(2½ hours)

Total Marks: 75

N. B.: (1) **All** questions are **compulsory**.
(2) Make **suitable assumptions** wherever necessary and **state the assumptions** made.
(3) Answers to the **same question** must be **written together**.
(4) Numbers to the **right** indicate **marks**.
(5) Draw **neat labeled diagrams** wherever **necessary**.
(6) Use of **Non-programmable** calculators is **allowed**.

1 **Attempt *any two* of the following:** **10**

a **Explain adapter classes and inner classes**.

Java provides a special feature, called an adapter class that can simplify the creation of event handlers in certain situations. An adapter class provides an empty implementation of all methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.
You can define a new class to act as an event listener by extending one of the adapter classes and implementing only those events in which you are interested.

The adapter classes are found
in **java.awt.event**, **java.awt.dnd** and **javax.swing.event**packages. The Adapter classes with their corresponding listener interfaces are given below.

Adapter class	Listener interface
WindowAdapter	WindowListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
FocusAdapter	FocusListener
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
HierarchyBoundsAdapter	HierarchyBoundsListener

Java inner class or nested class is a class which is declared inside the class or interface.

We use inner classes to logically group classes and interfaces in one place so that it can be more readable and maintainable.

Additionally, it can access all the members of outer class including private data members and methods.

The following program illustrates how to define and use an inner class. The class named **Outer** has one instance variable named **outer_x**, one instance method named **test( )**, and defines one inner class called **Inner**.
// Demonstrate an inner class.

```
class Outer {
int outer_x = 100;
void test() {
Inner inner = new Inner();
inner.display();
}
// this is an inner class
class Inner {
void display() {
System.out.println("display: outer_x = " + outer_x);
}
}
}
class InnerClassDemo {
public static void main(String args[]) {
Outer outer = new Outer();
outer.test();
}
}
```

Output from this application is shown here:

```
display: outer_x = 100
```

In the program, an inner class named **Inner** is defined within the scope of class **Outer**. Therefore, any code in class **Inner** can directly access the variable **outer_x**. An instance method named **display( )** is defined inside **Inner**. This method displays **outer_x** on the standard output stream. The **main( )** method of **InnerClassDemo** creates an instance of class **Outer** and invokes its **test( )** method. That method creates an instance of class **Inner** and the **display( )** method is called.

b **Explain the border layout used in java with the help of program.**

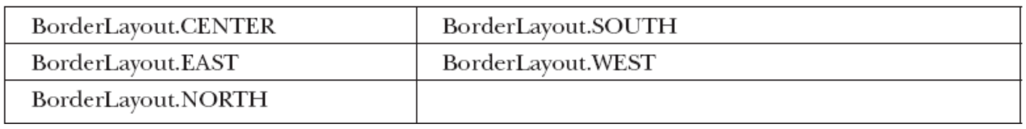
BorderLayout

The **BorderLayout** class implements a common layout style for top-level windows. It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north, south, east, and west. The middle area is called the center.

Here are the constructors defined by **BorderLayout**:

BorderLayout( )
BorderLayout(int *horz*, int *vert*)

The first form creates a default border layout. The second allows you to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively. **BorderLayout** defines the following constants that specify the regions:



BorderLayout.CENTER	BorderLayout.SOUTH
BorderLayout.EAST	BorderLayout.WEST
BorderLayout.NORTH	

When adding components, you will use these constants with the following form of **add( )**, which is defined by **Container**:

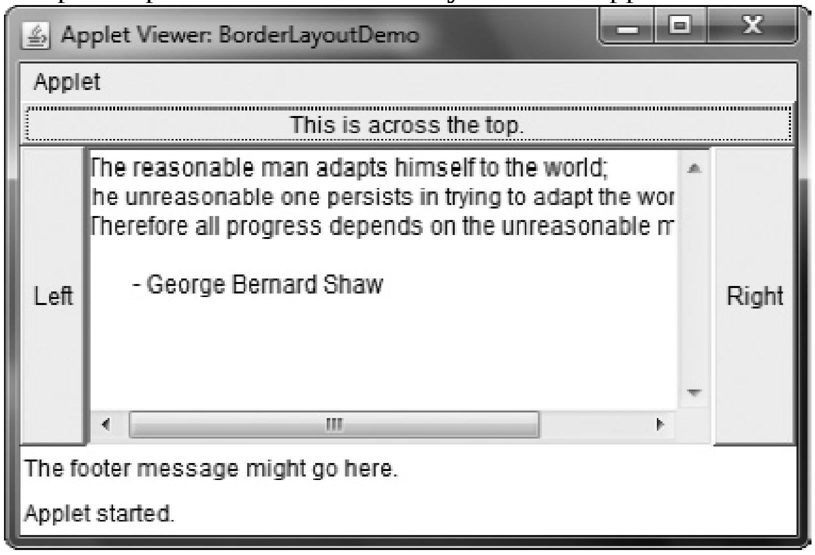
void add(Component *compRef*, Object *region*)

Here, *compRef* is a reference to the component to be added, and *region* specifies where the component will be added.

Here is an example of a **BorderLayout** with a component in each layout area:

```
// Demonstrate BorderLayout.
import java.awt.*;
import java.applet.*;
import java.util.*;
/*
<applet code="BorderLayoutDemo" width=400 height=200>
</applet>
*/
public class BorderLayoutDemo extends Applet {
public void init() {
setLayout(new BorderLayout());
add(new Button("This is across the top."), BorderLayout.NORTH);
add(new Label("The footer message might go here."), BorderLayout.SOUTH);
add(new Button("Right"), BorderLayout.EAST);
add(new Button("Left"), BorderLayout.WEST);
String msg = "The reasonable man adapts " +
"himself to the world;\n" +
"the unreasonable one persists in " +
"trying to adapt the world to himself.\n" +
"Therefore all progress depends " +
"on the unreasonable man.\n\n" +
" - George Bernard Shaw\n\n";
add(new TextArea(msg), BorderLayout.CENTER);
}
}
```

Sample output from the **BorderLayoutDemo** applet is shown here:



```
The reasonable man adapts himself to the world;
he unreasonable one persists in trying to adapt the wor
Therefore all progress depends on the unreasonable m

     - George Bernard Shaw
```

The footer message might go here.

Applet started.

c **<u>How does AWT create Check boxes? Explain with syntax and code specification.</u>**

A *check box* is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each check box that describes what option the box represents. You change the state of a check box by clicking on it. Check boxes can be used individually or as part of a group. Check boxes are objects of the **Checkbox** class.

Checkbox supports these constructors:
Checkbox( ) throws HeadlessException
Checkbox(String *str*) throws HeadlessException
Checkbox(String *str*, boolean *on*) throws HeadlessException
Checkbox(String *str*, boolean *on*, CheckboxGroup *cbGroup*) throws HeadlessException
Checkbox(String *str*, CheckboxGroup *cbGroup*, boolean *on*) throws HeadlessException

The first form creates a check box whose label is initially blank. The state of the check box is unchecked.

The second form creates a check box whose label is specified by *str*. The state of the check box is unchecked.
The third form allows you to set the initial state of the check box. If *on* is **true**, the check box is initially checked; otherwise, it is cleared.
The fourth and fifth forms create a check box whose label is specified by *str* and whose group is specified by *cbGroup*. If this check box is not part of a group, then *cbGroup* must be **null**. The value of *on* determines the initial state of the check box.

To retrieve the current state of a check box, call **getState( )**.
To set its state, call **setState( )**.
You can obtain the current label associated with a check box by calling **getLabel( )**. To set the label, call **setLabel( )**.

These methods are as follows:

```
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)
```

Here, if *on* is **true**, the box is checked. If it is **false**, the box is cleared. The string passed in *str* becomes the new label associated with the invoking check box.

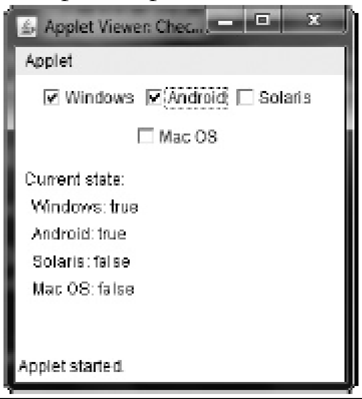
Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This is sent to any listeners that previously registered an interest in receiving item event notifications from that component. Each listener implements the **ItemListener** interface. That interface defines the **itemStateChanged( )** method. An **ItemEvent** object is supplied as the argument to this method. It contains information about the event (for example, whether it was a selection or deselection).

The following program creates four check boxes. The initial state of the first box is checked. The status of each check box is displayed. Each time you change the state of a check box, the status display is updated.

```
// Demonstrate check boxes.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CheckboxDemo" width=240 height=200>
</applet>
*/
public class CheckboxDemo extends Applet implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
public void init() {
windows = new Checkbox("Windows", null, true);
android = new Checkbox("Android");
solaris = new Checkbox("Solaris");
mac = new Checkbox("Mac OS");
add(windows);
add(android);
add(solaris);
add(mac);
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current state: ";
g.drawString(msg, 6, 80);
msg = " Windows: " + windows.getState();
g.drawString(msg, 6, 100);
msg = " Android: " + android.getState();
g.drawString(msg, 6, 120);
msg = " Solaris: " + solaris.getState();
g.drawString(msg, 6, 140);
msg = " Mac OS: " + mac.getState();
g.drawString(msg, 6, 160);
}
}
```

Sample output is shown in below.



```
Current state:
 Windows: true
 Android: true
 Solaris: false
 Mac OS: false

Applet started.
```

d **<u>Write a program demonstrating the use of ItemListener.</u>**

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="CBGroup" width=240 height=200>
</applet>
*/
public class CBGroup extends Applet implements ItemListener {
String msg = "";
Checkbox windows, android, solaris, mac;
CheckboxGroup cbg;
public void init() {
cbg = new CheckboxGroup();
windows = new Checkbox("Windows", cbg, true);
android = new Checkbox("Android", cbg, false);
solaris = new Checkbox("Solaris", cbg, false);
mac = new Checkbox("Mac OS", cbg, false);
add(windows);
add(android);
add(solaris);
add(mac);
windows.addItemListener(this);
android.addItemListener(this);
solaris.addItemListener(this);
mac.addItemListener(this);
}
public void itemStateChanged(ItemEvent ie) {
repaint();
}
// Display current state of the check boxes.
public void paint(Graphics g) {
msg = "Current selection: ";
msg += cbg.getSelectedCheckbox().getLabel();
g.drawString(msg, 6, 100);
}
}
```

2 Attempt *any two* of the following: 10

a **Explain the JMenuBar, JMenuItem and JMenu component's usage with illustration.**

JMenuBar

As mentioned, JMenuBar is essentially a container for menus. Like all components, it inherits JComponent (which inherits Container and Component). It has only one constructor, which is the default constructor. Therefore, initially the menu bar will be empty, and you will need to populate it with menus prior to use. Each application has one and only one menu bar.

JMenuBar defines several methods, but often you will only need to use one: add( ).
The add( ) method adds a JMenu to the menu bar. It is shown here:

JMenu add(JMenu menu)

Here, menu is a JMenu instance that is added to the menu bar. A reference to the menu is returned. Menus are positioned in the bar from left to right, in the order in which they are added. If you want to add a menu at a specific location, then use this version of add( ), which is inherited from Container:

Component add(Component menu, int idx)

Here, menu is added at the index specified by idx. Indexing begins at 0, with 0 being the left-most menu. In some cases, you might want to remove a menu that is no longer needed. You can do
this by calling remove( ), which is inherited from Container. It has these two forms:

void remove(Component menu)
void remove(int idx)

Here, menu is a reference to the menu to remove, and idx is the index of the menu to remove. Indexing begins at zero.
Another method that is sometimes useful is getMenuCount( ), shown here:
int getMenuCount( )
It returns the number of elements contained within the menu bar.

JMenu

JMenu encapsulates a menu, which is populated with JMenuItems. As mentioned, it is derived from JMenuItem. This means that one JMenu can be a selection in another JMenu. This enables one menu to be a submenu of another. JMenu defines a number of constructors.

JMenu(String name)
This constructor creates a menu that has the title specified by name. Of course, you don't have to give a menu a name. To create an unnamed menu, you can use the default constructor:
JMenu( )

Other constructors are also supported. In each case, the menu is empty until menu items are added to it.
JMenu defines many methods.

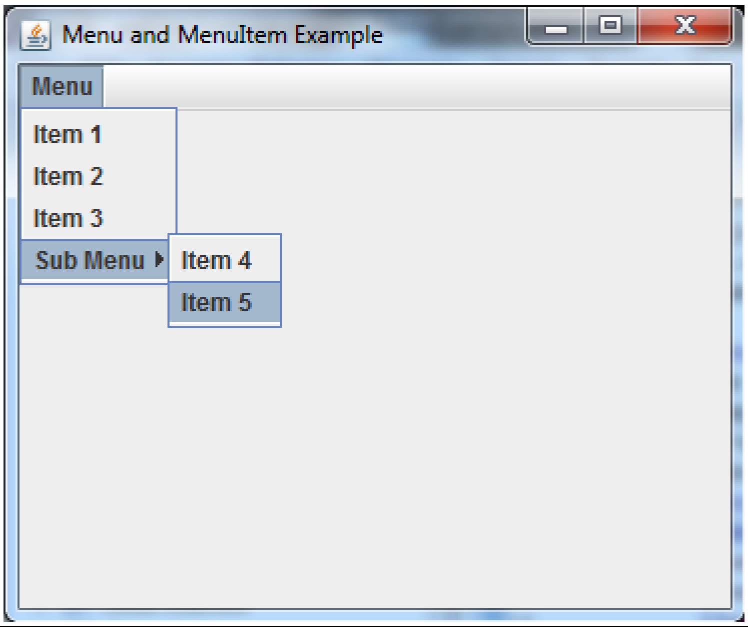
To add an item to the menu, use the add( ) method, which has a number of forms, including the two shown here:
JMenuItem add(JMenuItem item)
JMenuItem add(Component item, int idx)
Here, item is the menu item to add. The first form adds the item to the end of the menu. The second form adds the item at the index specified by idx. As expected, indexing starts at zero. Both forms return a reference to the item added. As a point of interest, you can also use insert( ) to add menu items to a menu.

You can add a separator (an object of type JSeparator) to a menu by calling addSeparator( ), shown here:
void addSeparator( )
The separator is added onto the end of the menu. You can insert a separator into a menu by calling insertSeparator( ), shown next:
void insertSeparator(int idx)
Here, idx specifies the zero-based index at which the separator will be added.
You can remove an item from a menu by calling remove( ). Two of its forms are shown here:
void remove(JMenuItem menu)
void remove(int idx)
In this case, menu is a reference to the item to remove and idx is the index of the item to remove.
You can obtain the number of items in the menu by calling getMenuComponentCount( ), shown here:
int getMenuComponentCount( )
You can get an array of the items in the menu by calling getMenuComponents( ), shown next:
Component[ ] getMenuComponents( )
An array containing the components is returned

JMenuItem

JMenuItem encapsulates an element in a menu. This element can be a selection linked to some program action, such as Save or Close, or it can cause a submenu to be displayed. As mentioned, JMenuItem is derived from AbstractButton, and every item in a menu can be thought of as a special kind of button. Therefore, when a menu item is selected, an action event is generated. (This is similar to the way a JButton fires an action event when it is pressed.) JMenuItem defines many constructors. The ones used in this chapter are shown here:
JMenuItem(String name)
JMenuItem(Icon image)
JMenuItem(String name, Icon image)
JMenuItem(String name, int mnem)
JMenuItem(Action action).

```
import javax.swing.*;
class MenuExample
{
        JMenu menu, submenu;
        JMenuItem i1, i2, i3, i4, i5;
        MenuExample(){
        JFrame f= new JFrame("Menu and MenuItem Example");
        JMenuBar mb=new JMenuBar();
        menu=new JMenu("Menu");
        submenu=new JMenu("Sub Menu");
        i1=new JMenuItem("Item 1");
        i2=new JMenuItem("Item 2");
        i3=new JMenuItem("Item 3");
        i4=new JMenuItem("Item 4");
        i5=new JMenuItem("Item 5");
        menu.add(i1); menu.add(i2); menu.add(i3);
        submenu.add(i4); submenu.add(i5);
        menu.add(submenu);
        mb.add(menu);
        f.setJMenuBar(mb);
        f.setSize(400,400);
        f.setLayout(null);
        f.setVisible(true);
}
public static void main(String args[])
{
new MenuExample();
}}
```



b **Explain the "JProgressBar" component's usage with illustration.**

Java JProgressBar

The JProgressBar class is used to display the progress of the task. It inherits JComponent class.

JProgressBar class declaration

Let's see the declaration for javax.swing.JProgressBar class.

public class JProgressBar extends JComponent implements SwingConstants, Accessible

Commonly used Constructors:

Constructor	Description
JProgressBar()	It is used to create a horizontal progress bar but no string text.

JProgressBar(int min, int max)	It is used to create a horizontal progress bar with the specified minimum and maximum value.
JProgressBar(int orient)	It is used to create a progress bar with the specified orientation, it can be either Vertical or Horizontal by using SwingConstants.VERTICAL and SwingConstants.HORIZONTAL constants.
JProgressBar(int orient, int min, int max)	It is used to create a progress bar with the specified orientation, minimum and maximum value.

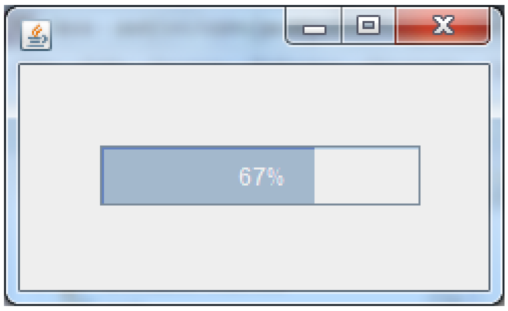
Commonly used Methods:

Method	**Description**
void setStringPainted(boolean b)	It is used to determine whether string should be displayed.
void setString(String s)	It is used to set value to the progress string.
void setOrientation(int orientation)	It is used to set the orientation, it may be either vertical or horizontal by using SwingConstants.VERTICAL and SwingConstants.HORIZONTAL constants.
void setValue(int value)	It is used to set the current value on the progress bar.

Example:

```
import javax.swing.*;
public class ProgressBarExample extends JFrame{
JProgressBar jb;
int i=0,num=0;
ProgressBarExample(){
jb=new JProgressBar(0,2000);
jb.setBounds(40,40,160,30);
jb.setValue(0);
jb.setStringPainted(true);
add(jb);
setSize(250,150);
setLayout(null);
}
public void iterate(){
while(i<=2000){
  jb.setValue(i);
  i=i+20;
  try{Thread.sleep(150);}catch(Exception e){}
}
}
public static void main(String[] args) {
    ProgressBarExample m=new ProgressBarExample();
    m.setVisible(true);
    m.iterate();
}
}
```

OutPut:



c **Explain Root Pane, Glass Pane, Layered Pane, Content Pane and Desktop Pane.**

Swing offers some top-level containers such as - JApplet, JDialog, and JFrame. There are some problems for mixing lightweight and heavyweight components together in Swing, we can't just add anything but first, we must get something called a "content pane," and then we can add Swing components to that.

The Root Pane :
We don't directly create a JRootPane object. As an alternative, we get a JRootPane when we instantiate JInternalFrame or one of the top-level Swing containers, such as JApplet, JDialog, and JFrame. It’s a lightweight container used behind the scenes by these toplevel containers. As the preceding figure shows, a root pane has four parts:

1) **The layered pane:** It Serves to position its contents, which consist of the content pane and the optional menu bar. It can also hold other components in a specified order. JLayeredPane adds depth to a JFC/Swing container, allowing components to overlap each other when needed.It allows for the definition of a several layers within itself for the child components. JLayeredPane manages its list of children like Container, but allows for the definition of a several layers within itself.
2) **The content pane:** The container of the root pane's visible components, excluding the menu bar.
3) **The optional menu bar:** It is the home for the root pane's container's menus. If the container has a menu bar, we generally use the container's setJMenuBar method to put the menu bar in the appropriate place.
4) **The glass pane:** It is hidden, by default. If we make the glass pane visible, then it's like a sheet of glass over all the other parts of the root pane. It's completely 9 transparent.The glass pane is useful when we want to be able to catch events or paint over an area that

already contains one or more components. We can display an image over multiple components using the glass pane.

JdesktopPane:

The concept of showing multiple windows inside a large frame is implemented using Desktop pane. If we minimize the application frame, all of its windows are hidden at the same time. In Windows environment, this is called as the multiple document interfaceor MDI. Using it we can resize the internal frames inDesktop pane by dragging the resize corners.To achieve this we have follow these steps:

1. We can use a regular JFrame window for the program.
2. Set the content pane of the JFrame to a JDesktopPane.

d **What is the difference between AWT and Swing?**

No.	Java AWT	Java Swing
1)	AWT components are **platform-dependent**.	Java swing components are **platform-independent**.
2)	AWT components are **heavyweight**.	Swing components are **lightweight**.
3)	AWT **doesn't support pluggable look and feel**.	Swing **supports pluggable look and feel**.
4)	AWT provides **less components** than Swing.	Swing provides **more powerful components** such as tables, lists, scrollpanes, colorchooser, tabbedpane etc.
5)	AWT **doesn't follows MVC**(Model View Controller) where model represents data, view represents presentation and controller acts as an interface between model and view.	Swing **follows MVC**.

3 **Attempt *any two* of the following:** 10

a **What are servlets? What are the advantages of servlet over CGI?**

What are Servlet?

A Java servlet is a server side program that services HTTP requests and return the results as HTTP responses. A good analogy for a servlet is a non-visual applet and runs on a webserver.it has a lifecycle similar to that of an applet and runs inside a JVM at the web server.The javax.Servlet and javax.Servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines lifecycle methods. When implementing a generic service, we can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as doGet() and doPost(), for handling HTTP-specific services.

What are the advantages of servlet over CGI?

1. Servlets are loaded into memory once and run from memory thereafter.
2. Servlets are swapped as a thread, not as a process.
3. Servlets are powerful object oriented abstraction of http.
4. Servlets are portable across multiple web servers and platforms.
5. Servlets are tightly integrated with web server.
6. Servlets run within the secure and reliable scope of JVM
7. Servlets provide direct database access using native and ODBC based Dbdrivers.
8. Being on the server side provide code protection.
9. Servlets are robust, scalable, secure CGI replacement.

b **Write short note on Servlet & ServletConfig by explaining its methods.**

Servlet
A servlet is a small Java program that runs within a Web server. Servlets receive and respond to requests from Web clients, usually across HTTP, the HyperText Transfer Protocol.

To implement this interface, you can write a generic servlet that extends javax.servlet.GenericServlet or an HTTP servlet that extends javax.servlet.http.HttpServlet.

This interface defines methods to initialize a servlet, to service requests, and to remove a servlet from the server. These are known as life-cycle methods and are called in the following sequence:

The servlet is constructed, then initialized with the init method.
Any calls from clients to the service method are handled.
The servlet is taken out of service, then destroyed with the destroy method, then garbage collected and finalized.

In addition to the life-cycle methods, this interface provides the getServletConfig method, which the servlet can use to get any startup information, and the getServletInfo method, which allows the servlet to return basic information about itself, such as author, version, and copyright.

Method Summary	
void	**destroy**() Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.
ServletConfig	**getServletConfig**() Returns a ServletConfig object, which contains initialization and startup parameters for this servlet.
java.lang.String	**getServletInfo**() Returns information about the servlet, such as author, version, and copyright.
void	**init**(ServletConfig config) Called by the servlet container to indicate to a servlet that the servlet is being placed into service.

void	**service**(ServletRequest req, ServletResponse res) Called by the servlet container to allow the servlet to respond to a request.

public interface **ServletConfig**

A servlet configuration object used by a servlet container to pass information to a servlet during initialization.

Method Summary	
java.lang.String	**getInitParameter**(java.lang.String name) Returns a String containing the value of the named initialization parameter, or null if the parameter does not exist.
java.util.Enumeration	**getInitParameterNames**() Returns the names of the servlet's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the servlet has no initialization parameters.
ServletContext	**getServletContext**() Returns a reference to the ServletContext in which the caller is executing.
java.lang.String	**getServletName**() Returns the name of this servlet instance.

c **Explain GenericServlet with its constructors and methods.**

public abstract class GenericServlet
extends java.lang.Object
implements Servlet, ServletConfig, java.io.Serializable

Defines a generic, protocol-independent servlet. To write an HTTP servlet for use on the Web, extend HttpServlet instead.

GenericServlet implements the Servlet and ServletConfig interfaces. GenericServlet may be directly extended by a servlet, although it's more common to extend a protocol-specific subclass such as HttpServlet.

GenericServlet makes writing servlets easier. It provides simple versions of the lifecycle methods init and destroy and of the methods in the ServletConfig interface. GenericServlet also implements the log method, declared in the ServletContext interface.

To write a generic servlet, you need only override the abstract service method.

Constructor Summary
GenericServlet() Does nothing.

Method Summary	
void	**destroy**() Called by the servlet container to indicate to a servlet that the servlet is being taken out of service.

java.lang.String	**getInitParameter**(java.lang.String name) Returns a String containing the value of the named initialization parameter, or null if the parameter does not exist.
java.util.Enumeration	**getInitParameterNames**() Returns the names of the servlet's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the servlet has no initialization parameters.
ServletConfig	**getServletConfig**() Returns this servlet's ServletConfig object.
ServletContext	**getServletContext**() Returns a reference to the ServletContext in which this servlet is running.
java.lang.String	**getServletInfo**() Returns information about the servlet, such as author, version, and copyright.
java.lang.String	**getServletName**() Returns the name of this servlet instance.
void	**init**() A convenience method which can be overridden so that there's no need to call super.init(config).
void	**init**(ServletConfig config) Called by the servlet container to indicate to a servlet that the servlet is being placed into service.
void	**log**(java.lang.String msg) Writes the specified message to a servlet log file, prepended by the servlet's name.
void	**log**(java.lang.String message, java.lang.Throwable t) Writes an explanatory message and a stack trace for a given Throwable exception to the servlet log file, prepended by the servlet's name.
abstract void	**service**(ServletRequest req, ServletResponse res) Called by the servlet container to allow the servlet to respond to a request.

d **Explain the ServletRequest interface of servlet API.**

public interface ServletRequest
Defines an object to provide client request information to a servlet. The servlet container creates a ServletRequest object and passes it as an argument to the servlet's service method.

A ServletRequest object provides data including parameter name and values, attributes, and an input stream. Interfaces that extend ServletRequest can provide additional protocol-specific data (for example, HTTP data is provided by HttpServletRequest.

Method Summary	
java.lang.Object	**getAttribute**(java.lang.String name) Returns the value of the named attribute as an Object, or null if no attribute of the given name exists.

java.util.Enumeration	**getAttributeNames**() Returns an Enumeration containing the names of the attributes available to this request.
java.lang.String	**getCharacterEncoding**() Returns the name of the character encoding used in the body of this request.
int	**getContentLength**() Returns the length, in bytes, of the request body and made available by the input stream, or -1 if the length is not known.
java.lang.String	**getContentType**() Returns the MIME type of the body of the request, or null if the type is not known.
ServletInputStream	**getInputStream**() Retrieves the body of the request as binary data using a ServletInputStream.
java.lang.String	**getLocalAddr**() Returns the Internet Protocol (IP) address of the interface on which the request was received.
java.util.Locale	**getLocale**() Returns the preferred Locale that the client will accept content in, based on the Accept-Language header.
java.util.Enumeration	**getLocales**() Returns an Enumeration of Locale objects indicating, in decreasing order starting with the preferred locale, the locales that are acceptable to the client based on the Accept-Language header.
java.lang.String	**getLocalName**() Returns the host name of the Internet Protocol (IP) interface on which the request was received.
int	**getLocalPort**() Returns the Internet Protocol (IP) port number of the interface on which the request was received.
java.lang.String	**getParameter**(java.lang.String name) Returns the value of a request parameter as a String, or null if the parameter does not exist.
java.util.Map	**getParameterMap**() Returns a java.util.Map of the parameters of this request.
java.util.Enumeration	**getParameterNames**() Returns an Enumeration of String objects containing the names of the parameters contained in this request.
java.lang.String[]	**getParameterValues**(java.lang.String name) Returns an array of String objects containing all of the values the given request parameter has, or null if the parameter does not exist.
java.lang.String	**getProtocol**() Returns the name and version of the protocol the request uses in the

	form *protocol/majorVersion.minorVersion*, for example, HTTP/1.1.
java.io.BufferedReader	**getReader**() Retrieves the body of the request as character data using a BufferedReader.
java.lang.String	**getRealPath**(java.lang.String path) **Deprecated.** *As of Version 2.1 of the Java Servlet API, use ServletContext.getRealPath(java.lang.String) instead.*
java.lang.String	**getRemoteAddr**() Returns the Internet Protocol (IP) address of the client or last proxy that sent the request.
java.lang.String	**getRemoteHost**() Returns the fully qualified name of the client or the last proxy that sent the request.
int	**getRemotePort**() Returns the Internet Protocol (IP) source port of the client or last proxy that sent the request.
RequestDispatcher	**getRequestDispatcher**(java.lang.String path) Returns a RequestDispatcher object that acts as a wrapper for the resource located at the given path.
java.lang.String	**getScheme**() Returns the name of the scheme used to make this request, for example, http, https, or ftp.
java.lang.String	**getServerName**() Returns the host name of the server to which the request was sent.
int	**getServerPort**() Returns the port number to which the request was sent.
boolean	**isSecure**() Returns a boolean indicating whether this request was made using a secure channel, such as HTTPS.
void	**removeAttribute**(java.lang.String name) Removes an attribute from this request.
void	**setAttribute**(java.lang.String name, java.lang.Object o) Stores an attribute in this request.
void	**setCharacterEncoding**(java.lang.String env) Overrides the name of the character encoding used in the body of this request.

4 **Attempt *any two* of the following:** **10**

a **List the Implicit objects of JSP. Explain any 4 of them in detail.**

JSP Implicit objects

JSP container makes available implicit objects to be used within scriptlets and expressions, without the programmer first having to first create them. These objects act as

wrappers around underlying java classes and interfaces defined in servlet API. They are as follows –
1. request : represents HttpServletReqeust object used to access client's info.
2. response : represents HttpServletResponse object used to send data to client.
3. out : represents JspWriter object that writes into the output stream.
4. config : represents ServletConfig object used to read initialization parameters.
5. session : represents HttpSession object used to identify a client and associate info with it.
6. page : represents HttpJspPage object. Alternatively, you can use this operator to refer to current page. This object is used in custom tag handler classes.
7. application : represents object of ServletContext which is used to obtain information about servlets that are running.
8. pageContext : represents object of PageContext which is used to access context data for page execution.
9. exception : represents Throwable object to handle uncaught errors and exceptions.

b Write JSP Code that will display factorial of numbers from 1 to 20.

c **What is statement in JDBC? Explain the various kinds of statements that can be created in JDBC**.

There are 3 types of Statements, as given below:

Statement:

It can be used for general-purpose access to the database. It is useful when you are using static SQL statements at runtime.

```
Statement stmt = null;
try {
   stmt = conn.createStatement( );
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   stmt.close();
}
```

PreparedStatement:

It can be used when you plan to use the same SQL statement many times. The PreparedStatement interface accepts input parameters at runtime.

```
PreparedStatement pstmt = null;
try {
   String SQL = "Update Employees SET age = ? WHERE id = ?";
   pstmt = conn.prepareStatement(SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   pstmt.close();
}
```

CallableStatement:

CallableStatement can be used when you want to access database stored procedures.

```
CallableStatement cstmt = null;
try {
   String SQL = "{call getEmpName (?, ?)}";
   cstmt = conn.prepareCall (SQL);
   . . .
}
catch (SQLException e) {
   . . .
}
finally {
   cstmt.close();
}
```

d **What is JDBC driver? Explain the types of JDBC drivers.**

What is JDBC Driver?
JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server.

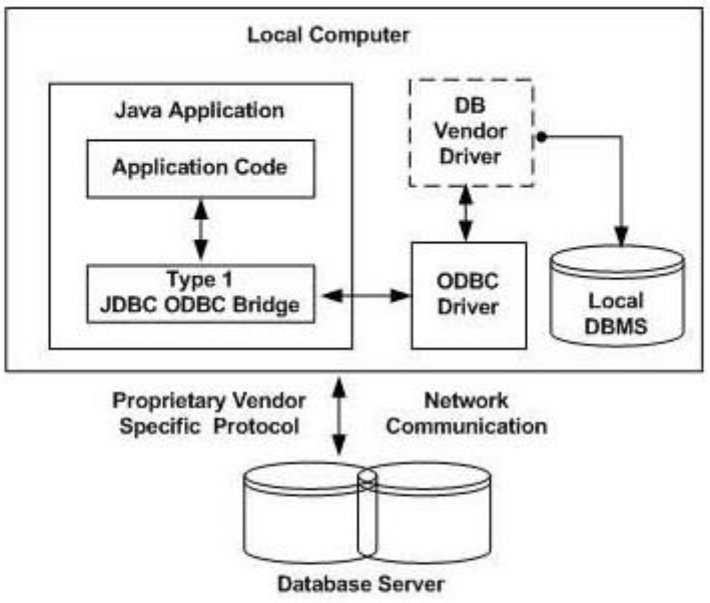
For example, using JDBC drivers enable you to open database connections and to interact with it by sending SQL or database commands then receiving results with Java.

The Java.sql package that ships with JDK, contains various classes with their behaviours defined and their actual implementaions are done in third-party drivers. Third party vendors implements the java.sql.Driver interface in their database driver.

JDBC Drivers Types
JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4, which is explained below −

Type 1: JDBC-ODBC Bridge Driver
In a Type 1 driver, a JDBC bridge is used to access ODBC drivers installed on each client machine. Using ODBC, requires configuring on your system a Data Source Name (DSN) that represents the target database.

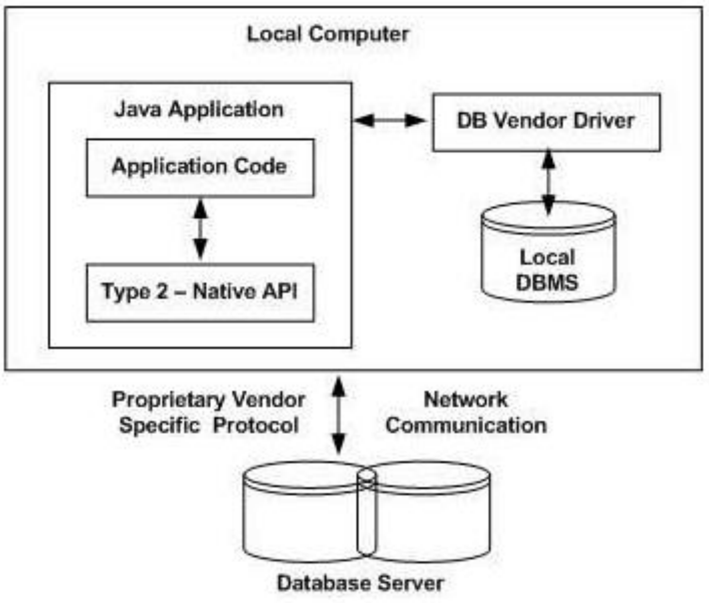
When Java first came out, this was a useful driver because most databases only supported ODBC access but now this type of driver is recommended only for experimental use or when no other alternative is available.



The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

Type 2: JDBC-Native API
In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.

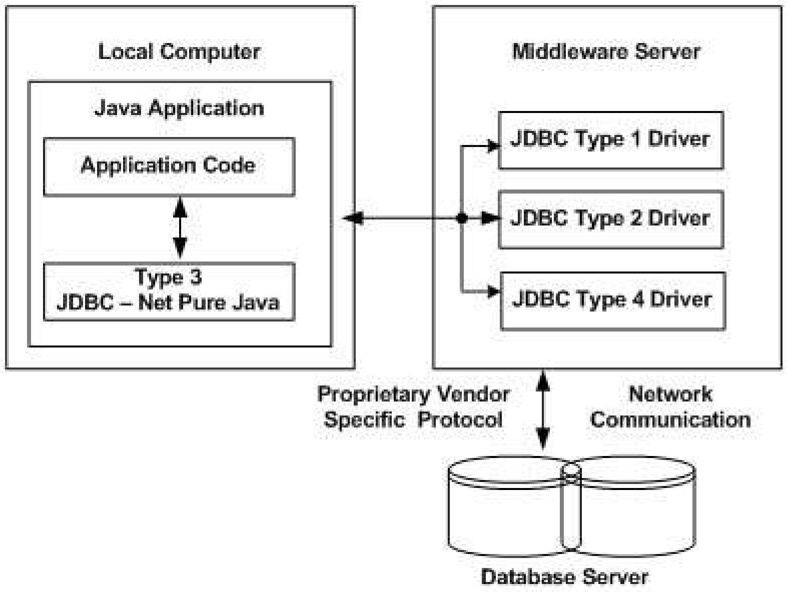
If we change the Database, we have to change the native API, as it is specific to a database and they are mostly obsolete now, but you may realize some speed increase with a Type 2 driver, because it eliminates ODBC's overhead.



The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

Type 3: JDBC-Net pure Java
In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS, and forwarded to the database server.

This kind of driver is extremely flexible, since it requires no code installed on the client and a single driver can actually provide access to multiple databases.

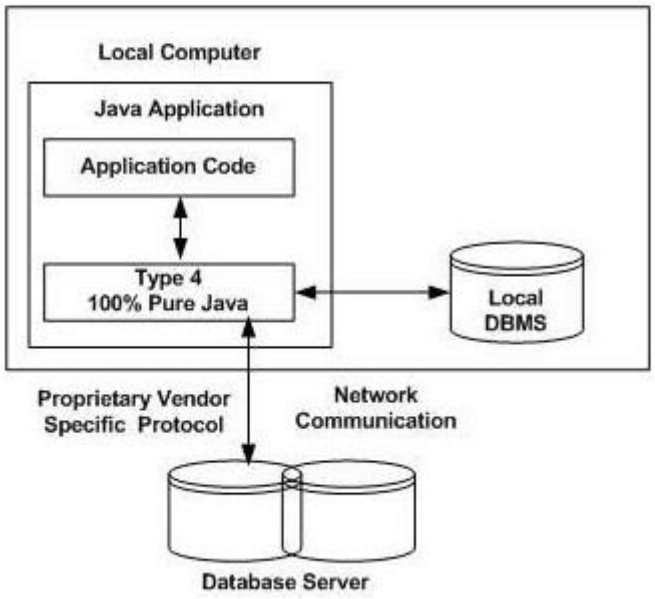


You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

Your application server might use a Type 1, 2, or 4 driver to communicate with the database, understanding the nuances will prove helpful.

Type 4: 100% Pure Java
In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.

This kind of driver is extremely flexible, you don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.



MySQL's Connector/J driver is a Type 4 driver. Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

Which Driver should be Used?

If you are accessing one type of database, such as Oracle, Sybase, or IBM, the preferred driver type is 4.

If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.

Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.

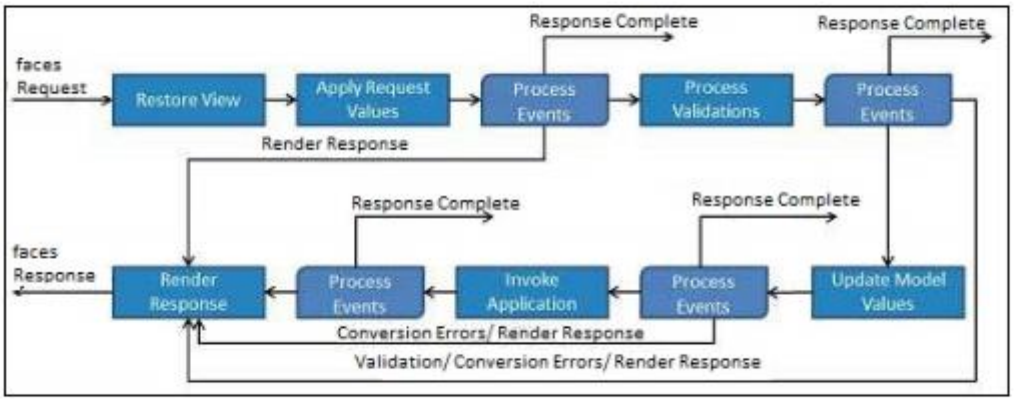
The type 1 driver is not considered a deployment-level driver, and is typically used for development and testing purposes only.

5 **Attempt *<u>any two</u>* of the following:** **10**

a **<u>Explain in details phases of JSF Life cycle.</u>**

JSF application life cycle consists of six phases which are as follows −

- Restore view phase
- Apply request values phase; process events
- Process validations phase; process events
- Update model values phase; process events
- Invoke application phase; process events
- Render response phase



The six phases show the order in which JSF processes a form. The list shows the phases in their likely order of execution with event processing at each phase.

Phase 1: Restore view

JSF begins the restore view phase as soon as a link or a button is clicked and JSF receives a request.

During this phase, JSF builds the view, wires event handlers and validators to UI components and saves the view in the FacesContext instance. The FacesContext instance will now contain all the information required to process a request.

Phase 2: Apply request values

After the component tree is created/restored, each component in the component tree uses the decode method to extract its new value from the request parameters. Component stores this value. If the conversion fails, an error message is generated and queued on FacesContext. This message will be displayed during the render response phase, along with any validation errors.

If any decode methods event listeners called renderResponse on the current FacesContext instance, the JSF moves to the render response phase.

Phase 3: Process validation

During this phase, JSF processes all validators registered on the component tree. It examines the component attribute rules for the validation and compares these rules to the local value stored for the component.

If the local value is invalid, JSF adds an error message to the FacesContext instance, and the life cycle advances to the render response phase and displays the same page again with the error message.

Phase 4: Update model values

After the JSF checks that the data is valid, it walks over the component tree and sets the corresponding server-side object properties to the components' local values. JSF will update the bean properties corresponding to the input component's value attribute.

If any updateModels methods called renderResponse on the current FacesContext instance, JSF moves to the render response phase.

Phase 5: Invoke application

During this phase, JSF handles any application-level events, such as submitting a form/linking to another page.

Phase 6: Render response

During this phase, JSF asks container/application server to render the page if the application is using JSP pages. For initial request, the components represented on the page will be added to the component tree as JSP container executes the page. If this is not an initial request, the component tree is already built so components need not be added again. In either case, the components will render themselves as the JSP container/Application server traverses the tags in the page.

After the content of the view is rendered, the response state is saved so that subsequent requests can access it and it is available to the restore view phase.

b **<u>Explain the use and structure of faces-config.xml file.</u>**

Location
WEB-INF/faces-config.xml

Purposes
Give navigation rules
Map return conditions to results pages
Declare beans
Map bean names to bean classes
Inject bean properties
Define properties files
Declare Locales
Register validators and renderers
Register custom components that use pure-Java syntax

General format

```
<?xml version="1.0"?>
<faces-config … version="2.2">
…
</faces-config>
```

Navigation rules

```
<navigation-rule>
<from-view-id>/some-start-page.xhtml</from-view-id>
<navigation-case>
<from-outcome>return-condition-1</from-outcome>
<to-view-id>/result-page-1.xhtml</to-view-id>
</navigation-case>
More navigation-case entries for other conditions
</navigation-rule>
```

Example

```
<?xml version="1.0"?>
<faces-config …>
<navigation-rule>
<from-view-id>/start-page-1.xhtml</from-view-id>
<navigation-case>
<from-outcome>too-short</from-outcome>
<to-view-id>/error-message.xhtml</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>page1</from-outcome>
<to-view-id>/result-page-1.xhtml</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>page2</from-outcome>
<to-view-id>/result-page-2.xhtml</to-view-id>
</navigation-case>
<navigation-case>
<from-outcome>page3</from-outcome>
<to-view-id>/result-page-3.xhtml</to-view-id>
</navigation-case>
</navigation-rule>
 </faces-config>
```

c **Explain MVC architecture.**

MVC Architecture

The MVC stands for Model, View, and Controller architecture. The MVC architecture separates the business logic and application data from the presentation data to the user.MVC is design pattern which allows a developer to write their applications in a specific format, following the same directory structure, using the same configuration, allowing making unique chain between the components & documents of the application.

Core parts of MVC architecture.

1) **Model:** The model object only represents the data of an application. The model object knows about all the data that need to be displayed. The model is aware about all the operations that can be applied to transform that object. The model represents enterprise data and the business rules that govern access to and updates of this data. Model is not concern about the presentation data and how that data will be displayed to the browser.

2) **View:** The view represents the presentation of the application. The view object refers to the model. It uses the query methods of the model to obtain the contents and renders it The view is not dependent on the application logic. It remains same if there is any modification in

the business logic. It is the responsibility of the view's to maintain the consistency in its presentation when the model (data or logic) changes.
3) **Controller:** Whenever the user sends a request for something then it always go through the controller. The controller is responsible for intercepting the requests from view and passes it to the model for the appropriate action. After the action has been taken on the data, the controller is responsible for directing the appropriate view to the user. In GUI applications the views and the controllers often work very closely together.

d **<u>What are EJB? Explain its benefits.</u>**

EJB is an acronym for *enterprise java bean*. It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications.

To run EJB application, you need an *application server* (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

a. life cycle management,
b. security,
c. transaction management, and
d. object pooling.

EJB application is deployed on the server, so it is called server side component also.

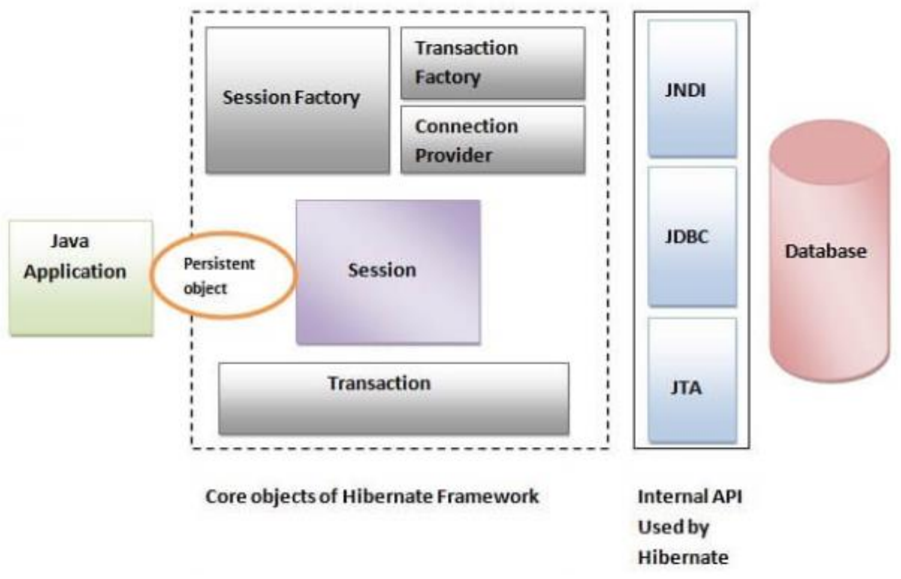
EJB is like COM (*Component Object Model*) provided by Microsoft. But, it is different from Java Bean, RMI and Web Services.

Benefits of EJB:
1. Complete focus only on business logic.
2. Reusable components
3. Portable
4. Fast building of application
5. One business logic having many presentation logic.
6. Distributed deployment.
7. Application interoperability.

6 Attempt *<u>any two</u>* of the following: **10**

a **<u>What is hibernate? Explain its architecture.</u>**

Hibernate is a Java framework that simplifies the development of Java application to interact with the database. It is an open source, lightweight, ORM (Object Relational Mapping) tool. Hibernate implements the specifications of JPA (Java Persistence API) for data persistence.



Hibernate framework uses many objects such as session factory, session, transaction etc. alongwith existing Java API such as JDBC (Java Database Connectivity), JTA (Java Transaction API) and JNDI (Java Naming Directory Interface).

Elements of Hibernate Architecture

For creating the first hibernate application, we must know the elements of Hibernate architecture. They are as follows:

SessionFactory

The SessionFactory is a factory of session and client of ConnectionProvider. It holds second level cache (optional) of data. The org.hibernate.SessionFactory interface provides factory method to get the object of Session.

Session

The session object provides an interface between the application and data stored in the database. It is a short-lived object and wraps the JDBC connection. It is factory of Transaction, Query and Criteria. It holds a first-level cache (mandatory) of data. The org.hibernate.Session interface provides methods to insert, update and delete the object. It also provides factory methods for Transaction, Query and Criteria.

Transaction

The transaction object specifies the atomic unit of work. It is optional. The org.hibernate.Transaction interface provides methods for transaction management.

ConnectionProvider

It is a factory of JDBC connections. It abstracts the application from DriverManager or DataSource. It is optional.

TransactionFactory

It is a factory of Transaction. It is optional.

b **Explain interceptors in struts.**

Interceptors allow developing code that can be run before and/or after the execution of an action. A request is usually processed as follows:

A user requests a resource that maps to an action
The Struts 2 framework invokes the appropriate action to serve the request
If interceptors are written, available and configured, then:
Before the action is executed the invocation could be intercepted by another object
After the action executes, the invocation could be intercepted again by another object
Such objects who intercept the invocation are called **Interceptors.**Conceptually, interceptors are very similar to **Servlet Filters** or the JDKs **Proxy class**.

Interceptor Configuration: Interceptors configured in the struts.xml file appear as:

```
<interceptors>
<interceptor name="test1" class="..."/>
<interceptor name=”test2" class="..."/>
</interceptors>
<action name="WelcomeSturts">
<interceptor-ref name=" test 1"/>
<interceptor-ref name=" test 2"/>
<result name="SUCCESS">/ Welcome.jsp</result>
<result name="ERROR">/ Error.jsp</result>
</action>
```

In above code two interceptors named test1 and test2 are defined. Both of these are them mapped to the action named WelcomeSturts.
Interceptor Stack: We can bind Interceptor together using an Interceptor **Stack** which can be referenced together. We can use the same set of interceptors multiple times. So, instead of configuring a number of interceptors every time, an interceptor stack can be configured with all the required interceptors held within. To use Intercept Stack we need to modify the struts.xml file as follows:

```
<interceptors>
<interceptor name="test1" class="..."/>
<interceptor name= test2" class="..."/>
<interceptor-stack name="MyStack">
<interceptor-ref name=" test1"/>
<interceptor-ref name=" test2"/>
</interceptor-stack>
</interceptors>
<action name="WelcomeSturts " class="test.WelcomeSturts ">
<result name="SUCCESS">/ Welcome.jsp</result>
<result name="ERROR">/ Error.jsp</result>
</action>
```

In above code two interceptors named **test1**and **test2** are defined and a stack named **MyStack**group them both.The stack holding both these interceptors is then mapped to the Action named **WelcomeStruts.**
Execution Flow of Interceptors: Interceptors are executed as follows:
1. The framework receives a request and decides on the action the URL maps to
2. The framework consults the application's configuration file, to discover which interceptors should fire and in what sequence
3. The framework starts the invocation process by executing the first Interceptor in the **Stack**
4. After all the interceptors are invoked, the framework causes the action itself to be **executed.**

c **<u>Explain in detail core components of Struts framework.</u>**

Following are the Important Components of Struts.

1. JSP Programs (View Layer Resources)
2. FormBean Class(Java Class) (Controller Layer Resources)
3. Action Servlet(Built-in Controller Servlet) (Controller Layer Resources)
4. Action Class (Java Class) (Controller Layer Resources)
5. web.xml (Deployment Descriptor file of web application)
6. Struts Configuration File (XML File) (Controller Layer Resources)

JSP Programs :

JSP Program contains presentation logic to generate user interface for end users and to format the result returned by model layer.It is always recommended to develop JSP pages which contains less number of Java codes.To reduce Java Code in JSP application use following JSP Tags.

Struts Tag Libraries
JSTL Tags
JSP Custom Tags

FormBean Class :

A form bean is a type of Java bean. A form bean is an instance of a subclass of an ActionForm class, which stores HTML form data from a submitted client request or that can store input data from a Struts action link that a user clicked. An HTML form comprises fields in which the user can enter information.

A form-bean mapping is an entry in a Struts configuration file that maps a form bean to an action

Need of FormBean Class in Struts Application :

When a browser submits an HTML form, the Struts action servlet does as follows:

Looks at the field names from the HTML form
Matches them to the properties' names in the form bean
Automatically calls the set methods of these variables to put the values retrieved from the HTML form

In addition, if you implement a validate method and set the validate flag in the corresponding action mapping entry in the Struts configuration file, the action servlet invokes the validate method to validate that the data that the servlet receives is of the appropriate type.

Action Servlet :

It is a built-in Controller Servlet for every Struts application.The integration logic of this servlet program will be generated dynamically based on the rules and guidelines given in Struts Configuration file.This integration logic controls and decides the flow of execution in struts application.

Action Class :

It is a Java class which extends org.apache.struts.action.Action class.
It can acts as model layer resources by having the logic to communicate with other model layer resources like EJB component, Spring Application etc.

web.xml :

The web.xml web application descriptor file represents the core of the Java web application, so it is appropriate that it is also part of the core of the Struts framework. In the web.xml file, Struts defines its FilterDispatcher, the Servlet Filter class that initializes the Struts framework and handles all requests. This filter can contain initialization parameters that affect what, if any, additional configuration files are loaded and how the framework should behave.

Struts Configuration File(xml file)

The struts-config.xml configuration file is a link between the View and Model components in the Web application. It plays an important role in building both Controller components and Application-specific configurations.This file should be created inside WEB-INF directory of the project.

d **<u>Explain Filter dispatcher and any two Actions components used by struts.</u>**

FilterDispatcher is used as the Front Controller. To use the FilterDispatcher it has to be registered in the deployment descriptor (web.xml) as below.

```
<filter>
<filter-name>struts2</filter-name>
<filter-class>
org.apache.struts2.dispatcher.FilterDispatcher
<filter-class>
</filter>
<filter-mapping>
<filter-name>struts2</filter-name>
<url-pattern>/*</url-patter>
</filter-mapping>
```

Struts splits the task processing in its filter dispatcher into subcomponents called interceptors. Eg the work of the first interceptor is to populate the action object with the request parameters.

In Struts action method is executed after the the action’s properties are populated.

The method in the action returns a String. Eg if it returns a “Success” , or “failure”, depending upon the return value it can be forwarded to the result page. The forwarded result may not always be a jsp, it can even be a file to be downloaded.

Working of the Filter Dispatcher

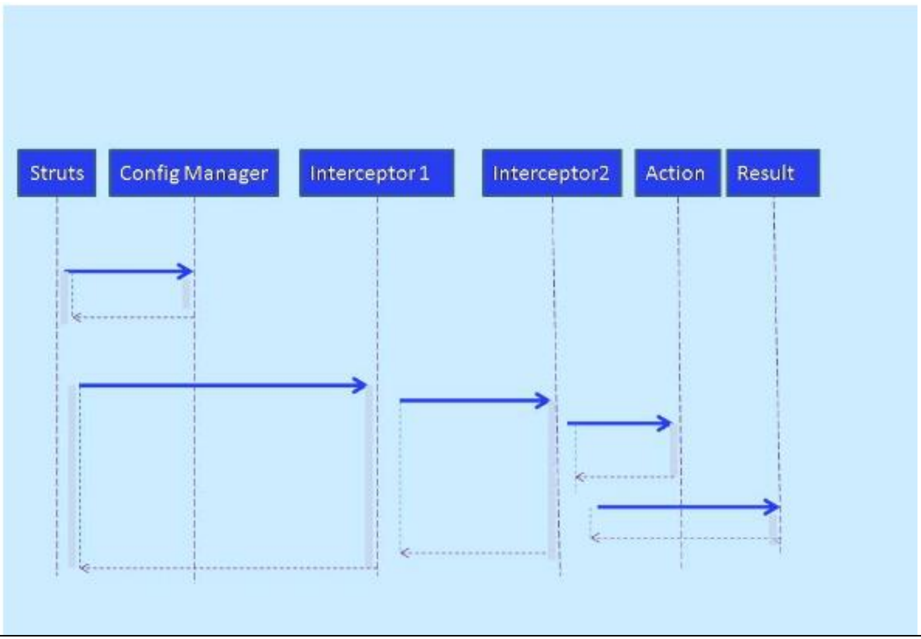
The work of the Filter Dispatcher is to first verify the request URI and determine which action to invoke. Struts uses the configration file struts.xml which matches URIs with action classes. One has to create a struts.xml and put it in the WEB-INF/classes folder. In this xml file the action definitions,concrete action class name, URI to invoke the action should be placed. If the name of the method in the action is other then execute then it should be also mentioned in the web.xml.

The action class should have at least one result to tell what to do after the action method is executed. It can have multiple results if the action method returns different results depending upon the user input.

When struts starts it reads the struts.xml. When ever struts gets a request it checks the timestamp of the struts.xm. Itf it is recent then when it was last time loaded then it is reloaded, thereby if one makes changes to the struts.xml, restarting the server is not required.

During each action invocation the filter dispatcher does the following steps.

1. It consults the Configuration Manager to find out which action to execute depending on the request URI.
2. Invoke the interceptors registered with this action. First interceptor will populate the properties of the action.
3. Executes the action method.
4. Executes the result.



7	**Attempt *__any three__* of the following:**	**15**

a **__Explain the Event Delegation Model.__**

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from the user interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code.

In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them .Previously, an event was propagated up the containment hierarchy until it was handled by a component. This required components to receive events that they did not process, and it wasted valuable time. The delegation event model eliminates this overhead.

1)Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface.

For example, an event may be generated when a timer expires, a counter exceeds a value, a software or hardware failure occurs, or an operation is completed.

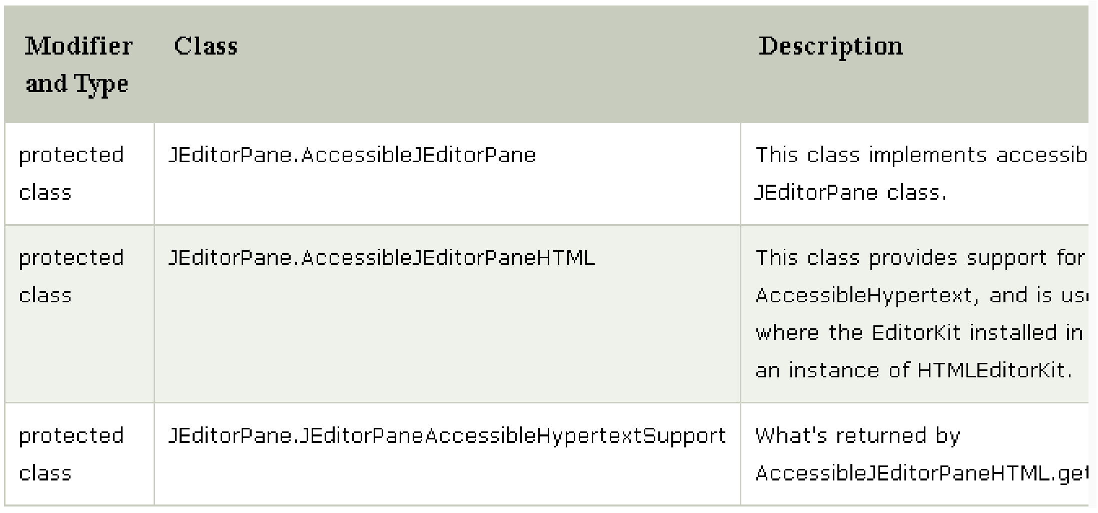
2)Event Sources
A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event.
A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method.
Here is the general form:
public void addTypeListener(TypeListener el)
Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called addKeyListener( ).
3) Event Listeners
A listener is an object that is notified when an event occurs. It has two major requirements.
First, it must have been registered with one or more sources to receive notifications about specific types of events. Second, it must implement methods to receive and process these notifications.
Just follow these two steps:
Implement the appropriate interface in the listener so that it can receive the type of event desired. Implement code to register and unregister (if necessary) the listener as a recipient for the event notifications.
A source may generate several types of events. Each event must be registered separately. Also, an object may register to receive several types of events, but it must implement all of the interfaces that are required to receive these events.

b **<u>Explain the use of Editor Panes in java swings.</u>**

EditorPane class is used to create a simple text editor window. This class has setContentType() and setText() methods.

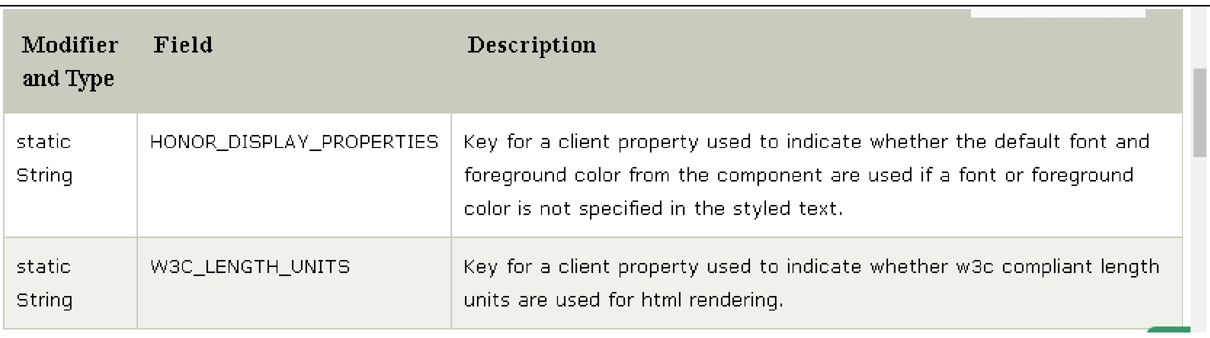
setContentType("text/plain"): This method is used to set the content type to be plain text.

setText(text): This method is used to set the initial text content



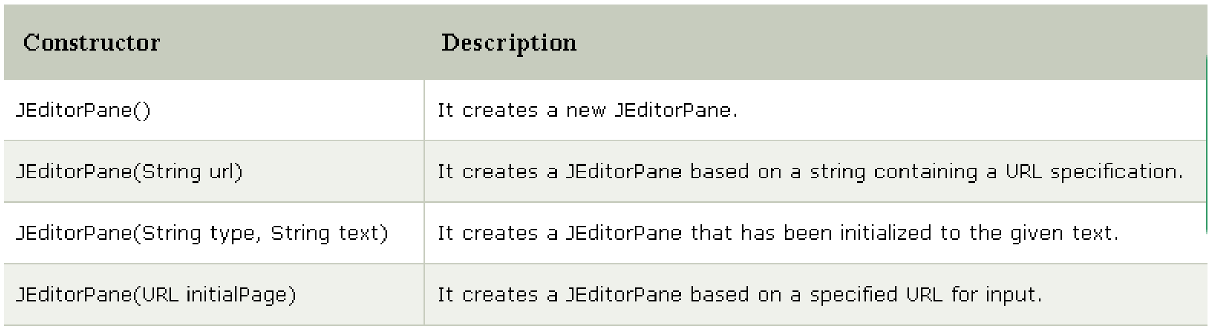
Modifier and Type	Class	Description
protected class	JEditorPane.AccessibleJEditorPane	This class implements accessib JEditorPane class.
protected class	JEditorPane.AccessibleJEditorPaneHTML	This class provides support for AccessibleHypertext, and is us where the EditorKit installed in an instance of HTMLEditorKit.
protected class	JEditorPane.JEditorPaneAccessibleHypertextSupport	What's returned by AccessibleJEditorPaneHTML.ge

Fields



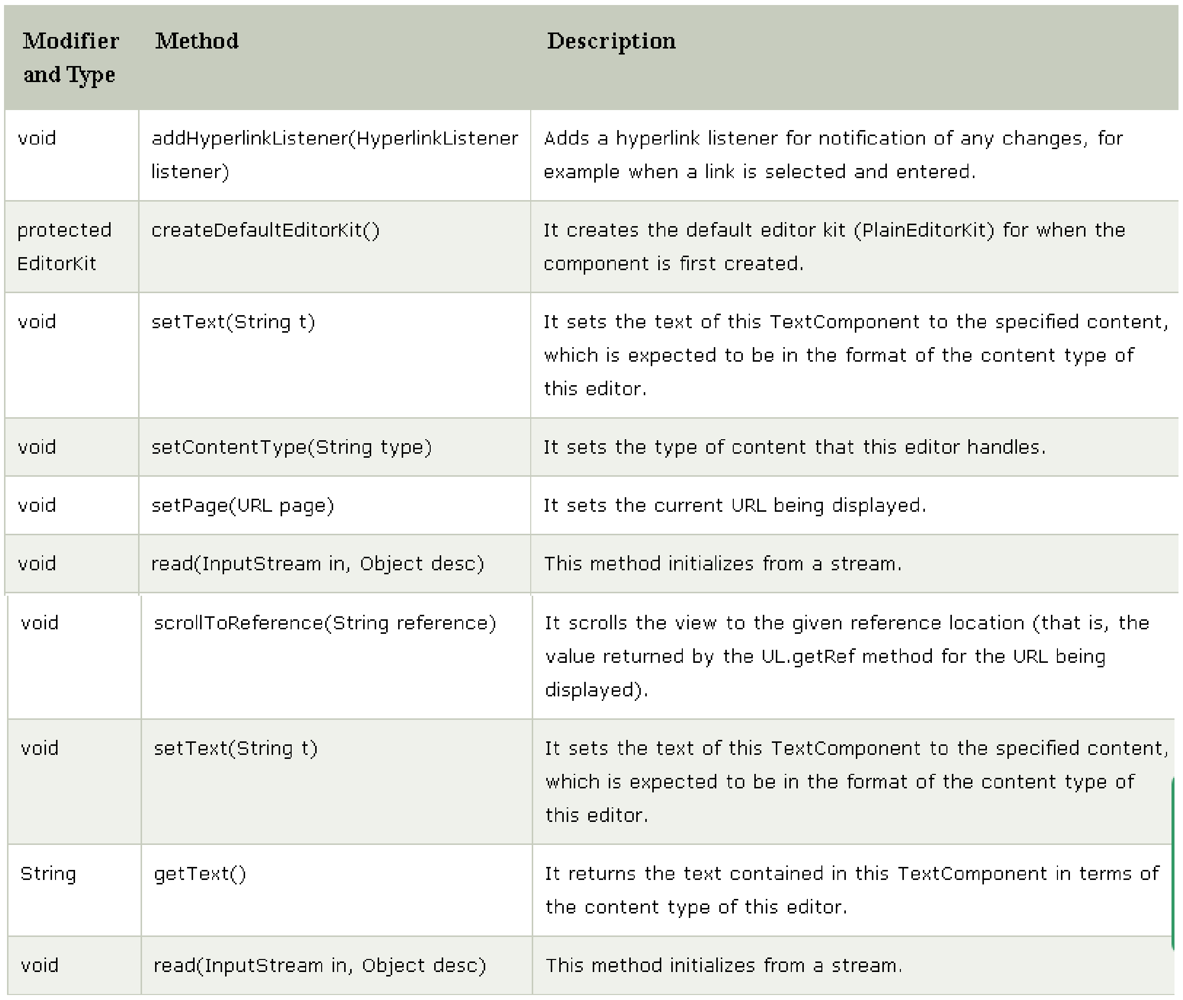
Modifier and Type	Field	Description
static String	HONOR_DISPLAY_PROPERTIES	Key for a client property used to indicate whether the default font and foreground color from the component are used if a font or foreground color is not specified in the styled text.
static String	W3C_LENGTH_UNITS	Key for a client property used to indicate whether w3c compliant length units are used for html rendering.

Constructors



Constructor	Description
JEditorPane()	It creates a new JEditorPane.
JEditorPane(String url)	It creates a JEditorPane based on a string containing a URL specification.
JEditorPane(String type, String text)	It creates a JEditorPane that has been initialized to the given text.
JEditorPane(URL initialPage)	It creates a JEditorPane based on a specified URL for input.

Useful Methods



Modifier and Type	Method	Description
void	addHyperlinkListener(HyperlinkListener listener)	Adds a hyperlink listener for notification of any changes, for example when a link is selected and entered.
protected EditorKit	createDefaultEditorKit()	It creates the default editor kit (PlainEditorKit) for when the component is first created.
void	setText(String t)	It sets the text of this TextComponent to the specified content, which is expected to be in the format of the content type of this editor.
void	setContentType(String type)	It sets the type of content that this editor handles.
void	setPage(URL page)	It sets the current URL being displayed.
void	read(InputStream in, Object desc)	This method initializes from a stream.
void	scrollToReference(String reference)	It scrolls the view to the given reference location (that is, the value returned by the UL.getRef method for the URL being displayed).
void	setText(String t)	It sets the text of this TextComponent to the specified content, which is expected to be in the format of the content type of this editor.
String	getText()	It returns the text contained in this TextComponent in terms of the content type of this editor.
void	read(InputStream in, Object desc)	This method initializes from a stream.

JEditorPane Example

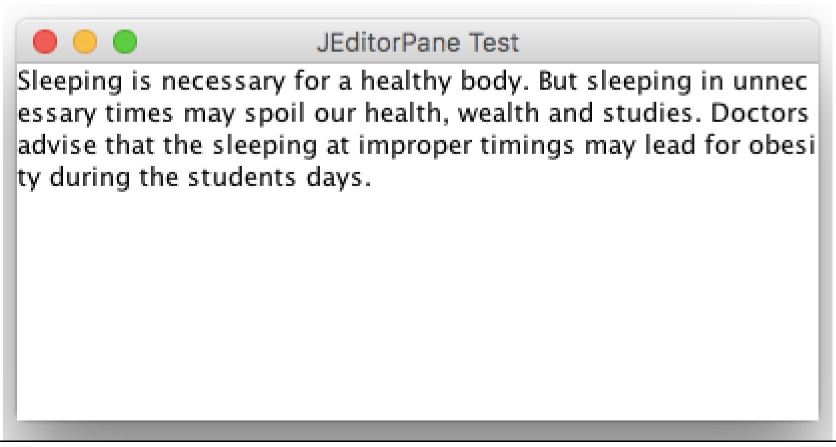
```
import javax.swing.JEditorPane;
import javax.swing.JFrame;

public class JEditorPaneExample {
    JFrame myFrame = null;

    public static void main(String[] a) {
        (new JEditorPaneExample()).test();
    }

    private void test() {
        myFrame = new JFrame("JEditorPane Test");
        myFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        myFrame.setSize(400, 200);
        JEditorPane myPane = new JEditorPane();
        myPane.setContentType("text/plain");
        myPane.setText("Sleeping is necessary for a healthy body."
                + " But sleeping in unnecessary times may spoil our health, wealth and studies."
                + " Doctors advise that the sleeping at improper timings may lead for obesity
during the students days.");
        myFrame.setContentPane(myPane);
        myFrame.setVisible(true);
    }
}
```

Output:

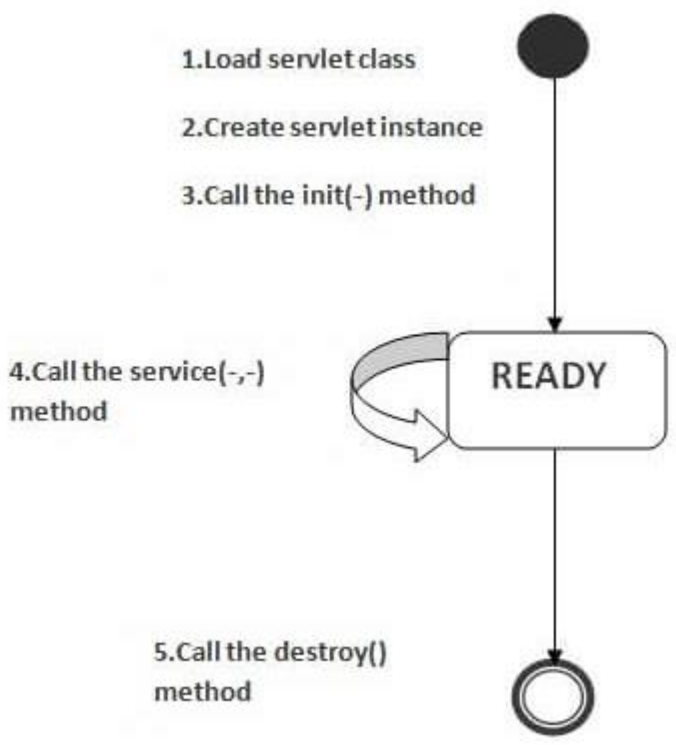


```
Sleeping is necessary for a healthy body. But sleeping in unnec
essary times may spoil our health, wealth and studies. Doctors
advise that the sleeping at improper timings may lead for obesi
ty during the students days.
```

c **Explain Servlet life cycle with help of suitable diagram.**

The web container maintains the life cycle of a servlet instance. Let's see the life cycle of the servlet:

1. Servlet class is loaded.
2. Servlet instance is created.
3. init method is invoked.
4. service method is invoked.
5. destroy method is invoked.


1.Load servlet class
2.Create servlet instance
3.Call the init(-) method
4.Call the service(-,-) method
READY
5.Call the destroy() method

As displayed in the above diagram, there are three states of a servlet: new, ready and end. The servlet is in new state if servlet instance is created. After invoking the init() method, Servlet comes in the ready state. In the ready state, servlet performs all the tasks. When the web container invokes the destroy() method, it shifts to the end state.

1) Servlet class is loaded

The classloader is responsible to load the servlet class. The servlet class is loaded when the first request for the servlet is received by the web container.

---

2) Servlet instance is created

The web container creates the instance of a servlet after loading the servlet class. The servlet instance is created only once in the servlet life cycle.

---

3) init method is invoked

The web container calls the init method only once after creating the servlet instance. The init method is used to initialize the servlet. It is the life cycle method of the javax.servlet.Servlet interface. Syntax of the init method is given below:

```
public void init(ServletConfig config) throws ServletException
```

4) service method is invoked

The web container calls the service method each time when request for the servlet is received. If servlet is not initialized, it follows the first three steps as described above then calls the service method. If servlet is initialized, it calls the service method. Notice that servlet is initialized only once. The syntax of the service method of the Servlet interface is given below:

```
public void service(ServletRequest request, ServletResponse response)
  throws ServletException, IOException
```

5) destroy method is invoked

The web container calls the destroy method before removing the servlet instance from the service. It gives the servlet an opportunity to clean up any resource for example memory, thread etc. The syntax of the destroy method of the Servlet interface is given below:

```
public void destroy()
```

d **<u>Write a JDBC program that inserts values in database. [table Name : Employee, Fileds : Empid, Name, Dept, Designation]</u>**

e **<u>List and explain elements of JSF.</u>**

1. User Interface Components
2. Managed Beans
3. Validators
4. Convertors
5. Events and Listeners
6. Page Navigation
7. Renderers

All the above-mentioned components can be found in separated packages in the API. For example, the ***User Interface Components***are available in javax.faces.component package, ***Validation API*** is available in javax.faces.validator package and so on.

1) User Interface Components
If Java Swings represent the UI components for a Desktop Java Application, then JSF UI Components are meant for the Web Applications. The JSF User Interface Components are developed with Java Beans Specifications and Standards in mind. It means that JSF Components have properties, methods and events as they are normally found for a traditional Java Bean. One of the peculiar features of the JSF UI Components is that they can manage the state of the component. So many built-in UI components are bundled with the JSF API, the most commonly used ones are Label, Text-Field, Form, Check-Box, Drop-Down Box etc. JSF also provides a framework for creating Customized UI Components.

It is very important to note that JSF UI Components only represent the attributes, behaviors and events for a component and not the actual display. For example in a Text-Field Control, the attributes may be the value of the text-field, the maximum number of characters that can be entered etc. Modifying the existing value and setting the maximum number of characters forms the behaviour for the Text-Field. Change in the value of the Text-Field may cause the Text-Field to cause some kind of Event to be emitted.

The manner is which text-field is displayed in the client is separated from the UI Data Representation itself. It means that there are separate components called Renderers which will take care of displaying the UI components in Different Client Surfaces, one client may be the HTML Browser running in a PC and the other may be the WML Browser running within a mobile phone. Each and every JSF component is uniquely identified by a Component Identifier.

2) Managed Beans

Managed Beans are standard Java classes that follow the Java Beans Specification.Generally, Managed Beans are used to represent the user inputs. They may even act as Listeners and can handle the appropriate Actions. Assume that there is an Encryption Application, which is presented with a Text-Field, wherein which user can enter a string that is to be encrypted. Below that is a Encrypt Button, which when clicked calls the server code that does the actual job of Encryption.

The following code snippet shows how to represent the Text-Field and the command button,

```
<html:inputText
                id = "strToBeEncryptedTextField"
                value = "#{EncryptionBean.strToBeEncrypted}">
</html:inputText>

        <html:commandButton
                id = "encryptButton"
                actionListener = "#{EncryptionBean.doEncryption}">
        </html:commandButton>
```

Don’t worry about the declaration of the Text-Field and the Command Button here. The syntax for their declaration in covered in the later sections. The first noticeable thing is that the Text-Field has two attributes namely id and value. The id attribute uniquely identifies the Text-Field component from other Components in the View.

The value for the attribute value is #{EncryptionBean.strToBeEncrypted}. This expression is a JSF EL Expression. The EL expression can be interpreted as follows, Take the value of the string to be encrypted from the UI Component and map it to the property called strToBeEncrypted which is inside the class called EncryptionBean.

It means that the declaration of the Managed Java Bean class may look something like the following,

```
class EncryptionBean{

                private String strToBeEncrypted;

                public String getStrToBeEncrypted(){
                        return strToBeencrypted;
                }

                public void setStrToBeEncrypted(String strToBeEncrypted){
                        this.strToBeEncrypted = strToBeEncrypted;
                }
                // Other things goes here
        }
```

From the declaration of the command button object, we can interpret that, ‘encryptButton’ is uniquely used to identify the command button object, the attribute actionListener is used to provide the listener method that will get invoked as a result of someone clicking the button. So, actionListener = “#{EncryptionBean.doEncryption}” essentially says that there is a method called doEncryption() within the EncryptionBean class. Following code snippet may prove this,

```
class EncryptionBean{
                …
                public void doEncryption(javax.faces.event.ActionEvent event){
                    }
                    …
          }
```
3) Validator

Validation is a must for almost any application. Data entered by the clients have to be validated before being sent to the Server for processing. JSF already have the Common Validation API being implemented for almost all controls in the javax.faces.validator package and also through various Custom Tags. The Validator Framework that is available with JSF is pluggable, i.e it also allows the developers to provide their own Custom Validation Classes through the help of Configuration Files.

Suppose say, there is a Text-Field Component in the form which inputs the age from the user to get some benefit from the organization. We can have a Simple Validation Rule telling that only Employees who are above 35 and below 45 are eligible for such a benefit. This can easily achieved through the use of Custom Validation Tags like the following one,

```
<html:inputText identifier = "employeeAgeTextField">
                    <f:validateLongRange minimum = "25" maximum = "35">
                    </f:validateLongRange>
          </html:inputText>
```
JSF will immediately report a Default Error Message whenever the value of the age entered by the user crosses the boundary range of 25-35.

Two types of Validation are possible in JSF. They are,

Direct Validation
Delegated Validation
i). Direct Validation:
Assume that there is a Customized UI component called E-Mail Text Field which allows the user to enter only email-ids in the appropriate format. Whenever a user enters some email-id, the UI component has to validate whether the given email-id is in the right format, something like username@someDomain.com. It is wise to embed the Validation Logic within the component itself. Such Validation Code which is found within the UI component and can be used only by that Component is called Direct Validation.

ii). Delegated Validation:

Delegation Validation comes into picture when the Validation Logic is about to be re-used across so many Components. Common Validation stuffs are ideal candidates for Delegated Validation. For example, consider in a form where we have a Text-Field representing the name of a customer and a Radio-Button for Marital Status with values 'Married' and 'Single'. If both are required fields, then before the submission of the form, both the input fields (text-field and the radio-button) have to be validated for empty (or null) values. So, in such a case we can have a Validator called NullCheckValidator and then bind both the input components to this Validator.

4) Convertors
Every piece of request that is passed from the client to the server is interpreted as a String value only. Manual conversion of the String object to the appropriate Data-type has to be done in the Application Code before carrying on with the application logic. Suppose in a form, there are fields like name, age and date. Corresponding to this form, we would have constructed a Managed Bean representing name, age and date as properties with String, integer and Date respectively. Certain amount of code has to be written for the conversion of the age and date values to their corresponding int and Date types.

But because of the availability of JSF Convertors functionality, this becomes easy. In the declaration of the UI Component itself within the form, we can mention which data-type the value for this control has to be converted. The Conversion API is available in javax.faces.convert.

For example, consider the following piece of code,

```
<html:inputText identifier = "numberTextField">
                <f:convertNumber pattern = "###,###">
                </f:convertNumber>
        </html:inputText>
```

In the above, we have a Number Converter which converts the number given by the user to the specified format. For example, if the original value entered by the user is 123456, then the after the conversion process the value becomes '123,456'.

5) Events and Listeners

JSF UI Event Mechanism is very similar to the one found in Swing UI Components. The Architecture remains the same in both the cases. In JSF, all the UI Components can emit any number of events. For example, a Button Component, when clicked by the user can emit an ActionEvent. Those which take appropriate actions after a component has emitted some kind of Events are called Event Listeners. In JSF, Java Managed Beans also acts as Listeners for the Events emitted by the components.

For example, consider the following code snippet,

```
<html:inputText
          identifier = "submitButton"
          value = "Click Me"
          actionListener = "#{SomeBean.submitButtonClicked}"
        </html:inputText>
```

In the above code, we can see that how a UI Component can be associated with an Event-Handler with the help of actionListener attribute. The code essentially says that whenever an ActionEvent is emitted by the button (which will happen usually when the user clicks the button or presses the Enter key over the button) call the Event Listener's method submitButtonClicked inside the SomeBean class.

6) Navigation

An user of a Web Application doesn't restrict himself in viewing one single Web Page. He/She will navigate from one page to another page. Technically the navigation of a user from one page to another page results in the generation of request and response for that page. Most of the boiler-plate work that is related to navigation stuffs in a web-application is handled by the Default JSF Navigation Handler itself. The Navigation Handler provides a simple yet a powerful framework for controlling the navigation.

For any single request page, there may be a number of response pages. Assuming that in a data-entry application, if the request page, say 'enterdata.jsp', is a view showing all the input controls to get data from the user, then the following responses may be available.

The user has entered all the data and the result is a 'success' so that a page called 'success.jsp' page is displayed

The user has entered a mix of correct and incorrect data in which case, the result or theresponse is a 'partialsuccess' and some jsp page called 'partialsuccess.jsp' is displayed.

The user hasn't entered any correct values which means that result is a 'failure' whichresults in the display of a page called 'failure.jsp'.

If we look at a very high-level, almost all of the pages will have this kind of Navigation Rule which will have N number of Navigational Cases. As such, we can define the Navigational Rules along with Navigational Cases for handling the navigational logic for the entire JSF application in the JSF Configuration File.

For example, the following is a simple navigation rule for the above sample scenario,

```
<navigation-rule>
                <from-view-id>/dataentry/enterdata.jsp</from-view-id>
                <navigation-case>
                        <from-outcome>success</from-outcome>
                        <to-view-id>/dataentry/success.jsp</to-view-id>
                </navigation-case>

                <navigation-case>
                        <from-outcome>partialsuccess</from-outcome>
                        <to-view-id>/dataentry/partialsuccess.jsp</to-view-id>
                </navigation-case>

                <navigation-case>
                        <from-outcome>failure</from-outcome>
                        <to-view-id>/dataentry/failure.jsp</to-view-id>
                </navigation-case>
        </navigation-rule>
```

The above code within the Configuration File states for the jsp file 'enterdata.jsp' (where '/dataentry/' is the context path), if the outcome is 'success', navigate to 'success.jsp, if the outcome is 'partialsuccess', then navigate to 'partialsuccess.jsp', else if the outcome is 'failure', then take the user to 'failure.jsp'.

f. **What is OGNL in struts? Explain the Execution Flow of OGNL.**

The **Object Graph Navigation Language** (OGNL) is an expression language. It simplifies the accessibility of data stored in the ActionContext.

The struts framework sets the **ValueStack** as the root object of OGNL. Notice that action object is pushed into the ValueStack. We can direct access the action property.

```
<s:property value="username"/>
```

Here, username is the property key.

The struts framework places other objects in ActionContext also e.g. map representing the **request**, **session**, **application** scopes.

To get these values i.e. not the action property, we need to use # notation. For example to get the data from session scope, we need to use #session as given in the following example:

```
<s:property name="#session.username"/>
```

(or)

```
<s:property name="#session['username']"/>
```

OGNL [Object-Graph Navigation Language]

The Object-Graph Navigation Language is a fully featured expression language for **Retrieving** and **Setting** properties of the Java object s. It helps **data transfer** and **type conversion.**

In the **Value Stack**,**Searching** or **evaluating**, a particular **expression** can be done using OGNL. OGNL provides a mechanism to navigate object graphs using a **dot notation** and **evaluate** expressions, including calling methods on the objects being retrieved.
OGNL supportsType conversion, calling methods, Collectionmanipulation, Generation, Projection across collections, Expression evaluation, Lambda expressions etc

OGNL Examples
The following are a few examples where OGNL is used:
emp.name
It returns the value that is actually returned when **getEmp().getName()** is Invoked
emp.toSpring()
It returns the value that is actually returned when **getEmp().toString()** is invoked.
@test.auth.name@fName()
It returns the value that is actual returned when the static method named fName() is invoked on the class name.
firstName in {"Sharanam","vaishali"}
invokes getfName() and determines he the value returned is either Tushar or Sonali. If it is, then returns True.

Execution flow of OGNL :
1. A user enters the data in and data entry form and submits the form
2. The Request enters the Struts 2 framework and is made available to the Java Language as an **HttpServletRequest** object.
3. The request paramets are stored as name/value paris where the name are the Names of the data entry from's text fields and the Value are the Value entered by the user when the form is submitted
4. Now the OGNL comes into picture , to handle the transfer and type conversion of the data from these request parameters
5. Using the OGNL expression , the value stack is scanned to locate the destination property where the data has to be moved
6. On locating the correct property in the Value Stack, the data is moved to the property by invoking the property's SETTER method with the appropriate value. The value stack acts as a place holder for viewing the data model throughout the execution
7. Whilst moving such data, OGNL consults its set of available type converters to determine if any of them can handle this particular conversion, if a conversion is required. The value is converted and set on the object's property. This makes the data available them action begins its job, immediately after the available Interceptors ate fired.
8. After Action completes to its job successfully, a Result fires that renders the result they to the user.
9. Results have access to the Value Stack, via the OGNL expression language with the help of tags. These tags retrieve data from the Value Stack by referencing specific values using OGNL expressions.
10. Whilst rendering the view , once again, the value that is accessed from the value Slack is converted from the java type to a String that is written on the HTML page