Java Mid-2 Important Question & Answers

1) Differentiate between Checked and Unchecked Exception with example program?

Ans:-

Checked Exception	Unchecked Exception
Checked exceptions occur at compile time.	Unchecked exceptions occur at runtime.
The compiler checks a checked exception.	The compiler does not check these types of exceptions.
These types of exceptions can be handled at the time of compilation.	These types of exceptions cannot be a catch or handle at the time of compilation, because they get generated by the mistakes in the program.
They are the sub-class of the exception class.	They are runtime exceptions and hence are not a part of the Exception class.
Here, the JVM needs the exception to catch and handle.	Here, the JVM does not require the exception to catch and handle.
Examples of Checked exceptions: • File Not Found Exception • No Such Field Exception • Interrupted Exception • No Such Method Exception • Class Not Found Exception	Examples of Unchecked Exceptions: • No Such Element Exception • Undeclared Throwable Exception • Empty Stack Exception • Arithmetic Exception • Null Pointer Exception • Array Index Out of Bounds Exception • Security Exception

Unchecked Exceptions

```
class excep1
{       public static void main(String ar[])
        {
                int a=10;
                int b=0;
                try
                {       a=a/b;
                System.out.println("This will not be printed");
                }
                catch(ArithmeticException ae)
                {
                System.out.println("Division by 0 error... change the value");
                }
                System.out.println("Quittting");
        }
}
```

Output:-

```
Division by 0 error... change the value
Quitting
```

Checked Exceptions

```
public class excep3
{
    public static void main(String ar[])
    {
            try
            {
             Class c=Class.forName(ar[0].trim());
             String name=c.getName();
             Class sc=c.getSuperclass();
             String sname=sc.getName();
           System.out.println("Name is : "+name+" and SuperClass name is :"+sname);
        }
        catch(ClassNotFoundException cnf)
        {           System.out.println("No such Class");
        }
    }
}
```

Output:-

```
java excep3 IT
```

```
No such Class
```

2) Write a Java program on User Defined Exceptions?

Ans:-

Throw (User-defined Exceptions)

```
class userdefined
{   static void throwfun(int a) throws UserException
    {  if(a<0)
            throw new UserException(a);
            System.out.println("Normal Exit");
     }
            public static void main(String args[])
            {           try
                        {           throwfun(10);
                                    throwfun(-5);
                        }
                        catch(UserException ue)
                        {
System.out.println("Exception caught: " + ue);
                        }
            }
}
```

```
class UserException extends Exception
{
            private int x;

            UserException(int a)
            {
                        x=a;
            }

            public String toString()
            {
return "IT students raised this exception "+x;
            }
}
```

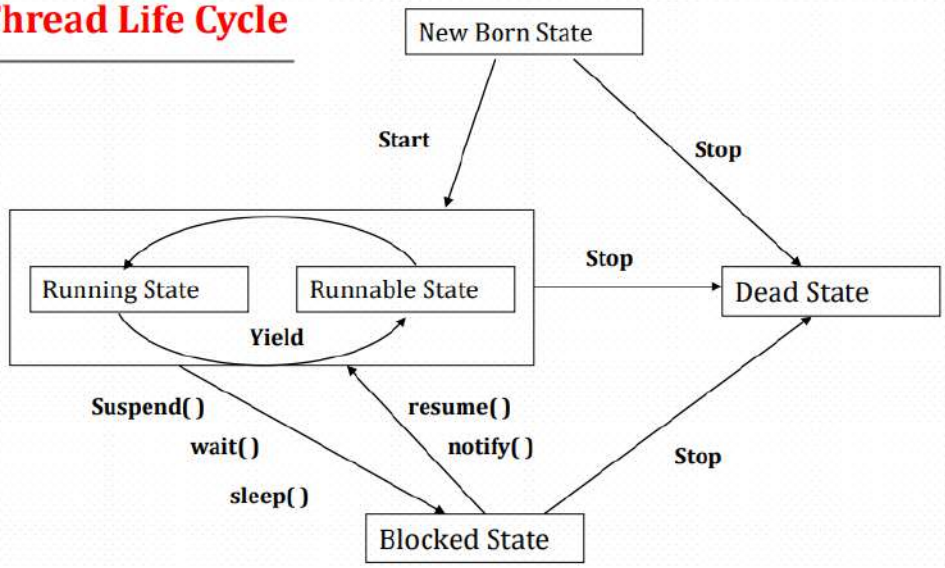
Output:-

```
Normal Exit
Exception caught:IT students raised this exception-5
```

3) With a neat diagram explain states of Thread Life Cycle in Java.

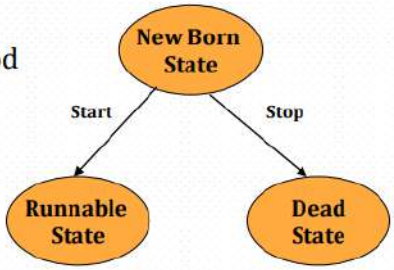
Ans:-

Thread Life Cycle



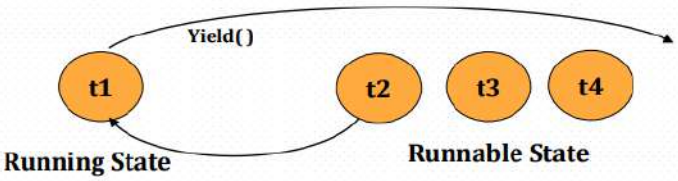
New born State

- When we create a thread object then the thread is in New born state.
- It has two alternatives
  - Schedule it for running using start() method
  - Kill it using stop() method



Runnable State

- It means thread is ready for execution and is waiting for the availability of the processor. If all threads have equal priority then they are given time slots for execution in Round Robin fashion (FIFO)
- We can move the threads from running state to runnable state by giving yield() command.



Running State

- It means the processor has given its time to the thread for its execution. The thread runs until it relinquishes the control on its own or it is preempted by a higher priority thread

Blocked State

- A thread is said to be blocked when it is prevented from entering into runnable state and subsequently running state. It is also called "Not runnable state"

Dead State

- A running thread ends its life when it is completed its execution is called " Natural Death" otherwise we can kill using a stop() message then it is called "Preemptive death"

4) How Synchronization be applied using Threads

Ans:-

Synchronization

- Key to synchronization is the concept of the monitor (also called a semaphore).
- A monitor is an object that is used as a mutually exclusive lock, or mutex. Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor.
- These other threads are said to be waiting for the monitor.

Synchronized Method

- When we declare a method synchronized, Java creates a "monitor" and hands it over to the thread that calls the method first time.
- As long as the thread holds the monitor, no other thread can enter the synchronized section of code.
- A monitor is like a key and the thread that holds the key can only open the lock.
- This is the general form of the synchronized statement:

```
synchronized method()
{
        // statements to be synchronized
}
```


Synchronization Program

```
class Callme
{       void call(String msg)
        {       System.out.print("[" + msg);
                try
                { Thread.sleep(1000); }
                catch (InterruptedException e)

        {System.out.println("Interrupted"); }
                System.out.println("]");
        }
}
 class Caller implements Runnable
{       String msg; Callme target;
        Thread t;
        public Caller(Callme targ, String s)
        {       target = targ;
                msg = s;
                t = new Thread(this);
                t.start();
        }
        public void run()
        {
                synchronized(target)
                {
                        target.call(msg);
                }
        }
}
 class Synch1
{       public static void main(String args[])
        {
Callme target = new Callme();

Caller ob1 = new Caller(target, "Hello");

Caller ob2 = new Caller(target, "Synchronized");

Caller ob3 = new Caller(target, "World");
        }
}
```

Output:-

```
[Hello]
[World]
[Synchronized]
```

5) Explain about Thread Priorities

Ans:-

Thread Priorities

- Thread priorities are used by the thread scheduler to decide when each thread should be allowed to run.
- A higher-priority thread can also preempt a lower-priority one.
- To set a thread's priority, use the setPriority( ) method, which is a member of Thread.
- final void setPriority(int level)

  Here, level specifies the new priority setting for the calling thread.

Thread Priorities

- The value of level must be within the range MIN_PRIORITY and MAX_PRIORITY. Currently, these values are 1 and 10, respectively.
- To return a thread to default priority, specify NORM_PRIORITY, which is currently 5. These priorities are defined as final variables within Thread.
- We can obtain the current priority setting by calling the getPriority( ) method of Thread, shown here:

  final int getPriority( )

Thread Priorities Program

```
class A extends Thread
{           public void run()
            {
                        for (int i=1;i<=5; i++)
                        {
System.out.println("\tFrom Thread A : i="+i);
                        }
            System.out.println("Exit From A ");
            }
}
 class B extends Thread
{           public void run()
            {
                        for (int j=1;j<=5 ;j++)
                        {
System.out.println("\tFrom Thread B : j="+j);
                        }
            System.out.println("Exit From B ");
            }
}
```

```
class C extends Thread
{
            public void run()
            {
                        for (int k=1;k<=5; k++)
                        {
System.out.println("\tFrom Thread C : K="+k);
                        }
            System.out.println("Exit From C ");
            }
}
```

Thread Priorities Program

```
class ThreadPriority
{                       public static void main(String []args)
                        {       A a=new A();
                                B b=new B();
                                C c=new C();

                                c.setPriority(Thread.MAX_PRIORITY);
                                b.setPriority(Thread.NORM_PRIORITY);
 //b.setPriority(c.getPriority()-2));
                                a.setPriority(Thread.MIN_PRIORITY);
//a.setPriority(b.getPriority()+1));
                                System.out.println("Start Thread C ");
                                c.start();
                                System.out.println("Start Thread B ");
                                b.start();
                                System.out.println("Start Thread A ");
                                a.start();
                                System.out.println("End of Main Thread ");
            }
}
```

Output:-

```
Start Thread C
Start Thread B
Start Thread A
End of Main Thread
         From Thread A: i=1
         From Thread C: k=1
         From Thread C: k=2
         From Thread C: k=3
         From Thread C: k=4
         From Thread C: k=5
         From Thread B: j=1
         From Thread B: j=2
Exit From C
         From Thread A: i=2
         From Thread B: j=3
         From Thread B: j=4
         From Thread A: i=3
         From Thread B: j=5
Exit From B
         From Thread A: i=4
         From Thread A: i=5
Exit From A
```

6) Write a Java Program for Producer Consumer Problem in Inter Thread Communication?

Ans:-

Producer Consumer Program

```
class Q
{       int n;
        boolean valueSet = false;

        synchronized int get()
        {       if(!valueSet)
                {       try
                        { wait();    }
                 catch(InterruptedException e)
                   {
        System.out.println("Exception caught");
                   }
                }

                System.out.println("Got: " + n);
                valueSet = false;
                notify();
                return n;
        }
    synchronized void put(int n)
    {
            if(valueSet)
            {
                try
                {       wait();
                }

            catch(InterruptedException e)
            {
            System.out.println("InterruptedException
caught");
            }
            }
                    this.n = n;
                    valueSet = true;
            System.out.println("Put: " + n);
                    notify();
    }
}
```

Producer Consumer Program

```
class Producer implements Runnable
{       Q q;
        Producer(Q q)
        {       this.q = q;
        new Thread(this, "Producer").start();
        }
        public void run()
        {       int i = 0;
                while(true)
                { q.put(i++);
                }
        }
}
 class Consumer implements Runnable
{       Q q;
        Consumer(Q q)
        {       this.q = q;
                new Thread(this,
"Consumer").start();
        }
        public void run()
        {
                while(true)
                {  q.get();
                }
        }
}

class PC1
{
        public static void main(String args[])
        {
                Q q = new Q();

                new Producer(q);
                new Consumer(q);

System.out.println("Press Control+C to stop.");
        }
}
```

Output:-

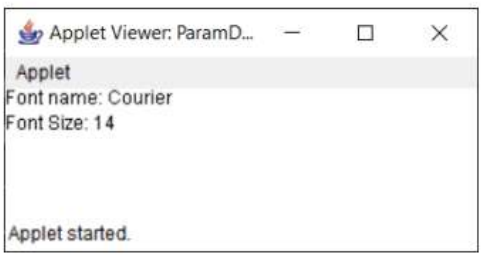
```
Press Control+C to stop.
Put:0
Got:0
Put:1
Got:1
Put:2
Got:2
Put:3
Got:3
Put:4
Got:4
Put:5
Got:5
Put:6
Got:6
Put:7
Got:7
Put:8
Got:8
Put:9
Got:9
Put:10
```

7) Write a java program to pass parameters to Applet via HTML

Ans:-

Passing Parameters to Applets: The APPLET tag in HTML allows you to pass parameters to out applet. To retrieve a parameter, use the **getParameter()** method. It returns the value of the specified parameter in the form of a **String** object.

```
// Use Parameters
import java.awt.*;
import java.applet.*;
/* <applet code="ParamDemo" width=300 height=80>
   <param name=fontName value=Courier>
        <param name=fontSize value=14>
        </applet>   */
public class ParamDemo extends Applet
{
        String fontName;
        int fontSize;
        float leading;
        boolean active;
        // Initialise the string to be displayed.

        public void start()
        {
                String param;
                fontName = getParameter("fontName");
                if(fontName==null)
                        fontName="Not Found";
                param = getParameter("fontSize");

                try
                {       if(param!=null)// if not found
                            fontSize=Integer.parseInt(param);
                        else
                                fontSize=0;
                }
                catch (NumberFormatException e)
                {         fontSize=-1;
                }
        }
        // Display parameters
        public void paint(Graphics g)
        {
                g.drawString("Font name: "+fontName,0,10);
                g.drawString("Font Size: "+fontSize,0,26);
        }
}
```



8) Discuss Applet and its life cycle.

Ans:-

Applet Architecture:

An applet is a window-based program. As such, its architecture is different from the normal, console-based programs. First, applets are event driven. Second, the user initiates interaction with an applet. For example, when the user clicks a mouse inside the applet's window, a mouse-clicked event is generated. If the user presses a key while the applet's window has input focus, a keypress event is generated. Applets can contain various controls, such as push buttons and check boxes. When the user interacts with one of these controls, an event is generated.

APPLET LIFE CYCLE

Applet Initialization:

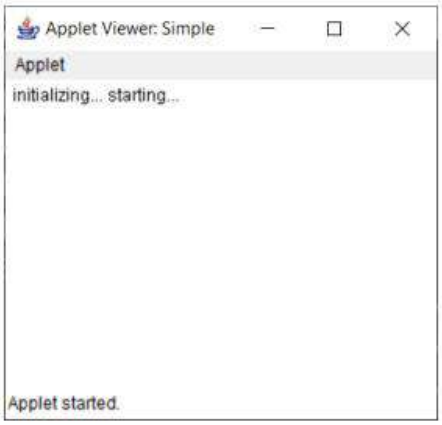
1. **init()** : The **init()** method is the first method the be called. This is where we should initialize variables. This method is called only once during the run time of our applet.
2. **start()** : The **start()** method is called after **init().** It is also called to restart an applet after it has been stopped. Whereas **init()** is called once – the first time an applet is loaded – **start()** is called each time as applet's HTML document is displayed onscreen.
3. **paint()** : The **paint()** method is called each time out applet's output must be redrawn. The **paint()** method has one parameter of type Graphics. This parameter contains the graphics context.

Applet Termination :

1. **stop()** : The **stop()** method is called when a web browser leaves the HTML document containing the applet – when it goes to another page.
2. **destroy()** : The **destroy()** method is called when the environment determines that out applet needs to be removed completely from memory. At this point, we should free up any resources the applet may be using. The **stop()** method is always called before **destroy()**.

```
import java.applet.Applet;
import java.awt.Graphics;
public class Simple extends Applet
{       StringBuffer buffer;
        public void init()
        {
                buffer = new StringBuffer();
                addItem("Initialising");
        }
        public void start()
        {
                addItem("starting....");
        }
        public void stop()
        {
                addItem("stopping....");
        }
```

```
    public void destroy()
    {
        addItem("preparing for unloading....");
    }
    private void addItem(String newWord)
    {
        System.out.println(newWord);
        buffer.append(newWord);
        repaint();
    }
    public void paint(Graphics g)
    {
        // Draw a Rectangle around the applet's display area.
        g.drawRect(0,0,getWidth()-1,getHeight()-1);
        // Draw the current string inside the rectangle
        g.drawString(buffer.toString(),5,15);
    }
}
```



9) Explain about Handling Mouse and Keyboard Events

Ans:-

The KeyListener Interface : This interface defines three methods. The **Keypressed()** and **KeyReleased()** method is invoked when a key is pressed and released, respectively. The **KeyTyped()** method is invoked when a character has been entered.

The general form of these methods are shown here:

void Keypressed(KeyEvent *ke*)

void KeyReleased(KeyEvent *ke*)

void KeyTyped(KeyEvent *ke*)

The MouseListener Interface : This interface defines five methods.

The general forms of these methods are shown here :

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

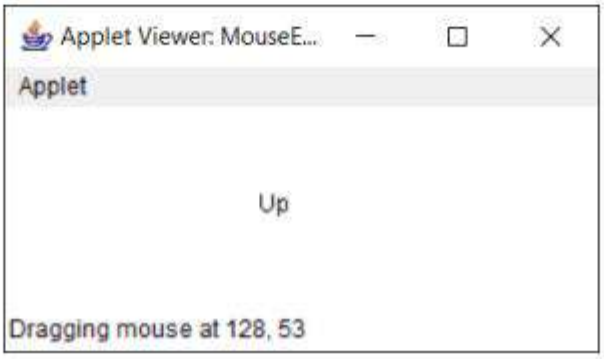
void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

```
//Demonstrate the mouse event handlers
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/
public class MouseEvents extends Applet implements MouseListener, MouseMotionListener
{       String msg = "";
        int mouseX = 0, mouseY = 0; // coordinates of mouse

        public void init()
        {       addMouseListener(this);
                addMouseMotionListener(this);
        }
        // Handle mouse clicked.
        public void mouseClicked(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse clicked.";
                repaint();
        }
        // Handle mouse entered.
        public void mouseEntered(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse entered.";
                repaint();
        }
        // Handle mouse exited.
        public void mouseExited(MouseEvent me)
        {       mouseX = 0;
                mouseY = 10;
                msg = "Mouse exited.";
                repaint();
        }
```

```
    // Handle button pressed.
    public void mousePressed(MouseEvent me)
    {       mouseX = me.getX();
            mouseY = me.getY();
            msg = "Down";
            repaint();
    }
    // Handle button released.
    public void mouseReleased(MouseEvent me)
    {       mouseX = me.getX();
            mouseY = me.getY();
            msg = "Up";
            repaint();
    }
    // Handle mouse dragged.
    public void mouseDragged(MouseEvent me)
    {       mouseX = me.getX();
            mouseY = me.getY();
            msg = "*";
            showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
            repaint();
    }
    // Handle mouse moved.
    public void mouseMoved(MouseEvent me)
    {       showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
    }
    // Display msg in applet window at current X,Y location.
    public void paint(Graphics g)
    {       g.drawString(msg, mouseX, mouseY);
    }
}
```



```
//Demonstrate the key event handlers
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/
public class SimpleKey extends Applet implements KeyListener
{
	String msg = "";
	int X = 10, Y = 20; // output coordinates
	public void init()
	{	addKeyListener(this);
		requestFocus(); // request input focus
	}
	public void keyPressed(KeyEvent ke)
	{	showStatus("Key Down");
	}
	public void keyReleased(KeyEvent ke)
	{	showStatus("Key Up");
	}
	public void keyTyped(KeyEvent ke)
	{	msg += ke.getKeyChar();
		repaint();
	}
	// Display keystrokes.
	public void paint(Graphics g)
	{	g.drawString(msg, X, Y);
	}
}
```

10) Discuss about Layout Manager Types? Write a program for Grid Layout Manager?

Ans:-

A layout manager automatically arranges our controls within a window. A layout manager is a instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout( )** method. If no calls to **setLayout( )** is made, then default layout manager is used.

FlowLayout :

FlowLayout is the default layout manager. Components are laid out from the upper-left corner, left to right and top to bottom. When on more components fit on line, the next one appears on the next line.

FlowLayout( )

FlowLayout(int *how*)

FlowLayout(int *how*,int *horz*,int *vert*)

Valid values for the *how* are as follows:

FlowLayout.LEFT

FlowLayout.CENTER

FlowLayout.RIGHT

The third form allows us to specify the horizontal and vertical space left between components in *horz* and *vert*, respectively.

BorderLayout

It has four narrow, fixed-width components at the edges and one large area in the center. The four sides are referred to as north,south,east and west. The middle area is called the center. Here are the constructors defined by **BorderLayout.**

BorderLayout( )

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

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

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

void add(Component *compObj,* Object *region*); → *compObj* is the component to be added, and *region* specifies where the component will be added.

GridBagLayout

The Grid bag layout (like grid layout) arranges components into a grid of rows and columns, but lets us specify a number of settings. Unlike the grid layout, the rows and columns are not constrained to be a uniform size. For example, a component can be set to span multiple rows or columns, or we can change its position on the grid. Each GridBagLayout object maintains a dynamic, rectangular grid of cells, with each component occupying one or more cells, called its *display* area. Each component managed by a GridBagLayout is associated with an instance of GridBagConstraints.

The constraints objects specifies where a component's display area should be located on the grid and how the component should be positioned within its display area.

Card Layout:

The CardLayout class is unique among the other layout managers in that it stores several different layouts. Each layout can be thought of as being on a separate index card in a deck that can be shuffled so that any card is on top at a given time. This can useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input.

CardLayout ()

CardLayout (int *horz*, int *vert*)

Specify the horizontal and vertical space left between components in horz and vert.

When card panels are added to a panel, they are usually given a name. Thus, most of the time, we will use this form of add () when adding cards to a panel:

Void add (Component *panelObj*, Object *name*);

Following methods defined by CardLayout:

Void first (Container *deck*)

void last(Container *deck*)

void next(Container *deck*)

void previous(Container *deck*)

void show(Container *deck*, String *cardName*)

deck is a reference to the container.

GridLayout

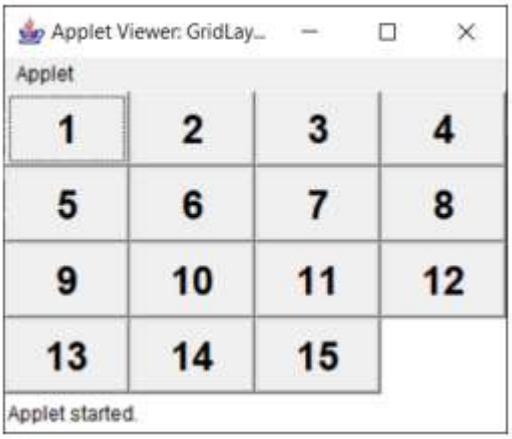
GridLayout lays out components in a tow-dimensional grid. We specify the number of rows and columns, the number of rows only and let the layout manager determine the number of columns, or the number of columns only and let the layout manager determine the number of rows.

The cells in the grid are equal size based on the largest component in the grid.

GridLayout() → creates a single-column grid layout
GridLayout(int *numRows,* int *numcolumn*) → creates a grid layout with the specified number of rows and columns.
GridLayout(int *numRow,* int *numColumn,* int *horz,* int *vert*) → specify the horizontal and vertical spaces left between components.

PROGRAM

```
//demonstrate GridLayout

import java.awt.*;
import java.applet.*;

/*
 <applet code="GridLayoutDemo" width=300 height=200>
 </applet>
*/

public class GridLayoutDemo extends Applet
{
   static final int n=4;

     public void init()
    {
       setLayout(new GridLayout(n,n));
       setFont(new Font("SansSerif",Font.BOLD,24));

                                    for(int i=0;i<n;i++)
                                     {
                                       for(int j=0;j<n;j++)
                                         {
                                                int k=i*n+j;
                                                if(k>0)
                                                  add(new Button(""+ k));
                                         }
                                     }
    }
}
```



11) Explain any eight AWT Controls or Components with example.

Ans:-

Buttons:

A push button is a component that contains a label and that generates an event when it is pressed. Push buttons are objects of type **Button. Button** defines these two constructors:

Button() -> creates an empty button

Button(String str) -> creates a button that contains *str* as a label

Void setLabel(String str)-> set its label

String getLabel() -> retrieve its label

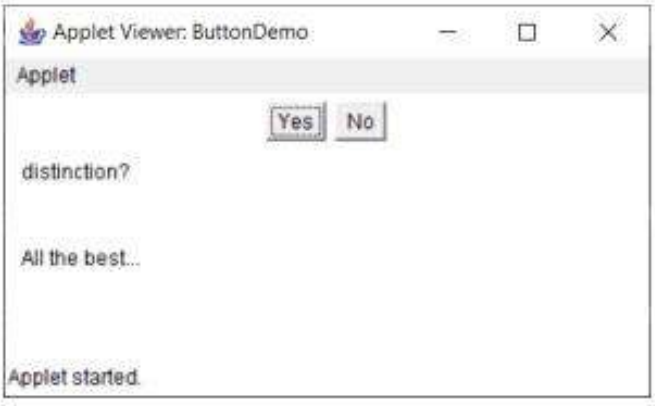
Handling Buttons

Each time a button is pressed, an action event is generated. This is sent to any listeners that previously registered. Each listener implements the **ActionListener** interface. That interface defines the **actionPerformed()** method, which is called when an event occurs. An **ActionEvent** object is supplied as the argument to this method.

//Demonstrate Buttons

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="BottonDemo" width=250 height=150>
</applet> */
```

```
public class ButtonDemo extends Applet implements ActionListener
{       String msg="";
        Button yes,no;

        public void init()
        {
                yes=new Button("yes");
                no=new Button("no");
                add(yes); add(no);
                yes.addActionListener(this);
                no.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
        {
                String str=ae.getActionCommand();
                if(str.equals("yes"))
                {
                        msg="all the best";
                }
                else if(str.equals("no"))
                {
                        msg="have high confidence in yourself";
                }
                repaint();
        }
        public void paint(Graphics g)
        {       g.drawString("distinction?"10,20);
                g.drawsString(msg,6,100);
        }
}
```



Labels:

A *label* is an object of type **Label**, and it contains a string, which it displays. Labels are passive controls that do not support any interaction with the user. **Label** defines the following constructors:

Label() -> creates a blank label.

Label(String str) -> creates a label that contains the string specified by *str*.

This string is left-justified.

Label(String *str*, int *how)* -> creates a label that contains the string specified by *str* using the alignment specified by *how*. The value of *how* must be one of these three constants: **Label.LEFT, Label.RIGHT** or **Label.CENTER.**

void setText(String *str*) -> set or change the text in the label

string getText() -> obtain the current label

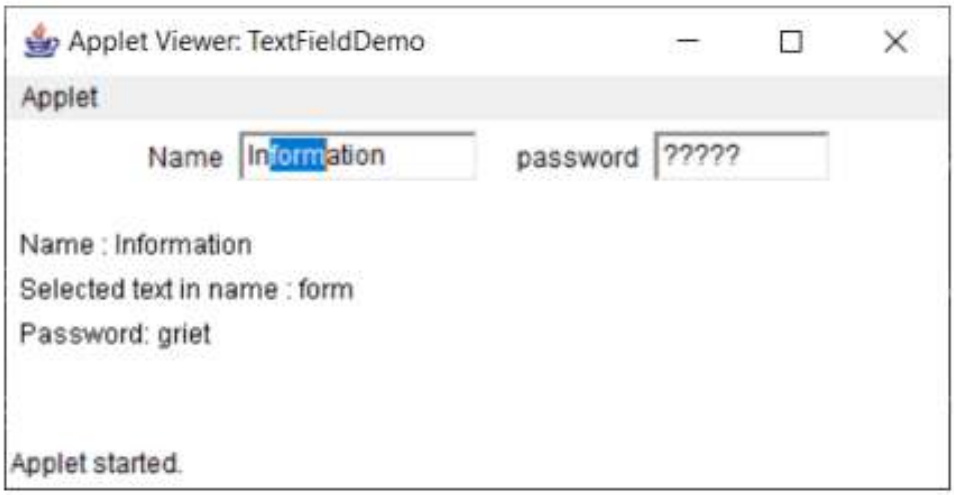
void setAlignment(int *how*) -> set the alignment of the string within the label

int getAlignment() -> obtain the current alignment

//Demonstrate Labels

```
import java.awt.*;
import java.applet.*;
/* <applet code="LabelDemo" width=300 height=200>
</applet> */
public class LabelDemo extends Applet
{       public void init()
        {       Label one=new label("one");
                Label two=new label("two");
                Label three=new label("three");
                // add labels to applet window
                add(one);  add(two);  add(three);
        }
}
```

```
                Label namep=new Label("Name",Label.RIGHT);
                Label passp=new Label("password",Label.RIGHT);

                name=new TextField(12);
                pass=new TextField(8);
                pass.setEchoChar('?');
                add(namep);  add(name);
                add(passp);   add(pass);
// register to receive action events
                name.addActionListener(this);
                pass.addActionListener(this);
        }
        //user pressed enter.

        public void actionPerformed(ActionEvent ae)
        {
                repaint();
        }
        public void paint(Graphics g)
        {
                g.drawString("name:"+name.getText(),6,60);
                g.drawString("selected text in name"+name.getSelectedText(),6,80);
                g.drawString("password:"+pass.getText(),6,100);
        }
}
```



TextArea

AWT includes a simple multiline editor called **TextArea.** Following are the constructors for **TextArea:**

TextArea()

TextArea(int *numlines,* int *numchars*)

TextArea(String *str*)

TextArea(String str, int numlines, int numchars)

TextArea(String str, int numlines, int numchars, int sBars)

Here, numlines specifies the height, in lines, of the text area, and numChars specifies its width, in characters. Initial text can be specified by str. In the fifth form we can specify the scroll bars that you want the control to have. sBars must be one of these values:

SCROLLBARS_BOTH

SCROLLBARS_NONE

SCROLLBARS_HORIZONTAL_ONLY

SCROLLBARS_VERTICAL_ONLY

TextArea is a subclass of **TextComponent.** Therefore, it supports the **getText(), setText(), getSelectedText(), select(), isEditable(),** and **setEditable()** methods

TextArea adds the following methods:

void append(String str) -> appends the string specified by str to the end

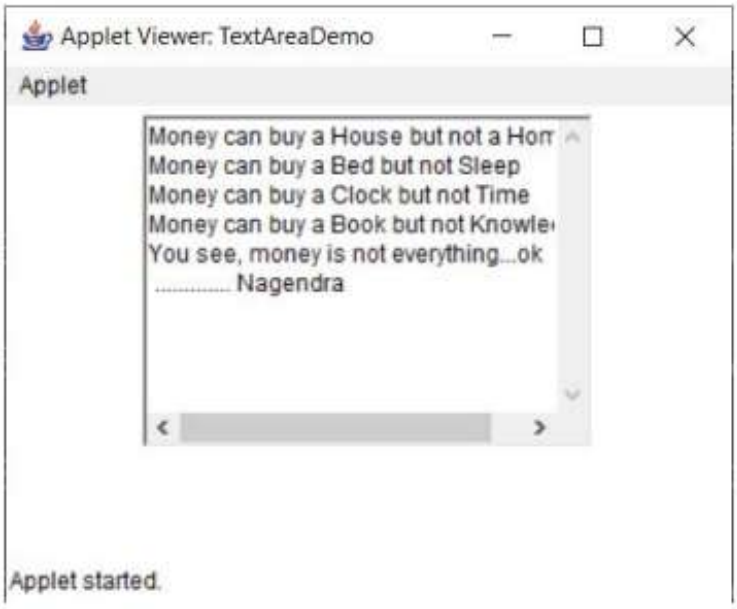
void insert(String str, int index) ->inserts the string at the specified index

void replaceRange(String str, int startIndex, int endIndex) -> it replaces the characters from startIndex to endIndex-1, with the replacement text passed in str

//Demonstrate Text Area

```
import java.awt.*;
import java.applet.*;

/*
<applet code="TextAreaDemo" width=380 height=150>
</applet>
*/
public class TextAreaDemo extends Applet
{
        public void init()
        {
```

```
        String val ="Money can buy a horse but not a home \n"+
                    "Money can buy a bed but not sleep\n"+
                    "Money can buy a clock but not time\n"+
                    "Money can buy a book but not knowledge\n"+
                    "you see money is not everything...ok\n";

    TextArea text=new TextArea(val,10,30);
    add(text);
    }
}
```



Check Boxes

A check box is a control that is used to turn an option on or off. It consists of a small box that can either contain a check mark or not. There is a label associated with each box that describes what option the box represents.

Checkbox() supports these constructors:

Checkbox() -> creates a check box whose label is initially blank

Checkbox(String str) -> creates a check box whose label is specified by str

Checkbox(String str, boolean on) -> set the initial state of the check box. If on is true, the check box is initially checked; otherwise, it is cleared

Checkbox(String str, boolean on, CheckboxGroup cbGroup)

Checkbox(String str, CheckboxGroup cbGroup, boolean on)->create a check box whose label is specified by str and whose group is specified by cbGroup. If this check box is not part of a group, then cbGroup must be null.

boolean getState() -> retrieve the current state of a check box

void setState(boolean on) -> to set its state

String getLabel() -> obtain the current label

void setLabel(String str) -> to set the label

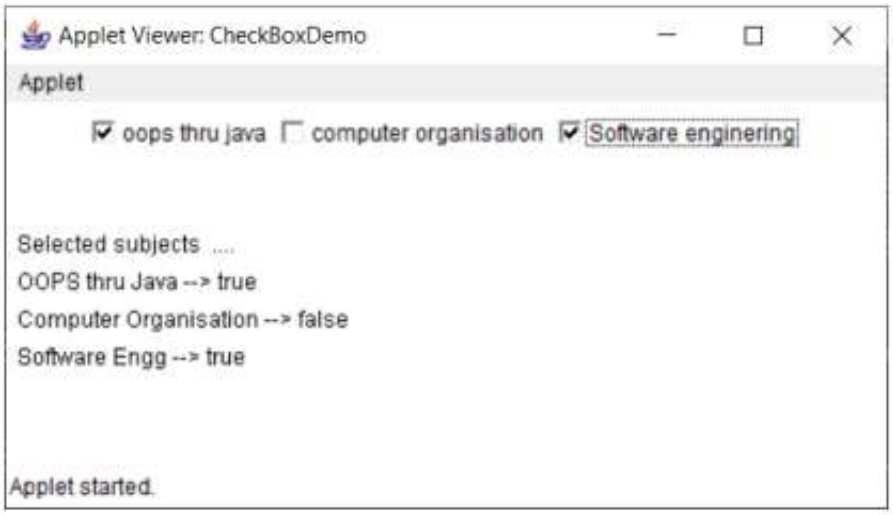
Handling Check Boxes

Each time a check box is selected or deselected, an item event is generated. This sent to any listeners that previously registered. Each listener implements the ItemListener interface. That interface defines the itemStateChanged() method. An ItemEvent object is supplied as the argument to this method. It contains information about the event.

//Demonstrate Check boxes

```
import java.awt.*;
import java.applet.*;
import java.applet.event;

/*<applet code="CheckBoxDemo" width=250 height=200>
</applet>
*/
public class CheckBoxDemo extends Applet implements ItemListener
{
        String msg="";
        Checkbox javaa,cg,se;

        public void init()
        {
                javaa=new Checkbox("oops thru java",null,true);
                cg=new Checkbox("computer graphics");
                se=new Checkbox("Software enginering");
                add(javaa);add(cg);add(se);
                javaa.addItemListener(this);
                cg.addItemListener(this);
                se.addItemListener(this);
        }

        public void itemStateChanged(ItemEvent ie)
        {
                repaint();
        }
```

```
        // display current state of the check boxes.
        public void paint(Graphics g)
        {       msg="Selected subjects";
                        g.drawString(msg,6,80);
                msg="oops thru java"+javaa.getState();
                        g.drawString(msg,6,100);
                msg="computer graphics"+cg.getState();
                        g.drawString(msg,6,120);
                msg="software engg"+se.getState();
                g.drawString(msg,6,140);
        }
}
```



CheckboxGroup

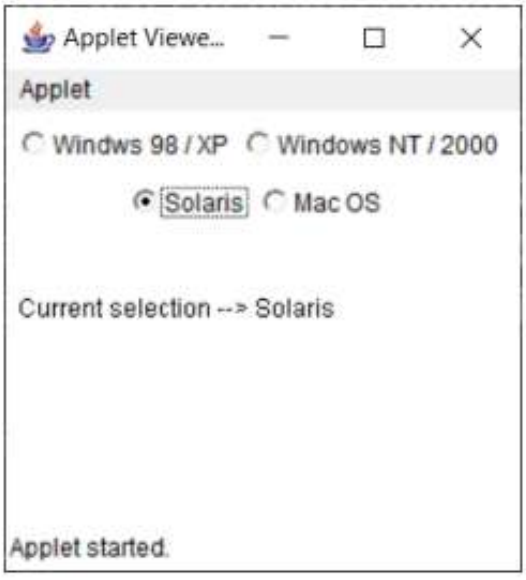
It is possible to create a set of mutually exclusive check boxes in which one and only one check box in the group can be checked at any one time. These check boxes are often called Radio buttons.

Checkbox getSelectedCheckbox() -> currently selected Checkbox
void setSelectedCheckbox(Checkbox which) -> set a check box

Program:

```
// Demonstrate check box group
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*<applet code="CBGroup" width=250 height=200>
</applet> */
```

```
public class CBGroup extends Applet implements ItemListener
{       String msg="";
        Checkbox win98,winNT,solaris,mac;
        CheckboxGroup cbg;

        public void init()
        {
                cbg=new CheckboxGroup();
                win98=new Checkbox("Windows 98/xp",cbg,true);
                winNT=new Checkbox("windows nt/2000",cbg,false);
                solaris=new Checkbox("solaris",cbg,false);
                mac=new Checkbox("mac os",cbg,false);
                add(win98); add(winNT); add(solaris); add(mac);
                win98.addItemListener(this);        winNT.addItemListener(this);
                solaris.addItemListener(this);      mac.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {       repaint();
        }
        //Display current state of the check boxes.
        public void paint(Graphics g)
        {
                msg="Current selection";
                msg+=cbg.getSelectedCheckbox().getLabel();
                g.drawString(msg,6,100);
        }
}
```



Choice

The **choice** class is used to create a *pop-up* list of items from which the user may chose.

Void add(String name) -> To add selection to the list

String getSelectedItem() -> To determine which item is currently selected, returns a string containg the name of the item

int getSelectedIndex() -> returns the index of the item

int getItemCount() ->To obtain the number of items in the list

void select(int index) -> currently selected item using index

void selected(String name) -> currently selected item using name

String getItem(int *index*) -> obtain the name associated with the item at that index

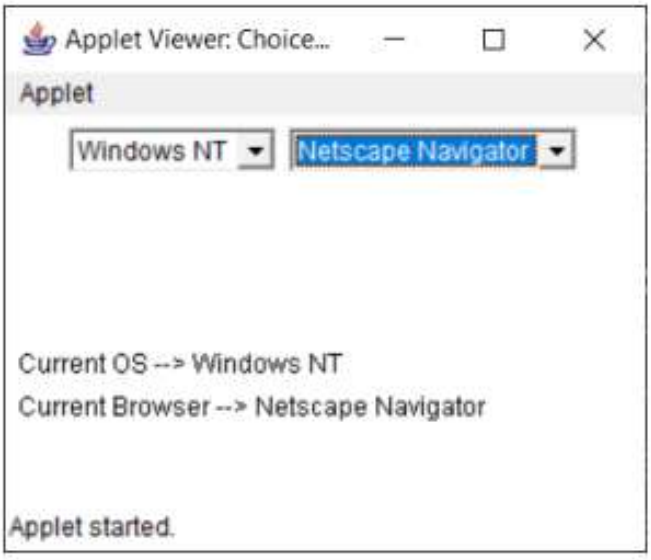
Handling Choice Lists

Each time a choice is selected, an item event is generated. This is sent to any Listeners that previously registered. Each listener implements the **ItemListener** Interface. That interface defines the **itemStateChange()** method. An **ItemEvent** Object is selected as the argument to this method.

Program:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/* <applet code="ChoiceDemo" width=300 height=180>
</applet> */

public class ChoiceDemo extends Applet implements ItemListener
{
        Choice os,browser;
        String msg="";
        public void init()
        {       os=new Choice();
                browser=new Choice();

                os.add("Windows 98");
                os.add("Windows NT");
                os.add("Solaris");

                browser.add("Mozilla");
```

```
        browser.add("Netscape Navigator");
        browser.add("IE 6");
        browser.add("Lynx 2.4");
        browser.select("IE 6");

        //add choice lists to window
        add(os);
        add(browser);
        //register to receive item events
        os.addItemListener(this);
        browser.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {       repaint();
    }
    //display current selections.

    public void paint(Graphics g)
    {
        msg="Current OS --> ";
        msg+=os.getSelectedItem();
        g.drawString(msg,6,120);

        msg="Current Browser --> ";
        msg+=browser.getSelectedItem();
        g.drawString(msg,6,140);
    }
}
```



Lists

The **list** class provides a compact, multiple-choice, scrolling selection list. Unlike the **Choice** object, which shows only the single selected item in the menu.

List provides these constructors:

List() -> allows only one item to be selected at one time

List(int numRows) -> the value of numRows specifies the number of entries in the list that will always be visible

List(int numRows, boolean multipleSelect) -> if multipleSelect is **true**, then the user may select two or more items at a time. If it is **false,** then only one item may be selected.

void add(String name) -> adds item to the end of the list

void add(String name, int index) -> adds the item at the index specified by index

Indexing begins at zero. You can specify -1 to add the item to the end of the list.

String getSelectedItem() -> returns a string containing the name of the item. If more than on item has been selected or if no selection has been made, -1 is returned.

String[] getSelectedItems() -> returns an array containing the names of the currently selected items

int[] getSelectedIndexes() -> returns an array containing the indexes of the currently selected items

int getItemCount() -> obtain the number of items in the list

void select(int index) -> currently selected item

String getItem(int index) -> obtain the name associated with the item at the index

Handling Lists

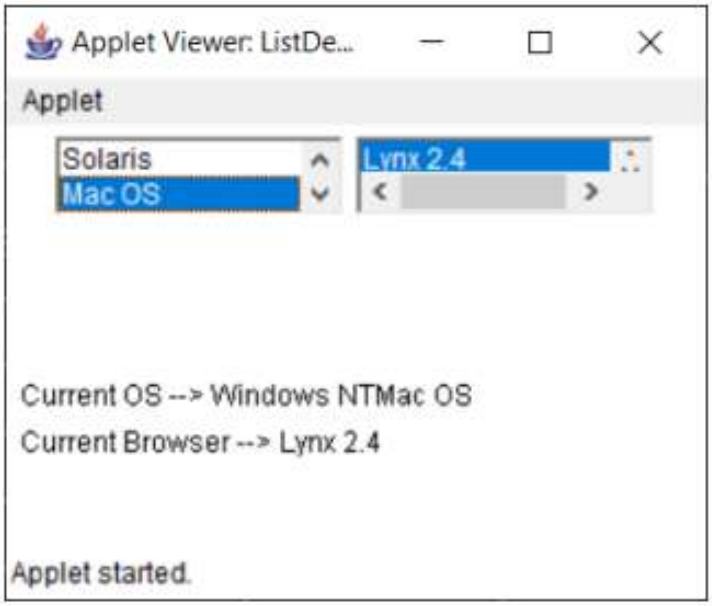
We will need to implement the ActionListener interface. Each time a List item is double-clicked, an ActionEvent object is generated.

Program:

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;

/*<applet code="ListDemo" width=300 height=180>
</applet>
*/
```

```
public class ListDemo extends Applet implements ActionListener
{
  List os,browser;
  String msg="";

  public void init()
  {       os=new List(2,true);
          browser=new List(2,false);

          //add items to os list
          os.add("Windows 98");
          os.add("Windows NT");
          os.add("Solaris");
          os.add("Mac OS");

          browser.add("Mozilla");
          browser.add("Netscape Navigator");
          browser.add("IE 6");
          browser.add("Lynx 2.4");
          browser.select(3);

          //add choice lists to window

          add(os);
          add(browser);

          //register to receive action events

          os.addActionListener(this);
          browser.addActionListener(this);
  }

  public void actionPerformed(ActionEvent ae)
  {
          repaint();
  }

  //display current selections.
```

```
public void paint(Graphics g)
{
        int idx[];

        msg="Current OS --> ";

        idx=os.getSelectedIndexes();

        for(int i=0;i<idx.length;i++)
                msg+=os.getItem(idx[i])+"";

        g.drawString(msg,6,120);

        msg="Current Browser --> ";

        msg+=browser.getSelectedItem();
        g.drawString(msg,6,140);
  }
}
```



12) Explain about JTable and JTree in Swings?

Ans:-

Tables

A *table* is a component that displays rows and columns if data. You can drag the cursor on column boundaries to resize columns. You can also drag a column to a new position. Tables are implemented by the **JTable** class, which extends **JComponent.**

JTable(Object *data*[][], Object *colHeads*[])à*data* is a two-dimensional array of the information to be presented, and *colHeads* is a one-dimensional array with the column headings.

Procedure:

1.Create a **JTable** object.

2.Create a **JScrollPane** object.

3.Add the table to the scroll pane.

4.Add the scroll pane to the content pane of the applet.

Program 7:

```
import java.awt.*;
import javax.swing.*;
/* <applet code="swing7" width=400 height=100></applet>*/
public class swing7 extends JApplet
{
  public void init()
  {
    Container contentPane=getContentPane();
    contentPane.setLayout(new BorderLayout());
    final String[] colHeads = {"Reg.No","Name","Phone No."};
```

```
    final Object[][] data={
                    {"06241A1204","Abhishek","23456789"},
                     {"06241A1246","Md.MehrajKhan","23456789"},
                     {"06241A1279","RamaKrishna","23456789"},
                     {"06241A1288","D. Sabitha","23456789"},
                     {"06241A12B3","UdithGoutham","23456789"},
                     {"07241A1201","Anvesh Reddy Ch","23456789"},
                     {"07241A1202","Bharath B","23456789"},
                     {"07241A1203","EstherManikya Latha","23456789"},
                {"07241A1205","Haritha Ch","23456789"},
                     };
    JTable table=new JTable(data,colHeads);
    int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane js=new JScrollPane(table,v,h);
    contentPane.add(js,BorderLayout.CENTER);
  }
}
```

Trees

A tree is a component that presents a hierarchical view of data. A user has the ability to expand or collapse individual subtrees in this display. Trees are implemented in Swing by the **Jtree** class, which extends **Jcomponent.**

Jtree(Hashable ht)àCreates a tree in which each element of the hash table ht is a child node

Jtree(Object obj[])àEach element of the array obj is a child node

Jtree(TreeNode tn)àThe tree node tn is the root of the tree

Jtree(Vector v)àThe elements if the vector v as child nodes

The **getPathForLocation()** method is used to translate a mouse click on a specific point of the tree to a tree path.

TreePath getPathForLocation(int x, int y)

A **JTree** object generates events when a node is expanded or collapsed.

Void addTreeExpansionListner(TreeExpansionListener tel)

Void removeTreeExpansionListner(TreeExpansionListener tel)

The **TreePath** class encapsulates information about a path to a particular node in a tree. The **TreeNode** interface declares methods that obtain information about a tree node. The **MutableTreeNode** interface extends **TreeNode.** It declares methods that can insert and remove child nodes or change the parent node. The **DefaultMutableTreeNode** class implements the **MutableTreeNode** interface.

DefaultMutableTreeNode(Object *obj*)

Void add(MutableTreeNode *child*)à*child* is a mutable tree node that is to be added as a child to the current node.

1.Create a **Jtree** object.

2.Create a **JScrollPane** object.

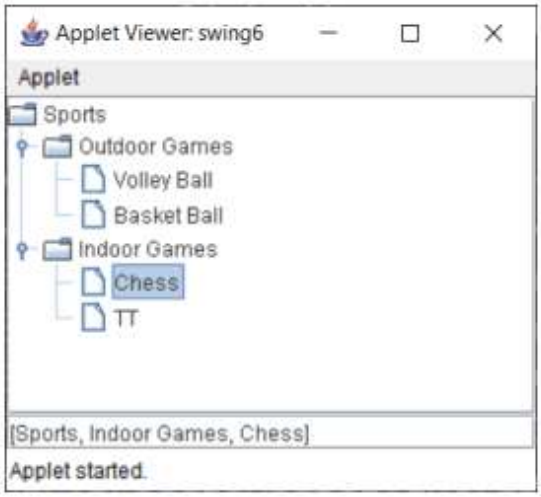
3.Add the tree to the scroll pane.

4.Add the scroll pane to the content pane of the applet.

Program 6:

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

public class swing6 extends JApplet
{
        JTree jt;
        JTextField jtext;
```

```
	public void init()
	{
		Container c = getContentPane();
		c.setLayout(new BorderLayout());

DefaultMutableTreeNode root = new DefaultMutableTreeNode("Sports");
DefaultMutableTreeNode out = new DefaultMutableTreeNode("Outdoor Games");
DefaultMutableTreeNode volley = new DefaultMutableTreeNode("Volley Ball");
	DefaultMutableTreeNode basket = new DefaultMutableTreeNode("Basket Ball");

	out.add(volley);	out.add(basket);	root.add(out);

	DefaultMutableTreeNode in = new DefaultMutableTreeNode("Indoor Games");
	DefaultMutableTreeNode chess = new DefaultMutableTreeNode("Chess");
	DefaultMutableTreeNode table = new DefaultMutableTreeNode("TT");

	in.add(chess);		in.add(table); root.add(in);

	jt = new JTree(root);
	int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
	int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;

	JScrollPane js = new JScrollPane(jt,v,h);
	c.add(js,BorderLayout.CENTER);
	jtext= new JTextField(20);
	c.add(jtext,BorderLayout.SOUTH);

	jt.addMouseListener(new MouseAdapter()
			{
				public void mouseClicked(MouseEvent me)
				{ display(me); }
			});
	}
	public void display(MouseEvent e)
	{	TreePath tp =jt.getPathForLocation(e.getX(),e.getY());
		jtext.setText(tp.toString());
	}
}
```



13) Create an application for Student details using Swing Components?

Ans:-

Program 7:

```
import java.awt.*;
import javax.swing.*;
/* <applet code="swing7" width=400 height=100></applet>*/
public class swing7 extends JApplet
{
  public void init()
  {
    Container contentPane=getContentPane();
    contentPane.setLayout(new BorderLayout());
    final String[] colHeads = {"Reg.No","Name","Phone No."};
```

```
    final Object[][] data={
                    {"06241A1204","Abhishek","23456789"},
                      {"06241A1246","Md.MehrajKhan","23456789"},
                      {"06241A1279","RamaKrishna","23456789"},
                      {"06241A1288","D. Sabitha","23456789"},
                      {"06241A12B3","UdithGoutham","23456789"},
                      {"07241A1201","Anvesh Reddy Ch","23456789"},
                      {"07241A1202","Bharath B","23456789"},
                      {"07241A1203","EstherManikya Latha","23456789"},
                {"07241A1205","Haritha Ch","23456789"},
                      };
    JTable table=new JTable(data,colHeads);
    int v=ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h=ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane js=new JScrollPane(table,v,h);
    contentPane.add(js,BorderLayout.CENTER);
  }
}
```

All the best

Now do the Rest…