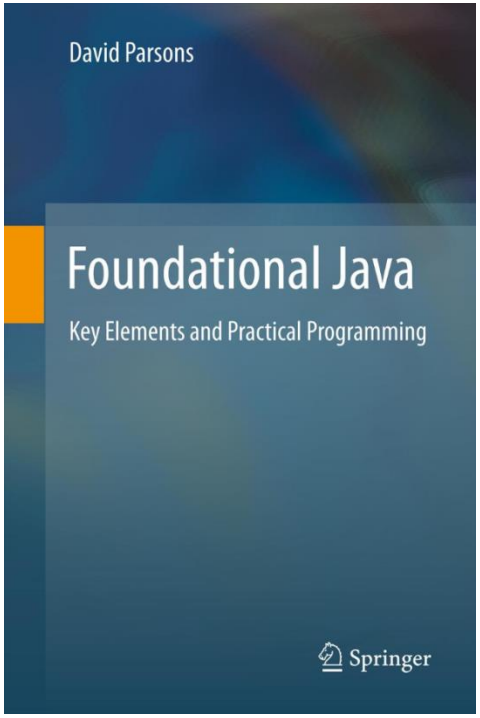


Chapter 19
Dialogs and Menus, Models and Views

Foundational Java

Key Elements and Practical Programming

What is a Dialog?

- In a typical application, you want to give some information to, or get some information from, the user in different contexts
- Dialogs provide pop-up windows that can provide focused modes of interaction for particular features of an application

Modal vs. Modeless Dialogs

- Dialogs can be modal
  - Modal dialogs prevent the user from doing anything in the dialog's parent application until the dialog has been dismissed
- Dialogs can be modeless
  - The user can work in other windows of the application without dismissing the dialog
- Defined in a parameter to the constructor

Parent Frames

- A dialog is dependent on a parent window
  - When the frame is minimized, maximized, or destroyed, its dependent dialogs follow
- The parent/owner of a dialog must be an instance of JFrame

Predefined Dialogs in Swing

- Swing provides some predefined types of dialog that have generic uses across many applications
  - Message boxes
  - Dialogs for navigating the file system
  - Dialogs for choosing colors
  - etc…
- This section provides a brief introduction to these dialogs

Message Box Dialogs using JOptionPane

- The simplest Swing dialogs are created using the JOptionPane class
- Supports the creation of modal dialogs that handle simple interactions with the user
- Three types of dialog can be created using a JOptionPane

Dialog Type	Description
Message Dialog	Tell the user about something that has happened
Confirm Dialog	Asks a confirming question, like yes/no/cancel
Input Dialog	Prompt for some input from the user

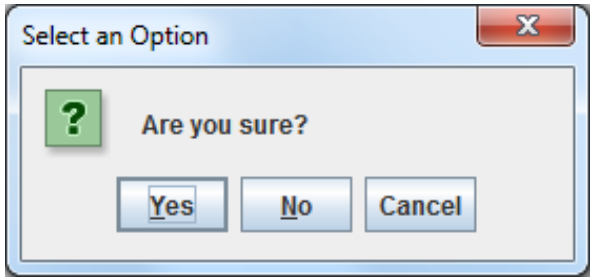
Dialog Configuration

- Instances of the dialog types can be created using various overloaded static methods of the JOptionPane class
  - e.g. to show a simple message dialog with an ‘OK’ button, you can use one of the ‘JOptionPane.showMessageDialog’ methods
- The following code shows examples of the three basic types of dialogs being created with their various ‘show…’ methods

```
JOptionPane.showMessageDialog(null, "Hello!");
JOptionPane.showConfirmDialog(null, "Are you sure?");
JOptionPane.showInputDialog("Please enter something");
```

JOptionPane Dialogs

- Message, confirmation and input dialogs
  - Confirmation and input dialogs both return values to the code that calls them
  - Messages, titles, icons, buttons etc. can all be configured in a variety of combinations



File Chooser Dialogs using JFileChooser

- The JFileChooser dialog can be used for loading and saving files
  - Configured using the 'showOpenDialog' or 'showSaveDialog' methods
  - The 'showDialog' method that can be used to create a dialog that has a customised message on the dialog's 'approve' button
- Create the JFileChooser using a simple constructor, then pass the parent container as a parameter to the chosen 'show…' method
- Once the dialog has been closed, use the other methods to retrieve the resulting file information

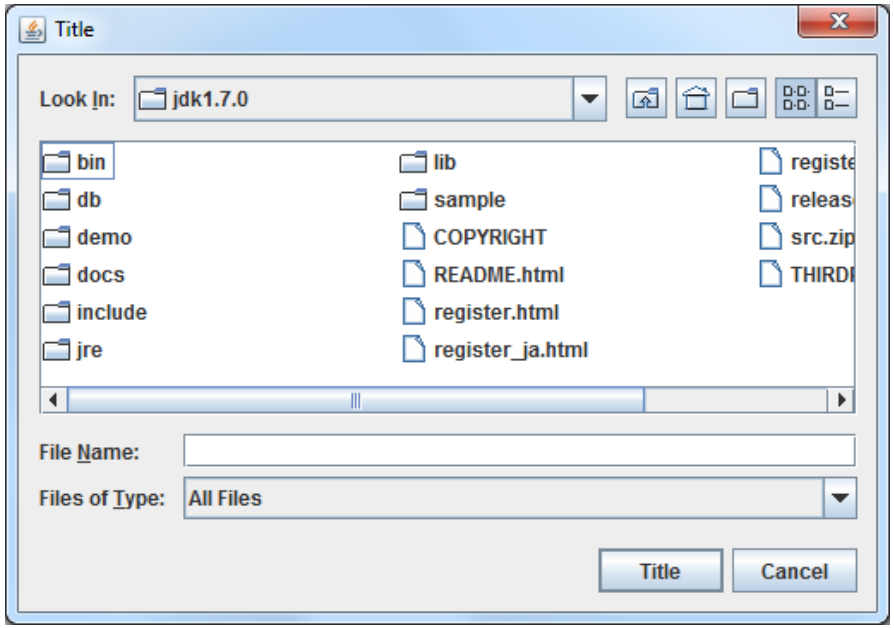
Creating a JFileChooser

- The following code fragment shows a 'file open' dialog being created
- If the value returned from the 'show…' method approves the action, then we retrieve the name of the chosen file from the dialog

```
JFileChooser chooser = new JFileChooser();
int returnVal = chooser.showOpenDialog(container);
if(returnVal == JFileChooser.APPROVE_OPTION)
{
  String filename = chooser.getSelectedFile().getName());
}
```

The JFileChooser Dialog

- A file dialog will appear similar to this
  - Other configurations of the dialog look much the same but will have different titles and button labels, such as ‘Save’



Exercise 19.1

- Write a subclass of JFrame that will display an ‘open file’ dialog when a button is pressed
- When the file dialog is closed, pop up a message box that says ‘file opened’

Color Chooser Dialogs using JColorChooser

- Another of the predefined Swing dialogs is the JColorChooser
  - Enables users to select colors
- We can interact with a JColorChooser to change the foreground or background colors of other components

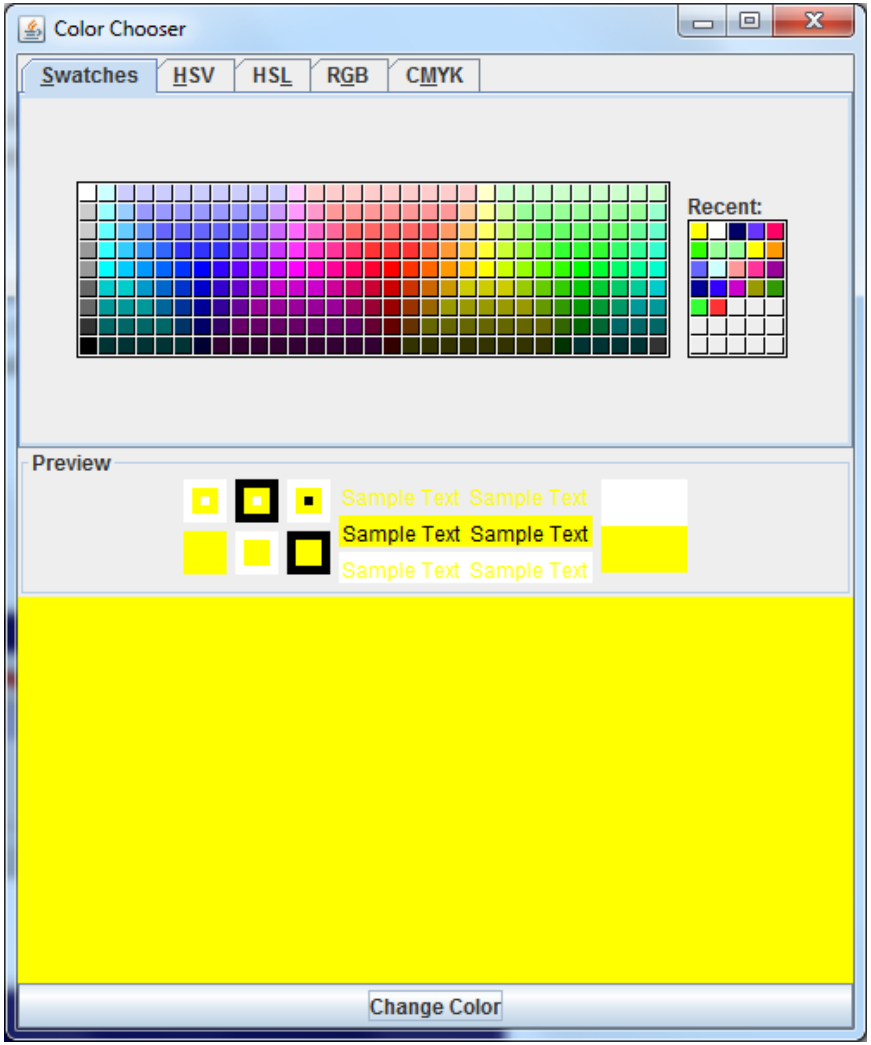
JColorChooser Example

- When the button is pressed, the panel will be repainted in the color currently selected by the color chooser

```
colorButton.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent e)
  {
    colorPanel.setBackground(colorChooser.getColor());
  }
});
```

JColorChooser Dialog

- The color chooser with one of its tabbed panes selected
- Each one has a different way of selecting the color

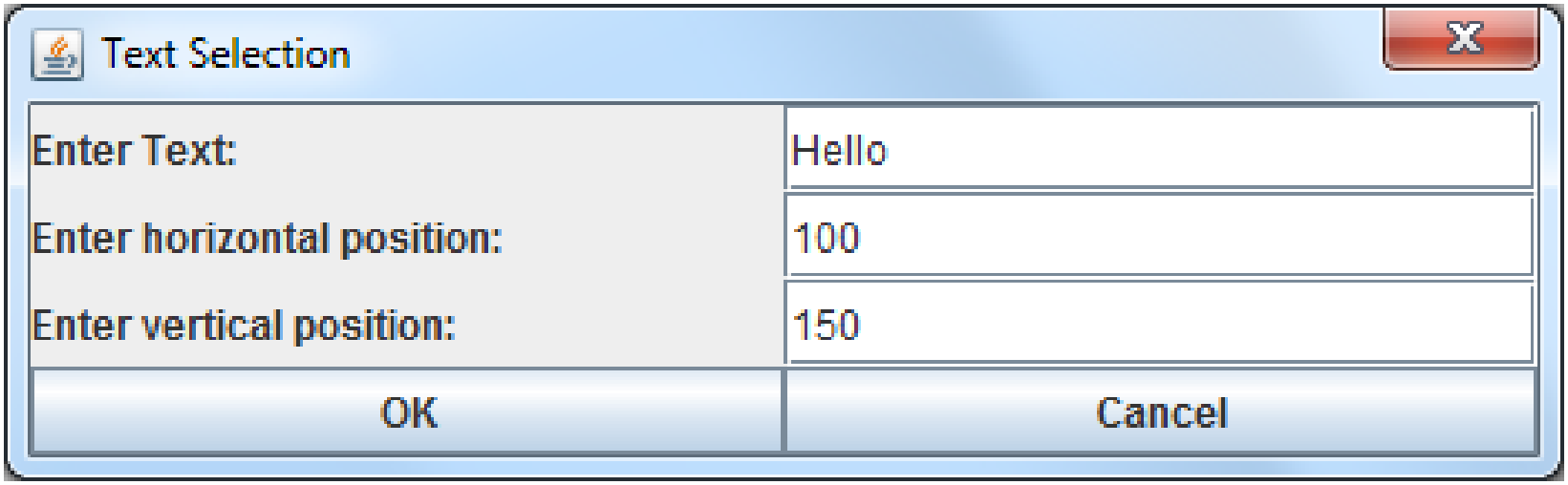


Custom Dialogs with JDialog

- Applications will need all kinds of dialogs customized to the requirements of those applications
- These customized dialogs can be created by using specialized subclasses of the JDialog class
- These dialogs can have complex sets of components and event listeners, just like frames

A Custom Dialog

- The example dialog below is designed to allow the user to enter three items of data that can be used with the ‘drawString’ method of the Graphics class



StringData Value Object

- To make it easy to pass these values from a dialog to the component that actually draws the String, a value object will be useful
- A value object represents a field that is more complex than a simple data type, encapsulating more than one value

```
public class StringData
{
  private String text;
  private int x;
  private int y;
//etc..
```

The DrawStringDialog Class

- The class used to create this dialog extends JDialog.

```
public class DrawStringDialog extends JDialog
```

- The dialog's constructor has the host JFrame passed to it
- Passes this to the superclass constructor with a Boolean that sets the modality for the dialog

```
public DrawStringDialog(JFrame owner)
{
// set the owning frame and make the dialog modal
  super(owner, true);
```

The Dialog Event Listener

- The main part of the dialog's functionality is contained in the event listener for the button
- If the "OK" button is pressed then the data from the three text fields is put into a StringData object

```
if ("OK".equals(event.getActionCommand()))
{
  stringData.setText(textField.getText());
  try
  {
    stringData.setX(Integer.parseInt(xField.getText()));
    stringData.setY(Integer.parseInt(yField.getText()));
  }
  catch(NumberFormatException e)
// etc…
```

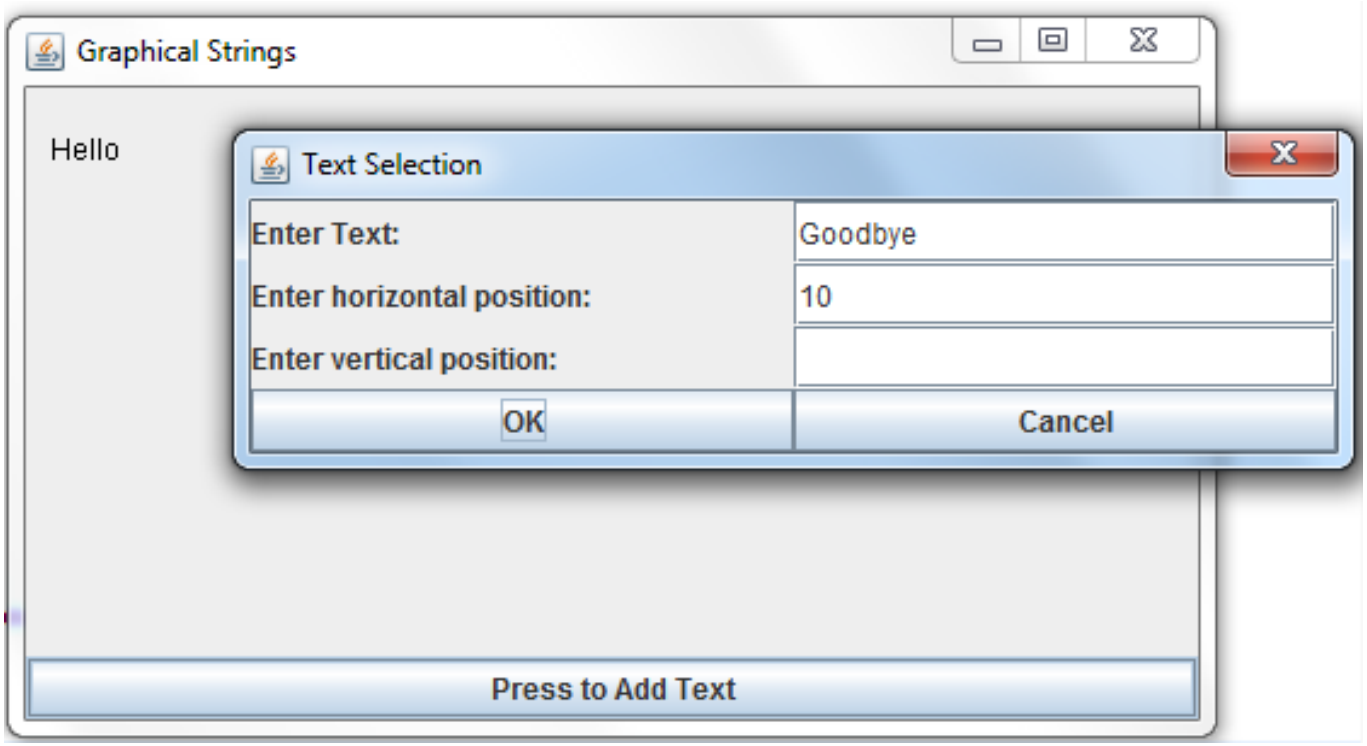
Showing the Dialog

- A frame event handler calls the 'showData' method of the DrawStringDialog, using a reference to the parent JFrame

```
public void actionPerformed(ActionEvent event)
{
  stringDialog = new DrawStringDialog(parent);
  StringData stringData = stringDialog.showDialog();
//…etc.
```

Invoking the Dialog

- The custom dialog being used to add graphical strings to a panel



Exercise 19.2

- Add a ‘size’ field to the StringData class
- Modify the ‘TextSelection’ dialog to allow this value to be entered
- Use this value to set the font size used when the string is drawn

Using Menus

- Application frames frequently provide menus so that the user can select different courses of action while using the application
- To add a menu to a JFrame, we use three classes; JMenu, JMenuItem, and JMenuBar
- One of the JMenu constructors allows you to set the menu's text label, which will appear in the menu bar, for example

```
JMenu fileMenu = new JMenu("File Menu");
```

Creating JMenuItems

- A JMenuItem is one selection from a menu, and is created using a constructor that takes the text label of the item as its parameter.
  - e.g. items on a ‘file’ menu

```
JMenuItem newItem = new JMenuItem("New");
JMenuItem openItem = new JMenuItem("Open");
JMenuItem saveItem = new JMenuItem("Save");
JMenuItem exitItem = new JMenuItem("Exit");
```

Adding Items to a Menu

- To add a JMenuItem to a particular menu, we use the ‘add’ method of the JMenu object
- This will put the four menu items into the file menu in the order in which they have been added

```
fileMenu.add(newItem);
fileMenu.add(openItem);
fileMenu.add(saveItem);
fileMenu.add(exitItem);
```

The Menu Bar

- A menu is placed in a menu bar that is added to a frame
- We can create a JMenuBar with a simple constructor

```
JMenuBar mainMenuBar = new JMenuBar();
```

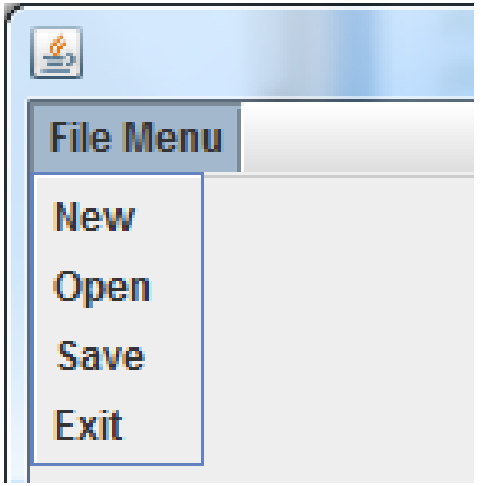
- The menu can then be added to the menu bar

```
mainMenuBar.add(fileMenu);
```

- The menu bar can be added to the JFrame

```
setJMenuBar(mainMenuBar);
```

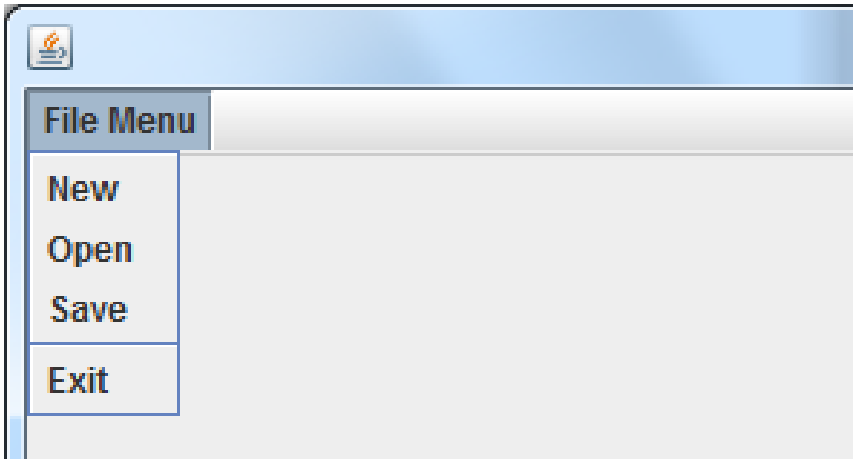
- The menu items will now automatically appear when the menu is selected



Menu Separators

- A menu separator is simply a horizontal line that appears between groups of items in a menu, to put them into categories and make them easier to navigate
- The 'addSeparator' method of the JMenu class allows the separator to be added between menu items
  - In this example a separator is added between the 'Save' and 'Exit' items

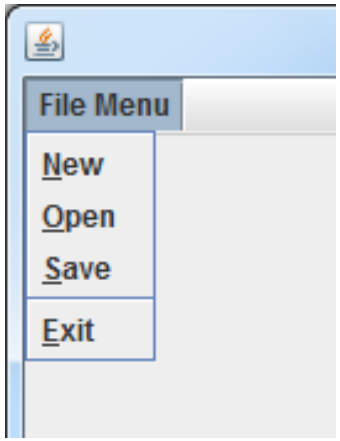
```
fileMenu.add(newItem);
fileMenu.add(openItem);
fileMenu.add(saveItem);
fileMenu.addSeparator();
fileMenu.add(exitItem);
```



Mnemonics

- Mnemonics allow a user to navigate menus using the keyboard
  - One letter in the menu or menu item text is underlined

```
newItem.setMnemonic('N');
openItem.setMnemonic('O');
saveItem.setMnemonic('S');
exitItem.setMnemonic('E');
```

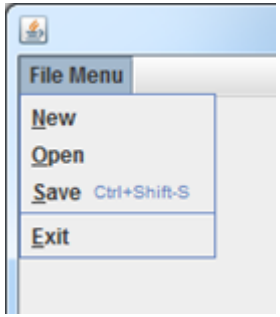


- Menus can be opened (in Windows) by typing the ALT key with the mnemonic letter
- Once a menu is opened, a menu item can be accessed by typing the mnemonic letter

Keyboard Accelerators

- Accelerators allow you to create keyboard shortcuts for accessing menu items directly
  - The accelerator is shown beside the menu item in the menu
- Menu items have an accelerator property of type Keystroke
  - Keystroke encapsulates the key that was pressed along with any modifiers (SHIFT, CTRL, ...)

```
int modifiers = ActionEvent.SHIFT_MASK | ActionEvent.CTRL_MASK;
KeyStroke ks = KeyStroke.getKeyStroke(KeyEvent.VK_S, modifiers);
saveItem.setAccelerator(ks);
```



MenuListeners

- We can add ActionListeners to menu items
- In this example we implement the listener as an inner class ('FileMenuListener') with a constructor that takes a reference to the parent frame

```
class FileMenuListener implements ActionListener
{
  private JFrame frame;
  public FileMenuListener(JFrame parent)
  {
    frame = parent;
  }
//etc.
```

Responding to Menu Events

- The MenuListener responds to different menu choices by getting the text label of the menu item from the 'getActionCommand' method of the event object

```
String command = e.getActionCommand();
```

- The String returned from this method can be used to choose the appropriate action, e.g.

```
if(command.equals("New"))
{
    int returnVal =  chooser.showDialog(frame, "Create New File");
    if(returnVal == JFileChooser.APPROVE_OPTION)
    {    // create new file…
```

File Menu ActionListeners

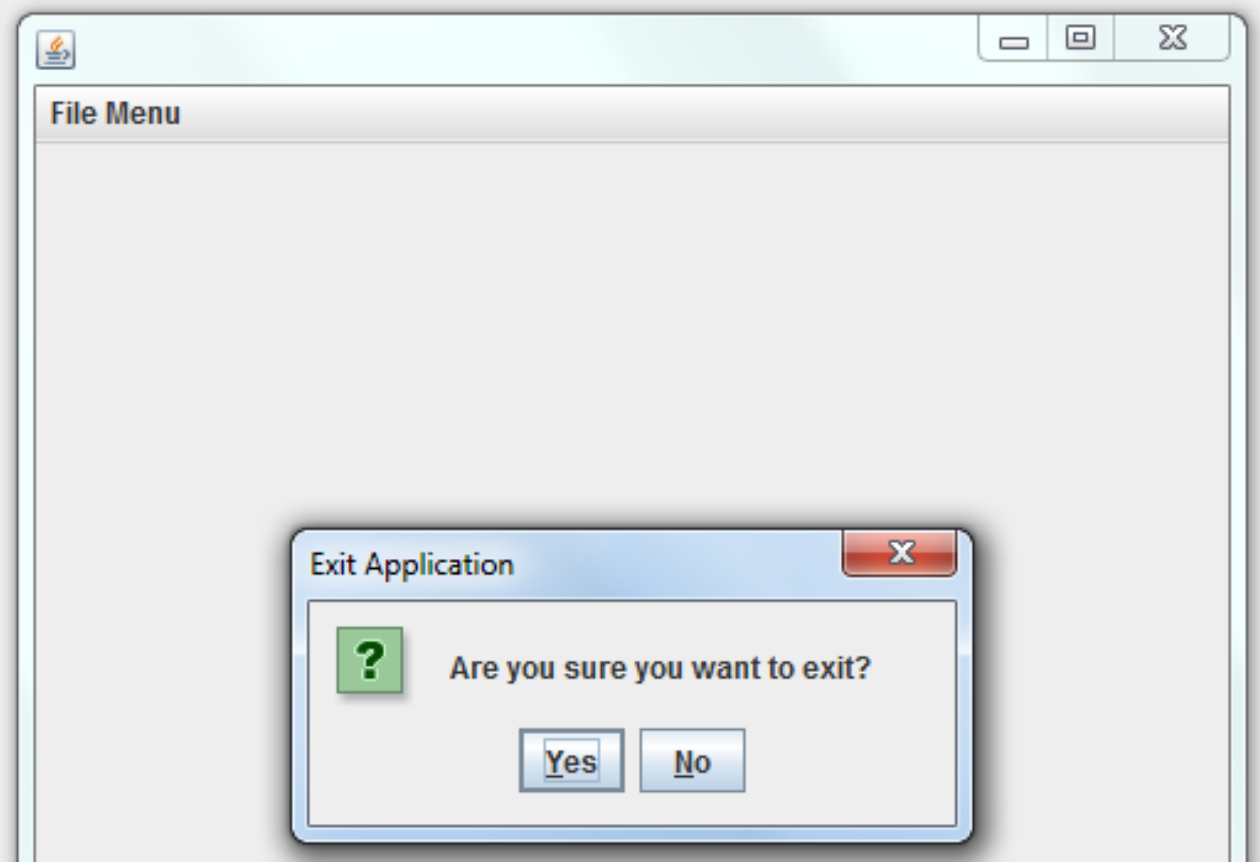
- e.g. Adding an action listener to the 'New' menu item:

```
newItem.addActionListener(new FileMenuListener(this));
```

- For the 'Exit' menu item, a confirmation dialog is displayed
- If the value returned from the dialog is 'YES_OPTION', then the frame is closed.
  - 'Yes' and 'No' buttons are set using the 'YES_NO' field from the JOptionPane as the fourth parameter to the 'showConfirmDialog' method
  - The third parameter is the title of the dialog

```
int dialogOption = JOptionPane.showConfirmDialog (getContentPane(),
  "Are you sure you want to exit?",  "Exit Application", JOptionPane.YES_NO_OPTION);
if (dialogOption == JOptionPane.YES_OPTION)
{
 frame.dispose();
}
```

The Exit Dialog



Exercise 19.3

- Create an application with a frame that hosts a 'Color' menu
- Add a few basic color choices to the menu
- Include a field in the frame class that can store a reference to the Graphics object that it uses, with getter and setter methods
  - Set the value of this field after the frame has been constructed
- Add a suitable event handler that will change the foreground color in response to the menu selection
- Use the color when drawing on the panel using the mouse, as we did in Chapter 18. You will need to access the Graphics reference from the frame
- Add another option to the menu that will invoke a JColorChooser, and use the value returned from that to set the drawing color
  - Invoke the JColorChooser as a modal dialog using the static 'showDialog' method, e.g.

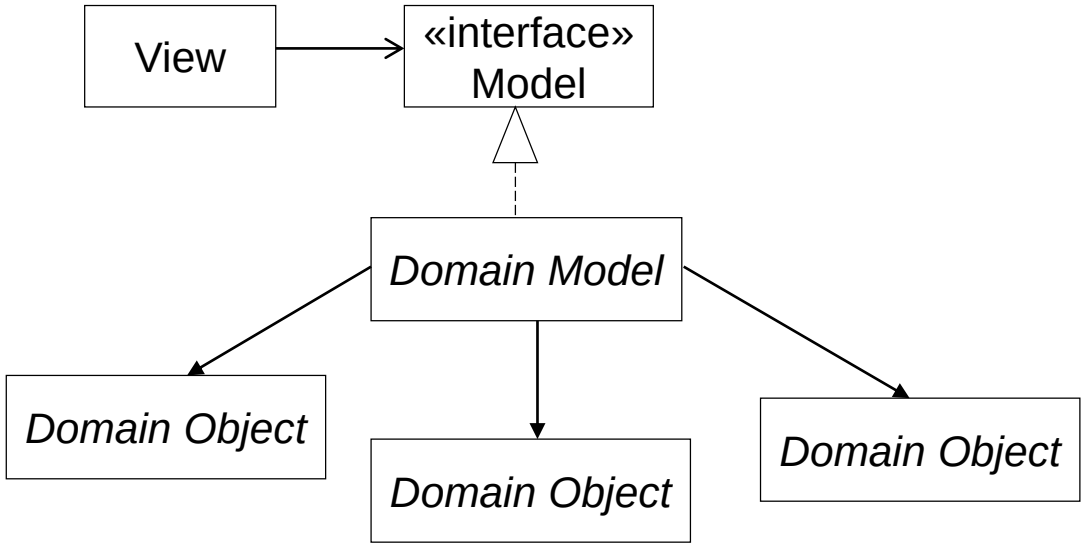
```
Color col = JColorChooser.showDialog(frame, "choose Color". Color.WHITE);
```

Model-View-Controller (MVC)

- Swing uses MVC to factor implementation responsibilities
- Model
  - Maintains domain-specific state information
- View
  - Displays what the model layer represents
- Controller
  - Controls interaction between the user, the display and the model
- In Swing, the view and controller are typically implemented as part of the same class

MVC Objects

- In order for a domain specific model to be plugged into standard view, the model must implement an interface that the view can work with
- The implementing class will then interact with the objects of the underlying domain
- Many Swing components need a separately coded model class



Model and View in JTextPane Components

- The JTextPane component is a text editor and/or formatter
- The ‘document’ property is the model
- The document defines the text and also the text formatting
  - Supports multiple text fonts, sizes and styles within the same component
  - Supports multiple paragraph formats
  - Provides word-wrap
- The JTextPane class provides the view and controller
  - Model provided by implementers of the Document interface
  - Changes to the model are immediately reflected in the view

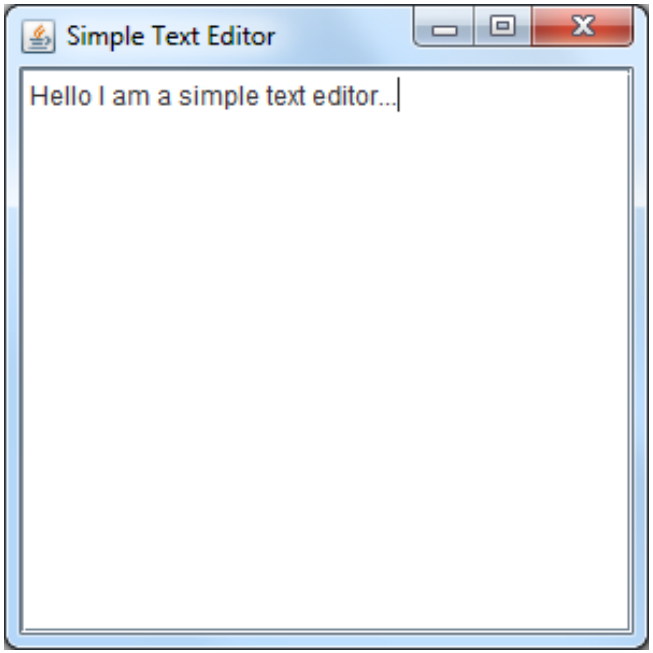
Adding a Document

- Swing provides an AbstractDocument class that implements the Document interface
- Concrete document classes allow different types of document to be created (RTF, HTML)
  - A document is added to a JTextPane
  - The text pane needs to be added to a JScrollPane, then added to the frame:

```
JTextPane textPane = new JTextPane();
Document document = new DefaultStyledDocument();
textPane.setDocument(document);
JScrollPane scroller = new JScrollPane(textPane);
getContentPane().add(scroller);
```

JTextPane

- A JTextPane in use, with some text entered
- Text typed into the view automatically updates the underlying document model



Text Styling with AttributeSets

- The SimpleAttributeSet can be used to format text

```
SimpleAttributeSet textStyle = new SimpleAttributeSet();
```

- Each attribute set defines a set of formatting information such as font size, bold, underline, text color, etc.
- Individual attributes are added to an attribute set using static ‘set’ methods from the StyleConstants class
- This attribute set is given a bold style
  - The second Boolean parameter specifies if we are adding or removing the style

```
StyleConstants.setBold(textStyle, true);
```

Applying AttributeSets

- A SimpleAttributeSet can be applied to selected parts of the text in the styled document
- The currently selected text can be identified using the ‘getSelectionStart’ and ‘getSelectionEnd’ methods.

```
int start = textPane.getSelectionStart();
int end = textPane.getSelectionEnd();
```

- The text style can be applied to the document using the ‘setCharacterAttributes’ method
- Specifies the character range to format, the style, and whether the new style replaces (or augments) the current style

```
document.setCharacterAttributes(start, end - start, textStyle, false);
```

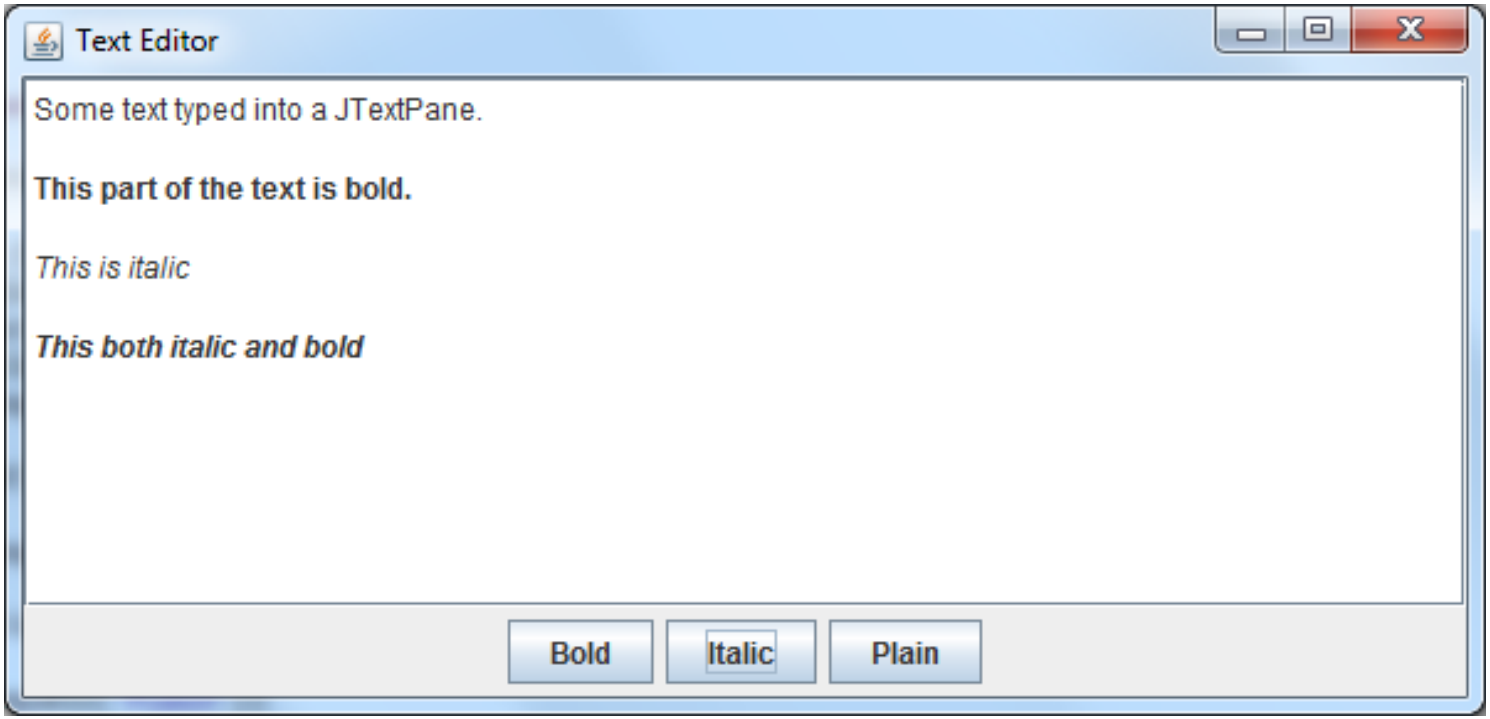
View Reflects Model

- When the model is changed, the view automatically (and immediately) updates

```
public void actionPerformed(ActionEvent e) {
      StyledDocument document = (StyledDocument) textPane.getDocument();
      int start = textPane.getSelectionStart();
      int end = textPane.getSelectionEnd();
      SimpleAttributeSet textStyle =  new SimpleAttributeSet();
      if (e.getActionCommand().equals("Bold"))
      {
         StyleConstants.setBold(textStyle, true);
      }
      if (e.getActionCommand().equals("Italic"))
      {
         StyleConstants.setItalic(textStyle, true);
       }
      document.setCharacterAttributes(start, end - start, textStyle, false);
    }
```

AttributeSets Applied

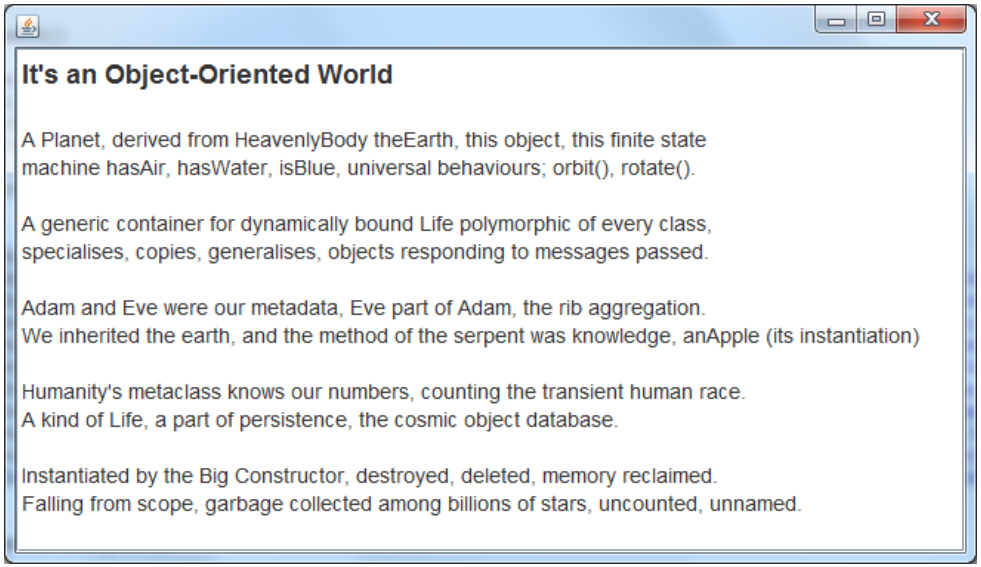
- Here is the example TextPane with some formatting applied to parts of the text



Presenting Read-Only text

- If required, editing can be disabled so the TextPane can be used simply to present text rather than edit it

```
document.insertString(document.getLength(), "text here", body);
```



Exercise 19.4

- The previous example showed a read only document being displayed in a text pane. Modify this code so that it can read the text data from a file
- Use the CENTER and SOUTH areas of a BorderLayout to build a text file reading window
- Add a button to a panel in the SOUTH area to invoke a JFileChooser to select a file
- Add a JTextPane to the CENTER to display the contents of the selected file
- In terms of reading a text file, remember that an InputStreamReader class can be used to open a text file for input. This object can then be passed to the constructor of a BufferedReader object which has a “readLine” method for reading in a line of text from a file

Model and View in the JTable Component

- The JTable component displays a graphical table based on a table model
- The JTable needs to be populated with data
- This comes from a separate table model object
- Implements the TableModel interface

Implementing the TableModel Interface

- Easiest way is to subclass AbstractTableModel
  - Provides default implementations for most of the methods in the TableModel interface
  - Including event handling
- You do need to implement these methods

```
public int getRowCount()
```

  - How many rows?

```
public int getColumnCount()
```

  - How many colums?

```
public Object getValueAt(int row, int colums)
```

  - What's in this cell?

Creating a Table Model

- This example is based of a table of Modules belonging to a Course
- The CourseTableModel class extends AbstractTableModel
  - Has a Course field from which it will obtain its list of Modules

```
public class CourseTableModel extends AbstractTableModel
{ …
```

- 'getColumnCount' is simple to override
  - 3 columns
  - module name, credit points and assessment data

```
public int getColumnCount()
{
  return 3;
}
```

Table Model Methods

- 'getRowCount' is implemented using the 'size' method of Collection
-
```
public int getRowCount()
{
  return course.getModules().size();
}
```
- Overriding 'getColumnName' is not essential, but replaces the default spreadsheet style titles

```
public String getColumnName(int column)
{
  switch (column) {
    case 0: return "Module Name";
    case 1: return "Credit Points";
    case 2: return "Assessment";
  }
  return null;
}
```

Implementing 'getValueAt'

- Need to deal with row and column positions, passed into the method as parameters
- Return the data for the table cell at that position
- First, get the Module from the collection based on the row number

```
Module m = course.getModules().get(row);
```

- Then select the appropriate data from the object that matches the chosen column

```
switch (col)
{
  case 0: return m.getName();
  case 1: return m.getCreditPoints();
  case 2: return m.getAssessment();
}
return null;
```

Editing Table Data

- Tables do not have to ‘read only’
- We can set any (or all) of the cells, rows or columns to allow them to be edited
  - return ‘true’ from the ‘isCellEditable’ method
- In this example, the ‘Assessment’ column is made editable

```
public boolean isCellEditable(int row, int column)
{
  if (column == findColumn("Assessment"))
  {
    return true;
  }
```

- Allowing a table to be editable is the easy part
- In a real application you would have to ensure that changes to the view are reflected in the underlying model

Creating a JTable View

- Create an instance of the table model

```
courseTable = new CourseTableModel();
```

- Pass the table model to the JTable constructor

```
JTable table = new JTable(courseTable);
```

- Add the JTable to a JScrollPane

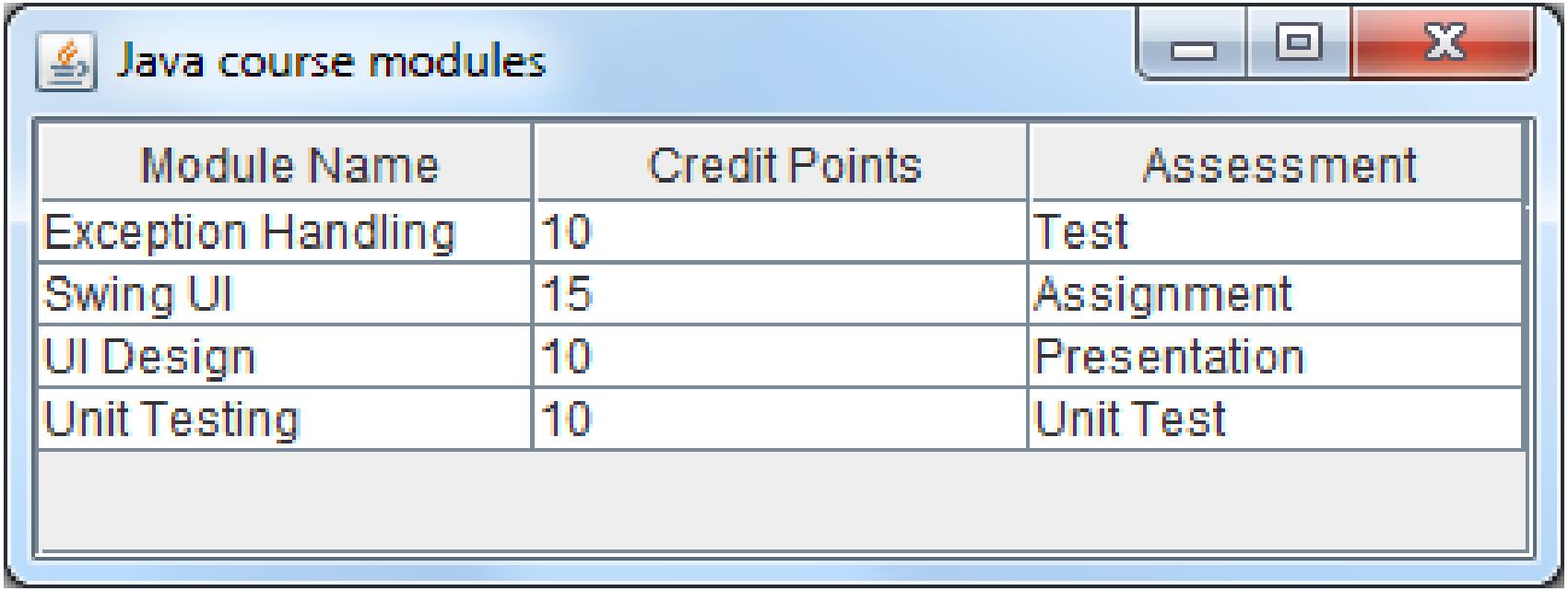
```
JScrollPane scrollpane = new JScrollPane(table);
```

- Add the scroll pane to the main window

```
Container container = getContentPane();
container.add(scrollpane);
```

JTable View

- The table displaying the data from the course modules



Exercise 19.6

- Put the module table into a frame that contains other components that can display the details of the current course to which the modules belong

Exercise 19.7

- Use a JTable to display the data from a suitable table model relating to bank account transactions
- Create some objects of the Transaction class from Exercise 13.2 and use these in your table model

Summary

- Built-in and custom dialogs to provide richer options for interacting with the user
- Adding menus to a Swing application
- Model View Controller architecture
- Model view separation the JTextPane and the JTable