAll );

        return dialog;
    }

    public boolean showDialog() {
        setVisible(true);
        return _okClicked;
    }

    public String getFindText() {
        return _textFind.getText();
```

```
	}

	public void setFindText( String text ) {
		_textFind.setText( text );
	}

	public String getReplaceText() {
		return _textReplace.getText();
	}

	public void setReplaceText( String text ) {
		_textReplace.setText( text );
	}

	public boolean getIgnoreCase() {
		return _chkIgnoreCase.isSelected();
	}

	public void setIgnoreCase( boolean ignoreCase ) {
		_chkIgnoreCase.setSelected( ignoreCase );
	}

	public boolean getReplaceAll() {
		return _chkReplaceAll.isSelected();
	}

	public void setReplaceAll( boolean replaceAll ) {
		_chkReplaceAll.setSelected( replaceAll );
	}

	public void actionPerformed(ActionEvent e) {
		_okClicked = false;

		if( "OK".equals(e.getActionCommand())) {
			_okClicked = true;
		}

		setVisible(false);
	}

}
```

NOTE: This class could be implemented a number of different ways.

Now, let's code up the doEditFind() and doEditFindAgain() methods.

Here is the doEditFind() method:

```
public void doEditFind() {
	FindDialog dlg = FindDialog.createDialog( this,
					_findText,
					_ignoreCase );

	if( !dlg.showDialog() )
```

```
            return;

      _findText = dlg.getFindText();
      _ignoreCase = dlg.getIgnoreCase();

      if( _findText.length() == 0 ) {
            JOptionPane.showMessageDialog( this,
                  "You did not enter anything!",
                  "Missing Value",
                  JOptionPane.INFORMATION_MESSAGE );
            return;
      }

      doEditFindAgain();
}
```

Of course, you must add the _findText and _ignoreCase fields to the EditorFrame class also.

Notice how this method will use the doEditFindAgain() method, which follows:

```
public void doEditFindAgain() {

      if( _findText == null ) {
            doEditFind();
            return;
      }

      int caretPos = _textArea.getCaretPosition();
      int docLength = _textArea.getText().length();

      if( _findText.length() >= docLength ) {
            showNotFoundMessage();
            return;
      }

      if( caretPos >= docLength ) {
            caretPos = 0;
      }

      try {
            if( findInText(
                  _textArea.getText( caretPos, _findText.length() ),
                  _findText,
                  _ignoreCase ) == 0 )

                  caretPos++;

            int where = findInText(
                  _findText,
                  _textArea.getText(caretPos, docLength-caretPos),
                   _ignoreCase );
```

```
            if( where != -1 ) {
                  _textArea.setSelectionStart( caretPos + where );
                  _textArea.setSelectionEnd( caretPos + where +
                        _findText.length() );
                  _statusBar.setStatus( "Search for '" +
                        _findText + "' succeeded.");
            } else {
                  if( caretPos > 0 ) {
                        int option = JOptionPane.showConfirmDialog(this,
                              "End of document reached.  Try again from
beginning?",
                              "Try again?",
                              JOptionPane.YES_NO_OPTION,
                              JOptionPane.QUESTION_MESSAGE);

                        if( option == JOptionPane.YES_OPTION ) {

                              where = findInText( _findText,
                                    _textArea.getText(0, caretPos-1),
                                    _ignoreCase );

                              if( where > 0 ) {
                                    _textArea.setSelectionStart
( where );
                                    _textArea.setSelectionEnd( where +
_findText.length() );
                                    _statusBar.setStatus( "Search for '"
+ _findText + "' succeeded.");
                              } else {
                                    showNotFoundMessage();
                              }
                        } else {
                              showNotFoundMessage();
                        }
                  }
            }
      } catch(Exception ex) {
            _statusBar.setStatus("Error: " + ex.getLocalizedMessage() );
      }
}
```

Let examine some of this code in more depth.

```
      if( _findText == null ) {
            doEditFind();
            return;
      }
```

First, these lines make sure that the user has entered a value to search for using the FindDialog input form.

```
      int caretPos = _textArea.getCaretPosition();
      int docLength = _textArea.getText().length();
```

```
      if( _findText.length() >= docLength ) {
            showNotFoundMessage();
            return;
      }

      if( caretPos >= docLength ) {
            caretPos = 0;
      }
```

These lines get the current position of the caret within the document and also check that the text we are search for is not longer than the entire document.

The final if statement forces the search to start from the beginning of the document if the caret is currently at the very end of the document.

```
            if( findInText(
                  _textArea.getText( caretPos, _findText.length() ),
                  _findText,
                  _ignoreCase ) == 0 )

                  caretPos++;
```

This section of code forces the search to move 1 character forward so we do not find the same string again and again, but only if the caret is currently positioned directly on the word we are searching for.

```
int where = findInText(
            _findText,
            _textArea.getText(caretPos, docLength-caretPos),
             _ignoreCase );

if( where != -1 ) {
      _textArea.setSelectionStart( caretPos + where );
      _textArea.setSelectionEnd( caretPos + where +
            _findText.length() );
      _statusBar.setStatus( "Search for '" +
            _findText + "' succeeded.");
} else {
      if( caretPos > 0 ) {
            int option = JOptionPane.showConfirmDialog(this,
                  "End of document reached.  Try again from beginning?",
                  "Try again?",
                  JOptionPane.YES_NO_OPTION,
                  JOptionPane.QUESTION_MESSAGE);

            if( option == JOptionPane.YES_OPTION ) {

                  where = findInText( _findText,
                        _textArea.getText(0, caretPos-1),
                        _ignoreCase );

                  if( where > 0 ) {
```

```
                        _textArea.setSelectionStart( where );
                        _textArea.setSelectionEnd( where +
_findText.length() );
                        _statusBar.setStatus( "Search for '" + _findText
+ "' succeeded.");
                  } else {
                        showNotFoundMessage();
                  }
            } else {
                  showNotFoundMessage();
            }
      }
}
```

This code calls the helper function findInText() and then analyzes the result. If we found the text, then the setSelectionStart() and setSelectionEnd() methods are used to highlight the text, otherwise we offer to search again from the beginning of the file.

Finally, here is the findInText() method itself. It is pretty straight forward, although in order to perform the case-insensitive search, a bit of memory is wasted copying the strings as they are converted to lower case.

```
private int findInText( String needle, String haystack, boolean
ignoreCase ) {
      if( !ignoreCase ) {
            return haystack.indexOf( needle );
      }

      String haystack2 = haystack.toLowerCase();
      String needle2 = needle.toLowerCase();

      return haystack2.indexOf(needle2);
}
```

After testing this code, let implement the search and replace feature.

Start by adding a ReplaceDialog class that derives from JDialog and implements the ActionListener interface, much like we did for the FindDialog. Here is the code, which is very similar:

```
package edu.uah.coned;

import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.Box;
import javax.swing.JButton;
```

```
import javax.swing.JCheckBox;
import javax.swing.JDialog;
import javax.swing.JLabel;
import javax.swing.JTextField;

@SuppressWarnings("serial")
public class ReplaceDialog extends JDialog implements ActionListener {

	private JTextField _textFind;
	private JTextField _textReplace;
	private JCheckBox _chkIgnoreCase;
	private JCheckBox _chkReplaceAll;
	private boolean _okClicked = false;

	private ReplaceDialog( Frame frame ) {
		super(frame, "Replace Text", true);

		_textFind = new JTextField();
		_textReplace = new JTextField();
		_chkIgnoreCase = new JCheckBox("Ignore Case");
		_chkReplaceAll = new JCheckBox("Replace All");

		Box vBox = Box.createVerticalBox();

		vBox.add( Box.createVerticalGlue());
		vBox.add(Box.createVerticalStrut(5));
		add(vBox);

		Box box = Box.createHorizontalBox();

		box.add(Box.createHorizontalGlue());
		box.add(new JLabel("Find:"));
		box.add(Box.createHorizontalStrut(5));
		box.add(_textFind);
		box.add(Box.createHorizontalGlue());

		vBox.add(box);
		vBox.add( Box.createVerticalStrut(5));

		box = Box.createHorizontalBox();
		box.add(Box.createHorizontalGlue());
		box.add(new JLabel("Replace:"));
		box.add(Box.createHorizontalStrut(5));
		box.add(_textReplace);
		box.add(Box.createHorizontalGlue());

		vBox.add(box);
		vBox.add(Box.createVerticalStrut(5));

		box = Box.createHorizontalBox();
		box.add(Box.createHorizontalGlue());
		box.add(_chkIgnoreCase);
		box.add(Box.createHorizontalStrut(5));
		box.add(_chkReplaceAll);
		box.add(Box.createHorizontalGlue());
```

```
        vBox.add(box);
        vBox.add(Box.createVerticalStrut(5));

        JButton okButton = new JButton("OK");
        okButton.setActionCommand("OK");
        okButton.addActionListener(this);

        JButton cancelButton = new JButton("Cancel");
        cancelButton.addActionListener(this);

        okButton.setPreferredSize(cancelButton.getPreferredSize());

        box = Box.createHorizontalBox();
        box.add(Box.createHorizontalGlue());
        box.add(Box.createHorizontalStrut(20));
        box.add(okButton);
        box.add(Box.createHorizontalStrut(20));
        box.add(cancelButton);
        box.add(Box.createHorizontalStrut(20));
        box.add(Box.createHorizontalGlue());

        vBox.add(box);
        vBox.add(Box.createVerticalStrut(5));
        vBox.add(Box.createVerticalGlue());

        getRootPane().setDefaultButton(okButton);

        pack();
        setResizable(false);
        setLocationRelativeTo(frame);
    }

    public static ReplaceDialog createDialog( Frame owner ) {
        ReplaceDialog dialog = new ReplaceDialog( owner );

        return dialog;
    }

    public static ReplaceDialog createDialog( Frame owner,
        String findText,
        String replaceText,
        boolean ignoreCase,
        boolean replaceAll ) {

        ReplaceDialog dialog = new ReplaceDialog( owner );

        dialog.setFindText( findText );
        dialog.setReplaceText( replaceText );
        dialog.setIgnoreCase( ignoreCase );
        dialog.setReplaceAll( replaceAll );

        return dialog;
    }

    public boolean showDialog() {
        setVisible(true);
```

```
            return _okClicked;
      }

      public String getFindText() {
            return _textFind.getText();
      }

      public void setFindText( String text ) {
            _textFind.setText( text );
      }

      public String getReplaceText() {
            return _textReplace.getText();
      }

      public void setReplaceText( String text ) {
            _textReplace.setText( text );
      }

      public boolean getIgnoreCase() {
            return _chkIgnoreCase.isSelected();
      }

      public void setIgnoreCase( boolean ignoreCase ) {
            _chkIgnoreCase.setSelected( ignoreCase );
      }

      public boolean getReplaceAll() {
            return _chkReplaceAll.isSelected();
      }

      public void setReplaceAll( boolean replaceAll ) {
            _chkReplaceAll.setSelected( replaceAll );
      }

      public void actionPerformed(ActionEvent e) {
            _okClicked = false;

            if( "OK".equals(e.getActionCommand())) {
                  _okClicked = true;
            }

            setVisible(false);
      }
}
```

And now you can implement the doEditReplace() method as follows:

```
public void doEditReplace() {
      ReplaceDialog dlg = ReplaceDialog.createDialog( this,
                              _findText,
                              _replaceText,
                              _ignoreCase,
                              _replaceAll );
```

```
      if( !dlg.showDialog() )
            return;

      _findText = dlg.getFindText();
      _replaceText = dlg.getReplaceText();
      _ignoreCase = dlg.getIgnoreCase();
      _replaceAll = dlg.getReplaceAll();

      if( _findText.length() == 0 ) {
            JOptionPane.showMessageDialog( this,
                  "You did not enter anything!", "Missing Value",
                  JOptionPane.INFORMATION_MESSAGE );
            return;
      }

      if( _findText.length() >= _textArea.getText().length() ) {
            showNotFoundMessage();
            return;
      }

      try {
            int where = 0;
            int lastWhere = 0;
            int replaceCount = 0;

            do {
                  lastWhere = where;
                  where = findInText( _findText, _textArea.getText
(where, _textArea.getText().length() - where), _ignoreCase );

                  if( where != -1 ) {
                        boolean replace = _replaceAll;

                        _textArea.setSelectionStart( lastWhere +
where );
                        _textArea.setSelectionEnd( lastWhere + where +
_findText.length() );

                        if( !_replaceAll ) {
                              int option = JOptionPane.showConfirmDialog
( this, "Replace this occurrance?", "Confirm Replacment",
JOptionPane.YES_NO_CANCEL_OPTION );
                              if( option == JOptionPane.YES_OPTION ) {
                                    replace = true;
                              } else if( option ==
JOptionPane.CANCEL_OPTION ) {
                                    break;
                              }
                        }

                        if( replace ) {
                              _textArea.replaceSelection
( _replaceText );
                              where += _replaceText.length();
                              replaceCount++;
                        } else {
```

```
                              where += _findText.length();
                        }
                  }
            }
            while( where != -1 && where < _textArea.getText().length
() );

            _statusBar.setStatus( replaceCount + " replacements
made." );

      } catch(Exception ex) {
            _statusBar.setStatus("Error: " + ex.getLocalizedMessage() );
      }
}
```

This routine is very similar to the doFindAgain() method, except for a couple of small changes. First we operate in a loop so we can locate all occurrences of the string and secondly, I decided to always start from the beginning of the document. A smarter version would support forward and backward modes and could also support only processing the current highlighted text in the document.

Step 27 – Packaging for Deployment

Now that the SimpleEditor is completed, you need to be able to deliver it to your customers. The best way to do this to put all the code into a JAR (Java Archive) file.

This can be done using Eclipse, or from the command line.

No matter which method you use, you should first create a file named MANIFEST.MF under a folder named META-INF. This file is used to describe your project to the Java VM and is the first file examined when a user launches your application that is stored inside the JAR file.

Create the new META-INF folder under the src/ folder and then create a new text file named MANIFEST.MF. Edit the file to read as follows:

```
Manifest-Version: 1.2.2
Main-Class: edu.uah.coned.SimpleEditor
```

To build the JAR using Eclipse you will have to install the JBoss-IDE plugin for Eclipse. It can build several different kinds of deployment files, primarily intended for installing J2EE project on a JBoss server, but can also build standard JAR files as well.

If you have the plugin, bring up the properties for the project and use the **Packaging Configuration** section to add a new standard JAR to your project. Make sure the JAR will include all the class files from the bin/ folder, the MANIFEST.MF file and the images/ folder with our icon graphics. Once that is completed, right-click on your project and select **Run Packaging** to build the new JAR file.

To build a JAR file from the command line, follow these steps:

1 – Open a command window.
2 – Change to the SimpleEditor/bin directory.
3 – Execute the following command

Linux:

```
jar -c -f ../SimpleEditor.jar -m ../src/META-INF/MANIFEST.MF
com/bamafolks/swing/*.class edu/uah/coned/*.class
```

Windows:

```
jar cfm ..\SimpleEditor.jar ..\src\META-INF\MANIFEST.MF
```

```
com\bamafolks\swing\*.class edu\uah\coned\*.class
```

Now you should be able to run the program by copying the SimpleEditor.jar file to a new computer and executing this command:

```
java -jar SimpleEditor.jar
```

This will work, but there will be one little problem. When we wrote our code, we loaded the images for the toolbar from a hard-coded path. Since those images are not copied to the user's computer, the toolbar will not have any icons!

The solution is to move the icons into a package that is a part of our project and then revise our image code to load the images from there instead of from a hard-coded path.

To do this, create a new package named edu.uah.coned.images in our project. Now, copy all the images you are using into the new package/folder.

Alternately, you can use the **Refactor > Move** option that pops up with you right-click on your images folder.

Next, revise the ActionItem constructor that accepts an icon name to read like this:

```
public ActionItem(String text, int mnemonic, int modifier, int key,
String tooltip, String icon ) {
      super(text);
      URL url = this.getClass().getResource(icon);
      if( url != null ) {
            try {
                  ImageIcon image = new ImageIcon(url);
                  putValue(SMALL_ICON,image);
            } catch( Exception ex ) {
                  System.err.println("Error loading icon from " +
url.toString() + ": " + ex.getLocalizedMessage() );
            }
      } else {
            System.err.println("Unable to find icon named " + icon);
      }
      _mnemonic = mnemonic;
      _key = key;
      _modifier = modifier;
      _tooltip = tooltip;
}
```

You also need to revise the image names to read as follows:

```
“images/New16.gif”
“images/Open16.gif”
... and so on ...
```

Finally, let's rebuild our JAR file following these steps:

1 – Open a command window
2 – Use CD to change to the SimpleEditor/bin folder.
3 – Recreate the JAR file with this command:

```
jar -c -f ../SimpleEditor.jar -m ../src/META-INF/MANIFEST.MF
com/bamafolks/swing/*.class edu/uah/coned/*.class
edu/uah/coned/images/*16.gif
```

Now when you copy this JAR file to a new computer, it should work correctly.

Step 28 – Using Java Web Start

Sun has another very useful method that you can use to easily distribute your Java projects to end-user call the Java Web Start service. This service allows you to upload your JAR file to a web server where users can download and run the application using their web browser. Let's see how this works.

First, you need to create a JNLP (Java Network Launching Protocol) file that describes the contents of your JAR file along with other options so that Java Web Start know how to download and run the application.

Here is the JNLP file I created for my server:

```
<?xml version=”1.0”?>
<!-- JNLP file for the SimpleEditor application -->
<jnlp
  codebase=”http://www.bamafolks.com”
  href=”~randy/students/java/SimpleEditor.jnlp”>
<information>
  <title>SimpleEditor Application</title>
  <vendor>Randy Pearson</vendor>
  <description>SimpleEditor Tutorial</description>
  <description kind=”short”>SimpleEditor Tutorial
Application</description>
  <offline-allowed/>
</information>
<security>
  <all-permissions/>
</security>
<resources>
```

```
  <j2se version="1.5"/>
  <jar href="~randy/students/java/SimpleEditor.jar"/>
</resources>
<application-desc main-class="edu.uah.coned.SimpleEditor"/>
</jnlp>
```

Most of this is self-explanatory, however there are a couple of important things to keep in mind.

First, since our application allows users to edit text files on their local computer, we need to have the <security> section, which allows our program to escape the normal Java security sandbox.

Second, the <resources> section tells the end user they require Java version 1.5 or higher and also names the JAR file. Java Web Start will append this to the end of the codebase setting provided earlier to build a complete URL.

There is still one more thing we must do however. Since we need extra security features in order to work with files on the user's hard drive, we must also mark our JAR file with a cryptographic signature. You cannot get outside the sandbox from a non-signed JAR file.

Normally, a company will purchase a cryptographic signature from a security company such as Verisign or Thawte. These signatures are very similar to the ones used for secure web sites protected via SSL.

For testing and development purposes, we can generate our own cryptographic signature however by following these steps:

Step 1 – Create a new signing key

```
keytool -genkey -keystore myKeystore -alias myself
```

Step 2 – Creating a new self-signed certificate

```
keytool -selfcert -alias myself -keystore myKeystore
```

Step 3 – Signing the JAR file

```
jarsigner -keystore myKeystore SimpleEditor.jar myself
```

Now you JAR file is cryptographic signed and ready to be uploaded onto a web server along with the JNLP file created earlier.

The last step is to create an HTML page with a link to the new JNLP file that was uploaded.

Here is the link I created for my server:

```
<a href="http://www.bamafolks.com/~randy/students/java/SimpleEditor.jnlp
">Launch SimpleEditor</a> (Java Web Start)
```

That completes the SimpleEditor tutorial.

Hope you enjoyed it!