Java Programming

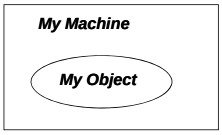
MCA-205
UNIT IV

Learning Objectives

- **RMI (Remote Method Invocation):** Introduction, Steps in creating a Remote Object, Generating Stub & Skeleton, RMI Architecture, RMI packages.
- **Java Bean:** Introduction, Bean Architecture, Using the Bean Development Kit, Creating simple bean-properties, methods and events, Packing beans- the manifest & the jar, Java bean package, Introduction to NetBean.
- Swing : **Introduction to JFC (Java Foundation Classes), Features of Swing, Comparison with AWT, Advanced Control.**

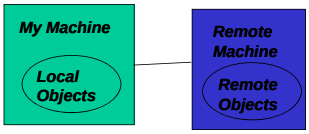
In the Good Old Days...

Only local objects existed



Today's World...

Network and Distributed Objects



Why not just use sockets?

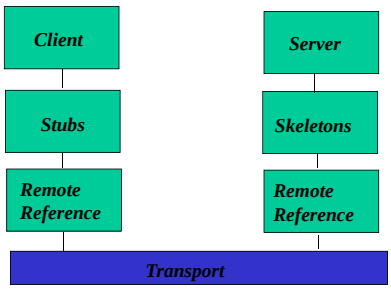
- Socket programming is tedious
- error prone for implementing complex protocols
- best left to “the experts”

- Java RMI allows...
- provide user with a “thin client “
  - allows good performance on lowend workstations
- run server on high end hardware
  - maximize $ investment over many clients
  - server remote from client
- Distributed network object

What RMI is...and isn’t

- Java only solution to the problem of distributed object computing
- Unlike CORBA it is not language independent
- Isn’t meant to replace CORBA
- Good for java only solutions, not easy to integrate with legacy code
- underlying wire protocol (object serialization) is not an open standard; the good news is that since JDK 1.2 it will optionally use the IIOP Protocol (RMI Over IIOP)
- Since it is a Java only solution objects are at the mercy of the java interpreter and JITs for run time performance

Architecture



Java Remote Object Invocation (RMI)

- Overview of RMI
- Java RMI allowed programmer to execute remote function class using the same semantics as local functions calls.

Local Machine (Client)

```
SampleServer remoteObject;
int s;
…
s = remoteObject.sum(1,2);
System.out.println(s);
```

1,2

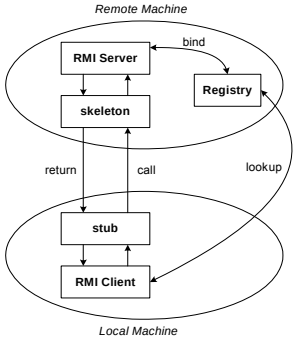
3

Remote Machine (Server)

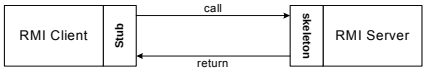
```
public int sum(int a,int b) {
    return a + b;
}
```


The General RMI Architecture

- The server must first bind its name to the registry
- The client lookup the server name in the registry to establish remote references.
- The Stub serializing the parameters to skeleton, the skeleton invoking the remote method and serializing the result back to the stub.



The Stub and Skeleton



- A client invokes a remote method, the call is first forwarded to stub.
- The stub is responsible for sending the remote call over to the server-side skeleton
- The stub opening a socket to the remote server, marshaling the object parameters and forwarding the data stream to the skeleton.
- A skeleton contains a method that receives the remote calls, unmarshals the parameters, and invokes the actual remote object implementation.

Steps for Developing an RMI System

1. Define the remote interface
2. Develop the remote object by implementing the remote interface.
3. Develop the client program.
4. Compile the Java source files.(javac)
5. Generate the client stubs and server skeletons. (rmic)
6. Start the RMI registry.
7. Start the remote server objects.
8. Run the client

The parts...

- **Client** - user interface
- **Server** - data source
- **Stubs** : marshals argument data (serialization) & unmarshals results data (deserialization)
- **Skeletons** (not reqd w/Java 2) :unmarshals argument data and marshals results data
- **Remote Reference Layer:** provides a RemoteRef object that represents the link to the remote service implementation object; encodes and decodes the on-wire protocol; implements the remote object protocols
- Transport layer
  - The Transport Layer makes the connection between JVMs. All connections are stream-based network connections that use TCP/IP.
  - handles the underlying socket handling required for communications
    - ✓sets up and maintains connections
    - ✓communications related error handling

The General Idea

- Instantiate an object on another machine
- Invoke methods on the remote object
- Step 1: Defining the Remote Interface
- To create an RMI application, the first step is the defining of a remote interface between the client and server objects.

```
/* SampleServer.java */
import java.rmi.*;

public interface HelloInterface extends Remote
{
  public String say(int a,int b) throws RemoteException;
}
```

Step 2: Develop the remote object and its interface

- The server is a simple unicast remote server.
- Create server by extending java.rmi.server.UnicastRemoteObject.
- The server uses the RMISecurityManager to protect its resources while engaging in remote communication.

```
/* SampleServerImpl.java */
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
public class Hello extends UnicastRemoteObject
    implementsHelloInterface
{ public Hello() throws RemoteException
  {message=msg;}
public String say() throws RemoteException {return message;}}
```


Step 2: Develop the remote object and its interface

- Implement the remote methods
- The server must bind its name to the registry, the client will look up the server name.
- Use java.rmi.Naming class to bind the server name to registry. In this example the name call "SAMPLE-SERVER".
- In the main method of your server object, the RMI security manager is created and installed.

Step 2: Develop the remote object and its interface

```
/* SampleServerImpl.java */
 public static void main(String args[])
 {try {
      System.setSecurityManager(new RMISecurityManager());
      //set the security manager
      //create a local instance of the object
      SampleServerImpl Server = new SampleServerImpl();
      //put the local instance in the registry
      Naming.rebind("SAMPLE-SERVER" , Server);
      System.out.println("Server waiting.....");}
    catch (java.net.MalformedURLException me)       {
      System.out.println("Malformed URL: " + me.toString());   }
    catch (RemoteException re)  {
       System.out.println("Remote exception: " + re.toString());  }
```

Step 3: Develop the client program

- In order for the client object to invoke methods on the server, it must first look up the name of server in the registry. You use the java.rmi.Naming class to lookup the server name.
- The server name is specified as URL in the from (rmi://host:port/name)
- Default RMI port is 1099.
- The name specified in the URL must exactly match the name that the server has bound to the registry. In this example, the name is “SAMPLE-SERVER”
- The remote method invocation is programmed using the remote interface name (remoteObject) as prefix and the remote method name (sum) as suffix.

Step 3: Develop the client program

```
import java.rmi.*;  import java.rmi.server.*;
public class SampleClient  {public static void main(String[]  args)
  {System.setSecurityManager(new RMISecurityManager());
 try{System.out.println("Security Manager loaded");
 String url = "//localhost/SAMPLE-SERVER";
SampleServer remoteObject = (SampleServer)Naming.lookup(url);
System.out.println("Got remote object");
System.out.println(" 1 + 2 = " + remoteObject.sum(1,2) );}
catch (RemoteException exc) {
 System.out.println("Error in lookup: " + exc.toString()); }
catch (java.net.MalformedURLException exc) {
System.out.println("Malformed URL: " + exc.toString());   }
catch (java.rmi.NotBoundException exc)  { System.out.println("NotBound: " +
   exc.toString());}}}
```


Step 4 & 5: Compile the Java source files & Generate the client stubs and server skeletons

- Assume the program compile and executing at elpis on ~/rmi
- Once the interface is completed, you need to generate stubs and skeleton code. The RMI system provides an RMI compiler (rmic) that takes your generated interface class and procedures stub code on its self.

```
elpis:~/rmi> set CLASSPATH=”~/rmi”
elpis:~/rmi> javac SampleServer.java
elpis:~/rmi> javac SampleServerImpl.java
elpis:~/rmi> rmic SampleServerImpl

elpis:~/rmi> javac SampleClient.java
```

Step 6: Start the RMI registry

- The RMI applications need install to Registry. And the Registry must start manual by call rmiregisty.
- The rmiregistry us uses port 1099 by default. You can also bind rmiregistry to a different port by indicating the new port number as : rmiregistry <new port>

elpis:~/rmi> rmiregistry

- *Remark: On Windows, you have to type in from the command line:*
  > start rmiregistry

Steps 7 & 8: Start the remote server objects & run the client

- Once the Registry is started, the server can be started and will be able to store itself in the Registry.
- Because of the grained security model in Java 2.0, you must setup a security policy for RMI by set java.security.policy to the file policy.all

elpis:~/rmi> java –Djava.security.policy=policy.all SampleServerImpl

elpis:~/rmi> java –Djava.security.policy=policy.all SampleClient

To run

- Start the RMI registry
  - rmiregistry is in the JSDK bin directory
- Start the RMI Server
- Start the RMI Client

RMI Registry

- The RMI Registry is a naming service provided with the JDK as a teaching tool or for a small number of Remote Objects
- Uses port 1099 as its default port
- Can be considered to be a reference implementation
- runs out of steam above a 100 objects
- runs on same machine as the remote object

Introduction to JavaBeans

- JavaBean is a reusable software component that can be manipulated visually in a 'builder tool'. (from JavaBean Specification)
- The JavaBeans API provides a framework for defining reusable, embeddable, modular software components.
- What are JavaBeans?
  - Software components written in Java
  - Connect and Configure Components
  - Builder Tools allow connection and configuration of Beans
  - Begins 'Age of Component Developer'
  - Bringing Engineering methods to Software Engineering (e.g. electronics…)

The JavaBeans API

- Features implemented as extensions to standard Java Class Library
- Main Component Services
  - GUI merging
  - Persistence
  - Event Handling
  - Introspection
  - Application Builder Support

User Interface Merging

- Containers usually have Menus and/or toolbars
- Allows components to add features to the menus and/or toolbars
- Define mechanism for interface layout between components and containers

Persistence

- Components can be stored and retrieved
- Default – inherit serialization
- Can define more complex solutions based on needs of the components

Event Handling

- Defines how components interact
- Java AWT event model serves as basis for the event handling API's
- Provides a consistent way for components to interact with each other

Introspection

- Defines techniques so components can expose internal structure at design time
- Allows development tools to query a component to determine member variables, methods, and interfaces
- Standard naming patterns used
- Based on java.lang.reflect

Application Builder Support

- Provides support for manipulating and editing components at design time
- Used by tools to provide layout and customizing during design
- Should be separate from component
- Not needed at run time

What is a Java Bean?

- A *Java Bean* is a software component that has been designed to be reusable in a variety of different environments.
- There is no restriction on the capability of a Bean. It may perform a simple function, such as checking the spelling of a document, or a complex function, such as forecasting the performance of a stock portfolio.
- A Bean may be visible to an end user. For example a button on a graphical user interface.
- A Bean may also be invisible to a user. Software to decode a stream of multimedia information in real time is an example of this type of building block.
- A Bean may also be designed to work autonomously on a user's workstation or to work in cooperation with a set of other distributed components. Software to generate a pie chart from a set of data points is an example of a Bean that can execute locally.
- However, a Bean that provides real-time price information from a stock or commodities exchange would need to work in cooperation with other distributed software to obtain its data

Advantages of Java Beans

- A software component architecture provides standard mechanisms to deal with software building blocks. Some of the specific benefits that Java technology provides for a developer:
- A Bean obtains all the benefits of Java's "write-once, run-anywhere" paradigm.
- The properties, events, and methods of a Bean that are exposed to an application builder tool can be controlled.
- A Bean may be designed to operate correctly in different locales, which makes it useful in global markets.
- Auxiliary software can be provided to help a person configure a Bean. This software is only needed when the design-time parameters for that component are being set. It does not need to be included in the run-time environment.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

JavaBeans vs. Class Libraries

- Beans are appropriate for software components that can be visually manipulated
- Class libraries are good for providing functionality that is useful to programmers, and doesn't benefit from visual manipulation

JavaBean Characteristics

- a public class with 0-argument constuctor
- it has properties with accessory methods
- it has events
- it can customized
- its state can be saved
- it can be analyzed by a builder tool

Application Builder Tools

- Working with Java Beans, most developers use an *application builder tool, a utility* that enables you to configure a set of Beans, connect them together, and produce a working application. Bean builder tools have the following capabilities.
- A palette that lists all of the available Beans. As additional Beans are developed or purchased, they can be added to the palette.
- A worksheet that allows the designer to lay out Beans in a GUI. A designer may drag and drop a Bean from the palette to worksheet.
- Special editors and customizers allow a Bean to be configured.
- Commands allow a designer to inquire about the state and behavior of a Bean. This information automatically becomes available when a Bean is added to the palette.
- Capabilities exist to interconnect Beans. This means that events generated by one component are mapped to method invocations on other components.

Bean Requirements

- Introspection--Exports: properties, methods, events
- Properties--Subset of components internal state
- Methods
  - Invoked to execute component code
- Events (If any needed)
  - Notification of a change in state
  - User activities (typing, mouse actions, …)
- Customization
  - Developer can change appearance
- Persistence
  - Save current state so it can be reloaded

Other properties

- Indexed properties
  - Array value with get and set elements
- Bound properties
  - Triggers event when value changed
- Constrained properties
  - Triggers event when value changes and allows listeners to 'veto' the change

BeanInfo class

- Provides more information using FeatureDescripter objects
- Subclasses:
  - BeanDescripter, PropertyDescripter, IndexedPropertyDescripter, EventSetDescripter, MethodDescripter, ParameterDescripter
- ICON to represent Bean
- Customizer Class (wizard for set up)
- Property Editor references
- List of properties with descriptions
- List of methods with descriptions
- Method to reset properties to defaults

The beanbox

- Primary task is setting property values
- Property editors for common types
  - Set Font
  - Set background/foreground colors
  - Set numeric values
  - Set string values

Using the Bean Developer Kit (BDK)

- The Bean Developer Kit (BDK), available from the JavaSoft site, is a simple example of a tool that enables you to create, configure, and connect a set of Beans. There is also a set of sample Beans with their source code.
- This section provides step-by-step instructions for installing and using this tool. Remember, the BDK is for use with versions of Java 2 prior to 1.4. For Java 2, v1.4 you must use the Bean Builder Tool
- Starting the BDK

1. Change to the directory **c:\bdk\beanbox.**
2. Execute the batch file called **run.bat. This causes the BDK to display the three** windows. ToolBox lists all of the different Beans that have been included with the BDK. BeanBox provides an area to lay out and connec the Beans selected from the ToolBox. Properties provides the ability to configure a selected Bean. You may also see a window called Method Tracer.

Creating a JavaBean

- Requirements for a simple Bean
- Packaging Bean in a JAR file
- Additional Information – BeanInfo
- Defining property editors
- Defining Bean customizers
- Naming Conventions

Bean NON Requirements

- No Bean Superclass
- Visible interface not required
  - 'Invisible' Beans (timer, random number generator, complex calculation)

Creating a Bean

- Usually extends Canvas (New window)
- Can extend Component ('lightweight')
- Needs constructor with no arguments
- Paint() method used to display
- getPreferredSize(), getMinimumSize()
  - For layout manager defaults
- get and set methods for each property

Packaging the Bean

- Create a JAR file (JavaARchive)
  - Patterned after tar utility
- Create 'stub' manifest
  - Name: smith/proj/beans/BeanName.class
  - Java-Bean: True
  - (forward slashes even under Windows!)

Installing the Bean

Beanbox: copy jar file to /jars directory within the BDK directory

Different depending on tool used

Some Naming Conventions

- Beans
  - Class name: any
  - Constructor: no argument or serialized template file
  - Packaging: jar file with Java-Bean: True
- Properties
  - Get and set using property name
  - Property name: message
    - ✓public String getMessage()
    - ✓Public void setMessage(String s)

More Naming Conventions

- Events
  - Event name: Answer
    - ✓Class name: AnswerEvent
    - ✓Listener name: AnswerListener
    - ✓Listener methods:
      - public void methodname(AnswerEvent e)
      - public void addAnswerListener(AnswerListener l)
      - public void removeAnswerListener(… l)

BDK

- ToolBox contains the beans available
- BeanBox window is the form where you visually wire beans together.
- Properties sheet: displays the properties for the Bean currently selected within the BeanBox window.

MyFirstBean

```
import java.awt.*;
import java.io.Serializable;
public class FirstBean extends Canvas implements Serializable {
    public FirstBean() {
      setSize(50,30);
      setBackground(Color.blue);
    }
}
```

First Bean

- Compile: javac FirstBean.java
- Create a manifest file:
- manifest.txt
  - Name: FirstBean.class
  - Java-Bean: True
- Create a jar file:
- jar cfm FirstBean.jar mani.txt FirstBean.class

Using Beans in hand-written app

- Use *Beans.instantiate*

```
Frame f;
f = new Frame("Testing Beans");
try {
 ClassLoader cl = this.getClass().getClassLoader();
 fb =(FirstBean)Beans.instantiate(cl,"FirstBean");
} catch(Exception e) {
  e.printStackTrace();
}
f.add(fb);
```


Abstract Window Toolkit

- AWT" was introduced in JDK 1.1.
- The packages are: java.awt, java.awt.event
- Present in all Java implementations
- Adequate for many applications
- Uses the controls defined by your OS
  - therefore it's "least common denominator"
- Difficult to build an attractive GUI
- import java.awt.*;
  import java.awt.event.*;

Swing

- Same concepts as AWT
- Doesn't work in ancient Java implementations (Java 1.1 and earlier)
- Many more controls, and they are more flexible
  - Some controls, but not all, are a lot more complicated
- Gives a choice of "look and feel" packages
- Much easier to build an attractive GUI
- `import javax.swing.*;`

Swing vs. AWT

- Swing is bigger, slower, and more complicated
  - But not as slow as it used to be
- Swing is more flexible and better looking
- Swing and AWT are *incompatible*--you can use either, but you can't mix them
  - Actually, you can, but it's tricky and not worth doing
- Learning the AWT is a good start on learning Swing
- Many of the most common controls are just renamed
  - AWT: `Button b = new Button ("OK");`
    Swing: `JButton b = new JButton("OK");`

Introduction to Swings

- In Java to build user interfaces one can use AWT classes.
- A supercharged alternative called *Swing*. Swing is a set of classes that provides more powerful and flexible components than are possible with the AWT.
- In addition to the familiar components, such as buttons, check boxes, and labels, Swing supplies several exciting additions, including tabbed panes, scroll panes, trees, and tables. Even familiar components such as buttons have more capabilities in Swing.
- For example, a button may have both an image and a text string associated with it. Also, the image can be changed as the state of the button changes.
- Unlike AWT components, Swing components are not implemented by platform-specific code. Instead, they are written entirely in Java and, therefore, are platform-independent. The term *lightweight* is used to describe such elements.

AbstractButton	Abstract superclass for Swing button
ButtonGroup	Encapsulates a mutually inclusive set of buttons
ImageIcon	Encapsulates an icon
JApplet	The Swing version of applet
JButton	The Swing PushButton class
JCheckBox	The Swing checkbox class
JComboBox	Encapsulates a combo box
JLabel	The Swing version of a Label
JRadioButton	The Swing version of a RadioButton
Jtable	Swing version of a Table

JApplet

- Fundamental to Swing is the **JApplet** class, which extends **Applet**.
- Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not found in **Applet**.
- For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane.
- One difference between **Applet** and **JApplet** is when adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the *content pane* of the **JApplet** object. The content pane can be obtained via the method :
- Container getContentPane( )
- The **add( )** method of **Container** can be used to add a component to a content pane. Its form is:
- void add(*comp*)
- Here, *comp* is the component to be added to the content pane.

Icons

- In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. Two of its constructors are:
- ImageIcon(String *filename*)
- ImageIcon(URL *url*)
- The **ImageIcon** class implements the **Icon** interface that declares the methods:

Method	Description
int getIconHeight( )	Returns the height of the icon in pixels.
int getIconWidth( )	Returns the width of the icon in pixels.
void paintIcon(Component *comp*, Graphics *g*, int *x*, int *y*)	Paints the icon at position *x*, *y* on the graphics context *g*. Additional information about the paint operation can be provided in *comp*.

Labels

- Swing labels are instances of the **JLabel** class, which extends **JComponent**. It can display text and/or an icon. Some of its constructors are shown here:
- JLabel(Icon *i*)
- Label(String *s*)
- JLabel(String *s*, Icon *i*, int *align*)
- The icon and text associated with the label can be read and written by the
- following methods:
- Icon getIcon( )
- String getText( )
- void setIcon(Icon *i*)
- void setText(String *s*)

Text Fields

- The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are:
- JTextField( )
- JTextField(int *cols*)
- JTextField(String *s*, int *cols*)
- JTextField(String *s*)

Buttons

- Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**.
- The following are the methods that control this behavior:
- void setDisabledIcon(Icon *di*)
- void setPressedIcon(Icon *pi*)
- void setSelectedIcon(Icon *si*)
- void setRolloverIcon(Icon *ri*)

The JButton Class

- The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are:
- JButton(Icon *i*)
- JButton(String *s*)
- JButton(String *s*, Icon *i*)

Check Boxes

- The **JCheckBox** class, which provides the functionality of a check box, is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are :
- JCheckBox(Icon *i*)
- JCheckBox(Icon *i*, boolean *state*)
- JCheckBox(String *s*)
- JCheckBox(String *s*, boolean *state*)
- JCheckBox(String *s*, Icon *i*)
- JCheckBox(String *s*, Icon *i*, boolean *state*)

Radio Buttons

- Radio buttons are supported by the **JRadioButton** class, which is a concrete implementation of **AbstractButton**. Its immediate superclass is **JToggleButton**, which provides support for two-state buttons. Some of its constructors are :
- JRadioButton(Icon *i*)
- JRadioButton(Icon *i*, boolean *state*)
- JRadioButton(String *s*)
- JRadioButton(String *s*, boolean *state*)
- JRadioButton(String *s*, Icon *i*)
- JRadioButton(String *s*, Icon *i*, boolean *state*)
- Combo Boxes
- Swing provides a *combo box* (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**. A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry. You can also type your selection into the text field. Two of **JComboBox**'s constructors are :
- JComboBox( ) JComboBox(Vector *v*)

Tabbed Panes

- A *tabbed pane* is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.
- Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. Tabs are defined via the following method:
- void addTab(String *str*, Component *comp*)
- Here, *str* is the title for the tab, and *comp* is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.
- The general procedure to use a tabbed pane in an applet is :

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane.
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

Scroll Panes

- A *scroll pane* is a component that presents a rectangular area in which a component may be viewed. Horizontal and/or vertical scroll bars may be provided if necessary.
- Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**. Some of its constructors are:
- JScrollPane(Component *comp*)
- JScrollPane(int *vsb*, int *hsb*)
- JScrollPane(Component *comp*, int *vsb*, int *hsb*)
- The steps that to use a scroll pane in an applet:

1. Create a **JComponent** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.

Trees

- A *tree* is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **JTree** class, which extends **JComponent**. Some of its constructors are :
- JTree(Hashtable *ht*)
- JTree(Object *obj*[ ])
- JTree(TreeNode *tn*)
- JTree(Vector *v*)
- The steps to use a tree in an applet:

1. Create a **JTree** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

Tables

- A *table* is a component that displays rows and columns of data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent**.
- One of its constructors is:
- JTable(Object *data*[ ][ ], Object *colHeads*[ ])
- Here, *data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.
- Here are the steps for using a table in an applet:

1. Create a **JTable** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the table and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

