Introduction to AWT

So far, we have looked at the basic constructs of Java. In Java, Graphics allow a user to see and interact with any application. Graphics consists of elements like frame, border, color etc. which a user can see and can interact with (for example, the user can push a button).

In this module, we shall learn to build applications that contain Graphical User Interface (GUI) using the graphic classes that are that are provided along with the JDK. A GUI is nothing but an interface which the user can use to see and interact with a Java program.

Introduction to AWT

The most basic provision in Java for developing a GUI is the Abstract Window Toolkit (AWT) API.

Contents of AWT

Although AWT is very vast and has about 12 packages, only two of these are most commonly used. We shall see these two packages in the next topic. In this topic, we will discuss the basic components that make up the GUI.

There are two types of GUI elements as follows:

1. Component are the elementary GUI entities. Buttons, labels, textfields, etc. are components.
2. Container are those that hold these Components together. Frames and panels are containers.

We should always keep in mind that a component must always be kept inside a container. Every container has a default method add(). Using this method, a container can add a component into itself.

Swings

The other alternative that evolved for GUI programming is Swings.

Swings first appeared in JDK 1.1 as an add-on and has been an integral part of Java from JDK 1.2. The main advantages of Swings is that it is totally portable. Swings are light weight components and provide almost the same look and feel on any platform. But Swings are more advanced than AWT and used for advanced programming of GUI.

The following code demonstrates the use of Container and Component classes provided by Java. In this code, we here create only a single frame.

```
import java.io.*;
import java.awt.*;
class Demo extends Frame
{
  public static void main(String args[])
  {
    Demo d = new Demo();
    d.setVisible(true);
  }
}
```

The java.awt.* line means including all the classes of the AWT library (set of packages). We have created a class Demo which extends the predefined Frame class , which is part of the package we included (java.awt). We have then created an object of Demo and set its visibility to be true. This is because, the frame type object, when created, is invisible by default. It will become visible only when its visibility is set to true.

Try this on your system. Create a Java file which contains the code above and save it on your computer. Then, compile the code, run it, and observe the output. You should be able to see a frame (like a popup). Ensure you are able to get this output before you proceed further.

Packages

A package can simply be thought of as containers for classes. This is because they keep the class name space compartmentalized. This means that, we can create two classes with the same name, but in different packages without any conflicts. In short, Packages help to resolve namespace conflicts.

Every source file that is in package p must begin with the declaration package p; and must be stored in a sub-directory called p. A class declared in a source file with no package belongs to the default 'anonymous' package. A source file not belonging to the package p may refer to class C from package p by using the syntax p.C;

In this section, we shall discuss three packages in the AWT.

java.awt.*

This package contains all the classes required for creating user interfaces and for printing graphics and images. The most widely used classes in this package are as follows:

Button	This class creates a labelled button
Canvas	This represents the blank rectangular area of the screen onto which the application can draw or print.
Frame	This class creates the top level window with title and border.
Label	This class is used for placing text into container.
Panel	Panel represents the simplest container.

We shall learn about these classes in more detail in the following tutorials.

java.awt.event.*

This package contains classes that are required for processing different types of events fired by the AWT components. This contains theActionListener interface, which we will be considering separately at a later part in this module.

javax.swing.*

This provides a set of light weight components that, to the maximum extent possible, work the same way on all platforms. In practice, the use of swings is similar to that of the AWT Components. However, these differ in terms of their internal implementation details.

The main classes in this package are JButton, JFrame, JPanel, JLabel. We will study more details of these in the following tutorials.

Test Yourself

Q1. What is the correct way of importing an entire package ‘pack’?

Show/Hide Answer

Ans. import pack.*

Classes 1

In this section, we shall deal exclusively with the classes JButton and JFrame which are included in the javax.swing.* package, which we have touched upon in the previous section.

We shall study some of the methods that are present in each of these classes and how they are used.

JButton

We will now look at some of the commonly used constructors and methods of the JButton class.

An instance of JButton class can be created using the constructors of the class. The two important constructors that are used to instantiate the JButton class are as follows:

JButton()	This creates a button with no text or icon.
JButton(String text)	This creates a button with the text that is specified within the parenthesis.

The following are the methods that are widely used in this class.

addActionListener(): Adds an actionListener to the button. ActionListener is an interface for receiving actions. This is used because, we sometimes need to process an action event associated with the button. For example, the click of the button may trigger some action. The addActionListener() method gives the button the capability to capture such an event.

setEnabled(): The button, after creation, remains disabled by default. In order to activate the button, we should enable it. This is done using the method setEnabled(true).

setVisible(): This method makes the button either visible or invisible (setVisible(true) for visible).

JFrame

JFrame creates a frame within which all other components are placed. In other words, JFrame creates a container. The constructors for JFrame class are as follows:

JFrame()	This creates a frame that is initially invisible.
JFrame(String text)	This creates a new frame, initially invisible, with the specified text.

The following are the most commonly used methods of the JFrame class.

add(Component comp): This method adds the specified component to end of the container. Suppose, we have a frame named f and have a button component b, then we use the syntax 'f.add(b)' to use this method. If we want to add the component to a specified position in the frame, that can be done using another variation of the add() method (e.g. add(component, BorderLayout.PAGE_START)).

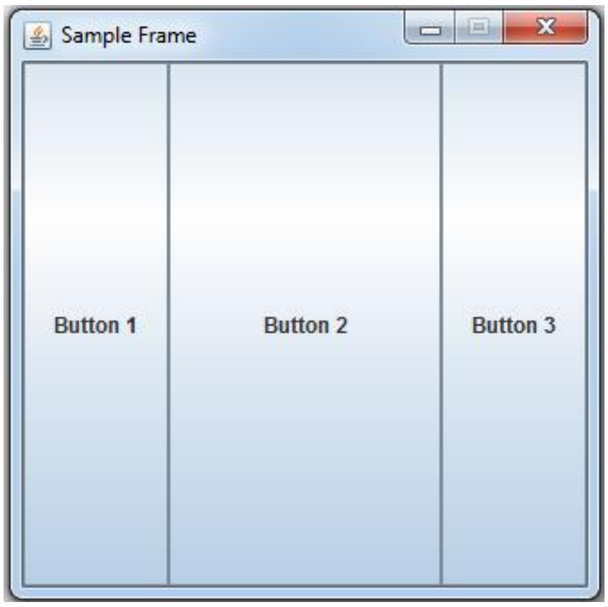
remove(Component comp): This method removes the specified component from the container.

setLayout(): This method is used to select the layout of our choice. Some of the available layouts are BorderLayout, GridLayout, FlowLayout etc. Let us have a look at how to use the setLayout() method with a couple of examples.

setLayout() Using BorderLayout()

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Demo
{
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  Demo()
  {
    JFrame frame = new JFrame("Sample Frame");
    frame.setSize(320,320);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.setLayout(new BorderLayout());
    frame.add(b1, BorderLayout.LINE_START);
    frame.add(b2, BorderLayout.CENTER);
    frame.add(b3, BorderLayout.LINE_END);
  }
  public static void main(String args[])
  {
    Demo d = new Demo();
  }
}
```

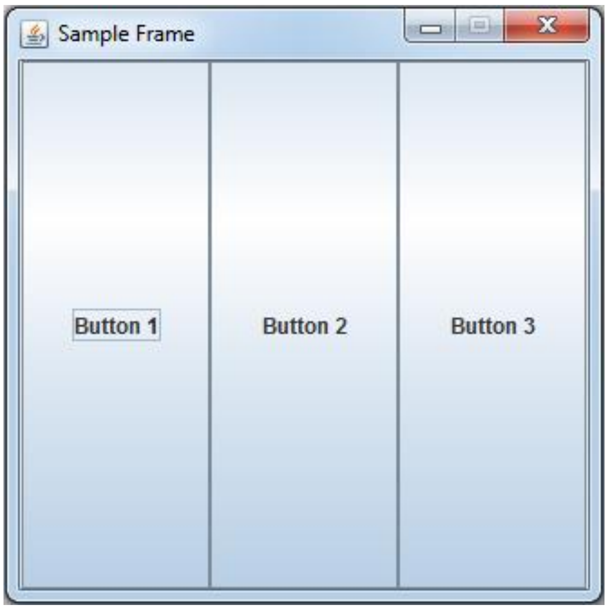
The output of the above code is as follows:



setLayout Using GridLayout()

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Demo
{
  JButton b1 = new JButton("Button 1");
  JButton b2 = new JButton("Button 2");
  JButton b3 = new JButton("Button 3");
  Demo()
  {
    JFrame frame = new JFrame("Sample Frame");
    frame.setSize(320,320);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.setLayout(new GridLayout());
    frame.add(b1);
    frame.add(b2);
    frame.add(b3);
  }
  public static void main(String args[])
  {
    Demo d = new Demo();
  }
}
```

The output of the above code is as follows:



Click here to read a good tutorial for advanced reading on layouts in Java.

setSize(height, width): This method is used to customize the size of the frame to the height and width details that are passed as arguments to it.

ActionEvent

We have another class by the name ActionEvent. This class represents a semantic event which indicates that a component-defined action had occurred. This high level event is generated by a component (may be a button) when the component-specific action occurs. The event is passed to every ActionListener object registered to receive such events using the addActionListener method of the components.

The methods that are associated with this class are:

getSource():	This method returns the object on which the event initially occurred.
getWhen():	This method returns the timestamp of when event has occurred.
getActionCommand():	This method returns the command string associated with this action.

The following code demonstrates the use of these classes and their methods. Here, we create a frame and add a button to it.

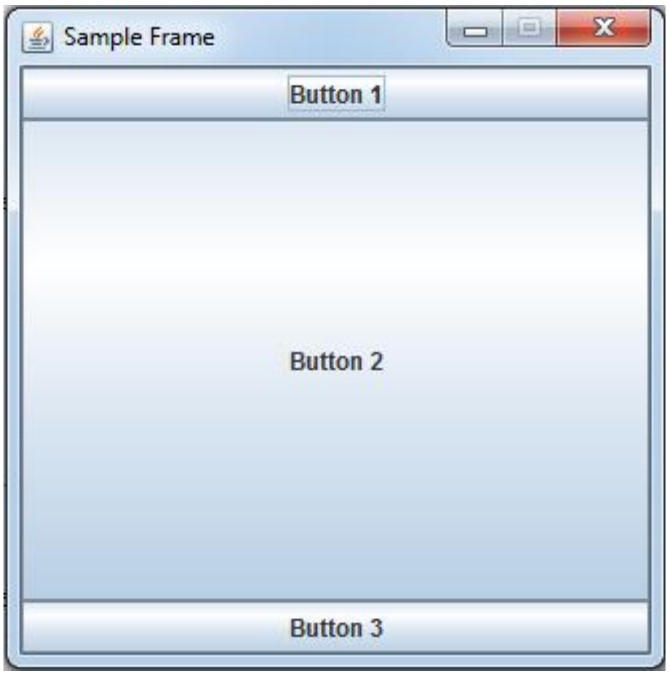
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Demo
{
  JButton b1 = new JButton("Button 1");
  Demo()
  {
    JFrame frame = new JFrame("Sample Frame");
    frame.setSize(320,320);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.setLayout(new BorderLayout());
    frame.add(b1);
  }
  public static void main(String args[])
  {
    Demo d = new Demo();
  }
}
```

In the above code, we have created a frame using the JFrame class and a button using the JButton class. We have named the frame "Sample Frame" and the button "Button 1". We then add the button directly into the frame that we created. Observe carefully the use of methods here. 'frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);' means addition of a close button to close the frame. The frame cannot be resized as we have disabled the same.

Try to implement this code on your system. Try changing the arguments passed to the methods and observe the output.

More Exercises

1. Create a class which display the buttons as shown in the below figure:



Classes 2

In this section, we will be dealing with two more classes; JPanel and TextField.

JPanel

JPanel is a generic light weight container. It is usually used to group a set of components together. Just like we place components in a container, we can also place a container within another container. We place the JPanel item into the JFrame and then place components like buttons, labels, etc into the panel.

The constructors for JPanel class are as follows:

JPanel():	Creates a JPanel item with a default flow layout.
JPanel(LayoutManager layout)	Creates a JPanel item with a specified layout type. These layout types are same as those that we encountered while learning JFrame

The following are some of the most commonly used methods of a JPanel object.

add(Component comp)	This adds the specified component to the panel at the end of the panel.
add(Component comp, int index)	This method adds the specified component at the position that is indicated by the index.

TextField

A TextField object is a text component that allows us to display and edit a single line of text. We can create a TextField object using four types of constructors. These are as follows:

TextField()	This creates blank text field.

TextField("", 20)	This creates blank text field of width 20 columns.
TextField("Hello")	This creates text field with predefined text displayed.
TextField("Hello", 30)	This creates a text field with predefined text and width 30 columns.

The most commonly used methods associated with the TextField are as follows:

addActionListener(ActionListener l):	Adds the specified ActionListener to receive action events from this text field.
setEditable(boolean b):	This method specifies if the text field can be edited or not.
setText(String t):	This method sets the text displayed in the text field to the text specified.

The following code demonstrates the use of these classes. Here, we will build a frame which has two buttons named "Hello" and "World". On clicking the "Hello" button, text "Hello" gets displayed in the textbox. Similarly if we press "World" button, text "World" gets appended to the already existing text in the text field. Note: We are also using some concepts of ActionListener in this code. We shall discuss ActionListener in more detail in the next topic. Don't worry if you do not understand this code completely.

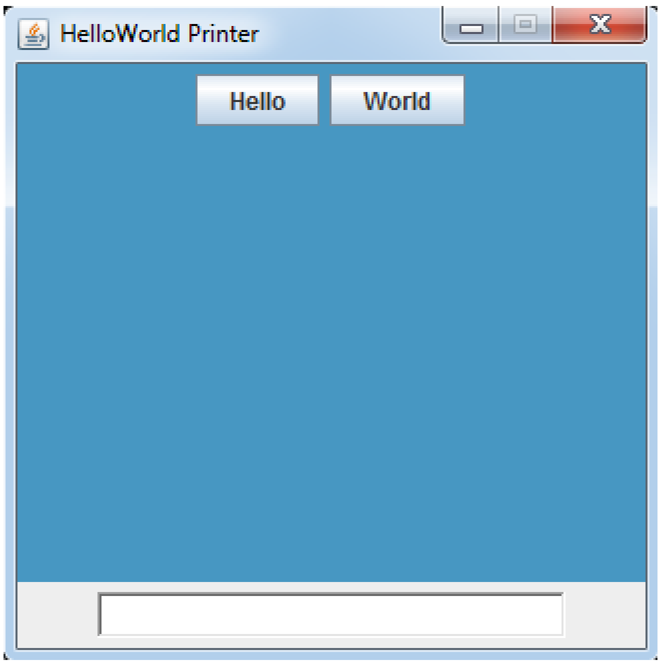
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Demo extends JFrame implements ActionListener
{
  JButton b1 = new JButton("Hello");
  JButton b2 = new JButton("World");
  TextField txt = new TextField("", 30);
  String s = "";
  Demo()
  {
    b1.setVisible(true);
    b2.setVisible(true);
    txt.setVisible(true);
    JFrame frame = new JFrame("HelloWorld Printer");
    frame.setSize(320, 320);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.setLayout(new BorderLayout());
    JPanel p = new JPanel();
    JPanel q = new JPanel();
    q.setVisible(true);
    p.setVisible(true);
    p.setLayout(new GridLayout(1,2));
    q.setLayout(new FlowLayout());
    p.add(b1);
    b1.addActionListener(this);
    p.add(b2);
    b2.addActionListener(this);
    q.add(txt);
    frame.add(p, BorderLayout.CENTER);
    frame.add(q, BorderLayout.SOUTH);
  }
  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == b1)
    {
      txt.setText("Hello");
      s += txt.getText();
    }
    else if(e.getSource() == b2)
    {
      txt.setText("World");
      s += txt.getText();
    }
    txt.setText(s);    }
  public static void main(String args[])
  {
    Demo d = new Demo();
  }
}
```

In the above code, we have first created two buttons and a text field. Then, we have made all of them as visible using the setVisible(true) method. Then, we have created a frame, customized it as shown in the code, and added two panels to it. To one of the panel, we add the two buttons and to the other, we add the textfield. Also, we add both the panels to the center and bottom of the frame, which is denoted by CENTER and SOUTH. The actionPerformed() method of ActionListener interface is implemented by this class. We shall learn more about the ActionListener Interface in the next section.

Try to implement this code on your system. Try to to modify the code by changing the parameters, functionality or by adding more buttons to the existing ones. Observe what changes have to be made in the layout of the panel, its functionality and the resulting output.

More Exercises

1. Create a class which can display the following figure:



Action Listener

ActionListener is an interface. An interface is a special type of class which has only abstract or unimplemented methods. Therefore, when we implement this interface on any class, we must make sure that we provide the implementation for the methods of the ActionListener.

The ActionListener interface is used for receiving action events. The class that is required to process an action event implements this interface. The object of the class that is required to process the action event is linked to a particular component, using the component'saddActionListener() method as we have seen in the previous tutorials.

ActionListener interface has one unimplemented method actionPerformed(ActionEvent e). So, when an action event occurs, actionPerformed() method is invoked on that object which is linked to that particular component. We can then define the implementation of the actionPerformed() method based on our requirement.

The following code demonstrates how the ActionListener Interface is used. This is the same code that we had seen in the previous topic.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
class Demo extends JFrame implements ActionListener
{
  JButton b1 = new JButton("Hello");
  JButton b2 = new JButton("World");
  TextField txt = new TextField("", 30);
  String s = "";
  Demo()
  {
    b1.setVisible(true);
    b2.setVisible(true);
    txt.setVisible(true);
    JFrame frame = new JFrame("HelloWorld Printer");
    frame.setSize(320, 320);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setResizable(false);
    frame.setVisible(true);
    frame.setLayout(new BorderLayout());
    JPanel p = new JPanel();
    JPanel q = new JPanel();
    q.setVisible(true);
    p.setVisible(true);
    p.setLayout(new GridLayout(1,2));
    q.setLayout(new FlowLayout());
    p.add(b1);
    b1.addActionListener(this);
    p.add(b2);
    b2.addActionListener(this);
    q.add(txt);
    frame.add(p, BorderLayout.CENTER);
    frame.add(q, BorderLayout.SOUTH);
  }
  public void actionPerformed(ActionEvent e)
  {
    if(e.getSource() == b1)
    {
      txt.setText("Hello");
      s += txt.getText();
    }
    else if(e.getSource() == b2)
    {
      txt.setText("World");
      s += txt.getText();
    }
    txt.setText(s);    }
  public static void main(String args[])
  {
    Demo d = new Demo();
  }
}
```

In this code, the class 'Demo' is implementing the ActionListener Interface. Note that, the two button components b1 and b2 are defined to invoke the actionPerformed() method. This is done using the addActionListener() method. Also, in class 'Demo', we have provided our implementation of the actionPerformed() method. First, we capture the source of the action event. Based on the source (either button component B1 or B2), we have appended and set the text in the text field.

In this manner, whenever we want to capture the action events that have occurred, and trigger some action based on those event, we should implement the ActionListener Interface and provide our implementation of the actionPerformed() method. The action that is performed may be of any form, for e.g. an arithmetic operation or something which may trigger another action in-turn.

Test Yourself

Q1. Which of the package contains all the event handling interfaces?

Show/Hide Answer

Ans. java.awt.event

More Exercises

1. Modify the above code such that the user is forced to print 'HelloWorld', but not 'HelloHello' or 'WorldWorld'. Hint : Initially only the 'Hello' button should be visible. When you click on it, the 'World' button should become visible and make the 'Hello' button invisible and so on.
2. Design a swing application which has a two text fields to take two numbers and display the H.C.F and L.C.M of the two.
3. Write an program to find the factorial of a number.

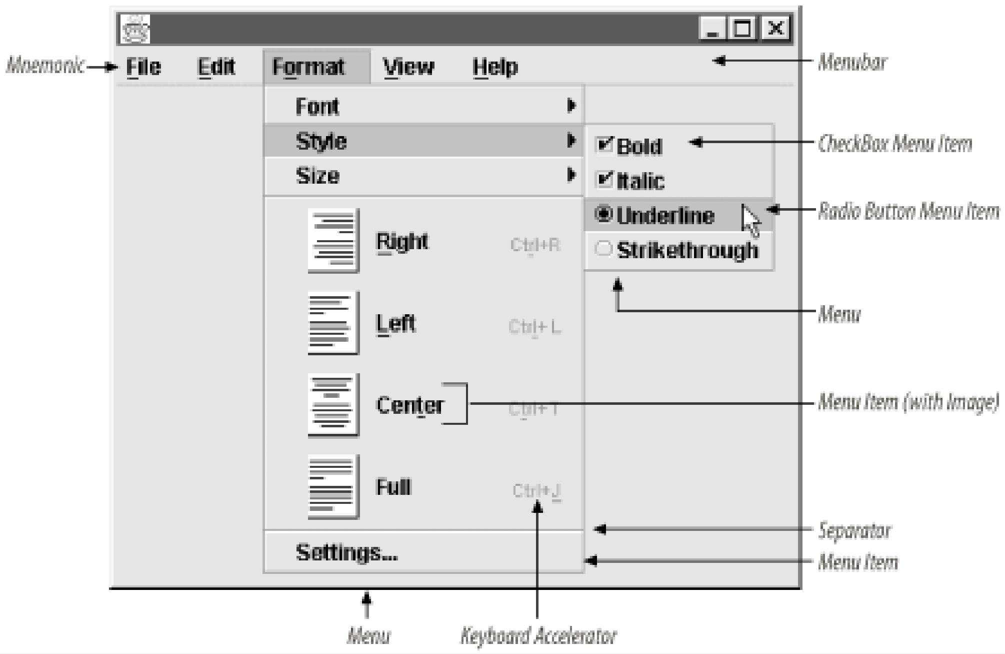
Menu

Swing menu components are subclasses of JComponent. Consequently, they have all the benefits of a Swing component, and you can treat them as such with respect to layout managers and containers. Here are some notable features of the Swing menu system:

- Icons can augment or replace menu items.
- Menu items can be radio buttons.

- Keyboard accelerators can be assigned to menu items; these appear next to the menu item text.
- Most standard Swing components can be used as menu items.

Swing provides familiar menu separators, checkbox menu items, pop-up menus, and submenus for use in your applications. In addition, Swing menus support keyboard accelerators and "underline" style (mnemonic) shortcuts, and you can attach menu bars to the top of Swing frames with a single function that adjusts the frame insets accordingly. On the Macintosh, your application can be configured so that this method places the menu bar at the top of the screen, where users expect to find it. Figure below defines the various elements that make up the menu system in Swing.



```
//Where the GUI is created:
JMenuBar menuBar;

JMenu menu, submenu;

JMenuItem menuItem;

JRadioButtonMenuItem rbMenuItem;

JCheckBoxMenuItem cbMenuItem;


//Create the menu bar.

menuBar = new JMenuBar();


//Build the first menu.

menu = new JMenu("A Menu");

menu.setMnemonic(KeyEvent.VK_A);
menu.getAccessibleContext().setAccessibleDescription(
        "The only menu in this program that has menu items");
menuBar.add(menu);

//a group of JMenuItems
menuItem = new JMenuItem("A text-only menu item",
                         KeyEvent.VK_T);
menuItem.setAccelerator(KeyStroke.getKeyStroke(
        KeyEvent.VK_1, ActionEvent.ALT_MASK));
menuItem.getAccessibleContext().setAccessibleDescription(
        "This doesn't really do anything");
menu.add(menuItem);

menuItem = new JMenuItem("Both text and icon",
                         new ImageIcon("images/middle.gif"));
menuItem.setMnemonic(KeyEvent.VK_B);
menu.add(menuItem);

menuItem = new JMenuItem(new ImageIcon("images/middle.gif"));
menuItem.setMnemonic(KeyEvent.VK_D);
menu.add(menuItem);

//a group of radio button menu items
menu.addSeparator();
ButtonGroup group = new ButtonGroup();
rbMenuItem = new JRadioButtonMenuItem("A radio button menu item");
rbMenuItem.setSelected(true);
rbMenuItem.setMnemonic(KeyEvent.VK_R);
group.add(rbMenuItem);
menu.add(rbMenuItem);

rbMenuItem = new JRadioButtonMenuItem("Another one");
rbMenuItem.setMnemonic(KeyEvent.VK_O);
group.add(rbMenuItem);
menu.add(rbMenuItem);
```

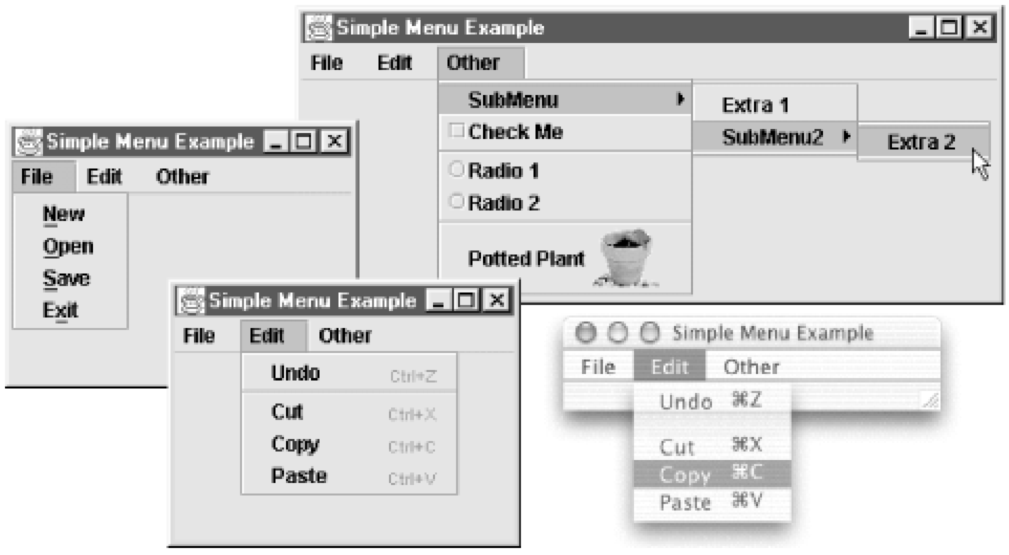
```
//a group of check box menu items
menu.addSeparator();
cbMenuItem = new JCheckBoxMenuItem("A check box menu item");
cbMenuItem.setMnemonic(KeyEvent.VK_C);
menu.add(cbMenuItem);

cbMenuItem = new JCheckBoxMenuItem("Another one");
cbMenuItem.setMnemonic(KeyEvent.VK_H);
menu.add(cbMenuItem);

//a submenu
menu.addSeparator();
submenu = new JMenu("A submenu");
submenu.setMnemonic(KeyEvent.VK_S);

menuItem = new JMenuItem("An item in the submenu");
menuItem.setAccelerator(KeyStroke.getKeyStroke(
        KeyEvent.VK_2, ActionEvent.ALT_MASK));
submenu.add(menuItem);

menuItem = new JMenuItem("Another item");
submenu.add(menuItem);
menu.add(submenu);

//Build second menu in the menu bar.
menu = new JMenu("Another Menu");
menu.setMnemonic(KeyEvent.VK_N);
menu.getAccessibleContext().setAccessibleDescription(
        "This menu does nothing");
menuBar.add(menu);


...
frame.setJMenuBar(theJMenuBar);
```

As the code shows, to set the menu bar for a JFrame, you use the setJMenuBar method. To add a JMenu to a JMenuBar, you use the add(JMenu) method. To add menu items and submenus to a JMenu, you use the add(JMenuItem) method.

Because this code has no event handling, the menus do nothing useful except to look as they should. If you run the example, you'll notice that despite the lack of custom eventhandling, menus and submenus appear when they should, and the check boxes and radio buttons respond appropriately when the user chooses them.

More Exercises

Create a menu bar with three simple menus, attaching mnemonics to the menu items of the File menu and keyboard accelerators to the menu items of the Edit menu. Figure below shows a mosaic of the different menus that the program produces. It also shows how the Edit menu looks on two different platforms, with the proper accelerator key (Control or Command) used on each.



Adapter Class

Inheritance also gives us a alternate technique for creating listener classes.We've seen that listener classes can be created by implementing a particular interface, such as MouseListener

We can also create a listener class by extending an event adapter class .Each listener interface that has more than one method has a corresponding adapter class, such as the MouseAdapter class.Each adapter class implements the corresponding listener and provides empty method definitions.

Java provides something called Adapter classes, which will overload these methods automatically to do nothing.When you derive a listener class from an adapter class, you only need to override the event methods that pertain to the program.

You can then overload the select ones to do what you want

```
public class winAdapter extends WindowAdapter {
//WindowAdapter overloads all 7 methods in the WindowListner interface
//Now we overload those we want to do something
public void windowClosing(WindowEvent e) {
System.exit(0);
}
}
JFrame ourFrame=new JFrame();
ourFrame.addWindowListener(new winAdapter());
We can even bring this a level further. Just use an anonymous inner
class to use the Adapter class.
JFrame ourFrame=new JFrame();
ourFrame.addWindowListener(new WindowAdapter()
{ public void windowClosing(WindowEvent e)
{ System.exit(0);
}
});
```

Empty definitions for unused event methods do not need to be defined because they are provided via inheritance.

An object that implements the WindowListener interface must override these methods:

void windowOpened(WindowEvent e);
void windowClosing(WindowEvent e);
void windowClosed(WindowEvent e);
void windowIconified(WindowEvent e);
void windowDeiconified(WindowEvent e);
void windowActivated(WindowEvent e);
void windowDeactivated(WindowEvent e);

This can obviously be a real pain if you only care about one event. To make it easier for programmers, the designers of Java supplied adapter classes.

Each AWT listener interface with more than one method comes with a companion adapter class.

These adapter classes implement all the methods in the interface but does nothing with them.

This means that the adapter classes satisfy all the technical requirements for implementing the listener interface.

You can then just extend your class from these adapter classes if you don't need to extend it from anything else.

The adapter class for the WindowListener interface is WindowAdapter. You can use it like this:

class Terminator extends WindowAdapter { }

Creating a listener class that extends the WindowAdapter is even easier than that. There is no need to give a name to the listener object.

frame.addWindowListener(new Terminator());

You can go even further by using an anonymous inner class:

```
        frame.addWindowListener(new

        WindowAdapter() {

                        public void windowClosing(WindowEvent e) {

                                System.exit(0)
```

```
                    }
            });

```

You can see how much less code there is.

Test Yourself

Q1. Which is the superclass of all Adapter classes?

Show/Hide Answer

Ans. Applet class.