14CS IT405

II/IV B.Tech (Regular/Supplementary) Degree Examination

Scheme of Evaluation

April, 2017 **Common for CSE & IT**

Fourth Semester **GUI Programming**

Answer Question No 1 compulsorily (1x12=12marks)

Answer One Question from each Unit (4x12=48marks)

I.Answer the following.

a)Define an object

An object is an instance of a class.

b) Explain the use of super keyword

super can be used to refer immediate parent class instance variable.
super can be used to invoke immediate parent class method.
super() can be used to invoke immediate parent class constructor.

c) List different access specifiers in java.

public
private
protected
default

d) Write String functions.

startsWith(), endsWith(), charAt(),length(),replace()

e) What is a collection in java

Collections in java is a framework that provides an architecture to store and manipulate the group of objects.

f) What are the two ways used for creating user defined exceptions?

Extending the Exception class , one can create their own exception.

g) What is delegation event model?

Delegation event model is a simple concept in which a source generates an event and sends it to one or more listeners.

h) What are the attributes of applet tag.

<applet code=”example.class” width=”300” height=”300”>

i)Write the syntax for creating simple panel in awt

Panel pan1=new Panel();

j) List some event classes in java

ActionEvent, MouseEvent, MouseWheelEvent, KeyEvent, ItemEvent, TextEvent, AdjustmentEvent, WindowEvent

k) Write short note on AWT components.

Java AWT components are platform-dependent i.e. components are displayed according to the view of operating system. AWT is heavyweight i.e. its components are using the resources of OS.

l) Differentiate process and thread.

Process	Thread
1.Process is a program in execution. 2.A process consists of multiple threads. 3. Often referred as task. 4. Has its own address space.	1 .A Thread is a subset of the process 2 .A thread is smallest part of process. 3 .Often referred as lightweight process 4 . Uses process's address space and share it.

UNIT-I

2a) Define a class which consist of properties and methods for Student. (6m)

```
import java.util.Scanner;
public class GetStudentDetails
{
    public static void main(String args[])
    {
        String name;
        int roll, math, phy, eng;

        Scanner SC=new Scanner(System.in);
        System.out.print("Enter Name: ");
        name=SC.nextLine();
        System.out.print("Enter Roll Number: ");
        roll=SC.nextInt();
        System.out.print("Enter marks in Maths, Physics and English: ");
        math=SC.nextInt();
        phy=SC.nextInt();
        eng=SC.nextInt();
        int total=math+eng+phy;
        float perc=(float)total/300*100;
        System.out.println("Roll Number:" + roll +"\tName: "+name);
        System.out.println("Marks (Maths, Physics, English): " +math+","+phy+","+eng);
        System.out.println("Total: "+total +"\tPercentage: "+perc);
    }
}
```

2b) What is interface? Explain with an example program how an interface can extend another interface. (6m)

Interfaces are syntactically similar to classes, but they lack instance variables, and, their methods are declared without any body

Defining an Interface

An interface is defined much like a class. This is a simplified general form of an interface:

access interface *name* {
return-type method-name1(*parameter-list*);
return-type method-name2(*parameter-list*);
type final-varname1 = value;
type final-varname2 = value;
//...
return-type method-nameN(*parameter-list*);
type final-varnameN = value;
}

Interfaces Can Be Extended

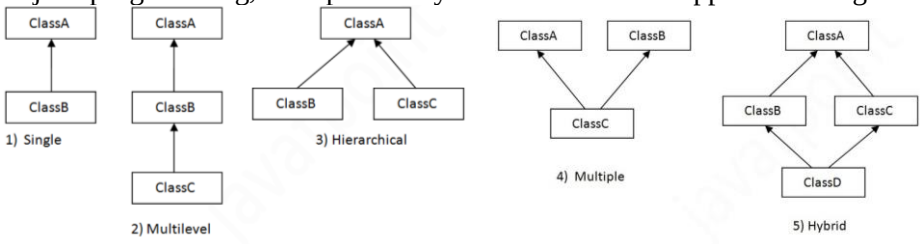
One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface,it must provide implementations for all methods required by the interface inheritance chain.

```
import java.lang.*;
import java.io.*;
interface A
{
        void show1();
        void show2();
}
interface B extends A
{
        void show3();
}
class C implements B
{
        public void show1()
        {
                System.out.println("This is interface A's method");
        }
        public void show2()
        {
                System.out.println("This is also interface A'smethod");
        }
        public void show3()
        {
                System.out.println("This is interface B's method extends from A");
        }
}
class Main
{
        public static void main(String args[])
        {
                C obj=new C();
                obj.show1();
                obj.show2();
                obj.show3();
        }
}
```

3a) Define inheritance? Explain the types of inheritance with examples. (12m)

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
In java programming, multiple and hybrid inheritance is supported through interface only.



Single level inheritance

```
class Bank
{
	int getRateOfInterest()
	{
		return 0;
	}
}

class SBI extends Bank
{
	int getRateOfInterest()
	{
		return 8;
	}
}
class Test2
{
	public static void main(String args[])
	{
		SBI s=new SBI();
		System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
	}
}
```

Multilevel Inheritance

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX();
    obj.methodY();
   obj.methodZ();
```

```
  }
}
```

Hierarchical inheritance

```
class Bank
{
	int getRateOfInterest()
	{
		return 0;
	}
}

class SBI extends Bank
{
	int getRateOfInterest()
	{
		return 8;
	}
}

class ICICI extends Bank
{
	int getRateOfInterest()
	{
		return 7;
	}
}
class AXIS extends Bank
{
	int getRateOfInterest()
	{
		return 9;
	}
}

class Test2
{
	public static void main(String args[])
	{
		SBI s=new SBI();
		ICICI i=new ICICI();
		AXIS a=new AXIS();
		System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
		System.out.println("ICICI Rate of Interest:"+i.getRateOfInterest());
		System.out.println("AXIS Rate of Interest:"+a.getRateOfInterest());
	}
}
```

Multiple Inheritance

```
interface Printable{
void print();
```

```
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

UNIT-II

4a) Write a java program to use command line arguments. (6m)

A command-line argument is the information that directly follows the program's name on the command line when it is executed. To access the command-line arguments inside a Java program is quite easy—they are stored as strings in a **String** array passed to the **args** parameter of **main( )**. The first command-line argument is stored at **args[0]**, the second at **args[1]**, and so on.

A simple program to demonstrate command line arguments:

```
class cmd
{
        public static void main(String[] args)
        {
                int n=args.length;
                System.out.println("no of args= "+ n);
                System.out.println("the  args are:");
                for(int i=0;i<n;i++)
                System.out.println(args[i]);
        }
}
```

4b) Write the basic interfaces of java collection. (6m)

The Collections Framework defines several core interfaces. the collection interfaces is necessary because they determine the fundamental nature of the collection classes.

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy
Deque	Extends **Queue** to handle a double-ended queue.
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches.
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements

SortedSet	**Extends** Set to handle SortedSets.

Collection: interface Collection<E>
List: interface List<E>
Set: interface Set<E>
SortedSet: interface SortedSet<E>
Queue: interface Queue<E>

5a) Explain throws and finally keywords in java with example program. (6m)

throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.

```
class ThrowsDemo
{
        static void throwOne() throws IllegalAccessException
        {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[])
        {
                try
                {
                        throwOne();
                }
                 catch (IllegalAccessException e)
                 {
                        System.out.println("Caught " + e);
                }
        }
}
```

finally:

finally creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

```
class demo
{
    public static void main(String args[])
    {
        try
        {
                System.out.println("open files");
                int n=args.length;
                System.out.println("n= " + n);
                int a=45/n;
                System.out.println("a= " + a);
        }
        catch(ArithmetricException ae)
```

```
        {
                System.out.println(ae);
                System.out.println
                ("please pass data while running this program");
        }
        finally
        {
                System.out.println("close files");
        }
    }
```

}**5b) Explain the concept of synchronization in Java with an example program. (6m)**

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*.
Java Synchronization is better option where we want to allow only one thread to access the shared resource.
The synchronization is mainly used to
To prevent thread interference.
To prevent consistency problem.

```
    import java.lang.*;
    import java.io.*;
    class Reserve implements Runnable
    {
        int available =1;
        int wanted;
        Reserve(int i)
        {
            wanted=i;
        }
        public void run()
        {
            synchronized(this)
            {
                System.out.println("Available Berths:"+available);
                if(available>=wanted)
                {
                    String name= Thread.currentThread().getName();
                    System.out.println(wanted+"Berths reserved for "+name);
                    try
                    {
                        Thread.sleep(1500);
                        available = available - wanted;
                    }
                    catch(InterruptedException ie)
                    {
                    }
                }
                else
                {
                    System.out.println("Sorry, No Berths...!");
                }
            }
        }
    }
```

```
class safe
{
        public static void main(String args[])
        {
                Reserve obj = new Reserve(1);
                Thread t1 = new Thread(obj);
                Thread t2 = new Thread(obj);
                t1.setName("bec");
                t2.setName("cse");
                t1.start();
                t2.start();
        }
   }
```

UNIT-III

6a) Define an Applet? Explain the life cycle of an applet with an example program (6m)

Applets are small applications that are accessed on an Internet server, transported over the Internet, automatically installed, and run as part of a web document.

Lifecycle of Java Applet

Applet is initialized.
Applet is started.
Applet is painted.
Applet is stopped.
Applet is destroyed

Four methods in the Applet class gives you the framework on which you build any serious applet –

init − This method is intended for whatever initialization is needed for your applet. It is called after the param tags inside the applet tag have been processed.

start − This method is automatically called after the browser calls the init method. It is also called whenever the user returns to the page containing the applet after having gone off to other pages.

stop − This method is automatically called when the user moves off the page on which the applet sits. It can, therefore, be called repeatedly in the same applet.

destroy − This method is only called when the browser shuts down normally. Because applets are meant to live on an HTML page, you should not normally leave resources behind after a user leaves the page that contains the applet.

paint − Invoked immediately after the start() method, and also any time the applet needs to repaint itself in the browser. The paint() method is actually inherited from the java.awt.

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

/*<applet code="AppletLifeCycle.class" width="350" height="150"> </applet>*/

public class AppletLifeCycle extends Applet
{
   public void init()
   {
     setBackground(Color.CYAN);
     System.out.println("init() called");
```

```
    }
    public void start()
    {
      System.out.println("Start() called");
    }
    public void paint(Graphics g)
    {
      System.out.println("Paint(() called");
    }
    public void stop()
    {
      System.out.println("Stop() Called");
    }
    public void destroy()
    {
      System.out.println("Destroy)() Called");
    }
}
```

6b) Write a Java application to handle Mouse Events and MouseMotion Events. (6m)

```
/*<applet code=scribblepad height=300 width=300></applet>*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class scribblepad extends Applet implements MouseListener,MouseMotionListener
{
      int x=0,y=0,nx=20,ny=20;
      String msg=" ";
      public void init()
      {
            addMouseListener(this);
            addMouseMotionListener(this);
      }
      public void mouseEntered(MouseEvent me)
      {
            msg="Mouse Entered("+me.getX()+","+me.getY()+")";
      }
      public void mouseExited(MouseEvent me){}
      public void mouseClicked(MouseEvent me)
      {
            x=me.getX();
            y=me.getY();
      }
      public void mousePressed(MouseEvent me){}
      public void mouseReleased(MouseEvent me){}
      public void mouseMoved(MouseEvent me){}
      public void mouseDragged(MouseEvent me){
      nx=me.getX();
      ny=me.getY();
      repaint();
}
public void update(Graphics g)
```

```
{
    showStatus(msg);
    g.setColor(Color.black);
    g.drawLine(x,y,nx,ny);
    x=nx;
    y=ny;
}
}
```

7a) Explain types of streams (I/O) with example. (12m)

Java programs perform I/O through streams. A *stream* is an abstraction that either produces or consumes information

Java defines two types of streams: byte and character. *Byte streams* provide a convenient means for handling input and output of bytes. Byte streams are used, for example, when reading or writing binary data. *Character streams* provide a convenient means for handling input and output of characters

The Byte Stream Classes

Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream.**

BufferedInputStream: Buffered input stream
BufferedOutputStream: Buffered output stream
ByteArrayInputStream: Input stream that reads from a byte array
ByteArrayOutputStream: Output stream that writes to a byte array
DataInputStream: An input stream that contains methods for reading the Java standard data types
DataOutputStream: An output stream that contains methods for writing the Java standard data types
FileInputStream: Input stream that reads from a file
FileOutputStream: Output stream that writes to a file
FilterInputStream: Implements **InputStream**
FilterOutputStream: Implements **OutputStream**
InputStream: Abstract class that describes stream input
ObjectInputStream: Input stream for objects
ObjectOutputStream: Output stream for objects
OutputStream: Abstract class that describes stream output
PipedInputStream: Input pipe
PipedOutputStream: Output pipe
PrintStream: Output stream that contains **print( )** and **println( )**
PushbackInputStream: Input stream that supports one-byte "unget," which returns a byte to the input stream
SequenceInputStream: Input stream that is a combination of two or more input streams that will be read sequentially, one after the other.

```
import java.io.*;
class ShowFile
 {
        public static void main(String args[])
        {
                int i;
                FileInputStream fin;
                if(args.length != 1)
                 {
                        System.out.println("Usage: ShowFile filename");
                        return;
                }
                try
                 {
```

```
                fin = new FileInputStream(args[0]);
            }
            catch(FileNotFoundException e)
            {
                System.out.println("Cannot Open File");
                return;
            }
            try
            {
                do
                {
                    i = fin.read();
                    if(i != -1)
                        System.out.print((char) i);
                }
                while(i != -1);
            }
            catch(IOException e)
            {
                System.out.println("Error Reading File");
            }
            try
            {
                fin.close();
            }
            catch(IOException e)
            {
                System.out.println("Error Closing File");
            }
        }
}
```

The Character Stream Classes

Character streams are defined by using two class hierarchies. At the top are two abstract classes: **Reader** and **Writer**. These abstract classes handle Unicode character streams.

BufferedReader: Buffered input character stream
BufferedWriter: Buffered output character stream
CharArrayReader: Input stream that reads from a character array
CharArrayWriter: Output stream that writes to a character array
FileReader: Input stream that reads from a file
FileWriter: Output stream that writes to a file
FilterReader: Filtered reader
FilterWriter: Filtered writer
InputStreamReader: Input stream that translates bytes to characters
LineNumberReader: Input stream that counts lines
OutputStreamWriter: Output stream that translates characters to bytes
PipedReader: Input pipe
PipedWriter: Output pipe
PrintWriter: Output stream that contains **print( )** and **println( )**
PushbackReader: Input stream that allows characters to be returned to the input stream
Reader: Abstract class that describes character stream input

StringReader: Input stream that reads from a string
StringWriter: Output stream that writes to a string
Writer: Abstract class that describes character stream output

```
// Use a BufferedReader to read characters from the console.
import java.io.*;
class BRRead {
public static void main(String args[]) throws IOException
{
char c;
BufferedReader br = new
BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter characters, 'q' to quit.");
// read characters
do {
c = (char) br.read();
System.out.println(c);
} while(c != 'q');
}
}
```

UNIT-IV

8a) Write a Java Program to display simple File dialog box with supported AWT components (6m)

```
import java.awt.*;
import java.awt.event.*;
public class MainClass extends Frame
{
    FileDialog fc;
    MainClass()
    {
        super("MainClass");
        setSize(200, 200);
        setVisible(true);

        fc = new FileDialog(this, "Choose a file", FileDialog.LOAD);
        /*fc.setDirectory("C:\\");*/

        Button b;
        add(b = new Button("Browse..."));
      b.addActionListener(new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                fc.setVisible(true);
                String fn = fc.getFile();
                if (fn == null)
                System.out.println("You cancelled the choice");
                else
                System.out.println("You chose " + fn);
            }
        });
    }
    public static void main(String[] args)
    {
```

```
            new MainClass();
        }
}
```

8b) Explain various adapter classes **(6m)**

An adapter class provides an empty implementation of all
methods in an event listener interface. Adapter classes are useful when you want to receive and process only some of the events that are handled by a particular event listener interface.

ComponentAdapter : ComponentListener
ContainerAdapter: ContainerListener
FocusAdapter: FocusListener
KeyAdapter: KeyListener
MouseAdapter: MouseListener and MouseMotionListener and MouseWheelListener
MouseMotionAdapter: MouseMotionListener
WindowAdapter: WindowListener, WindowFocusListener, and WindowStateListener

9a) Explain the creation of JTABLES with an example program. **(6m)**

JTable

JTable is a component that displays rows and columns of data
Here are the steps required to set up a simple **JTable** that can be used to display data:
1. Create an instance of **JTable**.
2. Create a **JScrollPane** object, specifying the table as the object to scroll.
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane.

```
import java.awt.*;
import javax.swing.*;
import javax.swing.table.*;
import java.awt.event.*;
class JTableDemo extends JFrame implements ActionListener{
        DefaultTableModel model;
        JTable deptt;
        JButton ab,rb;
        JPanel bp;
        JScrollPane tjp;
        JTableDemo(){
                super("Department Table");
                model=new DefaultTableModel();

                model.addColumn("Dept No");
                model.addColumn("Dept Name");
                model.addColumn("Dept Location");
                String[] row={"1","CSE","BEC"};
                model.addRow(row);
                String[] row1={"2","ECE","BEC"};
                model.addRow(row1);
                String[] row2={"3","EEE","BEC"};
                model.addRow(row2);
                deptt=new JTable(model);
                tjp=new JScrollPane(deptt);
                add(tjp);
                ab=new JButton("Add Button");
                rb=new JButton("Remove Button");
                bp=new JPanel();
                bp.setLayout(new FlowLayout());
                bp.add(ab);
```

```
                bp.add(rb);
                ab.addActionListener(this);
                rb.addActionListener(this);
                add(bp,BorderLayout.NORTH);
                setVisible(true);
                pack();
        }
        public void actionPerformed(ActionEvent ae){
                if(ae.getSource()==ab){
                        String[] row={"","",""};
                        model.addRow(row);
                }
                else if(ae.getSource()==rb){
                        model.removeRow(deptt.getSelectedRow());
                }
        }
        public static void main(String[] args){
                new JTableDemo();
        }

}
```

9b) Discuss about JTabbedPane and how it can helpful to create multiple tabs? (6m)

JTabbedPane encapsulates a *tabbed pane.* It manages a set of components by linking them with tabs. Selecting a tab causes the component associated with that tab to come to the forefront.

The general procedure to use a tabbed pane is outlined here:

1. Create an instance of **JTabbedPane**.
2. Add each tab by calling **addTab( )**.
3. Add the tabbed pane to the content pane

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
class JTPDemo extends JFrame{
    JTabbedPane jtp;
    JTPDemo(){
        jtp=new JTabbedPane();
        JPanel jp=new JPanel();
        jp.add(new Label("Hai"));
        jtp.addTab("Temp Converter",jp);
        jtp.addTab("JCB Demo",new JCBDemo());
        add(jtp);
        setVisible(true);

        pack();
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] rags){
        new JTPDemo();
    }
}
class TConverter extends JPanel implements ActionListener{
    JLabel cl,fl;
    JTextField ct,ft;
```

```
	JButton cb,sb;
	TConverter(){

		cl=new JLabel("Celsius");
		fl=new JLabel("Fahrenheit");
		ct=new JTextField(20);
		ft=new JTextField(20);
		cb=new JButton("Clear");
		sb=new JButton("Convert");
		setLayout(new FlowLayout());
		add(cl);
		add(ct);
		add(sb);
		add(cb);
		add(fl);
		add(ft);
		setVisible(true);
		setSize(200,200);

		sb.addActionListener(this);
		cb.addActionListener(this);
	}
	public void actionPerformed(ActionEvent ae){
		if(ae.getSource()==sb){
			String c=ct.getText();
			Double ctemp=Double.parseDouble(c);
			Double ftemp=(double)(9/5*ctemp)+32.0;
			ft.setText(ftemp.toString());
		}
		else{
			ft.setText("");
			ct.setText("");
		}
	}
}

class JCBDemo extends JPanel implements ItemListener{
	Object[] states={"AP","TG","TN","KN","MH"};
	String[] caps={"HYD","HYD","Chennai","Ben","Mumbai"};
	JComboBox scb;
	JTextField ctf;
	JLabel sl,cl;
	JPanel sp,cp;
	JCBDemo(){

		sl=new JLabel("Choose ur State");
		cl=new JLabel("Capital");
		scb=new JComboBox(states);
		ctf=new JTextField(20);
		sp=new JPanel();
		cp=new JPanel();
		sp.setLayout(new FlowLayout());
		sp.add(sl);
```

```
            sp.add(scb);
            cp.setLayout(new FlowLayout());
            cp.add(cl);
            cp.add(ctf);
            add(sp,BorderLayout.NORTH);
            add(cp);
            scb.addItemListener(this);

    }
    public void itemStateChanged(ItemEvent ie){
            int i=scb.getSelectedIndex();
            ctf.setText(caps[i]);
    }

}
```

Prepared By:

(K. Arun Babu)	**(SK. Nazeer)**
Assistant Professor	**HOD**
Dept. of CSE	**Dept. of CSE**