1. How to synchronize thread? Explain with example.

ANSWER

- When we start two or more threads within a program, there may be a situation when multiple threads try to access the same resource and finally they can produce unforeseen result due to concurrency issue. For example if multiple threads try to write within a same file then they may corrupt the data because one of the threads can overrite data or while one thread is opening the same file at the same time another thread might be closing the same file.

- So there is a need to synchronize the action of multiple threads and make sure that only one thread can access the resource at a given point in time. This is implemented using a concept called **monitors**. Each object in Java is associated with a monitor, which a thread can lock or unlock. Only one thread at a time may hold a lock on a monitor.

- Java programming language provides a very handy way of creating threads and synchronizing their task by using **synchronized** blocks. You keep shared resources within this block. Following is the general form of the synchronized statement:

```
synchronized(objectidentifier) {
  // Access shared variables and other shared resources

}
```

- Here, the **objectidentifier** is a reference to an object whose lock associates with the monitor that the synchronized statement represents.
- Example

```
class PrintDemo {

   public void printCount(){ try
    {

        for(int i = 5; i > 0; i--) {
           System.out.println("Counter --- " + i );

        }
     } catch (Exception e) { System.out.println("Thread
        interrupted.");

     }
   }

}

class ThreadDemo extends Thread {
   private Thread t;
```

```
        private String threadName;
        PrintDemo PD;

   ThreadDemo( String name, PrintDemo pd){ threadName = name;

      PD = pd;

   }

   public void run() { synchronized(PD) {
       PD.printCount();

     }

     System.out.println("Thread " +  threadName + " exiting.");
   }

   public void start ()
   {

     System.out.println("Starting " + threadName ); if (t == null)
     {

        t = new Thread (this, threadName); t.start ();

     }

   }

}

public class TestThread {
   public static void main(String args[]) {

     PrintDemo PD = new PrintDemo();

     ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD ); ThreadDemo T2 = new
     ThreadDemo( "Thread - 2 ", PD );

     T1.start();
     T2.start();

     // wait for threads to end try {
        T1.join();

        T2.join();

     } catch( Exception e) { System.out.println("Interrupted");
     }
   }
```

```
}
```

OUTPUT

```
Starting Thread - 1

Starting Thread - 2

Counter    ---    5

Counter      ---    4
Counter      ---    3
Counter      ---    2
Counter      ---    1
Thread Thread - 1    exiting.
Counter      ---    5
Counter      ---    4
Counter      ---    3
Counter      ---    2
Counter      ---    1
Thread Thread - 2    exiting.
```

2. What are the methods involved in communication between threads.

ANSWER:

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

METHOD	DESCRIPTION
public final void wait()throws	waits until object is notified.

InterruptedException	
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax: public final void notifyAll()

Example of inter thread communication in java

```
class Customer{
  int amount=10000;

  synchronized void withdraw(int amount){
  System.out.println("going to withdraw...");

  if(this.amount<amount){

  System.out.println("Less balance; waiting for deposit...");
  try{wait();}catch(Exception e){}

  }

  this.amount-=amount; System.out.println("withdraw completed...");
  }

  synchronized void deposit(int amount){
  System.out.println("going to deposit..."); this.amount+=amount;
  System.out.println("deposit completed... "); notify();

  }
  }

  class Test{

  public static void main(String args[]){ final Customer
  c=new Customer(); new Thread(){

  public void run(){c.withdraw(15000);} }.start();
  new Thread(){

  public void run(){c.deposit(10000);} }.start();
```

}}

Output:

going to withdraw...

Less balance; waiting for deposit...

going to deposit...

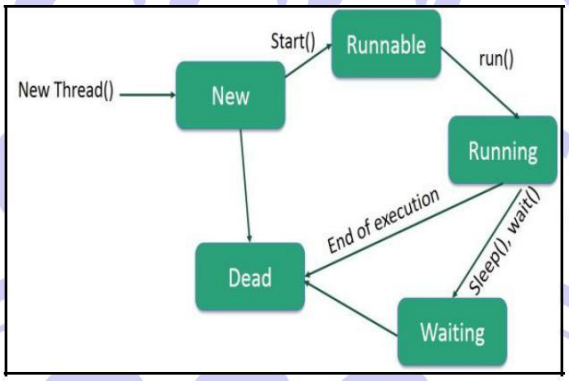
deposit completed..

withdraw completed

3. Explain the Lifecycle of a Thread.

ANSWER

A thread goes through various stages in its life cycle. For example, a thread is born, started, runs, and then dies. Following diagram shows complete life cycle of a thread.



Above-mentioned stages are explained here:

- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.
- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes, a thread transitions to the waiting state while the thread waits for another thread to perform a task.A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.

- **Terminated (Dead):** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Example

```
class RunnableDemo implements Runnable {
   private Thread t;

   private String threadName;

   RunnableDemo( String name){
       threadName = name;
       System.out.println("Creating " +  threadName );

   }
   public void run() {
   System.out.println("Running " + threadName ); try {
        for(int i = 4; i > 0; i--) {

           System.out.println("Thread: " + threadName + ", " + i); // Let
           the thread sleep for a while.

           Thread.sleep(50);

        }
     } catch (InterruptedException e) {

        System.out.println("Thread " +  threadName + " interrupted.");
     }

     System.out.println("Thread " +  threadName + " exiting.");

   }

   public void start ()
   {

      System.out.println("Starting " + threadName ); if (t
      == null)
      {

         t = new Thread (this, threadName);
         t.start ();

      }

   }

}

public class TestThread {
```

```
        public static void main(String args[]) {

           RunnableDemo R1 = new RunnableDemo( "Thread-1");

           R1.start();


           RunnableDemo R2 = new RunnableDemo( "Thread-2");
           R2.start();

        }

     }
```

4. What are Thread Safe variables?

ANSWER:

Thread safety simply means that the fields of an object or class always maintain a valid state, as observed by other objects and classes, even when used concurrently by multiple threads.

Multiple threads can spell trouble for your object because often, while a method is in the process of executing, the state of your object can be temporarily invalid. When just one thread is invoking the object's methods, only one method at a time will ever be executing, and each method will be allowed to finish before another method is invoked. Thus, in a single-threaded environment, each method will be given a chance to make sure that any temporarily invalid state is changed into a valid state before the method returns.

Once you introduce multiple threads, however, the JVM may interrupt the thread executing one method while the object's instance variables are still in a temporarily invalid state. The JVM could then give a different thread a chance to execute, and that thread could call a method on the same object. All your hard work to make your instance variables private and your methods perform only valid state transformations will not be enough to prevent this second thread from observing the object in an invalid state.

Such an object would not be thread-safe, because in a multithreaded environment, the object could become corrupted or be observed to have an invalid state. A thread-safe object is one that always maintains a valid state, as observed by other classes and objects, even in a multithreaded environment.

Example

```
public class ThreadSafety {

   public static void main(String[] args) throws InterruptedException {

      ProcessingThread pt = new ProcessingThread();
      Thread t1 = new Thread(pt, "t1");
      t1.start();
```

```
        Thread t2 = new Thread(pt, "t2");
        t2.start();

        //wait for threads to finish processing
        t1.join();
        t2.join();

        System.out.println("Processing count="+pt.getCount());

    }

}

class ProcessingThread implements Runnable{
    private int count;

    @Override

    public void run() { for(int
        i=1; i< 5; i++){
            processSomething(i);
            count++;

        }

    }

    public int getCount() {
        return this.count;

    }

    private void processSomething(int i) { //
        processing some job

        try { Thread.sleep(i*1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}
```

5. Explain with an example creation of thread using Runnable.

ANSWER

- A Thread can be created by extending Thread class also. But Java allows only one class to extend, it wont allow multiple inheritance. So it is always better to create a

thread by implementing Runnable interface. Java allows you to impliment multiple interfaces at a time.

- By implementing Runnable interface, you need to provide implementation for run() method.
- To run this implementation class, create a Thread object, pass Runnable implementation class object to its constructor. Call start() method on thread class to start executing run() method.
- Implementing Runnable interface does not create a Thread object, it only defines an entry point for threads in your object. It allows you to pass the object to the Thread(Runnable implementation) constructor.

Example

```
class MyRunnableThread implements Runnable{

    public static int myCount = 0;
    public MyRunnableThread(){

    }

    public void run() {
        while(MyRunnableThread.myCount <= 10){

            try{

                System.out.println("Expl Thread: "+(++MyRunnableThread.myCount));
                Thread.sleep(100);

            } catch (InterruptedException iex) { System.out.println("Exception in
                thread: "+iex.getMessage());

            }
        }

    }
}

public class RunMyThread {
```

```
public static void main(String a[]){ System.out.println("Starting Main
Thread..."); MyRunnableThread mrt = new MyRunnableThread();
Thread t = new Thread(mrt);

        t.start(); while(MyRunnableThread.myCount <= 10){

            try{

                System.out.println("Main Thread: "+(++MyRunnableThread.myCount));
                Thread.sleep(100);

            } catch (InterruptedException iex){
                System.out.println("Exception in main thread: "+iex.getMessage());

            }
        }

        System.out.println("End of Main Thread...");

    }
}
```

Example Output

```
Starting Main Thread...

Main Thread: 1
Expl Thread: 2

Main Thread: 3

Expl Thread: 4
Main Thread: 5

Expl Thread: 6
Main Thread: 7

Expl Thread: 8

Main Thread: 9
Expl Thread: 10

Main Thread: 11
End of Main Thread...
```

6. Differentiate between multitasking process & multitasking threads.
ANSWER

There are two types of multitasking:

1. Process Based

2. Thread based

SR. NO.	Multitasking Process	Multitasking Threads
1	Two or more thane two programs xecute concurrently. For example, we can do typing in MS-Word as well as listen the songs on Media Player.	Single program perform two or more task simultaneously. For example, at the same time we save the word file (.docx file) as well as we order same word file for printing.
2	More overhead compared to multitasking threads	Less overhead
3	Running heavyweight tasks	Running lightweight tasks
4	It has separate address space	It shares the address space.
5	Inter-process communication is expensive and limited	Inter-thread communication is inexpensive. Context switching between one thread to another is of low cost.

7. Explain any four methods of threads.
ANSWER

Multithreading refers to two or more tasks executing concurrently within a single program. A thread is an independent path of execution within a program. Many threads can run concurrently within a program. Every thread in Java is created and controlled by the **java.lang.Thread class**. A Java program can have many threads, and these threads can run concurrently, either asynchronously or synchronously.

Multithreading has several advantages over Multiprocessing such as;

- Threads are lightweight compared to processes
- Threads share the same address space and therefore can share both data and code
- Context switching between threads is usually less expensive than between processes
- Cost of thread intercommunication is relatively low that that of process intercommunication
- Threads allow different tasks to be performed concurrently.

The **java.lang.Thread** class is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.Following are the important points about Thread:

- Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority
- Each thread may or may not also be marked as a daemon.
- There are two ways to create a new thread of execution. One is to declare a class to be a subclass of Thread and,
- the other way to create a thread is to declare a class that implements the Runnable interface

- Class methods

S.N.	Method & Description
4	**static Thread currentThread()** This method returns a reference to the currently executing thread object.
10	**long getId()** This method returns the identifier of this Thread.
11	**String getName()** This method returns this thread's name.
12	**int getPriority()** This method Returns this thread's priority.
14	**Thread.State getState()** This method returns the state of this thread.
18	**void interrupt()** This method interrupts this thread.
19	**static boolean interrupted()** This method tests whether the current thread has been interrupted.
20	**boolean isAlive()** This method tests if this thread is alive.
21	**boolean isDaemon()** This method tests if this thread is a daemon thread.

22	**boolean isInterrupted()** This method tests whether this thread has been interrupted.
23	**void join()** Waits for this thread to die.
24	**void join(long millis)** Waits at most millis milliseconds for this thread to die.
25	**void join(long millis, int nanos)** Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.
26	**void run()** If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns
27	**void setContextClassLoader(ClassLoader cl)** This method sets the context ClassLoader for this Thread.
28	**void setDaemon(boolean on)** This method marks this thread as either a daemon thread or a user thread.
29	**static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)** This method set the default handler invoked when a thread abruptly terminates due to an uncaught exception, and no other handler has been defined for that thread.
30	**void setName(String name)** This method changes the name of this thread to be equal to the argument name.
31	**void setPriority(int newPriority)** This method changes the priority of this thread.

33	**static void sleep(long millis)** This method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
34	**static void sleep(long millis, int nanos)** This method causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
35	**void start()** This method causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
37	**static void yield()** This method causes the currently executing thread object to temporarily pause and allow other threads to execute.

8. Explain thread priorities.
ANSWER

Description

Each thread have a priority. Priorities are represented by a number between 1 and 10. In most cases, thread schedular schedules the threads according to their priority (known as preemptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

The **java.lang.Thread.setPriority()** method changes the priority of this thread.

3 constants defiend in Thread class:

1. public static int MIN_PRIORITY
2. public static int NORM_PRIORITY
3. public static int MAX_PRIORITY

Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

Declaration

Following is the declaration for **java.lang.Thread.setPriority()** method

```
public final void setPriority(int newPriority)
```

Parameters

- **newPriority** -- This is the priority to set this thread to.

Return Value

This method does not return any value.

Exception

- **IllegalArgumentException** -- if the priority is not in the range MIN_PRIORITY to MAX_PRIORITY.
- **SecurityException** -- if the current thread cannot modify this thread.

Example

The following example shows the usage of java.lang.Thread.setPriority() method.

```
import java.lang.*;
public class ThreadDemo {
  public static void main(String[] args) {
  Thread t = Thread.currentThread();
  t.setName("Admin Thread");
  // set thread priority to 1
  t.setPriority(1);
  // prints the current thread
  System.out.println("Thread = " + t);
  int count = Thread.activeCount();
  System.out.println("currently active threads = " + count);
  }
}
```

Let us compile and run the above program, this will produce the following result:

```
Thread = Thread[Admin Thread,1,main]
currently active threads = 1
```

9. Write short note on timer class of UTIL package.
ANSWER

A facility for threads to schedule tasks for future execution in a background thread. Tasks may be scheduled for one-time execution, or for repeated execution at regular intervals.

Corresponding to each Timer object is a single background thread that is used to execute all of the timer's tasks, sequentially. Timer tasks should complete quickly. If a timer task takes excessive time to complete, it "hogs" the timer's task execution thread. This can, in turn, delay the execution of subsequent tasks, which may "bunch up" and execute in rapid succession when (and if) the offending task finally completes.

After the last live reference to a Timer object goes away *and* all outstanding tasks have completed execution, the timer's task execution thread terminates gracefully (and becomes subject to garbage collection). However, this can take arbitrarily long to occur. By default, the task execution thread does not run as a *daemon thread*, so it is capable of keeping an application from terminating. If a caller wants to terminate a timer's task execution thread rapidly, the caller should invoke the timer's cancel method.

If the timer's task execution thread terminates unexpectedly, for example, because its stop method is invoked, any further attempt to schedule a task on the timer will result in an IllegalStateException, as if the timer's cancel method had been invoked.

This class is thread-safe: multiple threads can share a single Timer object without the need for external synchronization.

This class does *not* offer real-time guarantees: it schedules tasks using the Object.wait(long) method.

Java 5.0 introduced the java.util.concurrent package and one of the concurrency utilities therein is the ScheduledThreadPoolExecutor which is a thread pool for repeatedly executing tasks at a given rate or delay. It is effectively a more versatile replacement for the Timer/TimerTask combination, as it allows multiple service threads, accepts various time units, and doesn't require subclassing TimerTask (just implement Runnable). Configuring ScheduledThreadPoolExecutorwith one thread makes it equivalent to Timer.

Declaration

Following is the declaration for **java.util.Timer.schedule()** method.

```
public void schedule(TimerTask task,long delay,long period)
```

Parameters

- **task** -- This is the task to be scheduled.
- **delay** -- This is the delay in milliseconds before task is to be executed.
- **period** -- This is the time in milliseconds between successive task executions.

Return Value

- **NA**

Exception

- **IllegalArgumentException** -- This exception is thrown if time.getTime() is negative.

- **IllegalStateException** -- This is thrown if task was already scheduled or cancelled, timer was cancelled, or timer thread terminated.

Example

The following example shows the usage of java.util.Timer.schedule()

```
import java.util.*;
public class TimerDemo {
   public static void main(String[] args) {
   // creating timer task, timer
   TimerTask tasknew = new TimerSchedulePeriod();
   Timer timer = new Timer();

   // scheduling the task at interval
   timer.schedule(tasknew,100, 100);
   }
   // this method performs the task
   public void run() {
   System.out.println("timer working");
   }
}
```

Let us compile and run the above program, this will produce the following result.

```
timer working
timer working
timer working
timer working and so on ...
```

10. Write short note on Inter thread communication.
ANSWER

Inter-thread communication or **Co-operation** is all about allowing synchronized threads to communicate with each other.

Cooperation (Inter-thread communication) is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

Causes current thread to release the lock and wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.

The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	waits until object is notified.
public final void wait(long timeout)throws InterruptedException	waits for the specified amount of time.

2) notify() method

Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation. Syntax:

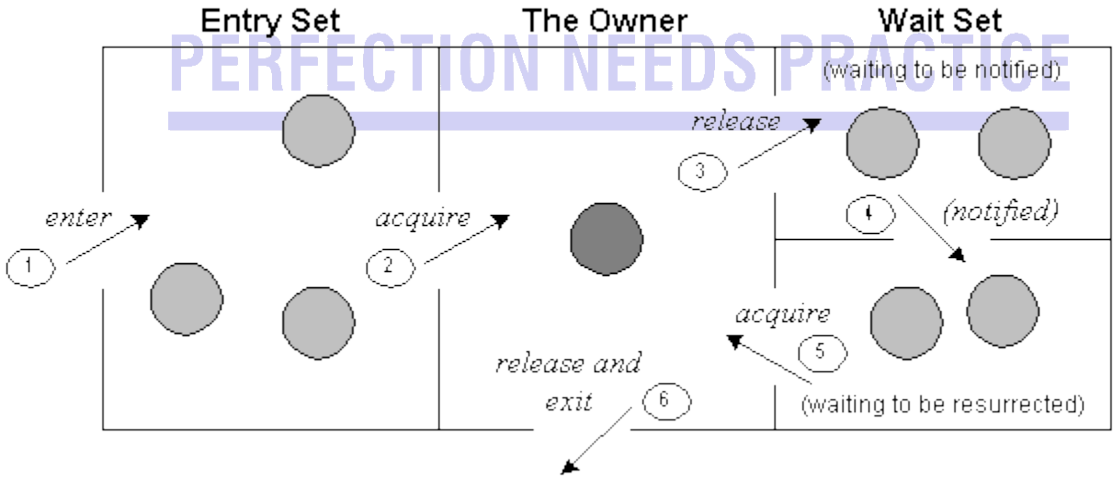
public final void notify()

3) notifyAll() method

Wakes up all threads that are waiting on this object's monitor. Syntax:

public final void notifyAll()

Understanding the process of inter-thread communication



The point to point explanation of the above diagram is as follows:

1. Threads enter to acquire lock.
2. Lock is acquired by on thread.
3. Now thread goes to waiting state if you call wait() method on the object. Otherwise it releases the lock and exits.
4. If you call notify() or notifyAll() method, thread moves to the notified state (runnable state).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Example of inter thread communication in java

Let's see the simple example of inter thread communication.

```
class Customer{
int amount=10000;

synchronized void withdraw(int amount){
System.out.println("going to withdraw...");

if(this.amount<amount){
System.out.println("Less balance; waiting for deposit...");
try{wait();}catch(Exception e){}
}
this.amount-=amount;
System.out.println("withdraw completed...");
}

synchronized void deposit(int amount){
System.out.println("going to deposit...");
this.amount+=amount;
System.out.println("deposit completed... ");
notify();
}
}

class Test{
public static void main(String args[]){
final Customer c=new Customer();
new Thread(){
public void run(){c.withdraw(15000);}
}.start();
new Thread(){
public void run(){c.deposit(10000);}
}.start();
}
}
```

Output: going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed

Q1)Explain in brief the Use of JInternalFrame.

1. A JInternalFrame is confined to a visible area of a container it is placed in. JInternalFrame a top level swing component that has a contentpane.

   - It can be iconified — in this case the icon remains in the main application container.
   - It can be maximized — Frame consumes the main application
   - It can be closed using standard popup window controls
   - It can be layered

2) With the JInternalFrame class you can display a JFrame-like window within another window. Usually, you add internal frames to a desktop pane. The desktop pane, in turn, might be used as the content pane of a JFrame. The desktop pane is an instance of JDesktopPane, which is a subclass of JLayeredPane that has added API for managing multiple overlapping internal frames.

3) **HIERARCHY**

javax.swing

Class JInternalFrame
java.lang.Object

|

+–java.awt.Component

|

+–java.awt.Container

|

+–javax.swing.JComponent

|

+–javax.swing.JInternalFrame

4) **CONSTRUCTORS**

- JInternalFrame()

Creates a non-resizable, non-closable, non-maximizable, non-iconifiable JInternalFrame with no title.

- JInternalFrame(String title)

Creates a non-resizable, non-closable, non-maximizable, non-iconifiable JInternalFrame with the specified title.

- JInternalFrame (String title, boolean resizable)

Creates a non-closable, non-maximizable, non-iconifiable JInternalFrame with the specified title and resizability.

- JInternalFrame (String title, boolean resizable, boolean closable)

Creates a non-maximizable, non-iconifiable JInternalFrame with the specified title, resizability, and closability.

- JInternalFrame (String title, boolean resizable, boolean closable, boolean maximizable)

Creates a non-iconifiable JInternalFrame with the specified title, resizability, closability, and maximizability.

- JInternalFrame(String title, boolean resizable, boolean closable, boolean maximizable, boolean iconifiable)

Creates a JInternalFrame with the specified title, resizability, closability, maximizability

Rules of Using Internal Frames

- You must set the size of the internal frame.
- As a rule, you should set the location of the internal frame.
- To add components to an internal frame, you add them to the internal frame's content pane.

- Dialogs that are internal frames should be implemented using **JOptionPane** or **JInternalFrame**, not **JDialog**.

  - You must add an internal frame to a container.
  - You need to call **show** or **setVisible** on internal frames.
  - Internal frames fire internal frame events, not window events.

Q2)Write a short note on JOptionPane.

- Using JOptionPane, you can quickly create and customize several different kinds of dialogs.
- JOptionPane provides support for laying out standard dialogs, providing icons, specifying the dialog title and text, and customizing the button text.
- Other features allow you to customize the components the dialog displays and specify where the dialog should appear onscreen.

You can even specify that an option pane put itself into an internal frame (JInternalFrame) instead of a JDialog.

1. When you create a JOptionPane, look-and-feel-specific code adds components to the JOptionPane and determines the layout of those components.
2. JOptionPane's icon support lets you easily specify which icon the dialog displays. You can use a custom icon, no icon at all, or any one of four

   standard JOptionPane icons (question, information, warning, and error). Each look and feel has its own versions of the four standard icons. The following figure shows the icons used in the Java (and Windows) look and feel.

Method Name	Description
showConfirmDialog	Asks a confirming question, like yes/no/cancel.
showInputDialog	Prompt for some input.
showMessageDialog	Tell the user about something that has happened.
showOptionDialog	The Grand Unification of the above three.

Q3)Explain any three text component in swing.

- The JTextComponent class is the foundation for Swing text components. This class provides the following customizable features for all of its descendants:

1. A model, known as a *document*, that manages the component's content.
2. A view, which displays the component on screen.
3. A controller, known as an *editor kit*, that reads and writes text and implements editing capabilities with actions.
4. Support for infinite undo and redo.
5. A pluggable caret and support for caret change listeners and navigation filters.

- Swing provides six text components, along with supporting classes and interfaces that meet even the most complex text requirements. In spite of their different uses and capabilities, all Swing text components inherit from the same

  superclass, JTextComponent, which provides a highly-configurable and powerful foundation for text manipulation.

- The following figure shows the JTextComponent hierarchy.

- **JTextField**

  JTextField is a lightweight component that allows the editing of a single line of text.

  JTextField is intended to be source-compatible with java.awt.TextField where it is reasonable to do so. This component has capabilities not found in

  the java.awt.TextField class. The superclass should be consulted for additional capabilities.

  JTextField has a method to establish the string used as the command string for the action event that gets fired. The java.awt.TextField used the text of the field as the command string for the ActionEvent. JTextField will use the command string set with the setActionCommand method if not null, otherwise it will use the text of the field as a compatibility with java.awt.TextField.

- **JFormattedTextField**

  JFormattedTextField extends JTextField adding support for formatting arbitrary values, as well as retrieving a particular object once the user has edited the text. Formatted text fields provide a way for developers to specify the valid set of characters that can be typed in a text field. Specifically,

  the JFormattedTextField class adds a *formatter* and an object *value* to the features inherited from the JTextField class. The formatter translates the field's value into the text it displays, and the text into the field's value.

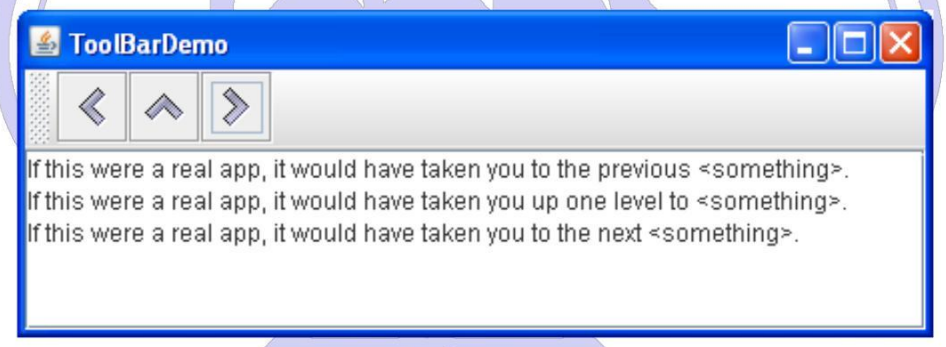
  Another kind of formatter enables you to use a character mask to specify the set of characters that can be typed at each position in the field. For example, you can specify a mask for typing phone numbers in a particular format, such as (XX) X-XX-XX-XX-XX.

the one typed, such as an asterisk '*'. As another security precaution, a password field stores its value as an array of characters, rather than as a string. JPasswordField is a lightweight component that allows the editing of a single line of text where the view indicates something was typed, but does not show the original characters.

JPasswordField is intended to be source-compatible with java.awt.TextField used with echoChar set. It is provided separately to make it easier to safely change the UI for the JTextField without affecting password entries.

Q4)Briefly explain JScrollPane.

- A JScrollPane provides a scrollable view of a component. When screen real estate is limited, use a scroll pane to display a component that is large or one whose size can change dynamically. Other containers used to save screen space include split panes and tabbed panes.
- The code to create a scroll pane can be minimal. For example, here's a picture of a demo program that puts a text area in a scroll pane because the text area's size grows dynamically as text is appended to it:

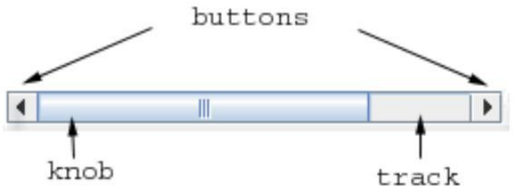


//In a container that uses a BorderLayout:

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

setPreferredSize(new Dimension(450, 110));
...

add(scrollPane, BorderLayout.CENTER);
```

- A scroll pane uses a JViewport instance to manage the visible area of the client. The viewport is responsible for positioning and sizing the client, based on the positions of the scroll bars, and displaying it.

- A scroll pane may use two separate instances of JScrollBar for the scroll bars. The scroll bars provide the interface for the user to manipulate the visible area. The following figure shows the three areas of a scroll bar: the knob (sometimes called the *thumb*), the (arrow) buttons, and the track.



- When the user moves the knob on the vertical scroll bar up and down, the visible area of the client moves up and down. Similarly, when the user moves the knob on the horizontal scroll bar to the right and left, the visible area of the client moves back and forth accordingly. The position of the knob relative to its track is proportionally equal to the position of the visible area relative to the client. In the Java look and feel and some others, the size of the knob gives a visual clue as to how much of the client is visible.

- By clicking an arrow button, the user can scroll by a *unit increment*. By clicking within the track, the user can scroll by a *block increment*. If the user has a mouse with a wheel, then the user can scroll vertically using the mouse wheel. The amount that the mouse wheel scrolls is platform dependent. For example, by default on Windows XP, the mouse wheel scrolls three unit increments; the Mouse control panel allows you to specify a different number of unit increments or to use a block increment instead.

- JScrollPane constructors and methods

Method or Constructor	Purpose
JScrollPane() JScrollPane(Component) JScrollPane(int, int) JScrollPane(Component, int, int)	Create a scroll pane. The Component parameter, when present, sets the scroll pane's client. The two int parameters, when present, set the vertical and horizontal scroll bar policies (respectively).
void setViewportView(Component)	Set the scroll pane's client.
void setVerticalScrollBarPolicy(int) int getVerticalScrollBarPolicy()	Set or get the vertical scroll policy. ScrollPaneConstants defines three values for specifying this policy:VERTICAL_SCROLLBAR_AS_NEEDED (the

	default), VERTICAL_SCROLLBAR_ALWAYS, and VERTICAL_SCROLLBAR_NEVER.
void setHorizontalScrollBarPolicy(int)	Set or get the horizontal scroll policy. ScrollPaneConstants defines three values for specifying this policy:HORIZONTAL_SCROLLBAR_AS_NEEDED (the default), HORIZONTAL_SCROLLBAR_ALWAYS, and HORIZONTAL_SCROLLBAR_NEVER.
int getHorizontalScrollBarPolicy()	
void setViewportBorder(Border) Border getViewportBorder()	Set or get the border around the viewport.This is preferred over setting the border on the component.
boolean isWheelScrollingEnabled()	Set or get whether scrolling occurs in response to the mouse wheel. Mouse-wheel scrolling is enabled by default.

Q5)Explain the JFileChooser class with one method in it.

JFileChooser

1) JFileChooser provides a simple mechanism for the user to choose a file.

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

   of JFileChooser to a container.

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

4) The following code pops up a file chooser for the user's home directory that sees only .jpg and .gif images:

```
JFileChooser chooser = new JFileChooser();

FileNameExtensionFilter filter = new FileNameExtensionFilter("JPG & GIF Images", "jpg",
"gif");

chooser.setFileFilter(filter);
```

```
int returnVal = chooser.showOpenDialog(parent);
if(returnVal ==
JFileChooser.APPROVE_OPTION)
{

System.out.println("You chose to open this file: " + chooser.getSelectedFile().getName());
}
```

Constructor and Description

JFileChooser()
Constructs a JFileChooser pointing to the user's default directory.

JFileChooser(**File** currentDirectory)
Constructs a JFileChooser using the given File as the path.

JFileChooser(**File** currentDirectory, **FileSystemView** fsv)
Constructs a JFileChooser using the given current directory and FileSystemView.

JFileChooser(**FileSystemView** fsv)

Constructs a JFileChooser using the given FileSystemView.

JFileChooser(**String** currentDirectoryPath)
Constructs a JFileChooser using the given path.

JFileChooser(**String** currentDirectoryPath, **FileSystemView** fsv)
Constructs a JFileChooser using the given current directory path and FileSystemView.

Q6)Explain JTree class with its constructors.

1) A JTree is used to display a set of hierarchical data as an outline.

2) A JTree has a 'root node' which is the top-most parent for all nodes in the tree. A node is an item in a tree. A node can have many children nodes. These children nodes themselves can have further children nodes. If a node doesn't have any children node, it is called a leaf node.

3) The leaf node is displayed with a different visual indicator. The nodes with children are displayed with a different visual indicator along with a visual 'handle' which can be used to expand or collapse that node. Expanding a node displays the children and collapsing hides them.

4) A specific node in a tree can be identified either by a TreePath (an object that encapsulates a node and all of its ancestors), or by its display row, where each row in the display area displays one node.

5) An *expanded* node is a non-leaf node (as identified by TreeModel.isLeaf (node) returning false) that will displays its children when all its ancestors

are *expanded*. A *collapsed* node is one which hides them. A *hidden* node is one which is under a collapsed ancestor. All of a *viewable* nodes parents are expanded, but may or may not be displayed. A *displayed* node is both viewable and in the display area, where it can be seen.

The following JTree methods use "visible" to mean "displayed":

- isRootVisible()
- setRootVisible()
- scrollPathToVisible()
- scrollRowToVisible()
- getVisibleRowCount()
- setVisibleRowCount()

The next group of JTree methods use "visible" to mean "viewable" (under an expanded parent):

1) isVisible()
2) makeVisible()

Constructor and Description

JTree()
Returns a JTree with a sample model.

JTree(**Hashtable**<?,?> value)

Returns a JTree created from a Hashtable which does not display with root.

JTree(**Object**[] value)

Returns a JTree with each element of the specified array as the child of a new root node which is not displayed.

JTree(**TreeModel** newModel)

Returns an instance of JTree which displays the root node -- the tree is created using the specified data model.

JTree(**TreeNode** root)
Returns a JTree with the specified TreeNode as its root, which displays the root node.

JTree(**TreeNode** root, boolean asksAllowsChildren)

Returns a JTree with the specified TreeNode as its root, which displays the root node and which decides whether a node is a leaf node in the specified manner.

JTree(**Vector**<?> value)

Returns a JTree with each element of the specified Vector as the child of a new root node which is not displayed.

Example:

```
package   net.codejava.swing;
import     javax.swing.JFrame;
import javax.swing.JTree;

import javax.swing.SwingUtilities;
import javax.swing.tree.DefaultMutableTreeNode;

public class TreeExample extends JFrame

{

   private JTree tree;
   public TreeExample()

   {

      //create the root node

      DefaultMutableTreeNode root = new DefaultMutableTreeNode("Root");
      //create the child nodes

      DefaultMutableTreeNode vegetableNode = new
DefaultMutableTreeNode("Vegetables");

      DefaultMutableTreeNode fruitNode = new DefaultMutableTreeNode("Fruits");
//add the child nodes to the root node root.add(vegetableNode); root.add(fruitNode);

      //create the tree by passing in the root node
      tree = new JTree(root);

      add(tree);

      this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)
      ; this.setTitle("JTree Example");
      this.pack();
      this.setVisible(true);
   }

   public static void main(String[] args)
   {

      SwingUtilities.invokeLater(new Runnable() {
         @Override

         public void run() { new
            TreeExample();

         }
```

```
        });
    }
}
```

Q7)Write a note on the JColorChooser Class.

- JColorChooser provides a pane of controls designed to allow a user to manipulate and select a color.
- A color chooser is a component that you can place anywhere within your program GUI. TheJColorChooser API also makes it easy to bring up a dialog (modal or not) that contains a color chooser.
- This class provides three levels of API:

1. A static convenience method which shows a modal color-chooser dialog and returns the color selected by the user.
2. A static convenience method for creating a color-chooser dialog

   where ActionListeners can be specified to be invoked when the user presses one of the dialog buttons.
3. The ability to create instances of JColorChooser panes directly (within any container). PropertyChange listeners can be added to detect when the current "color" property changes.

Constructor and Description
JColorChooser() Creates a color chooser pane with an initial color of white.
JColorChooser(**Color** initialColor) Creates a color chooser pane with the specified initial color.
JColorChooser(**ColorSelectionModel** model) Creates a color chooser pane with the specified ColorSelectionModel.

Example:

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

public class JColorChooserExample extends JFrame implements ActionListener{ JButton
b;

Container c;
```

```
JColorChooserExample(){

   c=getContentPane(); c.setLayout(new
   FlowLayout());

   b=new JButton("color");
   b.addActionListener(this);

   c.add(b);
}

public void actionPerformed(ActionEvent e) { Color
initialcolor=Color.RED;

Color color=JColorChooser.showDialog(this,"Select a color",initialcolor);
c.setBackground(color);

}

public static void main(String[] args) { JColorChooserExample
   ch=new JColorChooserExample(); ch.setSize(400,400);

   ch.setVisible(true);
   ch.setDefaultCloseOperation(EXIT_ON_CLOSE);

}
}
```

Q8)Short note on following JAVA Swing classes :

(i) JList

- A component that displays a list of objects and allows the user to select one or more items. A separate model, ListModel, maintains the contents of the list.
- A ListModel can be supplied directly to a JList by way of a constructor or

  the setModel method. The contents need not be static - the number of items, and the values of items can change over time.
- A correct ListModel implementation notifies the set

  of javax.swing.event.ListDataListeners that have been added to it, each time a change occurs. These changes are characterized by

  a javax.swing.event.ListDataEvent, which identifies the range of list indices that have been modified, added, or removed.
- JList's ListUI is responsible for keeping the visual representation up to date with changes, by listening to the model.

- Simple, dynamic-content, JList applications can use the DefaultListModel class to maintain list elements. This class implements the ListModel interface and also provides a java.util.Vector-like API. Applications that need a more

  custom ListModel implementation may instead wish to subclass AbstractListModel, which provides basic support for managing and notifying listeners.

Constructor and Description

JList()
Constructs a JList with an empty, read-only, model.

JList(**E**[] listData)
Constructs a JList that displays the elements in the specified array.

JList(**ListModel**<**E**> dataModel)
Constructs a JList that displays elements from the specified, non-null, model.

JList(**Vector**<? extends **E**> listData)
Constructs a JList that displays the elements in the specified Vector.

Example:

```
import java.awt.FlowLayout;
import javax.swing.JFrame;
import javax.swing.JList; import
javax.swing.JScrollPane;

public class JListTest {
 public static void main(String[] args) {

   JFrame.setDefaultLookAndFeelDecorated(true);
   JFrame frame = new JFrame("JList Test");
   frame.setLayout(new FlowLayout());

   frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  String[] selections = { "green", "red", "orange", "dark blue" };
  JList list = new JList(selections);

  list.setSelectedIndex(1);

  System.out.println(list.getSelectedValue());
  frame.add(new JScrollPane(list));
  frame.pack();

  frame.setVisible(true);
 }
```

}

(ii) JTabbedPane.

- JTabbedPane is a container component that lets the user switch between pages by clicking on a tab.
- With the JTabbedPane class, you can have several components, such as panels, share the same space. The user chooses which component to view by selecting the tab corresponding to the desired component. If you want similar functionality without the tab interface, you can use a card layout instead of a tabbed pane.
- To create a tabbed pane, instantiate JTabbedPane, create the components you wish it to display, and then add the components to the tabbed pane using

  the addTab method.
- The default tab placement is set to the TOP location. You can change the tab placement to LEFT, RIGHT, TOP or BOTTOM by using the setTabPlacement method.

There are three ways to switch to specific tabs using GUI.

1. **Using a mouse.** To switch to a specific tab, the user clicks it with the mouse.
2. **Using keyboard arrows.** When the JTabbedPane object has the focus, the keyboard arrows can be used to switch from tab to tab.
3. **Using key mnemonics.** The setMnemonicAt method allows the user to switch to a specific tab using the keyboard. For example, setMnemonicAt(3,

   KeyEvent.VK_4) makes '4' the mnemonic for the fourth tab (which is at index 3, since the indices start with 0); pressing Alt-4 makes the fourth tab's component appear. Often, a mnemonic uses a character in the tab's title that is then automatically underlined.

Creating and Setting Up a Tabbed Pane

Method or Constructor	Purpose
JTabbedPane()	Creates a tabbed pane. The first optional argument specifies where the tabs should appear. By default, the tabs appear at the
JTabbedPane(int)	

JTabbedPane(int, int)	top of the tabbed pane. You can specify these positions (defined in the SwingConstants interface, which JTabbedPane implements): TOP, BOTTOM, LEFT, RIGHT. The second optional argument specifies the tab layout policy. You can specify one of these policies (defined in JTabbedPane): WRAP_TAB_LAYOUT or SCROLL_TAB_LAYOUT.
addTab(String, Icon, Component, String) addTab(String, Icon, Component) addTab(String, Component)	Adds a new tab to the tabbed pane. The first argument specifies the text on the tab. The optional icon argument specifies the tab's icon. The component argument specifies the component that the tabbed pane should show when the tab is selected. The fourth argument, if present, specifies the tool tip text for the tab.
void setTabLayoutPolicy(int) int getTabLayoutPolicy()	Sets or gets the policy that the tabbed pane uses in laying out tabs when all tabs do not fit within a single run. Possible values are WRAP_TAB_LAYOUT andSCROLL_TAB_LAYOUT. The default policy is WRAP_TAB_LAYOUT.
void setTabPlacement(int) int getTabPlacement()	Sets or gets the location where the tabs appear relative to the content. Possible values (defined in SwingConstants, which is implemented by JTabbedPane) areTOP, BOTTOM, LEFT, and RIGHT.

Q9)Explain creation of menus.

1) A menu provides a space-saving way to let the user choose one of several options.

2) Menus are unique in that, by convention, they aren't placed with the other components in the UI. Instead, a menu usually appears either in a *menu bar* or as a *popup menu*.

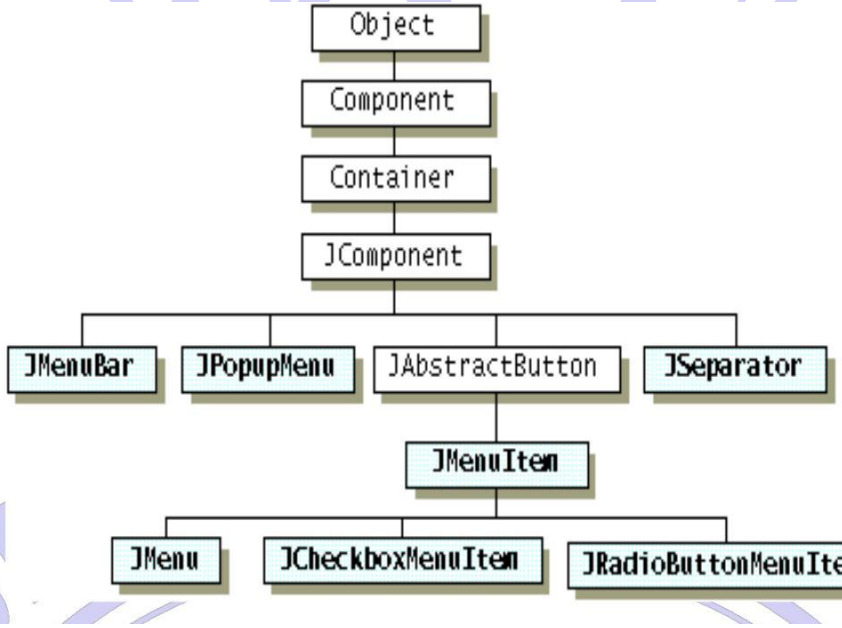
3) A menu bar contains one or more menus and has a customary, platform-dependent location — usually along the top of a window.

4) A popup menu is a menu that is invisible until the user makes a platform-specific mouse action, such as pressing the right mouse button, over a popup-enabled

component. The popup menu then appears under the cursor.

The Menu Component Hierarchy

Here is a picture of the inheritance hierarchy for the menu-related classes:



Constructor or Method	Purpose

JMenuBar()	Creates a menu bar.
JMenu add(JMenu)	Adds the menu to the end of the menu bar.
void setJMenuBar(JMenuBar) JMenuBar getJMenuBar() *(in JApplet, JDialog, JFrame, JInternalFrame, JRootPane)*	Sets or gets the menu bar of an applet, dialog, frame, internal frame, or root pane.

Example:

```
package com.tutorialspoint.gui;

import java.awt.*;
import java.awt.event.*;

public class SwingMenuDemo {
  private JFrame mainFrame;
  private JLabel headerLabel;
  private JLabel statusLabel;
  private JPanel controlPanel;

  public SwingMenuDemo(){
    prepareGUI();
  }

  public static void main(String[] args){
SwingMenuDemo swingMenuDemo = new SwingMenuDemo();
swingMenuDemo.showMenuDemo();
  }
  private void prepareGUI(){

    mainFrame = new JFrame("Java SWING Examples");
    mainFrame.setSize(400,400);
    mainFrame.setLayout(new GridLayout(3, 1));

    headerLabel = new JLabel("",JLabel.CENTER );
    statusLabel = new JLabel("",JLabel.CENTER);
```

```
    statusLabel.setSize(350,100);
    mainFrame.addWindowListener(new WindowAdapter() {

      public void windowClosing(WindowEvent windowEvent){
        System.exit(0);

      }
    });

    controlPanel = new JPanel();
    controlPanel.setLayout(new FlowLayout());

    mainFrame.add(headerLabel);

    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);
  }

  private void showMenuDemo(){
    //create a menu baral
    JMenuBar menuBar = new
    JMenuBar();

    //create menus
    JMenu fileMenu = new JMenu("File");
    JMenu editMenu = new JMenu("Edit");

    final JMenu aboutMenu = new JMenu("About");
    final JMenu linkMenu = new JMenu("Links");

    //create menu items

    JMenuItem newMenuItem = new JMenuItem("New");
    newMenuItem.setMnemonic(KeyEvent.VK_N);
    newMenuItem.setActionCommand("New");

    JMenuItem openMenuItem = new JMenuItem("Open");
    openMenuItem.setActionCommand("Open");
```

```
JMenuItem saveMenuItem = new JMenuItem("Save");
saveMenuItem.setActionCommand("Save");

    JMenuItem exitMenuItem = new JMenuItem("Exit");
    exitMenuItem.setActionCommand("Exit");

    JMenuItem cutMenuItem = new JMenuItem("Cut");
    cutMenuItem.setActionCommand("Cut");

    JMenuItem copyMenuItem = new JMenuItem("Copy");
    copyMenuItem.setActionCommand("Copy");

    JMenuItem pasteMenuItem = new JMenuItem("Paste");
    pasteMenuItem.setActionCommand("Paste");

    MenuItemListener menuItemListener = new MenuItemListener();

    newMenuItem.addActionListener(menuItemListener);
    openMenuItem.addActionListener(menuItemListener);
    saveMenuItem.addActionListener(menuItemListener);
    exitMenuItem.addActionListener(menuItemListener);

    cutMenuItem.addActionListener(menuItemListener);
    copyMenuItem.addActionListener(menuItemListener);
    pasteMenuItem.addActionListener(menuItemListener);

    final JCheckBoxMenuItem showWindowMenu = new JCheckBoxMenuItem("Show
About", true);

    showWindowMenu.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {

        if(showWindowMenu.getState()){
          menuBar.add(aboutMenu);
        }else{
          menuBar.remove(aboutMenu);
        }

      }
    });

    final JRadioButtonMenuItem showLinksMenu = new
      JRadioButtonMenuItem("Show Links", true);

    showLinksMenu.addItemListener(new ItemListener() {
      public void itemStateChanged(ItemEvent e) {

        if(menuBar.getMenu(3)!= null){
          menuBar.remove(linkMenu);
```

```
            mainFrame.repaint();

          }else{
            menuBar.add(linkMenu);
            mainFrame.repaint();
          }

        }
      });

      //add menu items to menus fileMenu.add(newMenuItem);
      fileMenu.add(openMenuItem); fileMenu.add(saveMenuItem);
      fileMenu.addSeparator(); fileMenu.add(showWindowMenu);
      fileMenu.addSeparator(); fileMenu.add(showLinksMenu);
      fileMenu.addSeparator(); fileMenu.add(exitMenuItem);
      editMenu.add(cutMenuItem); editMenu.add(copyMenuItem);
      editMenu.add(pasteMenuItem);

      //add menu to menubar menuBar.add(fileMenu);
      menuBar.add(editMenu); menuBar.add(aboutMenu);
      menuBar.add(linkMenu);

      //add menubar to the frame mainFrame.setJMenuBar(menuBar);
      mainFrame.setVisible(true);
    }

    class MenuItemListener implements ActionListener { public void
      actionPerformed(ActionEvent e) {

        statusLabel.setText(e.getActionCommand() + " JMenuItem clicked.");

      }
    }

}
```

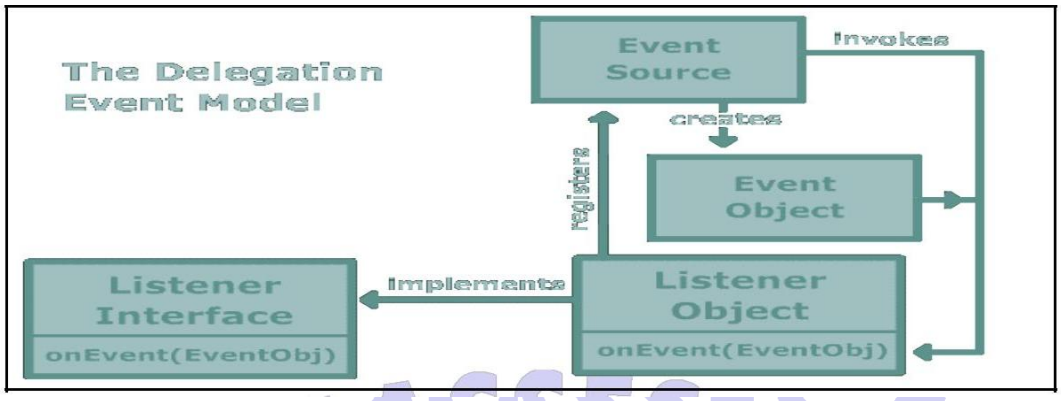
Q10)How are Swing component different from AWT components?

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**.

Q11)EXPLAIN DELIGATION EVENT 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.
- 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.



Event source

- registers/unregisters event listener objects
- broadcasts events to listeners (by invoking methods on them)
- pushes an "event object" to listeners (by passing it to the event methods)
- corresponding Observer pattern participant: Observable

Event object

- describes the event
- provides methods to access information about the event (e.g., *MouseEvent.getX()*)
- passed to listener by event source
- passed to listener by event source
- corresponding Observer pattern participant: *Observer.update()*method's infoObj parameter

Event listener interface

- defines agreed upon event methods (the event source invokes these on the listener object)
- is implemented by event listener
- corresponding Observer pattern participant: *Observer.update()* method

Event listener object

- registers with event source to receive events
- reacts to events broadcast by event source
- implements methods in event listener interface
- corresponding Observer pattern participant: Observer

Q12)What is swing? Explain its features?

Swing API is set of extensible GUI Components to ease developer's life to create JAVA based Front End/ GUI Applications. It is build upon top of AWT API and acts as replacement of AWT API as it has almost every control corresponding to AWT controls.

Swing component follows a Model-View-Controller architecture to fulfill the following criteria.

- A single API is to be sufficient to support multiple look and feel.
- API is to model driven so that highest level API is not required to have the data.
- API is to use the Java Bean model so that Builder Tools and IDE can provide better services to the developers to use it.

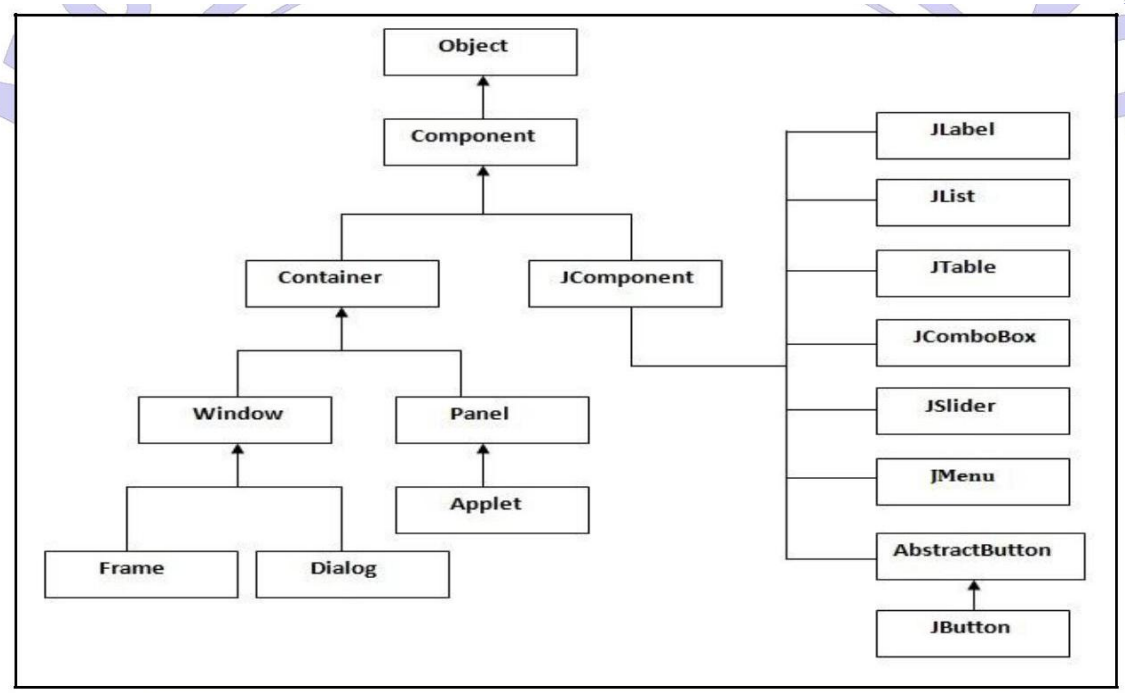
Unlike AWT, Java Swing provides platform-independent and lightweight components.

The javax.swing package provides classes for java swing API such as JButton, JTextField, JTextArea, JRadioButton, JCheckbox, JMenu, JColorChooser etc.

Swing features:

- **Light Weight** - Swing component are independent of native Operating System's API as Swing API controls are rendered mostly using pure JAVA code instead of underlying operating system calls.
- **Rich controls** - Swing provides a rich set of advanced controls like Tree, TabbedPane, slider, colorpicker, table controls
- **Highly Customizable** - Swing controls can be customized in very easy way as visual apperance is independent of internal representation.
- **Pluggable look-and-feel**- SWING based GUI Application look and feel can be changed at run time based on available values.

Hierarchy of Java Swing classes:



Example of swings:-

```
import javax.swing.*;

public class FirstSwingExample
{
public static void main(String[] args)
    {

    JFrame f=new JFrame();//creating instance of JFrame JButton
    b=new JButton("click");//creating instance of JButton

    b.setBounds(130,100,100, 40);//x axis, y axis, width, height
    f.add(b);//adding button in JFrame f.setSize(400,500);//400
    width and 500 height f.setLayout(null);//using no layout
    managers f.setVisible(true);//making the frame visible

    }

}
```

Q13)Short note on JTable.

JTable:-

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

- JTables are typically placed inside of a JScrollPane. By default, a JTable will adjust its width such that a horizontal scrollbar is unnecessary. To allow for a horizontal scrollbar, invoke setAutoResizeMode(int) with AUTO_RESIZE_OFF.

- To enable sorting and filtering of rows, use a RowSorter. You can set up a row sorter in either of two ways:
  - Directly set the RowSorter. For example: table.setRowSorter(new TableRowSorter(model)).
  - Set the autoCreateRowSorter property to true, so that the JTable creates a RowSorter for you. For example: setAutoCreateRowSorter(true).

Constructor and Description

JTable()

Constructs a default JTable that is initialized with a default data model, a default column model, and a default selection model.

JTable(int numRows, int numColumns)
Constructs a JTable with numRows and numColumns of empty cells using DefaultTableModel.

JTable(**Object**[][] rowData, **Object**[] columnNames)

Constructs a JTable to display the values in the two dimensional array, rowData, with column names, columnNames.

JTable(**TableModel** dm)

Constructs a JTable that is initialized with dm as the data model, a default column model, and a default selection model.

JTable(**TableModel** dm, **TableColumnModel** cm)

Constructs a JTable that is initialized with dm as the data model, cm as the column model, and a default selection model.

JTable(**TableModel** dm, **TableColumnModel** cm, **ListSelectionModel** sm)

Constructs a JTable that is initialized with dm as the data model, cm as the column model,

and sm as the selection model.

JTable(**Vector** rowData, **Vector** columnNames)

Constructs a JTable to display the values in the Vector of Vectors, rowData, with column names, columnNames.

Q13)Short note on JFrame.

The class **JFrame** is an extended version of java.awt.Frame that adds support for the JFC/Swing component architecture.

Class declaration

Following is the declaration for **javax.swing.JFrame** class:

```
public class JFrame
  extends Frame
    implements WindowConstants, Accessible, RootPaneContainer
```

Class constructors

S.N.	Constructor & Description
1	**JFrame()** Constructs a new frame that is initially invisible.
2	**JFrame(GraphicsConfiguration gc)** Creates a Frame in the specified GraphicsConfiguration of a screen device and a blank title.
3	**JFrame(String title)** Creates a new, initially invisible Frame with the specified title.
4	**JFrame(String title, GraphicsConfiguration gc)** Creates a JFrame with the specified title and the specified

	GraphicsConfiguration of a screen device.

Class methods

Component getGlassPane()

Returns the glassPane object for this frame.

Graphics getGraphics()

Creates a graphics context for this component.

JMenuBar getJMenuBar()

Returns the menubar set on this frame.

JLayeredPane getLayeredPane()

Returns the layeredPane object for this frame.

JRootPane getRootPane()

Returns the rootPane object for this frame.

EXAMPLE:

```
package tutorialspoint.gui;com.

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

public class SwingContainerDemo { private JFrame mainFrame; private JLabel
  headerLabel; private JLabel statusLabel; private JPanel controlPanel; private
  JLabel msglabel;
```

```
public SwingContainerDemo(){
  prepareGUI();

}


public static void main(String[] args){

  SwingContainerDemo swingContainerDemo = new SwingContainerDemo();
  swingContainerDemo.showJFrameDemo();

}


private void prepareGUI(){

  mainFrame = new JFrame("Java Swing Examples");
  mainFrame.setSize(400,400); mainFrame.setLayout(new
  GridLayout(3, 1)); mainFrame.addWindowListener(new
  WindowAdapter() {

   public void windowClosing(WindowEvent windowEvent){
    System.exit(0);

    }

  });

  headerLabel = new JLabel("", JLabel.CENTER);
  statusLabel = new JLabel("",JLabel.CENTER);


  statusLabel.setSize(350,100);


  msglabel = new JLabel("Welcome to TutorialsPoint SWING Tutorial.", JLabel.CENTER);


  controlPanel = new JPanel();
  controlPanel.setLayout(new FlowLayout());


  mainFrame.add(headerLabel);
```

```
    mainFrame.add(controlPanel);
    mainFrame.add(statusLabel);
    mainFrame.setVisible(true);

  }


  private void showJFrameDemo(){
    headerLabel.setText("Container in action: JFrame");

    final JFrame frame = new JFrame();
    frame.setSize(300, 300);
    frame.setLayout(new FlowLayout());
    frame.add(msglabel);

    frame.addWindowListener(new WindowAdapter() { public void
      windowClosing(WindowEvent windowEvent){

        frame.dispose();

      }

    });

    JButton okButton = new JButton("Open a Frame");
    okButton.addActionListener(new ActionListener() {

      public void actionPerformed(ActionEvent e) {
        statusLabel.setText("A Frame shown to the user.");
        frame.setVisible(true);

      }
    });
    controlPanel.add(okButton);
    mainFrame.setVisible(true);

  }

}
```

Q14)How to create dynamic JTable? Explain with an example.

1. Create table model with given column name and zero rows.
2. Create a JTable instance with table model as parameter instance.
3. The dynamically adding rows can be done by using addRows() method.

Example:

```
import java.awt.BorderLayout; import
java.awt.event.WindowAdapter; import
java.awt.event.WindowEvent;


import javax.swing.JFrame; import
javax.swing.JScrollPane; import
javax.swing.JTable;

import javax.swing.table.DefaultTableModel;


public class Main extends JFrame {

 DefaultTableModel model = new DefaultTableModel(new Object[][] {

    { "some", "text" }, { "any", "text" }, { "even", "more" },

    { "text", "strings" }, { "and", "other" }, { "text", "values" } },

    new Object[] { "Column 1", "Column 2" });


 public Main() {
  setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

  JTable table = new JTable(model);

  getContentPane().add(new JScrollPane(table), BorderLayout.CENTER);

  pack();

 }


 public static void main(String arg[]) {
```

```
    new Main().setVisible(true);
  }
}
```

Q15. Write short note on MVC.
ANSWER

MVC Pattern stands for Model-View-Controller Pattern. This pattern is used to separate application's concerns.

- **Model** - Model represents an object or JAVA POJO carrying data. It can also have logic to update controller if its data changes.
- **View** - View represents the visualization of the data that model contains.
- **Controller** - Controller acts on both model and view. It controls the data flow into model object and updates the view whenever data changes. It keeps view and model separate.

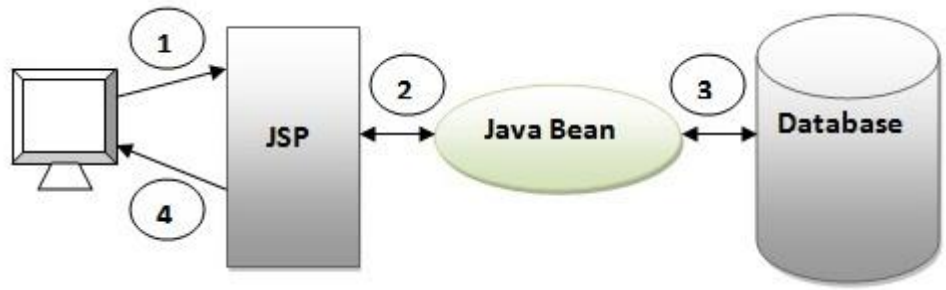
Model 1 Architecture

Servlet and JSP are the main technologies to develop the web applications.

Servlet was considered superior to CGI. Servlet technology doesn't create process, rather it creates thread to handle request. The advantage of creating thread over process is that it doesn't allocate separate memory area. Thus many subsequent requests can be easily handled by servlet.

Problem in Servlet technology Servlet needs to recompile if any designing code is modified. It doesn't provide separation of concern. Presentation and Business logic are mixed up.

JSP overcomes almost all the problems of Servlet. It provides better separation of concern, now presentation and business logic can be easily separated. You don't need to redeploy the application if JSP page is modified. JSP provides support to develop web application using JavaBean, custom tags and JSTL so that we can put the business logic separate from our JSP that will be easier to test and debug.



As you can see in the above figure, there is picture which show the flow of the model1 architecture.

1. Browser sends request for the JSP page
2. JSP accesses Java Bean and invokes business logic
3. Java Bean connects to the database and get/save data
4. Response is sent to the browser which is generated by JSP

Advantage of Model 1 Architecture

- Easy and Quick to develop web application

Disadvantage of Model 1 Architecture

- **Navigation control is decentralized** since every page contains the logic to determine the next page. If JSP page name is changed that is referred by other pages, we need to change it in all the pages that leads to the maintenance problem.
- **Time consuming** You need to spend more time to develop custom tags in JSP. So that we don't need to use scriptlet tag.
- **Hard to extend** It is better for small applications but not for large applications.

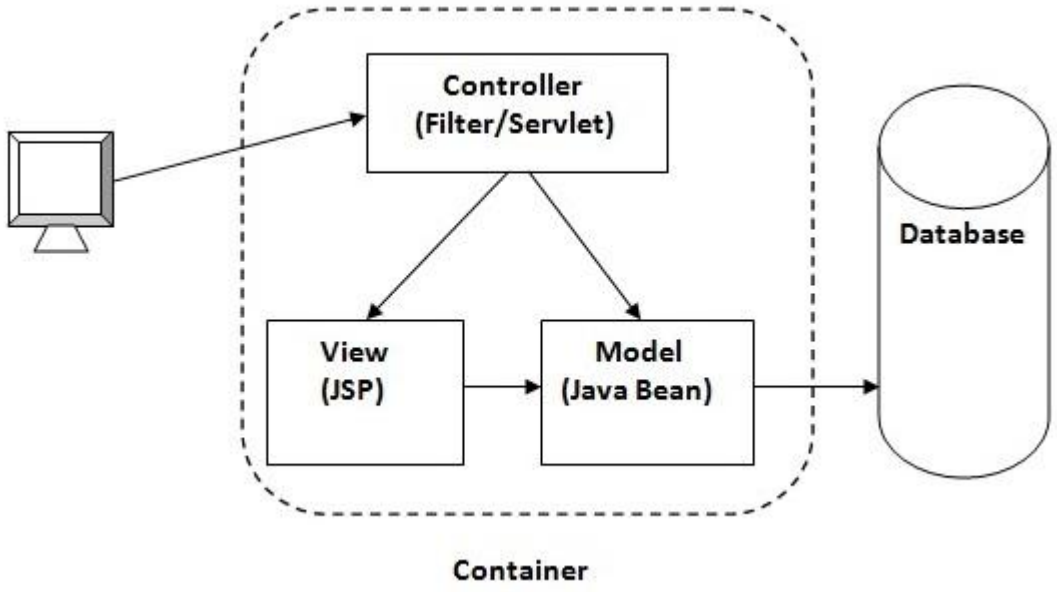
Model 2 (MVC) Architecture

Model 2 is based on the MVC (Model View Controller) design pattern. The MVC design pattern consists of three modules model, view and controller.

Model The model represents the state (data) and business logic of the application.

View The view module is responsible to display data i.e. it represents the presentation.

Controller The controller module acts as an interface between view and model. It intercepts all the requests i.e. receives input and commands to Model / View to change accordingly.



Advantage of Model 2 (MVC) Architecture

- **Navigation control is centralized** Now only controller contains the logic to determine the next page.
- **Easy to maintain**
- **Easy to extend**
- **Easy to test**
- **Better separation of concerns**

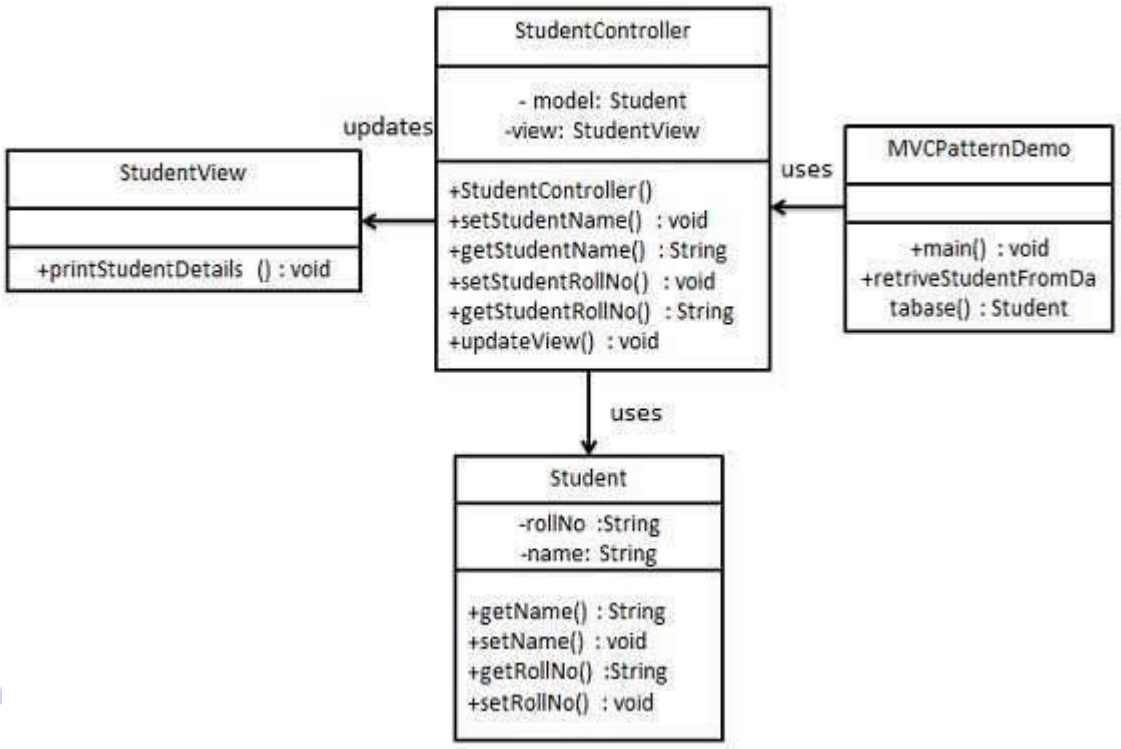
Disadvantage of Model 2 (MVC) Architecture

- We need to write the controller code self. If we change the controller code, we need to recompile the class and redeploy the application.

Implementation

We are going to create a Student object acting as a model.StudentView will be a view class which can print student details on console and StudentController is the controller class responsible to store data in Student object and update viewStudentView accordingly.

MVCPatternDemo, our demo class, will use StudentController to demonstrate use of MVC pattern.



Q16. Write short note on following components –

I. JCheckBox

ANSWER

Introduction

The class **JCheckBox** is an implementation of a check box - an item that can be selected or deselected, and which displays its state to the user.

Class declaration

Following is the declaration for **javax.swing.JCheckBox** class −

```
public class JCheckBox
  extends JToggleButton
    implements Accessible
```

Field

Following are the fields for **javax.swing.JCheckBox** class −

- **static String BORDER_PAINTED_FLAT_CHANGED_PROPERTY** − Identifies a change to the flat property.

Class constructors

S.N.	Constructor & Description

1	**JCheckBox()** Creates an initially unselected check box button with no text, no icon.
2	**JCheckBox(Action a)** Creates a check box where properties are taken from the Action supplied.
3	**JCheckBox(Icon icon)** Creates an initially unselected check box with an icon.
4	**JCheckBox(Icon icon, boolean selected)** Creates a check box with an icon and specifies whether or not it is initially selected.
5	**JCheckBox(String text)** Creates an initially unselected check box with text.
6	**JCheckBox(String text, boolean selected)** Creates a check box with text and specifies whether or not it is initially selected.
7	**JCheckBox(String text, Icon icon)** Creates an initially unselected check box with the specified text and icon.
8	**JCheckBox(String text, Icon icon, boolean selected)** Creates a check box with text and icon, and specifies whether or not it is initially selected.

Class methods

S.N.	Method & Description
1	**AccessibleContext getAccessibleContext()** Gets the AccessibleContext associated with this JCheckBox.
2	**String getUIClassID()** Returns a string that specifies the name of the L&F class that renders this component.

3	**boolean isBorderPaintedFlat()** Gets the value of the borderPaintedFlat property.
4	**protected String paramString()** Returns a string representation of this JCheckBox.
5	**void setBorderPaintedFlat(boolean b)** Sets the borderPaintedFlat property, which gives a hint to the look and feel as to the appearance of the check box border.
6	**void updateUI()** Resets the UI property to a value from the current look and feel.

Methods inherited

This class inherits methods from the following classes:

- javax.swing.AbstractButton
- javax.swing.JToggleButton
- javax.swing.JComponent
- java.awt.Container
- java.awt.Component
- java.lang.Object

JCheckBox Example

SwingControlDemo.java

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

public class SwingControlDemo {

   private JFrame mainFrame;
   private JLabel headerLabel;
   private JLabel statusLabel;
   private JPanel controlPanel;
```

```
public SwingControlDemo(){
   prepareGUI();
}

public static void main(String[] args){
   SwingControlDemo  swingControlDemo = new SwingControlDemo();
   swingControlDemo.showCheckBoxDemo();
}

private void prepareGUI(){
   mainFrame = new JFrame("Java Swing Examples");
   mainFrame.setSize(400,400);
   mainFrame.setLayout(new GridLayout(3, 1));
   mainFrame.addWindowListener(new WindowAdapter() {
      public void windowClosing(WindowEvent windowEvent){
         System.exit(0);
      }
   });
   headerLabel = new JLabel("", JLabel.CENTER);
   statusLabel = new JLabel("",JLabel.CENTER);

   statusLabel.setSize(350,100);

   controlPanel = new JPanel();
   controlPanel.setLayout(new FlowLayout());

   mainFrame.add(headerLabel);
   mainFrame.add(controlPanel);
   mainFrame.add(statusLabel);
   mainFrame.setVisible(true);
}
```

```
private void showCheckBoxDemo(){

  headerLabel.setText("Control in action: CheckBox");

  final JCheckBox chkApple = new JCheckBox("Apple");
  final JCheckBox chkMango = new JCheckBox("Mango");
  final JCheckBox chkPeer = new JCheckBox("Peer");

  chkApple.setMnemonic(KeyEvent.VK_C);
  chkMango.setMnemonic(KeyEvent.VK_M);
  chkPeer.setMnemonic(KeyEvent.VK_P);

  chkApple.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
      statusLabel.setText("Apple Checkbox: "
      + (e.getStateChange()==1?"checked":"unchecked"));
    }
  });

  chkMango.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
      statusLabel.setText("Mango Checkbox: "
      + (e.getStateChange()==1?"checked":"unchecked"));
    }
  });

  chkPeer.addItemListener(new ItemListener() {
    public void itemStateChanged(ItemEvent e) {
      statusLabel.setText("Peer Checkbox: "
      + (e.getStateChange()==1?"checked":"unchecked"));
    }
  });
```

```
    controlPanel.add(chkApple);
    controlPanel.add(chkMango);
    controlPanel.add(chkPeer);

    mainFrame.setVisible(true);
  }
}
```

II. JRadioButton

ANSWER

The JRadioButton class is used to create a radio button. It is used to choose one option from multiple options. It is widely used in exam systems or quiz.

It should be added in ButtonGroup to select one radio button only.

Commonly used Constructors of JRadioButton class:

- **JRadioButton():** creates an unselected radio button with no text.
- **JRadioButton(String s):** creates an unselected radio button with specified text.
- **JRadioButton(String s, boolean selected):** creates a radio button with the specified text and selected status.

Commonly used Methods of AbstractButton class:

1) public void setText(String s): is used to set specified text on button.

2) public String getText(): is used to return the text of the button.

3) public void setEnabled(boolean b): is used to enable or disable the button.

4) public void setIcon(Icon b): is used to set the specified Icon on the button.

5) public Icon getIcon(): is used to get the Icon of the button.

6) public void setMnemonic(int a): is used to set the mnemonic on the button.

7) public void addActionListener(ActionListener a): is used to add the action listener to this object.

Example of JRadioButton class:

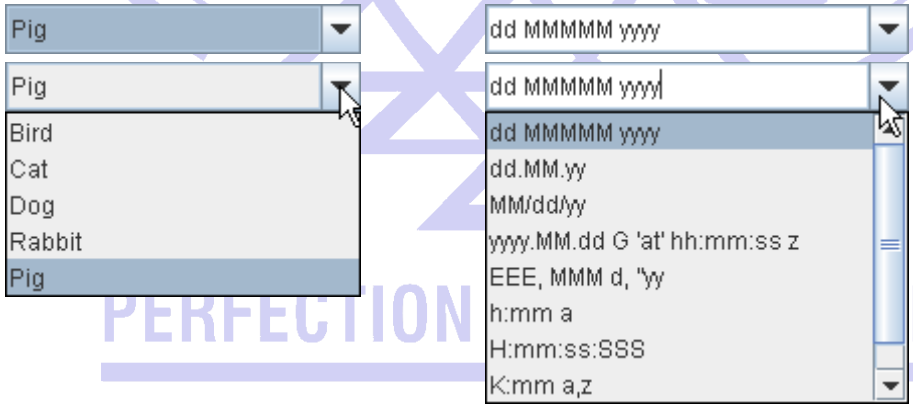
```
import javax.swing.*;
public class Radio {
JFrame f;

Radio(){
f=new JFrame();

JRadioButton r1=new JRadioButton("A) Male");
```

```
JRadioButton r2=new JRadioButton("B) FeMale");
r1.setBounds(50,100,70,30);
r2.setBounds(50,150,70,30);

ButtonGroup bg=new ButtonGroup();
bg.add(r1);bg.add(r2);

f.add(r1);f.add(r2);

f.setSize(300,300);
f.setLayout(null);
f.setVisible(true);
}
public static void main(String[] args) {
   new Radio();
}
}
```

III. JComboX

ANSWER

A JComboBox, which lets the user choose one of several choices, can have two very different forms. The default form is the uneditable combo box, which features a button and a drop-down list of values. The second form, called the editable combo box, features a text field with a small button abutting it. The user can type a value in the text field or click the button to display a drop-down list. Here's what the two forms of combo boxes look like in the Java look and feel:



Uneditable combo box, before (top) and after the button is clicked

Editable combo box, before and after the arrow button is clicked

Combo boxes require little screen space, and their editable (text field) form is useful for letting the user quickly choose a value without limiting the user to the displayed values. Other components that can display one-of-many choices are groups of radio buttons and lists. Groups of radio buttons are generally the easiest for users to understand, but combo boxes can be more appropriate when space is limited or more than a few choices are available. Lists are not

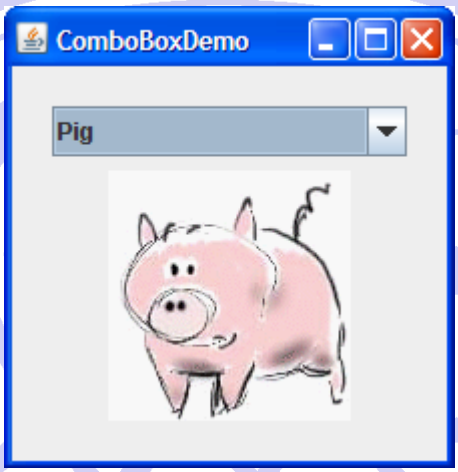
terribly attractive, but they're more appropriate than combo boxes when the number of items is large (say, over 20) or when selecting multiple items might be valid.

Because editable and uneditable combo boxes are so different, this section treats them separately. This section covers these topics:

- Using an Uneditable Combo Box
- Handling Events on a Combo Box
- Using an Editable Combo Box
- Providing a Custom Renderer
- The Combo Box API

Using an Uneditable Combo Box

The application shown here uses an uneditable combo box for choosing a pet picture:




```
String[] petStrings = { "Bird", "Cat", "Dog", "Rabbit", "Pig" };

//Create the combo box, select item at index 4.
//Indices start at 0, so 4 specifies the pig.
JComboBox petList = new JComboBox(petStrings);
petList.setSelectedIndex(4);
petList.addActionListener(this);
```

This combo box contains an array of strings, but you could just as easily use icons instead. To put anything else into a combo box or to customize how the items in a combo box look, you need to write a custom renderer. An editable combo box would also need a custom editor. Refer to Providing a Custom Renderer for information and an example.

The preceding code registers an action listener on the combo box. To see the action listener implementation and learn about other types of listeners supported by combo box, refer to Handling Events on a Combo Box.

No matter which constructor you use, a combo box uses a combo box model to contain and manage the items in its menu. When you initialize a combo box with an array or a vector, the

combo box creates a default model object for you. As with other Swing components, you can customize a combo box in part by implementing a custom model — an object that implements theComboBoxModel interface.

Handling Events on a Combo Box

```
public class ComboBoxDemo ... implements ActionListener {
    . . .
        petList.addActionListener(this) {
    . . .
    public void actionPerformed(ActionEvent e) {
        JComboBox cb = (JComboBox)e.getSource();
        String petName = (String)cb.getSelectedItem();
        updateLabel(petName);
    }
    . . .
}
```

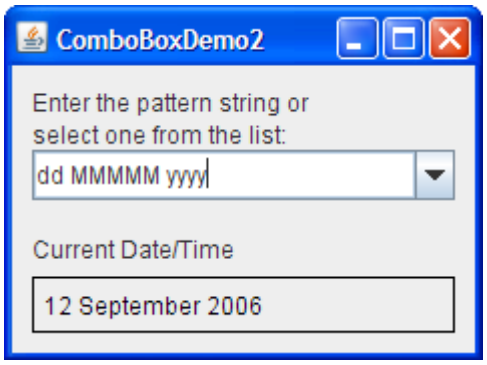
This action listener gets the newly selected item from the combo box, uses it to compute the name of an image file, and updates a label to display the image. The combo box fires an action event when the user selects an item from the combo box's menu. See How to Write an Action Listener, for general information about implementing action listeners.

Combo boxes also generate item events, which are fired when any of the items' selection state changes. Only one item at a time can be selected in a combo box, so when the user makes a new selection the previously selected item becomes unselected. Thus two item events are fired each time the user selects a different item from the menu. If the user chooses the same item, no item events are fired. Use addItemListener to register an item listener on a combo box. How to Write an Item Listener gives general information about implementing item listeners.

Although JComboBox inherits methods to register listeners for low-level events — focus, key, and mouse events, for example — we recommend that you don't listen for low-level events on a combo box. Here's why: A combo box is a *compound component* — it is comprised of two or more other components. The combo box itself fires high-level events such as action events. Its subcomponents fire low-level events such as mouse, key, and focus events. The low-level events and the subcomponent that fires them are look-and-feel-dependent. To avoid writing look-and-feel-dependent code, you should listen only for high-level events on a compound component such as a combo box. For information about events, including a discussion about high- and low-level events, refer to Writing Event Listeners.

Using an Editable Combo Box

Here's a picture of a demo application that uses an editable combo box to enter a pattern with which to format dates.




```
String[] patternExamples = {
        "dd MMMMM yyyy",
        "dd.MM.yy",
        "MM/dd/yy",
        "yyyy.MM.dd G 'at' hh:mm:ss z",
        "EEE, MMM d, "yy",
        "h:mm a",
        "H:mm:ss:SSS",
        "K:mm a,z",
        "yyyy.MMMMM.dd GGG hh:mm aaa"
};
. . .
JComboBox patternList = new JComboBox(patternExamples);
patternList.setEditable(true);
patternList.addActionListener(this);
```

This code is very similar to the previous example, but warrants a few words of explanation. The bold line of code explicitly turns on editing to allow the user to type values in. This is necessary because, by default, a combo box is not editable. This particular example allows editing on the combo box because its menu does not provide all possible date formatting patterns, just shortcuts to frequently used patterns.

An editable combo box fires an action event when the user chooses an item from the menu and when the user types Enter. Note that the menu remains unchanged when the user enters a value into the combo box. If you want, you can easily write an action listener that adds a new item to the combo box's menu each time the user types in a unique value.

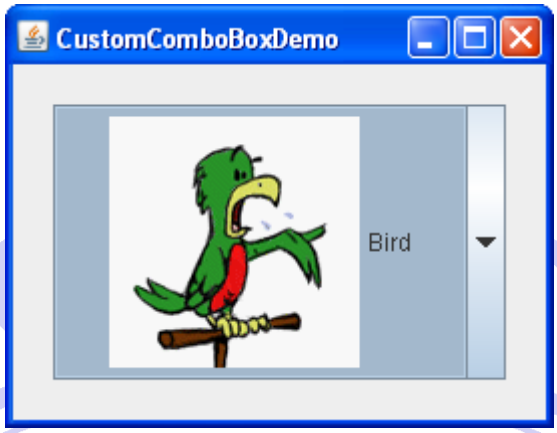
Providing a Custom Renderer

A combo box uses a *renderer* to display each item in its menu. If the combo box is uneditable, it also uses the renderer to display the currently selected item. An editable combo box, on the other hand, uses an *editor* to display the selected item. A renderer for a combo box must implement the ListCellRenderer interface. A combo box's editor must implementComboBoxEditor. This section shows how to provide a custom renderer for an uneditable combo box.

The default renderer knows how to render strings and icons. If you put other objects in a combo box, the default renderer calls the toString method to provide a string to display. You can

customize the way a combo box renders itself and its items by implementing your own ListCellRenderer.

Here's a picture of an application that uses a combo box with a custom renderer:




```
JComboBox petList = new JComboBox(intArray);
. . .
ComboBoxRenderer renderer = new ComboBoxRenderer();
renderer.setPreferredSize(new Dimension(200, 130));
petList.setRenderer(renderer);
petList.setMaximumRowCount(3);
```

The last line sets the combo box's maximum row count, which determines the number of items visible when the menu is displayed. If the number of items in the combo box is larger than its maximum row count, the menu has a scroll bar. The icons are pretty big for a menu, so our code limits the number of rows to 3. Here's the implementation of ComboBoxRenderer, a renderer that puts an icon and text side-by-side:

```
class ComboBoxRenderer extends JLabel
                        implements ListCellRenderer {
    . . .
    public ComboBoxRenderer() {
        setOpaque(true);
        setHorizontalAlignment(CENTER);
        setVerticalAlignment(CENTER);
    }

    /*
     * This method finds the image and text corresponding
     * to the selected value and returns the label, set up
     * to display the text and image.
     */
    public Component getListCellRendererComponent(
                                    JList list,
                                    Object value,
                                    int index,
```

```
                                boolean isSelected,
                                boolean cellHasFocus) {
        //Get the selected index. (The index param isn't
        //always valid, so just use the value.)
        int selectedIndex = ((Integer)value).intValue();

        if (isSelected) {
            setBackground(list.getSelectionBackground());
            setForeground(list.getSelectionForeground());
        } else {
            setBackground(list.getBackground());
            setForeground(list.getForeground());
        }

        //Set the icon and text.  If icon was null, say so.
        ImageIcon icon = images[selectedIndex];
        String pet = petStrings[selectedIndex];
        setIcon(icon);
        if (icon != null) {
            setText(pet);
            setFont(list.getFont());
        } else {
            setUhOhText(pet + " (no image available)",
                        list.getFont());
        }

        return this;
    }
    . . .
}
```

As a ListCellRenderer, ComboBoxRenderer implements a method called getListCellRendererComponent, which returns a component whose paintComponent method is used to display the combo box and each of its items. The easiest way to display an image and an icon is to use a label. So ComboBoxRenderer is a subclass of label and returns itself. The implementation of getListCellRendererComponent configures the renderer to display the currently selected icon and its description.

These arguments are passed to getListCellRendererComponent:

- JList list — a list object used behind the scenes to display the items. The example uses this object's colors to set up foreground and background colors.
- Object value — the object to render. An Integer in this example.
- int index — the index of the object to render.
- boolean isSelected — indicates whether the object to render is selected. Used by the example to determine which colors to use.
- boolean cellHasFocus — indicates whether the object to render has the focus.

Note that combo boxes and lists use the same type of renderer — ListCellRenderer. You can save yourself some time by sharing renderers between combo boxes and lists, if it makes sense for your program.

The Combo Box API

The API for using combo boxes falls into two categories:

- Setting or Getting the Items in the Combo Box's Menu
- Customizing the Combo Box's Operation

Setting or Getting the Items in the Combo Boxes's Menu	
Method	**Purpose**
JComboBox() JComboBox(ComboBoxModel) JComboBox(Object[]) JComboBox(Vector)	Create a combo box with the specified items in its menu. A combo box created with the default constructor has no items in the menu initially. Each of the other constructors initializes the menu from its argument: a model object, an array of objects, or a Vector of objects.
void addItem(Object) void insertItemAt(Object, int)	Add or insert the specified object into the combo box's menu. The insert method places the specified object *at* the specified index, thus inserting it before the object currently at that index. These methods require that the combo box's data model be an instance of MutableComboBoxModel.
Object getItemAt(int) Object getSelectedItem()	Get an item from the combo box's menu.
void removeAllItems() void removeItemAt(int) void removeItem(Object)	Remove one or more items from the combo box's menu. These methods require that the combo box's data model be an instance ofMutableComboBoxModel.
int getItemCount()	Get the number of items in the combo box's menu.
void setModel(ComboBoxModel) ComboBoxModel getModel()	Set or get the data model that provides the items in the combo box's menu.
void setAction(Action) Action getAction()	Set or get the Action associated with the combo box. For further information, see How to Use Actions.

Customizing the Combo Box's Operation	
Method or Constructor	**Purpose**
void addActionListener(ActionListener)	Add an action listener to the combo box. The listener's actionPerformed method is called when the user selects an item from the combo box's menu or, in an editable combo box, when the user presses Enter.
void addItemListener(ItemListener)	Add an item listener to the combo box. The listener's itemStateChanged method is called when the selection state of any of the combo box's items change.
void setEditable(boolean) boolean isEditable()	Set or get whether the user can type in the combo box.

void setRenderer(ListCellRenderer) ListCellRenderer getRenderer()	Set or get the object responsible for painting the selected item in the combo box. The renderer is used only when the combo box is uneditable. If the combo box is editable, the editor is used to paint the selected item instead.
void setEditor(ComboBoxEditor) ComboBoxEditor getEditor()	Set or get the object responsible for painting and editing the selected item in the combo box. The editor is used only when the combo box is editable. If the combo box is uneditable, the renderer is used to paint the selected item instead.

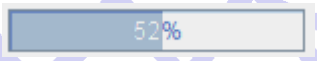
Q17. Write short note on progress bar.
ANSWER

Sometimes a task running within a program might take a while to complete. A user-friendly program provides some indication to the user that the task is occurring, how long the task might take, and how much work has already been done. One way of indicating work, and perhaps the amount of progress, is to use an animated image.

Another way of indicating work is to set the wait cursor, using the Cursor class and the Component-defined setCursor method. For example, the following code makes the wait cursor be displayed when the cursor is over container (including any components it contains that have no cursor specified):

```
container.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
```

To convey how complete a task is, you can use a progress bar like this one:



Sometimes you can't immediately determine the length of a long-running task, or the task might stay stuck at the same state of completion for a long time. You can show work without measurable progress by putting the progress bar in *indeterminate mode.* A progress bar in indeterminate mode displays animation to indicate that work is occurring. As soon as the progress bar can display more meaningful information, you should switch it back into its default, determinate mode. In the Java look and feel, indeterminate progress bars look like this:

Swing provides three classes to help you use progress bars:

JProgressBar
A visible component to graphically display how much of a total task has completed. See Using Determinate Progress Bars for information and an example of using a typical progress bar. The section Using Indeterminate Mode tells you how to animate a progress bar to show activity before the task's scope is known.

ProgressMonitor

Not a visible component. Instead, an instance of this class monitors the progress of a task and pops up a dialog if necessary. See How to Use Progress Monitors for details and an example of using a progress monitor.

ProgressMonitorInputStream

An input stream with an attached progress monitor, which monitors reading from the stream. You use an instance of this stream like any of the other input streams described in Basic I/O. You can get the stream's progress monitor with a call to getProgressMonitor and configure it as described in How to Use Progress Monitors.

Commonly used Constructors of JProgressBar class:

- **JProgressBar():** is used to create a horizontal progress bar but no string text.
- **JProgressBar(int min, int max):** is used to create a horizontal progress bar with the specified minimum and maximum value.
- **JProgressBar(int orient):** 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):** is used to create a progress bar with the specified orientation, minimum and maximum value.

Commonly used methods of JProgressBar class:

1) public void setStringPainted(boolean b): is used to determine whether string should be displayed.

2) public void setString(String s): is used to set value to the progress string.

3) public void setOrientation(int orientation): is used to set the orientation, it may be either vertical or horizontal by using SwingConstants.VERTICAL and SwingConstants.HORIZONTAL constants..

4) public void setValue(int value): is used to set the current value on the progress bar.

Example of JProgressBar class:

```
import javax.swing.*;
public class MyProgress extends JFrame{
JProgressBar jb;
int i=0,num=0;

MyProgress(){
jb=new JProgressBar(0,2000);
jb.setBounds(40,40,200,30);

jb.setValue(0);
jb.setStringPainted(true);

add(jb);
setSize(400,400);
setLayout(null);
}

public void iterate(){
while(i<=2000){
  jb.setValue(i);
```

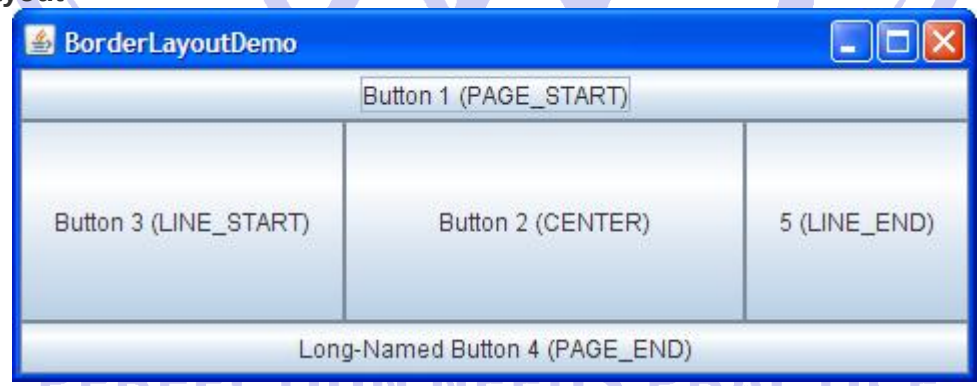
```
  i=i+20;
  try{Thread.sleep(150);}catch(Exception e){}
}
}
public static void main(String[] args) {
    MyProgress m=new MyProgress();
    m.setVisible(true);
    m.iterate();
}
}
```

Q18. Write short note on layouts in swings.
ANSWER

Several AWT and Swing classes provide layout managers for general use:

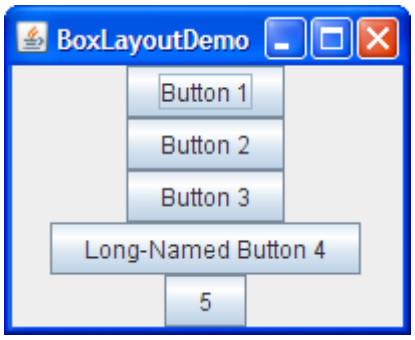
- BorderLayout
- BoxLayout
- CardLayout
- FlowLayout
- GridBagLayout
- GridLayout
- GroupLayout
- SpringLayout

BorderLayout



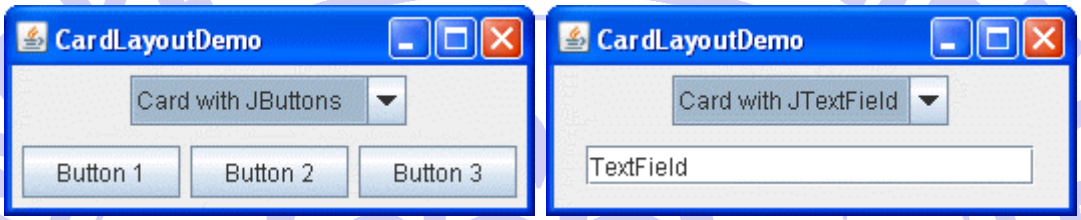
Every content pane is initialized to use a BorderLayout. (As Using Top-Level Containers explains, the content pane is the main container in all frames, applets, and dialogs.) ABorderLayout places components in up to five areas: top, bottom, left, right, and center. All extra space is placed in the center area. Tool bars that are created using JToolBar must be created within a BorderLayout container, if you want to be able to drag and drop the bars away from their starting positions.

BoxLayout



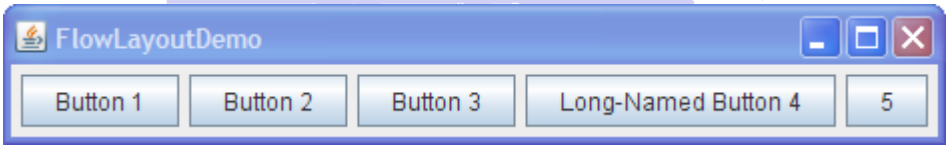
The BoxLayout class puts components in a single row or column. It respects the components' requested maximum sizes and also lets you align components.

CardLayout



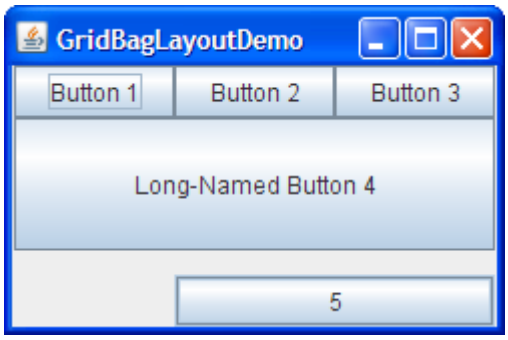
The CardLayout class lets you implement an area that contains different components at different times. A CardLayout is often controlled by a combo box, with the state of the combo box determining which panel (group of components) the CardLayout displays. An alternative to using CardLayout is using a tabbed pane, which provides similar functionality but with a pre-defined GUI.

FlowLayout



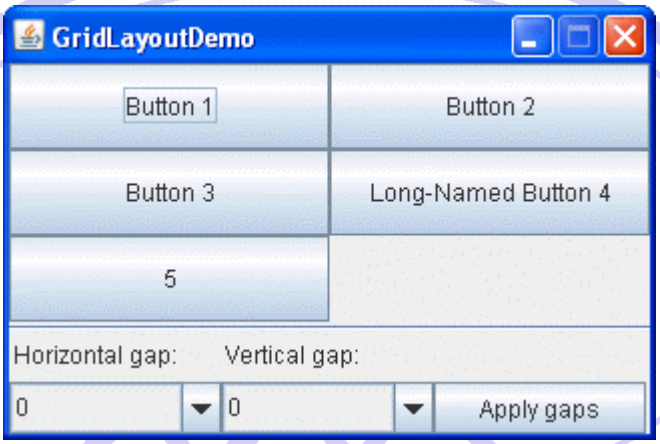
FlowLayout is the default layout manager for every JPanel. It simply lays out components in a single row, starting a new row if its container is not sufficiently wide. Both panels in CardLayoutDemo, shown previously, use FlowLayout.

GridBagLayout



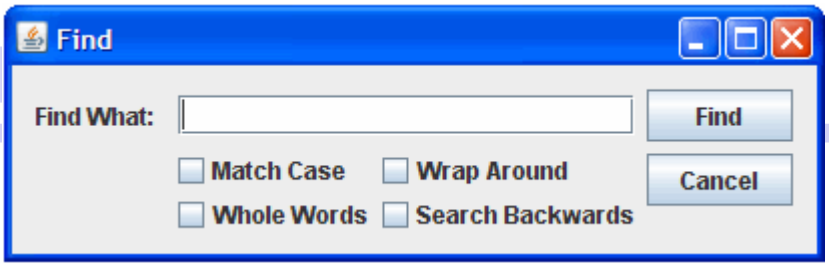
GridBagLayout is a sophisticated, flexible layout manager. It aligns components by placing them within a grid of cells, allowing components to span more than one cell. The rows in the grid can have different heights, and grid columns can have different widths.

GridLayout



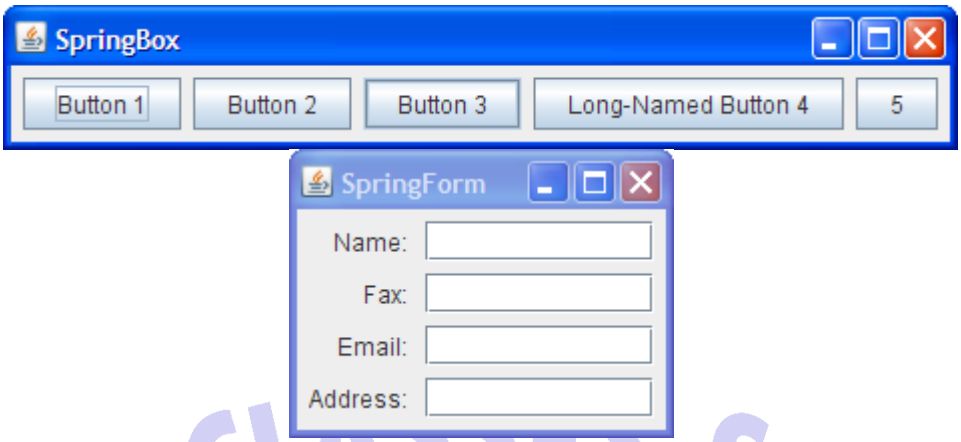
GridLayout simply makes a bunch of components equal in size and displays them in the requested number of rows and columns.

GroupLayout



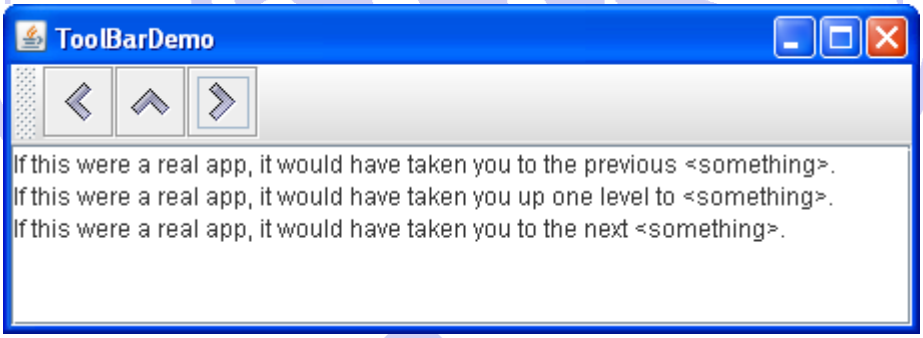
GroupLayout is a layout manager that was developed for use by GUI builder tools, but it can also be used manually. GroupLayout works with the horizontal and vertical layouts separately. The layout is defined for each dimension independently. Consequently, however, each component needs to be defined twice in the layout.

SpringLayout

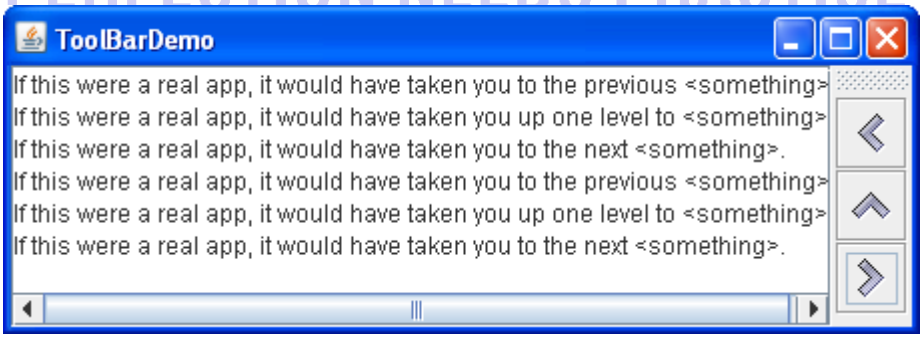


SpringLayout is a flexible layout manager designed for use by GUI builders. It lets you specify precise relationships between the edges of components under its control. For example, you might define that the left edge of one component is a certain distance (which can be dynamically calculated) from the right edge of a second component.

Q19. Write short note on Toolbars.
ANSWER
A JToolBar is a container that groups several components — usually buttons with icons — into a row or column. Often, tool bars provide easy access to functionality that is also in menus.

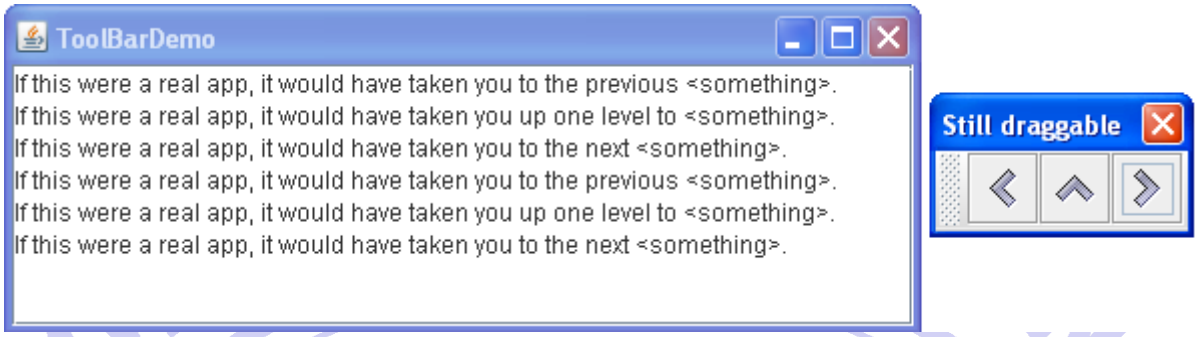


By default, the user can drag the tool bar to another edge of its container or out into a window of its own. The next figure shows how the application looks after the user has dragged the tool bar to the right edge of its container.



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

The next figure shows how the application looks after the user has dragged the tool bar outside its window.




```
public class ToolBarDemo extends JPanel
                implements ActionListener {
   ...
   public ToolBarDemo() {
      super(new BorderLayout());
      ...
      JToolBar toolBar = new JToolBar("Still draggable");
      addButtons(toolBar);
      ...
      setPreferredSize(new Dimension(450, 130));
      add(toolBar, BorderLayout.PAGE_START);
      add(scrollPane, BorderLayout.CENTER);
   }
   ...
}
```

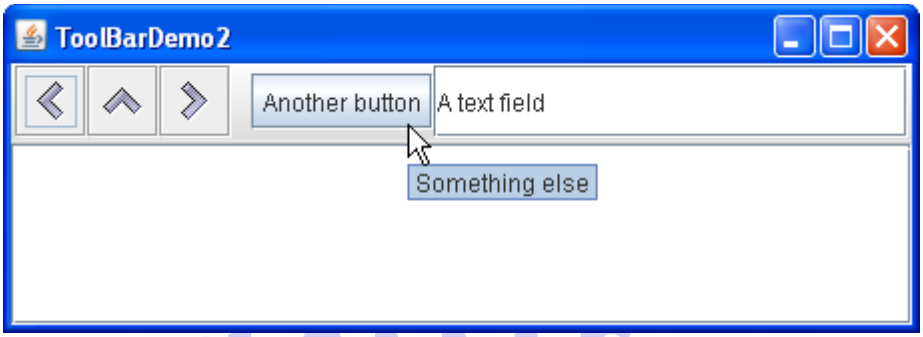
This code positions the tool bar above the scroll pane by placing both components in a panel controlled by a border layout, with the tool bar in the PAGE_START position and the scroll pane in the CENTER position. Because the scroll pane is in the center and no other components except the tool bar are in the container, by default the tool bar can be dragged to other edges of the container. The tool bar can also be dragged out into its own window, in which case the window has the title "Still draggable", as specified by the JToolBar constructor.

Customizing Tool Bars

By adding a few lines of code to the preceding example, we can demonstrate some more tool bar features:

- Using setFloatable(false) to make a tool bar immovable.
- Using setRollover(true) to visually indicate tool bar buttons when the user passes over them with the cursor.
- Adding a separator to a tool bar.

- Adding a non-button component to a tool bar.



Because the tool bar can no longer be dragged, it no longer has bumps at its left edge. Here is the code that turns off dragging:

```
toolBar.setFloatable(false);
```

The tool bar is in rollover mode, so the button under the cursor has a visual indicator. The kind of visual indicator depends on the look and feel. For example, the Metal look and feel uses a gradient effect to indicate the button under the cursor while other types of look and feel use borders for this purpose. Here is the code that sets rollover mode:

```
toolBar.setRollover(true);
```

Another visible difference in the example above is that the tool bar contains two new components, which are preceded by a blank space called a separator. Here is the code that adds the separator:

```
toolBar.addSeparator();
```

Here is the code that adds the new components:

```
//fourth button
button = new JButton("Another button");
...
toolBar.add(button);

//fifth component is NOT a button!
JTextField textField = new JTextField("A text field");
...
toolBar.add(textField);
```

You can easily make the tool bar components either top-aligned or bottom-aligned instead of centered by invoking the setAlignmentY method. For example, to align the tops of all the components in a tool bar, invoke setAlignmentY(TOP_ALIGNMENT) on each component. Similarly, you can use the setAlignmentX method to specify the alignment of components when the tool bar is vertical. This layout flexibility is possible because tool bars use BoxLayout to position their components.

The Tool Bar API

The following table lists the commonly used JToolBar constructors and methods.l

Method or Constructor	Purpose
JToolBar() JToolBar(int) JToolBar(String) JToolBar(String, int)	Creates a tool bar. The optional int parameter lets you specify the orientation; the default is HORIZONTAL. The optional String parameter allows you to specify the title of the tool bar's window if it is dragged outside of its container.
Component add(Component)	Adds a component to the tool bar. You can associate a button with an Action using the setAction(Action) method defined by the AbstractButton.
void addSeparator()	Adds a separator to the end of the tool bar.
void setFloatable(boolean) boolean isFloatable()	The floatable property is true by default, and indicates that the user can drag the tool bar out into a separate window. To turn off tool bar dragging, usetoolBar.setFloatable(false). Some types of look and feel might ignore this property.
void setRollover(boolean) boolean isRollover()	The rollover property is false by default. To make tool bar buttons be indicated visually when the user passes over them with the cursor, set this property to true. Some types of look and feel might ignore this property.

Q20. Explain event handling in java.
ANSWER

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.

Types of Event

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 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.Let's have a brief introduction to this 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.

The benefit of this approach is that the user interface logic is completely separated from the logic that generates the event. The user interface element is able to delegate the processing of an event to the separate piece of code. 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.

Steps involved in event handling

- The User clicks the button and the event is generated.

- Now the object of concerned event class is created automatically and information about the source and the event get populated with in same object.

- Event object is forwarded to the method of registered listener class.

- the method is now get executed and returns.

Points to remember about listener

- In order to design a listener class we have to develop some listener interfaces.These Listener interfaces forecast some public abstract callback methods which must be implemented by the listener class.

- If you do not implement the any if the predefined interfaces then your class can not act as a listener class for a source object.

Callback Methods

These are the methods that are provided by API provider and are defined by the application programmer and invoked by the application developer. Here the callback methods represents an event method. In response to an event java jre will fire callback method. All such callback methods are provided in listener interfaces.

If a component wants some listener will listen to it's events the the source must register itself to the listener.

Programming :

1. Write a program in java for the following :

   The Jtabbed pane class contains two tabs "city" and "bank".

   First is titled with panel name "city panel" and include two buttons labelled "Mumbai" and "Delhi".

   The second tab titled as bank with panel name " Bank Panel", includes list of Nationalised bank viz. SBI, PNB, SBBJ.

2. Write a Swing program with two text boxes labeled "First Number" and "Second Number" respectively. Include a Button labeled "Reverse" on the click of which the number entered in the first box should be reversed and displayed in the second box.

3. Write a Swing program containing three text fields. First text field accepts Last Name and Second text field accept First Name. On click of button full name is displayed in third box.

4. write a swing program containing a button with the caption "Now" and a textfield. On click of the button, the current date and time should be displayed in a textfield.

5. Write a program in java which creates a list containing at least 3 states of India. On the click of any state, the capital of that state should be displayed in a Text field.

6. Write a program in java that reads an integer number 'n' in a text field. Create a JButton labeled "Sum". On click of the JButton 1+2+3+.....+n should be calculated and displayed in another text field.

TY BSc CS, ADVANCED JAVA JDBC

Q1) Write a brief account of the drivers of JDBC (any two).

JDBC Drivers Types

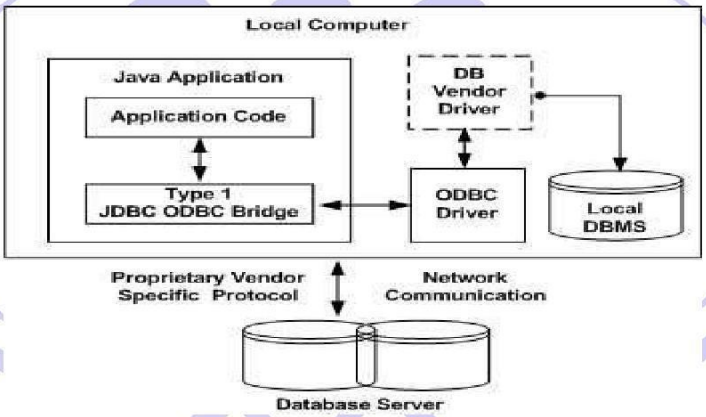
JDBC driver implementations vary because of the wide variety of operating systems and hardware platforms in which Java operates. Sun has divided the implementation types into four categories, Types 1, 2, 3, and 4.

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.

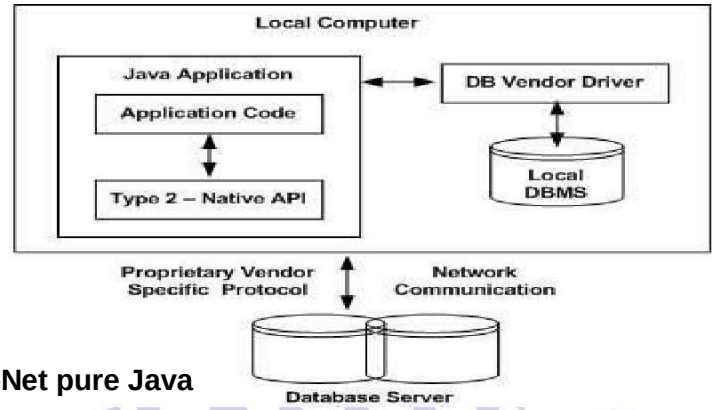


Type 2: JDBC-Native API

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.

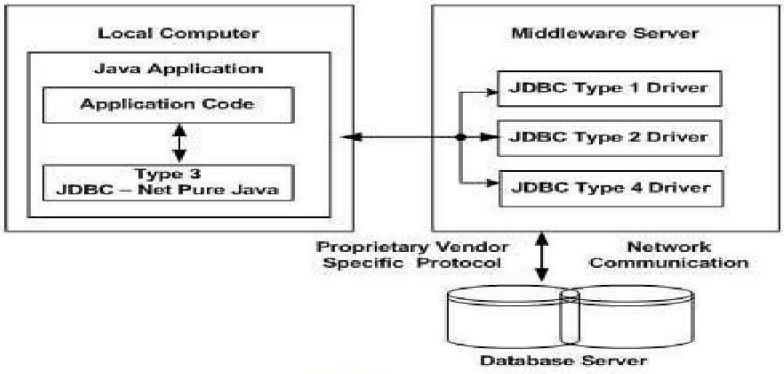
The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.



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.

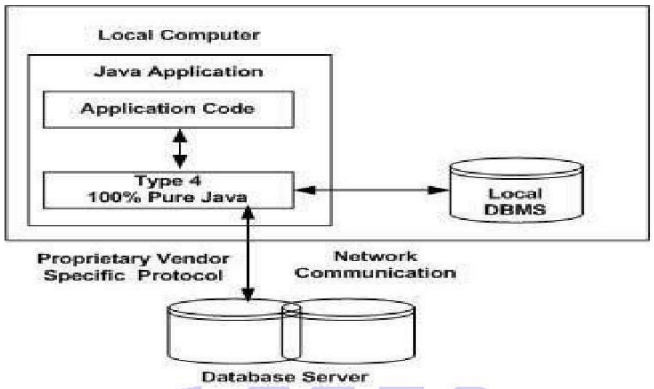


You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application.

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.

Q2)Explain Class.forName() method.

The most common approach to register a driver is to use Java's **Class.forName()** method, to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

The following example uses Class.forName( ) to register the Oracle driver −

```
try { Class.forName("oracle.jdbc.driver.OracleDriver");

}

catch(ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver
  class!"); System.exit(1);

}
```

You can use **getInstance()** method to work around noncompliant JVMs, but then you'll have to code for two extra Exceptions as follows −

```
try { Class.forName("oracle.jdbc.driver.OracleDriver").newInstance();

}

catch(ClassNotFoundException ex) {
  System.out.println("Error: unable to load driver
  class!"); System.exit(1);
```

```
catch(IllegalAccessException ex) {
  System.out.println("Error: access problem while
  loading!"); System.exit(2);

catch(InstantiationException ex) { System.out.println("Error:
  unable to instantiate driver!"); System.exit(3);

}
```

Q3) Explain the importance of DriverManager.getConnection() method.

After you've loaded the driver, you can establish a connection using the**DriverManager.getConnection()** method. For easy reference, let me list the three overloaded DriverManager.getConnection() methods −

2. getConnection(String url)
3. getConnection(String url, Properties prop)
4. getConnection(String url, String user, String password)

Here each form requires a database **URL**. A database URL is an address that points to your database.Formulating a database URL is where most of the problems associated with establishing a connection occurs.

Following table lists down the popular JDBC driver names and database URL.

RDBMS	JDBC driver name	URL format
MySQL	com.mysql.jdbc.Driver	**jdbc:mysql://**hostname/ databaseName
ORACLE	oracle.jdbc.driver.OracleDriver	**jdbc:oracle:thin:**@hostname:port Number:databaseName
DB2	COM.ibm.db2.jdbc.net.DB2Driver	**jdbc:db2:**hostname:port Number/databaseName
Sybase	com.sybase.jdbc.SybDriver	**jdbc:sybase:Tds:**hostname: port Number/databaseName

TY BSc IT, AD

Create Connection Object

We have listed down three forms of **DriverManager.getConnection()** method to create a connection object.

Using a Database URL with a username and password

The most commonly used form of getConnection() requires you to pass a database URL, a*username*, and a *password*:

Assuming you are using Oracle's **thin** driver, you'll specify a host:port:databaseName value for the database portion of the URL.

If you have a host at TCP/IP address 192.0.0.1 with a host name of amrood, and your Oracle listener is configured to listen on port 1521, and your database name is EMP, then complete database URL would be −

```
jdbc:oracle:thin:@amrood:1521:EMP
```

Now you have to call getConnection() method with appropriate username and password to get a **Connection** object as follows −

```
String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
String USER = "username";
String PASS = "password"
Connection conn = DriverManager.getConnection(URL, USER, PASS);
```

Using Only a Database URL

A second form of the DriverManager.getConnection( ) method requires only a database URL −

```
DriverManager.getConnection(String url);
```

However, in this case, the database URL includes the username and password and has the following general form −

```
jdbc:oracle:driver:username/password@database So,
the above connection can be created as follows −

String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```

Using a Database URL and a Properties Object

A third form of the DriverManager.getConnection( ) method requires a database URL and a Properties object −

```
DriverManager.getConnection(String url, Properties info);
```

A Properties object holds a set of keyword-value pairs. It is used to pass driver properties to the driver during a call to the getConnection() method.

To make the same connection made by the previous examples, use the following code −

```
import java.util.*;

String URL = "jdbc:oracle:thin:@amrood:1521:EMP";
Properties info = new Properties( );

info.put( "user", "username" ); info.put(
"password", "password" );

Connection conn = DriverManager.getConnection(URL, info);
```

Closing JDBC Connections

At the end of your JDBC program, it is required explicitly to close all the connections to the database to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

Relying on the garbage collection, especially in database programming, is a very poor programming practice. You should make a habit of always closing the connection with the close() method associated with connection object.

To ensure that a connection is closed, you could provide a 'finally' block in your code. A *finally* block always executes, regardless of an exception occurs or not.

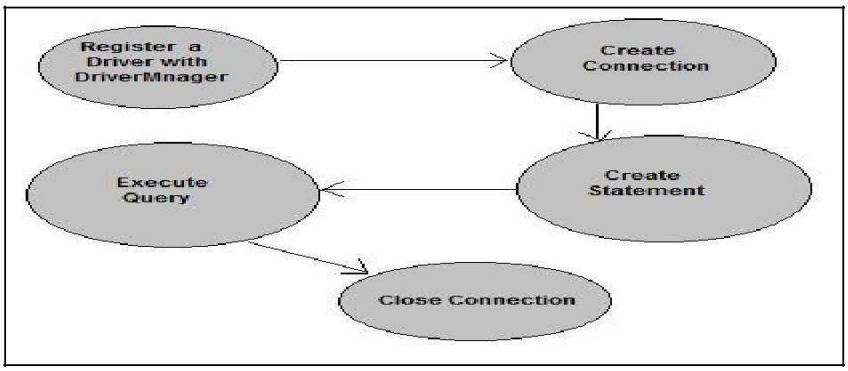
To close the above opened connection, you should call close() method as follows −

```
conn.close();
```

Q4) **Explain the steps of database connectivity.**

The following 5 steps are the basic steps involve in connecting a Java application with Database using JDBC.

- Register the Driver
- Create a Connection
- Create SQL Statement
- Execute SQL Statement
- Closing the connection



1**) Register the driver class**

The forName() method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

Syntax of forName() method

public static void forName(String className)throws ClassNotFoundException

2) Create the connection object

The getConnection() method of DriverManager class is used to establish connection with 3the database.

Syntax of getConnection() method

5) public static Connection getConnection(String url)throws SQLException

6) public static Connection getConnection(String url,String name,String password) throws SQLException

7) Create the Statement object

The createStatement() method of Connection interface is used to create statement. The object of statement is responsible to execute queries with the database.

Syntax of createStatement() method

public Statement createStatement()throws SQLException

4) Execute the query

The executeQuery() method of Statement interface is used to execute queries to the database. This method returns the object of ResultSet that can be used to get all the records of a table.

Syntax of executeQuery() method

public ResultSet executeQuery(String sql)throws SQLException

5) Close the connection object

By closing connection object statement and ResultSet will be closed automatically. The close() method of Connection interface is used to close the connection.

Syntax of close() method

public void close()throws SQLException

Q5) Short note on result sets.

- The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.
- A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created. JDBC provides the following connection methods to create statements with desired

ResultSet

- **createStatement(int RSType, int RSConcurrency);**

- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
  prepareCall(String sql, int RSType, int RSConcurrency);

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
8	
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including −

S.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.

Q6)Short note on PreparedStatement.

Sometimes it is more convenient to use a PreparedStatement object for sending SQL statements to the database. This special type of statement is derived from the more general class,Statement.

If you want to execute a Statement object many times, it usually reduces execution time to use a PreparedStatement object instead.

The main feature of a PreparedStatement object is that, unlike a Statement object, it is given a SQL statement when it is created. The advantage to this is that in most cases, this SQL statement is sent to the DBMS right away, where it is compiled. As a result,

the PreparedStatement object contains not just a SQL statement, but a SQL statement that has been precompiled. This means that when the PreparedStatement is executed, the DBMS can just run the PreparedStatement SQL statement without having to compile it first.

Although PreparedStatement objects can be used for SQL statements with no parameters, you probably use them most often for SQL statements that take parameters. The advantage

of using SQL statements that take parameters is that you can use the same statement and supply it with different values each time you execute it.

Following example uses PrepareStatement method to create PreparedStatement. It also uses setInt & setString methods of PreparedStatement to set parameters of PreparedStatement.

```
import java.sql.*;


public class jdbcConn {

  public static void main(String[] args) throws Exception{
    Class.forName("org.apache.derby.jdbc.ClientDriver");
    Connection con = DriverManager.getConnection
    ("jdbc:derby://localhost:1527/testDb","name","pass");
    PreparedStatement updateemp = con.prepareStatement
    ("insert into emp values(?,?,?)"); updateemp.setInt(1,23);
    updateemp.setString(2,"Roshan"); updateemp.setString(3,
    "CEO"); updateemp.executeUpdate();

    Statement stmt = con.createStatement();
    String query = "select * from emp";
    ResultSet rs = stmt.executeQuery(query);
    System.out.println("Id Name Job"); while
    (rs.next()) {

      int id = rs.getInt("id");

      String name = rs.getString("name");

      String job = rs.getString("job");

      System.out.println(id + " " + name+" "+job);

    }

  }
```

```
}
```

Result:

The above code sample will produce the following result.The result may vary.

```
Id Name Job

23 Roshan CEO
```

Q6)Explain the types of statements.

Once a connection is obtained we can interact with the database. The JDBC*Statement*, *CallableStatement*, and *PreparedStatement* interfaces define the methods and properties that enable you to send SQL or PL/SQL commands and receive data from your database.

They also define methods that help bridge data type differences between Java and SQL data types used in a database.

The following table provides a summary of each interface's purpose to decide on the interface to use.

Interfaces	Recommended Use
Statement	Use the for general-purpose access to your database. Useful when you are using static SQL statements at runtime. The Statement interface cannot accept parameters.
PreparedStatement	Use the when you plan to use the SQL statements many times. The PreparedStatement interface accepts input parameters at runtime.
CallableStatement	Use the when you want to access the database stored procedures. The CallableStatement interface can also accept runtime input parameters.

The Statement Objects

Creating Statement Object

Before you can use a Statement object to execute a SQL statement, you need to create one using the Connection object's createStatement( ) method, as in the following example −

```
Statement stmt =
null; try {

  stmt = conn.createStatement( );

  . . .

}

catch (SQLException e) {

  . . .

}

finally {

  . . .

}
```

Once you've created a Statement object, you can then use it to execute an SQL statement with execute methods.

The PreparedStatement Objects

The *PreparedStatement* interface extends the Statement interface, which gives you added functionality with a couple of advantages over a generic Statement object.

This statement gives you the flexibility of supplying arguments dynamically.

Creating PreparedStatement Object

```
PreparedStatement pstmt = null; try {

  String SQL = "Update Employees SET age = ? WHERE id = ?";
  pstmt = conn.prepareStatement(SQL);

  . . .

}

catch (SQLException e) {

  . . .
```

```
}

finally {

  . . .

}
```

The CallableStatement Objects

Just as a Connection object creates the Statement and PreparedStatement objects, it also creates the CallableStatement object, which would be used to execute a call to a database stored procedure.

Creating CallableStatement Object

Suppose, you need to execute the following Oracle stored procedure −

```
CREATE OR REPLACE PROCEDURE getEmpName

  (EMP_ID IN NUMBER, EMP_FIRST OUT VARCHAR) AS

BEGIN

  SELECT first INTO EMP_FIRST

  FROM Employees

  WHERE ID = EMP_ID;

END;
```

Q7) Difference between JDBC and ODBC.

ODBC

ODBC is a multidatabase API for programs that use SQL statements to access data. An ODBC-based program can access heterogeneous databases without needing source code changes-one program can retrieve and store content in different vendors' databases via the ODBC interface. ODBC thus provides database-neutral delivery of both SQL and database content. Be aware, however, that you must load ODBC driver software for each vendor's database you want to access.

JDBC

JDBC is a collection of database access middleware drivers that provide Java programs with a call-level SQL API. Java applets and applications can use the drivers' API to connect to databases, store and retrieve database content and execute stored procedures, thus making

JDBC a Java-enabled delivery mechanism for SQL. JDBC is to Java programs what ODBC is to programs written in languages other than Java. In fact, JDBC's design is based on ODBC's

Multithread: - JDBC is multi-threaded - ODBC is not multi-threaded (at least not thread safe)

Flexibility: - ODBC is a windows-specific technology - JDBC is specific to Java, and is therefore supported on whatever OS supports Java

Power : you can do everything with JDBC that you can do with ODBC, on any platform.
Language: - ODBC is procedural and language independent - JDBC is object oriented and language dependent (specific to java).

Heavy load: - JDBC is faster - ODBC is slower

ODBC limitation: it is a relational API and can only work with data types that can be expressed in rectangular or two-dimensional format. (it will not work with data types like Oracle's spatial data type)

API: JDBC API is a natural Java Interface and is built on ODBC, and therefore JDBC retains some of the basic feature of ODBC

Q8) Write short note on JDBC.
ANSWER

JDBC stands for **J**ava **D**ata**b**ase **C**onnectivity, which is a standard Java API for database-independent connectivity between the Java programming language and a wide range of databases.

The JDBC library includes APIs for each of the tasks mentioned below that are commonly associated with database usage.

- Making a connection to a database.
- Creating SQL or MySQL statements.
- Executing SQL or MySQL queries in the database.
- Viewing & Modifying the resulting records.

Fundamentally, JDBC is a specification that provides a complete set of interfaces that allows for portable access to an underlying database. Java can be used to write different types of executables, such as −

- Java Applications
- Java Applets
- Java Servlets
- Java ServerPages (JSPs)

- Enterprise JavaBeans (EJBs).

All of these different executables are able to use a JDBC driver to access a database, and take advantage of the stored data.

JDBC provides the same capabilities as ODBC, allowing Java programs to contain database-independent code.

JDBC Architecture

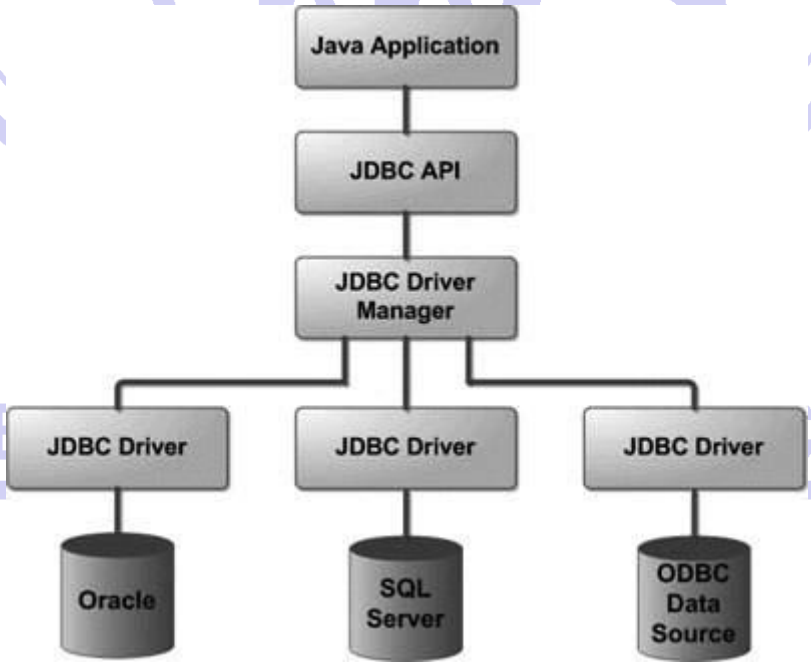
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers −

- **JDBC API:** This provides the application-to-JDBC Manager connection.
- **JDBC Driver API:** This supports the JDBC Manager-to-Driver Connection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases.

The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

Following is the architectural diagram, which shows the location of the driver manager with respect to the JDBC drivers and the Java application −



Common JDBC Components

The JDBC API provides the following interfaces and classes −

- **DriverManager:** This class manages a list of database drivers. Matches connection requests from the java application with the proper database driver using communication

sub protocol. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database Connection.

- **Driver:** This interface handles the communications with the database server. You will interact directly with Driver objects very rarely. Instead, you use DriverManager objects, which manages objects of this type. It also abstracts the details associated with working with Driver objects.

- **Connection:** This interface with all methods for contacting a database. The connection object represents communication context, i.e., all communication with database is through connection object only.

- **Statement:** You use objects created from this interface to submit the SQL statements to the database. Some derived interfaces accept parameters in addition to executing stored procedures.

- **ResultSet:** These objects hold data retrieved from a database after you execute an SQL query using Statement objects. It acts as an iterator to allow you to move through its data.

- **SQLException:** This class handles any errors that occur in a database application.

The JDBC 4.0 Packages

The java.sql and javax.sql are the primary packages for JDBC 4.0. This is the latest JDBC version at the time of writing this tutorial. It offers the main classes for interacting with your data sources.

The new features in these packages include changes in the following areas −

- Automatic database driver loading.
- Exception handling improvements.
- Enhanced BLOB/CLOB functionality.
- Connection and statement interface enhancements.
- National character set support.
- SQL ROWID access.
- SQL 2003 XML data type support.
- Annotations.

Q9) Why Java prefer JDBC over ODBC.
ANSWER

Defining Database Drivers

Different database vendors, such as Microsoft® or Oracle®, implement their database systems using various technologies depending on customer needs, market demands, and several other factors. Software applications written in popular programming languages, such as C, C++, or Java®, need a way to communicate with these databases. Open Database Connectivity (ODBC) and Java Database Connectivity (JDBC) are standards for drivers that enable programmers to write database-agnostic software applications. ODBC and JDBC are simply standards, or a set of rules recommended for efficient communication with a database. The database vendor is responsible for implementing and providing drivers that are committed to follow these rules.

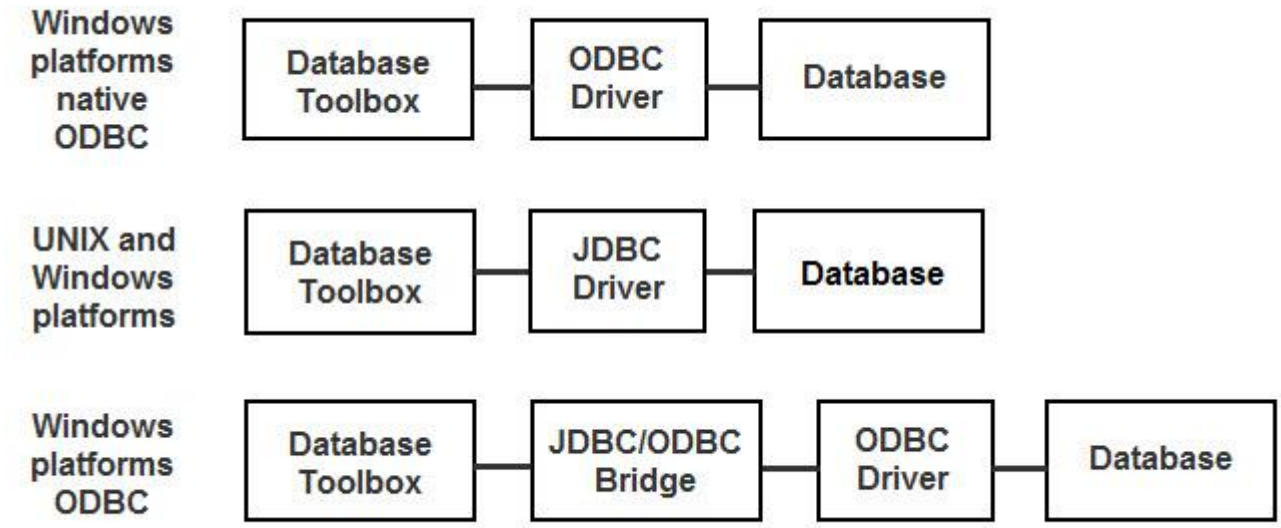
Deciding Between ODBC and JDBC Drivers

ODBC is a standard Microsoft Windows® interface that enables communication between database management systems and applications typically written in C or C++.

JDBC is a standard interface that enables communication between applications based on Oracle Java and database management systems.

The JDBC/ODBC bridge is a Java library that allows Java applications to access the ODBC interface.

Database Toolbox™ has a C++ library that connects natively to an ODBC driver. Database Toolbox has a Java library that connects directly to a pure JDBC driver or uses the JDBC/ODBC bridge to connect to an ODBC driver. The JDBC/ODBC bridge is automatically installed as part of the MATLAB® JVM™.



Depending on your environment and what you want to accomplish, decide whether using an ODBC driver or a JDBC driver suits your needs the best. Use the following to help you decide.

Use native ODBC for:

- Fastest performance for data imports and exports
- Memory-intensive data imports and exports

Use JDBC for:

- Platform independence allowing you to work with any operating system (including Mac and Linux®), driver version, or bitness
- Using Database Toolbox functions not supported by the native ODBC interface (such as runstoredprocedure)
- Working with complex or long data types (e.g., LONG, BLOB, text, etc.)

Q10) Explain two tier & three tier module.

ANSWER

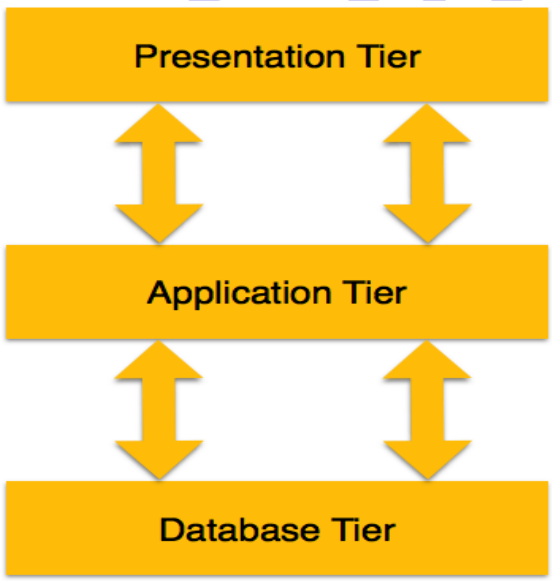
The design of a DBMS depends on its architecture. It can be centralized or decentralized or hierarchical. The architecture of a DBMS can be seen as either single tier or multi-tier. An n-tier architecture divides the whole system into related but independent **n** modules, which can be independently modified, altered, changed, or replaced.

In 1-tier architecture, the DBMS is the only entity where the user directly sits on the DBMS and uses it. Any changes done here will directly be done on the DBMS itself. It does not provide handy tools for end-users. Database designers and programmers normally prefer to use single-tier architecture.

If the architecture of DBMS is 2-tier, then it must have an application through which the DBMS can be accessed. Programmers use 2-tier architecture where they access the DBMS by means of an application. Here the application tier is entirely independent of the database in terms of operation, design, and programming.

3-tier Architecture

A 3-tier architecture separates its tiers from each other based on the complexity of the users and how they use the data present in the database. It is the most widely used architecture to design a DBMS.



- **Database (Data) Tier** − At this tier, the database resides along with its query processing languages. We also have the relations that define the data and their constraints at this level.

- **Application (Middle) Tier** − At this tier reside the application server and the programs that access the database. For a user, this application tier presents an abstracted view of the database. End-users are unaware of any existence of the database beyond the application. At the other end, the database tier is not aware of any other user beyond the application tier. Hence, the application layer sits in the middle and acts as a mediator between the end-user and the database.

- **User (Presentation) Tier** − End-users operate on this tier and they know nothing about any existence of the database beyond this layer. At this layer, multiple views of the database can be provided by the application. All views are generated by applications that reside in the application tier.

Multiple-tier database architecture is highly modifiable, as almost all its components are independent and can be changed independently.

Q11) What are JDBC drivers.

ANSWER

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.

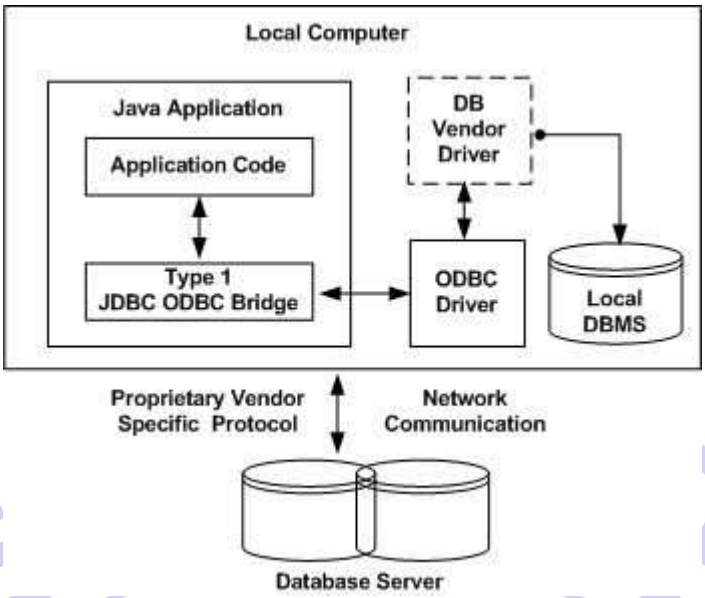
JDBC Drivers Types

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.

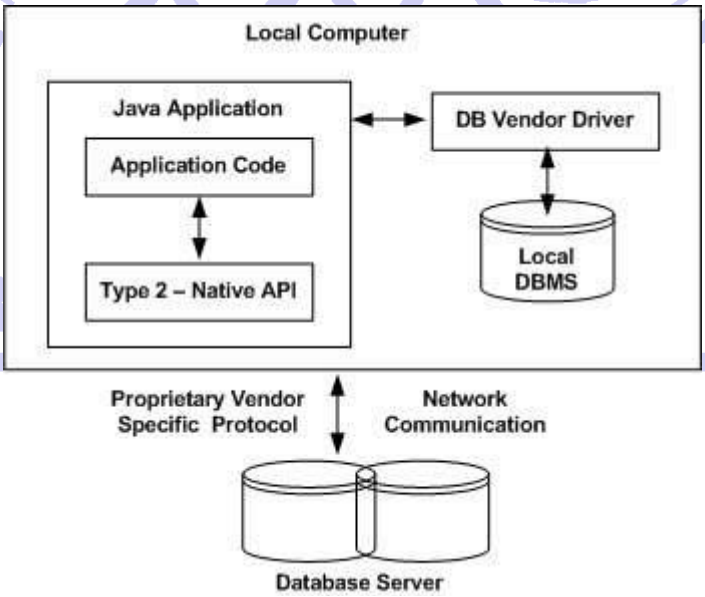


The JDBC-ODBC Bridge that comes with JDK 1.2 is a good example of this kind of driver.

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.

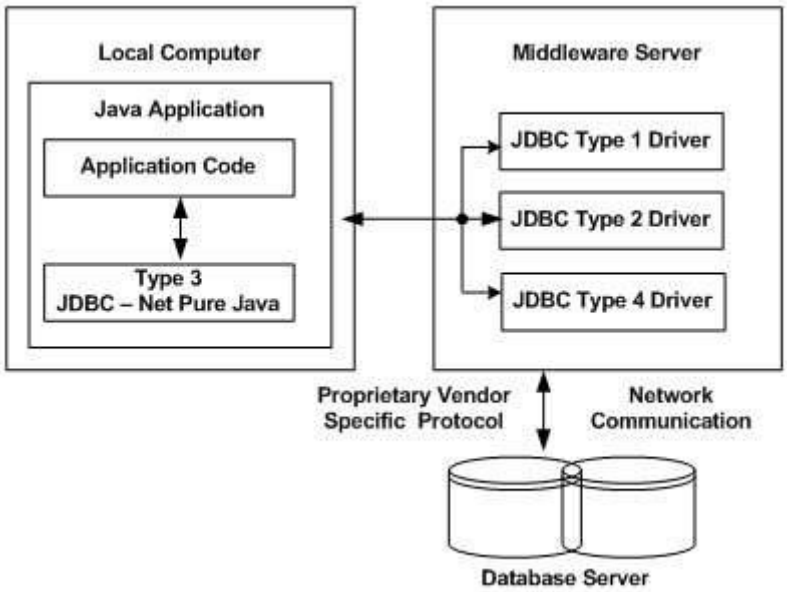


The Oracle Call Interface (OCI) driver is an example of a Type 2 driver.

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.



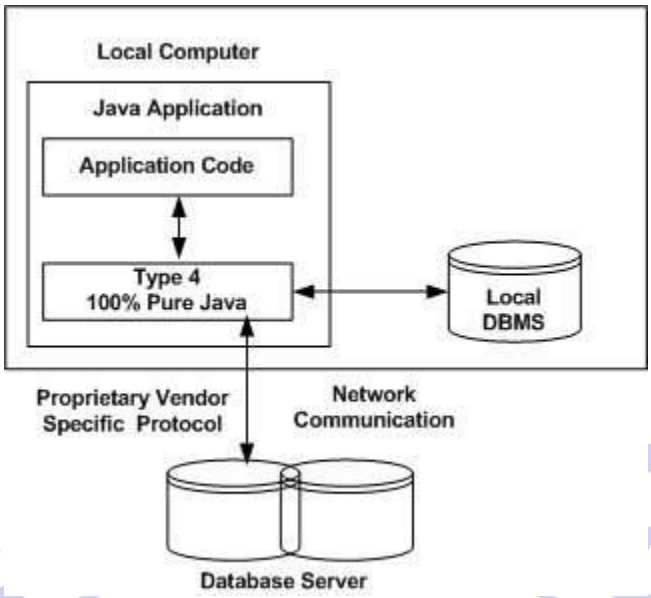
You can think of the application server as a JDBC "proxy," meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type.

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.

Q12) Explain resultset in JDBC

ANSWER

The SQL statements that read data from a database query, return the data in a result set. The SELECT statement is the standard way to select rows from a database and view them in a result set. The *java.sql.ResultSet* interface represents the result set of a database query.

A ResultSet object maintains a cursor that points to the current row in the result set. The term "result set" refers to the row and column data contained in a ResultSet object.

The methods of the ResultSet interface can be broken down into three categories −

- **Navigational methods:** Used to move the cursor around.
- **Get methods:** Used to view the data in the columns of the current row being pointed by the cursor.
- **Update methods:** Used to update the data in the columns of the current row. The updates can then be updated in the underlying database as well.

The cursor is movable based on the properties of the ResultSet. These properties are designated when the corresponding Statement that generates the ResultSet is created.

JDBC provides the following connection methods to create statements with desired ResultSet −

- **createStatement(int RSType, int RSConcurrency);**
- **prepareStatement(String SQL, int RSType, int RSConcurrency);**
- **prepareCall(String sql, int RSType, int RSConcurrency);**

The first argument indicates the type of a ResultSet object and the second argument is one of two ResultSet constants for specifying whether a result set is read-only or updatable.

Type of ResultSet

The possible RSType are given below. If you do not specify any ResultSet type, you will automatically get one that is TYPE_FORWARD_ONLY.

Type	Description
ResultSet.TYPE_FORWARD_ONLY	The cursor can only move forward in the result set.
ResultSet.TYPE_SCROLL_INSENSITIVE	The cursor can scroll forward and backward, and the result set is not sensitive to changes made by others to the database that occur after the result set was created.
ResultSet.TYPE_SCROLL_SENSITIVE.	The cursor can scroll forward and backward, and the result set is sensitive to changes made by others to the database that occur after the result set was created.

Concurrency of ResultSet

The possible RSConcurrency are given below. If you do not specify any Concurrency type, you will automatically get one that is CONCUR_READ_ONLY.

Concurrency	Description
ResultSet.CONCUR_READ_ONLY	Creates a read-only result set. This is the default
ResultSet.CONCUR_UPDATABLE	Creates an updateable result set.

All our examples written so far can be written as follows, which initializes a Statement object to create a forward-only, read only ResultSet object −

```
try {
   Statement stmt = conn.createStatement(
                        ResultSet.TYPE_FORWARD_ONLY,
                        ResultSet.CONCUR_READ_ONLY);
}
```

```
catch(Exception ex) {
   ....
}
finally {
   ....
}
```

Navigating a Result Set

There are several methods in the ResultSet interface that involve moving the cursor, including −

S.N.	Methods & Description
1	**public void beforeFirst() throws SQLException** Moves the cursor just before the first row.
2	**public void afterLast() throws SQLException** Moves the cursor just after the last row.
3	**public boolean first() throws SQLException** Moves the cursor to the first row.
4	**public void last() throws SQLException** Moves the cursor to the last row.
5	**public boolean absolute(int row) throws SQLException** Moves the cursor to the specified row.
6	**public boolean relative(int row) throws SQLException** Moves the cursor the given number of rows forward or backward, from where it is currently pointing.
7	**public boolean previous() throws SQLException** Moves the cursor to the previous row. This method returns false if the previous row is off the result set.
8	**public boolean next() throws SQLException** Moves the cursor to the next row. This method returns false if there are no more rows in

	the result set.
9	**public int getRow() throws SQLException** Returns the row number that the cursor is pointing to.
10	**public void moveToInsertRow() throws SQLException** Moves the cursor to a special row in the result set that can be used to insert a new row into the database. The current cursor location is remembered.
11	**public void moveToCurrentRow() throws SQLException** Moves the cursor back to the current row if the cursor is currently at the insert row; otherwise, this method does nothing

For a better understanding, let us study Navigate - Example Code.

Viewing a Result Set

The ResultSet interface contains dozens of methods for getting the data of the current row.

There is a get method for each of the possible data types, and each get method has two versions −

- One that takes in a column name.
- One that takes in a column index.

For example, if the column you are interested in viewing contains an int, you need to use one of the getInt() methods of ResultSet −

S.N.	Methods & Description
1	**public int getInt(String columnName) throws SQLException** Returns the int in the current row in the column named columnName.
2	**public int getInt(int columnIndex) throws SQLException** Returns the int in the current row in the specified column index. The column index starts at 1, meaning the first column of a row is 1, the second column of a row is 2, and so on.

Similarly, there are get methods in the ResultSet interface for each of the eight Java primitive types, as well as common types such as java.lang.String, java.lang.Object, and java.net.URL.

There are also methods for getting SQL data types java.sql.Date, java.sql.Time, java.sql.TimeStamp, java.sql.Clob, and java.sql.Blob. Check the documentation for more information about using these SQL data types.

For a better understanding, let us study Viewing - Example Code.

Updating a Result Set

The ResultSet interface contains a collection of update methods for updating the data of a result set.

As with the get methods, there are two update methods for each data type −

- One that takes in a column name.
- One that takes in a column index.

For example, to update a String column of the current row of a result set, you would use one of the following updateString() methods −

S.N.	Methods & Description
1	**public void updateString(int columnIndex, String s) throws SQLException** Changes the String in the specified column to the value of s.
2	**public void updateString(String columnName, String s) throws SQLException** Similar to the previous method, except that the column is specified by its name instead of its index.

There are update methods for the eight primitive data types, as well as String, Object, URL, and the SQL data types in the java.sql package.

Updating a row in the result set changes the columns of the current row in the ResultSet object, but not in the underlying database. To update your changes to the row in the database, you need to invoke one of the following methods.

S.N.	Methods & Description
1	**public void updateRow()** Updates the current row by updating the corresponding row in the database.
2	**public void deleteRow()** Deletes the current row from the database
3	**public void refreshRow()**

	Refreshes the data in the result set to reflect any recent changes in the database.
4	**public void cancelRowUpdates()** Cancels any updates made on the current row.
5	**public void insertRow()** Inserts a row into the database. This method can only be invoked when the cursor is pointing to the insert row.

PROGRAMMING:

1 Write a JDBC program that counts the total number of managers from emp table (emp code → number, emp name Varchar, emp desig → Varchar).
2 Write a JDBC program that would create the Friends (F _Name, F _Mobile _Email) table and insert three records in it.
3 Write a JDBC program that accepts a table name and displays total number of records present in it.
4 Write a JDBC program to accept table name and display only column names?
5 Write a JDBC program that accepts a table name and displays total numbers of records present in i

Q1) Explain the class URL Connection.

1. URL stands for Uniform Resource Locator and represents a resource on the World Wide Web, such as a Web page or FTP directory.

2. This section shows you how to write Java programs that communicate with a URL.

   A URL can be broken down into parts, as follows:

```
protocol://host:port/path?query#ref
```

- Examples of protocols include HTTP, HTTPS, FTP, and File. The path is also referred to as the filename, and the host is also called the authority.

- The following is a URL to a Web page whose protocol is HTTP:

```
http://www.amrood.com/index.htm?language=en#j2se
```

1) Notice that this URL does not specify a port, in which case the default port for the protocol is used. With HTTP, the default port is 80.

2) URL Class Methods:

The **java.net.URL** class represents a URL and has complete set of methods to manipulate URL in Java.

The URL class has several constructors for creating URLs, including the following:

SN	Methods with Description
1	**public URL(String protocol, String host, int port, String file) throws MalformedURLException.** Creates a URL by putting together the given parts.
2	**public URL(String protocol, String host, String file) throws MalformedURLException** Identical to the previous constructor, except that the default port for the given protocol is used.
3	**public URL(String url) throws MalformedURLException** Creates a URL from the given String
4	**public URL(URL context, String url) throws MalformedURLException** Creates a URL by parsing the together the URL and String arguments

- The URL class contains many methods for accessing the various parts of the URL being represented. Some of the methods in the URL class include the following:

SN	Methods with Description
1	**public String getPath()** Returns the path of the URL.
2	**public String getQuery()** Returns the query part of the URL.
3	**public String getAuthority()** Returns the authority of the URL.
4	**public int getPort()** Returns the port of the URL.

5	**public int getDefaultPort()** Returns the default port for the protocol of the URL.

- **URLConnections Class Methods:**

The openConnection() method returns a **java.net.URLConnection**, an abstract class whose subclasses represent the various types of URL connections.

The URLConnection class has many methods for setting or determining information about the connection, including the following:

SN	Methods with Description
1	**Object getContent()** Retrieves the contents of this URL connection.
2	**Object getContent(Class[] classes)** Retrieves the contents of this URL connection.
3	**String getContentEncoding()** Returns the value of the content-encoding header field.
4	**int getContentLength()** Returns the value of the content-length header field.
5	**String getContentType()** Returns the value of the content-type header field.

Q2) Describe the classes used for Socket Programming.

- Java Socket programming is used for communication between the applications running on different JRE.
- Java Socket programming can be connection-oriented or connection-less.
- Socket and ServerSocket classes are used for connection-oriented socket programming and DatagramSocket and DatagramPacket classes are used for connection-less socket programming.
- The client in socket programming must know two information:

1) IP Address of Server
2) Port number.

Socket class

- A socket is simply an endpoint for communications between the machines. The Socket class can be used to create a socket.

Important methods

Method	Description
1) public InputStream getInputStream()	returns the InputStream attached with this socket.
2) public OutputStream getOutputStream()	returns the OutputStream attached with this socket.
3) public synchronized void close()	closes this socket

ServerSocket class

- The ServerSocket class can be used to create a server socket. This object is used to establish communication with the clients.

Important methods

Method	Description
1) public Socket accept()	returns the socket and establish a connection between server and client.
2) public synchronized void close()	closes the server socket.

Example of Java Socket Programming

Let's see a simple of java socket programming in which client sends a text and server receives it.

File: MyServer.java

```
import java.io.*;
import java.net.*;
public class MyServer {

public static void main(String[] args){
try{

ServerSocket ss=new ServerSocket(6666);
Socket s=ss.accept();//establishes connection

DataInputStream dis=new DataInputStream(s.getInputStream());
String str=(String)dis.readUTF();

System.out.println("message= "+str);
ss.close();

}catch(Exception e){System.out.println(e);}
}

}
```

File: MyClient.java

```
import java.io.*;
import java.net.*;
public class MyClient {

public static void main(String[] args) {
try{

Socket s=new Socket("localhost",6666);

DataOutputStream dout=new DataOutputStream(s.getOutputStream());
dout.writeUTF("Hello Server");

dout.flush();
dout.close();

s.close();
}catch(Exception e){System.out.println(e);}
}
}
```

Q3)Short note on InetAddress class.

3. **Java InetAddress** class represents an IP address. The java.net.InetAddress class provides methods to get the IP of any host name *for example* www.javatpoint.com, www.google.com, www.facebook.com etc.

5 Address types

unicast	An identifier for a single interface. A packet sent to a unicast address is delivered to the interface identified by that address. The Unspecified Address -- Also called anylocal or wildcard address. It must never be assigned to any node. It indicates the absence of an address. One example of its use is as the target of bind, which allows a server to accept a client connection on any interface, in case the server host has multiple interfaces. The unspecified address must not be used as the destination address of an IP packet. The Loopback Addresses -- This is the address assigned to the loopback interface. Anything sent to this IP address loops around and becomes IP input on the local host. This address is often used when testing a client.
multicast	An identifier for a set of interfaces (typically belonging to different nodes). A packet sent to a multicast address is delivered to all interfaces identified by that address.

6 **IP address scope**

- **Link-local** addresses are designed to be used for addressing on a single link for purposes such as auto-address configuration, neighbor discovery, or when no routers are present.
- **Site-local** addresses are designed to be used for addressing inside of a site without the need for a global prefix.
- **Global** addresses are unique across the internet.

Commonly used methods of InetAddress class

Method	Description
public static InetAddress getByName(String host) throws UnknownHostException	it returns the instance of InetAddress containing LocalHost IP and name.

public static InetAddress getLocalHost() throws UnknownHostException	it returns the instance of InetAdddress containing local host name and address.
public String getHostName()	it returns the host name of the IP address.
public String getHostAddress()	it returns the IP address in string format.

5

Example of Java InetAddress class

```
import java.io.*; import
java.net.*; public class
InetDemo{

public static void main(String[] args){
try{

InetAddress ip=InetAddress.getByName("www.javatpoint.com");

System.out.println("Host Name: "+ip.getHostName());
System.out.println("IP Address: "+ip.getHostAddress());
}catch(Exception e){System.out.println(e);}

}
}
```

Output:

```
Host Name: www.javatpoint.com
IP Address: 206.51.231.148
```

Q4) What is a Network interface? How can Network interface information be retrieved?

- A *network interface* is the point of interconnection between a computer and a private or public network.
- This class represents a Network Interface made up of a name, and a list of IP addresses assigned to this interface. It is used to identify the local interface on which a multicast group is joined.

- A network interface is generally a network interface card (NIC), but does not have to have a physical form. Instead, the network interface can be implemented in software. For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software simulating a network interface. The loopback interface is commonly used in test environments.

Retrieving Network Interfaces information

- The NetworkInterface class has no public constructor. Therefore, you cannot just create a new instance of this class with the new operator.
- Instead, the following static methods are available so that you can retrieve the interface details from the system: getByInetAddress(), getByName(), and getNetworkInterfaces().
- The first two methods are used when you already know the IP address or the name of the particular interface. The third method, getNetworkInterfaces() returns the complete list of interfaces on the machine.
- Network interfaces can be hierarchically organized. The NetworkInterface class includes two methods, getParent() and getSubInterfaces(), that are pertinent to a network interface hierarchy.
- The getParent() method returns the parent NetworkInterface of an interface. If a network interface is a subinterface, getParent() returns a non-null value. ThegetSubInterfaces() method returns all the subinterfaces of a network interface.

Q5)Explain Socket class in detail.

- The **java.net.Socket** class represents the socket that both the client and server use to communicate with each other. The client obtains a Socket object by instantiating one, whereas the server obtains a Socket object from the return value of the accept() method.
- The Socket class has five constructors that a client uses to connect to a server:

SN	Methods with Description
1)	**public Socket(String host, int port) throws UnknownHostException, IOException.** This method attempts to connect to the specified server at the specified port. If this constructor does not throw an exception, the connection is successful and the client is connected to the server.
2)	**public Socket(InetAddress host, int port) throws IOException**

This method is identical to the previous constructor, except that the host is denoted by an InetAddress object.

public Socket(String host, int port, InetAddress localAddress, int localPort) throws IOException.

Connects to the specified host and port, creating a socket on the local host at the specified address and port.

3) **public Socket(InetAddress host, int port, InetAddress localAddress, int localPort) throws IOException.**

This method is identical to the previous constructor, except that the host is denoted by an InetAddress object instead of a String

5	**public Socket()** Creates an unconnected socket. Use the connect() method to connect this socket to a server.

- When the Socket constructor returns, it does not simply instantiate a Socket object but it actually attempts to connect to the specified server and port.
- Some methods of interest in the Socket class are listed here. Notice that both the client and server have a Socket object, so these methods can be invoked by both the client and server.

SN	Methods with Description
1	**public void connect(SocketAddress host, int timeout) throws IOException** This method connects the socket to the specified host. This method is needed only when you instantiated the Socket using the no-argument constructor.
2	**public InetAddress getInetAddress()** This method returns the address of the other computer that this socket is connected to.
3	**public int getPort()** Returns the port the socket is bound to on the remote machine.
4	**public int getLocalPort()** Returns the port the socket is bound to on the local machine.

5	**public SocketAddress getRemoteSocketAddress()** Returns the address of the remote socket.
6	**public InputStream getInputStream() throws IOException** Returns the input stream of the socket. The input stream is connected to the output stream of the remote socket.
7	**public OutputStream getOutputStream() throws IOException** Returns the output stream of the socket. The output stream is connected to the input stream of the remote socket
8	**public void close() throws IOException** Closes the socket, which makes this Socket object no longer capable of

	connecting again to any server

Q6)Write a short note on socket programming.

Sockets provide the communication mechanism between two computers using TCP. A client program creates a socket on its end of the communication and attempts to connect that socket to a server.

When the connection is made, the server creates a socket object on its end of the communication. The client and server can now communicate by writing to and reading from the socket.

The java.net.Socket class represents a socket, and the java.net.ServerSocket class provides a mechanism for the server program to listen for clients and establish connections with them.

The following steps occur when establishing a TCP connection between two computers using sockets:

- The server instantiates a ServerSocket object, denoting which port number communication is to occur on.
- The server invokes the accept() method of the ServerSocket class. This method waits until a client connects to the server on the given port.
- After the server is waiting, a client instantiates a Socket object, specifying the server name and port number to connect to.

- The constructor of the Socket class attempts to connect the client to the specified server and port number. If communication is established, the client now has a Socket object capable of communicating with the server.
- On the server side, the accept() method returns a reference to a new socket on the server that is connected to the client's socket.

After the connections are established, communication can occur using I/O streams. Each socket has both an OutputStream and an InputStream. The client's OutputStream is connected to the server's InputStream, and the client's InputStream is connected to the server's OutputStream.

TCP is a two way communication protocol, so data can be sent across both streams at the same time.

Q7)Explain methods from URL class.

1. **getProtocol**
   - Returns the protocol identifier component of the URL.
4. **getAuthority**

   Returns the authority component of the URL.
5. **getHost**

   Returns the host name component of the URL.
6. **getPort**

   Returns the port number component of the URL. The getPort method returns an integer that is the port number. If the port is not set, getPort returns -1.
7. **getPath**

   Returns the path component of this URL.
8. **getQuery**

   Returns the query component of this URL.
9. **getFile**

   Returns the filename component of the URL. The getFile method returns the same as getPath, plus the concatenation of the value of getQuery, if any.
10. **getRef**

TY

Returns the reference component of the URL.

EXAMPLE:

```
import java.net.*;
import java.io.*;

public class ParseURL {
   public static void main(String[] args) throws Exception {

      URL aURL = new URL("http://example.com:80/docs/books/tutorial" +
                  "/index.html?name=networking#DOWNLOADING");

      System.out.println("protocol = " + aURL.getProtocol());
      System.out.println("authority = " + aURL.getAuthority());
      System.out.println("host = " + aURL.getHost());
      System.out.println("port = " + aURL.getPort());
      System.out.println("path = " + aURL.getPath());
      System.out.println("query = " + aURL.getQuery());
      System.out.println("filename = " + aURL.getFile());
      System.out.println("ref = " + aURL.getRef());

   }
}
```

Here is the output displayed by the program:

```
protocol = http

authority = example.com:80
host = example.com

port = 80

path = /docs/books/tutorial/index.html
query = name=networking

filename = /docs/books/tutorial/index.html?name=networking ref =
DOWNLOADING
```

PROGRAMMING:

1) Write a TCP server program in java that receives an Integer number from a client, checks wheather it is a palindrome and sends an output to the client?

2) Write a java program that accepts coputer name as a command line argument and to display its Inet Address.

3) Write a TCP client server program wherein the client sends a number to the server and the server replies with its factorial.

Q1) What are the step involved in creating and running RMI programs.

1) create the remote interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;
public interface Adder extends Remote{

public int add(int x,int y)throws RemoteException;
}
```

2) Provide the implementation of the remote interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to

1. Either extend the UnicastRemoteObject class,
2. or use the exportObject() method of the UnicastRemoteObject class

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
import java.rmi.*; import
java.rmi.server.*;

public class AdderRemote extends UnicastRemoteObject implements Adder{
AdderRemote()throws RemoteException{

super();
}
public int add(int x,int y){return x+y;}
}
```

3) create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```


Siddharth Chambers, Basement, Opp. Gaondevi Maidan, Thane (W)
022-25429358 / 8433611832
8-15, Everest Shopping Centre, 3rd Floor, Opp. Railway Station, Dombivli (W)
0251-2489958 / 8433611830

4) Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

rmiregistry 5000

5) Create and run the server application

Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object. The Naming class provides 5 methods.

- **public static java.rmi.Remote lookup(java.lang.String) throws java.rmi.NotBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it returns the reference of the remote object.
- **public static void bind(java.lang.String, java.rmi.Remote) throws java.rmi.AlreadyBoundException, java.net.MalformedURLException, java.rmi.RemoteException;** it binds the remote object with the given name.
- **public static void unbind(java.lang.String) throws java.rmi.RemoteException, java.rmi.NotBoundException, java.net.MalformedURLException;** it destroys the remote object which is bound with the given name.
- **public static void rebind(java.lang.String, java.rmi.Remote) throws java.rmi.RemoteException, java.net.MalformedURLException;** it binds the remote object to the new name.
- **public static java.lang.String[] list(java.lang.String) throws java.rmi.RemoteException, java.net.MalformedURLException;** it returns an array of the names of the remote objects bound in the registry.

In this example, we are binding the remote object by the name sonoo.

```
import java.rmi.*; import
java.rmi.registry.*; public
class MyServer{

public static void main(String args[]){
try{

Adder stub=new AdderRemote();
Naming.rebind("rmi://localhost:5000/sonoo",stub);
}catch(Exception e){System.out.println(e);}
}
}
```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;
public class MyClient{

public static void main(String args[]){
try{

Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
System.out.println(stub.add(34,4));

}catch(Exception e){}
}
}
```

Q2) Write a short note on RMI architecture.

RMI Architecture

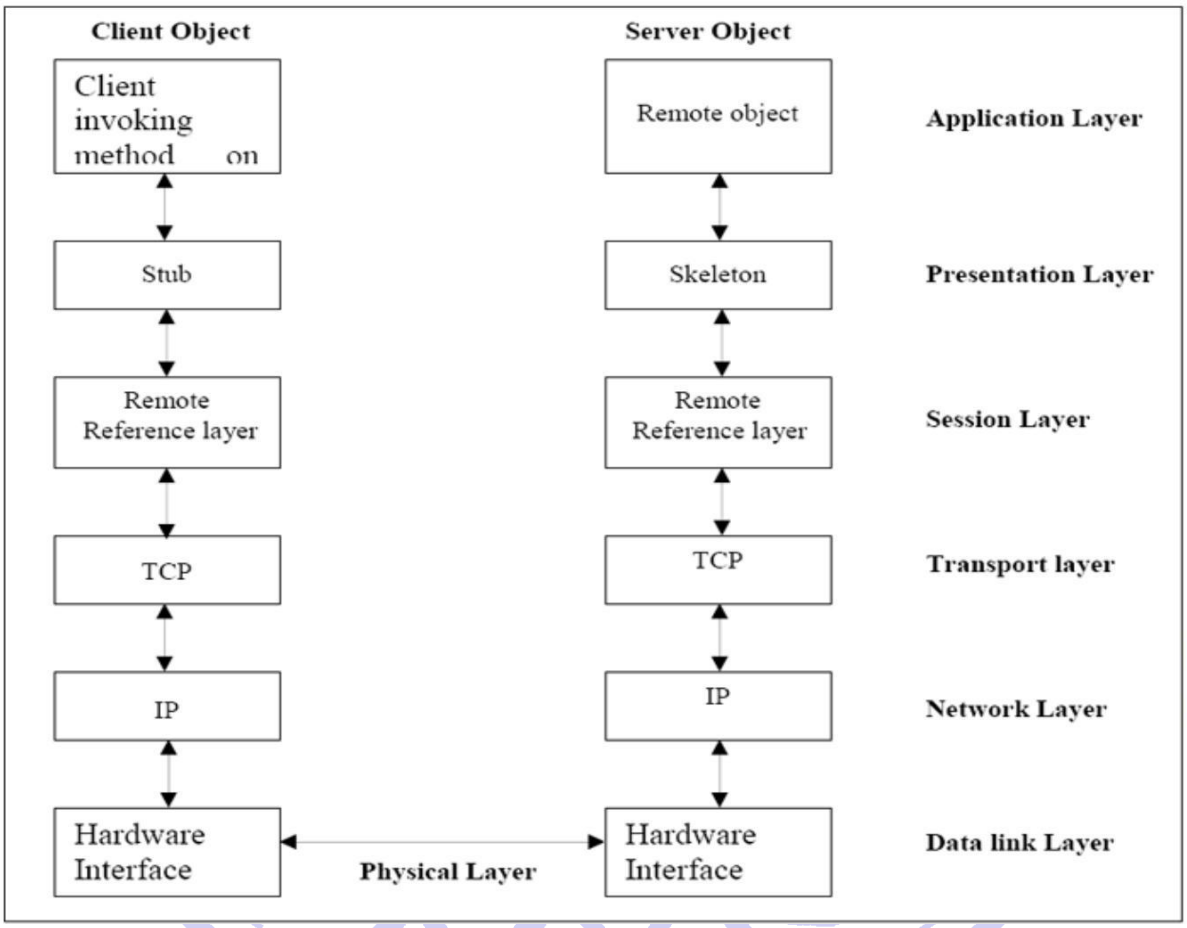
RMI's purpose is to make objects in separate JVM's look alike and act like local objects. The JVM that calls the remote object is usually referred to as a client and the JVM that contains the remote object is the server.

The RMI implementation is essentially built from three abstraction layers:

3) **The Stub/Skeleton Layer**
4) **The Remote Reference Layer**

•**The Transport Layer**

The following diagram shows the RMI Architecture:



The Stub/Skeleton Layer

This layer intercepts method calls made by the client to the interface reference and redirects these calls to a remote object. Stubs are specific to the client side, whereas skeletons are found on the server side.

To achieve location transparency, RMI introduces two special kinds of objects known as stubs and skeletons that serve as an interface between an application and rest of the RMI system.

Stub

The stub is a client-side object that represents (or acts as a proxy for) the remote object. The stub has the same interface, or list of methods, as the remote object. However when the client calls a stub method, the stub forwards the request via the RMI infrastructure to the remote object (via the skeleton), which actually executes it.

Sequence of events performed by the stub:

- Initiates a connection with the remote VM containing the remote object
- Marshals (writes and transmits) the parameters to the remote VM.

- Waits for the result of the method invocation.
- Unmarshals (reads) the return value or exception returned.
- Return the value to the caller

In the remote VM, each remote object may have a corresponding skeleton.

Skeleton

On the server side, the skeleton object takes care of all the details of "remoteness" so that the actual remote object does not need to worry about them. In other words we can pretty much code a remote object the same way as if it were local; the skeleton insulates the remote object from the RMI infrastructure.

Sequence of events performed by the skeleton:

- Unmarshals (reads) the parameters for the remote method (remember that these were marshaled by the stub on the client side)
- Invokes the method on the actual remote object implementation.
- Marshals (writes and transmits) the result (return value or exception) to the caller (which is then unmarshalled by the stub)

The Remote Reference Layer

The remote reference layer defines and supports the invocation semantics of the RMI connection. This layer maintains the session during the method call.

The Transport Layer

The Transport layer makes the stream-based network connections over TCP/IP between the JVMs, and is responsible for setting and managing those connections. Even if two JVMs are running on the same physical computer, they connect through their host computers TCP/IP network protocol stack. RMI uses a protocol called JRMP (Java Remote Method Protocol) on top of TCP/IP (an analogy is HTTP over TCP/IP).
From the JDK 1.2 the JRMP protocol was modified to eliminate the need for skeletons and instead use

reflection to make connections to the remote services objects. Thus we only need to generate stub classes in system implementation compatible with jdk 1.2 and above. To generate stubs we use the Version 1.2 option with rmic.
The RMI transport layer is designed to make a connection between clients and server, even in the face of networking obstacles. The transport layer in the current RMI implementation is TCP-based, but again a UDPbased transport layer could be substituted in a different implementation.

Q3) Explain RMI.

The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another JVM.

The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.

Understanding stub and skeleton

RMI uses stub and skeleton object for communication with the remote object.

A **remote object** is an object whose method can be invoked from another JVM. Let's understand the stub and skeleton objects:

stub

The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:

p It initiates a connection with remote Virtual Machine (JVM),

q It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),

r It waits for the result

s It reads (unmarshals) the return value or exception, and

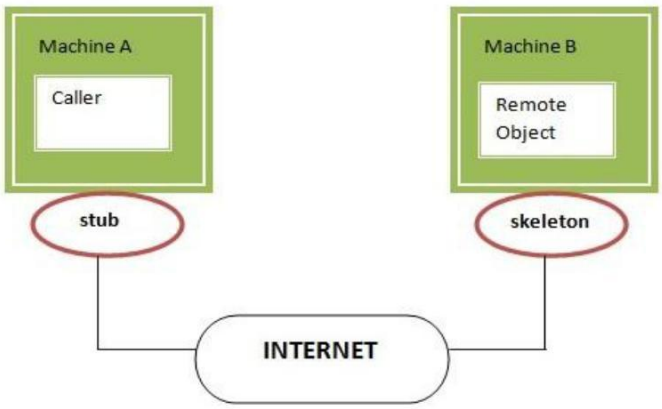
t It finally, returns the value to the caller.

skeleton

The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:

- It reads the parameter for the remote method
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.

In the Java 2 SDK, an stub protocol was introduced that eliminates the need for skeletons.



Understanding requirements for the distributed applications

If any application performs these tasks, it can be distributed application.

- The application need to locate the remote method
- It need to provide the communication with the remote objects, and
- The application need to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

Steps to write the RMI program

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

Q4)Write a short note on Implementing a Remote Interface

In general, a class that implements a remote interface should at least do the following:

1 Declare the remote interfaces being implemented

2 Define the constructor for each remote object

3 Provide an implementation for each remote method in the remote interfaces

An RMI server program needs to create the initial remote objects and *export* them to the RMI runtime, which makes them available to receive incoming remote invocations. This setup procedure can be either encapsulated in a method of the remote object implementation class itself or included in another class entirely. The setup procedure should do the following:

- Create and install a security manager
- Create and export one or more remote objects
- Register at least one remote object with the RMI registry (or with another naming service, such as a service accessible through the Java Naming and Directory Interface) for bootstrapping purposes

Declaring the Remote Interfaces Being Implemented

The implementation class for the compute engine is declared as follows:

public class ComputeEngine implements Compute

This declaration states that the class implements the Compute remote interface and therefore can be used for a remote object.

The ComputeEngine class defines a remote object implementation class that implements a single remote interface and no other interfaces. The ComputeEngine class also contains two executable program elements that can only be invoked locally. The first of these elements is a constructor for ComputeEngine instances. The second of these elements is a main method that is used to create a ComputeEngine instance and make it available to clients.

Defining the Constructor for the Remote Object

The ComputeEngine class has a single constructor that takes no arguments. The code for the constructor is as follows:

```
public
   ComputeEngine()
   { super();

}
```

This constructor just invokes the superclass constructor, which is the no-argument constructor of the Object class. Although the superclass constructor gets invoked even if omitted from theComputeEngine constructor, it is included for clarity.

Providing Implementations for Each Remote Method

The class for a remote object provides implementations for each remote method specified in the remote interfaces. The Compute interface contains a single remote

method, executeTask, which is implemented as follows:

```
public <T> T
   executeTask(Task<T> t) { return
   t.execute();

}
```

This method implements the protocol between the ComputeEngine remote object and its clients. Each client provides the ComputeEngine with a Task object that has a particular implementation of the Task interface's execute method. The ComputeEngine executes each client's task and returns the result of the task's execute method directly to the client.

PROGRAMMING:

Define the interface and implimentation class of a RMI system with a method that accept a number and returns its reverse.