LAB MANUAL

NETWORK PROGRAMMING LAB

SEMESTER 6: COMPUTER ENGINEERING

EXPERIMENT:1

AIM: TO familiarize with java programming

DESCRIPTION:

The Java API

The Java language is actually rather small and simple - an order of magnitude smaller and simpler than C++, and in some ways, even smaller and simpler than C. However, it comes with a very large and constantly growing library of utility classes. Fortunately, you only need to know about the parts of this library that you really need, you can learn about it a little at a time, and there is excellent, browsable, on-line documentation. These libraries are grouped into *packages*. One set of packages, called the *Java 2 Platform API* comes bundled with the language (API stands for "Application Programming Interface"). At last count, there were over 166 packages in the Platform API, but you will probably only use classes from three of them:

- Java.lang contains things like character strings, that are essentially "built in" to the langauge.
- java.io contains support for input and output, and
- java.util contains some handy data structures such as lists and hash tables.

Values, Objects, and Pointers

It is sometimes said that Java doesn't have pointers. That is not true. In fact, objects can *only* be referenced with pointers. More precisely, variables can hold primitive values (such as integers or floating-point numbers) or*references* (pointers) to objects. A variable cannot hold an object, and you cannot make a pointer to a primitive value.

There are exactly eight primitive types in Java, boolean, char, byte, short, int, long, float, and double.

A boolean value is either true or false. You cannot use an integer where a boolean is required (e.g. in an if or while statement) nor is there any automatic conversion between boolean and integer.

A char value is 16 bits rather than 8 bits, as it is in C or C++, to allow for all sorts of international alphabets. As a practical matter, however, you are unlikely to notice the difference.

There are four integer types, each of which represents a signed integer with a specific number of bits.

Type	Bits	Min Value	Max Value
byte	8	-128	127
short	16	-32,768	32,767
int	32	-2,147,483,648	2,147,483,647
long	64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807

The types float and double represent 32-bit and 64-bit floating point values.

Objects are *instances* of *classes*. They are created by the new operator. Each object is an instance of a unique class, which is itself an object. Class objects are automatically created whenever you refer the class; there is no need to use new. Each object "knows" what class it is an instance of.

```
Pair p = new Pair();
Class c = p.getClass();
System.out.println(c);  // prints "class Pair"
```

Each object has a set of *fields* and *methods*, collectively called *members*. (Fields and methods correspond to data members and function members in C++). Like variables, each field can hold either a primitive value (aboolean, int, etc.) or a reference, which is either null or points to another object. When a new object is created, its fields are initialized to zero, null or false as appropriate, but a *constructor* (a method with the same name as the class) can supply different initial values (see below). By contrast, variables are not automatically initialized. It is a compile-time error to use a variable that has not been initialized. The compiler may complain if it's not "obvious" that a variable is initialized before use. You can always make it "obvious" by initializing the variable when it is declared.

Portability: Java programs are portable across operating systems and hardware environments.

Portability is to your advantage because:

• You need only one version of your software to serve a broad market.

• The Internet, in effect, becomes one giant, dynamic library.

• You are no longer limited by your particular computer platform.

Three features make Java String programs portable:

1. The language. The Java language is completely specified; all data-type sizes and formats are defined as part of the language. By contrast, C/C++ leaves these "details" up to the compiler implementor, and many C/C++ programs therefore are not portable.

2. The library. The Java class library is available on any machine with a Java runtime system, because a portable program is of no use if you cannot use the same class library on every platform. Window-manager function calls in a Mac application written in C/C++, for example, do not port well to a PC.

3. The byte code. The Java runtime system does not compile your source code directly into machine language, an inflexible and nonportable representation of your program. Instead, Java programs are translated into machine-independent byte code. The byte code is easily interpreted and therefore can be executed on any platform having a Java runtime system. (The latest versions of the Netscape Navigator browser, for example, can run applets on virtually any platform).

Security

The Java language is secure in that it is very difficult to write incorrect code or viruses that can corrupt/steal your data, or harm hardware such as hard disks.

There are two main lines of defense:

- Interpreter level:
- No pointer arithmetic
- Garbage collection
- Array bounds checking
- No illegal data conversions

• Browser level (applies to applets only):

• No local file I/O

• Sockets back to host only

• No calls to native methods

Robustness:

The Java language is robust. It has several features designed to avoid crashes during program execution, including:

• No pointer arithmeticJava Basics

• Garbage collection--no bad addresses

• Array and string bounds checking

• No jumping to bad method addresses

• Interfaces and exceptions

Java Program Structure:

A file containing Java source code is considered a compilation unit. Such a compilation unit contains a set of classes and, optionally, a package definition to group related classes together. Classes contain data and method members that specify the state and behavior of the objects in your program.

Java programs come in two flavors:

• Standalone applications that have no initial context such as a pre-existing main window

• Applets for WWW programming

The major differences between applications and applets are:

• Applets are not allowed to use file I/O and sockets (other than to the host

platform). Applications do not have these restrictions.

• An applet must be a subclass of the Java Applet class. Aplications do not need to subclass any particular class.

• Unlike applets, applications can have menus.

• Unlike applications, applets need to respond to predefined lifecycle messages from the WWW browser in which they're running.

RESULT: familiarized with java language

EXPERIMENT:2

AIM: familiarize with swing in java

DESCRIPTION:

Swing is the primary Java GUI widget toolkit. It is part of Oracle's Java Foundation Classes (JFC) — an API for providing a graphical user interface (GUI) for Java programs.

Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT). Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

Unlike AWT components, Swing components are not implemented by platform-specific code. Instead they are written entirely in Java and therefore are platform-independent. The term "lightweight" is used to describe such an element Swing is a platform-independent, *Model-View-Controller* GUI framework for Java, which follows a single-threaded programming model.[4] Additionally, this framework provides a layer of abstraction between the code structure and graphic presentation of a Swing-based GUI. Swing is platform-independent because it is completely written in Java. Complete documentation for all Swing classes can be found in the Java API Guide . Swing is a highly modular-based architecture, which allows for the "plugging" of various custom implementations of specified framework interfaces: Users can provide their own custom implementation(s) of these components to override the default implementations using Java's inheritance mechanism.[5]

Swing is a component-based framework, whose components are all ultimately derived from the javax.swing.JComponent class. Swing objects asynchronously fire events, have bound properties, and respond to a documented set of methods specific to the component. Swing components are Java Beans components, compliant with the Java Beans Component Architecture specifications

Given the programmatic rendering model of the Swing framework, fine control over the details of rendering of a component is possible. As a general pattern, the visual representation of a Swing component is a composition of a standard set of elements, such as a border, inset, decorations, and other properties. Typically, users will programmatically customize a standard Swing component (such as a JTable) by assigning specific borders, colors, backgrounds, opacities, etc. The core component will then use these properties to render itself. However, it is also completely possible to create unique GUI controls with highly customized visual representation

RESULT: familiarized with swing in java

EXPERIMENT:3

AIM: to get familiarized with network programming with java

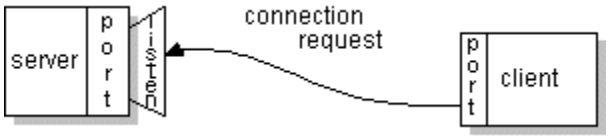
DESCRIPTION:

In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel that client-server applications on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

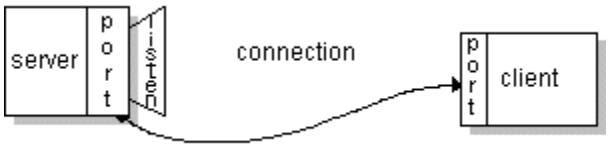
Socket: A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively. Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.



If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to

listen to the original socket for connection requests while tending to the needs of the connected client.



On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server.

The client and server can now communicate by writing to or reading from their sockets.

RESULT: familiarized with network programming.

EXPERIMENT: 4

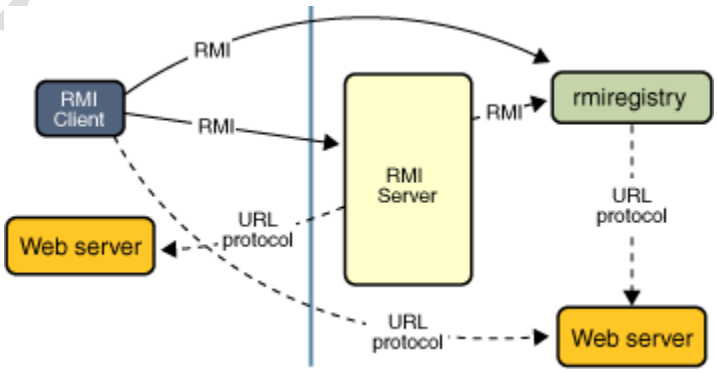
AIM: familiarize with RMI in java

DESCRIPTION: RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

Distributed object applications need to do the following:

- **Locate remote objects.** Applications can use various mechanisms to obtain references to remote objects. For example, an application can register its remote objects with RMI's simple naming facility, the RMI registry. Alternatively, an application can pass and return remote object references as part of other remote invocations.
- **Communicate with remote objects.** Details of communication between remote objects are handled by RMI. To the programmer, remote communication looks similar to regular Java method invocations.
- **Load class definitions for objects that are passed around.** Because RMI enables objects to be passed back and forth, it provides mechanisms for loading an object's class definitions as well as for transmitting an object's data.

The following illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object. The server calls the registry to associate (or bind) a name with a remote object. The client looks up the remote object by its name in the server's registry and then invokes a method on it. The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed.



Advantages of Dynamic Code Loading

One of the central and unique features of RMI is its ability to download the definition of an object's class if the class is not defined in the receiver's Java virtual machine. All of the types and behavior of an object, previously available only in a single Java virtual machine, can be transmitted to another, possibly remote, Java virtual machine. RMI passes objects by their actual classes, so the behavior of the objects is not changed when they are sent to another Java virtual machine. This capability enables new types and behaviors to be introduced into a remote Java virtual machine, thus dynamically extending the behavior of an application. The compute engine example in this trail uses this capability to introduce new behavior to a distributed program.

Remote Interfaces, Objects, and Methods

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called *remote objects*.

An object becomes remote by implementing a *remote interface*, which has the following characteristics:

- A remote interface extends the interface java.rmi.Remote.
- Each method of the interface declares java.rmi.RemoteException in its throws clause, in addition to any application-specific exceptions.

RMI treats a remote object differently from a non-remote object when the object is passed from one Java virtual machine to another Java virtual machine. Rather than making a copy of the implementation object in the receiving Java virtual machine, RMI passes a remote *stub* for a remote object. The stub acts as the local representative, or proxy, for the remote object and basically is, to the client, the remote reference. The client invokes a method on the local stub, which is responsible for carrying out the method invocation on the remote object.

A stub for a remote object implements the same set of remote interfaces that the remote object implements. This property enables a stub to be cast to any of the interfaces that the remote object implements. However, *only* those methods defined in a remote interface are available to be called from the receiving Java virtual machine.

Creating Distributed Applications by Using RMI

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.
2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

Designing and Implementing the Application Components

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

- **Defining the remote interfaces.** A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- **Implementing the remote objects.** Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.
- **Implementing the clients.** Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Compiling Sources

As with any Java program, you use the javac compiler to compile the source files. The source files contain the declarations of the remote interfaces, their implementations, any other server classes, and the client classes.

Making Classes Network Accessible

In this step, you make certain class definitions network accessible, such as the definitions for the remote interfaces and their associated types, and the definitions for classes that need to be downloaded to the clients or servers. Classes definitions are typically made network accessible through a web server.

Starting the Application

Starting the application includes running the RMI remote object registry, the server, and the client.The rest of this section walks through the steps used to create a compute engine

RESULT: familiarized with the concepts of RMI

PROGRAMS

1.