BASIC SWING COMPONENTS

Unit Structure:

2.0 OBJECTIVES

The objective of this chapter is to lear how to use the Swing Components to create a user interface. We will start the chapter with a few components and then lear how to incorporate the Java Print API.

2.1 THE JCOMPONENT CLASS

With the exception of top-level containers, all Swing components whose names begin with "J" descend from the JComponent class. For example, JLabel, JButton, JTree, and JTable all inherit from JComponent. However, JFrame doesn't because it implements a top-level container.

The JComponent class extends the Container class, which itself extends Component. The Component class includes everything from providing layout hints to supporting painting and events. The Container class has support for adding components to the container and laying them out.

JComponent Features

The JComponent class provides the following functionality to its descendants:

Tool tips - By specifying a string with the setToolTipText method, you can provide help to users of a component. When the cursor pauses over the component, the specified string is displayed in a small window that appears near the component.

Borders - The setBorder method allows you to specify the border that a component displays around its edges.

Keyboard-generated actions - Using the registerKeyboardAction method, you can enable the user to use the keyboard, instead of the mouse, to operate the GUI. The combination of character and modifier keys that the user must press to start an action is represented by a KeyStroke object. The resulting action event must be handled by an action listener. Each keyboard action works under exactly one of three conditions: only when the actual component has the focus, only when the component or one of its containers has the focus, or any time that anything in the component's window has the focus.

Application-wide pluggable look and feel - Behind the scenes, each JComponent object has a corresponding ComponentUI object that performs all the drawing, event handling, size determination, and so on for that JComponent. Exactly which ComponentUI object is used depends on the current look and feel, which you can set using the UIManager.setLookAndFeel method.

Support for layout - To give you a way to set layout hints, the JComponent class adds setter methods -- setPreferredSize, setMinimumSize, setMaximumSize, setAlignmentX, and setAlignmentY.

Double buffering - Double buffering smooths on-screen painting.

Methods to increase efficiency - JComponent has a few methods that provide more efficient ways to get information than the JDK 1.1 API allowed. The methods include getX and getY, which you can use instead of getLocation; and getWidth and getHeight, which you can use instead of getSize. It also adds one-argument forms of getBounds, getLocation, and getSize for which you specify the object to be modified and returned, letting you avoid unnecessary object creation. These methods have been added to Component for Java 2 (JDK 1.2).

2.1.2 JLabel

A label object is a single line of read only text. A common use of JLabel objects is to position descriptive text above or besides other components. JLabel extends the JComponent class. It can display text and/or icon.

Method or Constructor	Purpose

JLabel(Icon)
JLabel(Icon, int)
JLabel(String)
JLabel(String, Icon, int)
JLabel(String, int)
JLabel()
	Creates a JLabel instance, initializing it to have the specified text/image/alignment. The int argument specifies the horizontal alignment of the label's contents within its drawing area. The horizontal alignment must be one of the following constants defined in the SwingConstants interface (which JLabel implements): LEFT, CENTER, RIGHT, LEADING, or TRAILING. For ease of localization, we strongly recommend using LEADING and TRAILING, rather than LEFT and RIGHT.

void setText(String) String getText()	Sets or gets the text displayed by the label.
void setIcon(Icon) Icon getIcon()	Sets or gets the image displayed by the label.
void setDisplayedMnemonicIndex(int) int getDisplayedMnemonicIndex()	Sets or gets a hint as to which character in the text should be decorated to represent the mnemonic. This is useful when you have two instances of the same character and wish to decorate the second instance. For example, setDisplayedMnemonicIndex(5) decorates the character that is at position 5 (that is, the 6th character in the text). Not all types of look and feel may support this feature.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Sets or gets the image displayed by the label when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image.

2.1.3 JTextField

A text field is a basic text control that enables the user to type a small amount of text. When the user indicates that text entry is complete (usually by pressing Enter), the text field fires an action event. If you need to obtain more than one line of input from the user, use a text area. The horizontal alignment of JTextField can be set to be left justified, leading justified, centered, right justified or trailing justified. Right/trailing justification is useful if the required size of the field text is smaller than the size allocated to it. This is determined by the setHorizontalAlignment and getHorizontalAlignment methods. The default is to be leading justified.

Method or Constructor	Purpose

JTextField()
JTextField(String)
JTextField(String, int)
JTextField(int)
	Creates a text field. When present, the int argument specifies the desired width in columns. The String argument contains the field's initial text.

void setText(String) String getText()	Sets or obtains the text displayed by the text field.
void setEditable(boolean) boolean isEditable()	Sets or indicates whether the user can edit the text in the text field.
void setColumns(int) int getColumns()	Sets or obtains the number of columns displayed by the text field. This is really just a hint for computing the field's preferred width.
void setHorizontalAlignment(int) int getHorizontalAlignment()	Sets or obtains how the text is aligned horizontally within its area. You can use JTextField.LEADING, JTextField.CENTER, and JTextField.TRAILING for arguments.

2.1.4 JButton

A JButton class provides the functionality of a push button. JButton allows an icon, a string or both to be associated with the push button. JButton is a subclass of AbstractButton which extends JComponent.

Method or Constructor	Purpose
JButton(Action)	Create a JButton instance, initializing it to

JButton(String, Icon)
JButton(String)
JButton(Icon)
JButton()
	have the specified text/image/action.

void setAction(Action) Action getAction()	Set or get the button's properties according to values from the Action instance.
void setText(String) String getText()	Set or get the text displayed by the button.
void setIcon(Icon) Icon getIcon()	Set or get the image displayed by the button when the button isn't selected or pressed.
void setDisabledIcon(Icon) Icon getDisabledIcon()	Set or get the image displayed by the button when it is disabled. If you do not specify a disabled image, then the look and feel creates one by manipulating the default image.
void setPressedIcon(Icon) Icon getPressedIcon()	Set or get the image displayed by the button when it is being pressed.

Example: Write a program to create a user interface for students biodata. (Demo example for JLabel, JTextfield and JButton).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData01 extends JFrame
{
//Step 3 – Create all the instances required
        JTextField txtName,txtMobNo;
        JLabel lblName, lblMobNo;
        JButton cmdOk,cmdCancel;
        public StudentBioData01(){
                //Step 4- Create objects for the declared instances
                txtName=new JTextField(20);
                txtMobNo=new JTextField(20);
                lblName=new JLabel("Student Name");
                lblMobNo=new JLabel("Mobile No.");
                cmdOk =new JButton("Ok");
                cmdCancel=new JButton("Cancel");
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblName); con.add(txtName);
                con.add(lblMobNo); con.add(txtMobNo);
                con.add(cmdOk); con.add(cmdCancel);
        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData01 sbd=new StudentBioData01();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.5 JCheckBox

A JCheckBox class provides the functionality of a Check box. Its immediate super class is JToggleButton which provides support for 2 state buttons.

Constructor	Purpose

JCheckBox(String)
JCheckBox(String, boolean)
JCheckBox(Icon)
JCheckBox(Icon, boolean)
JCheckBox(String, Icon)
JCheckBox(String, Icon, boolean)
JCheckBox()
	Create a JCheckBox instance. The string argument specifies the text, if any, that the check box should display. Similarly, the Icon argument specifies the image that should be used instead of the look and feel's default check box image. Specifying the boolean argument as true initializes the check box to be selected. If the boolean argument is absent or false, then the check box is initially unselected.

String getActionCommand()	Returns the action command for this button.

String getText()	Returns the button's text.
boolean isSelected()	Returns the state of the button.
void setEnabled(boolean b)	Enables (or disables) the button.
void setSelected(boolean b)	Sets the state of the button.
void setText(String text)	Sets the button's text.

2.1.6 JRadioButton

A JRadioButton class provides the functionality of a radio button. Its immediate super class is JToggleButton which provides support for 2 state buttons.

Constructor	Purpose

JRadioButton(String)
JRadioButton(String, boolean)
JRadioButton(Icon)
JRadioButton(Icon, boolean)
JRadioButton(String, Icon)
JRadioButton(String, Icon, boolean)
JRadioButton()
	Create a JRadioButton instance. The string argument specifies the text, if any, that the radio button should display. Similarly, the Icon argument specifies the image that should be used instead of the look and feel's default radio button image. Specifying the boolean argument as true initializes the radio button to be selected, subject to the approval of the ButtonGroup object. If the boolean argument is absent or false, then the radio button is initially unselected.

Methods same as JCheckBox	

Example: Write a program to create a user interface for students biodata. (Demo example for JCheckbox, JRadioButton).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData02 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lbllang,lblstream;
        JCheckBox cbeng,cbhin,cbmar;
        JRadioButton rbart,rbcomm,rbsci;
        ButtonGroup bg;
        public StudentBioData02()
        {
                //Step 4- Create objects for the declared instances
                lbllang=new JLabel("Languages Known");
                lblstream=new JLabel("Stream");
                cbeng=new JCheckBox("English",true);
                cbhin=new JCheckBox("Hindi");
                cbmar=new JCheckBox("Marathi");
                rbart=new JRadioButton("Arts");
                rbcomm=new JRadioButton("Commerce");
                rbsci=new JRadioButton("Science");
                bg=new ButtonGroup();
                bg.add(rbart); bg.add(rbcomm); bg.add(rbsci);
//Step 5 – Add all the objects in the Content Pane with Layout
                Container  con=getContentPane();
                con.setLayout(new  FlowLayout());
                con.add(lbllang);
                con.add(cbeng); con.add(cbhin); con.add(cbmar);
                con.add(lblstream);
                                        con.add(rbart);                                        con.add(rbcomm);con.add(rbsci);

        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData02 sbd=new StudentBioData02();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.7 JComboBox

A component that combines a button or editable field and a drop-down list. The user can select a value from the drop-down list, which appears at the user's request. If you make the combo box editable, then the combo box includes an editable field into which the user can type a value.

Constructors & Method	**Purpose**

JComboBox()
JComboBox(Object[])
JComboBox(Vector)
	Create a combo box with the specified items in its menu. A combo box created with the default constructor has no items in the menu initially. Each of the other constructors initializes the menu from its argument: a model object, an array of objects, or a Vector of objects.

void addItem(Object) void insertItemAt(Object, int)	Add or insert the specified object into the combo box's menu. The insert method places the specified object at the specified index, thus inserting it before the object currently at that index. These methods require that the combo box's data model be an instance of MutableComboBoxModel.
Object getItemAt(int) Object getSelectedItem()	Get an item from the combo box's menu.
void removeAllItems() void removeItemAt(int) void removeItem(Object)	Remove one or more items from the combo box's menu. These methods require that the combo box's data model be an instance of MutableComboBoxModel.
int getItemCount()	Get the number of items in the combo box's menu.
void addActionListener(ActionListener)	Add an action listener to the combo box. The listener's actionPerformed method is called when the user selects an item from the combo box's menu or, in an editable combo box,

	when the user presses Enter.
void addItemListener(ItemListener)	Add an item listener to the combo box. The listener's itemStateChanged method is called when the selection state of any of the combo box's items change.

2.1.8 JList

A component that allows the user to select one or more objects from a list. A separate model, ListModel, represents the contents of the list. It's easy to display an array or vector of objects, using a JList constructor that builds a ListModel instance for you.

Method or Constructor	**Purpose**

JList(Object[])
JList(Vector)
JList()
	Create a list with the initial list items specified. The second and third constructors implicitly create an immutable ListModel; you should not subsequently modify the passed-in array or Vector.

void setListData(Object[]) void setListData(Vector)	Set the items in the list. These methods implicitly create an immutable ListModel.
void setVisibleRowCount(int) int getVisibleRowCount()	Set or get the visibleRowCount property. For a VERTICAL layout orientation, this sets or gets the preferred number of rows to display without requiring scrolling. For the HORIZONTAL_WRAP or VERTICAL_WRAP layout orientations, it defines how the cells wrap. The default value of this property is VERTICAL.
void setSelectionMode(int) int getSelectionMode()	Set or get the selection mode. Acceptable values are: SINGLE_SELECTION, SINGLE_INTERVAL_SELECTION, or MULTIPLE_INTERVAL_SELECTION (the default), which are defined in ListSelectionModel.

int getAnchorSelectionIndex() int getLeadSelectionIndex() int getSelectedIndex() int getMinSelectionIndex() int getMaxSelectionIndex() int[] getSelectedIndices() Object getSelectedValue() Object[] getSelectedValues()	Get information about the current selection as indicated.

Example: Write a program to create a user interface for students biodata. (Demo example for JComboBox, JList).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData03 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lblplang,lblyear;
        JList lst;
        JComboBox jcb;
        public StudentBioData03()
        {
                //Step 4- Create objects for the declared instances
                lblplang=new JLabel("Programming Lang.");
                lblyear=new JLabel("Academic Year");
                Object obj[]={"C","C++","C#","Java"};
                lst=new JList(obj);
                jcb=new JComboBox();
                jcb.addItem("First Year");
                jcb.addItem("Second Year");
                jcb.addItem("Third Year");
//Step 5 – Add all the objects in the Content Pane with Layout
                Container con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblplang);
                con.add(lst);
                con.add(lblyear);
                con.add(jcb);
        }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
public static void main(String args[])
{
        //Step 7 – Create object of class in main method
        StudentBioData03 sbd=new StudentBioData03();
        sbd.setSize(150,200);
        sbd.setVisible(true);
}//main
}//class
```

2.1.9 Menus

JMenuBar

An implementation of a menu bar. You add JMenu objects to the menu bar to construct a menu. When the user selects a JMenu object, its associated JPopupMenu is displayed, allowing the user to select one of the JMenuItems on it.

JMenu

An implementation of a menu -- a popup window containing JMenuItems that is displayed when the user selects an item on the JMenuBar. In addition to JMenuItems, a JMenu can also contain JSeparators.

In essence, a menu is a button with an associated JPopupMenu. When the "button" is pressed, the JPopupMenu appears. If the "button" is on the JMenuBar, the menu is a top-level window. If the "button" is another menu item, then the JPopupMenu is "pull-right" menu.

JMenuItem

An implementation of an item in a menu. A menu item is essentially a button sitting in a list. When the user selects the "button", the action associated with the menu item is performed. A JMenuItem contained in a JPopupMenu performs exactly that function.

Constructor or Method	**Purpose**
JMenuBar()	Creates a menu bar.

JMenu()
JMenu(String)
	Creates a menu. The string specifies the text to display for

	the menu.

JMenuItem()
JMenuItem(String)
JMenuItem(Icon)
JMenuItem(String, Icon)
JMenuItem(String, int)
	Creates an ordinary menu item. The icon argument, if present, specifies the icon that the menu item should display. Similarly, the string argument specifies the text that the menu item should display. The integer argument specifies the keyboard mnemonic to use. You can specify any of the relevant VK constants defined in the KeyEvent class. For example, to specify the A key, use KeyEvent.VK_A.

JMenuItem add(JMenuItem) JMenuItem add(String)	Adds a menu item to the current end of the popup menu. If the argument is a string, then the menu automatically creates a JMenuItem object that displays the specified text.
JMenu add(JMenu)	Creates a menu bar.
void setJMenuBar(JMenuBar) JMenuBar getJMenuBar()	Sets or gets the menu bar of an applet, dialog, frame, internal frame, or root pane.
void setEnabled(boolean)	If the argument is true, enable the menu item. Otherwise, disable the menu item.
void setMnemonic(int)	Set the mnemonic that enables keyboard navigation to the menu or menu item. Use one of the VK constants defined in the KeyEvent class.
void setAccelerator(KeyStroke)	Set the accelerator that activates the menu item.
void addActionListener(ActionListener) void addItemListener(ItemListener)	Add an event listener to the menu item. See Handling Events from Menu Items for details.

Example:Create an animation with a single line, which changes its position in a clock-wise direction. This line should

produce an effect of a spinning line. In the same example give option for clock-wise or anti -clock-wise spinning. Also provide options to start and stop the animation. Demonstrate the animation on the screen.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class LineAnimation extends JFrame implements
ActionListener
{
        Timer t; JLabel
        lblText;
        JMenuBar mb;
        JMenu m1,m2;
        JMenuItem mi1,mi2,mi3,mi4;
        double theta=0.0;
        int x,y,incr=-1;
        //Graphics g;
        Container con;
        public LineAnimation()
        {
                super("Line Animation");
                lblText=new JLabel("Line Animation");
                mb=new JMenuBar();
                m1=new JMenu("Motion"); m2=new
                JMenu("Direction"); mi1=new
                JMenuItem("Start"); mi2=new
                JMenuItem("Stop"); mi3=new
                JMenuItem("Clock-wise");
                mi4=new JMenuItem("Anti-Clock-wise");
                t=new Timer(100,this);

                mi1.addActionListener(this);
                mi2.addActionListener(this);
                mi3.addActionListener(this);
                mi4.addActionListener(this);

                con=getContentPane();
                con.setLayout(new FlowLayout());
                con.add(lblText);
                m1.add(mi1); m1.add(mi2);
                m2.add(mi3); m2.add(mi4);
                mb.add(m1); mb.add(m2);
                setJMenuBar(mb);
                //g=getGraphics();
                 setDefaultCloseOperation(EXIT_ON_CLOSE);
        }
        public void actionPerformed(ActionEvent ae)
        {
                if(ae.getSource()==t)
                        repaint();

                if(ae.getSource()==mi1)
                        t.start();

                if(ae.getSource()==mi2)
                        t.stop();

                if(ae.getSource()==mi3)
                        incr=1;

                if(ae.getSource()==mi4)
                        incr=-1;
        }
        public void paint(Graphics g)
        {
                g.clearRect(0,0,300,300);
                x=(int)(100*Math.cos(theta*Math.PI/180));
                y=(int)(100*Math.sin(theta*Math.PI/180));
                g.drawLine(150+x,150+y,150-x,150-y);
                theta+=incr;
        }
        public static void main(String args[])
        {
                LineAnimation la=new LineAnimation();
                la.setVisible(true);
                la.setSize(300,300);
        }//main
}//class
```

2.1.10 JTable

The JTable is used to display and edit regular two-dimensional tables of cells. The JTable has many facilities that make it possible to customize its rendering and editing but provides defaults for these features so that simple tables can be set up easily.

The JTable uses integers exclusively to refer to both the rows and the columns of the model that it displays. The JTable simply takes a tabular range of cells and uses getValueAt(int, int) to retrieve the values from the model during painting. By default, columns may be rearranged in the JTable so that the view's columns appear in a different order to the columns in the model. This does not affect the implementation of the model at all: when the columns are reordered, the JTable maintains the new order of the columns internally and converts its column indices before querying the model.

Constructors	
JTable()	Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.
JTable(int numRows, int numColumns)	Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.
JTable(Object[][] rowData, Object[] columnNames)	Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.
JTable(Vector rowData, Vector columnNames)	Constructs a JTable to display the values in the Vector of Vectors, rowData, with column names, columnNames.
Methods	
void clearSelection()	Deselects all selected columns and rows.
int getColumnCount()	Returns the number of columns in the column model.

String getColumnName(int column)	Returns the name of the column appearing in the view at column position column.
int getRowCount()	Returns the number of rows in this table's model.
int getRowHeight()	Returns the height of a table row, in pixels.
int getSelectedColumn()	Returns the index of the first selected column, -1 if no column is selected.
int getSelectedRow()	Returns the index of the first selected row, -1 if no row is selected.
Object getValueAt(int row, int column)	Returns the cell value at row and column.
void selectAll()	Selects all rows, columns, and cells in the table.
void setValueAt(Object aValue, int row, int column)	Sets the value for the cell in the table model at row and column.

Example: Write a program to create a user interface for students biodata. (Demo example for JTable, JScrollPane).

```
//Step 1 – import all the required packages
import javax.swing.*;
import java.awt.*;
/* Step 2 – Decide the class name & Container class
(JFrame/JApplet) to be used */
public class StudentBioData04 extends JFrame
{
        //Step 3 – Create all the instances required
        JLabel lbldata;
        JTable tbldata;
        JScrollPane jsp;
        public StudentBioData04()
        {
                //Step 4- Create objects for the declared instances
                lbldata=new JLabel("Educational Details");
                Object header[]={"Year","Degree","Class"};
```

```
                    Object rowdata[][]={
                            {"2005","SSC","First"},
                            {"2007","HSC","Second"},
                            {"2011","BSc","Distinction"}
                        };
                    tbldata=new JTable(rowdata,header);
                    jsp=new JScrollPane(tbldata);
//Step 5 – Add all the objects in the Content Pane with Layout
                    Container con=getContentPane();
                    con.setLayout(new FlowLayout());
                    con.add(lbldata);
                    con.add(jsp);
            }//constructor
//Step 6 – Event Handling. (Event handling code will come here)
            public static void main(String args[])
            {
            //Step 7 – Create object of class in main method
            StudentBioData04 sbd=new StudentBioData04();
            sbd.setSize(150,200);
            sbd.setVisible(true);
            }//main
}//class
```

Note: If you combine all the four student bio-data program, you would end up with the final program with all the components included. Just remove the common code from all the files.

2.1.11 JTree

A JTree is a component that displays information in a hierarchical format. Steps for creating a JTree are as follows:

Create an Instance of JTree.

Create objects for all the nodes required with the help of DefaultMutableTreeNode, which implemets Mutable TreeNode interface which extends the TreeNode interface. The TreeNode interface declares methods that obtains information about a TreeNode. To create a heirarchy of TreeNodes the add() of the DefaultMutableTreeNode can be used.

Methods of TreeNode interface	
TreeNode getChildAt(int childIndex)	Returns the child TreeNode at index childIndex.
int getChildCount()	Returns the number of children TreeNodes the receiver contains.
int getIndex(TreeNode node)	Returns the index of node in the receivers children.
TreeNode getParent()	Returns the parent TreeNode of the receiver.

Methods of MutuableTreeNode	
void insert(MutableTreeNode child, int index)	Adds child to the receiver at index.
void remove(int index)	Removes the child at index from the receiver.
void remove (MutableTreeNode node)	Removes node from the receiver.
void setParent (MutableTreeNode newParent)	Sets the parent of the receiver to newParent.

Methods of DefaultMutableTreeNode	
void add (MutableTreeNode newChild)	Removes newChild from its parent and makes it a child of this node by adding it to the end of this node's child array.
int getChildCount()	Returns the number of children of this node.
TreeNode[] getPath()	Returns the path from the root, to get to this node.
TreeNode getRoot()	Returns the root of the tree that contains this node.
boolean isRoot()	Returns true if this node is the root of the tree.

Create the object of the tree with the top most node as an argument.

Use the add method to create the heirarchy.

Create an object of JScrollpane and add the JTree to it. Add the scroll pane to the content pane.

Event Handling

The JTree generates a TreeExpansionEvent which is in the package javax.swing.event. The getPath() of this class returns a Tree Path object that describes the path to the changed node. The addTreeExpansionListener and removeTreeExpansionListener methods allows listeners to register and unregister for the notifications. The TreeExpansionListener interface provides two methods:

void treeCollapsed (TreeExpansionEvent event)	Called whenever an item in the tree has been collapsed.
void treeExpanded (TreeExpansionEvent event)	Called whenever an item in the tree has been expanded.

Example: Demo example for tree.

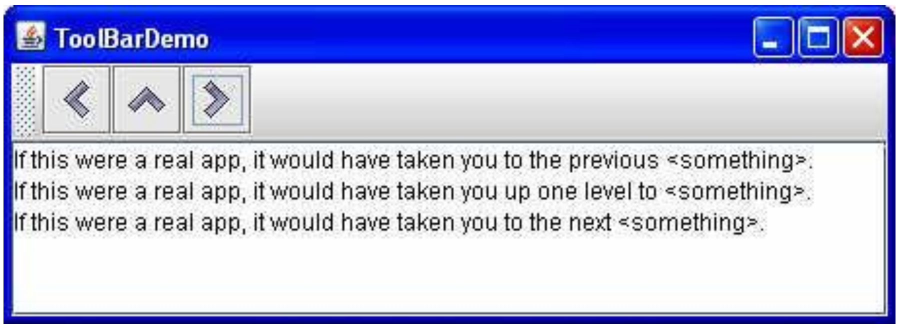
```
import java.awt.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.awt.event.*;

//<applet code="TreeDemo" height=150 width=120></applet>

public class TreeDemo extends JApplet
{
JTree tree;
JTextField txt1;
public void init()
{
        Container con=getContentPane();
        con.setLayout(new BorderLayout());

        DefaultMutableTreeNode top=new
                        DefaultMutableTreeNode("Option");

        DefaultMutableTreeNode stream=new
                        DefaultMutableTreeNode("Stream");
```

```
DefaultMutableTreeNode arts=new
                    DefaultMutableTreeNode("Arts");
DefaultMutableTreeNode science=new
                    DefaultMutableTreeNode("Science");
DefaultMutableTreeNode comm=new
                    DefaultMutableTreeNode("Commerce");

DefaultMutableTreeNode year=new
                    DefaultMutableTreeNode("Year");

DefaultMutableTreeNode fy=new
                    DefaultMutableTreeNode("FY");
DefaultMutableTreeNode sy=new
                    DefaultMutableTreeNode("SY");
DefaultMutableTreeNode ty=new
                    DefaultMutableTreeNode("TY");

top.add(stream);
stream.add(arts); stream.add(comm); stream.add(science);

top.add(year);
year.add(fy); year.add(sy); year.add(ty);

tree=new JTree(top);
JScrollPane jsp=new JScrollPane(
tree,
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED,
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED);

con.add(jsp,BorderLayout.CENTER);
txt1=new JTextField("",20);
con.add(txt1,BorderLayout.SOUTH);

tree.addMouseListener(new MouseAdapter()
{
        public void mouseClicked(MouseEvent me)
        {
                    doMouseClicked(me);
        }
});
```

```
}// init()

void doMouseClicked(MouseEvent me)
{
        TreePath tp=tree.getPathForLocation(me.getX(),me.getY());
        if(tp!=null)
        {
                txt1.setText(tp.toString());
        }
        else
        {
                txt1.setText("");
        }
}//doMouse
}//class
```

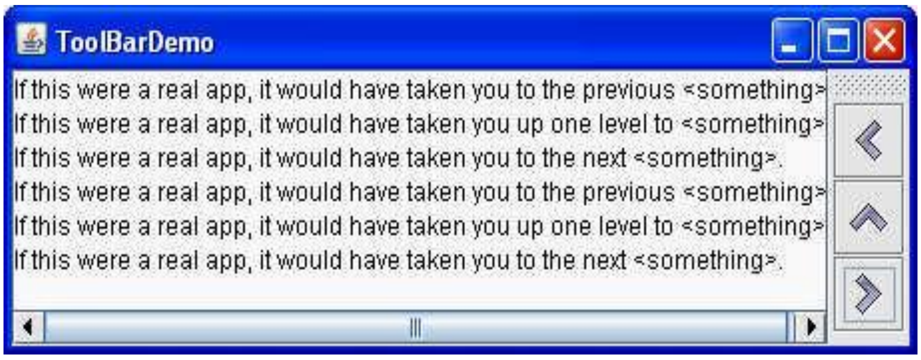
2.1.12 JToolBar

A JToolBar is a container that groups several components — usually buttons with icons — into a row or column. Often, tool bars provide easy access to functionality that is also in menus. The following images show an application named ToolBarDemo that contains a tool bar above a text area.



```
If this were a real app, it would have taken you to the previous <something>.
If this were a real app, it would have taken you up one level to <something>.
If this were a real app, it would have taken you to the next <something>.
```

By default, the user can drag the tool bar to another edge of its container or out into a window of its own. The next figure shows how the application looks after the user has dragged the tool bar to the right edge of its container.



If this were a real app, it would have taken you to the previous <something>
If this were a real app, it would have taken you up one level to <something>
If this were a real app, it would have taken you to the next <something>.
If this were a real app, it would have taken you to the previous <something>
If this were a real app, it would have taken you up one level to <something>
If this were a real app, it would have taken you to the next <something>.

For the drag behavior to work correctly, the tool bar must be in a container that uses the BorderLayout layout manager. The component that the tool bar affects is generally in the center of the container. The tool bar must be the only other component in the container, and it must not be in the center.

Method or Constructor	**Purpose**

JToolBar()
JToolBar(int)
JToolBar(String)
JToolBar(String, int)
	Creates a tool bar. The optional int parameter lets you specify the orientation; the default is HORIZONTAL. The optional String parameter allows you to specify the title of the tool bar's window if it is dragged outside of its container.

Component add(Component)	Adds a component to the tool bar. You can associate a button with an Action using the setAction(Action) method defined by the AbstractButton.
void addSeparator()	Adds a separator to the end of the tool bar.
void setFloatable(boolean) boolean isFloatable()	The floatable property is true by default, and indicates that the user can drag the tool bar out into a separate window. To turn off tool bar dragging, use toolBar.setFloatable(false). Some types of look and feel might ignore this property.
void setRollover(boolean) boolean isRollover()	The rollover property is false by default. To make tool bar buttons be indicated visually when the user passes over them with the cursor, set this property to true. Some types of look and feel might ignore this property.

Example: Demo example for JToolbar

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class ToolBarDemo extends JFrame implements
ActionListener
{
JToolBar toolBar;
JButton cmdPrev,cmdUp,cmdNext;
JTextArea textArea;
JScrollPane scrollPane;
String newline = "\n";

public ToolBarDemo()
{
super("Tool bar Demo");
toolBar = new JToolBar("Still draggable");

cmdPrev=new JButton("Prev",new ImageIcon("Back24.gif"));
cmdUp=new JButton("Up",new ImageIcon("Up24.gif"));
cmdNext=new JButton("Next",new ImageIcon("Forward24.gif"));

toolBar.add(cmdPrev);
toolBar.add(cmdUp);
toolBar.add(cmdNext);

textArea = new JTextArea(5, 30);
textArea.setEditable(false);
scrollPane = new JScrollPane(textArea);

cmdPrev.addActionListener(this);
cmdUp.addActionListener(this);
cmdNext.addActionListener(this);

Container con=getContentPane();
con.setLayout(new BorderLayout());
con.add(toolBar, BorderLayout.NORTH);
con.add(scrollPane, BorderLayout.CENTER);
setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}
public void actionPerformed(ActionEvent e)
{
String cmd = e.getActionCommand();
String description = null;
if(cmd.equals("Prev"))
{description = "taken you to the previous <something>.";}
if(cmd.equals("Up"))
{description = "taken you up one level to <something>.";}
if(cmd.equals("Next"))
{description = "taken you to the next <something>.";
}
textArea.append("If this were a real app, it would have "+description
+ newline);
textArea.setCaretPosition(textArea.getDocument().getLength());
}

public static void main(String[] args)
{
ToolBarDemo tb=new ToolBarDemo();
tb.setSize(300,300); tb.setVisible(true);

}
}
```

2.1.13 JColorChooser

JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color. This class provides three levels of API:

1. A static convenience method which shows a modal color-chooser dialog and returns the color selected by the user.
2. A static convenience method for creating a color-chooser dialog where ActionListeners can be specified to be invoked when the user presses one of the dialog buttons.
3. The ability to create instances of JColorChooser panes directly (within any container). PropertyChange listeners can be added to detect when the current "color" property changes.

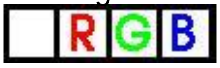
Creating and Displaying the Color Chooser

Method or Constructor	**Purpose**

JColorChooser()
JColorChooser(Color)
JColorChooser(ColorSelectionModel)
	Create a color chooser. The default constructor creates a color chooser with an initial color of Color.white. Use the second constructor to specify a differentinitialcolor.The ColorSelectionModel argument, when present, provides the color chooser with a color selection model.

Color showDialog(Component, String, Color)	Create and show a color chooser in a modal dialog. The Component argument is the parent of the dialog, the String argument specifies the dialog title, and the Color argument specifies the chooser's initial color.
JDialog createDialog(Component, String, boolean, JColorChooser, ActionListener, ActionListener)	Create a dialog for the specified color chooser. As with showDialog, the Component argument is the parent of the dialog and the String argument specifies the dialog title. The other arguments are as follows: the boolean specifies whether the dialog is modal, the JColorChooser is the color chooser to display in the dialog, the first ActionListener is for the **OK** button, and the second is for the **Cancel** button.

Setting or Getting the Current Color

Method	Purpose
void setColor(Color) void setColor(int, int, int) void setColor(int) Color getColor()	Set or get the currently selected color. The three integer version of the setColor method interprets the three integers together as an RGB color. The single integer version of the setColor method divides the integer into four 8-bit bytes and interprets the integer as an RGB color as follows: 

2.1.14 JFileChooser

File choosers provide a GUI for navigating the file system, and then either choosing a file or directory from a list, or entering the name of a file or directory. To display a file chooser, you usually use the JFileChooser API to show a modal dialog containing the file chooser. Another way to present a file chooser is to add an instance of JFileChooser to a container.

The JFileChooser API makes it easy to bring up open and save dialogs. The type of look and feel determines what these standard dialogs look like and how they differ. In the Java look and feel, the save dialog looks the same as the open dialog, except for the title on the dialog's window and the text on the button that approves the operation.

Creating and Showing the File Chooser

Method or Constructor	Purpose

JFileChooser()
JFileChooser(File)
JFileChooser(String)
	Creates a file chooser instance. The File and String arguments, when present, provide the initial directory.

int showOpenDialog(Component) int showSaveDialog(Component) int showDialog(Component, String)	Shows a modal dialog containing the file chooser. These methods return APPROVE_OPTION if the user approved the operation and CANCEL_OPTION if the user cancelled it. Another possible return value is ERROR_OPTION, which means an unanticipated error occurred.

Selecting Files and Directories

Method	Purpose
void setSelectedFile(File) File getSelectedFile()	Sets or obtains the currently selected file or (if directory selection has been enabled) directory.
void setSelectedFiles(File[]) File[] getSelectedFiles()	Sets or obtains the currently selected files if the file chooser is set to allow multiple selection.

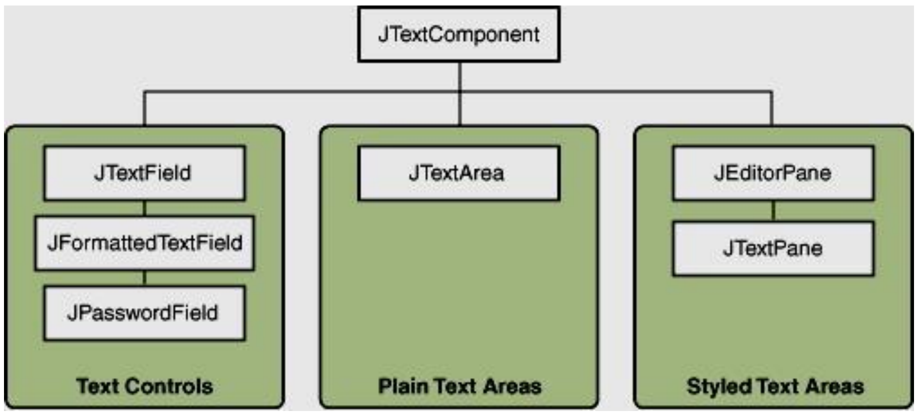
void setFileSelectionMode(int) void getFileSelectionMode() boolean is Directory Selection Enabled() boolean is File Selection Enabled()	Sets or obtains the file selection mode. Acceptable values are FILES_ONLY (the default), DIRECTORIES_ONLY, and FILES_AND_DIRECTORIES. Interprets whether directories or files are selectable according to the current selection mode.
void setMultiSelectionEnabled(boolean) boolean isMultiSelectionEnabled()	Sets or interprets whether multiple files can be selected at once. By default, a user can choose only one file.

2.1.15 Using Text Components

Swing text components display text and optionally allow the user to edit the text. Programs need text components for tasks ranging from the straightforward (enter a word and press Enter) to the complex (display and edit styled text with embedded images in an Asian language).

Swing provides six text components, along with supporting classes and interfaces that meet even the most complex text requirements. In spite of their different uses and capabilities, all Swing text components inherit from the same superclass, JTextComponent, which provides a highly-configurable and powerful foundation for text manipulation.

The following figure shows the JTextComponent hierarchy.



The following table tells you more about what you can do with each kind of text component.

Group	**Description**	**Swing Classes**
Text Controls	Also known simply as text fields, text controls can display only one line of editable text. Like buttons, they generate action events. Use them to get a small amount of textual information from the user and perform an action after the text entry is complete.	JTextField and its sub classes JPassword Field and JFormattedTextField
Plain Text Areas	JTextArea can display multiple lines of editable text. Although a text area can display text in any font, all of the text is in the same font. Use a text area to allow the user to enter unformatted text of any length or to display unformatted help information.	JTextArea
Styled Text Areas	A styled text component can display editable text using more than one font. Some styled text components allow embedded images and even embedded components. Styled text components are powerful and multi-faceted components suitable for high-end needs, and offer more avenues for customization than the other text components. Because they are so powerful and flexible, styled text components typically require more initial programming to set up and use. One exception is that editor panes can be easily loaded with formatted text from a URL, which makes them useful for displaying uneditable help information.	JEditorPane and its subclass JTextPane

2.2 INTERFACE ACTION

The Action interface provides a useful extension to the ActionListener interface in cases where the same functionality may be accessed by several controls.

public interface Action extends ActionListener

In addition to the actionPerformed method defined by the ActionListener interface, this interface allows the application to define, in a single place:

One or more text strings that describe the function. These strings can be used, for example, to display the flyover text for a button or to set the text in a menu item.

One or more icons that depict the function. These icons can be used for the images in a menu control, or for composite entries in a more sophisticated user interface.

The enabled/disabled state of the functionality. Instead of having to separately disable the menu item and the toolbar button, the application can disable the function that implements this interface. All components which are registered as listeners for the state change then know to disable event generation for that item and to modify the display accordingly.

Certain containers, including menus and tool bars, know how to add an Action object. When an Action object is added to such a container, the container:

Creates a component that is appropriate for that container (a tool bar creates a button component, for example).

Gets the appropriate property(s) from the Action object to customize the component (for example, the icon image and flyover text).

Checks the intial state of the Action object to determine if it is enabled or disabled, and renders the component in the appropriate fashion.

Registers a listener with the Action object so that is notified of state changes. When the Action object changes from enabled to disabled, or back, the container makes the appropriate revisions to the event-generation mechanisms and renders the component accordingly.

For example, both a menu item and a toolbar button could access a Cut action object. The text associated with the object is specified as "Cut", and an image depicting a pair of scissors is specified as its icon. The Cut action-object can then be added to a menu and to a tool bar. Each container does the appropriate things with the object, and invokes its actionPerformed method when the component associated with it is activated. The application can then disable or enable the application object without worrying about what user-interface components are connected to it.

This interface can be added to an existing class or used to create an adapter (typically, by subclassing AbstractActio). The Action object can then be added to multiple action-aware containers and connected to Action-capable components. The GUI controls can then be activated or deactivated all at once by invoking theAction object's setEnabled method.

Note that Action implementations tend to be more expensive in terms of storage than a typical ActionListener, which does not offer the benefits of centralized control of functionality and broadcast of property changes. For th is reason, you should take care to only use Actions where their benefits are desired, and use simple ActionListeners elsewhere.

Method Summary	
void	addPropertyChangeListener(PropertyChangeListener listener) Adds a PropertyChange listener.
Object	getValue(String key) Gets one of this object's properties using the associated key.
boolean	isEnabled() Returns the enabled state of the Action.
void	putValue(String key, Object value) Sets one of this object's properties using the associated key.
void	removePropertyChangeListener(PropertyChangeListener listener) Removes a PropertyChange listener.
void	setEnabled(boolean b) Sets the enabled state of the Action.

Methods inherited from interface java.awt.event.ActionListener
actionPerformed

2.3 PRINTING WITH 2D API

The Java 2D printing API is the java.awt.print package that is part of the Java 2 SE,version 1.2 and later. The Java 2D printing API provides for creating a PrinterJob, displaying a printer dialog to the user, and printing paginated graphics using the same java.awt.Graphics and java.awt.Graphics2D classes that are used to draw to the screen.

Many of the features that are new in the Java Print Service, such as printer discovery and specification of printing attributes, are also very important to users of the Java 2D printing API. To make these features available to users of Java 2D printing, the java.awt.print package has been updated for version 1.4 of the JavaTM 2 SE to allow access to the Java^{TM} Print Service from the Java 2D printing API.

Developers of Java 2D printing applications have four ways of using the Java Print Service with the Java 2D API:

- Print 2D graphics using PrinterJob.
- Stream 2D graphics using PrinterJob
- Print 2D graphics using using DocPrintJob and a service-formatted DocFlavor
- Stream 2D graphics using DocPrintJob and a service-formatted DocFlavor

2.4 JAVA PRINT SERVICES API

The Java Print Service (JPS) is a new Java Print API that is designed to support printing on all Java platforms, including platforms requiring a small footprint, but also supports the current Java 2 Print API. This unified Java Print API includes extensible print attributes based on the standard attributes specified in the Internet Printing Protocol (IPP) 1.1 from the IETF Specification, RFC 2911. With the attributes, client and server applications can discover and select printers that have the capabilities specified by the attributes. In addition to the included StreamPrintService, which allows applications to transcode print data to different formats, a third party can dynamically install their own print services through the Service Provider Interface.

The Java Print Service API unifies and extends printing on the Java platform by addressing many of the common printing needs of both client and server applications that are not met by the current Java printing APIs. In addition to supporting the current Java 2D printing features, the Java Print Service offers many improvements, including:

• Both client and server applications can discover and select printers based on their capabilities and specify the properties of a print job. Thus, the JPS provides the missing component in a printing subsystem: programmatic printer discovery.

• Implementations of standard IPP attributes are included in the JPS API as first-class objects.

• Applications can extend the attributes included with the JPS API.

• Third parties can plug in their own print services with the Service Provider Interface.

How Applications Use the Java Print Service

A typical application using the Java Print Service API performs these steps to process a print request:

1. Obtain a suitable DocFlavor, which is a class that defines the format of the print data.
2. Create and populate an AttributeSet, which encapsulates a set of attributes that describe the desired print service capabilities, such as the ability to print five copies, stapled, and double-sided.
3. Lookup a print service that can handle the print request as specified by the DocFlavor and the attribute set.
4. Create a print job from the print service.
5. Call the print job's print method.

The application performs these steps differently depending on what and how it intends to print. The application can either send print data to a printer or to an output stream. The print data can either be a document in the form of text or images, or a Java object encapsulating 2D Graphics. If the print data is 2D graphics , the print job can be represented by either a DocPrintJob or a PrinterJob. If the print data is a document then a DocPrintJob must be used.

2.6 UNIT END EXERCISE

1) Explain the importance of JComponent Class.
2) Write a Swing program containing a button with the caption "Now" and a textfield. On click of the button, the current date and time should be displayed in a textfield?
3) State and explain any three classes used to create Menus in Swing?
4) How to create a JMenu and add it to a JMenuBar inside a JFrame?
5) Explain the classes used to create a tree in Swing?
6) List and explain any three Text-Entry Components.
7) How Applications Use the Java Print Service?
8) Write a swing program containing three text fields. The first text field accepts first name, second accepts last name and the third displays full name on click of a button.
9) Define a class that enables the drawing of freehand lines on a screen through mouse clicking and dragging. Use anonymous inner classes to implement event listeners.
10) Define a class that displays circle when key C is typed and rectangle when key R is typed.
11) Write a program containing JMenu. It contains menus such as Circle, Rectangle and Exit. When the menu is clicked, appropriate function should be executed as suggested by the menu

JDialog

The JDialog is the main class for creating a dialog window. You can use this class to create a custom dialog, or invoke the many class methods in JOptionPane to create a variety of standard dialogs. Every dialog is dependent on a frame. When that frame is destroyed, so are its dependent dialogs. When the frame is iconified, its dependent dialogs disappear from the screen. When the frame is deiconified, its dependent dialogs return to the screen.

A dialog can be modal. When a modal dialog is visible, it blocks user input to all other windows in the program. The dialogs that JOptionPane provides are modal. To create a non-modal dialog, you must use the JDialog class directly. To create simple, standard dialogs, you use the JOptionPane class. The ProgressMonitor class can put up a dialog that shows the progress of an operation. Two other classes, JColorChooser and JFileChooser, also supply standard dialogs.

Constructors :

1	JDialog(Frame owner)	Creates a non-modal dialog without a title with the specifed Frame as its owner.
2	JDialog(Frame owner, String title, boolean modal)	Creates a modal or non-modal dialog with the specified title and the specified owner Frame.

Methods:

1	protected void dialogInit() - Called by the constructors to init the JDialog properly.
2	Container getContentPane() - Returns the contentPane object for this dialog.
3	void setDefaultCloseOperation(int operation) - Sets the operation which will happen by default when the user initiates a "close" on this dialog.

4	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.

1.3.3 JApplet

JApplet is an extended version of java.applet.Applet that adds support for the JFC/Swing component architecture. The JApplet class is slightly incompatible with java.applet.Applet. JApplet contains a JRootPane as it's only child. The contentPane should be the parent of any children of the JApplet.

To add the child to the JApplet's contentPane we use the getContentPane() method and add the components to the contentPane. The same is true for setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane() instead of the JApplet itself. The contentPane() will always be non-null. Attempting to set it to null will cause the JApplet to throw an exception. The default contentPane() will have a BorderLayout manager set on it.

JApplet adds two major features to the functionality that it inherits from java.applet.Applet. First, Swing applets provide support for assistive technologies. Second, because JApplet is a top-level Swing container, each Swing applet has a root pane. The most noticeable results of the root pane's presence are support for adding a menu bar and the need to use a content pane.

Constructors :

JApplet() - Creates a swing applet instance.

Methods:

1	Container getContentPane() - Returns the contentPane object for this applet.
2	void setJMenuBar(JMenuBar menuBar) - Sets the menubar for this applet.
3	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.
4	void update(Graphics g) - Just calls paint(g).

JWindow

A JWindow is a container that can be displayed anywhere on the user's desktop. It does not have the title bar, window-management buttons, or other trimmings associated with a JFrame, but it is still a "first-class citizen" of the user's desktop, and can exist anywhere on it. The JWindow component contains a JRootPane as its only child. The contentPane should be the parent of any children of the JWindow. From the older java.awt.Window object you would normally do something like this:

```
window.add(child);
```

However, using JWindow you would code:

```
window.getContentPane().add(child);
```

The same is true of setting LayoutManagers, removing components, listing children, etc. All these methods should normally be sent to the contentPane instead of the JWindow itself. The contentPane will always be non-null. Attempting to set it to null will cause the JWindow to throw an exception. The default contentPane will have a BorderLayout manager set on it.

Constructors:

1	JWindow() - Creates a window with no specified owner.
2	JWindow(Frame owner) - Creates a window with the specified owner frame.

Methods :

1	Conatiner getContentPane() - Returns the contentPane object for this applet.
2	void setLayout(LayoutManager manager) - By default the layout of this component may not be set, the layout of its contentPane should be set instead.
3	void update(Graphics g) - Just calls paint(g).
4	void windowInit() - Called by the constructors to init the JWindow properly.