

B. E.

Eighth Semester Examination, Dec. 2008

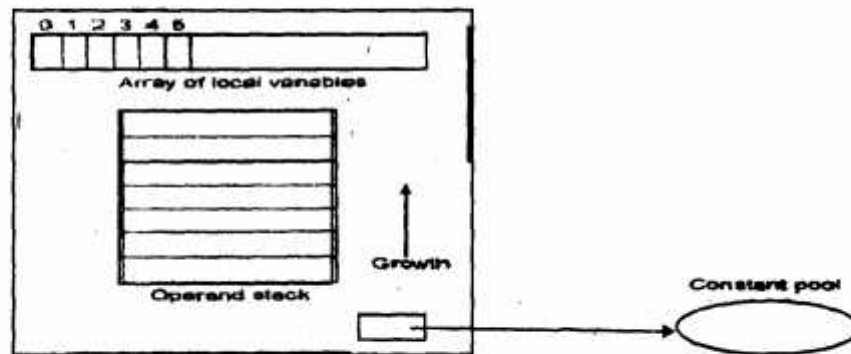
ADVANCE JAVA

Note : Attempt any five questions.

Q. 1. (a) What do you mean by byte code? Explain its working.

Ans. Byte Code :

Java bytecode is the form of instructions that the Java virtual machine executes. Each bytecode instruction or opcode is one byte in length; however, not all of the possible 256 instructions are used. In fact, Sun Microsystems, the original creators of the Java programming language, the Java virtual machine and other components of the Java Runtime Environment (JRE), have set aside 3 values to be permanently unimplemented. The most common language targeting Java Virtual Machine by producing Java bytecode is Java. Originally only one compiler existed, the javac compiler from Sun Microsystems, which compiles Java source code to Java bytecode; but because all the specifications for Java bytecode are now available, other parties have supplied compilers that produce Java bytecode. The Java Virtual Machine has currently no built-in support for dynamically typed languages, because the existing JVM instruction set is statically typed - in the sense that method calls have their signatures type-checked at compile time, without a mechanism to defer this decision to run time, or to choose the method dispatch by an alternative approach. A JVM is a stack-based machine. Each thread has a JVM stack which stores frames. A frame is created each time a method is invoked, and consists of an operand stack, an array of local variables, and a reference to the runtime constant pool of the class of the current method. Conceptually, it might look like this :



The array of local variables, also called the local variable table, contains the parameters of the method and is also used to hold the values of the local variables. The parameters are stored first, beginning at index 0. If the frame is for a constructor or an instance method, the reference is stored at location 0. Then location 1 contains the first formal parameter, location 2 the second, and so on. For a static method, the first formal method

parameter is stored in location 0, the second in location 1, and so on.

The size of the array of local variables is determined at compile time and is dependent on the number and size of local variables and formal method parameters. The operand stack is a LIFO stack used to push and pop values. Its size is also determined at compile time. Certain opcode instructions push values onto the operand stack; others take operands from the stack, manipulate them, and push the result. The operand stack is also used to receive return values from methods.

```
public String employeeName( )
{return name;}

Method java.lang.String employeeName( )
0 aload_0
1 getfield #5 <Field java.lang.String name>
4 areturn
```

The bytecode for this method consists of three opcode instructions. The first opcode, `aload_0`, pushes the value from index 0 of the local variable table onto the operand stack. Earlier, it was mentioned that the local variable table is used to pass parameters to methods. The `this` reference is always stored at location 0 of the local variable table for constructors and instance methods. The `this` reference must be pushed because the method is accessing the instance data, name, of the class. The next opcode instruction, `getfield`, is used to fetch a field from an object. When this opcode is executed, the top value from the stack, `this`, is popped. Then the #5 is used to build an index into the runtime constant pool of the class where the reference to name is stored. When this reference is fetched, it is pushed onto the operand stack. The last instruction, `areturn`, returns a reference from a method. More specifically, the execution of `areturn` causes the top value on the operand stack, the reference to name, to be popped and pushed onto the operand stack of the calling method.

Q. 1. (b) Why is Java more suitable as compared to other languages?

Ans. Java More suitable as compared to other languages: Features are as follows :

- * Platform Independence.
 - The Write-Once-Run-Anywhere ideal has not been achieved (tuning for different platforms usually required), but closer than with other languages.
- * Compiler/Interpreter Combo.
 - Code is compiled to bytecodes that are interpreted by a Java virtual machines (JVM).
 - This provides portability to any machine for which a virtual machine has been written.
 - The two steps of compilation and interpretation allow for extensive code checking and improved security.

*** Robust :**

- Exception handling built-in, strong type checking (that is, all data must be declared an explicit type), local variables must be initialized.

*** Several dangerous features of C & C++ eliminated :**

- No memory pointers.
- No preprocessor
- Array index limit checking

*** Automatic Memory Management :**

- Automatic garbage collection - memory management handled by JVM.

*** Security :**

- No memory pointers.
- Programs runs inside the virtual machine sandbox.
- Array index limit checking.
- *Code psthologies reduced by*

*** Bytecode verifier - checks classes after loading.**

*** Class loader- confines objects to unique namespaces. Prevents loading a hacked "java.lang.SecurityManager" class, for example.**

*** Security manager - determines what resources a class can access such as reading and writing to the local disk.**

*** Threading.**

- Lightweight processes, called threads, can easily be spun off to perform multiprocessing.
- Can take advantage of multiprocessors where available.
- Great for multimedia displays.

*** Built-in Networking :**

- Java was designed with networking in mind and comes with many classes to develop sophisticated

Internet communications.

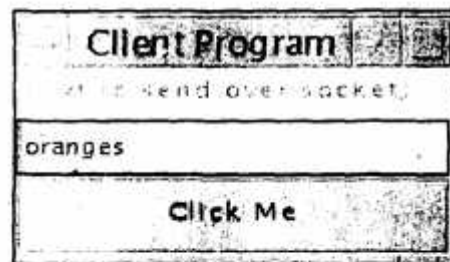
Q. 2. (a) Describe how a server sends data to a client.

Ans. Socket :

A socket is a software endpoint that establishes bidirectional communication between a server program and one or more client programs. The socket associates the server program with a specific hardware port on the machine where it runs so any client program anywhere in the network with a socket associated with that same port can communicate with the server program. A server program typically provides resources to a network of client programs. Client programs send requests to the server program, and the server program responds to the request. One way to handle requests from more than one client is to make the server program multi-threaded. A multi-threaded server creates a thread for each communication it accepts from a client. A thread is a sequence of instructions that run independently of the program and of any other threads. Using threads, a multi-threaded server program can accept a connection from a client, start a thread for that communication, and continue listening for requests from other clients.

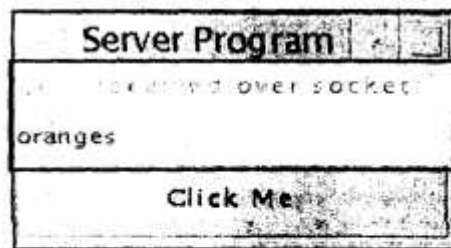
Client-Side Behavior :

The client program presents a simple user interface and prompts for text input. When you click the Click Me button, the text is sent to the server program. The client program expects an echo from the server and prints the echo it receives on its standard output.



Server-side behavior :

The server program presents a simple user interface, and when you click the Click Me button, the text received from the client is displayed. The server echoes the text it receives whether or not you click the Click Me button.



Server-Side Program :

The server program establishes a socket connection on Port 4321 in its listen Socket method. It reads data sent to it and sends that same data back to the server in its action Performed method.

Q. 2. (b) How a client application can read a file from a server through a URL connection, discuss with a suitable program?

Ans. Making connection :

listenSocket Method: The listenSocket method creates a ServerSocket object with the port number on which the server program is going to listen for client communications. The port number must be an available port, which means the number cannot be reserved or already in use. For example, Unix systems reserve ports 1 through 1023 for administrative functions leaving port numbers greater than 1024 available for use.

```
public void listenSocket() { try { server = new ServerSocket(4321); } catch (IOException e)
{ System.out.println("Could not listen on port 4321"); System.exit(-1); }
```

Next, the listenSocket method creates a Socket connection for the requesting client. This code executes when a client starts up and requests the connection on the host and port where this server program is running. When the connection is successfully established, the server. accept method returns a new Socket object

```
try { client = server.accept(); } catch (IOException e) {
System.out.println("Accept failed: 4321"); System.exit(-1); }
```

Then, the listen Socket method creates a BufferedReader object to read the data sent over the socket connection from the client program. It also creates a PrintWriter object to send the data received from the client back to the server.

```
try { in = new BufferedReader(new InputStreamReader( client.getInputStream()));
out = new PrintWriter(client.getOutputStream(), true);
```

```
} catch (IOException e) { System.out.println("Read failed"); System.exit(-1);}}
```

Lastly, the listenSocket method loops on the input stream to read data as it comes in from the client and writes to the output stream to send the data back.

```
while(true){ try{ line = in.readLine();//Send data back to client  
out. println(line); } catch (IOException e) {System.out.println("Read failed");System.exit(-  
1); } }
```

Action performed method :

The action Performed method is called by the Java platform for action events such as button clicks. This action Performed method uses the text stored in the line object to initialize the textArea object so the retrieved text can be displayed to the end user.

```
public void action Performed (ActionEvent event) {Object source = event.getSource( ); if(source ==  
button){  
textArea.setText(line); }}
```

Client-side program :

The client program establishes a connection to the server program on a particular host and port number in its listen Socket method, and sends the data entered by the end user to the server program in its action Performed method. The action Performed method also receives the data back from the server and prints it to the command line.

Listen socket method :

The listen Socket method first creates a Socket object with the computer name (kq6py) and port number (4321) where the server program is listening for client connection requests. Next, it creates a PrintWriter object to send data over the socket connection to the server program. It also creates a BufferedReader object to read the text sent by the server back to the client

```
public void listenSocket( ){//Create socket connection  
  
try{ socket = new Socket("kq6py", 4321); out = new PrintWriter(socket.getOutputStream( ),  
true);  
  
in = new BufferedReader(new InputStreamReader(socket.getInputStream()));  
  
} catch (UnknownHostException e) { System.out.println("Unknown host: kq6py"); System.exit(1);  
  
} catch (IOException e) {System.out.println("No I/O"); System.exit(1);}}
```

Action performed method :

The action Performed method is called by the Java platform for action events such as button clicks. This action Performed method code gets the text in the Textfield object and passes it to the PrintWriter object, which then sends it over the socket connection to the server program. The action Performed method then makes the Textfield object blank so it is ready for more end user input. Lastly, it receives the text sent back to it by the server and prints the text out.

```
public void actionPerformed(ActionEvent event){ Object source = event.getSource( ); if(source == button){ //Send data over socket

String text = textField.getText( ); out.println(text); textField.setText(new String("")); out.println(text); }

//Receive text from server

try{ String line = in.readLine(); System.out.println("Text received: " + line); } catch (IOException e){

System.out.println("Read failed"); System.exit(1);}}
```

Q. 3. (a) What is usage of prepared statement? Discuss with an example.

Ans. Usage of prepared statement :

It is more convenient to use a Prepared Statement object for sending SQL statements to the database. This special type of statement is derived from the more general class, Statement, that you already know. If you want to execute a Statement object many times, it normally reduces execution time to use a Prepared Statement object instead. The main feature of a Prepared Statement object is that, unlike a Statement object, it is given an 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 Prepared Statement object contains not just an SQL statement, but an SQL statement that has been precompiled. This means that when the Prepared Statement is executed, the DBMS can just run the Prepared Statement SQL statement without having to compile it first. Although Prepared Statement 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. Examples of this are in the following sections :

Creating a prepared statement object :

As with Statement objects, you create PreparedStatement objects with a Connection method. Using our open connection con from previous examples, you might write code such as the following to create a PreparedStatement object that takes two input parameters :

```
PreparedStatement update Sales = con. prepare Statement(

"UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE?");
```

The variable `updateSales` now contains the SQL statement, "UPDATE COFFEES SET SALES = ? WHERE COF_NAME LIKE ?", which has also, in most cases, been sent to the DBMS and been precompiled.

Q. 3. (b) Write a program to connect to a database, query it and display the result.

Ans. Connection to database In java : There are two sources of database connections—either a `DataSource` or a `DriverManager`. If available, JNDI and the `DataSource` interface should be used to get a `Connection` instead of `DriverManager`. The JNDI style is typical when using an application server or a web container.

Options for specifying the connection parameters include :

- Server configuration settings (likely the most common style).
- Direct user input for user name and password.
- A properties file, web.xml file, or `ResourceBundle` to keep parameters out of compiled code.
- The `jdbc.drivers` property of the `System` class.

Example :

```
import java.sql. *;import javax.naming. *;import javax.sql. *;

final class GetConnection { /** Uses JNDI and DataSource (preferred style). */

static Connection getJNDIConnection(){ String DATASOURCE_CONTEXT = "java:comp/env/jdbc/blah";
Connection result = null; try { Context initialContext = new InitialContext();

If( initialContext == null){ log("JNDI problem. Cannot get InitialContext"); }

DataSource datasource = (DataSource)initialContext.lookup(DATASOURCE-CONTEXT);

if( datasource != null) { result = datasource.getConnection(); }

else { log("Failed to lookup datasource."); } }

catch (NamingException ex ) { log("Cannot get connection: " + ex); }

catch(SQLException ex){ log("Cannot get connection: " + ex); } return result; }

/** Uses DriverManager. */

Static Connection getSimpleConnection() { //See your driver documentation for the proper format of this
string :
```

```
String DB_CONN_STRING = "jdbc:mysql://localhost:3306/airplanes";  
  
//Provided by your driver documentation. In this case, a MySql driver is used :  
  
String DRIVER_CLASS_NAME = "org.gjt.mm.mysql.Driver"; String USER_NAME = "juliex";  
  
String PASSWORD = "ui893djf";  
  
Connection result = null; try { Class.forName(DRIVER_CLASS_NAME).newInstance(); }  
catch (Exception ex) { log("Check classpath. Cannot load db driver : " + DRIVER_CLASS_NAME); }  
  
try { result = DriverManager.getConnection(DB_CONN_STRING, USER_NAME, PASSWORD); }  
catch (SQLException e) { log("Driver loaded, but cannot connect to db: " + DB_CONN_STRING); }  
  
return result; }  
  
private static void log(Object aObject) { System.out.println(aObject); } }
```

Q. 4. (a) What is SOAP? Explain various remote method calls with SOAP.

Ans. SOAP :

SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) as its message format and usually relies on other Application Layer protocols, most notably Remote Procedure Call (RPC) and HTTP for message negotiation and transmission. SOAP can form the foundation layer of a web services protocol stack, providing a basic messaging framework upon which web services can be built. As a layman's example of how SOAP procedures can be used, a SOAP message could be sent to a web service enabled web site, for example, a house price database, with the parameters needed for a search. The site will return an XML-formatted document with the resulting data (prices, location, features, etc). As the data is returned in a standardized machine-parseable format, it may be integrated directly into a third-party site. The SOAP architecture consists of several layers of specifications for message format, message exchange patterns (MEPs), underlying transport protocol bindings, message processing models, and protocol extensibility. SOAP is the successor of XML-RPC, though it borrows its transport and interaction neutrality and the envelope/header/body from elsewhere, probably from WDDX.

Remote method calls with SOAP: Remote procedure calls in SOAP are essentially client-server interactions over HTTP where the request and response comply with SOAP encoding rules. The Request-URI (Universal Resource Identifier) in HTTP is typically used at the server end to map to a class or an object, but this is not mandated by SOAP. Additionally, the HTTP header SOAPAction specifies the interface name (a URI) and the name of the method to be called on the server. The SOAP message is an XML document whose root element, the Envelope, specifies the overall structure of the message, its intended recipient, and other at-

tributes of the message. SOAP specifies a remote procedure call convention, which includes the representation and format to be used for calls and responses. A method call is modeled as a compound data element consisting of a sequence of fields (accessors), one for each parameter. A return structure consists of the return value as well as the out and in/out parameters. SOAP encoding rules specify the serialization for primitive and application-defined datatypes. SOAP allows hierarchically structured queries and responses, and specifies serialization of primitive string, numeric and date datatypes, and aggregates like arrays and vectors. Sparse arrays, and protocols for sending parts of them are also supported. New types may be defined using the <complexType> construct inside a schema definition. Overall, SOAP provides many advantages. Unfortunately, its universality comes with a performance penalty: XML messages are textual and so the sizes of its messages are significantly larger than protocols which send binary data. Since a distinguishing characteristic of scientific computation is the need to handle large data sets, the performance of SOAP relative to specialized protocols that can use binary representations is an important issue. The next section tests SOAP performance relative to other communication protocols.

Q. 4. (b) What is distributed computing, discuss it with reference to Remote Method Invocation.

Ans. Distributed computing :

Distributed computing is a method of computer processing in which different parts of a program are run simultaneously on two or more computers that are communicating with each other over a network. Distributed computing is a type of segmented or parallel computing, but the latter term is most commonly used to refer to processing in which different parts of a program run simultaneously on two or more processors that are part of the same computer. While both types of processing require that a program be segmented-divided into sections that can run simultaneously, distributed computing also requires that the division of the program take into account the different environments on which the different sections of the program will be running. For example, two computers are likely to have different file systems and different hardware components. The Java Remote Method Invocation (RMI) mechanism and the Common Object Request Broker Architecture (CORBA) are the two most important and widely used distributed object systems

The client/server model :

The client/server model is a form of distributed computing in which one program (the client) communicates with another program (the server) for the purpose of exchanging information. In this model, both the client and server usually speak the same language--a protocol that both the client and server understand--so they are able to communicate. While the client/server model can be implemented in various ways, it is typically done using low-level sockets. Using sockets to develop client/server systems means that we must design a protocol, which is a set of commands agreed upon by the client and server through which they will be able to communicate. As an example, consider the HTTP protocol that provides a method called GET, which must be implemented by web servers and used by web clients (browsers) in order to retrieve documents.

The distributed objects model :

A distributed object-based system is a collection of objects that isolates the requesters of services (clients) from the providers of services (servers) by a well-defined encapsulating interface. In other words, clients are isolated from the implementation of services as data representations and executable code. This is one of the main differences that distinguishes the distributed object-based model from the pure client/server model.

In the distributed object-based model, a client sends a message to an object, which in turn interprets the message to decide what service to perform. This service, or method, selection could be performed by either the object or a broker. The Java Remote Method Invocation (RMI) and the Common Object Request Broker Architecture (CORBA) are examples of this model.

RMI:

RMI is a distributed object system that enables you to easily develop distributed Java applications. Developing distributed applications in RMI is simpler than developing with sockets since there is no need to design a protocol, which is an error-prone task. In RMI, the developer has the illusion of calling a local method from a local class file, when in fact the arguments are shipped to the remote target and interpreted, and the results are sent back to the callers.

The Genesis of an RMI Application: Developing a distributed application using RMI involves the following steps :

1. Define a remote interface.
2. Implement the remote interface.
3. Develop the server.
4. Develop a client.
5. Generate Stubs and Skeletons, start the RMI registry, server, and client.

File transfer application :

This application allows a client to transfer (or download) any type of file (plain text or binary) from a remote machine. The first step is to define a remote interface that specifies the signatures of the methods to be provided by the server and invoked by clients.

Define a remote interface :

The remote interface for the file download application is shown in following Code. The interface `FileInterface` provides one method `download File` that takes a `String` argument (the name of the file) and returns the data of the file as an array of bytes.

```
import java.rmi.Remote; import java.rmi. RemoteException; public interface FileInterface extends Remote
{
    public byte[] downloadFile(String fileName) throws RemoteException;}
}
```

Q. 5. (a) Discuss the Bean writing process.

Ans. Bean writing process :

Q. 5. (b) What are Java Beans? How do they facilitate component orient software construction?

Ans. Java beans :

The JavaBeans architecture is based on a component model which enables developers to create software units called components. Components are self-contained, reusable software units that can be visually assembled into composite components, applets, applications, and servlets using visual application builder tools. JavaBean components are known as beans. A set of APIs describes a component model for a particular language. The JavaBeans API specification describes the core detailed elaboration for the JavaBeans component architecture. Beans are dynamic in that they can be changed or customized. Through the design mode of a builder tool you can use the Properties window of the bean to customize the bean and then save (persist) your beans using visual manipulation. You can select a bean from the toolbox, drop it into a form, modify its appearance and behavior, define its interaction with other beans, and combine it and other beans into an applet, application, or a new bean. The following list briefly describes key bean concepts.

- * Builder tools discover a bean's features (that is, its properties, methods, and events) by a process known as introspection. Beans support introspection in two ways :
 - By adhering to specific rules, known as design patterns, when naming bean features. The Introspector class examines beans for these design patterns to discover bean features. The Introspector class relies on the core reflection API. The trail The Reflection API is an excellent place to learn about reflection.
 - By explicitly providing property, method, and event information with a related bean information class. A bean information class implements the BeanInfo interface. A BeanInfo class explicitly lists those bean features that are to be exposed to application builder tools.
- * Properties are the appearance and behavior characteristics of a bean that can be changed at design time. Builder tools introspect on a bean to discover its properties and expose those properties for manipulation.
- * Beans expose properties so they can be customized at design time. Customization is supported in two ways: by using property editors, or by using more sophisticated bean customizers.

- * Beans use events to communicate with other beans. A bean that is to receive events (a listener bean) registers with the bean that fires the event (a source bean). Builder tools can examine a bean and determine which events that bean can fire (send) and which it can handle (receive).
- * Persistence enables beans to save and restore their state. After changing a bean's properties, you can save the state of the bean and restore that bean at a later time with the property changes intact. The JavaBeans architecture uses Java Object Serialization to support persistence.
- * A bean's methods are no different from Java methods, and can be called from other beans or a scripting environment. By default all public methods are exported.

Beans vary in functionality and purpose. You have probably met some of the following beans in your programming practice :

- * GUI (graphical user interface).
- * Non-visual beans, such as a spelling checker Animation applet.
- * Spreadsheet application.

Q. 6. (a) What are component organizers? How are swing components different from AWT components?

Ans. Swing components vs. AWT components :

There are, of course, both pros and cons to using either set of components from the JFC in your Java applications. Here is a summary :

AWT : Pros Speed :

Use of native peers speeds component performance.

Applet Portability :

Most Web browsers support AWT classes so AWT applets can run without the Java plugin. Look and Feel: AWT components more closely reflect the look and feel of the as they run on.

Cons :

Portability' use of native peers creates platform specific limitations. Some components may not function at all on some platforms.

Third party development :

The majority of component makers, including Borland and Sun, base new component development on Swing components. There is a much smaller set of AWT components available, thus placing the burden on the

programmer to create his or her own AWT-based components.

Features :

AWT components do not support features like icons and tool-tips.

Swing" pros portability :

Pure Java design provides for fewer platform specific limitations.

Behavior Pure Java design allows for a greater range of behavior for Swing components since they are not limited by the native peers that AWT uses.

Features : Swing supports a wider range of features like icons and pop-up tool-tips for components.

Vendor Support :

Swing development is more active. Sun puts much more energy into making Swing robust.

Look and Feel :

The pluggable look and feel lets you design set of GUI components that can automatically have the look and feel of any OS platform (Microsoft Windows, Solaris, Macintosh, etc.). It also makes it easier to make global changes to your Java programs that provide greater accessibility (like picking a hi-contrast color scheme or changing all the fonts in all dialogs, etc.).

Cons : Applet Portability :

Most Web browsers do not include the Swing classes, so the Java plugin must be used.

Performance :

Swing components are generally slower and buggier than AWT, due to both the fact that they are pure Java and to video issues on various platforms. Since Swing components handle their own painting (rather than using native API's like DirectX on Windows) you may run into graphical glitches.

Look and feel :

Even when swing components are set to use the look and feel of the OS they are run on, they may not look like their native counterparts.

In general, AWT components are appropriate for simple applet development or development that targets a specific platform (i.e. the Java program will run on only one platform).

For most any other Java GUI development you will want to use Swing components. Also note that the Borland value-added components included with JBuilder, like dbSwing and JBCL, are based on Swing components so if you wish to use these components you will want to base your development on Swing.

Q. 6. (b) Discuss these components with implementation :

(i) Lists

(ii) Trees.

Ans.

(i) List : import java.awt BorderLayout; import java.awt.event.ActionEvent;

import java. awt. event. Action Listener; import javax.swing. DefaultListSelectionModel;

import javax.swing.JButton; import javax.swing.JFrame;import javax.swing.JList;

import javax.swing.JPanel; import javax.swing.JScrollPane;import javax.swing.ListSelectionModel;

import javax.swing.event ListSelectionEvent; import javax.swing.event ListSelectionListener;

public class SimpleList2 extends JPanel {String label[] = { "Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine", "Ten", "Eleven"};

JList list; public SimpleList2() { setLayout(new BorderLayout()); list = new JList(label);

JButton button=new JButton("Print"); JScrollPane pane = new JScrollPane(list);

DefaultListSelectionModel m = new DefaultListSelectionModel();

m. setSelectionMode(ListSelectionModel. SINGLE- SELECTION);

m. setLeadAnchorNotification Enabled(false); list.setSelectionModel(m);

list.addListSelectionListener(new ListSelectionListener() { public void valueChanged(ListSelection

Event e) { System.out.println(e.toString());} }); button.addActionListener(new PrintListener());

add(pane, BorderLayout.NORTH); add(button, BorderLayout.SOUTH); }

public static void main(String s[]) { JFrame frame = new JFrame("List Example");

frame. setDefaultCloseOperation(J Frame. EXIT_ON - CLOS E);

frame.setContentPane(new SimpleList2()); frame.pack(); frame.setVisible(true);}

// An inner class to respond to clicks on the Print button

```
class PrintListener implements ActionListener { public void actionPerformed(ActionEvent e) {  
  
    int selected[] = list.getSelectedIndices(); System.outprintln("Selected Elements: ");  
  
    for (int i = 0; i < selected.length; i++) {String element = (String)  
list.getModel().getElementAt(selected[  
  
i]);    System.outprintln(" "+ element); } } }.
```

(ii) **Tree** : import java.awt.BorderLayout; import java. awt. Container;

```
import javax.swing.JFrame;import javax.swing.JScrollPane; import javax.swing.JTree;
```

```
public class TreeSample {public static void main(String args[]) {JFrame f= new JFrame("JTree  
Sample");f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); Container content =  
f.getContentPane();
```

- JTree tree = new JTree(); JScrollPane scrollPane = new JScrollPane(tree); content.add(scrollPane,
BorderLayout.CENTER); f.setSize(300, 200); f.setVisible(true); } }.

Q. 7. (a) How Coordinate Transformations are implemented in AWT? How images are manipulated?

Ans. Coordinate transformation in AWT :

Java 2D allows you to easily translate, rotate, scale, or shear the coordinate system. This capability is very convenient: moving the coordinate system is often much easier than calculating new coordinates for each of your points. Besides, for some data structures like ellipses and strings, the only way to create a rotated or stretched version is through a transformation. The meanings of translate, rotate, and scale are clear: to move, to spin, or to stretch/shrink evenly in the x and/or y direction. Shear means to stretch unevenly: an x shear moves points to the right, based on how far they are from the y-axis; a y shear moves points down, based on how far they are from the x-axis. The easiest way to picture what is happening in a transformation is to imagine that the person doing the drawing has a picture frame that he lays down on top of a sheet of paper. The drawer always sits at the bottom of the frame. To apply a translation, you move the frame (also moving the drawer) and do the drawing in the new location. You then move the frame back to its original location, and what you now see is the final result Similarly, for a rotation, you spin the frame (and the drawer), draw, then spin back to see the result Similarly for scaling and shears: modify the frame without touching the underlying sheet of paper, draw, then reverse the process to see the final result An outside observer watching this process would see the frame move in the direction specified by the transformation but see the sheet of paper stay fixed. On the other hand, to the person doing the drawing, it would appear that the sheet of paper moved in the opposite way from that specified in the transformation but that he didn't move at all. You can also perform complex transformations by directly manipulating the underlying arrays that control the transformations. This type of manipulation is a bit more complicated to envision than the basic translation, rotation, scaling, and shear transformations. The idea is that a new point (x_2, y_2) can be derived from an original point (x_1, y_1) as follows :

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}x_1 + m_{01}y_1 + m_{02} \\ m_{10}x_1 + m_{11}y_1 + m_{12} \\ 1 \end{bmatrix}$$

Image manipulation :

Because Java has been conceived with networking as a primary application, the image support is written in such a way that images can be transferred around a network, as is often seen on web pages containing Java applets. To achieve this the image loading function is non-blocking, that is, a request for the image is posted, and the program which made the request will carry on executing while not waiting for the image to load. The filename of an image can be in the form of a URL, for pulling images from a remote location, or a simple filename, for loading images from the current local directory. The get/mage function is part of AWT's Toolkit class which is a class containing native functions - functions specific for the platform on which they are running. Only Components, and classes which inherit from one, have toolkits, and they are accessed by the Component's get Too/kit function.

For example :

```
class test extends Component { test()      /* Get the toolkit from this component */
Toolkit t = getToolkit();/* Begin a retrieval of a remote image */
Image i = t.getImage( ..http://www.ecs.soton.ac.uk/~dpd98r/gfx1Mavis2.jpg.. );}
```

Q. 7. (b) Write a program to print a text on screen when a particular key has been hit.

Ans. Printing text when a particular key is pressed :

For key input, the Canvas class provides three callback methods : keyPressed(), keyReleased(), and keyRepeated(). As the names suggest, key Pressed() is called when a key is pressed, keyRepeated() is called when the user holds down the key for a longer period of time, and key Release() is called when the user releases the key. All three callback methods provide an integer parameter, denoting the Unicode character code assigned to the corresponding key. If a key has no Unicode correspondence, the given integer is negative. MIDP defines the following constnat for the keys of a standard ITU-T keypad : KEY_NUM0, KEY_NUM1, KEY_NUM2, KEY_NUM3, KEY_NUM4, KEY_NUM5, KEY_NUM6, KEY_NUM7, KEY_NUM8, KEY_NUM9, KEY_POUND, and KEY_STAR. Applications should not rely on the presence of any additional key codes. In particular, upper-and lowercase or characters generated by pressing a key multiple times are not supported by low-level key events. A "name" assigned to the key can be queried using the getKeyname() method. Some keys may have an additional meaning in games. For this purpose, MIDP provides the constants UP, DOWN, LEFT, RIGHT, FIRE, GAME_A, GAME_B, GAME_C, and GAME_D. The "game" meaninhg of a keypress can be determjined by calling the getGameAction() method. The mapping from key codes to game actions is device dependent, so different keys may map to the same game action on different devices. For example, some devices may have separate cursor keys; others may map the number pad to four-way movement. Also, several keys may be mapped to the same game code. The game code can be translated back to key code using the getKeyCode() method. This also offers a way to get the name of the key assigned to a game action. For example, the help screen of an applicaltion may display.

```
"press" + getKeyname (getKeyCode(GAME_A))
```

instead of "press GAME_A."

The following canvas implementation shows the usage of the key event methods. For each key pressed, repeated, or released, it shows the event type, character and code, key name, and game action. The first part of the implementation stores the event type and code in two variables and schedules a repaint whenever a key event occurs :

```
import javax.microedition.lcdui.*; class KeyDemoCanvas extends Canvas {  
    String eventType = "- Press any!"; int keyCode;  
    public void key Pressed (int keyCode) { eventType = "pressed"; this.keyCode = keyCode;  
        repaint(); }  
    public void key Released (int keyCode) { eventType = "released"; this.keyCode = keyCode;  
        repaint(); }  
    public void keyRepeated (int keyCode) {eventType = "repeated"; this.keyCode = keyCode;  
        repaint(); }
```

The second part prints all event properties available to the device screen. For this purpose, you first implement an additional write() method that helps the paint() method to identify the current y position on the screen. This is necessary because drawText() does not advance to a new line automatically. The write() method draws the string at the given y position and returns the y position plus the line height of the current font, so paint() knows where to draw the next line :

```
public int write (Graphics g, int y, String s) { g.drawString (s, 0, y, Graphics.LEFT|Graphics.TOP);  
    return y + g.getFont().getHeight(); }
```

The paint() method analyzes the keyCode and prints the result by calling the write() method defined previously, as shown in

```
public void paint (Graphics g) { g.setGrayScale (255); g.fillRect (0, 0, getWidth(), getHeight());  
    g.setGrayScale (0);  
    int y = 0; y = write (g, y, "Key " + eventType);  
    if (keyCode == 0) return;  
    y = write (g, y, "Char/Code: " + ((keyCode < 0) ? "N/A" : "" + (char) keyCode) + "/" + keyCode);  
    y = write (g, y, "Name: " + getKeyname (keyCode)); String gameAction;  
    switch (getGameAction (keyCode)) { case LEFT : gameAction = "LEFT"; break;  
    case RIGHT: gameAction = "RIGHT"; break; case UP : gameAction = "UP"; break;  
    case DOWN: gameAction = "DOWN"; break; case FIRE : gameAction = "FIRE"; break;  
    case GAME_A : gameAction = "GAME_A"; break; case GAME_B: gameAction = "GAME_B"; break;  
    case GAME_C: gameAction = "GAME_C"; break; case GAME_D: gameAction = "GAME_D"; break;
```

```
default: gameAction = "N/A"; } write (g, y, "Action: "+gameAction); } }.
```

Q. 8. Write short notes on :

(i) Class loaders

(ii) Multi threading

(iii) Applets

(iv) Security Managers.

Ans. (i) Class loaders :

The class loader concept, one of the cornerstones of the Java virtual machine, describes the behavior of converting a named class into the bits responsible for implementing that class. Because class loaders exist, the Java run time does not need to know anything about files and file systems when running Java programs. Classes are introduced into the Java environment when they are referenced by name in a class that is already running. There is a bit of magic that goes on to get the first class running (which is why you have to declare the main() method as static, taking a string array as an argument), but once that class is running, future attempts at loading classes are done by the class loader. At its simplest, a class loader creates a flat name space of class bodies that are referenced by a string name. The method definition is: `Class r = loadClass(String className, boolean resolve)`.

(ii) Multi-threading :

Concurrent programming.

- * Writing programs divided into independent tasks.
- * Tasks may be executed in parallel on multiprocessors.
- * Multithreading.
- * Executing program with multiple threads in parallel.
- * Special form of multiprocessing.
- * Two approaches.
- * Thread class.

```
public class Thread extends Object { ... }
```

- * Runnable interface.

```
public interface Runnable { public void run(); // work => thread }
```

(iii) Applet :

"An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application. The Applet class provides a standard interface between applets and their environment. "

Four definitions of applet :

- * A small application.

- A secure program that runs inside a web browser.
- A subclass of java.applet.Applet

An instance of a subclass of java.applet.Applet

(iv) Security Manager :

A software component that the JVM uses to enforce a security policy.

- The security manager defines the outer boundaries of the sandbox. Because it is customizable, the security manager allows a custom security policy to be established for an application.
- The Java API enforces the custom security policy by asking the security manager for permission before it takes any action that is potentially unsafe.
- A software component used by a security manager to implement a security policy.
- Checks permissions against protection domains provides. It provides a default security policy enforcement mechanism that uses stack inspection to determine whether potentially unsafe actions should be permitted.
- Class java.security.AccessControler.
- A class's protection domain is established by the class loader when the class is loaded (only the class loader knows the class file's origin).
- A protection domain defines all the permissions that are granted to a particular code source, (Protection domain = Code source + collection of permissions)
- The security manager uses a class's protection domain to make decisions about what code in the class is or is not allowed to do.
- Class java. security. Protection Domain.