Object Oriented Programming (with JAVA)

by
Dr. Partha Roy,
Professor,
Bhilai Institute of Technology, Durg

UNIT- IV

AWT & SWING: Frame, Panel, Dialog, CheckBox, Choice, List, JComboBox, JFrame, JPanel,

JRadioButton, JScrollPane, JTabbedPane, Using Listeners: ActionListener, ContainerListener,

FocusListener, ItemListener, KeyListener, MouseListener, TextListener, WindowListener.
Applets.

JDBC: Type1 to Type4 drivers.

Java Networking: ServerSocket, Socket, RMI.

JFC (Java Foundation Classes)

- JFC is short for Java Foundation Classes, which encompass a group of features for building graphical user interfaces (GUIs) and adding rich graphics functionality and interactivity to Java applications.

AWT and SWING

- •There are two kinds of graphics components in the Java programming language: heavyweight and lightweight. A heavyweight component is associated with its own native screen resource (commonly known as a peer). Components from the **java.awt** package, such as Button and Label, are heavyweight components.
- •A lightweight component has no native screen resource of its own, so it is "lighter." A lightweight component relies on the screen resource from an ancestor in the containment hierarchy, possibly the underlying Frame object. Components from the **javax.swing** package, such as JButton and JLabel, are lightweight components.

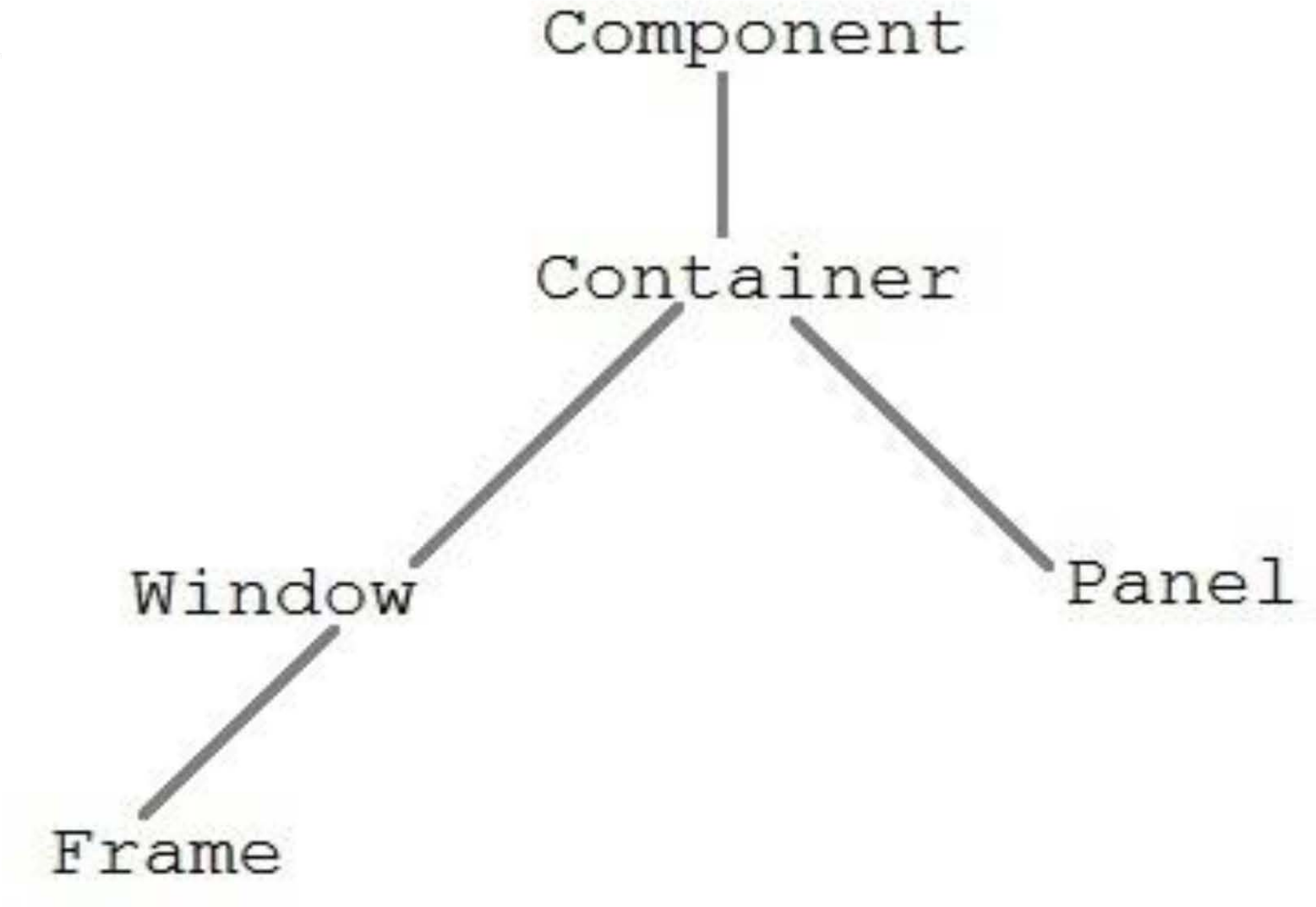
AWT (Abstract Window Toolkit)

- •The AWT contains numerous classes and methods that allow us to create and manage windows.
- •It is also the foundation upon which Swing based applications are built.
- •It contains all classes to write the program that is an interface between the user and GUI objects.
- •We can use the AWT package to develop user interface objects like buttons, checkboxes, radio buttons and menus etc.
- •Although a common use of the AWT is in applets, it is also used to create stand-alone windows that run in a GUI environment, such as Windows.
- •The AWT classes are contained in the **java.awt** package. It is one of Java's largest packages.

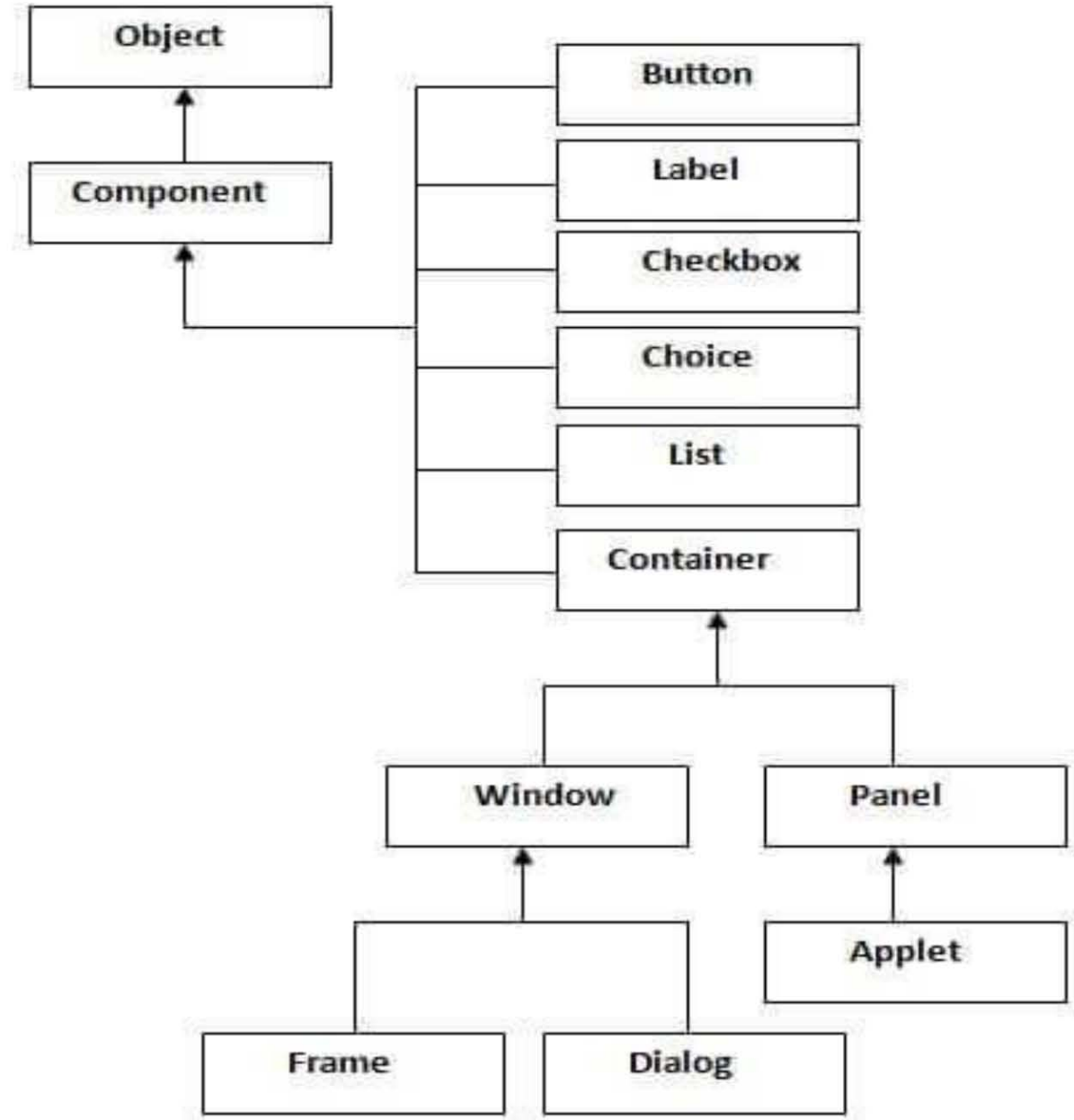
SWING

- •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.

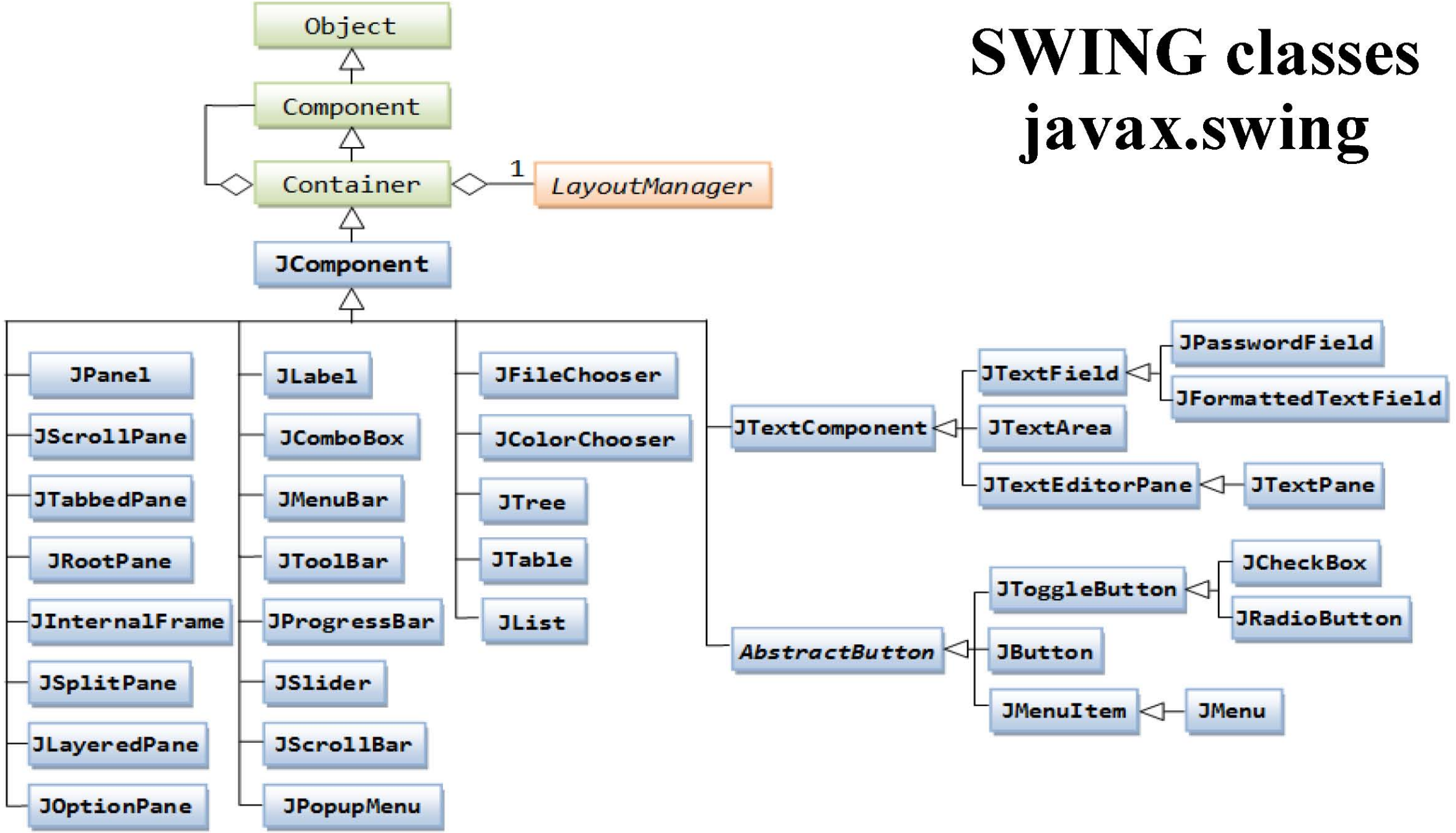
AWT Hierarchy



AWT classes
java.awt



SWING classes
javax.swing



AWT Components

The Frame class

- •**public class Frame extends Window**
- •A Frame is a top-level window with a title and a border.
- •The default layout for a frame is **BorderLayout**.
- •**Constructors:**
- **Frame()**

Constructs a new instance of Frame that is initially invisible.

- **Frame(String title)**

Constructs a new, initially invisible Frame object with the specified title.

The Frame class

- Frame class example code-1 (with icon image)
- Frame class example code-2 (with opacity settings)

The Panel class

- •**public class Panel extends Container**
- •Panel is the simplest container.
- •A panel provides space in which an application can attach any other component, including other panels.
- •The default layout manager for a panel is the **FlowLayout**.
- •**Constructors:**
- **Panel()**

Creates a new panel using the default layout manager.

- **Panel(LayoutManager layout)**

Creates a new panel with the specified layout manager.

Panel class example code

The Dialog class

- •**public class Dialog extends Window**
- •A Dialog is a top-level window with a title and a border that is typically used to take some form of input from the user.
- •The default layout for a dialog is **BorderLayout**.
- •A dialog may have another window as its owner when it's constructed.
- When the owner window of a visible dialog is minimized, the dialog will automatically be hidden from the user.
- When the owner window is subsequently restored, the dialog is made visible to the user again.

The Dialog class

- •**Constructor**:
- **Dialog(Dialog owner)**

Constructs an initially invisible, modeless Dialog with the specified owner Dialog and an empty title.

- **Dialog(Dialog owner, String title)**

Constructs an initially invisible, modeless Dialog with the specified owner Dialog and title.

- **Dialog(Dialog owner, String title, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Dialog, title, and modality.

The Dialog class

- •**Constructor**:
- **Dialog(Frame owner)**

Constructs an initially invisible, modeless Dialog with the specified owner Frame and an empty title.

- **Dialog(Frame owner, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Frame and modality and an empty title.

- **Dialog(Frame owner, String title)**

Constructs an initially invisible, modeless Dialog with the specified owner Frame and title.

- **Dialog(Frame owner, String title, boolean modal)**

Constructs an initially invisible Dialog with the specified owner Frame, title and modality.

Dialog class example code.

The Label class

- •**public class Label extends Component**
- •A Label object is a component for placing text in a container.
- •A Label displays a single line of read-only text. The text can be changed by the application, but a user cannot edit it directly.
- •**Constructors**:
- **Label()**

Constructs an empty label.

- **Label(String text)**

Constructs a new label with the specified string of text, left justified.

- **Label(String text, int alignment)**

Constructs a new label that presents the specified string of text with the specified alignment.

Label class example code

The Button class

- •**public class Button extends Component**
- •This class creates a labelled button. The application can cause some action to happen when the button is pushed.
- •The gesture of clicking on a button with the mouse is associated with one instance of **ActionEvent**, which is sent out when the mouse is both pressed and released over the button.
- •If an application is interested in knowing when the button has been pressed but not released, as a separate gesture, it can specialize **processMouseEvent**, or it can register itself as a listener for mouse events by calling **addMouseListener**. Both of these methods are defined by **Component**, the abstract superclass of all components.

The Button class

- •When a button is pressed and released, AWT sends an instance of **ActionEvent** to the button, by calling **processEvent** on the button. The button's **processEvent** method receives all events for the button; it passes an action event along by calling its own **processActionEvent** method. The latter method passes the action event on to any action listeners that have registered an interest in action events generated by this button.
- •If an application wants to perform some action based on a button being pressed and released, it should implement **ActionListener** and register the new listener to receive events from this button, by calling the button's **addActionListener** method. The application can make use of the button's action command as a messaging protocol.

The Button class

- **Constructors:**
- **Button()**

Constructs a button with an empty string for its label.

- **Button(String label)**

Constructs a button with the specified label.

Button class example code

The Checkbox class

- **•public class Checkbox extends Component**
- •A check box is a graphical component that can be in either an "on" (true) or "off" (false) state.
- Clicking on a check box changes its state from "on" to "off," or from "off" to "on."

The Checkbox class

- •**Constructors**:
- **Checkbox()**

Creates a check box with an empty string for its label.

- **Checkbox(String label)**

Creates a check box with the specified label.

- **Checkbox(String label, boolean state)**

Creates a check box with the specified label and sets the specified state.

- **Checkbox(String label, boolean state, CheckboxGroup group)**

Constructs a Checkbox with the specified label, set to the specified state, and in the specified check box group. Using CheckboxGroup makes the check boxes mutually exclusive.

Checkbox class example code

The TextField class

- •**public class TextField extends TextComponent**
- •A TextField object is a text component that allows for the editing of a single line of text.
- •Every time the user types a key in the text field, one or more key events are sent to the text field.
- A **KeyEvent** may be one of three types: **keyPressed, keyReleased, or keyTyped**.
- •The properties of a key event indicate which of these types it is, as well as additional information about the event, such as what modifiers are applied to the key event and the time at which the event occurred.

The TextField class

- •The key event is passed to every **KeyListener** or **KeyAdapter** object which registered to receive such events using the component's **addKeyListener** method. (**KeyAdapter** objects implement the **KeyListener** interface.)
- •It is also possible to fire an **ActionEvent**. If action events are enabled for the text field, they may be fired by pressing the Return key.
- •The TextField class's processEvent method examines the action event and passes it along to processActionEvent. The latter method redirects the event to any **ActionListener** objects that have registered to receive action events generated by this text field.

The TextField class

- •**Constructors:**
- **TextField**()

Constructs a new text field.

- **TextField(int columns)**

Constructs a new empty text field with the specified number of columns.

- **TextField(String text)**

Constructs a new text field initialized with the specified text.

- **TextField(String text, int columns)**

Constructs a new text field initialized with the specified text to be displayed, and wide enough to hold the specified number of columns.

TextField class example code

The TextArea class

- •**public class TextArea extends TextComponent**
- •A TextArea object is a multi-line region that displays text. It can be set to allow editing or to be read-only.
- •**Constructors:**
- **TextArea()**

Constructs a new text area with the empty string as text.

- **TextArea(int rows, int columns)**

Constructs a new text area with the specified number of rows and columns and the empty string as text.

The TextArea class

- •**Constructors:**
- **TextArea(String text)**

Constructs a new text area with the specified text.

- **TextArea(String text, int rows, int columns)**

Constructs a new text area with the specified text, and with the specified number of rows and columns.

- **TextArea(String text, int rows, int columns, int scrollbars)**

Constructs a new text area with the specified text, and with the rows, columns, and scroll bar visibility as specified.

TextArea class code example

The Choice class

- •**public class Choice extends Component**
- •The Choice class presents a pop-up menu of choices containing String texts as its elements.
- The current choice is displayed as the title of the menu of choices.
- •It is also known as combo-box.
- •**Constructor:**
- **Choice()**

This constructor creates an empty choice list.

The Choice class

- •**Member Methods:**
- **void add(String item)**
- Adds an item to this Choice menu.
- **String getItem(int index)**
- Gets the string at the specified index in this Choice menu.
- **int getItemCount()**
- Returns the number of items in this Choice menu.

The Choice class

- •**Member Methods:**
- **int getSelectedIndex()**
- Returns the index of the currently selected item.
- **void insert(String item, int index)**
- Inserts the item into this choice at the specified position.
- **void remove(int position)**
- Removes an item from the choice menu at the specified position.
- **void remove(String item)**
- Removes the first occurrence of item from the Choice menu.

The Choice class

- •**Member Methods:**
- **void remove(String item)**
- Removes the first occurrence of item from the Choice menu.
- **void removeAll()**
- Removes all items from the choice menu.
- **void select(int pos)**
- Sets the selected item in this Choice menu to be the item at the specified position.
- **void select(String str)**
- Sets the selected item in this Choice menu to be the item whose name is equal to the specified string.
- **void addItemListener(ItemListener l)**
- Adds the specified item listener to receive item events from this Choice menu.

Choice class example code

The List class

- •**public class List extends Component**
- •The List component presents the user with a scrolling list of text items. The list can be set up so that the user can choose either one item or multiple items.
- •**Constructor:**
- **List ()**

This constructor creates an empty List with four visible lines. A List created with this constructor is in single-selection mode, so the user can select only one item at a time.

The List class

- •**Constructor:**
- **List (int rows)**

This constructor creates a List that has specified rows visible lines. This is just a request; the LayoutManager is free to adjust the height of the List to some other amount based upon available space. A List created with this constructor is in single-selection mode, so the user will be able to select only one item at a time.

- **List (int rows, boolean multipleSelections)**

The final constructor for List creates a List that has specified rows visible lines. This is just a request; the LayoutManager is free to adjust the height of the List to some other amount based upon available space. If multipleSelections is true, this List permits multiple items to be selected. If false, this is a single-selection list.

The List class

- **Member Methods:**
- **void add(String item)**

Adds the specified item to the end of scrolling list.

- **void add(String item, int index)**

Adds the specified item to the the scrolling list at the position indicated by the index.

- **String getItem(int index)**

Gets the item associated with the specified index.

The List class

- •**Member Methods:**
- **int getItemCount()**

Gets the number of items in the list.

- **String[] getItems()**

Gets the items in the list.

- **int getSelectedIndex()**

Gets the index of the selected item on the list,

- **int[] getSelectedIndexes()**

Gets the selected indexes on the list.

The List class

- •**Member Methods:**
- **String getSelectedItem()**

Gets the selected item on this scrolling list.

- **String[] getSelectedItems()**

Gets the selected items on this scrolling list.

- **boolean isIndexSelected(int index)**

Determines if the specified item in this scrolling list is selected.

- **void setMultipleMode(boolean b)**

Sets the flag that determines whether this list allows multiple selections.

- **boolean isMultipleMode()**

Determines whether this list allows multiple selections.

The List class

- •**Member Methods:**
- **void remove(int position)**

Removes the item at the specified position from this scrolling list.

- **void remove(String item)**

Removes the first occurrence of an item from the list.

- **void removeAll()**

Removes all items from this list.

- **void replaceItem(String newValue, int index)**

Replaces the item at the specified index in the scrolling list with the new string.

- **void select(int index)**

Selects the item at the specified index in the scrolling list.

List class example code

SWING Components

The JFrame class

- •**public class JFrame extends Frame**
- •Inherits all the methods of **Container** class.
- •The class JFrame is an extended version of **java.awt.Frame** that adds support for the JFC/Swing component architecture.
- •The JFrame class is slightly incompatible with Frame. Like all other JFC/Swing top-level containers, a JFrame contains a JRootPane as its only child.
- The content pane provided by the root pane should, as a rule, contain all the non-menu components displayed by the JFrame.
- This is different from the AWT Frame case.

The JFrame class

- ••**Constructors:**
- **JFrame()**

Constructs a new frame that is initially invisible.

- **JFrame(String title)**

Creates a new, initially invisible Frame with the specified title.

The JFrame class

- **•One of the member Methods:**
- **public void setDefaultCloseOperation(int operation)**

Sets the operation that will happen by default when the user initiates a "close" on this frame. We must specify one of the following choices:

- **WindowConstants.DO_NOTHING_ON_CLOSE** (defined in WindowConstants): Don't do anything; require the program to handle the operation in the windowClosing method of a registered WindowListener object.
- **WindowConstants.HIDE_ON_CLOSE** (defined in WindowConstants): Automatically hide the frame after invoking any registered WindowListener objects.
- **WindowConstants.DISPOSE_ON_CLOSE** (defined in WindowConstants): Automatically hide and dispose the frame after invoking any registered WindowListener objects. If more than one frames are created then only that frame would be disposed that has used this option, the other frame would not close.
- **WindowConstants.EXIT_ON_CLOSE** (defined in JFrame): Exit the application using the System exit method. Use this only in applications. If more than one frames are created then all the frames would be closed irrespective of which frame object has used this option.
- **The value is set to HIDE_ON_CLOSE** by default. Changes to the value of this property cause the firing of a property change event, with property name "defaultCloseOperation".

The JFrame class

- •**This class inherits methods from the following classes:**
- java.lang.Object
- java.awt.Component
- java.awt.Container
- java.awt.Window
- java.awt.Frame

JFrame class code example

The JPanel class

- •**public class JPanel extends JComponent**
- •JPanel is a generic lightweight container
- •**Constructors:**
- **JPanel()**

Creates a new JPanel with a double buffer and a flow layout.

- **JPanel(LayoutManager layout)**

Create a new buffered JPanel with the specified layout manager.

- •**Inherits all the methods from:**
- java.lang.Object
- java.awt.Component
- java.awt.Container
- javax.swing.Jcomponent

JPanel class code example

The JScrollPane class

- •**public class JScrollPane extends JComponent**
- •Provides a scrollable view of a lightweight component. A JScrollPane manages a **viewport**, optional vertical and horizontal scroll bars, and optional row and column heading viewports.
- •Note that JScrollPane does not support heavyweight components.
- •The JViewport provides a window, or "viewport" onto a data source -- for example, an image file. That data source is the "scrollable client" (aka data model) displayed by the JViewport view. A JScrollPane basically consists of JScrollBars, a JViewport, and the wiring between them.

The JScrollPane class

- •**Constructors:**
- **JScrollPane()**

Creates an empty (no viewport view) JScrollPane where both horizontal and vertical scrollbars appear when needed.

- **JScrollPane(Component view)**

Creates a JScrollPane that displays the contents of the specified component, where both horizontal and vertical scrollbars appear whenever the component's contents are larger than the view.

The JScrollPane class

- •**One of the Member Methods:**
- **public void setViewportView(Component view)**

Creates a viewport if necessary and then sets its view. Applications that don't provide the view directly to the JScrollPane constructor should use this method to specify the scrollable child that's going to be displayed in the scrollpane.

For example:

```
JScrollPane scrollpane = new JScrollPane();
scrollpane.setViewportView(myComponent);
```

Note: Applications should not add children directly to the scrollpane.

The JScrollPane class

- JScrollPane class code example-1
- JScrollPane class code example-2

The JTabbedPane class

- •**public class JTabbedPane extends JComponent**
- •A component that lets the user switch between a group of components by clicking on a tab with a given title and/or icon.
- •Tabs/components are added to a TabbedPane object by using the **addTab** and **insertTab** methods.
- A tab is represented by an index corresponding to the position it was added in, where the first tab has an index equal to 0 and the last tab has an index equal to the tab count minus 1.

The JTabbedPane class

- ••One of the Constructors:
- **JTabbedPane()**

Creates an empty TabbedPane with a default tab placement of JTabbedPane.TOP.

The JTabbedPane class

- **Some Member Methods:**
- **public int getSelectedIndex()**

Returns the currently selected index for this tabbedpane. Returns -1 if there is no currently selected tab.

- **public void setSelectedIndex(int index)**

Sets the selected index for this tabbedpane. The index must be a valid tab index or -1, which indicates that no tab should be selected (can also be used when there are no tabs in the tabbedpane). If a -1 value is specified when the tabbedpane contains one or more tabs, then the results will be implementation defined.

The JTabbedPane class

- **Some Member Methods:**
- **public void insertTab(String title, Icon icon,Component component,String tip,int index)**

Inserts a new tab for the given component, at the given index, represented by the given title and/or icon, either of which may be null.

- Parameters:
  - title - the title to be displayed on the tab
  - icon - the icon to be displayed on the tab
  - component - the component to be displayed when this tab is clicked.
  - tip - the tooltip to be displayed for this tab
  - index - the position to insert this new tab (> 0 and <= getTabCount())

The JTabbedPane class

- **Some Member Methods:**
- **public void addTab(String title, Icon icon,Component cp,String tip)**

Adds a component and tip represented by a title and/or icon, either of which can be null. Cover method for insertTab.

- Parameters:
  - title - the title to be displayed in this tab
  - icon - the icon to be displayed in this tab
  - component - the component to be displayed when this tab is clicked
  - tip - the tooltip to be displayed for this tab

The JTabbedPane class

- **Some Member Methods:**
- **public void addTab(String title,Icon icon,Component component)**

Adds a component represented by a title and/or icon, either of which can be null.

- **public void addTab(String title, Component component)**

Adds a component represented by a title and no icon.

- **public Component add(Component component)**

Adds a component with a tab title defaulting to the name of the component which is the result of calling component.getName. Cover method for insertTab.

Overrides: add in class Container

The JTabbedPane class

- **Some Member Methods:**
- **public void removeTabAt(int index)**

Removes the tab at index. After the component associated with index is removed, its visibility is reset to true to ensure it will be visible if added to other containers.

- **public int getTabCount()**

Returns the number of tabs in this tabbedpane.

JTabbedPane code example

The JComboBox class

- **public class JComboBox extends JComponent**
- 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 we make the combo box editable, then the combo box includes an editable field into which the user can type a value.

The JComboBox class

- •**Constructors:**
- **JComboBox()**

Creates a JComboBox with a default data model.

- **JComboBox(E[] items)**

Creates a JComboBox that contains the elements in the specified array.

E is the type of object that need to be present in the combobox.

- **JComboBox(Vector items)**

Creates a JComboBox that contains the elements in the specified Vector.

The JComboBox class

- •**Some Member Methods:**
- **public void setSelectedItem(Object anObject)**

Sets the selected item in the combo box display area to the object in the argument.

If anObject is in the list, the display area shows anObject selected.

If anObject is not in the list and the combo box is uneditable, it will not change the current selection.

For editable combo boxes, the selection will change to anObject.

- **public Object getSelectedItem()**

Returns the current selected item.

The JComboBox class

- •**Some Member Methods:**
- **public void setSelectedIndex(int anIndex)**

Selects the item at index anIndex.

- **public int getSelectedIndex()**

Returns the first item in the list that matches the given item.

The result is not always defined if the JComboBox allows selected items that are not in the list.

Returns -1 if there is no selected item or if the user specified an item which is not in the list.

- **public void insertItemAt(E item,int index)**

Inserts an item into the item list at a given index.

E is the type of object present in the combobox.

The JComboBox class

- **•Some Member Methods:**
- **public void addItem(E item)**

Adds an item to the item list.

E is the type of object to be added in the combobox.

- **public E getItemAt(int index)**

Returns the list item at the specified index.

If index is out of range (less than zero or greater than or equal to size)it will return null.

E is the type of object present in the combobox.

The JComboBox class

- •Some Member Methods:
- **public void removeItem(Object anObject)**

Removes an item from the item list.

- **public void removeItemAt(int anIndex)**

Removes the item at anIndex

- **public void removeAllItems()**

Removes all items from the item list.

JComboBox code example

The JRadioButton class

- **•public class JRadioButton extends JToggleButton**
- •An implementation of a radio button is an item that can be selected or deselected, and which displays its state to the user.
- •Used with a ButtonGroup object to create a group of buttons in which only one button at a time can be selected. (Create a ButtonGroup object and use its add method to include the JRadioButton objects in the group.)
- •Note: The ButtonGroup object is a logical grouping -- not a physical grouping. To create a button panel, we should still create a JPanel or similar container-object and add a Border to it to set it off from surrounding components.

About ButtonGroup class

- **•public class ButtonGroup extends Object**
- **ButtonGroup()**

Create a ButtonGroup instance.

- **void add(AbstractButton) and void remove(AbstractButton)**

Add a button to the group, or remove a button from the group.

- **public ButtonGroup getGroup()**

Get the ButtonGroup, if any, that controls a button. For example:

ButtonGroup group = ((DefaultButtonModel)button.getModel()).getGroup();

- **public ButtonGroup clearSelection()**

Clears the state of selected buttons in the ButtonGroup.

None of the buttons in the ButtonGroup are selected .

The JRadioButton class

- **•One of the Constructors:**
- **JRadioButton()**

Creates an initially unselected radio button with no set text.

- **•Some of the Member Methods:**
- **public boolean isSelected()**

Returns the state of the button. True if the toggle button is selected, false if it's not.

- **public void setSelected(boolean b)**

Sets the state of the button. Note that this method does not trigger an actionEvent.

JRadioButton code example

Using Listeners

What is an Event?

- Change in the state of an object is known as event i.e. event describes the change in state of source.
- Events are generated as result of user interaction with the graphical user interface components.
- For example, clicking on a button, moving the mouse, entering a character through keyboard,selecting an item from list, scrolling the page are the activities that causes an event to happen.
- Events are represented as Objects in Java technology.
- The super class of all event classes is **java.util.EventObject**.

Types of Events

- The events can be broadly classified into two categories:
- **Foreground Events -** Those events which require the direct interaction of user. They are generated as consequences of a person interacting with the graphical components in Graphical User Interface. For example, clicking on a button, moving the mouse, entering a character through keyboard, selecting an item from list, scrolling the page etc.
- **Background Events -** Those events that do not require the interaction of end user are known as background events. Operating system interrupts, hardware or software failure, timer expires, an operation completion are the example of background events.

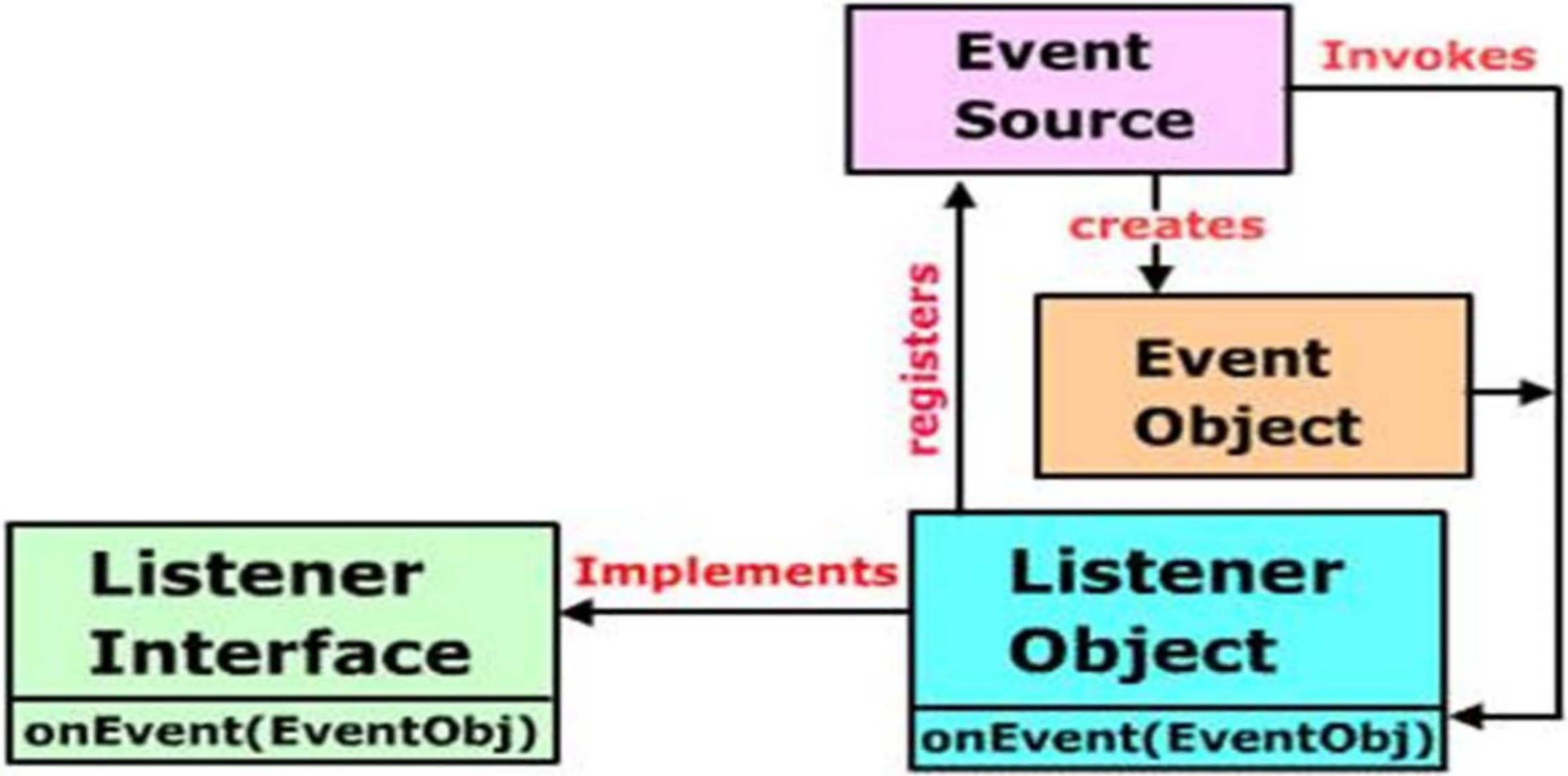
What is Event Handling?

- Event Handling is the mechanism that controls the event and decides what should happen if an event occurs.
- This mechanism have the code which is known as event handler that is executed when an event occurs.
- Java Uses the **Delegation Event Model** to handle the events. This model defines the standard mechanism to generate and handle the events.

The Delegation Event Model

- **The Delegation Event Model has the following key participants namely:**
- **Source -** The source is an object on which event occurs. Source is responsible for providing information of the occurred event to it's handler. Java provide as with classes for source object.
- **Listener -** It is also known as event handler. Listener is responsible for generating response to an event. From java implementation point of view the listener is also an object. Listener waits until it receives an event. Once the event is received , the listener process the event an then returns.
- In this model ,Listener needs to be registered with the source object so that the listener can receive the event notification. This is an efficient way of handling the event because the event notifications are sent only to those listener that want to receive them.

The Delegation Event Model



The Delegation Event Model process

- The event source which is a GUI component has to register itself to a particular listener by calling the method of the listener and it has the format addZZZListener(), where ZZZ should be the name of a specific listener, ex. addActionListener().
- When an event occurs the listener gets notified about the event in the form of an event object.
- This event object contains all the information related to the source from which the event has occured.
- The listener passes the event object to the event handling method for further action.
- We need to override the event handling method in order to generate our desired response to the occurred event.

ActionListener interface

- **public interface ActionListener extends EventListener**
- This interface is used for receiving the action events.
- The class which processes the ActionEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addActionListener() method.
- When the action event occurs, that object's actionPerformed method is invoked.
- **Methods:**
- **void actionPerformed(ActionEvent e)**

Invoked when an action occurs

Action Listener code example

ItemListener interface

- **public interface ItemListener extends EventListener**
- This interface is used for receiving the item events.
- The class which processes the ItemEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addItemListener() method.
- When the action event occurs, that object's itemStateChanged method is invoked.
- **Methods:**
- **void itemStateChanged(ItemEvent e)**

Invoked when an item has been selected or deselected by the user.

ItemListener code example

KeyListener interface

- **public interface KeyListener extends EventListener**
- This interface is used for receiving the key events.
- The class which processes the KeyEvent should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the addKeyListener() method.
- **Methods:**
- **void keyPressed(KeyEvent e)**

Invoked when a key has been pressed.

- **void keyReleased(KeyEvent e)**

Invoked when a key has been released.

- **void keyTyped(KeyEvent e)**

Invoked when a key has been typed.

KeyListener code example

MouseListener interface

- **public interface MouseListener extends EventListener**
- This interface is used for receiving the mouse events.
- The class which processes the **MouseEvent** should implement this interface.
- The object of that class must be registered with a component.
- The object can be registered using the **addMouseListener()** method.

MouseListener interface

- **Methods**:
- **void mouseClicked(MouseEvent e)**

Invoked when the mouse button has been clicked (pressed and released) on a component.

- **void mouseEntered(MouseEvent e)**

Invoked when the mouse enters a component.

- **void mouseExited(MouseEvent e)**

Invoked when the mouse exits a component.

- **void mousePressed(MouseEvent e)**

Invoked when a mouse button has been pressed on a component.

- **void mouseReleased(MouseEvent e)**

Invoked when a mouse button has been released on a component.

MouseListener code example

TextListener interface

- **public interface TextListener extends EventListener**
- Only Applicable to AWT text components not for SWING components.
- This interface is used for receiving the text events.
- The class which processes the **TextEvent** should implement this interface.
- The object of that class must be registered with a component. The object can be registered using the **addTextListener()** method.
- **Methods:**
- **void textValueChanged(TextEvent e)**

Invoked when the value of the text has changed.

TextListener code example

WindowListener interface

- **public interface WindowListener extends EventListener**
- This interface is used for receiving the window events.
- The class which processes the **WindowEvent** should implement this interface.
- The object of that class must be registered with a component. The object can be registered using the **addWindowListener()** method.

WindowListener interface

- **Methods:**
- **void windowActivated(WindowEvent e)**

Invoked when the Window is set to be the active Window.

- **void windowClosed(WindowEvent e)**

Invoked when a window has been closed as the result of calling dispose on the window.

- **void windowClosing(WindowEvent e)**

Invoked when the user attempts to close the window from the window's system menu.

WindowListener interface

- **Methods:**
- **void windowDeactivated(WindowEvent e)**

Invoked when a Window is no longer the active Window.

- **void windowDeiconified(WindowEvent e)**

Invoked when a window is changed from a minimized to a normal state.

- **void windowIconified(WindowEvent e)**

Invoked when a window is changed from a normal to a minimized state.

- **void windowOpened(WindowEvent e)**

Invoked the first time a window is made visible.

WindowListener code example

ContainerListener interface

- **public interface ContainerListener extends EventListener**
- This interface is used for receiving the container events.
- The interface ContainerListener is used for receiving container events. The class that process container events needs to implements this interface.
- **Methods:**
- **void componentAdded(ContainerEvent e)**

Invoked when a component has been added to the container.

- **void componentRemoved(ContainerEvent e)**

Invoked when a component has been removed from the container.

ContainerListener code example

FocusListener interface

- **public interface FocusListener extends EventListener**
- This interface is used for receiving the focus events.
- The interfaceFocusListener is used for receiving keyboard focus events. The class that process focus events needs to implements this interface.
- **Methods:**
- **void focusGained(FocusEvent e)**

Invoked when a component gains the keyboard focus.

- **void focusLost(FocusEvent e)**

Invoked when a component loses the keyboard focus.

FocusListener code example

Applets

Applets

- •A Java applet is a special kind of Java program that a browser enabled with Java technology can download from the internet and run.
- •An applet is typically embedded inside a web page and runs in the context of a browser. An applet must be a subclass of the **java.applet.Applet** class.
- •The Applet class provides the standard interface between the applet and the browser environment.

Applets

- •Swing provides a special subclass of the **Applet** class called **javax.swing.JApplet**.
- •The JApplet class should be used for all applets that use Swing components to construct their graphical user interfaces (GUIs).
- •The browser's Java Plug-in software manages the lifecycle of an applet.
- •We need to use a web server to test the applets.

Applets

- •An applet may be invoked by embedding directives in an HTML file and viewing the file through an applet viewer or Java-enabled browser.
- •The <applet> tag is the basis for embedding an applet in an HTML file.

<applet code="MyApplet.class" width="320" height="120"> </applet>

Applets

- ••If an applet takes parameters, values may be passed for the parameters by adding <param> tags between <applet> and </applet>.

```
<applet code="MyApplet.class" width="480"
height="320">
<param name="color" value="blue">
</applet>
```

Life Cycle of an Applet

- •Four methods in the Applet class give you the framework on which you build any serious applet:
- •**init:** This method is intended for whatever initialization is needed for the applet. It is called after the param tags inside the applet tag have been processed.
- •**start**: This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

Life Cycle of an Applet

- •**stop**: This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.
- •**destroy**: This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, we should not normally leave resources behind after a user leaves the page that contains the applet.
- •**paint**: Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

Life Cycle of an Applet

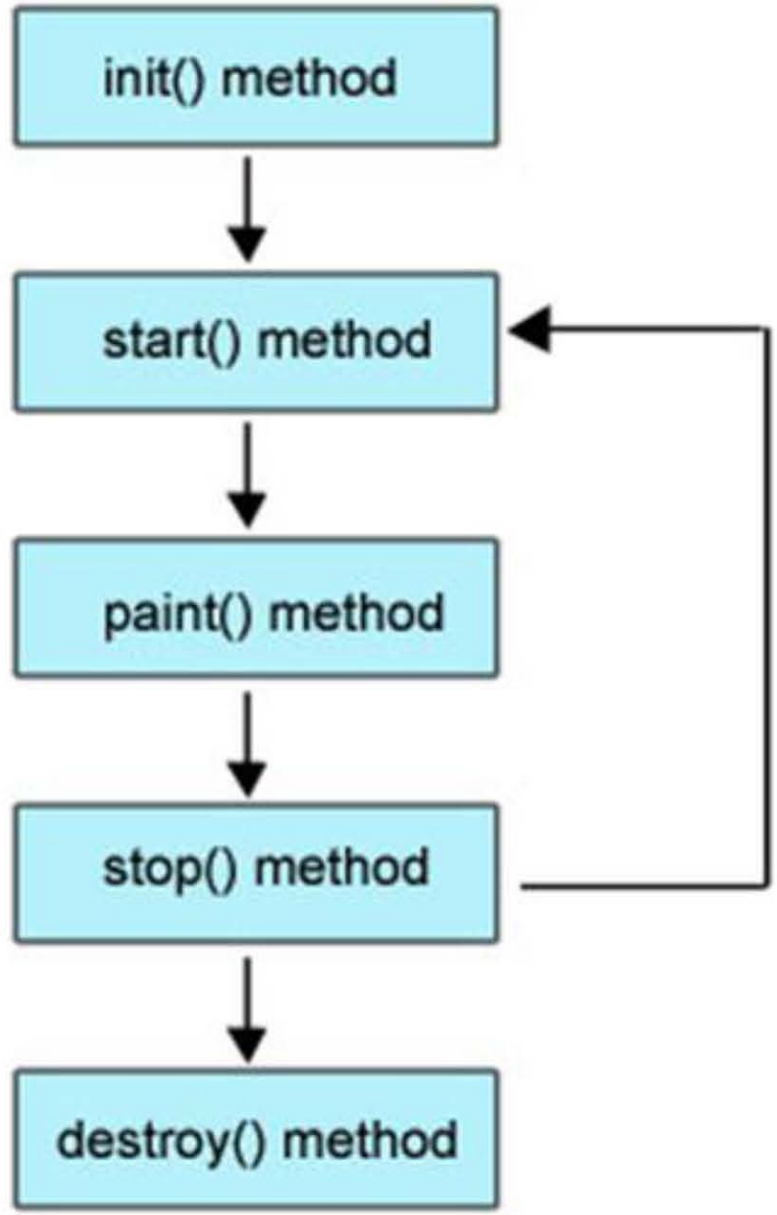


Figure: Life cycle of Applet

The Applet class

- **•public class Applet extends Panel**
- **•Inherits the following classes:**
- java.lang.Object
- java.awt.Component
- java.awt.Container
- java.awt.Panel
- **•Constructor**
- **Applet()**

Constructs a new Applet.

The Applet class

- •**Some of the Member Methods:**
- **public String getParameter(String name)**

Returns the value of the named parameter in the HTML tag. For example, if this applet is specified as

<applet code="Clock" width=50 height=50>

<param name=Color value="blue">

</applet>

then a call to getParameter("Color") returns the value "blue".

The name argument is case insensitive.

The Applet class

- •**Some of the Member Methods:**
- **public void init()**

Called by the browser or applet viewer to inform this applet that it has been loaded into the system.

It is always called before the first time that the start method is called.

A subclass of Applet should override this method if it has initialization to perform.

For example, an applet with threads would use the init() method to create the threads and the destroy method to kill them.

The Applet class

- **•Some of the Member Methods:**
- **public void start()**

Called by the browser or applet viewer to inform this applet that it should start its execution.

It is called after the init() method and each time the applet is revisited in a Web page.

A subclass of Applet should override this method if it has any operation that it wants to perform each time the Web page containing it is visited.

For example, an applet with animation might want to use the start method to resume animation, and the stop method to suspend the animation.

The Applet class

- **•Some of the Member Methods:**
- **public void stop()**

Called by the browser or applet viewer to inform this applet that it should stop its execution.

It is called when the Web page that contains this applet has been replaced by another page, and also just before the applet is to be destroyed.

A subclass of Applet should override this method if it has any operation that it wants to perform each time the Web page containing it is no longer visible.

For example, an applet with animation might want to use the start method to resume animation, and the stop method to suspend the animation.

The Applet class

- •**Some of the Member Methods:**
- **public void destroy()**

Called by the browser or applet viewer to inform this applet that it is being reclaimed and that it should destroy any resources that it has allocated.

The stop method will always be called before destroy.

A subclass of Applet should override this method if it has any operation that it wants to perform before it is destroyed.

For example, an applet with threads would use the init method to create the threads and the destroy method to kill them.

The JApplet class

- •**public class JApplet extends Applet**
- •**Inherits member methods from following classes:**
- □ java.lang.Object
- □ java.awt.Component
- □ java.awt.Container
- □ java.awt.Panel
- □ java.applet.Applet
- •An extended version of java.applet.Applet that adds support for the JFC/Swing component architecture.
- •**Constructor**
- **JApplet()**

Creates a swing applet instance.

Applet code example

JDBC
Java Database Connectivity

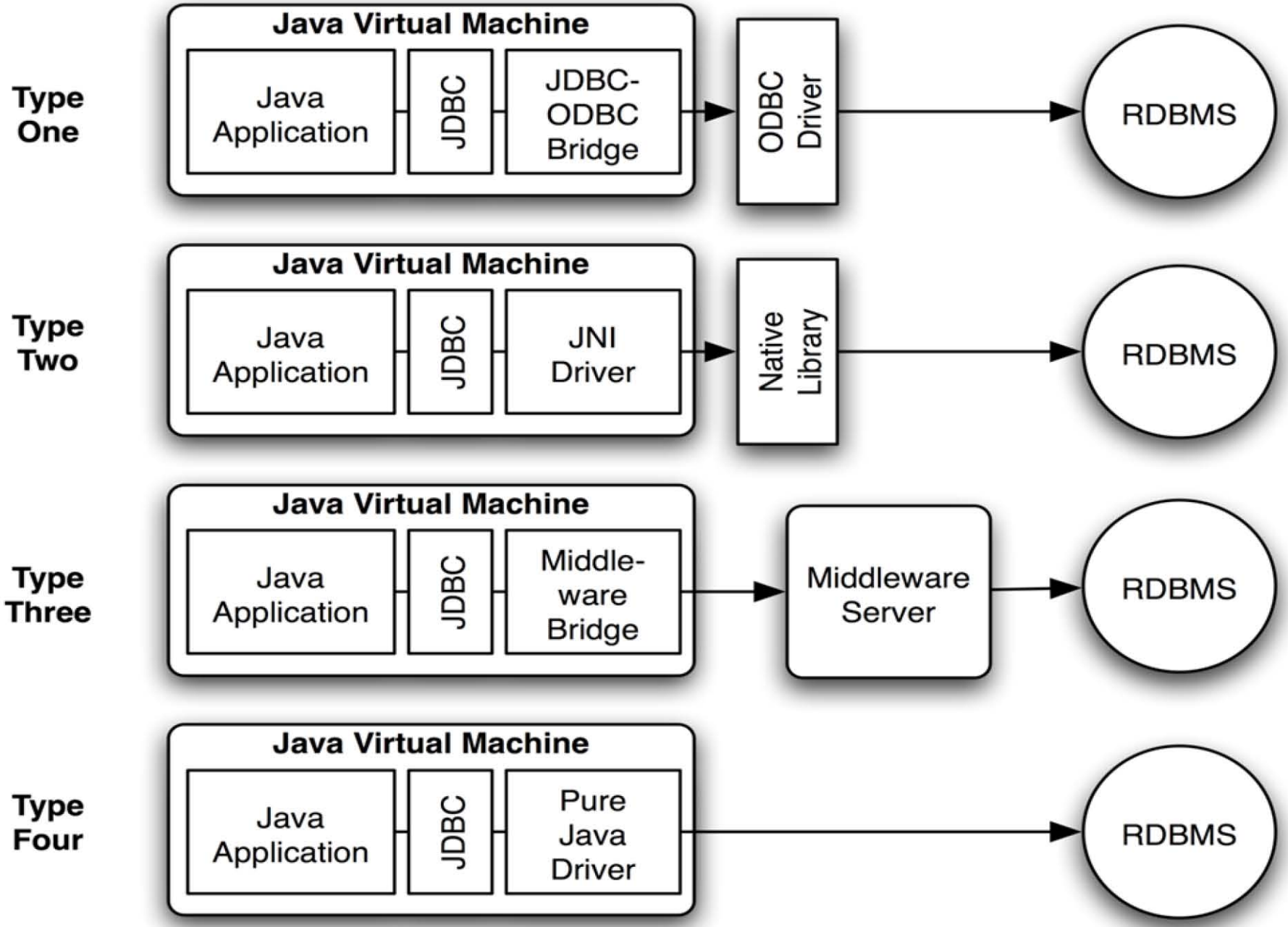
JDBC (Java Database Connectivity)

- •JDBC stands for Java Database Connectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.
- •JDBC API is a Java API that can access any kind of tabular data, especially data stored in a Relational Database.
- •JDBC works with Java on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.
- •The **java.sql** package contains the core JDBC API.
- •It includes basic support for **DriverManager, Connection, Statement and ResultSet**
- •Metadata support for the database and result sets are supported for advanced use.

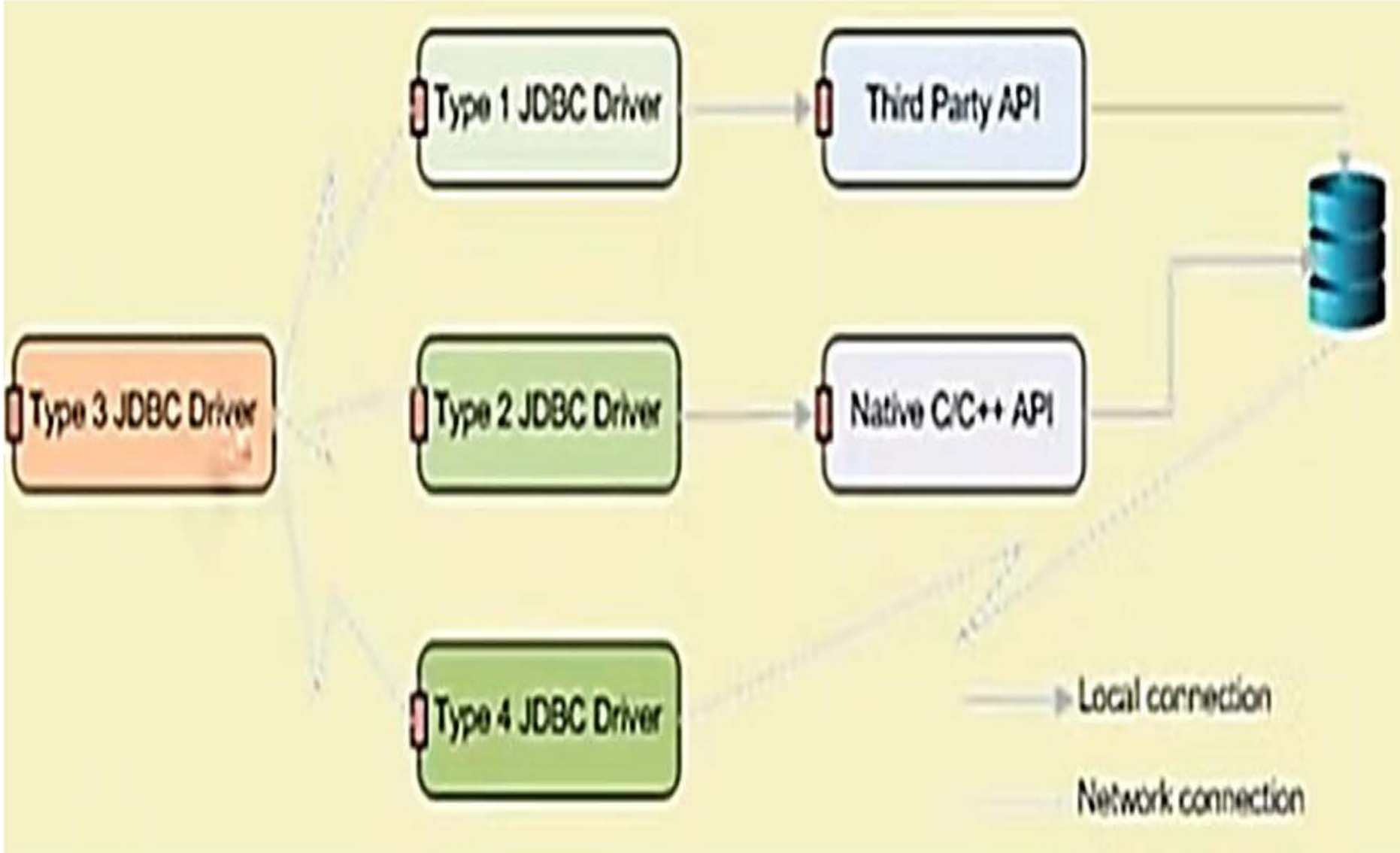
Fundamental Steps in JDBC

- 1. Import JDBC packages.
- 2. Load and register the JDBC driver.
- 3. Open a **connection** to the database.
- 4. Create a **statement** object to perform a query.
- 5. Execute the statement object and return a query resultset.
- 6. Process the **resultset**.
- 7. Close the resultset and statement objects.
- 8. Close the connection.

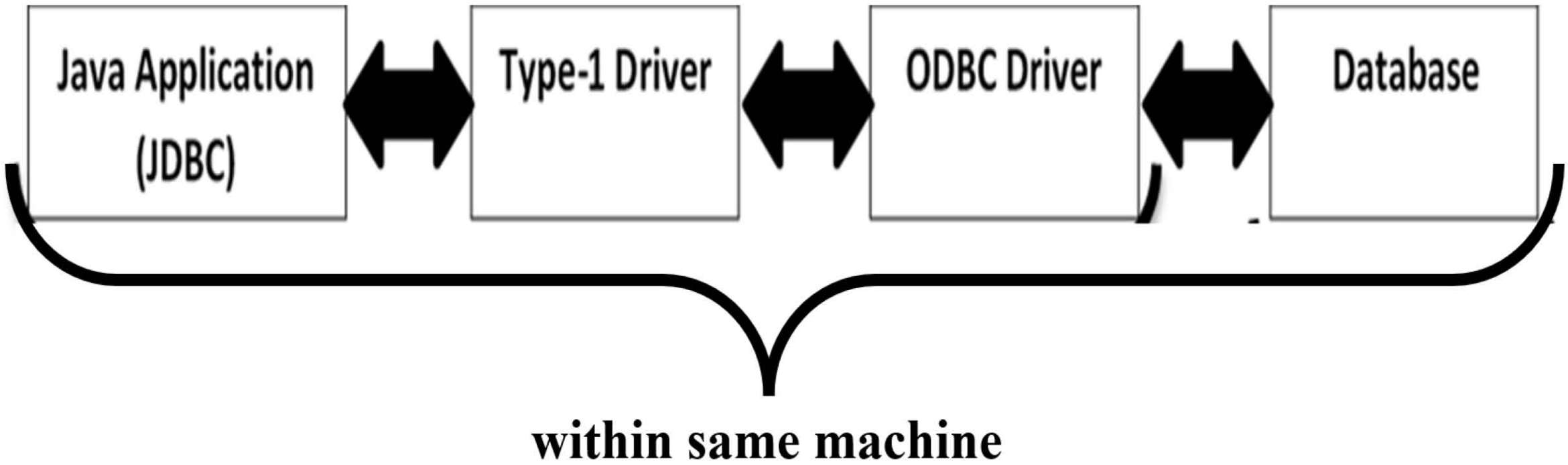
JDBC Drivers



JDBC Drivers



Type1 driver: JDBC-ODBC bridge driver or Bridge driver



Type1 driver: JDBC-ODBC bridge driver or Bridge driver

- •The driver software is provided as part of JDK installation.
- •Only available till JDK-1.7. Not available in JDK-1.8 onwards.
- •Type-1 driver converts the JDBC calls generated by the Java applications into ODBC calls and the ODBC driver converts ODBC calls into database specific calls.
- The ODBC driver is a Third-Party driver (means its not created in Java, but some other language that the manufacturer has selected)
- •Type-1 driver works as bridge between JDBC and ODBC.

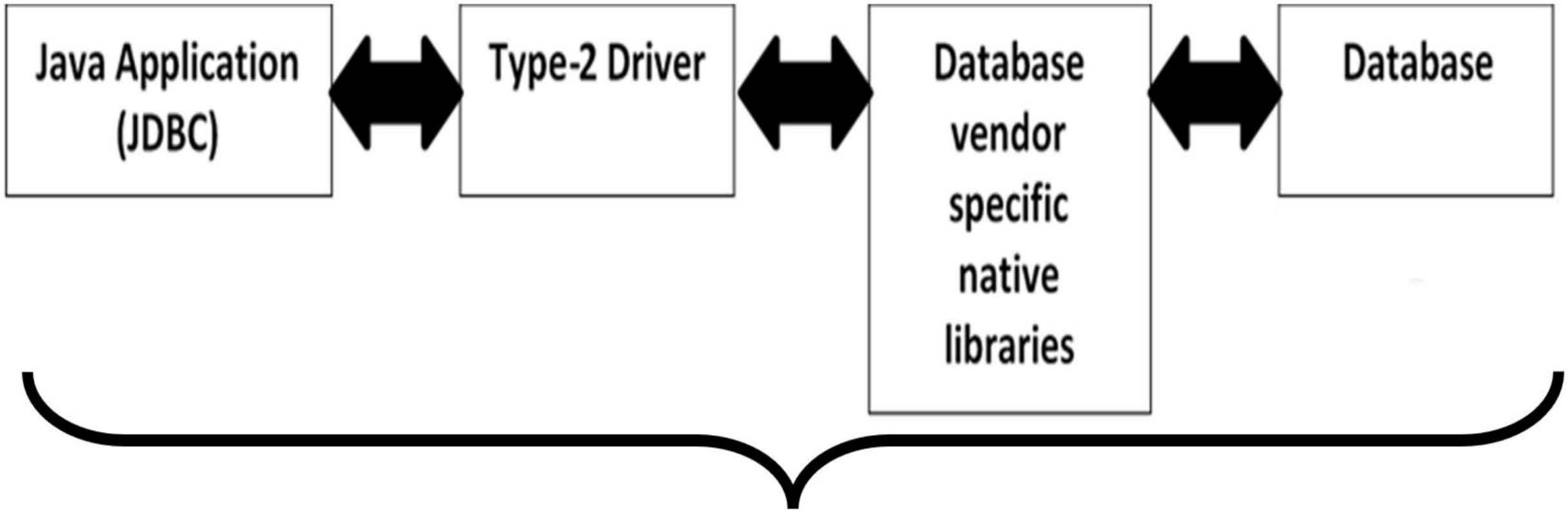
Type1 driver: JDBC-ODBC bridge driver or Bridge driver

- •Helps to connect to any database without any need to change the driver.
- •It is only available for Windows operating system. Other operating systems do not have ODBC facility.
- •The type-1 driver is database independent but not operating system independent.
- •Its operations are considered the slowest among all the drivers.
- **While using Type-1 drivers it is mandatory that the driver and database both remain in the same machine. Remote connections are not supported.**

Type1 driver: JDBC-ODBC bridge driver or Bridge driver

- It is also called thick driver because to communicate with the database additional component is required and pure java driver cannot be used alone.
- Not recommended to use.
- The Bridge is itself a driver based on JDBC technology ("JDBC driver") that is defined in the class **sun.jdbc.odbc.JdbcOdbcDriver.** The Bridge defines the JDBC sub-protocol odbc.

Type2 driver: Partly Java native API driver or Native driver



within the same machine

Type2 driver: Partly Java native API driver or Native driver

- This driver software is NOT provided as part of JDK installation. The database vendor has to provide the driver software.
- The type-2 driver converts the JDBC calls into database vendor specific native library calls which then is understandable by the underlying database software.

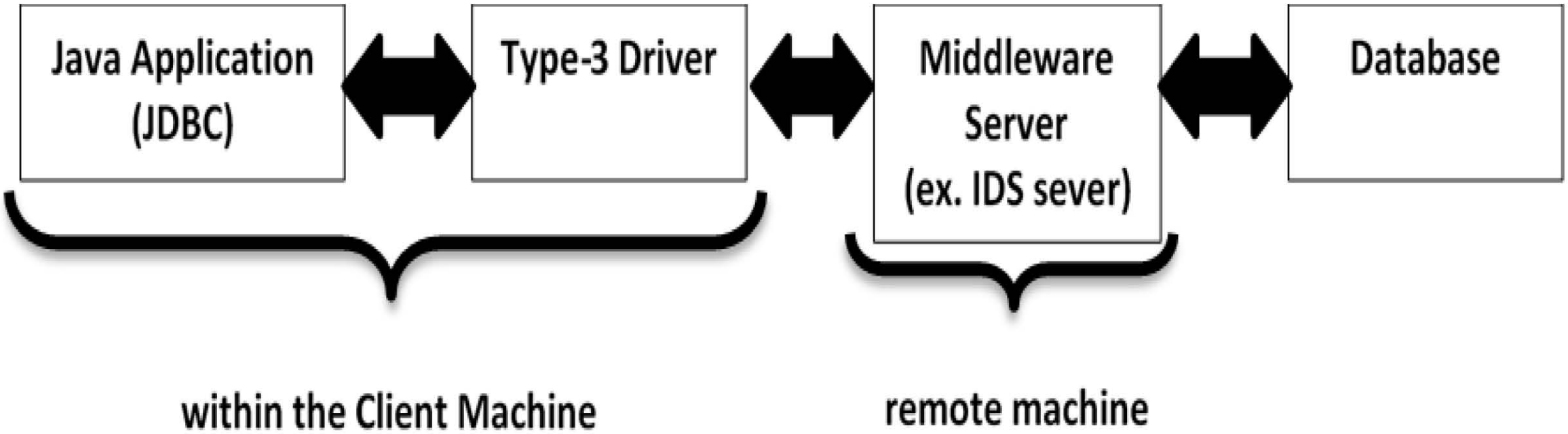
Type2 driver: Partly Java native API driver or Native driver

- •As the database vendor specific native libraries are needed, so if the underlying operating system is changed then the driver software also needs to be changed. Also if the underlying database is changed then again vendor specific driver software is needed.
- •The type-2 driver is neither database independent nor it is operating system independent.
- **While using Type-2 drivers it is mandatory that the driver and database both remain in the same machine. Remote connections are not supported.**

Type2 driver: Partly Java native API driver or Native driver

- •Its operations are considered to be faster than type-1 driver.
- •Not recommended to use.
- JNI/Native Java driver
- Requires DB native library
- Not portable
- Drivers that are written partly in the Java programming language and partly in native code.

Type3 driver: Pure Java Net protocol driver or Network protocol



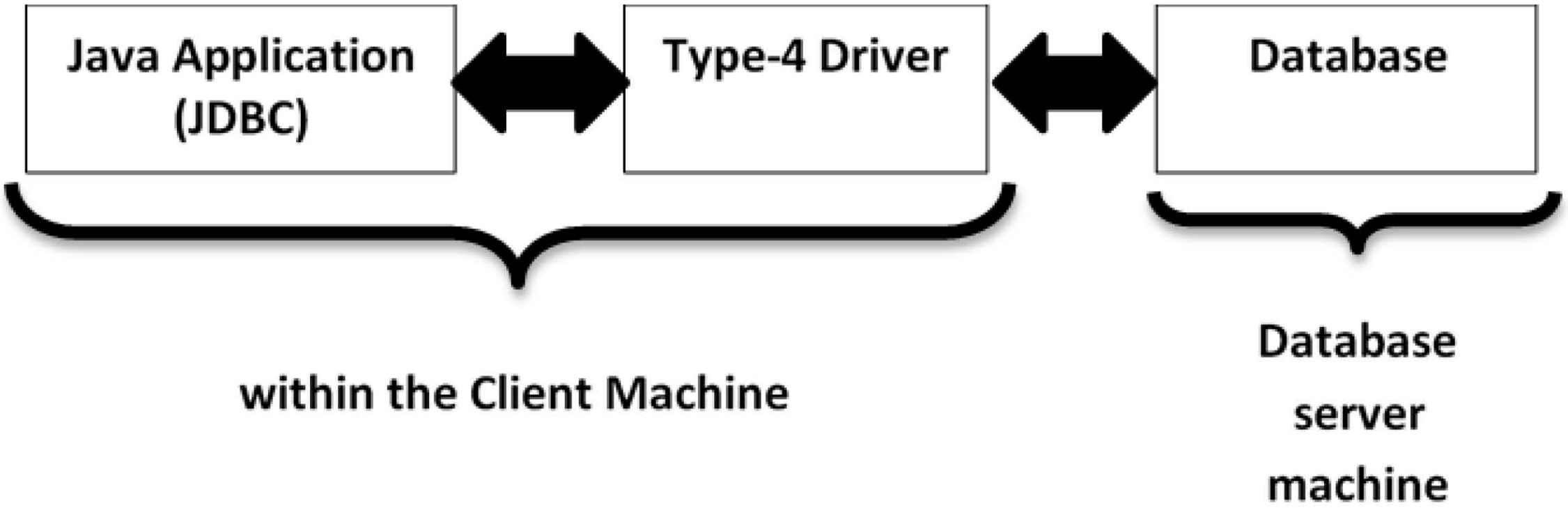
Type3 driver: Pure Java Net protocol driver or Network protocol

- •This driver software is NOT provided as part of JDK installation.
- •Type-3 driver converts the JDBC calls to middleware server specific calls and the middleware server converts the middleware server specific calls to database specific calls.
- •Type-3 driver is platform independent as it is pure java.
- •Type-3 driver is database independent as the middleware server has to adjust according to database but not the type-3 driver.
- •Type-3 driver uses net protocol in Java to communicate with the middleware server machine.

Type3 driver: Pure Java Net protocol driver or Network protocol

- •Examples of middleware server is IDS (Internet Database access Server).
- Pure Java to Middleware driver
- Depends on Middleware server
- Driver is portable, but the middleware server might not be.

Type4 driver: Pure java driver or Thin driver



Type4 driver: Pure java driver or Thin driver

- •This driver software is NOT provided as part of JDK installation.
- •It is created purely in Java, hence it is called pure java or thin driver.
- •It is also called thin driver because to communicate with the database no additional component is required and pure java driver is used alone.
- •It uses database specific native protocols to communicate with the database.

Type4 driver: Pure java driver or Thin driver

- •Database specific type-4 driver is required, so it not database independent.
- •As it is pure Java driver so it is platform independent.
- •The JDBC calls are directly converted into database specific calls and vice-versa, which means only one level of conversion is required so the speed is fastest as compared to other drivers.

Type4 driver: Pure java driver or Thin driver

- Oracle Thin Type 4 Driver for Oracle 8i, 9i, and 10g Databases
- The JAR file for the Oracle driver is **ojdbc14.jar**
- Website: http://www.oracle.com/technetwork/database/enterprise-edition/jdbc-10201-088211.html
- MySQL Server Database Type 4 JDBC Driver
- The JAR file for the MySQL driver is **mysql-connector-java-5.1.7-bin.jar**

Type4 driver: Pure java driver or Thin driver

- Website: http://dev.mysql.com/downloads/connector/j/5.0.html.
- The **com.mysql.jdbc.Driver** class is JDBC driver for MySQL database.
- The **oracle.jdbc.driver.OracleDriver** class is JDBC driver for Oracle database.

Type4 driver: Pure java driver or Thin driver

- **For MySQL:**

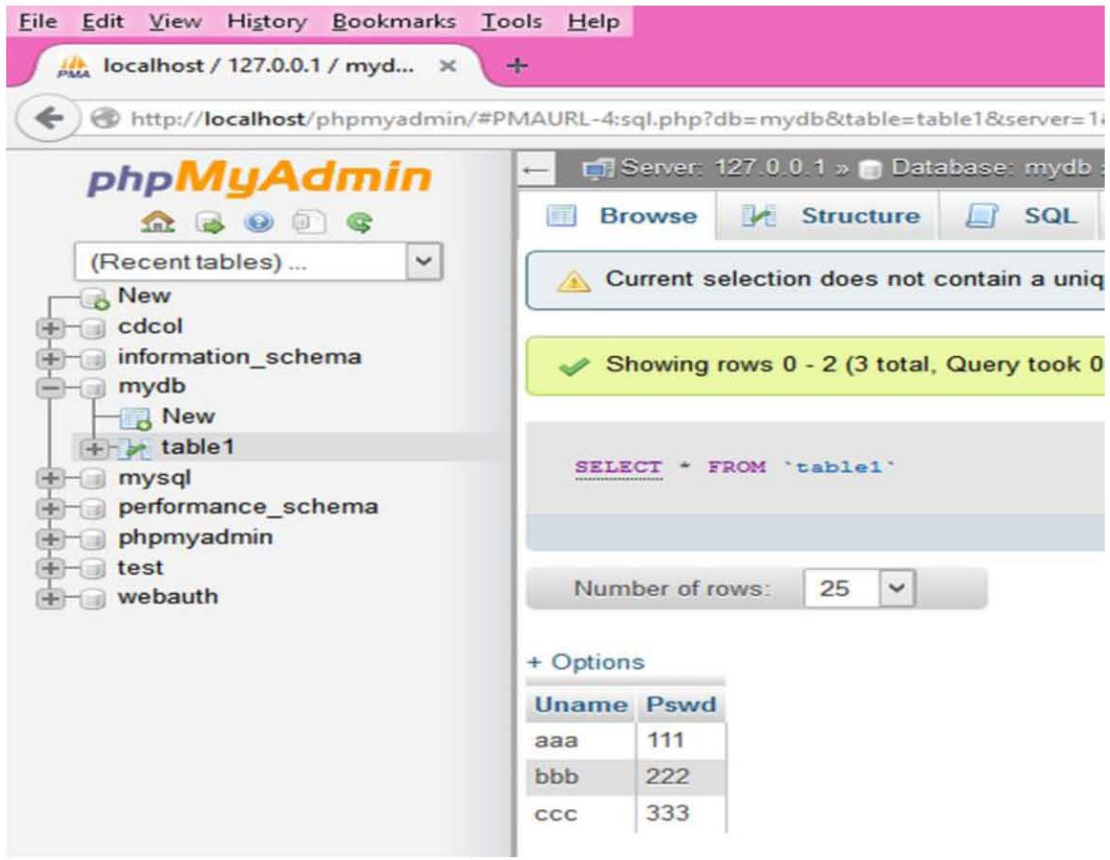
```
Class.forName("com.mysql.jdbc.Driver");
Connection con=DriverManager.getConnection
("jdbc:mysql://hostname/ databaseName","userid","password");
```

- **For ORACLE:**

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection con=DriverManager.getConnection
("jdbc:oracle:thin:@hostname:port
Number:databaseName","userid","password");
```

Type4 driver example

- Code for testing Type4 driver
- SQL commands quick reference.
- **Before running the code there should be a table in MySQL database as follows:**



Java Networking

Dr. Partha Roy, Professor, B.I.T, Durg

Networking in JAVA

- •Computers running on the Internet communicate to each other using either the **Transmission Control Protocol (TCP)** or the **User Datagram Protocol (UDP).**
- •When we write Java programs that communicate over the network, we are programming at the application layer.
- Typically, we don't need to concern ourself with the TCP and UDP layers.
- Instead, we can use the classes in the **java.net** package.
- These classes provide system-independent network communication.

Networking in JAVA

- TCP: TCP stands for Transmission Control Protocol, which allows for reliable communication between two applications. TCP is typically used over the Internet Protocol, which is referred to as TCP/IP.
- When two applications want to communicate to each other reliably, they establish a connection and send data back and forth over that connection. This is analogous to making a telephone call.
- TCP guarantees that data sent from one end of the connection actually gets to the other end and in the same order it was sent. Otherwise, an error is reported.
- TCP provides a point-to-point channel for applications that require reliable communications.

Networking in JAVA

- The **Hypertext Transfer Protocol (HTTP), File Transfer Protocol (FTP), and Telnet** are all examples of applications that require a reliable communication channel. The order in which the data is sent and received over the network is critical to the success of these applications.
- When HTTP is used to read from a URL, the data must be received in the order in which it was sent. Otherwise, we end up with a jumbled HTML file, a corrupt zip file, or some other invalid information.
- The java.net package contains two classes to help us write Java programs that use TCP/IP to send and receive packets over the network: **Socket** and **ServerSocket**. Only one client can connect to a server at a time for communication.

Ports

- A computer has a single physical connection to the network. All data destined for a particular computer arrives through that connection. However, the data may be intended for different applications running on the same computer. So how does the computer know to which application to forward the data? Through the use of **ports**.
- Data transmitted over the Internet is accompanied by addressing information that identifies the computer and the port for which it is destined.
- The computer machine is identified by its **32-bit IP address**, which IP uses to deliver data to the right computer on the network..

Ports

- **Ports** are identified by a **16-bit number**, which TCP and UDP use to deliver the data to the right application in the same computer machine.
- In connection-based communication such as TCP, a server application binds a socket to a specific port number. This has the effect of registering the server with the system to receive all data destined for that port. A client can then communicate with the server at the server's port.
- Port numbers range from **0 to 65,535** because ports are represented by **16-bit numbers.**
- The port numbers ranging from **0 to 1023** are restricted or reserved; they are reserved for use by well-known services such as HTTP and FTP and other system services. These ports are called well-known ports.

URL (Uniform Resource Locator)

- •URL is an acronym for Uniform Resource Locator and is a reference (an address) to a resource on the Internet or Intranet.
- •A URL has two main components:
- Protocol identifier: For the URL http://example.com, the protocol identifier is **http**.
- Resource name: For the URL http://example.com, the resource name is **example.com**.
- •Note that the protocol identifier and the resource name are separated by a colon and two forward slashes. The protocol identifier indicates the name of the protocol to be used to fetch the resource. The example uses the Hypertext Transfer Protocol (HTTP), which is typically used to serve up hypertext documents.

URL (Uniform Resource Locator)

- •HTTP is just one of many different protocols used to access different types of resources on the net. Other protocols include File Transfer Protocol (FTP), Gopher, File, and News.
- •The resource name is the complete address to the resource. The format of the resource name depends entirely on the protocol used, but for many protocols, including HTTP, the resource name contains one or more of the following components:
  - **Host Name:** The name of the machine on which the resource lives.
  - **Filename:** The pathname to the file on the machine.
  - **Port Number:** The port number to which to connect (typically optional).
  - **Reference:** A reference to a named anchor within a resource that usually identifies a specific location within a file (typically optional).

The URL class

- **•public final class URL extends Object implements Serializable**
- **•Inheritance hierarchy:**
  - java.lang.Object
  - java.net.URL
- •Class URL represents a Uniform Resource Locator, a pointer to a "resource" on the World Wide Web.
- A resource can be something as simple as a file or a directory, or it can be a reference to a more complicated object, such as a query to a database or to a search engine.

The URL class

- **Constructors:**
- **URL(String s)**

Creates a URL object from the String representation.

- **URL(String protocol, String host, int port, String file)**

Creates a URL object from the specified protocol, host, port number, and file.

- **URL(String protocol, String host, String file)**

Creates a URL from the specified protocol name, host name, and file name.

URL class code example

The URL class

- URL class methods code example
- Reading site contents using URL class object.

TCP Sockets in JAVA

- Package needed is **java.net**.
- TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other.
- To communicate over TCP, a client program and a server program establish a connection to one another.
- Each program binds a socket to its end of the connection.
- To communicate, the client and the server each reads from and writes to the socket bound to the connection.

TCP Sockets in JAVA

- A socket is one end-point of a two-way communication link between two programs running on the network.
- Socket classes are used to represent the connection between a client program and a server program.
- The java.net package provides two classes--**Socket** and **ServerSocket**--that implement the client side of the connection and the server side of the connection, respectively.
- Normally, a server runs on a specific computer and has a socket that is bound to a specific **port number**.
- The server just waits, listening to the socket for a client to make a connection request.

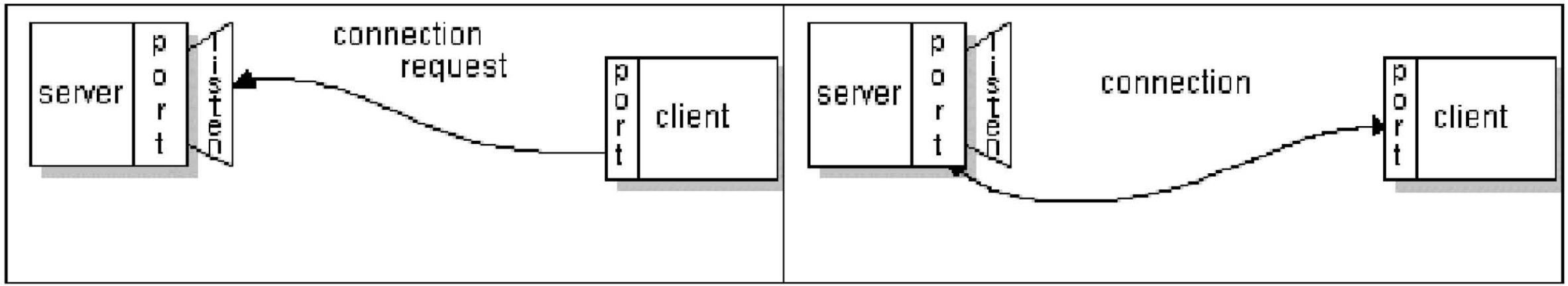
TCP Sockets in JAVA

- **On the client-side:** The client knows the hostname of the machine on which the server is running and the port number on which the server is listening.
- To make a connection request, the client tries to rendezvous with the server on the server's machine and port.
- The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection.
- This is usually assigned by the system.

TCP Sockets in JAVA

- If everything goes well, the server accepts the connection.
- Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client.
- It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.
- On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.
- The client and server can now communicate by writing to or reading from their sockets.

TCP Sockets in JAVA



The Socket class

- **•public class Socket extends Object**
- •The class hierarchy is as follows:
  - java.lang.Object
  - java.net.Socket
- •This class implements client sockets (also called just "sockets").
- A socket is an endpoint for communication between two machines.

The Socket class

- **Constructors:**
- **public Socket()**

Creates an unconnected socket.

- **public Socket(String host, int port) throws UnknownHostException, IOException**

Creates a stream socket and connects it to the specified port number on the named host.

Parameters: host - the host name, or null for the loopback address and port - the port number.

Throws: UnknownHostException - if the IP address of the host could not be determined and IOException - if an I/O error occurs when creating the socket.

The ServerSocket class

- **•public class ServerSocket extends Object**
- •The class hierarchy is as follows:
  - java.lang.Object
  - java.net.ServerSocket
- •This class implements server sockets.
- A server socket waits for requests to come in over the network.
- It performs some operation based on that request, and then possibly returns a result to the requester.

The ServerSocket class

- **•Constructors:**
- **public ServerSocket()throws IOException**

Creates an unbound server socket.

- **public ServerSocket(int port) throws IOException**

Creates a server socket, bound to the specified port.
A port number of 0 means that the port number is automatically allocated.
This port number can then be retrieved by calling **getLocalPort().**

Socket and ServerSocket code examples

- **Server side ServerSocket code.**

- **Client side Socket code.**

RMI
(Remote Method Invocation)

RMI (Remote Method Invocation)

- The base package used is **java.rmi.**
- RMI applications often comprise two separate programs, a server and a client.
- A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
- A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI (Remote Method Invocation)

- •RMI provides the mechanism by which the server and the client communicate and pass information back and forth.
- Such an application is sometimes referred to as a distributed object application.

RMI (Remote Method Invocation)

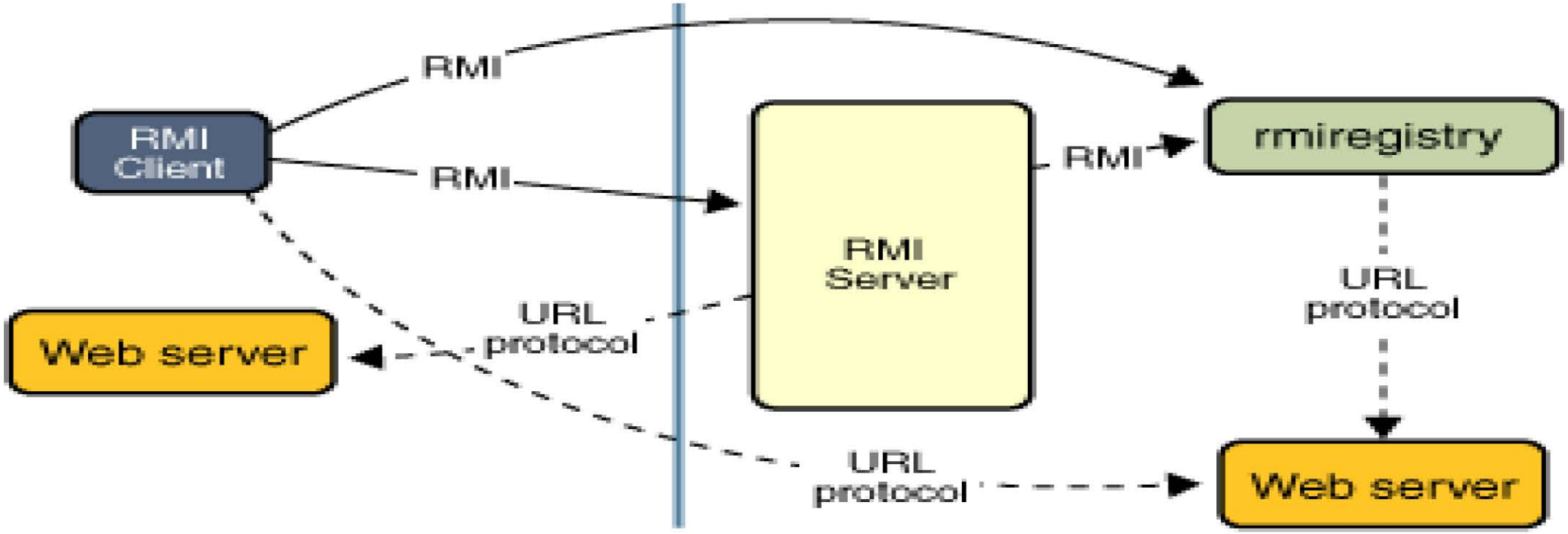
- **•Distributed object applications need to do the following:**
- **1) Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects.
- For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry.
- Alternatively, an application can pass and return remote object references as part of other remote invocations.

RMI (Remote Method Invocation)

- **•Distributed object applications need to do the following:**
- **2) Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **3) Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

RMI (Remote Method Invocation)

- •The server calls the registry to associate (or bind) a name with a remote object.
- •The client looks up the remote object by its name in the server's registry and then invokes a method on it.



Remote Interfaces, Objects, and Methods

- Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes.
- The **interfaces** declare methods.
- The **classes** implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others.
- **Objects** with methods that can be invoked across Java virtual machines are called remote objects.

Remote Interfaces, Objects, and Methods

- **An object becomes remote by implementing the Remote interface**, which has the following characteristics:
  - A remote interface extends the interface **java.rmi.Remote.**
  - Each method of the interface declares **java.rmi.RemoteException** in its throws clause, in addition to any application-specific exceptions.

Remote Interfaces, Objects, and Methods

- RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine.
- Rather than making a copy of the implementation object in the receiving Java virtual machine, **RMI passes a remote stub for a remote object.**

Remote Interfaces, Objects, and Methods

- The **stub** acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference.
- The client invokes a method on the **local stub**, which is responsible for carrying out the method invocation on the **remote object**.

Remote Interfaces, Objects, and Methods

- A **stub for a remote object** implements the same set of remote interfaces that the remote object implements.
- This property enables a stub to be cast to any of the interfaces that the remote object implements.
- However, only those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Steps followed while establishing an RMI application

- **•Defining the remote interfaces.**
- A remote interface specifies the methods that can be invoked remotely by a client.
- Clients program use the remote interfaces, not to the implementation classes of those interfaces.
- The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods.
- If any of these interfaces or classes do not yet exist, we need to define them as well.

Steps followed while establishing an RMI application

- **•Implementing the remote objects.**
- Remote objects must implement one or more remote interfaces.
- The remote object class may include implementations of other interfaces and methods that are available only locally.
- If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- **•Implementing the clients.**
- Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Following steps are required at the SERVER side of an RMI application

- 1. Creating the Interface class containing the methods that the client may invoke.
- 2. Creating the Server class that implements the interface by defining the methods declared in the interface.
- 3. Compiling the Interface to generate the class file.
- 4. Compiling the Server class to generate the class file.
- 5. Generating the stub file from the Server class by using **rmic** command.

Following steps are required at the SERVER side of an RMI application

- 6. The interface class file and the stub class file has to be copied to the client end as both the stubs at client and server ends actually perform the communication.
- 7. Then a **security policy file** needs to be created: The server and client programs run with a security manager installed.
- When we run either program, we need to specify a security policy file so that the code is granted the security permissions it needs to run.

Following steps are required at the SERVER side of an RMI application

- 8. Running the "**rmiregistry**" application.
- The RMI registry is a simple remote object's naming facility that enables remote clients to obtain a reference to remote objects.
- It can be started with the **rmiregistry** command.
- 9. Once the registry is started, we can start the server by using **java –Djava** command with some additional parameters.

Following steps are required at the CLIENT side of an RMI application

- 1. Creating the Client class that looks up for the RMI object by invoking the **lookup method on the registry** to look up the remote object **by name** in the server host's registry.
- 2. Compiling the Client class to generate the class file.
- 3. The **interface class file and the stub class file** from the server has to be copied to the client end as both the stubs at client and server ends actually perform the communication.

Following steps are required at the CLIENT side of an RMI application

- 4. **Then a security policy file needs to be created:** The server and client programs run with a security manager installed. When we run either program, we need to specify a security policy file so that the code is granted the security permissions it needs to run.
- 5. **Running the "rmiregistry" application.** The RMI registry is a simple remote object's naming facility that enables remote clients to obtain a reference to remote objects. It can be started with the **rmiregistry** command.
- 6. Once the registry is started, we can start the client by using **java –Djava** command along with other parameters.

RMI application code example

- Example code for RMI-SERVER
- Example code for RMI-CLIENT

End of UNIT-IV