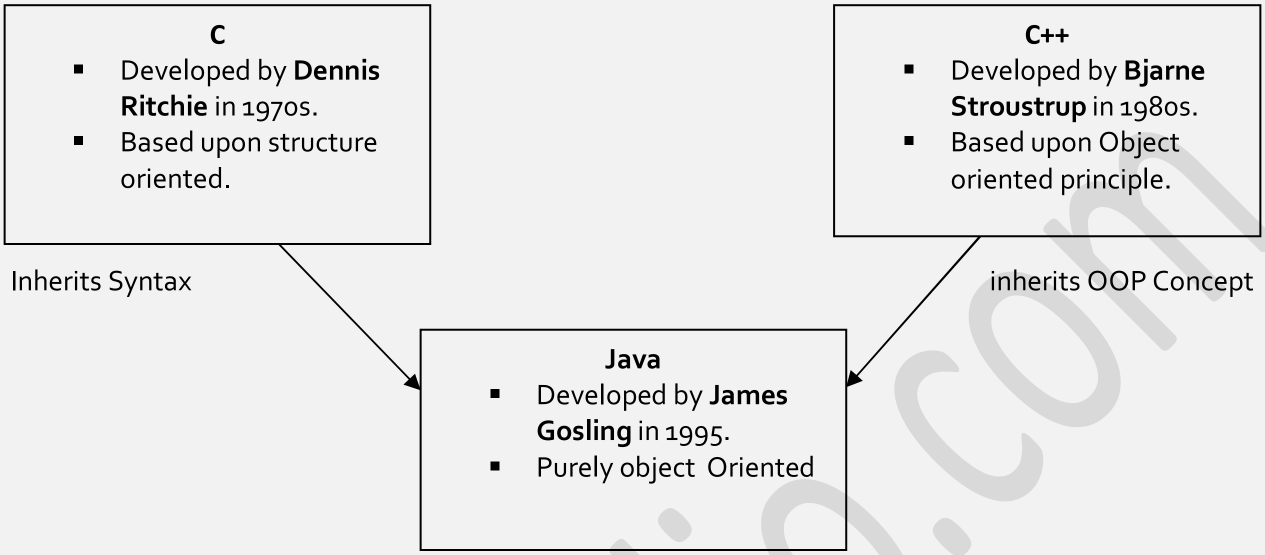
Core Java Concepts

CONTENTS

Chapter1

Introduction



C

- Developed by **Dennis Ritchie** in 1970s.
- Based upon structure oriented.

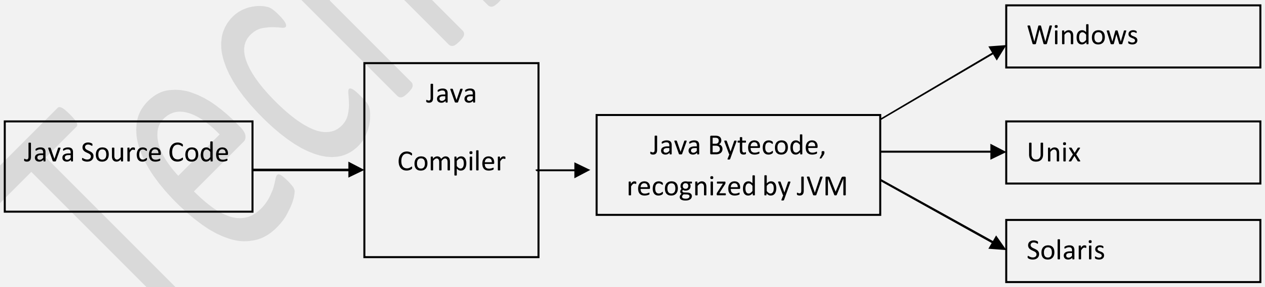
C++

- Developed by **Bjarne Stroustrup** in 1980s.
- Based upon Object oriented principle.

Java

- Developed by **James Gosling** in 1995.
- Purely object Oriented

- Java was developed in the year 1995 at Sun Microsystems.
- Java was developed by James Gosling.
- Initially it was called Oak, in honor of the tree outside James Gosling's window; its name was changed to Java because there was already a language called Oak.
- Java is based upon the concept "**Write once, run anywhere**". The idea is that the same software should run on different kinds of computers, consumer gadgets, and the other devices.
- Unlike most programming languages that generates executable code upon compilation; the JVM generates byte codes as a result of compilation. Java byte codes are form of instructions understood by Java Virtual Machine and usually generated as a result of compiling Java languages source code.



Java Programs are basically of two types:

(i) Applications - Applications are the java programs that are designed to run on the local system and do not need any web browser to execute.

(ii) Applets – Applets are the java programs that are created specially to work over internet and are executed in a java enabled Web browsers (i.e. Internet Explorer, Mozilla Firefox, Google Chrome and Netscape etc).

Java's Magic: the Byte Code

- Bytecode is a highly optimized set of instructions designed to be executed by the Java run-time system, which is called the Java Virtual Machine (JVM).
- Translating a Java program into bytecode helps makes it much easier to run a program in a wide variety of environments i.e. Windows, Unix, Solaris etc. The reason is straightforward:only the JVM needs to be implemented for each platform.

Java Virtual Machine

- The JVM or Java Virtual Machine has an interpreter component that enables communication between Java Byte Code and a computer's operating system. Using a JVM we can run a java code on any platform such as windox xp, windox 98, unix etc.
- JVM normally reads and execute java statements once at a time.
- However a JVM can include JIT (Just in Time) compiler within it. A JIT compiler converts all programs to byte code and is thus faster than a conventional JVM.

What makes Java, most powerful language........... Java Buzz Words

(i) Simple

Java was designed to be easy for the professional programmer to learn and use effectively. Because Java inherits the C/C++ syntax and many of the object-oriented features of C++, most programmers have little trouble learning Java. Also, some of the more confusing concepts from C++ are either left outof Java or implemented in a cleaner, more approachable manner. In Java, there are a small number of clearly defined ways to accomplish a given task.

(ii) Security

When we use a Java-compatible Web browser, we can safely download Java applets without fear of viral infection or malicious intent. Java achieves this protection by confining a Java program to the Java execution environment (sand box area) and not allowing it access to other parts of the computer. This control is exercised by JVM.

(iii) Portability

Many types of computers and operating systems are in use throughout the world and many are connected to the Internet. For programs to be dynamically downloaded to various types of platforms connected to the Internet, some portable executable code is needed that can run on any environment i.e. the Byte Code.

(iv) Object-Oriented

Like C++ , java uses all the OOP concepts like encapsulation, inheritance and polymorphism etc. Java is called a purely object oriented language as everything we write inside class in a java program.

(v) Robust

The multiplatform environment of the Web places extraordinary demands on a program, because the program must execute reliably in a variety of systems. Thus, the ability to create robust programs was given a high priority in the design of Java. To gain reliability, Java restricts us in a few key areas, to force you to find our mistakes early in program development.

(vi) Multithreaded

Java was designed to meet the real-world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously. The Java run-time system comes with an elegant yet sophisticated solution for multiprocess synchronization that enables us to construct smoothly running interactive systems.

(vii) Architecture-Neutral

A central issue for the Java designers was that of code longevity and portability. One of the main problems facing programmers is that no guarantee exists that if you write a program today, it will run tomorrow, even on the same machine. Operating system upgrades, processor upgrades, and changes in core system resources can all combine to make a program malfunction. The Java designers made several hard decisions in the Java language and the Java Virtual Machine in an attempt to alter this situation. Their goal was “write once; run anywhere, anytime, forever.” To a great extent, this goal was accomplished.

(viii) Distributed

Java is designed for the distributed environment of the Internet, because it handles TCP/IP protocols. In fact, accessing a resource using a URL is not much different from accessing a file. The original version of Java (Oak) included features for intra address- space messaging. This allowed objects on two different computers to execute procedures remotely.

(ix) Dynamic

Java programs carry with them substantial amounts of run-time type information that is used to verify and resolve accesses to objects at run time. This makes it possible to dynamically link code in a safe and expedient manner. This is crucial to the robustness of the applet environment, in which small fragments of byte code may be dynamically updated on a running system.

First Java Program

Program 1.1 **WAP to print a message “Welcome to Java World”.**

Solution:

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

        System.out.println ("Welcome to Java World");
        }

}
```

Output:

```
Welcome to Java World
```

How to compile the above program:

To compile the Example program, execute the compiler, javac, specifying the name of the source file on the command line, as shown here:

```
C:\>javac Example.java
```

The javac compiler creates a file called Example.class that contains the byte code version of the program. As discussed earlier, the Java byte code is the intermediate representation of our program that contains instructions the Java interpreter will execute. Thus, the output of javac is not code that can be directly executed. To actually run the program, we must use the Java interpreter, called java. To do so, pass the class name Example as a command-line argument, as shown here:

```
C:\>java Example
```

When the program is run, the following output is displayed:

```
Welcome to Java  World
```

Explanation of above program

public static void main(String args[])

All Java applications begin execution from **main ( )** function. (This is just like C/C++.)

- The **public** keyword is an access specifier, which allows the programmer to control the visibility of class members. When a class member is preceded by public, then that member may be accessed by code outside the class in which it is declared.
- The keyword **static** allows main ( ) to be called without having to instantiate a particular instance of the class.
- The keyword **void** simply tells the compiler that main ( ) does not return a value.

- As stated, main ( ) is the method called when a Java application begins. Keep in mind that Java is case-sensitive. Thus, Main is different from main.
- **String args[ ]** declares a parameter named args, which is an array of instances of the class String. (Arrays are collections of similar objects.) Objects of type String store character strings. In this case, args receives any command-line arguments present when the program is executed.

System.out.println ("This is a simple Java program.");
This line outputs the string "This is a simple Java program." followed by a new line on the screen. Output is actually accomplished by the built-in **println( )** method. In this case, **println( )** displays the string which is passed to it.

Program 1.2 **WAP to initialize an integer variable num to 100, then display the value of num and num × 2.**

Solution:

```
class Test
 {
        public static void main(String args[])
        {
                int num;
                num = 100;
                System.out.println("This is num: " + num);
                num = num * 2;
                System.out.print("The value of num * 2 is ");
                System.out.println(num);
        }
}
```

Output:

```
This is num: 100

The value of num * 2 is 200
```

Note: *Here this program must saved with the file name Test.java*

Data Types

Java defines eight simple types of data: byte, short, int, long, float, double and Boolean. These can be put in four groups:

■ **Integers:** This group includes **byte**, **short**, **int**, and **long**, which are for whole valued signed numbers.

■ **Floating-point numbers:** This group includes **float** and **double**, which represent numbers with fractional precision.

■ **Characters:** This group includes **char**, which represents symbols in a character set, like letters and numbers.

■ **Boolean:** This group includes **boolean**, which is a special type for representing true/false values.

Sl No	Name	Size	Range	Type
1	**byte**	1 byte	-2^{8} to 2^{8}-1	Integer
2	**short**	2 byte	-2^{16} to 2^{16}-1	
3	**int**	4 byte	-2^{32} to 2^{32}-1	
4	**long**	8 byte	-2^{64} to 2^{64}-1	
5	**float**	4 byte	-2^{32} to 2^{32}-1	Floating Point
6	**double**	8 byte	-2^{64} to 2^{64}-1	
7	**char**	2 byte	-2^{16} to 2^{16}-1	Character
8	**boolean**	1 byte (Holds either T or F)	-2^{8} to 2^{8}-1	Boolean

➡ **Why Char in Java is not same as char in C/C++?**
Java uses Unicode to represent characters. Unicode defines fully international character set that can represent all of the characters found in human languages.

Java programs are a collection of identifiers, comments, literals, operators, and keywords etc. Let us discuss one by one:

1. Identifiers

Identifiers are used for class names, method names, and variable names. An identifier may be any descriptive sequence of uppercase and lowercase letters, numbers, or the underscore and dollar-sign characters. They must not begin with a number, lest they be confused with a numeric literal. Again, Java is case-sensitive, so **VALUE** is a different identifier than **Value**.
Some examples of valid identifiers/ variables are:
AvgTemp, count , a4, test , this_is_ok
Invalid variable names include:
2count, high-temp, Not/ok

2. Literals

A constant value in Java is created by using a literal representation of it. For example, here are some literals:
100, 98.6, ‘X’, “This is a test”
Left to right, the first literal specifies an integer, the next is a floating-point value, the third is a character constant, and the last is a string. A literal can be used anywhere a value of its type is allowed.

3. Comments

Three types of comments or comment lines are defined by Java i.e. single-line, multiline and documentation comment line. Comment lines are never executed during program execution. They included in a program just to convey some message to the programmer or user.

// single line comment: Single line comment is used, when comments consists of a line.

/* multi line
Comment*/: Multiline comment is used, when comments consists of more than one line.

/ documentation Comment */**: Documentation comment is used to include documentation part of a program.

4. Java Keywords

There are 49 reserved keywords currently defined in the Java language. These keywords, combined with the syntax of the operators and separators, form the definition of the Java language. These keywords cannot be used as names for a variable, class, or method.

abstract
assert
boolean
break
byte
case
catch
char
class
const
continue
default
do
double
else
extends
final
finally
float
for
goto
if
implements
import
instanceof
int
interface
long
native
new
package
private
protected
public
return
short
static
strictfp
super
switch
synchronized
this
throw
throws
transient
try
void
volatile
while

Operators

1. Arithmetic Operators

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators:

Operator	Result
+	Addition
–	Subtraction (also unary minus)
*	Multiplication
/	Division
%	Modulus
++	Increment
+=	Addition assignment
–=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
%=	Modulus assignment
––	Decrement

The operands of the arithmetic operators must be of a numeric type.

2. The Bitwise Operators

Java defines several bitwise operators which can be applied to the integer types, **long**, **int**, **short**, **char**, and **byte**. These operators act upon the individual bits of their operands. They are summarized in the following table:

Operator	Result
~	Bitwise unary NOT
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
>>	Shift right
>>>	Shift right zero fill
<<	Shift left
&=	Bitwise AND assignment
|=	Bitwise OR assignment
^=	Bitwise exclusive OR assignment
>>=	Shift right assignment
>>>=	Shift right zero fill assignment
<<=	Shift left assignment

3. Relational Operators

The relational operators determine the relationship that one operand has to the other. Specifically,they determine equality and ordering. The relational operators are shown here:

Operator	**Result**
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to

The outcome of these operations is a **boolean** value. The relational operators are most frequently used in the expressions that control the **if** statement and the various loop statements.

4. The Assignment Operator

The assignment operator is the single equal sign, =. The assignment operator works in Java much as it does in any other computer language. It has this general form:

```
var = expression;
```

Here, the type of var must be compatible with the type of expression. The assignment operator does have one interesting attribute that you may not be familiar with: it allows you to create a chain of assignments. For example, consider this fragment:

```
int x, y, z;
x = y = z = 100; // set x, y, and z to 100
```

This fragment sets the variables x, y, and z to 100 using a single statement. This works because the = is an operator that yields the value of the right-hand expression. Thus, the value of z = 100 is 100, which is then assigned to y, which in turn is assigned to x. Using a “chain of assignment” is an easy way to set a group of variables to a common value.

5. Conditional Operator

Java includes a special ternary (three-way) operator that can replace certain types of if-then-else statements. The conditional operator has this general form:

expression1 ? expression2 : expression3

Here, expression1 can be any expression that evaluates to a boolean value. If expression1 is true, then expression2 is evaluated; otherwise, expression3 is evaluated. The result of the ? operation is that of the expression evaluated.

Program 1.3 **Demonstration of conditional operator.**

Solution:

```
class Ternary
```

```
{
        public static void main(String args[])
        {
                int i, k;
                i = 10;
                k = i < 0 ? -i : i;
                System.out.print("Absolute value of ");
                System.out.println(i + " is " + k);
                i = -10;
                k = i < 0 ? -i : i;
                System.out.print("Absolute value of ");
                System.out.println(i + " is " + k);
        }
}
```

Output:

```
Absolute value of 10 is 10
Absolute value of -10 is 10
```

Operators Precedence Table

Highest precedence

( )	[ ]	.	
++	−−	~	!
*	/	%	
+ −			
>>	>>>	<<	
>	>=	<	<=
== !=			
&			
^			
|			
&&			
||			
?:			
=	op=		

Lowest Precedence

Assignment 1

Short Type Questions

1. Define Byte Code in Java.

2. Why Java is called Platform independent language.

3. Why java programs are secure over internet.

4. Discuss the role of JVM.

5. What is conditional operator? Discuss its syntax.

6. Why java is called write once, run anywhere language?

7. Why the function main () is always declared as static?

Long Type Questions

1. Discuss the features of java which makes it most powerful language.

2. Write a Java program to display your college name.

Java’s Control Statements

A programming language uses control statements to cause the flow of execution to advance andbranch based on changes to the state of a program. Java’s program control statements can be put into the following categories: **selection, iteration, and jump.**

- Selection statements allows program to choose different paths of execution based upon the outcome of an expression or the state of a variable.
- Iteration statements enable program execution to repeat one or more statements (that is, iteration statements form loops).
- Jump statements allows program to execute in a nonlinear fashion. All of Java’s control statements are examined here.

Selection Statements

1. if Statement

The **if** statement is Java’s conditional branch statement. It can be used to route program execution through two different paths. Here is the general form of the **if** statement:

```
if (condition)
{
        statement1;
}
else
{
        statement2;
}
```

Here, each statement may be a single statement or a compound statement enclosed in curly braces (that is, a block). The condition is any expression that returns a **boolean** value. The **else** clause is optional. The **if statement works** like this:

If the condition is true, then statement1 is executed. Otherwise, statement2 (if it exists) is executed.

Program 2.1 **WAP to check whether a person is old or young. A person will be said old if his age is above 35 and young if his age is below 35.**

Solution:

```
class Man
{
        public static void main(String args[])
        {
                int age= 49;

                if(age>=35)
                System.out.println("Old");

                else
                System.out.println("Young");
        }
}
```

Output: Old

2. Nested ifs

A nested if is an if statement that is the target of another if or else. Nested ifs are very common in programming. When you nest ifs, the main thing to remember is that an else statement always refers to the nearest if statement that is within the same block as the else and that is not already associated with an else.

3. The if-else-if Ladder

A common programming construct that is based upon a sequence of nested ifs is the if-else-if ladder. It looks like this:

```
        if (condition1)
        statement1;
        else if (condition2)
        statement2;
        ............................
        ............................
        else
        statement3;
```

The if statements are executed from the top to down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the ladder is bypassed. If none of the conditions is true, then the final else statement will be executed.

Program 2.2 **WAP to calculate division obtained by a student with his percentage mark given.**

Solution:

```
class Result
{
        public static void main(String args[])
        {
                int mark= 76;

                if(mark>=60)
                System.out.println("1st Division");

                else if(mark>=50 && mark<60)
                System.out.println("2nd Division");

                else
                System.out.println("3rd  Division");

        }
}
```

Output: 1st Division

Switch Statements

The **switch** statement is Java’s multi way branch statement. It provides an easy way to dispatch execution to different parts of your code based on the value of an expression. As such, it often provides a better alternative than a large series of **if-else-if** statements. Here is the general form of a **switch** statement:

```
        switch (expression)
         {
                case value1:
                // statement sequence
                break;
                case value2:
                // statement sequence
                break;
                ...
                case valueN:
```

```
        // statement sequence
        break;
        default:
        // default statement sequence
    }
```

Program 2.3 **WAP to illustrate the use of switch case statements.**

Solution:

```
class SampleSwitch
{
    public static void main(String args[])
    {
        for(int i=0; i<6; i++)
        switch(i)
        {
            case 0:
            System.out.println("i is zero.");
            break;
            case 1:
            System.out.println("i is one.");
            break;
            case 2:
            System.out.println("i is two.");
            break;
            case 3:
            System.out.println("i is three.");
            break;
            default:
            System.out.println("i is greater than 3.");
        }
    }
}
```

Output:

```
i is zero.
i  is  one.
i is two.
i is three.
```

```
i is greater than 3.
i is greater than 3.
```

Iteration Statements

Java's iteration statements are for, while, and do-while. These statements create what we commonly call loops. As you probably know, a loop repeatedly executes the same set of instructions until a termination condition is met.

1. while loop

The while loop is Java's most fundamental looping statement. It repeats a statement or block while its controlling expression is true. Here is its general form:

```
    initialization
    while (condition)
    {
    // body of loop
    Increment/ decrement
    }
```

The condition can be any Boolean expression. The body of the loop will be executed as long as the conditional expression is true. When condition becomes false, control passes to the next line of code immediately following the loop. The curly braces are unnecessary if only a single statement is being repeated.

Program 2.4 **WAP to print the message "Java is good" 10 times using while loop.**

Solution:

```
class Print
{
        public static void main(String args[])
        {
        int n = 0;
                while(n <10)
                {
                System.out.println("Java is good");
                n++;
                }
        }
}
```

2. do while Loop

If the conditional expression controlling a **while** loop is initially false, then the body of the loop will not be executed at all. However, sometimes it is desirable to execute the body of a **while** loop at least once, even if the conditional expression is false to begin with. The **do-while** loop always executes its body at least once, because its conditional expression is at the bottom of the loop. Its general form is:

```
        initialization
        do
        {
        // body of loop
        Increment/ decrement
        } while (condition);
```

Program 2.5 **WAP to print the message "Java is good" 10 times using do while loop.**

Solution:

```
class Print
{
        public static void main(String args[])
        {
                int n = 0;
                do
                {
                System.out.println("Java is good");
                n++;
                } while(n<9);
        }
}
```

3. for loop

The general form of the **for** statement is:

```
for(initialization; condition; iteration)
 {
// body
}
```

The **for** loop operates as follows:

- When the loop first starts, the initialization portion of the loop is executed. Generally, this is an expression that sets the value of the loop control variable, which acts as a counter that controls the loop. It is important to understand that the initialization expression is only executed once.

- Next, condition is evaluated. This must be a Boolean expression. It usually tests the loop control variable against a target value. If this expression is true, then the body of the loop is executed. If it is false, the loop terminates.
- Next, the iteration portion of the loop is executed. This is usually an expression that increments or decrements the loop control variable. The loop then iterates, first evaluating the conditional expression, then executing the body of the loop, and then executing the iteration expression with each pass. This process repeats until the controlling expression is false.

Program 2.6 **WAP to print the message "Java is good" 10 times using for loop.**

Solution:

```
class Print
{
        public static void main(String args[])
        {
                int n ;
                for(n=0; n<10; n++)
                {
                System.out.println("Java is good");
                }
        }
}
```

Jump Statements

Java offers following jump statements:

- **break statement:** A break statement takes control out of the loop.
- **continue statement:** A continue statement takes control to the beginning of the loop.
- **goto statement:** A goto Statement take control to a desired line of a program.

Program 2.7 **WAP to check whether a number is prime or not.**

Solution:

```
class Prime
{
        public static void main(String args[])
        {
                int n=13 , i;
                for(i=2; i<n; i++)
                {
                        if(n%i==0)
                        {
```

```
                        System.out.println(“Number is not Prime”);
                        break;
                        }
                }
                if(i==n)

                System.out.println(“Number is  Prime”);
        }
}
```

Output:

Number is Prime

Class and object

A class is a user defined data type which binds data and function into a single unit and objects are the instance of a class.

A class is declared by use of the **class** keyword. The general form of a **class** definition is shown here:

```
class classname
{
        type  instance-variable1;
        type instance-variable2;
        type instance-variablen;

        type methodname1(parameter-list)
        {
        // body of method
        }
        type methodname2(parameter-list)
        {
        // body of method
        }
        type methodnamen(parameter-list)
        {
        // body of method
        }
}
```

The data, or variables, defined within a **class** are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

Program 2.8 **WAP to calculate volume of a box with its width, height and depth given.**

Solution:

```
class Box
 {
        double width;
        double height;
        double depth;
        void volume()
         {
                System.out.print("Volume is ");
                System.out.println(width * height * depth);
        }
}
class BoxDemo
 {
        public static void main(String args[])
        {
                Box mybox1 = new Box();      //create an object mybox1 and allocates memory
                Box mybox2 = new Box();     //create an object mybox2 and allocates memory

                mybox1.width = 10;
                mybox1.height = 20;
                mybox1.depth = 15;

                mybox2.width = 3;
                mybox2.height = 6;
                mybox2.depth = 9;

                mybox1.volume();
                mybox2.volume();
        }
}
```

Output:

```
Volume is 3000.0
Volume is 162.0
```

The **new** operator dynamically allocates memory for an object. It has this general form:

class-var = new classname( );

Here, class-var is a variable of the class type being created. The classname is the name of the class that is being instantiated. The class name followed by parentheses specifies the constructor for the class. A constructor defines what occurs when an object of a class is created.
Thus the statement:

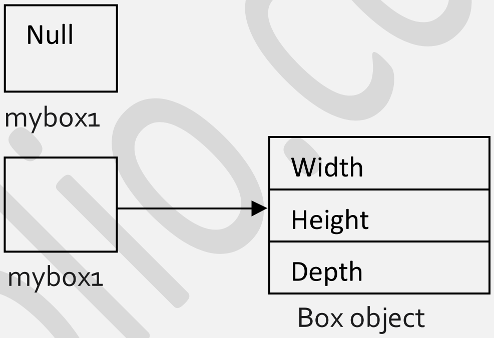
Box mybox1 = new Box(); create an object mybox1 and allocate memory to it.

The above statement can be written in two steps:
Box mybox1; // declare reference to object
mybox1 = new Box(); // allocate memory to a Box object

Statement **Effect**

Box mybox1

mybox1=new Box()



Constructor

- A constructor is a special member function having name same as class name.
- The constructor is automatically called immediately after the object is created, before the **new** operator completes.
- Constructors look a little strange because they have no return type, not even **void**.
- Its purpose is to initialize the object of a class.
- It is of three types:
  - ✓ **Default Constructor:** Constructor which takes no argument(s) is called Default Constructor.
  - ✓ **Parameterized constructor:** Constructor which takes argument(s) is called parameterized Constructor.
  - ✓ **Copy Constructor:** Constructor which takes object as its argument is called copy constructor.
- When a single program contains more than one constructor, then the constructor is said to be overloaded.

Program 2.9 **WAP to illustrate constructor overloading in java.**

Solution:

```
class Box
 {
    double width;
    double height;
    double depth;
    Box(double w, double h, double d)           // Parameterized Constructor
    {
    width = w;
    height = h;
    depth = d;
    }

    Box()                                       // Default Constructor
    {
    width = -1;
    height = -1;
    depth = -1;
    }

    Box(double len)                              // Parameterized Constructor
     {
    width = height = depth = len;
    }

    double volume()
    {
    return width * height * depth;
    }
}

class Overload
{
    public static void main(String args[])
    {
        Box mybox1 = new Box(10, 20, 15);
        Box mybox2 = new Box();
        Box mycube = new Box(7);
        double vol;
        vol = mybox1.volume();
```

```
		System.out.println("Volume of mybox1 is " + vol);
		vol = mybox2.volume();
		System.out.println("Volume of mybox2 is " + vol);
		vol = mycube.volume();
		System.out.println("Volume of mycube is " + vol);
	}
}
```

Output:

```
Volume of mybox1 is 3000.0
Volume of mybox2 is -1.0
Volume of mycube is 343.0
```

The this Keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines this keyword. this can be used inside any method to refer to the current object. **It always points to the object through which a method is called.** We can use this anywhere a reference to an object of the current class' type is permitted.

To better understand what this refers to, consider the following version of Box ( ):

```
// A redundant use of this.

Box(double w, double h, double d)
{
	this.width = w;
	this.height = h;
	this.depth = d;
}
```

This version of Box ( ) operates exactly like the earlier version. The use of this is redundant, but perfectly correct. Inside Box ( ), this will always refer to the invoking object.

The static Keyword

To create such a member, precede its declaration with the keyword static. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object. We can declare both methods and variables to be static.

The most common example of a static member is main ( ). main ( ) is declared as static because it must be called before any objects exist.

Program 2.10 **WAP to demonstrate use of static variables and methods.**

Solution:

```
class UseStatic
{

        static int a = 3;
        static int b;
        static void meth(int x)
        {
        System.out.println("x = " + x);
        System.out.println("a = " + a);
        System.out.println("b = " + b);
        }
        static
        {
        System.out.println("Static block initialized.");
        b = a * 4;
        }
        public static void main(String args[])
        {
        meth(42);
        }
}
```

Here is the output of the program:

```
Static block initialized.
x = 42
a = 3
b = 12
```

As soon as the UseStatic class is loaded, all of the static statements are run. First, a is set to 3, then the static block executes (printing a message), and finally, b is initialized to a * 4 or 12. Then main( )is called, which calls meth( ), passing 42 to x. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Note: *It is illegal to refer to any instance variables inside of a static method.*

The final Keyword

A variable can be declared as **final**. Doing so prevents its contents from being modified. This means that you must initialize a **final** variable when it is declared. (In this usage, **final** is similar to **const** in C/C++/C#.) For example:

```
final int FILE_NEW = 1;
final int FILE_OPEN = 2;
```

Variables declared as **final** do not occupy memory on a per-instance basis. Thus, a **final** variable is essentially a constant.

The keyword **final** can also be applied to methods (will be discussed in inheritance), but its meaning is substantially different than when it is applied to variables.

Garbage Collection

Since objects are dynamically allocated by using the new operator, you might be wondering how such objects are destroyed and their memory released for later reallocation.

- In some languages, such as C++, dynamically allocated objects must be manually released by use of a delete operator. Java takes a different approach; it handles deallocation automatically. The technique that accomplishes this is called garbage collection.
- It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.
- There is no explicit need to destroy objects as in C++. Garbage collection only occurs sporadically (if at all) during the execution of your program. It will not occur simply because one or more objects exist that are no longer used.

The finalize () method

- Sometimes an object will need to perform some action when it is destroyed. For example, if an object is holding some non-Java resource such as a file handle or window character font, then you might want to make sure these resources are freed before an object is destroyed. To handle such situations, Java provides a mechanism called finalization.
- By using finalization, we can define specific actions that will occur when an object is just about to be reclaimed by the garbage collector.
- To add a finalizer to a class, we simply define the **finalize ()** method. The Java run time calls that method whenever it is about to recycle an object of that class. Inside the **finalize ()** method you will specify those actions that must be performed before an object is destroyed. The garbage collector runs periodically, checking for objects that are no longer referenced by any running state or indirectly through other referenced objects. Right before an asset is freed, the Java run time calls the **finalize ()** method on the object.

  The **finalize ()** method has this general form:

  ```
  protected void finalize( )
  {
  // finalization code here
  }
  ```

Assignment 2

Short Type Questions

1. Why java does not provide any destructor?
2. Why java characters takes 2 byte unlike c/c++, which takes 1 byte of memory.
3. Define Constructor.
4. Define class. How memory is allocated dynamically to an object of a class?
5. What is the meaning of the statement: final int var=10;
6. Define a static method.

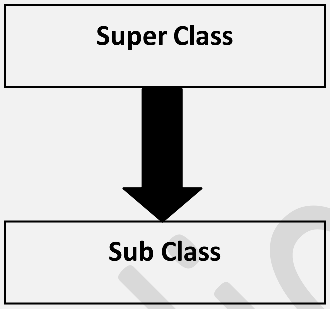
Long Type Questions

1. Write a program that gives the output: "Welcome to the World of Java".
2. Write two explicit constructors for a class to calculate the area of rectangle. When a single value is passed to the constructor, it should assume that the width and height is the same as the passed variable. It should calculate the area accordingly. When two values are passed to the constructor, it should calculate the area.
3. Write a program that does the following:
   - Declare s and initializes variables m and n to 100 and 200 respectively.
   - Under the condition, if m equals 0, it displays the appropriate result.
   - If m is greater than n, it displays the appropriate result.
   - Check whether the value of n is even or odd.
4. Write a program that displays the total no of multiples of 7 between 1 and 100.
5. Write short notes on following:
   - Static
   - Final
   - Garbage Collection
   - Constructor
6. What is constructor overloading? Discuss with an example.
7. Write a java program to display all the prime factors of 9999.
8. Write a java program to find sum of all even factors of 5000.
9. Write a java program to find 1+2+3+...........50.
10. Write java program to display following Pattern:

```
   *
  ***
 *****
*******
```

Inheritance

- It is the process by which object of one class can acquire the properties of object of another class.
- The class, from which properties are inherited, is known as **super class**.
- The class, to which properties are inherited, is known as **sub class**. A sub class inherits all of the variables and methods defined by the super class and add its own, unique elements.
- Inheritance allows the creation of hierarchical classifications.



- To derive a sub class from super class, we use the **extends** keyword.
- The general form of a class declaration that inherits a superclass is :

```
class subclass-name extends superclass-name
 {
 // body of class
}
```

- A sub class can access all the public and protected members of a super class. (By default, the variable or function of a class are public in nature)

Program 3.1 **Demonstration of Simple Inheritance.**

Solution:

```
class A
 {
        int i, j;
        void showij()
        {
        System.out.println("i and j: " + i + " " + j);
        }
}
```

```
class B extends A
 {
        int k;
        void showk()
        {
        System.out.println("k:" + k);
        }
        void sum()
         {
        System.out.println (i+j+k);
        }
}

class SimpleInheritance
{
        public static void main(String args[])
        {
        A sup = new A ();
        B sub = new B ();

        sup.i = 10;
        sup.j = 20;
        System.out.println("Contents of super class ");
        sup.showij();

        sub.i = 7;
        sub.j = 8;
        sub.k = 9;
        System.out.println("Contents of sub class");
        sub.showij();
        sub.showk();

        System.out.println("Sum of i, j and k in sub class");
        sub.sum();
        }
}
```

A sub class cannot access private members of a super class.

```
class A
{
        int i;                          // public by default
        private int j;                   // private to A
        void setij(int x, int y)
        {
        i = x;
        j = y;
        }
}

class B extends A
{
        int total;
        void sum()
        {
        total = i + j;                   // ERROR, j is not accessible as it is private to A
        }
}
class Access
{
        public static void main(String args[])
        {
        B sub= new B();
        sub.setij(10, 12);
        sub.sum();
        System.out.println("Total is " + sub.total);
        }
}
```

Remember
A class member that has been declared as private will remain private to its class. It is not accessible by any code outside its class, including subclasses. (i.e. a subclass cannot access the private or protected member of a super class.)

Program 3.2 **Demonstration of Inheritance.**

Solution:

```
class Box
{
        double width, height, depth;
        Box(Box ob)
```

```
         {
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
        }
        Box(double w, double h, double d)
        {
        width = w;
        height = h;
        depth = d;
        }
        Box()
        {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
        }
        Box(double len)
        {
        width = height = depth = len;
        }
        double volume()
        {
        return (width * height * depth);
        }
}

class BoxWeight extends  Box
 {
        double weight;
        BoxWeight(double w, double h, double d, double m)
        {
        width = w;
        height = h;
        depth = d;
        weight = m;
        }
}

class DemoBoxWeight
```

```
{
        public static void main(String args[])
        {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        double vol;

        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);

        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        }
}
```

How Constructor method of a Super class gets called

- A subclass constructor invokes the constructor of the super class implicitly.
    - When a BoxWeight object, a subclass, is instantiated, the default constructor of its super class, Box class, is invoked implicitly before sub-class's constructor method is invoked.
- A subclass constructor can invoke the constructor of the super explicitly by using the "super" keyword.
    - The constructor of the BoxWeight class can explicitly invoke the constructor of the Box class using "super" keyword.

- In a class hierarchy, constructors are called in order of derivation, from super class to subclass.
- Since **super( )** must be the first statement executed in a subclass' constructor, this order is the same whether or not **super( )** is used.
- If **super( )** is not used, then the default of each super class will be executed.

Program 3.3 **Demonstration of subclass constructor invoking the constructor of the super class implicitly.**

Solution:

```
class A
{
        A()
        {
```

```
        System.out.println("Inside A's constructor.");
        }
}

class B extends A
{
        B()
        {
        System.out.println("Inside B's constructor.");
        }
}

class C extends B
{
        C()
        {
        System.out.println("Inside C's constructor.");
        }
}

class CallingCons
{
        public static void main(String args[])
        {
        C c = new C();
        }
}
```

Output:

```
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

Program 3.4 **Demonstration of Subclass constructor calling super class’s constructor using the keyword super.**

Solution:

```
class Rect
{
        int length, breadth;
        Rect (int  l, int b)
        {
        length=l;
        breadth=b;
        }
}
class Cuboid extends Rect
 {
        double height;
        Cuboid (int  l, int b, int h)
        {
        super(l, b);
        height=h;
        }
        void  volume()
        {
        System.out.println (length*breadth*height);
        }
}
class Vol
 {
        public static void main(String args[])
        {
        Cuboid v=new Cuboid(1, 2, 3);
        v.volume();
        }
}
```

Output: 6

Method Overriding

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its super class, then the method in the subclass is said to override the method in the super class.
- When an overridden method is called from within a subclass, it will always refer to the version of that method defined by the subclass. The version of the method defined by the super class will be hidden.

Program 3.5 **Demonstration of Method Overriding.**

Solution:

```
class A
{
        int i, j;
        A(int a, int b)
        {
        i = a;
        j = b;
        }
        void show()
        {
        System.out.println("i and j: " + i + " " + j);
        }
}
class B extends A
 {
        int k;
        B(int a, int b, int c)
        {
        super(a, b);
        k = c;
        }
        void show()
        {
        System.out.println("k: " + k);
        }
}
class Override
 {
        public static void main(String args[])
        {
        B sub= new B(1, 2, 3);
        sub.show();                                   // this calls show() in B
        }
}
```

When show ( ) is invoked on an object of type B, the version of show ( ) defined within B is used. That is, the version of show ( ) inside B overrides the version declared in A.
If you wish to access the super class version of an overridden function, you can do so by using **super.**

Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism.

Program 3.6 **Demonstration of Dynamic Method Dispatch.**

Solution:

```
class A
{
        void callme()
        {
        System.out.println("Inside A's callme method");
        }
}

class B extends A
{
        void callme()
        {
        System.out.println("Inside B's callme method");
        }
}
class C extends A
{
        void callme()
        {
        System.out.println("Inside C's callme method");
        }
}
class Dispatch
{
        public static void main(String args[])
        {
        A a = new A();
        B b = new B();
        C c = new C();
        A r;                            // obtain a reference of type A
        r = a;                          // r refers to an A object
        r.callme();                     // calls A's version of callme
        r = b;                           // r refers to a B object
        r.callme();                     // calls B's version of callme

        r = c;                          // r refers to a C object
        r.callme();                     // calls C's version of callme
```

```
            }
}
```

Output:

```
Inside A’s callme method
Inside B’s callme method
Inside C’s callme method
```

Abstract class

Abstract classes are the class which contains method without providing a complete implementation (or without having background details). It is not possible create object of an abstract class. Abstract classes can include methods having all details.

Program 3.7 Demonstration of abstract class.

Solution:.

```
abstract class A
{
            abstract void callme();
            // concrete methods are still allowed in abstract classes
            void callmetoo()
            {
            System.out.println("This is a concrete method.");
            }
}
class B extends A
{
            void callme()
            {
            System.out.println("B's implementation of callme.");
            }
}
class AbstractDemo
{
            public static void main(String args[])
            {
            B b = new B();
            b.callme();
            b.callmetoo();
            }
}
```

Although abstract classes cannot be used to instantiate objects, they can be used to create object references, because Java's approach to run-time polymorphism is implemented through the use of super class references. Thus, it must be possible to create a reference to an abstract class so that it can be used to point to a subclass object.

Uses of final Keyword: To Prevent Inheritance

The keyword final has three uses:

1. First, it can be used **to create constant variable** whose value cannot be changed.

2. To Prevent method Overriding

- While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring.
- To disallow a method from being overridden, specify final as a modifier at the start of its declaration. Methods declared as final cannot be overridden.

```
class A
{
        final void meth()
        {
        System.out.println("This is a final method.");
        }
}
class B extends A
{
        void meth()
        {
        System.out.println("Illegal!");
        }
}
```

Because meth ( ) is declared as final, it cannot be overridden in B. If you attempt to do so, a **compile-time error** will result.

3. To Prevent Inheritance

Sometimes we want to prevent a class from being inherited. To do this, precede the class declaration with the keyword final. Declaring a class as final implicitly declares all of its methods as final, too.

```
final class A
{
        // ...
}
// The following class is illegal.
class B extends A
{
        // ERROR! Can't subclass A
        // ...
}
```

As the comments imply, it is illegal for B to inherit A since A is declared as final.

The Object Class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a super class of all other classes. This means that a reference variable of type Object can refer to an object of any other class.

Assignment 3

Short Type Questions

1. Define inheritance. What are sub class and super class?
2. Can a top level class be made private or protected? Give reason for your answer.
3. Write any two use of final keyword.
4. What is dynamic method dispatch?
5. Java does not provide multiple inheritances. Give reason. How multiple inheritances can be achieved in java?
6. Define Object class.
7. Differentiate between early binding and late binding.
8. How can you call the constructor of a super class explicitly? Illustrate with an example.
9. Write any two use of super keyword.
10. Define abstract class.

Long Type Questions

1. Define inheritance. Multiple inheritances in java cannot be achieved as it leads to ambiguity problem. Justify your answer with suitable example.
2. What is method overriding? How it can prevented? Explain with example.
3. Classes declared as final cannot be inherited (T or F)? Justify your answer with suitable example.
4. How calls to overridden methods are resolved during program execution? Explain.

Chapter 4

Package and Interface

- Packages are containers for classes that are used to keep the class name space compartmentalized.
- For example, a package allows us to create a class named **List**, which you can store in your own package without concern that it will collide with some other class named **List** stored elsewhere.
- Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

Defining a Package

- To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
- The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name. (This is why you haven't had to worry about packages before now.)
- While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.
- This is the general form of the **package** statement:

  package pkg;

  Here, pkg is the name of the package. For example, the following statement creates a package called **MyPackage**. package MyPackage;

```
package MyPack;
class Balance
{
        String name;
        double bal;
        Balance(String n, double b)
        {
        name = n;
        bal = b;
        }
        void show()
        {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
        }
        }
        class AccountBalance
        {
```

```
        public static void main(String args[])
        {
        Balance current[] = new Balance[3];

        current[0] = new Balance("Sanjib", 123.23);
        current[1] = new Balance("Chandan", 157.02);
        current[2] = new Balance("Palak", -12.33);

        for(int i=0; i<3; i++)
        current[i].show();
        }
}
```

Call this file **AccountBalance.java**, and put it in a directory called **MyPack**. Next, compile the file. Make sure that the resulting **.class** file is also in the **MyPack** directory. Then try executing the **AccountBalance** class, using the following command line:

```
java MyPack.AccountBalance
```

Remember, you will need to be in the directory above **MyPack** when you execute this command, or to have your **CLASSPATH** environmental variable set appropriately. As explained, **AccountBalance** is now part of the package **MyPack**. This means that it cannot be executed by itself. That is, you cannot use this command line:

```
java AccountBalance
```

AccountBalance must be qualified with its package name.

Access Protection

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Importing Packages

- Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages.
- In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. This is the general form of the **import** statement:
  import pkg1 [.pkg2].(classname|*);

```
package MyPack;
public class Balance
{
	String name;
	double bal;
	public Balance(String n, double b)
	{
	name = n;
	bal = b;
	}
	public void show()
	{
	if(bal<0)
	System.out.print("--> ");
	System.out.println(name + ": $" + bal);
	}
}
```

As we can see, the **Balance** class is now **public**. Also, its constructor and its **show( )** method are **public**, too. This means that they can be accessed by any type of code outside the **MyPack** package. For example, here **TestBalance** imports **MyPack** and is then able to make use of the **Balance** class:

```
import MyPack.*;
class TestBalance
{
	public static void main(String args[])
	{
	Balance test = new Balance("J. J. Jaspers", 99.88);
	test.show(); // you may also call show()
	}
}
```

Interface

- An interface is a collection of abstract methods (i.e. methods without having definition).
- A class that implements an interface inherits abstract methods of the interface.
- An interface is not a class.
- Writing an interface is similar to writing a class, but they differ in concepts.
- A class describes both attributes (i.e. variables) and behaviors (i.e. methods) of an object.
- An interface contains only behaviors (i.e. methods) that a class implements.
- If a class (**provided it is not abstract**) implements an interface, then all the methods of interface need to be defined in it.

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface:

```
public interface NameOfInterface
{
  //Any number of final, static variables
  //Any number of abstract method declarations (i.e. method without definitions)
}
```

Interfaces have the following properties:

- An interface is implicitly (i.e. by default) abstract. We do not need to use the **abstract** keyword when declaring an interface.
- Each method in an interface is also implicitly (i.e. by default) abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly (i.e. by default) public.

Implementing Interfaces

- A class can implement more than one interface at a time.
- A class can extend only one class, but can implement many interfaces.
- An interface itself can extend another interface.
- The general form of a class that includes the **implements** clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]]
{
// class-body
}
```

Example:

```
interface Message
{
void message1();
void message2();
}

class A implements Message
{
        void message1()
        {
        System.out.println("Good Morning");
        }
        void message2()
        {
        System.out.println("Good Evening");
        }

        public static void main(String args[])
        {
        A a=new A();
        a.message1();
        a.message2();
        }
}
```

Similarities between class and interface

- Both class and interface can contain any number of methods.
- Both class and interface are written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The bytecode of both class and interface appears in a **.class** file.

Dissimilarities between class and interface

- We cannot instantiate (i.e. create object) an interface but we can instantiate a class.
- An interface does not contain any constructors but a class may contain any constructors.
- All the methods in an interface are abstract (i.e. without definitions) but in a class method may or may not be abstract.
- An interface cannot contain variables. The only variable that can appear in an interface must be declared both static and final. But a class can contain any variable.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces (i.e. multiple inheritances can be achieved through it). But a class can not extend multiple classes (i.e multiple inheritance can not be achieved).

Variables in Interfaces

- We can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values.
- When we include that interface in a class (that is, when we “implement” the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of **#defined** constants or **const** declarations.
- If an interface contains no methods, then any class that includes such an interface doesn’t actually implement anything.

```
interface SharedConstants
{
int X = 0;
int Y = 1;
}
class A implements SharedConstants
{
        static void show()
        {
        System.out.println(“X=”+x+”and Y=”+y);
        }

        public static void main(String args[])
        {
        A a = new A();
        show();
        }
}
```

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 defined within the interface inheritance chain.

```
interface A
{
void meth1();
void meth2();
}

interface B extends A
{
void meth3();
}

class MyClass implements B
{
        public void meth1()
        {
        System.out.println("Implement meth1().");
```

```
    }
    public void meth2()
    {
    System.out.println("Implement meth2().");
    }
    public void meth3()
    {
    System.out.println("Implement meth3().");
    }
}
class IFExtend
{
    public static void main(String arg[])
    {
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
    }
}
```

Assignment 4

Short Type Questions

1. Define package.
2. Define Interface.
3. Is the following interface valid?

```
public interface Marker
{
}
```

4. Which package is always imported by default?

Long Type Questions

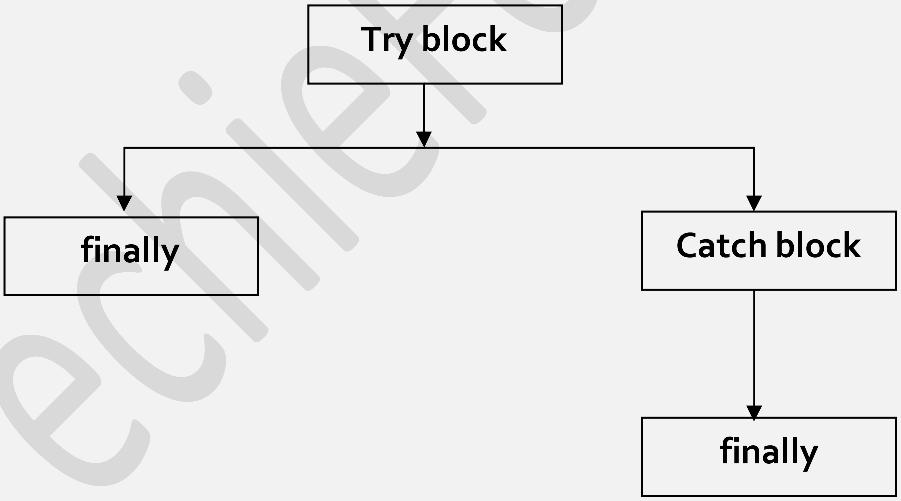
1. Discuss the difference between a class and an interface.
2. Differentiate between interface and abstract class.
3. Java supports multiple inheritances through interface. Discuss.
4. Explain the usage of Java packages.

Exception Handling

- Error occurred in a program can be of two types: syntax error or logical error.
- An **exception** is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a **run-time error**.
- A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- Exception Handling is a mechanism by which exception occurred in a program is handled.
- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.

Keyword/Block	Meanings
try block	The statements which are expected to throw exception are kept inside try block.
catch block	If an exception occurs within the try block, it is throws an exception to the catch block which handle it in some rational manner.
throw	To manually throw an exception, use the keyword throw.
throws	A throws clause lists the types of exceptions that a method might throw.
finally	Any code that absolutely must be executed before a method returns is put in a finally block.

Execution Flow of the try, catch and finally block



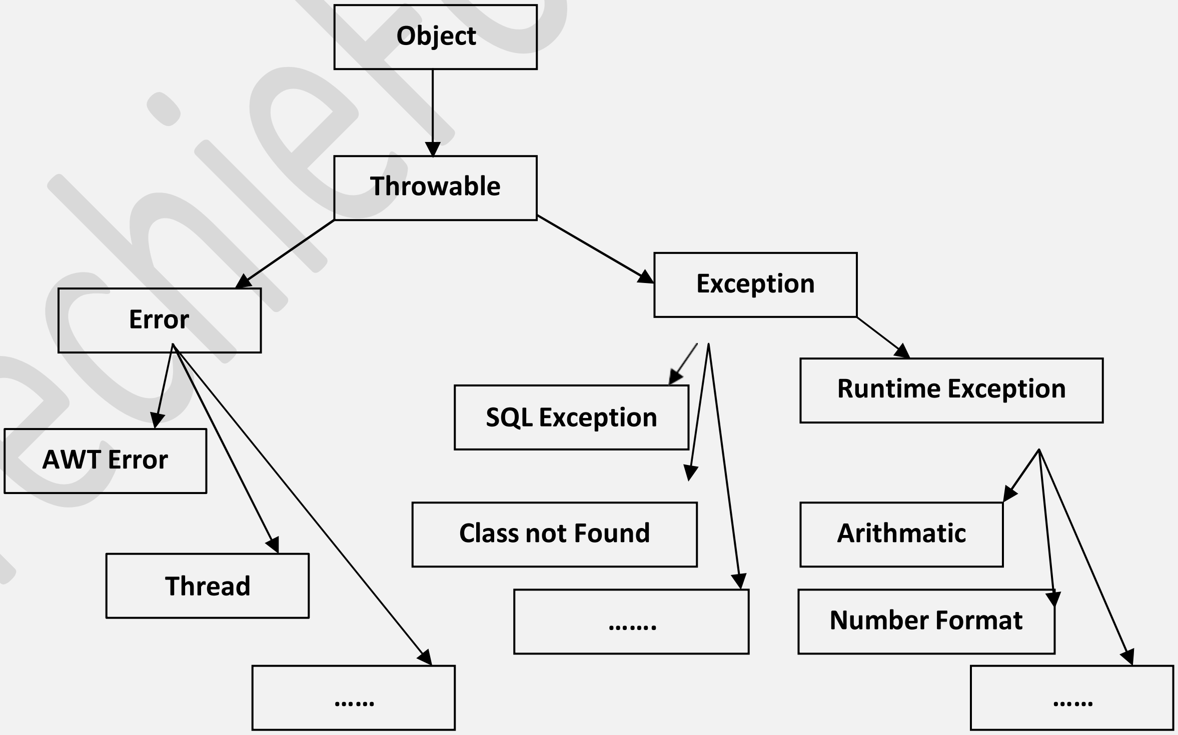
The general form of an exception-handling block is:

```
try
{
        // block of code to monitor for errors
}
```

```
catch (ExceptionType1 exOb)
{
        // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
        // exception handler for ExceptionType2
}
...........
...........
finally
{
        // block of code to be executed before try block ends
}
```

Exception Types

- All exception types are subclasses of the built-in class **Throwable**. Thus, **Throwable** is at the top of the exception class hierarchy.
- The **Throwable** class has two subclasses **Error** and **Exception**.
- **Error** and **Exception** classes are used for handling errors in java.



Unchecked exceptions

Before we learn how to handle exceptions in your program, it is useful to see what happens when we don’t handle them. This small program includes an expression that intentionally causes a divide-by-zero error.

```
class E
{
        public static void main(String args[])
        {
        int x = 0;
        int y = 1 / x;
        }
}
```

Here is the output generated when this example is executed.

```
java.lang.ArithmeticException: / by zero
at E.main(E.java:4)
```

Using try and catch

The following program includes a **try** block and a **catch** clause which processes the **ArithmeticException** generated by the division-by-zero error.

Program 5.1 **Demonstration of Division by zero Exception.**

Solution:

```
class Ex
{
        public static void main(String args[])
        {
        int x, y;
                try
                {
                x = 0;
                y= 1/ x;
                System.out.println("This will not be printed.");
                }
                catch (ArithmeticException e)
                {
                System.out.println("Division by zero.");
                }
        System.out.println("After catch statement.");
        }
}
```

This program generates the following output:

```
Division by zero.
After catch statement.
```

Notice that the call to **println ( )** inside the **try** block is never executed. Once an exception is thrown, program control transfers out of the **try** block into the **catch** block. Put differently, **catch** is not "called," so execution never "returns" to the **try** block from a **catch**. Thus, the line "This will not be printed." is not displayed. Once the **catch** statement has executed, program control continues with the next line in the program following the entire **try/catch** mechanism.

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more **catch** clauses, each catching a different type of exception. When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed. After one **catch** statement executes, the others are bypassed, and execution continues after the **try/catch** block. The following example traps two different exception types:

Program 5.2 **Demonstration of multiple catch block.**

Solution:

```
class MultiCatch
{
        public static void main(String args[])
        {
                try
                {
                int a = args.length;
                System.out.println("a = " + a);
                int b = 1 / a;
                int c[] = { 1 };
                c[42] = 99;
                }
                catch(ArithmeticException e)
                {
                System.out.println("Divide by 0: " + e);
                }
                catch(ArrayIndexOutOfBoundsException e)
                {
                System.out.println("Array index oob: " + e);
                }
        System.out.println("After try/catch blocks.");
        }
}
```

This program will cause a division-by-zero exception if it is started with no commandline parameters, since **a** will equal zero. It will survive the division if you provide a command-line argument, setting **a** to something larger than zero. But it will cause an **ArrayIndexOutOfBoundsException**, since the **int** array **c** has a length of 1, yet the
program attempts to assign a value to **c[42]**.
Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.
C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException
After try/catch blocks.
```

throw

So far, you have only been catching exceptions that are thrown by the Java run-time system. However, it is possible for our program to throw an exception explicitly, using the **throw** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, ThrowableInstance must be an object of type **Throwable** or a subclass of **Throwable**.

Program 5.3 **Demonstration of throw statement.**

Solution:

```
class ThrowDemo
{
        public static void main(String args[])
        {
                int x=2, y=0;
                try
                {
                if(y==0)
                throw new ArithmeticException();
                }
```

```
                    catch(ArithmeticException e)
                    {
                    System.out.println("Exception Occurred: Division by 0 " + e);
                    }
          }
}
```

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. We do this by including **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw.

This is the general form of a method declaration that includes a **throws** clause:

```
Return_type method-name (parameter-list) throws exception-list
{
// body of method
}
```

Program 5.4 **Demonstration of thows statement.**

Solution:

```
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.

Program 5.5 **Demonstration of finally block.**

Solution:

```
class Ex
{
        public static void main(String args[])
        {
        int x, y;
                try
                {
                x = 0;
                y= 1/ x;
                System.out.println("This will not be printed.");
                }
                catch (ArithmeticException e)
                {
                System.out.println("Division by zero.");
                }
                finally
                {
                System.out.println("End of Try/Catch Block");
                }
        }
}
```

Output:

```
Division by zero.
End of Try/Catch Block
```

Java’s Built-in Exceptions

- Inside the standard package **java.lang**, Java defines several exception classes.
- The most general of these exceptions,are subclasses of the standard type **RuntimeException**.
- Since **java.lang** is implicitly,imported into all Java programs, most exceptions derived from **RuntimeException** are automatically available.

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.

User Defined Exceptions

Although Java’s built-in exceptions handle most common errors, we will probably want to create our own exception types to handle situations specific to your applications.

This is quite easy to do: just define a subclass of **Exception** (which is, of course, a subclass of **Throwable**).

Program 5.6 **Demonstration of User Defined Exceptions.**

Solution:

```
class MyException extends Exception
{
        private int detail;
        MyException(int a)
        {
                detail = a;
        }
        public String toString()
        {
                return "MyException[" + detail + "]";
        }
}
class ExceptionDemo
{
        static void compute(int a) throws MyException
        {
                System.out.println("Called compute(" + a + ")");
                if(a > 10)
                throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[])
        {
                try
                {
                compute(1);
                compute(20);
                }
                catch (MyException e)
                {
                System.out.println("Caught " + e);
                }
        }
}
```

Assignment 5

Short Type Questions

1. Differentiate error and exception.

2. What is exception handling mechanism?

3. Can a try block stand alone? Explain with an example.

4. Name the super class of error and exception class.

5. What is Java's built in exceptions?

6. Name 5 keywords used to handle exception in java.

7. Differentiate between final, finalize and finally.

8. What is ArrayIndexOutOfBoundsException?

9. Which block is always executed, no matter whether exception occurs or not?

a) try b) catch c)finally d)finalize

Long Type Questions

1. Discuss exception handling mechanism.

2. Write a program to evaluate x/y, where x and y are integers using exception handling mechanism.

3. Discuss user defined exception in java with an example.

4. Explain the meaning of each keyword: try, catch, throw, throws and finally.

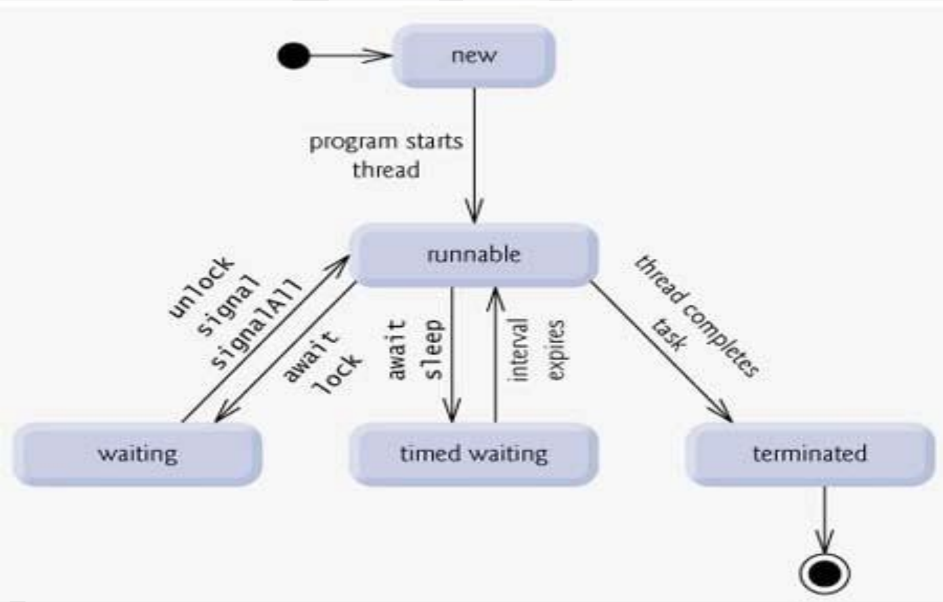
Chapter 6

Multi Threading

- A multithreaded program contains two or more parts that can run concurrently. Each part of such a program is called a thread, and each thread defines a separate path of execution. Thus, multithreading is a specialized form of multitasking.
- There are two distinct types of multitasking: **process-based** and **thread-based**.
- A **process-based** multitasking is the feature that allows our computer to run two or more programs concurrently. For example, process-based multitasking enables us to run the Java compiler at the same time that we are using a music player.
- In a **thread-based** multitasking environment, the thread is the smallest unit of dispatchable code. This means that a single program can perform two or more tasks simultaneously.
- Multitasking threads require less overhead than multitasking processes.
- Multithreading enables us to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum.

Life Cycle of a Thread

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



- **New:** A new thread begins its life cycle in the new state. It remains in this state until the program starts the thread. It is also referred to as a born thread.

- **Runnable:** After a newly born thread is started, the thread becomes runnable. A thread in this state is considered to be executing its task.
- **Waiting:** Sometimes a thread transitions to the waiting state while the thread waits for another thread to perform a task. A thread transitions back to the runnable state only when another thread signals the waiting thread to continue executing.
- **Timed waiting:** A runnable thread can enter the timed waiting state for a specified interval of time. A thread in this state transitions back to the runnable state when that time interval expires or when the event it is waiting for occurs.
- **Terminated:** A runnable thread enters the terminated state when it completes its task or otherwise terminates.

Thread Priority

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others.
- Thread priorities are integers that specify the relative priority of one thread to another.
- Java priorities are in the range between MIN_PRIORITY (a constant of 1) and MAX_PRIORITY (a constant of 10). By default, every thread is given priority NORM_PRIORITY (a constant of 5).
- Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependentant.
- A thread's priority is used to decide when to switch from one running thread to the next. This is called a context switch.

The rules that determine when a context switch takes place are simple:

- **A thread can voluntarily relinquish control.** This is done by explicitly yielding, sleeping, or blocking on pending I/O. In this scenario, all other threads are examined, and the highest-priority thread that is ready to run is given the CPU.
- **A thread can be preempted by a higher-priority thread.** In this case, a lower-priority thread that does not yield the processor is simply preempted—no matter what it is doing— by a higher-priority thread. Basically, as soon as a higher-priority thread wants to run, it does. This is called preemptive multitasking.

How to create Thread

Java defines two ways in which a thread can be created:

I. We can implement the Runnable interface.
II. We can extend the Thread class, itself.

Creating Thread by Implementing Runnable interface

The easiest way to create a thread is to create a class that implements the **Runnable** interface.

- To implement Runnable, a class need only implement a single method called **run( )**, which is declared like this:

```
public void run( )
```

  We will define the code that constitutes the new thread inside run() method. It is important to understand that run() can call other methods, use other classes, and declare variables, just like the main thread can.

- After we create a class that implements Runnable, you will instantiate an object of type Thread from within that class. Thread defines several constructors. The one that we will use is shown here:

```
Thread(Runnable threadOb, String threadName);
```

  Here *threadOb* is an instance of a class that implements the Runnable interface and the name of the new thread is specified by *threadName*.

- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread. The start( ) method is shown here:

```
void start( );
```

Here is an example that creates a new thread and starts it running:

Program 6.1 **Creating thread by implementing Runnable interface.**

Solution:

```
Class NewThread implements Runnable
{
        Thread t;
        NewThread()
         {
         t = new Thread(this, "Demo Thread");
         System.out.println("Child thread: " + t);
         t.start();
         }
        public void run()
        {
                try
                {
                         for(int i = 5; i > 0; i--)
                         {
                         System.out.println("Child Thread: " + i);
                         Thread.sleep(500);
                        }
                }

                catch (InterruptedException e)
                {
                        System.out.println("Child interrupted.");
                }

                System.out.println("Exiting child thread.");
        }
}
class ThreadDemo
{
         public static void main(String args[])
        {
                 new NewThread();
                 try
                {
                         for(int i = 5; i > 0; i--)
                        {
                         System.out.println("Main Thread: " + i);
                        Thread.sleep(1000);
                         }
                 }
                catch (InterruptedException e)
                {
                    System.out.println("Main thread interrupted.");
                }
                System.out.println("Main thread exiting.");
         }
}
```

Output:

```
Child thread: Thread [Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
Main Thread: 2
Main Thread:  1
Main thread exiting.
```

Creating Thread by Extending Thread class

The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class.

The extending class must override the **run ( )** method, which is the entry point for the new thread. It must also call **start ( )** to begin execution of the new thread.

Program 6.2 Creating Thread by Extending Thread class.

Solution:

```
class NewThread extends Thread
{
        NewThread()
        {
        super("Demo Thread");
        System.out.println("Child thread: " + this);
        start();
        }
        public void run()
        {
                try
                {
                        for(int i = 5; i > 0; i--)
                        {
                        System.out.println("Child Thread: " + i);
                        Thread.sleep(500);
```

```
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Child interrupted.");
        }
        System.out.println("Exiting child thread.");
    }
}
class ExtendThread
{
    public static void main(String args[])
    {
        new NewThread();
        try
        {
            for(int i = 5; i > 0; i--)
            {
            System.out.println("Main Thread: " + i);
            Thread.sleep(1000);
            }
        }
        catch (InterruptedException e)
        {
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}
```

Output:

```
Child thread: Thread[Demo Thread,5,main]
Main Thread: 5
Child Thread: 5
Child Thread: 4
Main Thread: 4
Child Thread: 3
Child Thread: 2
Main Thread: 3
Child Thread: 1
Exiting child thread.
```

```
Main Thread: 2
Main Thread:  1
Main thread exiting.
```

How to create Multiple Threads

So far, you have been using only two threads: the main thread and one child thread. However, our program can spawn as many threads as it needs. For example, the following program creates three child threads:

Program 6.3 **Multiple Thread Creation by implementing Runnable interface**

Solution:

```
class NewThread implements Runnable
{
        String name;
        Thread t;
        NewThread(String threadname)
        {
                name = threadname;
                t = new Thread(this, name);
                System.out.println("New thread: " + t);
                t.start(); // Start the thread
        }
        public void run()
        {
        try
        {
                for(int i = 5; i > 0; i--)
                {
                System.out.println(name + ": " + i);
                Thread.sleep(1000);
                }
        }
        catch (InterruptedException e)
        {
                System.out.println(name + "Interrupted");
        }
        System.out.println(name + " exiting.");
        }
}
class MultiThreadDemo
```

```
{
        public static void main(String args[])
        {
                new NewThread("One");
                new NewThread("Two");
                new NewThread("Three");
                try
                {
                        Thread.sleep(10000);
                }
                catch (InterruptedException e)
                {
                        System.out.println("Main thread Interrupted");
                }
                System.out.println("Main thread exiting.");
        }
}
```

Output

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Three: 3
Two: 3
One: 2
Three: 2
Two: 2
One: 1
Three: 1
Two: 1
One exiting.
Two exiting.
Three exiting.
Main thread exiting.
```

As we can see, once started, all three child threads share the CPU. Notice the call to **sleep (10000)** in **main( )**. This causes the main thread to sleep for ten seconds and ensures that it will finish last.

Using isAlive( ) and join( )

Two ways exist to determine whether a thread has finished. First, we can call **isAlive( )** on the thread. This method is defined by **Thread**, and its general form is shown here:

final boolean isAlive( )

The **isAlive( )** method returns **true** if the thread upon which it is called is still running. It returns **false** otherwise.

While **isAlive( )** is occasionally useful, the method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:

final void join ( ) throws InterruptedException

This method waits until the thread on which it is called terminates. Its name comes from the concept of the calling thread waiting until the specified thread joins it. Additional forms of **join ( )** allow you to specify a maximum amount of time that you want to wait for the specified thread to terminate.

Program 6.4 **Use of isAlive() and join() methods.**

Solution:

```
class NewThread implements Runnable
{
        String name;
        Thread t;
        NewThread(String threadname)
        {
        name = threadname;
        t = new Thread(this, name);
        System.out.println("New thread: " + t);
        t.start();
        }

        public void run()
        {
        try
        {
        for(int i = 5; i > 0; i--)
        {
        System.out.println(name + ": " + i);
        Thread.sleep(1000);
        }
        }
        catch (InterruptedException e)
        {
        System.out.println(name + " interrupted.");
        }
        System.out.println(name + " exiting.");
        }
}
class DemoJoin
{
        public static void main(String args[])
        {
        NewThread  ob1  =  new  NewThread("One");
        NewThread  ob2  =  new  NewThread("Two");
        NewThread ob3 = new NewThread("Three");
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        try
        {
        System.out.println("Waiting for threads to finish.");
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
        }
        catch (InterruptedException e)
        {
        System.out.println("Main thread Interrupted");
        }
        System.out.println("Thread One is alive: "+ ob1.t.isAlive());
        System.out.println("Thread Two is alive: "+ ob2.t.isAlive());
        System.out.println("Thread Three is alive: "+ ob3.t.isAlive());
        System.out.println("Main thread exiting.");
        }
}
```

Output:

```
New thread: Thread[One,5,main]
New thread: Thread[Two,5,main]
New thread: Thread[Three,5,main]
```

```
Thread One is alive: true
Thread Two is alive: true
Thread Three is alive: true
Waiting for threads to finish.
One: 5
Two: 5
Three: 5
One: 4
Two: 4
Three: 4
One: 3
Two: 3
Three: 3
One: 2
Two: 2
Three: 2
One: 1
Two: 1
Three: 1
Two exiting.
Three exiting.
One exiting.
Thread One is alive: false
Thread Two is alive: false
Thread Three is alive: false
Main thread exiting.
```

Synchronization

When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called 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. A thread that owns a monitor can reenter the same monitor if it so desires.

Using Synchronized Methods:

Synchronization is easy in Java, because all objects have their own implicit monitor associated with them. To enter an object's monitor, just call a method that has been modified with the **synchronized** keyword. While a thread is inside a synchronized method, all other threads that try to call it (or any other synchronized method) on the same instance have to wait. To exit the monitor and relinquish control of the object to the next waiting thread, the owner of the monitor simply returns from the synchronized method.

Program 6.5 **Thread Synchronization using synchronized method.**

Solution:

```
class Callme
 {
        synchronized 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()
        {
                target.call(msg);
        }
}
class Synch
{
        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");
                // wait for threads to end
                try {
                ob1.t.join();
                ob2.t.join();
                ob3.t.join();
                }
                catch(InterruptedException e)
                {
                System.out.println("Interrupted");
                }
        }
}
```

Output:

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

Using synchronized Statement:

While creating **synchronized** methods within classes that you create is an easy and effective means of achieving synchronization, it will not work in all cases. To understand why, consider the following. Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access. That is, the class does not use **synchronized** methods. Further, this class was not created by you, but by a third party and you do not have access to the source code. Thus, you can't add **synchronized** to the appropriate methods within the class. How can access to an object of this class be synchronized? Fortunately, the solution to this problem is quite easy: You simply put calls to the methods defined by this class inside a **synchronized** block.

This is the general form of the **synchronized** statement:

```
synchronized (object) {
```

```
// statements to be synchronized
}
```

Here, object is a reference to the object being synchronized. A synchronized block ensures that a call to a method that is a member of object occurs only after the current thread has successfully entered object's monitor.

Program 6.6 **Thread Synchronization using synchronized statement.**

Solution:

```
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");
        try
         {
        ob1.t.join();
        ob2.t.join();
        ob3.t.join();
        }
        catch(InterruptedException e)
        {

        System.out.println("Interrupted");
        }
        }
}
```

Output:

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

Wait, notify and notifyAll

wait( ) tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**. **notify( )** wakes up the first thread that called **wait( )** on the same object. **notifyAll( )** wakes up all the threads that called **wait( )** on the same object. The highest priority thread will run first.

These methods are declared within **Object**, as shown here:

final void wait( ) throws InterruptedException
final void notify( )
final void notifyAll( )

Assignment 6

Short Type Questions

1. Define thread. Differentiate between thread based multi tasking and process based multi tasking.
2. Why thread priorities are set?
3. Define race condition.
4. Define thread synchronization.
5. How a thread is implemented in java?
6. Differentiate between wait () and sleep method.
7. Differentiate between notify () and notifyAll().

Long Type Questions

1. Define thread. Discuss the life cycle of a thread with neat diagram.
2. Discuss the importance of thread priorities.
3. Discuss how synchronization in threads is achieved using synchronized method.
4. Explain how a thread is created by implementing Runnable interface.
5. Explain how a thread is created by extending thread class.
6. Write Short notes on following:
    a) isAlive()
    b) join()
    c) notify()
    d) notifyAll()
    e) wait()
    f) sleep()

String

- A string is a set of one or more characters enclosed in double quotes or simply a string is an array of characters.
- Strings, which are widely used in Java programming, are a sequence of characters. In the Java programming language, **strings are objects**.
- The Java platform provides the String class to create and manipulate strings.

String Constructor

The **String** class supports several constructors as follows:

1. To create an empty **String**, we call the default constructor.

For example,

String s = new String(); will create an instance of **String** with no characters in it.

2. To create a **String** initialized by an array of characters, use the constructor shown here:

String(char chars[ ])

Here is an example:

char chars[] = { 'a', 'b', 'c' };

String s = new String(chars);

This constructor initializes **s** with the string “abc”.

3. We can specify a subrange of a character array as an initializer using the following constructor:

String(char chars[ ], int startIndex, int numChars)

Here, startIndex specifies the index at which the subrange begins, and numChars specifies the number of characters to use. Here is an example:

char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };

String s = new String(chars, 2, 3);

This initializes **s** with the characters **cde**.

4. We can construct a **String** object that contains the same character sequence as another **String** object using this constructor:

String(String strObj)

Here, strObj is a **String** object. Consider this example:

Program 7.1 Initialization of string using string constructor.

Solution:

```
class MakeString
{
        public static void main(String args[])
        {
        char c[ ] = {'J', 'a', 'v', 'a'};
        String s1 = new String (c);
        String s2 = new String (s1);
        System.out.println (s1);
        System.out.println (s2);
        }
}
```

Output:

```
Java
Java
```

Even though Java’s **char** type uses 16 bits to represent the Unicode character set, the typical format for strings on the Internet uses arrays of 8-bit bytes constructed from the ASCII character set. Because 8-bit ASCII strings are common, the **String** class provides constructors that initialize a string when given a **byte** array. Their forms are shown here:

String (byte asciiChars[ ])
String(byte asciiChars [ ], int startIndex, int numChars)
Here, asciiChars specifies the array of bytes. The second form allows you to specify a subrange. In each of these constructors, the byte-to-character conversion is done by using the default character encoding of the platform. The following program illustrates these constructors:

Program 7.2 String initialization using byte array.

Solution:

```
        class SubStringCons
        {
                public static void main(String args[])
                {
                        byte ascii[] = {65, 66, 67, 68, 69, 70 };
                        String s1 = new String(ascii);
                        System.out.println(s1);
                        String s2 = new String(ascii, 2, 3);
                        System.out.println(s2);
                }
        }
```

Output:

```
ABCDEF
CDE
```

String Length

The length of a string is the number of characters that it contains. To obtain this value, call the **length( )** method, shown here:

int length( )

The following fragment prints “3”, since there are three characters in the string **s**:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s.length());
```

Character Extraction

The **String** class provides a number of ways in which characters can be extracted from a **String** object. Each is examined here. Although the characters that comprise a string within a **String** object cannot be indexed as if they were a character array, many of the **String** methods employ an index (or offset) into the string for their operation. Like arrays, the string indexes begin at zero.

charAt( )

To extract a single character from a **String**, you can refer directly to an individual character via the **charAt( )** method. It has this general form:

char charAt(int where)

Here, where is the index of the character that you want to obtain. The value of where must be nonnegative and specify a location within the string. **charAt( )** returns the character at the specified location. For example,

```
char ch;
ch = "abc".charAt(1);
```

assigns the value “**b**” to **ch**.

getChars( )

If you need to extract more than one character at a time, you can use the **getChars( )** method. It has this general form:

void getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart) Here, sourceStart specifies the index of the beginning of the substring, and sourceEnd specifies an index that is one past the end of the desired substring. Thus, the substring contains the characters from sourceStart through sourceEnd–1. The array that will receive the characters is specified by target. The index within target at which the substring will be copied is passed in targetStart. Care must be taken to assure that the target array is large enough to hold the number of characters in the specified substring.

Program 7.3 **Demonstration of getChars( ).**

Solution:

```
class getCharsDemo
{
        public static void main(String args[])
        {
        String s = "This is a demo of the getChars method.";
        int start = 10;
        int end = 14;
        char buf[] = new char[end - start];
        s.getChars(start, end, buf, 0);
        System.out.println(buf);
        }
}
```

Output:

```
demo
```

String Comparisons

To compare two strings for equality, use **equals( )**. It has this general form:

boolean equals(Object str)

Here, str is the **String** object being compared with the invoking **String** object. It returns **true** if the strings contain the same characters in the same order, and **false** otherwise. The comparison is case- sensitive. To perform a comparison that ignores case differences, call **equalsIgnoreCase( )**. When it compares two strings, it considers **A-Z** to be the same as **a-z**. It has this general form:

boolean equalsIgnoreCase(String str)

Here, str is the **String** object being compared with the invoking **String** object. It, too, returns **true** if the strings contain the same characters in the same order, and **false** otherwise.

Here is an example that demonstrates **equals( )** and **equalsIgnoreCase( )**:

Program 7.4 **Demonstration of String comparisons.**

Solution:

```
class equalsDemo
{
        public static void main(String args[])
        {
        String s1 = "Hello";
        String s2 = "Hello";
        String s3 = "Good-bye";
        String s4 = "HELLO";
        System.out.println(s1 + " equals " + s2 + " -> " +
        s1.equals(s2));
        System.out.println(s1 + " equals " + s3 + " -> " +
        s1.equals(s3));
        System.out.println(s1 + " equals " + s4 + " -> " +
        s1.equals(s4));
        System.out.println(s1 + " equalsIgnoreCase " + s4 + " -> " +
        s1.equalsIgnoreCase(s4));
        }
}
```

Output:

```
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
```

Modifying a String

Because **String** objects are immutable, whenever we want to modify a **String**, we must either copy it into a **StringBuffer** or use one of the following **String** methods, which will construct a new copy of the string with your modifications complete. We can extract a substring using **substring( )**. It has two forms:

The first is String substring(int startIndex).Here, startIndex specifies the index at which the substring will begin. This form returns a copy of the substring that begins at startIndex and runs to the end of the invoking string.

The second form of **substring( )** allows you to specify both the beginning and ending index of the substring:String substring(int startIndex, int endIndex). Here, startIndex specifies the beginning index, and endIndex specifies the stopping point. The string returned contains all the characters from the beginning index, up to, but not including, the ending index.

Program 7.5 **Demonstration of string modification.**

Solution:

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

```
          {
          String org = "This is a test. This is, too.";
          String search = "is";
          String sub = "was";
          String result = "";
          int i;
          do
           {
                    System.out.println(org);
                    i = org.indexOf(search);
                    if(i != -1)
                    {
                    result = org.substring(0, i);
                    result = result + sub;
                    result = result + org.substring(i + search.length());
                    org = result;
                    }
                    } while(i != -1);
          }
}
```

Output:

```
This is a test. This is, too.
Thwas is a test. This is, too.
Thwas was a test. This is, too.
Thwas was a test. Thwas is, too.
Thwas was a test. Thwas was, too.
```

Some Important String function

We can concatenate two strings using **concat( )**, shown here:

String concat(String str)

This method creates a new object that contains the invoking string with the contents of str appended to the end. **concat( )** performs the same function as **+**. For example, String s1 = "one";

String s2 = s1.concat("two"); puts the string “onetwo” into **s2**. It generates the same result as the following sequence:

String s1 = "one";

String s2 = s1 + "two";

The **replace ( )** method replaces all occurrences of one character in the invoking string with another character. It has the following general form:

String replace (char original, char replacement)
Here, original specifies the character to be replaced by the character specified by replacement. The resulting string is returned. For example,
String s = "Hello".replace ('l', 'w'); puts the string “Hewwo” into **s**.

The **trim ( )** method returns a copy of the invoking string from which any leading and trailing whitespace has been removed.

Assignment 7

Short Type Questions

Long Type Questions

Java I/O

- A stream is continuous group of data or a channel through which data flows from one point to another point.
- Java implements streams within class hierarchies defined in the **java.io** package.
- **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. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

Stream

Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system. All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection. Streams are a clean way to deal with input/output without having every part of your code understands the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the **java.io** package. Java 2 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. They use Unicode and, therefore, can be internationalized. Also, in some cases, character streams are more efficient than byte streams.

The Byte Stream Classes

- Byte streams are defined by using two class hierarchies. At the top are two abstract classes: **InputStream** and **OutputStream**.
- To use the stream classes, we must import **java.io**.
- The abstract classes **InputStream** and **OutputStream** define several key methods that the other stream classes implement.
- Two of the most important are **read ( )** and **write ( )**, which, respectively, read and write bytes of data. Both methods are declared as abstract inside **InputStream** and **OutputStream**. They are overridden by derived stream classes.

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. Java has several concrete subclasses of each of these.
- The abstract classes **Reader** and **Writer** define several key methods that the other stream classes implement. Two of the most important methods are **read ( )** and write **( )**, which read and write characters of data, respectively. These methods are overridden by derived stream classes.

Java System Class

- All Java programs automatically import the **java.lang** package.
- This package defines a class called **System**, which encapsulates several aspects of the run-time environment.
- **System** also contains three predefined stream variables, **in**, **out**, and **err**. These fields are declared as **public** and **static** within **System**. This means that they can be used by any other part of your program and without reference to a specific **System** object.
- **System.out** refers to the standard output stream. By default, this is the console.
- **System.in** refers to standard input, which is the keyboard by default.
- **System.err** refers to the standard error stream, which also is the console by default.
- **System.in** is an object of type **InputStream**; **System.out** and **System.err** are objects of type **PrintStream**. These are byte streams, even though they typically are used to read and write characters from and to the console.

Byte Stream Class

Stream	**Class Meaning**
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
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
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

Character Stream Class

Stream	**Class Meaning**
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

Reading Console Input

In Java, console input is accomplished by reading from **System.in**. To obtain a character-based stream that is attached to the console, you wrap **System.in** in a **BufferedReader** object, to create a character stream. **BuffereredReader** supports a buffered input stream. Its most commonly used constructor is shown here:

BufferedReader(Reader inputReader)

Here, inputReader is the stream that is linked to the instance of **BufferedReader** that is being created. **Reader** is an abstract class. One of its concrete subclasses is **InputStreamReader**, which converts bytes to characters. To obtain an **InputStreamReader** object that is linked to **System.in**, use the following constructor:

```
InputStreamReader(InputStream inputStream)
```

Because **System.in** refers to an object of type **InputStream**, it can be used for inputStream. Putting it all together, the following line of code creates a **BufferedReader** that is connected to the keyboard:

```
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

After this statement executes, **br** is a character-based stream that is linked to the console through **System.in.**

Reading Characters

To read a character from a **BufferedReader**, use **read( )**. The version of **read( )** that we will be using is int read( ) throws IOException Each time that **read( )** is called, it reads a character from the input streamand returns it as an integer value. It returns –1 when the end of the stream is encountered.

Reading Strings

To read a string from the keyboard, use the version of **readLine( )** that is a member of the **BufferedReader** class. Its general form is shown here:

```
String readLine( ) throws IOException
```

Program 8. 1 **Write a program to find addition of two numbers given from the keyboard. Using exception handling, display appropriate message if the inputs are not valid numbers.**

Solution:

```
import java.io.*;
class Addition
{
         public static void main(String args[])
        {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        int x, y, z;
                try
                {
                System.out.println("Enter 1st number");
                str = br.readLine();
                x=Integer.parseInt(str);
                System.out.println("Enter 2nd number");
                str = br.readLine();
                y=Integer.parseInt(str);
                z=x+y;
                System.out.print(“Addition Result is: ”+z);
                }
                catch (Exception e)
                {
                System.out.println("Invalid Number ..Try Again!!!!!!!");
                }
        }
}
```

Program 8. 2 **Write a program to find factorial of an integer given from the keyboard. Using exception handling mechanism, display appropriate message if the input from keyboard is not a valid integer.**

Solution:

```
import java.io.*;
class Factorial
{
         public static void main(String args[])
        {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String str;
        int n, f=1, i;
                try
                {
                System.out.println("Enter the number");
                str = br.readLine();
                n=Integer.parseInt(str);

                for(i=1; i<=n; i++)
                f=f*i;
                System.out.print(“Factorial is: ”+f);
                }
                catch (Exception e)
                {
                System.out.println("Invalid  Input ..Try Again!!!!!!!");
                }
        }
}
```

Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for deserializing and serializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out:

```
public final void writeObject(Object x) throws IOException
```

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object:

```
public final Object readObject() throws IOException, ClassNotFoundException
```

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note: When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Program 8. 3 **Demonstration of object serialization.**

Solution:

```
import java.io.*;
class Employee implements java.io.Serializable
{
  public String name;
  public String address;
  public transient int age;
  public int number;
  public void mailCheck()
  {
    System.out.println("Mailing a check to  " + name + "  " + address);
  }
}

public class SerializeDemo
{
  public static void main(String [] args)
  {
                Employee e = new Employee();
                e.name = "Sourav Kumar Giri";
                e.address = "Baleswar, Odisha.";
                e.age =25;
                e.number = 101;
                try
                {
                  FileOutputStream fileOut =new FileOutputStream("employee.ser");
                  ObjectOutputStream out =new ObjectOutputStream(fileOut);
                  out.writeObject(e);
                  out.close();
                  fileOut.close();
                }
                catch(IOException i)
                {
                   i.printStackTrace();
                }
  }
}
```

Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output:

Program 8. 4 **Demonstration of object de-serialization.**

Solution:

```
import java.io.*;
public class DeserializeDemo
{
  public static void main(String [] args)
  {
               Employee e = null;
               try
               {
                 FileInputStream fileIn =new FileInputStream("employee.ser");
                 ObjectInputStream in = new ObjectInputStream(fileIn);
                 e = (Employee) in.readObject();
                 in.close();
                 fileIn.close();
               }
               catch(IOException i)
               {
                 i.printStackTrace();
                 return;
               }
               catch(ClassNotFoundException c)
               {
                 System.out.println("Employee class not found");
                 c.printStackTrace();
                 return;
               }
               System.out.println("Deserialized Employee...");
               System.out.println("Name: " + e.name);
               System.out.println("Address: " + e.address);
               System.out.println("Age " + e.age);
               System.out.println("Number: " + e.number);
  }
}
```

Output:

```
Deserialized Employee...
Name: Sourav Kumar Giri
Address: Baleswar, Odisha
Age: 0
Number:101
```

Here are following important points to be noted:

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the age field was 25 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The age field of the deserialized Employee object is 0.

Assignment 8

Short Type Questions

1. Define Stream.
2. Differentiate between byte stream and character stream.
3. Define Serialization.
4. Differentiate between read() and readLine() in BufferedReader class.

Long Type Questions

1. Explain byte stream and character stream in java.
2. What is serialization in java? How to serialize and de-serialize an object in java? Explain with an example.
3. Write a program to accept 3 numbers from the keyboard and display greatest among them. Use exception handling for appropriate input (i.e. the inputs must be number only).
4. Write a program to accept a number from the keyboard and check whether the number is prime or not.

JDBC

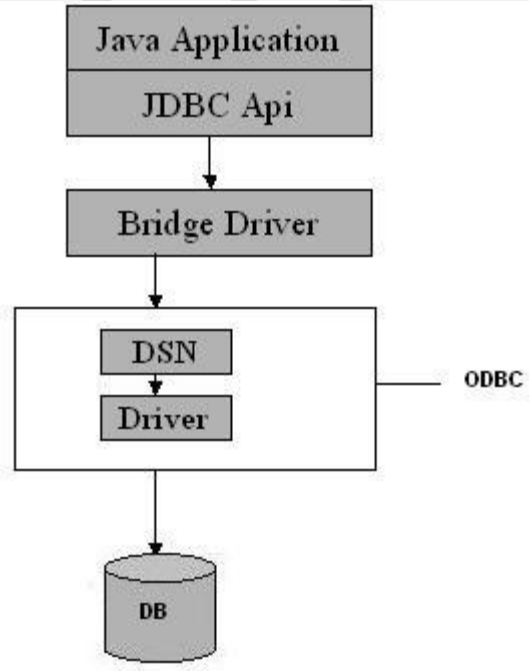
Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases. The application program interface lets you encode access request statements in Structured Query Language (SQL) that are then passed to the program that manages the database.

JDBC drivers are divided into four types or levels. The different types of jdbc drivers are:

Type 1: JDBC-ODBC Bridge driver (Bridge)
Type 2: Native-API/partly Java driver (Native)
Type 3: All Java/Net-protocol driver (Middleware)
Type 4: All Java/Native-protocol driver (Pure)

JDBC-ODBC Bridge driver (Type 1)

The Type 1 driver translates all JDBC calls into ODBC calls and sends them to the ODBC driver. ODBC is a generic API. The JDBC-ODBC Bridge driver is recommended only for experimental use or when no other alternative is available.



[Type 1: JDBC-ODBC Bridge]

Advantage

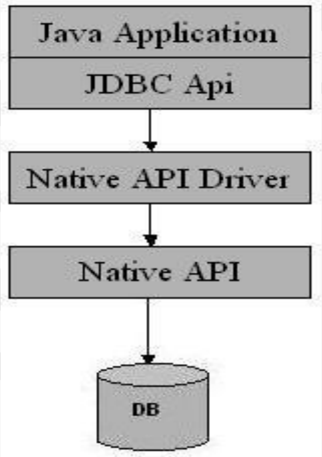
The JDBC-ODBC Bridge allows access to almost any database, since the database's ODBC drivers are already available.

Disadvantages

1. Since the Bridge driver is not written fully in Java, Type 1 drivers are not portable.
2. A performance issue is seen as a JDBC call goes through the bridge to the ODBC driver, then to the database, and this applies even in the reverse process. They are the slowest of all driver types.
3. The client system requires the ODBC Installation to use the driver.
4. Not good for the Web.

Native-API/ partly Java driver(Type 2)

The distinctive characteristic of type 2 jdbc drivers is that Type 2 drivers convert JDBC calls into database-specific calls i.e. this driver is specific to a particular database. Some distinctive characteristic of type 2 jdbc drivers are shown below. Example: Oracle will have oracle native API.



[Type 2: Native api/ Partly Java Driver]

Advantage

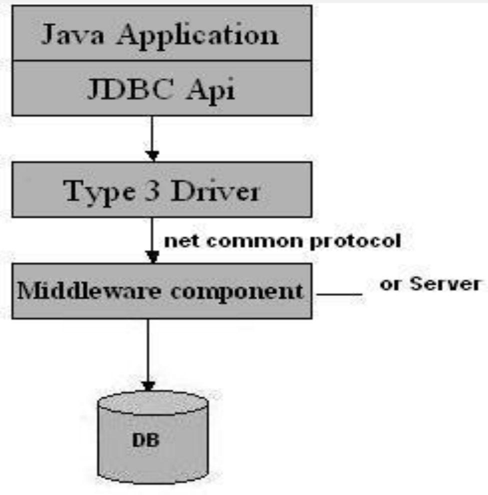
The distinctive characteristic of type 2 jdbc drivers are that they are typically offer better performance than the JDBC-ODBC Bridge as the layers of communication (tiers) are less than that of Type 1 and also it uses Native api which is Database specific.

Disadvantages

1. Native API must be installed in the Client System and hence type 2 drivers cannot be used for the Internet.
2. Like Type 1 drivers, it's not written in Java Language which forms a portability issue.
3. If we change the Database we have to change the native API as it is specific to a database.
4. Mostly obsolete now.
5. Usually not thread safe.

All Java/Net-protocol driver (Type 3)

Type 3 database requests are passed through the network to the middle-tier server. The middle-tier then translates the request to the database. If the middle-tier server can in turn use Type1, Type 2 or Type 4 drivers.



[Type 3: All Java/ Net-Protocol Driver]

Advantages

1. This driver is server-based, so there is no need for any vendor database library to be present on client machines.
2. This driver is fully written in Java and hence Portable. It is suitable for the web.
3. There are many opportunities to optimize portability, performance, and scalability.
4. The net protocol can be designed to make the client JDBC driver very small and fast to load.
5. The type 3 driver typically provides support for features such as caching (connections, query results, and so on), load balancing, and advanced
system administration such as logging and auditing.
6. This driver is very flexible allows access to multiple databases using one driver.

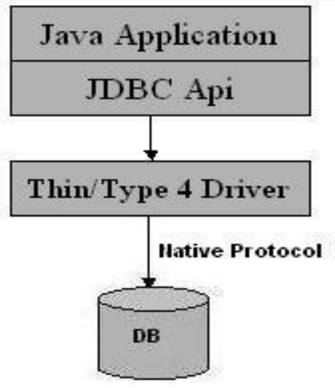
7. They are the most efficient amongst all driver types.

Disadvantage

It requires another server application to install and maintain. Traversing the record set may take longer, since the data comes through the backend server.

Native-protocol/all-Java driver(Type 4)

The Type 4 uses java networking libraries to communicate directly with the database server.



[Type 4: Native-protocol/all-Java driver]

Advantages

1. The major benefit of using a type 4 jdbc drivers are that they are completely written in Java to achieve platform independence and eliminate deployment administration issues. It is most suitable for the web.
2. Number of translation layers is very less i.e. type 4 JDBC drivers don't have to translate database requests to ODBC or a native connectivity interface or to pass the request on to another server, performance is typically quite good.
3. You don't need to install special software on the client or server. Further, these drivers can be downloaded dynamically.

Disadvantage

With type 4 drivers, the user needs a different driver for each database.

Assignment 9

Short Type Questions

1. What is JDBC?

Long Type Questions

1. Explain different JDBC drivers in details. Also draw the diagrams.

Java Networking

A socket is an unique address which is combination of IP and Port address. An IP address uniquely identifies a particular computer whereas each process has a unique port number. Each computer connected to the Internet has a unique IP address. An IP address is a 32-bit number used to uniquely identify each computer connected to the Internet. Example: 192.1.1.1. A **port address** is a 16-bit number that identifies each service offered by a network server. The **FTP** service is located on port 21. The **HTTP** service is located on port 80. A Domain Naming Service (DNS) is the name given to an IP address.

The Networking Classes and Interfaces

The classes contained in the **java.net** package are as follows:

Authenticator (Java 2)
ContentHandler
DatagramPacket
DatagramSocket
DatagramSocket
HttpURLConnection
InetAddress
Inet4Address (Java 2, v1.4)
Inet6Address (Java 2, v1.4)
InetSocketAddress (Java 2, v1.4)
JarURLConnection (Java 2)
MulticastSocket
NetPermission
Impl NetworkInterface (Java 2, v1.4)
PasswordAuthentication (Java 2)
ServerSocket
Socket
SocketAddress (Java 2, v1.4)
SocketImpl
SocketPermission
URI (Java 2, v1.4)
URL
URLClassLoader (Java 2)
URLConnection
URLDecoder (Java 2)
URLEncoder
URLStreamHandler

The **InetAddress** class is used to encapsulate both the numerical IP address and the domain name for that address. **InetAddress** can handle both IPv4 and IPv6 addresses. The **InetAddress** class has no visible constructors. To create an **InetAddress** object, you have to use one of the available factory methods. Factory methods are merely a convention whereby static methods in a class return an instance of that class.

Three commonly used **InetAddress** factory methods are as follows:

- static InetAddress getLocalHost( ) throws UnknownHostException
- static InetAddress getByName(String hostName) throws UnknownHostException
- static InetAddress[ ] getAllByName(String hostName) throws UnknownHostException

The **getLocalHost( )** method simply returns the **InetAddress** object that represents the local host. The **getByName( )** method returns an **InetAddress** for a host name passed to it. If these methods are unable to resolve the host name, they throw an **UnknownHostException**. On the Internet, it is common for a single name to be used to represent several machines. In the world of web servers, this is one way to provide some degree of scaling. The getAllByName**( )** factory method returns an array of **InetAddress**es that represent all of the addresses that a particular name resolves to. It will also throw an **UnknownHostException** if it can't resolve the name to at least one address.

TCP/IP Socket

A socket can be used to connect Java's I/O system to other programs that may reside either on the local machine or on any other machine on the Internet. There are two kinds of TCP sockets in Java. One is for servers, and the other is for clients. The **ServerSocket** class is designed to be a "listener," which waitsfor clients to connect before doing anything. The **Socket** class is designed to connect to server sockets and initiate protocol exchanges.

TCP/IP Client Sockets

Here are two constructors used to create client sockets:

Socket(String hostName, int port)	Creates a socket connecting the localhost to the named host and port; can throw an **UnknownHostException** oran **IOException**.
Socket(InetAddress ipAddress, int port)	Creates a socket using a preexisting **InetAddress** object and a port; can throw an **IOException**.

A socket can be examined at any time for the address and port information associated with it, by use of the following methods:

InetAddress getInetAddress( )	Returns the **InetAddress** associated with the **Socket** object.
int getPort( )	Returns the remote port to which this **Socket** object is connected.
int getLocalPort( )	Returns the local port to which this **Socket** object is connected

Once the **Socket** object has been created, it can also be examined to gain access to the input and output streams associated with it.

InputStream getInputStream( )	Returns the **InputStream** associated with the invoking socket.
OutputStream getOutputStream( )	Returns the **OutputStream** associated with the invoking socket.

Whois

The very simple example that follows opens a connection to a whois port on the InterNIC server, sends the command -line argument down the socket, and then prints the data that is returned.InterNIC will try to lsookup the argument as a registered Internet domain name, then send back the IPaddress and contact information for that site.

Program 10.1

```
import java.net.*;
import java.io.*;
class Whois
{
        public static void main(String args[]) throws Exception
        {
        int c;
        Socket s = new Socket("internic.net", 43);
        InputStream in = s.getInputStream();
        OutputStream out = s.getOutputStream();
        Stringstr=(args.length==0? "osborne.com":args[0])+"n";
        byte buf[] = str.getBytes();
        out.write(buf);
        while ((c = in.read()) != -1)
        System.out.print((char) c);
        s.close();
        }
}
```

URL (Uniform Resource Locator)

The URL (Uniform Resource Locator) provides a reasonably intelligible form to uniquely identify or address information on the Internet. URLs are ubiquitous; every browser uses them to identify information on the Web. In fact, the Web is really just that same old Internet with all of its resources addressed as URLs plus HTML. Within Java's network class library, the **URL** class provides a simple, concise API to access information across the Internet using URLs. Two examples of URLs are **http://www.osborne.com/** and **http://www.osborne.com:80/ index.htm**.

A URL specification is based on four components:

- The first is the protocol to use, separated from the rest of the locator by a colon (:). Common protocols are http, ftp, gopher, and file, although these days almost everything is being done via HTTP (in fact, most browsers will proceed correctly if you leave off the "http://" from your URL specification).

- The second component is the host name or IP address of the host to use; this is delimited on the left by double slashes (//) and on the right by a slash (/) or optionally a colon (:).
- The third component, the port number, is an optional parameter, delimited on the left from the host name by a colon (:) and on the right by a slash (/). (It defaults to port 80, the predefined HTTP port; thus ":80" is redundant.)
- The fourth part is the actual file path. Most HTTP servers will append a file named **index.html** or **index.htm** to URLs that refer directly to a directory resource.

Program 10.2 Demonstration of URL information

Solution:

```
import java.net.*;
class URLDemo
{
        public static void main(String args[]) throws MalformedURLException
         {
        URL hp = new URL("http://www.srinix.org/downloads");
        System.out.println("Protocol: " + hp.getProtocol());
        System.out.println("Port: " + hp.getPort());
        System.out.println("Host: " + hp.getHost());
        System.out.println("File: " + hp.getFile());
        System.out.println("Ext:" + hp.toExternalForm());
        }
}
```

Output:

```
Protocol: http
Port: -1
Host: www.srinix.org
File: /downloads
Ext: http://www.srinix.org/downloads
```

URLConnection

URLConnection is a general-purpose class for accessing the attributes of a remote resource. Once you make a connection to a remote server, you can use **URLConnection** to inspect the properties of the remote object before actually transporting it locally. These attributes are exposed by the HTTP protocol specification and, as such, only make sense for **URL** objects that are using the HTTP protocol.

To access the actual bits or content information of a **URL**, we create a **URLConnection** object from it, using its **openConnection( )** method, like this:
url.openConnection()

openConnection( ) has the following general form:

URLConnection openConnection( )

It returns a **URLConnection** object associated with the invoking **URL** object. It may throw an **IOException**.

Program 10.3 **Demonstration of URL connection.**

Solution:

```
import java.net.*;
import java.io.*;
import java.util.Date;
class UCDemo
{
        public static void main(String args[]) throws Exception
         {
        int c;
        URL hp = new URL("http://www.srinix.org");
        URLConnection hpCon = hp.openConnection();
        long d = hpCon.getDate();
        if(d==0)
        System.out.println("No date information.");
        else
        System.out.println("Date: " + new Date(d));

        System.out.println("Content-Type: " + hpCon.getContentType());
        }
}
```

Output:

```
Date: Wed June 12 23:10:53 IST 2013
Content-Type: text/html
```

TCP/IP Server Sockets

The **ServerSocket** class is used to create servers that listen for either local or remote client programs to connect to them on published ports. Since the Web is driving most of the activity on the Internet, this section develops an operational web (http) server. **ServerSocket**s are quite different from normal **Socket**s. When you create a **ServerSocket**, it will register itself with the system as having an interest in client connections. The constructors for **ServerSocket** reflect the port number that you wish to accept connections on and, optionally, how long you want the queue for said port to be. The queue length tells the system how many client connections it can leave pending before it should simply refuse connections. The default is 50. The constructors might throw an **IOException** under adverse conditions. Here are the constructors:

ServerSocket(int port)	Creates server socket on the specified port with a queue length of 50.
ServerSocket(int port, int maxQueue)	Creates a server socket on the specified port with a maximum queue length of maxQueue.
ServerSocket(int port, int maxQueue, InetAddress localAddress)	Creates a server socket on the specified port with a maximum queue length of maxQueue. On a multihomed host, localAddress specifies the IP address to which this socket binds.

ServerSocket has a method called **accept( )**, which is a blocking call that will wait for a client to initiate communications, and then return with a normal **Socket** that is then used for communication with the client.

Serve- Client Program in JAVA

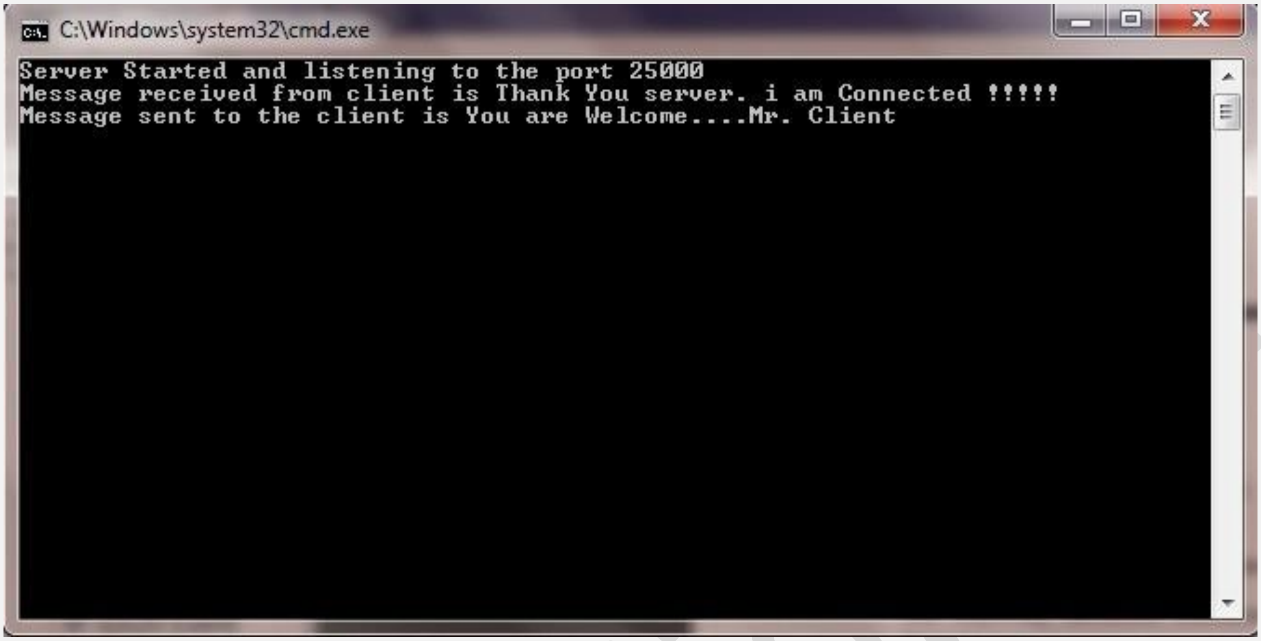
Program 10.4 Server Side Program (Server.java)

```
import java.io.*;
import java.net.*;
public class Server
{
private static Socket socket;
public static void main(String[] args)
{
        try
        {

              int port = 25000;
              ServerSocket serverSocket = new ServerSocket(port);
              System.out.println("Server Started and listening to the port 25000");

              while(true)
              {
                socket = serverSocket.accept();
                InputStream is = socket.getInputStream();
                InputStreamReader isr = new InputStreamReader(is);
                BufferedReader br = new BufferedReader(isr);
                String number = br.readLine();
                System.out.println("Message received from client is "+number);

                String returnMessage="You are Welcome....Mr. Client";
                OutputStream os = socket.getOutputStream();
                OutputStreamWriter osw = new OutputStreamWriter(os);
                BufferedWriter bw = new BufferedWriter(osw);
                bw.write(returnMessage);
                System.out.println("Message sent to the client is "+returnMessage);
                bw.flush();
              }
         }
        catch (Exception e)
        {
                e.printStackTrace();
        }
        finally
         {
                 try
                {
                socket.close();
                }
                catch(Exception e){}
        }
}
}
```

Program 10.5 Client Side Program (Client.java)

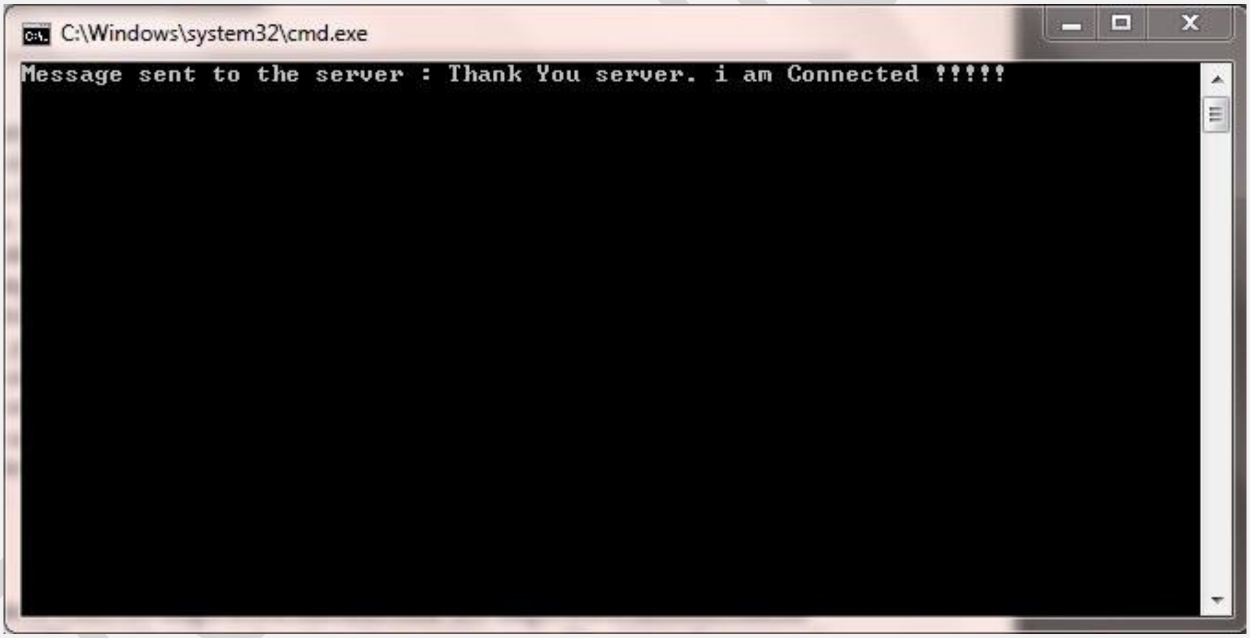
```
import java.io.*;
import java.net.*;
public class Client
{
private static Socket socket;
public static void main(String args[])
{
        try
        {
              String host = "localhost";
              int port = 25000;
              InetAddress address = InetAddress.getByName(host);
              socket = new Socket(address, port);

              OutputStream os = socket.getOutputStream();
              OutputStreamWriter osw = new OutputStreamWriter(os);
              BufferedWriter bw = new BufferedWriter(osw);

              String number = "Thank You server. i am Connected !!!!!";
              String sendMessage = number + "\n";
              bw.write(sendMessage);
              bw.flush();
              System.out.println("Message sent to the server : "+sendMessage);

              InputStream is = socket.getInputStream();
              InputStreamReader isr = new InputStreamReader(is);
              BufferedReader br = new BufferedReader(isr);
              String message = br.readLine();
              System.out.println("Message received from the server : " +message);
          }
            catch (Exception exception)
            {
              exception.printStackTrace();
            }
          finally
          {
              try
              {
               socket.close();
              }
              catch(Exception e)
              {
               e.printStackTrace();
              }
        }
}
}
```

Server Side Output:



```
Server Started and listening to the port 25000
Message received from client is Thank You server. i am Connected !!!!!
Message sent to the client is You are Welcome....Mr. Client
```

Client Side Output:



```
Message sent to the server : Thank You server. i am Connected !!!!!
```

Assignment 10

Short Type Questions

1. Define a socket.
2. Define IP address and Port address.
3. Define Whois.
4. Define URL.
5. Differentiate between Socket and ServerSocket.
6. What is the purpose of URLConnection class.
7. Write the purpose of getLocalHost() and getByName() method.

Long Type Questions

1. Define URL. Explain URL format in details.
2. What are client and server socket in java? Explain them with an example.
3. Write a server client program in java to exchange information among them.

APPLETS

Applets are the java program that can be run on a java enabled web browser. An applet is typically embedded inside a web page and runs in the context of a browser. The Applet class provides the standard interface between the applet and the browser environment.

All applets are subclasses of **Applet**. Thus, all applets must import **java.applet**. Applets must also import **java.awt**. Execution of an applet does not begin at **main( )**. Actually, few applets even have **main( )** methods. Instead, execution of an applet is started and controlled with an entirely different mechanism.

Once an applet has been compiled, it is included in an HTML file using the APPLET tag. The applet will be executed by a Java-enabled web browser when it encounters the APPLET tag within the HTML file. To view and test an applet more conveniently, simply include a comment at the head of your Java source code file that contains the APPLET tag. This way, your code is documented with the necessary HTML statements needed by your applet, and you can test the compiled applet by starting the applet viewer with your Java source code file specified as the target. Here is an example of such a comment:

```
/*
<applet code="MyApplet" width=200 height=60>
</applet>
*/
```

This comment contains an APPLET tag that will run an applet called MyApplet in a window that is 200 pixels wide and 60 pixels high. Since the inclusion of an APPLET command makes testing applets easier, all of the applets shown in this book will contain the appropriate APPLET tag embedded in a comment.

Applet Initialization and Termination

It is important to understand the order in which the various methods shown in the skeleton are called. When an applet begins, the AWT calls the following methods, in this sequence:

1. init( )
2. start( )
3. paint( )

When an applet is terminated, the following sequence of method calls takes place:

1. stop( )
2. destroy( )

Let’s look more closely at these methods.

init( )

The **init( )** method is the first method to be called. This is where you should initialize variables. This method is called only once during the run time of your applet.

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 an applet's HTML document is displayed onscreen. So, if a user leaves a web page and comes back, the applet resumes execution at start( ).

paint( )

The paint( ) method is called each time your applet's output must be redrawn. This situation can occur for several reasons. For example, the window in which the applet is running may be overwritten by another window and then uncovered. Or the applet window may be minimized and then restored. paint( ) is also called when the applet begins execution. Whatever the cause, whenever the applet must redraw its output, paint( ) is called. The paint( ) method has one parameter of type Graphics. This parameter will contain the graphics context, which describes the graphics environment in which the applet is running. This context is used whenever output to the applet is required.

stop( )

The stop( ) method is called when a web browser leaves the HTML document containing the applet—when it goes to another page, for example. When stop( ) is called, the applet is probably running. You should use stop( ) to suspend threads that don't need to run when the applet is not visible. You can restart them when start( ) is called if the user returns to the page.

destroy( )

The destroy( ) method is called when the environment determines that your applet needs to be removed completely from memory. At this point, you should free up any resources the applet may be using. The stop( ) method is always called before destroy( ).

Program 11.1 **Demonstration of an applet skeleton.**

Solution:

```
import java.awt.*;
import java.applet.*;
/* <applet code="AppletSkel" width=300 height=100> </applet> */

public class AppletSkel extends Applet
 {

        public void init()
         {
                // initialization
        }
        public void start()
        {
                // start or resume execution
        }
        public void stop()
         {
                // suspends execution
        }
        public void destroy()
        {
                // perform shutdown activities
        }
        public void paint(Graphics g)
        {
                // redisplay contents of window
        }
}
```

The HTML APPLET Tag

The APPLET tag is used to start an applet from both an HTML document and from an applet viewer. An applet viewer will execute each APPLET tag that it finds in a separate window, while web browsers like Netscape Navigator, Internet Explorer, and Hot Java will allow many applets on a single page.

The syntax for the standard APPLET tag is shown here. Bracketed items are optional.

```
< APPLET   [CODEBASE = codebaseURL]        CODE = appletFile      [ALT = alternateText]    [NAME
= appletInstanceName]          WIDTH = pixels          HEIGHT = pixels          [ALIGN = alignment]
[VSPACE = pixels]        [HSPACE = pixels] >

[< PARAM NAME = AttributeName VALUE = AttributeValue>]
[< PARAM NAME = AttributeName2 VALUE = AttributeValue>]
................................................
[HTML Displayed in the absence of Java]

</APPLET>
```

CODEBASE: CODEBASE is an optional attribute that specifies the base URL of the applet code, which is the directory that will be searched for the applet's executable class file (specified by the CODE tag).

CODE: CODE is a required attribute that gives the name of the file containing your applet's compiled .class file.

ALT: The ALT tag is an optional attribute used to specify a short text message that should be displayed if the browser understands the APPLET tag but can't currently run Java applets.

NAME: NAME is an optional attribute used to specify a name for the applet instance. Applets must be named in order for other applets on the same page to find them by name and communicate with them.

WIDTH AND HEIGHT: WIDTH and HEIGHT are required attributes that give the size (in pixels) of the applet display area.

ALIGN: ALIGN is an optional attribute that specifies the alignment of the applet. This attribute is treated the same as the HTML IMG tag with these possible values: LEFT, RIGHT, TOP, BOTTOM, MIDDLE, BASELINE, TEXTTOP, ABSMIDDLE, and ABSBOTTOM.

VSPACE AND HSPACE: These attributes are optional. VSPACE specifies the space, in pixels, above and below the applet. HSPACE specifies the space, in pixels, on each side of the applet. They're treated the same as the IMG tag's VSPACE and HSPACE attributes.

PARAM NAME AND VALUE: The PARAM tag allows you to specify applet specific arguments in an HTML page. Applets access their attributes with the **getParameter ( )** method.

Passing Parameters to Applets

The APPLET tag in HTML allows us to pass parameters to an applet. To retrieve a parameter, the **getParameter( )** method is used. It returns the value of the specified parameter in the form of a **String** object. Thus, for numeric and **Boolean** values, we need to convert their string representations into their internal formats. Here is an example that demonstrates passing parameters:

Program 11.2 **Demonstration of passing parameters to applet.**

Solution:

```
import java.awt.*;
import java.applet.*;
/*
<applet code="ParamDemo" width=300 height=80>
<param name=fontName value=Courier>
<param name=fontSize value=14>
<param name=leading value=2>
<param name=accountEnabled value=true>
</applet>
*/
public class ParamDemo extends Applet
{
        String fontName;
        int fontSize;
```

```
float leading;
boolean active;

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

        param = getParameter("fontSize");
        try
        {
                if(param != null)
                fontSize = Integer.parseInt(param);
                else
                fontSize = 0;
        }
        catch(NumberFormatException e)
         {
                fontSize = -1;
        }

        param = getParameter("leading");
        try
         {
                if(param != null) // if not found
                leading = Float.valueOf(param).floatValue();
                else
                leading = 0;
        }
        catch(NumberFormatException e)
        {
                leading = -1;
        }

        param = getParameter("accountEnabled");
        if(param != null)
        active = Boolean.valueOf(param).booleanValue();
}
```

```
            .
            public void paint(Graphics g)
             {
                        g.drawString("Font name: " + fontName, 0, 10);
                        g.drawString("Font size: " + fontSize, 0, 26);
                        g.drawString("Leading: " + leading, 0, 42);
                        g.drawString("Account Active: " + active, 0, 58);
            }
}
```

AppletContext and showDocument( )

One application of Java is to use active images and animation to provide a graphical means of navigating the Web that is more interesting than the underlined blue words used by hypertext. To allowyour applet to transfer control to another URL, you must use the **showDocument( )** method defined by the **AppletContext** interface.

AppletContext is an interface that lets you get information from the applet's execution environment. The methods defined by AppletContext are shown in Table 19-2. The context of the currently executing applet is obtained by a call to the getAppletContext( ) method defined by Applet. Within an applet, once you have obtained the applet's context, you can bring another document into view by calling showDocument( ). This method has no return value and throws no exception if it fails, so use it carefully. There are two showDocument( ) methods.

- The method showDocument(URL) displays the document at the specified URL.
- The method showDocument(URL, where) displays the specified document at the specified location within the browser window. Valid arguments for where are "_self" (show in current frame), "_parent" (show in parent frame), "_top" (show in topmost frame), and "_blank" (show in new browser window).

Program 11.3 **Demonstration of applet context and showdocument.**

Solution:

```
import java.awt.*;
import java.applet.*;
import java.net.*;
/*
<applet code="ACDemo" swidth=300 height=50>
</applet>
*/
public class ACDemo extends Applet
```

```
{
        public void start()
        {
        AppletContext ac = getAppletContext();
        URL url = getCodeBase(); // get url of this applet
                try
                {
                ac.showDocument(new URL(url+"Test.html"));
                }
                catch(MalformedURLException e)
                {
                showStatus("URL not found");
                }
        }
}
```

Program 11.4 Java Music Player

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
public class PlaySoundApplet extends Applet implements ActionListener
{
         Button play,stop;
        AudioClip audioClip;
        public void init ()
        {
         play = new Button(" Play in Loop ");
         add(play);
         play.addActionListener(this);
         stop = new Button(" Stop ");
         add(stop);
         stop.addActionListener(this);
         audioClip = getAudioClip(getCodeBase(), "TestSnd.wav");
         }
        public void actionPerformed(ActionEvent ae)
        {
        Button source = (Button)ae.getSource();
                 if (source.getLabel() == "  Play in Loop  ")
                {
                 audioClip.play();
                }
                 else if(source.getLabel() == "  Stop  ")
                {
                 audioClip.stop();
                }
        }
}
```

Here is the HTML code :

```
<HTML>
<BODY>
<APPLET CODE="PlaySoundApplet" WIDTH="200" HEIGHT="300"></APPLET>
</BODY>
</HTML>
```

Assignment 11

Short Type Questions

1. Define Applet.
2. Define PARAM tag.

Long Type Questions

1. Discuss the skeleton of an applet.
2. Discuss AppletContext and showDocument( ).
3. How parameters are passed to an applet. Discuss.
4. Discuss in details the structure of an HTML applet tag.
5. Write a java program using applet to design a screen through which you can enter name, address and mobile number of a person. Then display the information in the same window whenever a submit button is pressed.
6. Write a java program using applet to draw a rectangle using mouse.

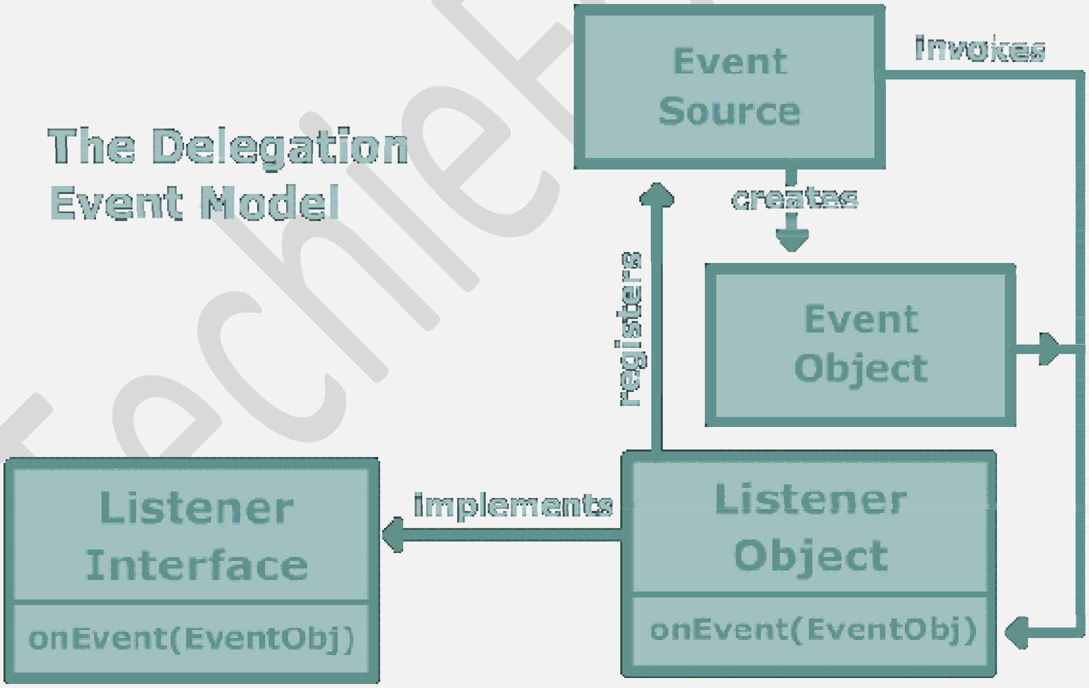
Event Handling

An **Event** is generated whenever the user perform on action such as clicking mouse, pressing a key on keyboard, releasing a key from the keyboard etc.

Event delegation model

The modern approach to handling events is based on the delegation event model, which defines standard and consistent mechanisms to generate and process events. Its concept is quite simple: a source generates an event and sends it to one or more listeners. In this scheme, the listener simply waits until it receives an event. Once received, the listener processes the event and then returns. The advantage of this design is that the application logic that processes events is cleanly separated from theuser interface logic that generates those events. A user interface element is able to "delegate" the processing of an event to a separate piece of code. In the delegation event model, listeners must register with a source in order to receive an event notification. This provides an important benefit: notifications are sent only to listeners that want to receive them.

The **event delegation model** is based on the Event Source and Event Listeners. Event Listener is an object that receives the messages / events. The Event Source is any object which creates the message/ event. The Event Delegation model is based on – The Event Classes, The Event Listeners, Event Objects.



There are three participants in event delegation model in Java:

Events

In the delegation model, an event is an object that describes a state change in a source. It can be generated as a consequence of a person interacting with the elements in a graphical user interface. Some of the activities that cause events to be generated are pressing a button, entering a character via the keyboard, selecting an item in a list, and clicking the mouse. Many other user operations could also be cited as examples. Events may also occur that are not directly caused by interactions with a user interface. For example, an event may be generated when a timer expires, a counter exceeds a value, software or hardware failure occurs, or an operation is completed. You are free to define events that areappropriate for your application.

Event Sources

A source is an object that generates an event. This occurs when the internal state of that object changes in some way. Sources may generate more than one type of event. A source must register listeners in order for the listeners to receive notifications about a specific type of event. Each type of event has its own registration method. Here is the general form:

public void addTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the event listener. For example, the method that registers a keyboard event listener is called **addKeyListener( )**. The method that registers a mouse motion listener is called **addMouseMotionListener( )**. When an event occurs, all registered listeners are notified and receive a copy of the event object. This is known as multicasting the event. In all cases, notifications are sent only to listeners that register to receive them. Some sources may allow only one listener to register. The general form of such a method is this:

public void addTypeListener(TypeListener el) throws java.util.TooManyListenersException

Here, Type is the name of the event and el is a reference to the event listener. When such an event occurs, the registered listener is notified. This is known as unicasting the event. A source must also provide a method that allows a listener to unregister an interest in a specific type of event. The general form of such a method is this:

public void removeTypeListener(TypeListener el)

Here, Type is the name of the event and el is a reference to the event listener. For example, to remove a keyboard listener, you would call **removeKeyListener( )**. The methods that add or remove listeners are provided by the source that generates events. For example, the **Component** class provides methods to add and remove keyboard and mouse event listeners.

Event Listeners

A listener is an object that is notified when an event occurs. It has two major requirements. First, it must have been registered with one or more sources to receive notifications about specific types of events.

Second, it must implement methods to receive and process these notifications. The methods that receive and process events are defined in a set of interfaces found in **java.awt.event**. For example, the **MouseMotionListener** interface defines two methods to receive notifications when the mouse is dragged or moved. Any object may receive and process one or both of these events if it provides an implementation of this interface. Many other listener interfaces are discussed later in this and other chapters.

Event Class

An event occurs (like mouse click, key press, etc) which is followed by the event is broadcasted by the event source by invoking an agreed method on all event listeners. The event object is passed as argument to the agreed-upon method. Later the event listeners respond as they fit, like submit a form, displaying a message / alert etc.

Some methods related to Events are

- **Object getSource()**
  This method is defined in the Event Object class. It returns the object that originated the event. So if the user has clicked a Push button, the method getSource returns the object corresponding to Push button is generated.
- **int getID()**
  This method returns the integer value corresponding to the type of event. For example if the user has clicked on the mouse, the method getID returns the integer defined as MouseEvent.MOUSE_CLICKED.

The package **java.awt.event** defines several types of events that are generated by various user interface elements. Below table enumerates the most important of these event classes and provides a brief description of when they are generated.

Event Type	Event Source	Event Listener interface
ActionEvent	Button, List, MenuItem, TextField	ActionListener
AdjustmentEvent	Scrollbar	AdjustmentListener
ItemEvent	Choice, Checkbox, CheckboxMenuItem, List	ItemListener
TextEvent	TextArea, TextField	TextListener
ComponentEvent	Component	ComponentListener
ContainerEvent	Container	ContainerListener
FocusEvent	Component	FocusListener
KeyEvent	Component	KeyListener
MouseEvent	Component	MouseListener, MouseMotionListener
WindowEvent	Window	WindowListener

Action Event: The GUI components that generate **ActionEvent** are:

- Button: When the user clicks on a PushButton.
- List: When an item in list box is double clicked.
- MenuItem : When a MenuItem is selected.
- Text Field: When the user clicks the Enter key in a text box.

AdjustmentEvent is generated when the user adjusts the position of a scrollbar. Scrollbar is the only GUI control that receives the AdjustmentEvent.

ItemEvent is generated when an item is selected or deselected. Following components generate ItemEvents.
CheckBox: When the state of the CheckBox is changed.
CheckBoxMenuItem : When the state of a MenuItem is changed.
Choice : When the state of a ChoiceBox is changed.
List : When an item in list is selected or deselected.

TextEvent is generated when the contents of text component are changed. The components that generate TextEvent are TextArea and TextField.

FocusEvent: This event is generated when a component gains or looses focus. Focus may be gained by bringing the mouse over a component (or by using the tab key). The component that has the focus receives all user keyboard events. Focus events are generated by objects of Component class and all its subclasses.

KeyEvent and **MouseEvent** are subclasses of abstract InputEvent class. Both these events are generated by objects of type Component class and its subclasses. The KeyEvent is generated when the user presses or releases a key on the keyboard. The MouseEvent is generated when the user presses the mouse or moves the mouse.

WindowEvent are generated for the Window class and its subclasses. These events are generated if an operation is performed on a window. The operation could be closing of window, opening of window, activating a window etc.

PaintEvent is generated when a component needs to be repainted (for example when an application which is in front is closed and the component needs to be redrawn.) PaintEvents are internally handled by AWT, and cannot (and should not) be handled by you in the application.

ComponentEvent are also generated by objects of type Component class and its subclasses. This event is generated when a component is hidden, shown, moved or resized.

ContainerEvents are generated when a component is added or removed from a container. ComponentEvent and ContainerEvent are handled by AWT and are not normally handled by the user.

Event Listener Interfaces and corresponding methods which it defines

Event Listener interface	**Event Listener Methods**
ActionListener	actionPerformed(ActionEvent evt)
AdjustmentListener	adjustmentValueChanged(AjustmentEvent evt)
ItemListener	itemStateChanged(ItemEvent evt)
TextListener	textValueChanged(TextEvent evt)
ComponentListener	componentHidden(ComponentEvent evt), componentMoved(ComponentEvent evt), componentResized(ComponentEvent evt), componentShown(ComponentEvent evt)
ContainerListener	componentAdded(ContainerEvent evt), componentRemoved(ContainerEvent evt)
FocusListener	focusGained(FocusEvent evt), focusLost(FocusEvent evt)
KeyListener	keyPressed(KeyEvent evt), keyReleased(KeyEvent evt), keyTyped(KeyEvent evt)
MouseListener	mouseClicked(MouseEvent evt), mouseEntered(MouseEvent evt), mouseExited(MouseEvent evt), mousePressed(MouseEvent evt), mouseReleased(MouseEvent evt)
MouseMotionListener	mouseDragged(MouseEvent evt), mouseMoved(MouseEvent evt)
WindowListener	windowActivated(WindowEvent evt), windowClosed(WindowEvent evt), windowClosing(WindowEvent evt), windowDeactivated(WindowEvent evt), windowDeiconified(WindowEvent evt), windowIconified(WindowEvent evt), windowOpened(WindowEvent evt)

Adapter classes

Event adapters facilitate implementing listener interfaces. Many event listener interfaces have more than one event listener methods. For such interfaces, Java technology defines adapter classes. These have empty implementation (stubs) of all the event listener methods defined in the interface they implement. A listener can subclass the adapter and override only stub methods for handling events of interest. The table below lists the low level event listener interfaces and their adapters.

Event Listener Interfaces and their corresponding adapter classes

Event Listener Adapter

Event Listener interface	
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

Program 12.1 **Write a program to find addition of two numbers using Frame, TextField, Label, Button, Exception Handling and Event Handling.**

Solution:

```
import java.awt.*;
import java.awt.event.*;
class Addition extends Frame implements ActionListener
{
        String s="Result is";
        Label l1=new Label("Enter Number1");
        Label l2=new Label("Enter Number2");
        Label l3=new Label(s);
        TextField t1=new TextField(5);
        TextField t2=new TextField(5);
        Button b1=new Button("Get Result");

        public Addition(String s)
        {
                 super(s);
                setLayout(new FlowLayout());
                add(l1);
                add(t1);
                add(l2);
                add(t2);
                add(b1);
                add(l3);
                b1.addActionListener(this);
        }

        public void actionPerformed(ActionEvent ae)
```

```
    {
        if(ae.getSource()==b1)
        {
        try
        {
        s="Result is";
        int  n1=Integer.parseInt(t1.getText());
        int n2=Integer.parseInt(t2.getText());
        int r=n1+n2;
        s=s+String.valueOf(r);
        l3.setText(s);
        }
        catch(Exception e)
        {
        l3.setText(“Please enter valid inputs and try again ! !”);
        }
        }
    }
    public static void main(String args[])
    {
        Addition a=new Addition("Addition of two Numbers");
        a.setSize(800, 600);
        a.show();
    }
}
```

Program 12.2 **Demonstration of Key Event: Number Grid Puzzle…..A Java Game.**

Solution:

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

class Num extends Frame implements ActionListener, KeyListener
{
int x, i, j, h, v, move, c;
String msg="Number of Attempts is";
Button b[][]=new Button[4][4];
```

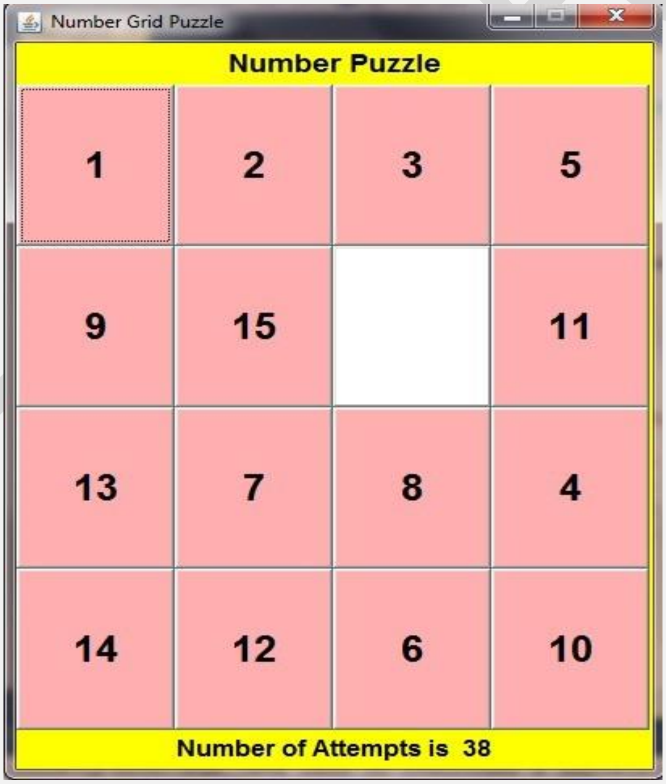
```
Label l1=new Label("Number Puzzle",Label.CENTER);
Label l2=new Label(msg,Label.CENTER);
Font f1 = new Font("SansSerif", Font.BOLD, 18);
Font f2 = new Font("SansSerif", Font.BOLD, 16);
Font f3 = new Font("SansSerif", Font.BOLD, 26);
int[] solutionArray = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,13,14,15 };
Panel p=new Panel();
public Num(String s)
{
super(s);
setLayout(new BorderLayout());
move=0;
shuffleArray(solutionArray);
setBackground(Color.yellow);
l1.setFont(f1);
l2.setFont(f2);
c=0;
for(i=0;i<4;i++)
{
        for(j=0;j<4;j++)
        {
        if(solutionArray[c]==0)
        {
        h=i;v=j;
        b[i][j]=new Button("");
        b[i][j].setBackground(Color.white);
        }
        else
        {
        b[i][j]=new Button(String.valueOf(solutionArray[c]));
        b[i][j].setBackground(Color.pink);
        }
        c++;
        }
}
p.setLayout(new GridLayout(4,4));
for(i=0;i<4;i++)
{
        for(j=0;j<4;j++)
        {
        b[i][j].setFont(f3);
        p.add(b[i][j]);
        }
}
for(i=0;i<4;i++)
for(j=0;j<4;j++)
b[i][j].addKeyListener(this);
add(l1,  BorderLayout.NORTH);
add(p, BorderLayout.CENTER);
add(l2, BorderLayout.SOUTH);
}

static void shuffleArray(int[] ar)
{
  Random rnd = new Random();
  for (int i = ar.length - 1; i >= 0; i--)
  {
  int index = rnd.nextInt(i + 1);
  int a = ar[index];
  ar[index] = ar[i];
  ar[i] = a;
  }
}

boolean checkWinner()
{
        int w=1;
        for(i=0;i<4;i++)
        {
                for(j=0;j<4;j++)
                {
                if(b[i][j].getLabel()!="")
                {
                if(Integer.parseInt(b[i][j].getLabel())!=w)
                break;
                else
                w++;
                }
                }
        }
```

```
        if(w==15)
        return(true);
        else
        return(false);
}

public void actionPerformed(ActionEvent e)
{

}
public void keyPressed(KeyEvent e)
{
        if(e.getKeyCode( )==KeyEvent.VK_LEFT)
        {
                if(v<3)
                {
                b[h][v].setLabel(b[h][v+1].getLabel());
                b[h][v+1].setLabel("");
                b[h][v+1].setBackground(Color.white);
                b[h][v].setBackground(Color.pink);
                v++;
                move++;
                if(checkWinner())
                l2.setText("Congrats !!! Number of attempt= "+move);
                else
                l2.setText(msg+"  "+move);
                }
                else
                l2.setText("Invalid Move !!!");
        }

        if(e.getKeyCode( )==KeyEvent.VK_RIGHT)
        {
                if(v>0)
                {
                b[h][v].setLabel(b[h][v-1].getLabel());
                b[h][v-1].setLabel("");
                b[h][v-1].setBackground(Color.white);
                b[h][v].setBackground(Color.pink);
                v--;
```

```
    move++;
    if(checkWinner())
    l2.setText("Congrats !!! Number of attempt= "+move);
    else
    l2.setText(msg+"  "+move);
    }
    else
    l2.setText("Invalid Move !!!");
}

if(e.getKeyCode( )==KeyEvent.VK_DOWN)
{
    if(h>0)
    {
    b[h][v].setLabel(b[h-1][v].getLabel());
    b[h-1][v].setLabel("");
    b[h-1][v].setBackground(Color.white);
    b[h][v].setBackground(Color.pink);
    h--;
    move++;
    if(checkWinner())
    l2.setText("Congrats !!! Number of attempt= "+move);
    else
    l2.setText(msg+"  "+move);
    }
    else
    l2.setText("Invalid Move !!!");
}
if(e.getKeyCode( )==KeyEvent.VK_UP)
{
    if(h<3)
    {
    b[h][v].setLabel(b[h+1][v].getLabel());
    b[h+1][v].setLabel("");
    b[h+1][v].setBackground(Color.white);
    b[h][v].setBackground(Color.pink);
    h++;
    move++;
    if(checkWinner())
    l2.setText("Congrats !!! Number of attempt= "+move);
```

```
                else
                l2.setText(msg+"  "+move);
                }
                else
                l2.setText("Invalid Move !!!");
        }
}
public void keyReleased(KeyEvent e)
{
}
public void keyTyped(KeyEvent e)
{
}
public static void main(String args[])
{
Num c=new Num("Number Grid Puzzle");
c.setSize(400,550);
c.show();
c.setResizable(false);
}
}
```

Output:



Number Grid Puzzle

Number Puzzle

1	2	3	5
9	15		11
13	7	8	4
14	12	6	10

Number of Attempts is 38

Assignment 12

Abstract Windows Toolkit (AWT)

- AWT is the set of java classes that allows us to create GUI (Graphical User Interfaces) component and manipulate them.
- GUI Components are used to take the input from user in a user friendly manner.

Container

The **Container** class is a subclass of **Component**. It has additional methods that allow other **Component** objects to be nested within it. Other **Container** objects can be stored inside of a **Container** (since they are themselves instances of **Component**). This makes for a multileveled containment system. A container is responsible for laying out (that is, positioning) any components that it contains.

Panel

The **Panel** class is a concrete subclass of **Container**. It doesn't add any new methods; it simply implements **Container**. A **Panel** may be thought of as a recursively nestable, concrete screen component. **Panel** is the superclass for **Applet**. When screen output is directed to an applet, it is drawn on the surface of a **Panel** object. In essence, a **Panel** is a window that does not contain a title bar, menu bar, or border. This is why you don't see these items when an applet is run inside a browser. When you run an applet using an applet viewer, the applet viewer provides the title and border. Other components can be added to a **Panel** object by its **add( )** method (inherited from **Container**). Once these components have been added, you can position and resize them manually using the **setLocation( )**, **setSize( )**, or **setBounds( )** methods defined by **Component**.

Window

The **Window** class creates a top-level window. A top-level window is not contained within any other object; it sits directly on the desktop. Generally, you won't create **Window** objects directly. Instead, you will use a subclass of **Window** called **Frame**, described next.

Frame

Frame encapsulates what is commonly thought of as a "window." It is a subclass of **Window** and has a title bar, menu bar, borders, and resizing corners. If you create a **Frame** object from within an applet, it will contain a warning message, such as "Java Applet Window," to the user that an applet window has been created. This message warns users that the window they see was started by an applet and not by software running on their computer. (An applet that could masquerade as a host-based application could be used to obtain passwords and other sensitive information without the user's knowledge.) When a **Frame** window is created by a program rather than an applet, a normal window is created.

Canvas

Although it is not part of the hierarchy for applet or frame windows, there is one other type of window that you will find valuable: **Canvas**. **Canvas** encapsulates a blank window upon which you can draw. You will see an example of **Canvas** later in this book.

Here are two of **Frame**'s constructors:
Frame( )
Frame(String title)
The first form creates a standard window that does not contain a title. The second form creates a window with the title specified by title.

There are several methods you will use when working with **Frame** windows.
The **setSize( )** method is used to set the dimensions of the window. Its signature is shown here:
void setSize(int newWidth, int newHeight)

After a frame window has been created, it will not be visible until you call **setVisible( )**. Its signature is shown here:
void setVisible(boolean visibleFlag)
The component is visible if the argument to this method is **true**. Otherwise, it is hidden.

You can change the title in a frame window using **setTitle( )**, which has this general form:
void setTitle(String newTitle)
Here, newTitle is the new title for the window.

Control Fundamentals

The AWT supports the following types of controls:

1. Label

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
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.

We can set or change the text in a label by using the **setText( )** method.We can obtain the current label by calling **getText( )**. These methods are:

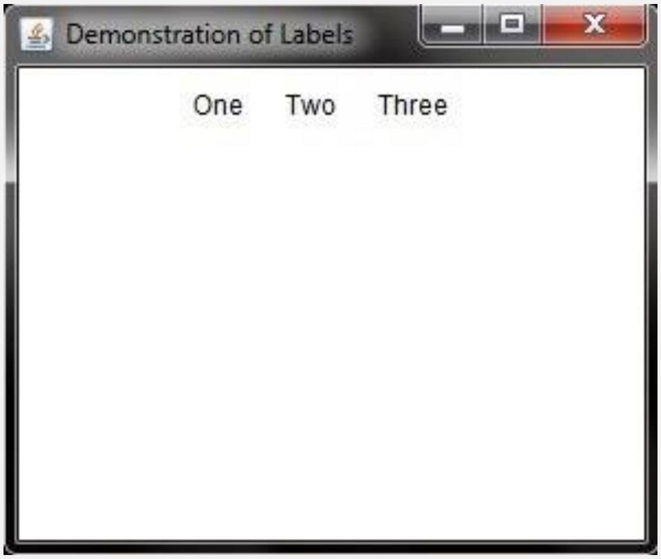
```
 void setText(String str)
String getText( )
```

Program 13.1 **Demonstration of Label.**

Solution:

```
import java.awt.*;
class LabelDemo extends Frame
 {
        Label one = new Label("One");
        Label two = new Label("Two");
        Label three = new Label("Three");
        public LabelDemo(String s)
         {
        super(s);
        setLayout(new FlowLayout());
        add(one);
        add(two);s
        add(three);
        }

        public static void main(String args[])
        {
        LabelDemo l=new LabelDemo("Demonstration of Labels");
        l.setSize(300, 250);
        l.setVisible(true);
        }
}
```

Output:



2. Button

A 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

After a button has been created, we can set its label by calling **setLabel( )**. We can retrieve its label by calling **getLabel( )**. These methods are as follows:

void setLabel(String str)

String getLabel( )

Here, str becomes the new label for the button.

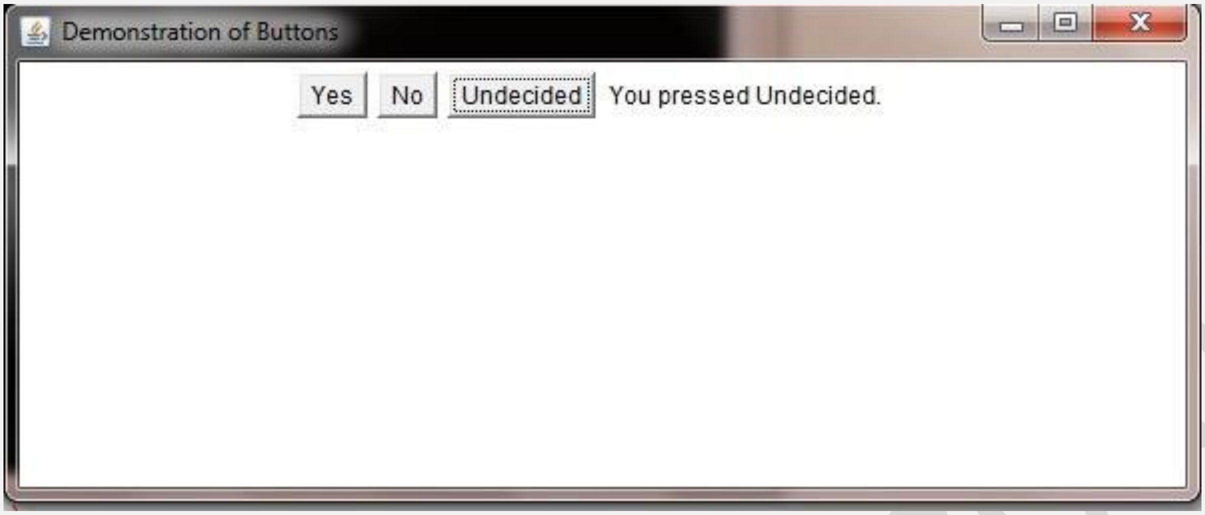
Program 13.2 **Demonstration of Button.**

Solution:

```
import java.awt.*;
import java.awt.event.*;
class ButtonDemo extends Frame implements ActionListener
{
        String msg = " ";
        Button yes= new Button("Yes");
        Button no= new Button("No");
        Button maybe= new Button("Undecided");
        Label l=new Label(msg);
        public ButtonDemo(String s)
```

```
         {
        super(s);
        setLayout(new FlowLayout());
        add(yes);
        add(no);
        add(maybe);add(l);
        yes.addActionListener(this);
        no.addActionListener(this);
        maybe.addActionListener(this);
        }
        public void actionPerformed(ActionEvent ae)
         {
        String str = ae.getActionCommand();
        if(str.equals("Yes"))
        {
        msg = "You pressed Yes.";
        }
        else if(str.equals("No"))
        {
        msg = "You pressed No.";
        }
        else
        {
        msg = "You pressed Undecided.";
        }
        l.setText(msg);
        }
        public static void main(String args[])
        {
        ButtonDemo b=new ButtonDemo("Demonstration of Buttons");
        b.setSize(600, 250);
        b.setVisible(true);
        }
}
```

Output:



You pressed Undecided.

3. Checkbox

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. 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)	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**.
Checkbox(String str, CheckboxGroup cbGroup, boolean on)	

To retrieve the current state of a check box, call getState( ). To set its state, call setState( ). We can obtain the current label associated with a check box by calling getLabel( ). To set the label, call setLabel( ). These methods are as follows:

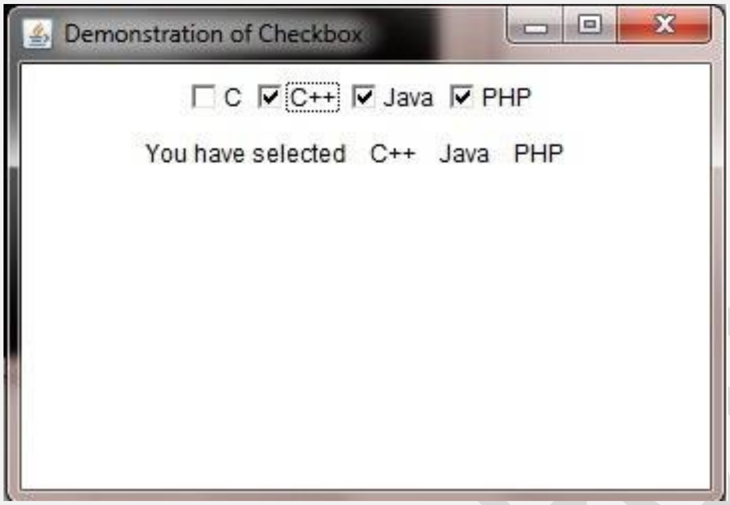
boolean getState( )
void setState(boolean on)
String getLabel( )
void setLabel(String str)

Program 13.3 Demonstration of Checkbox.

Solution:

```
import java.awt.*;
import java.awt.event.*;
class CheckboxDemo extends Frame implements ItemListener
 {
        String msg = " ";
        Checkbox c1 = new Checkbox("C", null, true);
        Checkbox c2 = new Checkbox("C++");
        Checkbox c3 = new Checkbox("Java");
        Checkbox c4 = new Checkbox("PHP");
        Label l=new Label(msg);
        public CheckboxDemo(String s)
         {
        super(s);
        setLayout(new FlowLayout());
        add(c1);
        add(c2);
        add(c3);
        add(c4);
        add(l);
        c1.addItemListener(this);
        c2.addItemListener(this);
        c3.addItemListener(this);
        c4.addItemListener(this);
        }
        public void itemStateChanged(ItemEvent ie)
        {
        msg="You have selected";
        if(c1.getState())
        msg=msg+" C";
        if(c2.getState())
        msg=msg+" C++";
        if(c3.getState())
        msg=msg+" Java";
        if(c4.getState())
        msg=msg+" PHP";
        l.setText(msg);
        }
        public static void main(String args[])
        {
        CheckboxDemo b=new CheckboxDemo ("Demonstration of Checkbox");
        b.setSize(350, 250);
        b.setVisible(true);
        }
}
```

Output:



```
You have selected   C++   Java   PHP
```

4. Radio Buttons/ Checkbox Group

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, because they act like the station selector on a car radio—only one station can be selected at any one time. We can determine which check box in a group is currently selected by calling **getSelectedCheckbox( )**. We can set a check box by calling **setSelectedCheckbox( )**.

These methods are as follows:

Checkbox getSelectedCheckbox( )

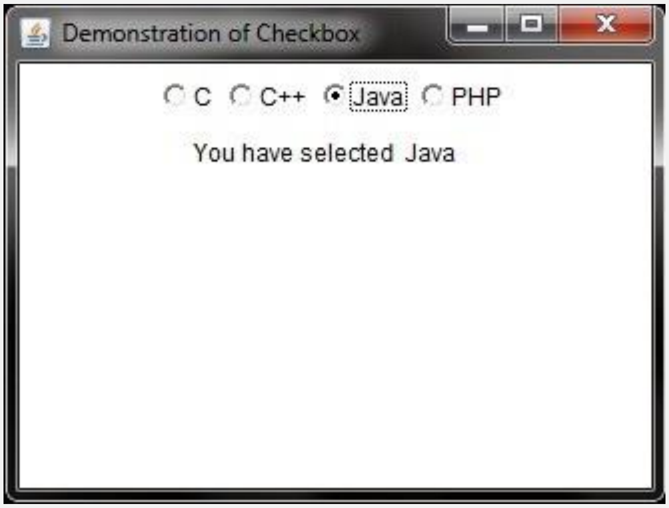
void setSelectedCheckbox(Checkbox which)

Program 13.4 **Demonstration of Radio Buttons.**

Solution:

```
import java.awt.*;
import java.awt.event.*;
class RadioDemo extends Frame implements ItemListener
 {
    String msg = " ";

    CheckboxGroup cbg=new CheckboxGroup();
    Checkbox c1 = new Checkbox("C", cbg, true);
    Checkbox c2 = new Checkbox("C++",cbg,false);
    Checkbox c3 = new Checkbox("Java",cbg,false);
    Checkbox c4 = new Checkbox("PHP",cbg,false);
    Label l=new Label(msg);
    public RadioDemo(String s)
     {
    super(s);
    setLayout(new FlowLayout());
    add(c1);
    add(c2);
    add(c3);
    add(c4);
    add(l);
    c1.addItemListener(this);
    c2.addItemListener(this);
    c3.addItemListener(this);
    c4.addItemListener(this);
    }
    public void itemStateChanged(ItemEvent ie)
    {
    msg="You have selected "+cbg.getSelectedCheckbox().getLabel();
    l.setText(msg);
    }
    public static void main(String args[])
    {
    RadioDemo b=new RadioDemo ("Demonstration of Checkbox");
    b.setSize(350, 250);
    b.setVisible(true);
    }
}
```

Output:



○ C ○ C++ ● Java ○ PHP

You have selected Java

5. Choice

The Choice class is used to create a pop-up list of items from which the user may choose. Thus, a Choice control is a form of menu. Each item in the list is a string that appears as a left-justified label in the order it is added to the Choice object. Choice only defines the default constructor, which creates an empty list. To add a selection to the list, call add( ). It has this general form:

void add(String name)

Here, name is the name of the item being added. Items are added to the list in the order in which calls to add( ) occur.

To determine which item is currently selected, we may call either getSelectedItem( ) or getSelectedIndex( ). These methods are shown here:

String getSelectedItem( )
int getSelectedIndex( )

The getSelectedItem( ) method returns a string containing the name of the item. getSelectedIndex( ) returns the index of the item. The first item is at index 0. By default, the first item added to the list is selected. To obtain the number of items in the list, call getItemCount( ). We can set the currently selected item using the select( ) method with either a zero-based integer index or a string that will match a name in the list. These methods are shown here:

int getItemCount( )
void select(int index)
void select(String name)

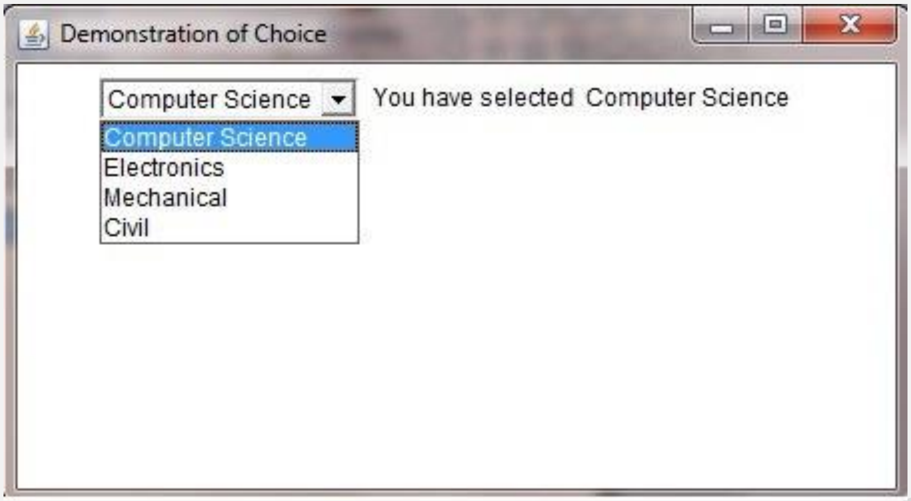
Program 13.5 **Demonstration of Choice.**

Solution:

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

```
class ChoiceDemo extends Frame implements ItemListener
 {
            String msg = " ";
            Choice dept=new Choice();
            Label l=new Label(msg);

            public ChoiceDemo(String s)
             {
            super(s);
            setLayout(new FlowLayout());
            dept.add("Computer Science");
            dept.add("Electronics");
            dept.add("Mechanical");
            dept.add("Civil");
            add(dept);
            add(l);
            dept.addItemListener(this);
            }

            public void itemStateChanged(ItemEvent ie)
            {
            msg="You have selected "+dept.getSelectedItem();
            l.setText(msg);
            }

            public static void main(String args[])
            {
            ChoiceDemo b=new ChoiceDemo("Demonstration of Choice");
            b.setSize(450, 250);
            b.setVisible(true);
            }
}
```

Output:



6. List

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, a **List** object can be constructed to show any number of choices in the visible window. It can also be created to allow multiple selections. **List** provides these constructors:

List( )	creates a **List** control that allows only one item to be selected at any one time
List(int numRows)	the value of numRows specifies the number of entries in the list that will always be visible (others can be scrolled into view as needed)
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.

To add a selection to the list, call **add( )**. It has the following two forms:
void add(String name)
void add(String name, int index)

Here, name is the name of the item added to the list. The first form adds items to the end of the list. The second form 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. For lists that allow only single selection, you can determine which item is currently selected by calling either **getSelectedItem( )** or **getSelectedIndex( )**. These methods are shown here:
String getSelectedItem( )
int getSelectedIndex( )

The **getSelectedItem( )** method returns a string containing the name of the item. If more than one item is selected or if no selection has yet been made, **null** is returned. **getSelectedIndex( )** returns the index of the item. The first item is at index 0. If more than one item is selected, or if no selection has yet been made, –1 is returned.

For lists that allow multiple selection, you must use either **getSelectedItems( )** or **getSelectedIndexes ( )**, shown here, to determine the current selections:

String[ ] getSelectedItems( )

int[ ] getSelectedIndexes( )

getSelectedItems( ) returns an array containing the names of the currently selected items. **getSelectedIndexes( )** returns an array containing the indexes of the currently selected items.

To obtain the number of items in the list, call **getItemCount( )**. You can set the currently selected item by using the **select( )** method with a zero-based integer index. These methods are shown here:

int getItemCount( )

void select(int index)

Given an index, you can obtain the name associated with the item at that index by calling **getItem( )**, which has this general form:

String getItem(int index)

Here, index specifies the index of the desired item.

Program 13.6 **Demonstration of List box.**

Solution:

```
import java.awt.*;
import java.awt.event.*;
class ListDemo extends Frame implements ItemListener
 {
            String msg = " ";
            List dept=new List(4, true);
            Label l=new Label(msg);

            public ListDemo(String s)
             {
            super(s);
            setLayout(new FlowLayout());
            dept.add("Computer Science");
            dept.add("Electronics");
            dept.add("Mechanical");
            dept.add("Civil");
            add(dept);
            add(l);
            dept.addItemListener(this);
            }
```

```
        public void itemStateChanged(ItemEvent ie)
        {
        int idx[];
        msg="You have selected  ";
        idx = dept.getSelectedIndexes();
        for(int i=0; i<idx.length; i++)
        msg += dept.getItem(idx[i]) + " ";
        l.setText(msg);}

        public static void main(String args[])
        {
        ListDemo b=new ListDemo("Demonstration of List");
        b.setSize(450, 250);
        b.setVisible(true);
        }
}
```

7. TextField

The **TextField** class implements a single-line text-entry area, usually called an edit control. Text fields allow the user to enter strings and to edit the text using the arrow keys, cut and paste keys, and mouse selections. **TextField** is a subclass of **TextComponent**. **TextField** defines the following constructors:

TextField( )	creates a default text field
TextField(int numChars)	creates a text field that is numChars characters wide
TextField(String str)	initializes the text field with the string contained in str
TextField(String str, int numChars)	initializes a text field and sets its width.

TextField (and its superclass TextComponent) provides several methods that allow you to utilize a text field. To obtain the string currently contained in the text field, call getText( ). To set the text, call setText( ). These methods are as follows:

String getText( )
void setText(String str)s
Here, str is the new string.

The user can select a portion of the text in a text field. Also, you can select a portion of text under program control by using select( ). Your program can obtain the currently selected text by calling getSelectedText( ). These methods are shown here:

String getSelectedText( )
void select(int startIndex, int endIndex)

getSelectedText( ) returns the selected text. The select( ) method selects the characters beginning at startIndex and ending at endIndex–1. You can control whether the contents of a text field may be modified by the user by calling setEditable( ). You can determine editability by calling isEditable( ). These methods are shown here:

boolean isEditable( )
void setEditable(boolean canEdit)

isEditable( ) returns true if the text may be changed and false if not. In setEditable( ), if canEdit is true, the text may be changed. If it is false, the text cannot be altered. There may be times when you will want the user to enter text that is not displayed, such as a password. You can disable the echoing of the characters as they are typed by calling setEchoChar( ). This method specifies a single character that the TextField will display when characters are entered (thus, the actual characters typed will not be shown). You can check a text field to see if it is in this mode with the echoCharIsSet( ) method. You can retrieve the echo character by calling the getEchoChar( ) method.

These methods are as follows:

void setEchoChar(char ch)
boolean echoCharIsSet( )
char getEchoChar( )

Here, ch specifies the character to be echoed.

8. TextArea

Sometimes a single line of text input is not enough for a given task. To handle these situations, the 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)

Layout Managers

Each **Container** object has a layout manager associated with it. A layout manager is an instance of any class that implements the **LayoutManager** interface. The layout manager is set by the **setLayout ( )** method. If no call to **setLayout ( )** is made, then the default layout manager is used. Whenever a

container is resized (or sized for the first time), the layout manager is used to position each of the components within it.
The **setLayout( )** method has the following general form:
void setLayout(LayoutManager layoutObj)
Java has several predefined **LayoutManager** classes, several of which are: FlowLayout, BorderLayout, GridLayout.

FlowLayout

FlowLayout is the default layout manager. This is the layout manager that the preceding examples have used. **FlowLayout** implements a simple layout style, which is similar to how words flow in a text editor. Components are laid out from the upper-left corner, left to right and top to bottom. When no more components fit on a line, the next one appears on the next line. A small space is left between each component, above and below, as well as left and right. Here are the constructors for **FlowLayout**:

FlowLayout( )	creates the default layout, which centers components and leaves five pixels of space between each component
FlowLayout(int how)	specify how each line is aligned. Valid values for how are as follows: FlowLayout.LEFT FlowLayout.CENTER FlowLayout.RIGHTs
FlowLayout(int how, int horz, int vert)	specify the horizontal and vertical space left between components in horz and vert, respectively.

BorderLayout

The BorderLayout class implements a common layout style for top-level windows. 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( )	creates a default border layout
BorderLayout(int horz, int vert)	specify the horizontal and vertical space left between components in horz and vert, respectively

BorderLayout defines the following constants that specify the regions:

BorderLayout.CENTER

BorderLayout.SOUTH

BorderLayout.EAST

BorderLayout.WEST

BorderLayout.NORTH

When adding components, you will use these constants with the following form of add( ), which is defined by Container:

void add(Component compObj, Object region);

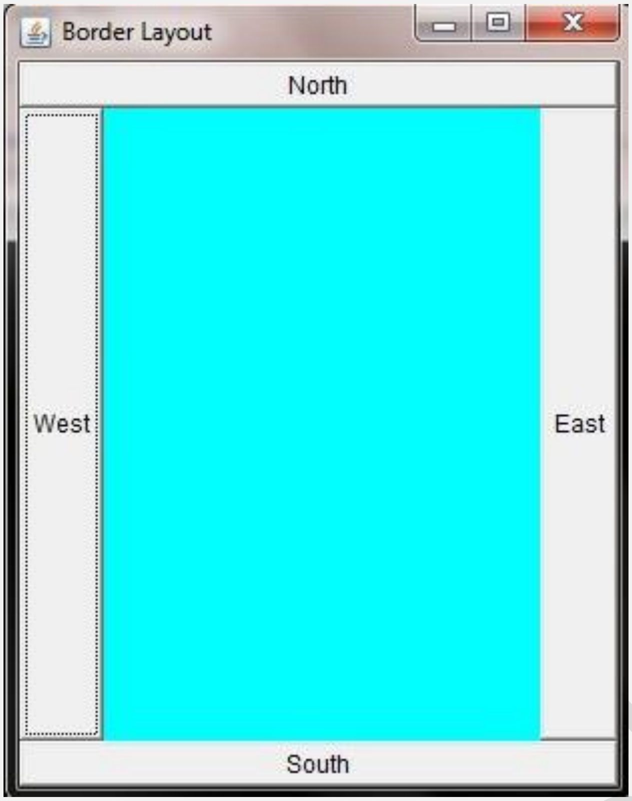
Here, compObj is the component to be added, and region specifies where the component will be added.

Program 13.7 **Demonstration of Border Layout.**

Solution:

```
import java.awt.*;
class Border extends Frame
{
        Button b1=new Button("East");
        Button b2=new Button("West");
        Button b3=new Button("North");
        Button b4=new Button("South");
        public Border(String s1)
        {
        super(s1);
        setBackground(Color.cyan);
        setLayout(new BorderLayout());
        add(b1, BorderLayout.EAST);
        add(b2, BorderLayout.WEST);
        add(b3, BorderLayout.NORTH);
        add(b4, BorderLayout.SOUTH);
        }
        public static void main(String args[])
        {
        Border c=new Border("Border Layout");
        c.setSize(300,400);
        c.show();
        }
}
```

Output:



GridLayout

GridLayout lays out components in a two-dimensional grid. When you instantiate a GridLayout, you define the number of rows and columns. The constructors supported by GridLayout are shown here:

GridLayout( )	creates a single-column grid layout
GridLayout(int numRows, int numColumns)	creates a grid layout with the specified number of rows and columns
GridLayout(int numRows, int numColumns, int horz, int vert)	specify the horizontal and vertical space left between components in horz and vert, respectively.

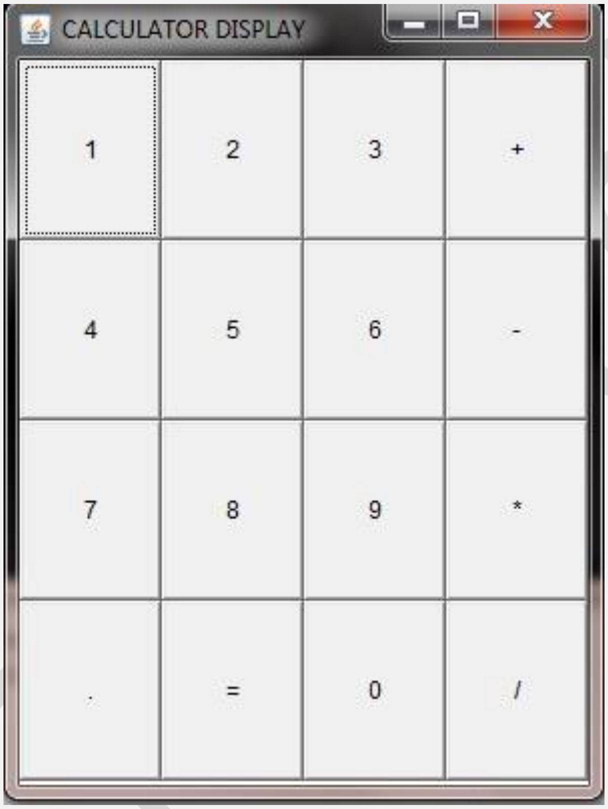
Program 13.8 **Demonstartion of GridLayout.**

Solution:

```
import java.awt.*;
class Cal extends Frame
{
	String s=new String("123+456-789*.=0/");
	int i;
	Button b[]=new Button[16];
	public Cal(String s1)
```

```
        {
        super(s1);
        for(i=0;i<16;i++)
        b[i]=new Button(String.valueOf(s.charAt(i)));
        setLayout(new GridLayout(4,4));
        for(i=0;i<16;i++)
        add(b[i]);
        }
        public static void main(String args[])
        {
        Cal c=new Cal("CALCULATOR DISPLAY");
        c.setSize(300,400);
        c.show();
        }
}
```

Output:



CardLayout

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 be useful for user interfaces with optional components that can be dynamically enabled and disabled upon user input. You can prepare the other

layouts and have them hidden, ready to be activated when needed. CardLayout provides these two constructors:
CardLayout( )
CardLayout(int horz, int vert)
The first form creates a default card layout. The second form allows you to specify the horizontal and vertical space left between components in horz and vert, respectively.

Core java API package

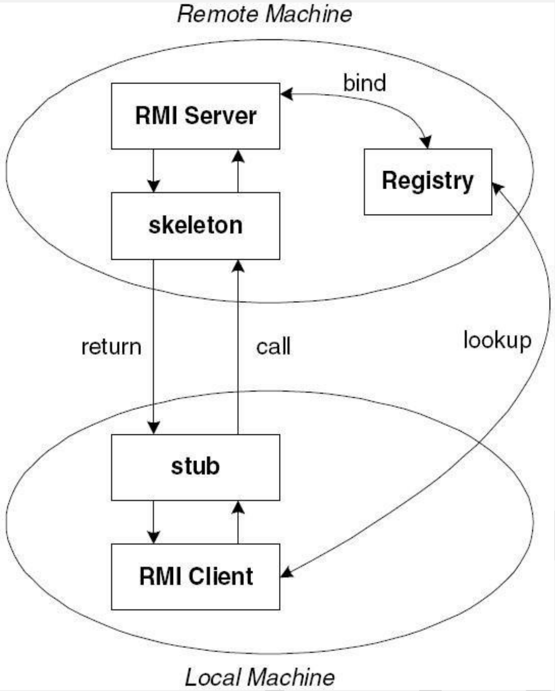
When Java 1.0 was released, it included a set of eight packages, called the core API. These are the packages described in the preceding chapters and are the ones that you will use most often in your day-to-day programming. Today, the Java API contains a large number of packages. For example **java.nio**, **java.util.regex**, **java.lang.reflect**, **java.rmi**, and **java.text** supports support the new I/O system, regular expression processing, reflection, Remote Method Invocation (RMI), and text formatting, respectively.

JAVA Reflection

- **Reflection** is the process by which a computer program can observe (do type introspection) and modify its own structure and behavior at runtime.
- Reflection can be used for observing and/or modifying program execution at runtime. A reflection-oriented program component can monitor the execution of an enclosure of code and can modify itself according to a desired goal related to that enclosure. This is typically accomplished by dynamically assigning program code at runtime.
- In object oriented programming languages such as Java, reflection allows inspection of classes, interfaces, fields and methods at runtime without knowing the names of the interfaces, fields, methods at compile time. It also allows instantiation of new objects and invocation of methods.

Remote Method Invocation (RMI)

Remote Method Invocation (RMI) allows a Java object that executes on one machine to invoke a method of a Java object that executes on another machine. This is an important feature, because it allows us to build distributed applications.



Simple Client/Server Application Using RMI

This section provides step-by-step directions for building a simple client/server application by using RMI. The server receives a request from a client, processes it, and returns a result. In this example, the request specifies two numbers. The server adds these together and returns the sum.

Step One: Enter and Compile the Source Code

The first file, **AddServerIntf.java**, defines the remote interface that is provided by the server. It contains one method that accepts two **double** arguments and returns their sum. All remote interfaces must extend the **Remote** interface, which is part of **java.rmi**. **Remote** defines no members.

```
import java.rmi.*;
public interface AddServerIntf extends Remote
 {
double add(double d1, double d2) throws RemoteException;
}
```

The second source file, **AddServerImpl.java**, implements the remote interface. The implementation of the **add( )** method is straightforward. All remote objects must extend **UnicastRemoteObject**, which provides functionality that is needed to make objects available from remote machines.

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

```
public class AddServerImpl extends UnicastRemoteObject implements AddServerIntf
 {
        public AddServerImpl() throws RemoteException
        {
        }
        public double add(double d1, double d2) throws RemoteException
         {
        return d1 + d2;
        }
}
```

The third source file, **AddServer.java**, contains the main program for the server machine. Its primary function is to update the RMI registry on that machine. This is done by using the **rebind( )** method ofthe **Naming** class (found in **java.rmi**). That method associates a name with an object reference. Thefirst argument to the **rebind( )** method is a string that names the server as “AddServer”. Its second argument is a reference to an instance of **AddServerImpl**.

```
import java.net.*;
import java.rmi.*;
public class AddServer
 {
        public static void main(String args[])
        {
                try
                {
                AddServerImpl addServerImpl = new AddServerImpl();
                Naming.rebind("AddServer", addServerImpl);
                }
                catch(Exception e)
                {
                System.out.println("Exception: " + e);
                }
        }
}
```

The fourth source file, **AddClient.java**, implements the client side of this distributed application. **AddClient.java** requires three command line arguments. The first is the IP address or name of the server machine. The second and third arguments are the two numbers that are to be summed.

```
import      java.rmi.*;
public class AddClient
```

```
{
        public static void main(String args[])
         {
                try
                 {
                String addServerURL = "rmi://" + args[0] + "/AddServer";
                AddServerIntf addServerIntf =
                (AddServerIntf)Naming.lookup(addServerURL);
                System.out.println("The first number is: " + args[1]);
                double d1 = Double.valueOf(args[1]).doubleValue();
                System.out.println("The second number is: " + args[2]);
                double d2 = Double.valueOf(args[2]).doubleValue();
                System.out.println("The sum is: " + addServerIntf.add(d1, d2));
                }
                catch(Exception e)
                {
                System.out.println("Exception: " + e);
                }
        }
}
```

Step Two: Generate Stubs and Skeletons

In the context of RMI, a stub is a Java object that resides on the client machine. Its function is to present the same interfaces as the remote server. Remote method calls initiated by the client are actually directed to the stub.

A skeleton is a Java object that resides on the server machine. Its purpose in RMI system is to receive requests, perform de-serialization, and invoke the appropriate code on the server.

To generate stubs and skeletons, we use a tool called the RMI compiler, which is invoked from the command line, as shown here:

```
rmic AddServerImpl
```

This command generates two new files: **AddServerImpl_Skel.class** (skeleton) and **AddServerImpl_Stub.class** (stub).

Step Three: Install Files on the Client and Server Machines

Copy **AddClient.class**, **AddServerImpl_Stub.class**, and **AddServerIntf.class** to a directory on the client machine. Copy **AddServerIntf.class**, **AddServerImpl.class**, **AddServerImpl_Skel.class**, **AddServerImpl_Stub.class**, and **AddServer.class** to a directory on the server machine.

Step Four: Start the RMI Registry on the Server Machine

The Java 2 SDK provides a program called **rmiregistry**, which executes on the server machine. It maps names to object references. First, check that the **CLASSPATH** environment variable includes the directory in which your files are located. Then, start the RMI Registry from the command line, as shown here: start rmiregistry

Step Five: Start the Server

The server code is started from the command line, as shown here:

```
java AddServer
```

Recall that the **AddServer** code instantiates **AddServerImpl** and registers that object with the name "AddServer".

Step Six: Start the Client

The **AddClient** software requires three arguments: the name or IP address of the server machine and the two numbers that are to be summed together. You may invoke it from the command line by using one of the two formats shown here:

```
java AddClient server1 8 9
java AddClient 11.12.13.14 8 9
```

In the first line, the name of the server is provided. The second line uses its IP address (11.12.13.14).

Output from this program is shown here:

```
The first number is: 8
The second number is: 9
The sum is: 17.0
```

Assignment 13

Short Type Questions

1. Define AWT.
2. List at least 5 AWT controls.
3. Define Textfield. How is it created?
4. Define RMI
5. Define Java Reflection.
6. Define Layout Manager.
8. Differentiate between checkbox and radio button.
9. Distinguish between choice and list box.
10. Which method of TextField class is used to create a password field?

Long Type Questions

1. Discuss RMI (Remote Method Invocation) with neat diagram.
2. What are layout managers? Discuss different layout managers available in java.
3. Define check box. Discuss with an example how to handle events in a checkbox.
4. Write a java program to design a data enter from to enter name, branch, address of the students using text fields, buttons and scroll bar respectively.
5. Differentiate between AWT and Swing based GUI.

Swing

- Swing, which is an extension library to the AWT, includes new and improved components that enhance the look and functionality of GUIs.
- Swing can be used to build Standalone swing GUI Applications as well as Servlets and Applets.
- It employs model/view design architecture. Swing is more portable and more flexible than AWT.

JApplet

Fundamental to Swing is the **JApplet** class, which extends **Applet**. Applets that use Swing must be subclasses of **JApplet**. **JApplet** is rich with functionality that is not found in **Applet**. For example, **JApplet** supports various "panes," such as the content pane, the glass pane, and the root pane. For the examples in this chapter, we will not be using most of **JApplet**'s enhanced features. However, one difference between **Applet** and **JApplet** is important to this discussion, because it is used by the sampleapplets in this chapter. When adding a component to an instance of **JApplet**, do not invoke the **add( )** method of the applet. Instead, call **add( )** for the content pane of the **JApplet** object. The content pane can be obtained via the method shown here:

Container getContentPane( )

The **add( )** method of **Container** can be used to add a component to a content pane.

Its form is shown here:

void add(comp)

Here, comp is the component to be added to the content pane.

Icons and Labels

In Swing, icons are encapsulated by the **ImageIcon** class, which paints an icon from an image. Two of its constructors are shown here:

ImageIcon(String filename)

ImageIcon(URL url)

The first form uses the image in the file named filename. The second form uses the image in the resource identified by url.

The **ImageIcon** class implements the **Icon** interface that declares the methods shown here:

Method	**Description**
int getIconHeight( )	Returns the height of the icon in pixels.
int getIconWidth( )	Returns the width of the icon in pixels.
void paintIcon(Component comp, Graphics g, int x, int y)	Paints the icon at position x, y on the graphics context g. Additional information about the paint operation can be provided in comp.

Swing labels are instances of the JLabel class, which extends JComponent. It can display text and/or an icon. Some of its constructors are shown here:
JLabel(Icon i)
Label(String s)
JLabel(String s, Icon i, int align)
Here, s and i are the text and icon used for the label. The align argument is either LEFT, RIGHT, CENTER, LEADING, or TRAILING. These constants are defined in the SwingConstants interface, along with several others used by the Swing classes.
The icon and text associated with the label can be read and written by the following methods:
Icon getIcon( )
String getText( )
void setIcon(Icon i)
void setText(String s)
Here, i and s are the icon and text, respectively.

TextField

The Swing text field is encapsulated by the **JTextComponent** class, which extends **JComponent**. It provides functionality that is common to Swing text components. One of its subclasses is **JTextField**, which allows you to edit one line of text. Some of its constructors are shown here:

JTextField( )
JTextField(int cols)
JTextField(String s, int cols)
JTextField(String s)

Here, s is the string to be presented, and cols is the number of columns in the text field.

Buttons

- Swing buttons provide features that are not found in the **Button** class defined by the AWT. For example, you can associate an icon with a Swing button.
- Swing buttons are subclasses of the **AbstractButton** class, which extends **JComponent**. **AbstractButton** contains many methods that allow you to control the behavior of buttons, check boxes, and radio buttons.
- For example, you can define different icons that are displayed for the component when it is disabled, pressed, or selected. Another icon can be used as a rollover icon, which is displayed when the mouse is positioned over that component.
- The following are the methods that control this behavior:

  void setDisabledIcon(Icon di)
  void setPressedIcon(Icon pi)
  void setSelectedIcon(Icon si)
  void setRolloverIcon(Icon ri)

The JButton Class

- The **JButton** class provides the functionality of a push button. **JButton** allows an icon, a string, or both to be associated with the push button. Some of its constructors are shown here:
  JButton(Icon i)
  JButton(String s)
  JButton(String s, Icon i)
  Here, s and i are the string and icon used for the button.

Combo Boxes

- Swing provides a combo box (a combination of a text field and a drop-down list) through the **JComboBox** class, which extends **JComponent**.
- A combo box normally displays one entry. However, it can also display a drop-down list that allows a user to select a different entry.
- You can also type your selection into the text field. Two of **JComboBox**'s constructors are shown here:
  JComboBox( )
  JComboBox(Vector v)

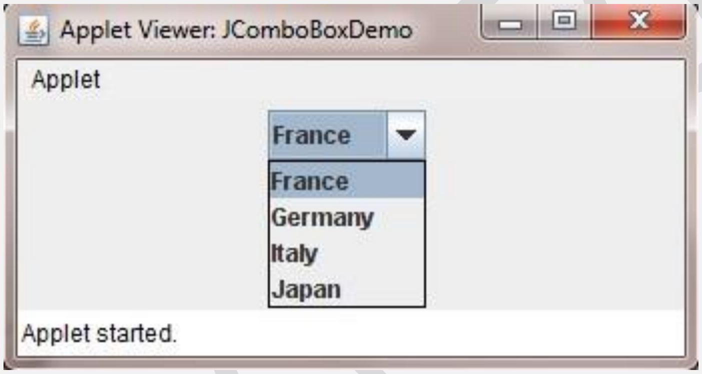
Here, v is a vector that initializes the combo box. Items are added to the list of choices via the addItem( ) method, whose signature is shown here:
void addItem(Object obj) .Here, obj is the object to be added to the combo box.

Program 14.1 **Demonstration of combo box.**

Solution:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
/*
<applet code="JComboBoxDemo" width=300 height=100>
</applet>
*/
public class JComboBoxDemo extends JApplet implements ItemListener
{
        JLabel jl;
        ImageIcon france, germany, italy, japan;
        public void init()
        {
                Container contentPane = getContentPane();
                contentPane.setLayout(new FlowLayout());
                JComboBox jc = new JComboBox();
                jc.addItem("France");
                jc.addItem("Germany");
                jc.addItem("Italy");
                jc.addItem("Japan");
                jc.addItemListener(this);
                contentPane.add(jc);
                jl = new JLabel(new ImageIcon("france.gif"));
                contentPane.add(jl);
        }
        public void itemStateChanged(ItemEvent ie)
        {
                String s = (String)ie.getItem();
                jl.setIcon(new ImageIcon(s + ".gif"));
        }
}
```

Output:



```
Applet started.
```

Tabbed Panes

- A tabbed pane is a component that appears as a group of folders in a file cabinet. Each folder has a title. When a user selects a folder, its contents become visible. Only one of the folders may be selected at a time. Tabbed panes are commonly used for setting configuration options.
- Tabbed panes are encapsulated by the **JTabbedPane** class, which extends **JComponent**. We will use its default constructor. Tabs are defined via the following method:
  void addTab(String str, Component comp)

Here, str is the title for the tab, and comp is the component that should be added to the tab. Typically, a **JPanel** or a subclass of it is added.

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

1. Create a **JTabbedPane** object.
2. Call **addTab( )** to add a tab to the pane. (The arguments to this method define the title of the tab and the component it contains.)
3. Repeat step 2 for each tab.
4. Add the tabbed pane to the content pane of the applet.

Program 14.2 **Demonstration of Tabbed Panes.**

Solution:

```
import javax.swing.*;
/* <applet code="JTabbedPaneDemo" width=400 height=100></applet> */

public class JTabbedPaneDemo extends JApplet
{
        public void init()
        {
        JTabbedPane jtp = new JTabbedPane();
        jtp.addTab("Cities", new CitiesPanel());
        jtp.addTab("Colors", new ColorsPanel());
        jtp.addTab("Flavors", new FlavorsPanel());
        getContentPane().add(jtp);
        }
}
class CitiesPanel extends JPanel
{
        public CitiesPanel()
        {
        JButton b1 = new JButton("New York");
        add(b1);
        JButton b2 = new JButton("London");
        add(b2);
        JButton b3 = new JButton("Hong Kong");
        add(b3);
        JButton b4 = new JButton("Tokyo");
        add(b4);
        }
}
```

```
class ColorsPanel extends JPanel
{
        public ColorsPanel()
        {
                JCheckBox cb1 = new JCheckBox("Red");
                add(cb1);
                JCheckBox cb2 = new JCheckBox("Green");
                add(cb2);
                JCheckBox cb3 = new JCheckBox("Blue");
                add(cb3);
        }
}
class FlavorsPanel extends JPanel
{
        public FlavorsPanel()
        {
        JComboBox jcb = new JComboBox();
        jcb.addItem("Vanilla");
        jcb.addItem("Chocolate");
        jcb.addItem("Strawberry");
        add(jcb);
        }
}
```

Scroll Pane

- A scroll pane is a component that presents a rectangular area in which a component may be viewed.
- Horizontal and/or vertical scroll bars may be provided if necessary. Scroll panes are implemented in Swing by the **JScrollPane** class, which extends **JComponent**.
- Some of its constructors are shown here:
  JScrollPane(Component comp)
  JScrollPane(int vsb, int hsb)
  JScrollPane(Component comp, int vsb, int hsb)

Here, comp is the component to be added to the scroll pane. vsb and hsb are int constants that define when vertical and horizontal scroll bars for this scroll pane are shown. These constants are defined by the ScrollPaneConstants interface. Some examples of these constants are described as follows:

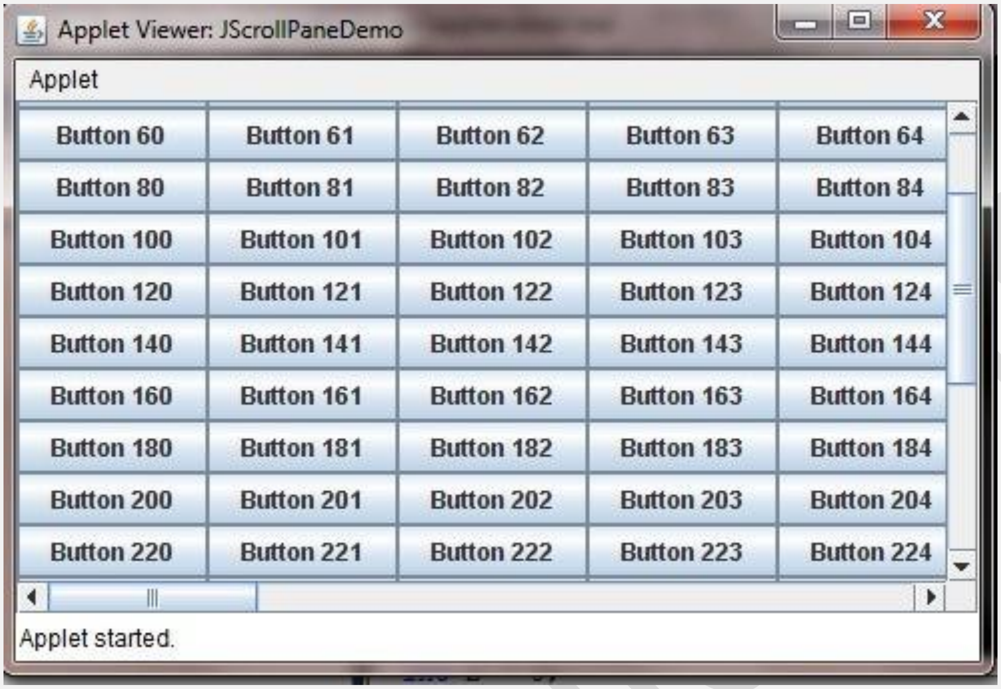
Constant	**Description**
HORIZONTAL_SCROLLBAR_ALWAYS	Always provide horizontal scroll bar

HORIZONTAL_SCROLLBAR_AS_NEEDED	Provide horizontal scroll bar, if needed
VERTICAL_SCROLLBAR_ALWAYS	Always provide vertical scroll bar
VERTICAL_SCROLLBAR_AS_NEEDED	Provide vertical scroll bar, if needed

Here are the steps that you should follow to use a scroll pane in an applet:
1. Create a JComponent object.
2. Create a JScrollPane object. (The arguments to the constructor specify the component and the policies for vertical and horizontal scroll bars.)
3. Add the scroll pane to the content pane of the applet.

```
import java.awt.*;
import javax.swing.*;
/* <applet code="JScrollPaneDemo" width=300 height=250> </applet> */
public class JScrollPaneDemo extends JApplet
{
        public void init()
        {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
        JPanel jp = new JPanel();
        jp.setLayout(new GridLayout(20, 20));
        int b = 0;
        for(int i = 0; i < 20; i++)
        {
                for(int j = 0; j < 20; j++)
                {
                jp.add(new JButton("Button " + b));
                ++b;
                }
        }
        int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
        int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
        JScrollPane jsp = new JScrollPane(jp, v, h);
        contentPane.add(jsp, BorderLayout.CENTER);
        }
}
```

Output:



Applet started.

Tress

- 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**.
- Some of its constructors are shown here:
  JTree(Hashtable ht)
  JTree(Object obj[ ])
  JTree(TreeNode tn)
  JTree(Vector v)

The first form creates a tree in which each element of the hash table ht is a child node.Each element of the array obj is a child node in the second form. The tree node tn is the root of the tree in the third form. Finally, the last form uses the elements of vector v as child nodes.
A JTree object generates events when a node is expanded or collapsed. The addTreeExpansionListener( ) and removeTreeExpansionListener( ) methods allow listeners to register and unregister for these notifications. The signatures of these methods are shown here:
void addTreeExpansionListener(TreeExpansionListener tel)
void removeTreeExpansionListener(TreeExpansionListener tel)
Here, tel is the listener object.

The getPathForLocation( ) method is used to translate a mouse click on a specific point of the tree to a tree path. Its signature is shown here:

TreePath getPathForLocation(int x, int y)

Here, x and y are the coordinates at which the mouse is clicked. The return value is a TreePath object that encapsulates information about the tree node that was selected by the user. The TreePath class encapsulates information about a path to a particular node in a tree. It provides several constructors and methods. In this book, only the toString( ) method is used. It returns a string equivalent of the tree path.

Tree expansion events are described by the class TreeExpansionEvent in the javax.swing.event package. The getPath( ) method of this class returns a TreePath object that describes the path to the changed node. Its signature is shown here:

TreePath getPath( )

The TreeExpansionListener interface provides the following two methods:

void treeCollapsed(TreeExpansionEvent *tee*)

void treeExpanded(TreeExpansionEvent *tee*)

Here, *tee* is the tree expansion event. The first method is called when a subtree is hidden, and the second method is called when a subtree becomes visible.

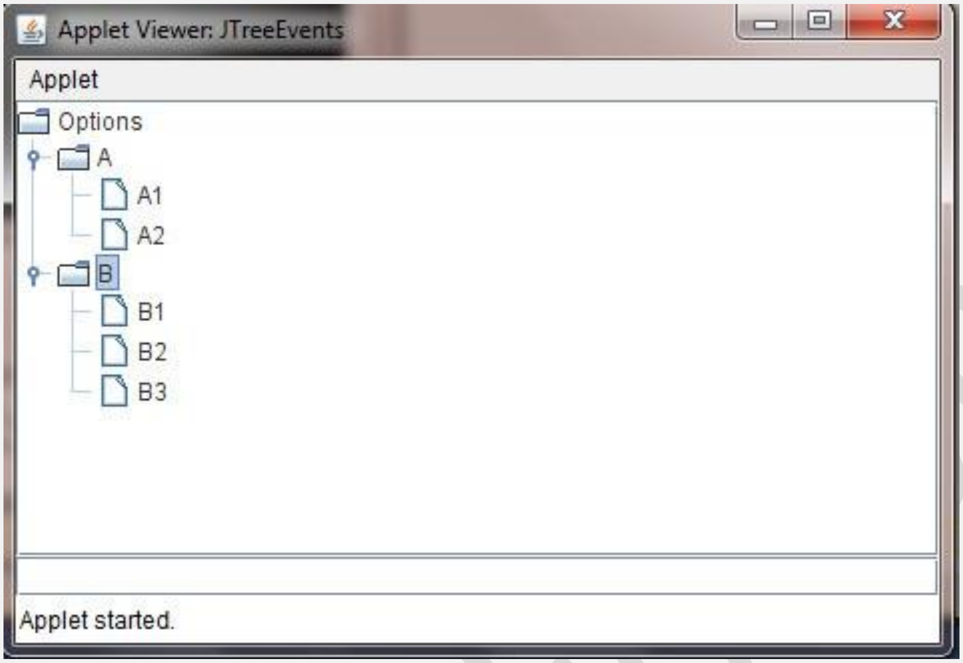
Here are the steps that you should follow to use a tree in an applet:

1. Create a JTree object.
2. Create a JScrollPane object. (The arguments to the constructor specify the tree and the policies for vertical and horizontal scroll bars.)
3. Add the tree to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
/*
<applet code="JTreeEvents" width=400 height=200> </applet> */
public class JTreeEvents extends JApplet
{
        JTree tree;
        JTextField jtf;
        public void init()
        {
        Container contentPane = getContentPane();
        contentPane.setLayout(new BorderLayout());
```

```
    DefaultMutableTreeNode top = new DefaultMutableTreeNode("Options");
    DefaultMutableTreeNode a = new DefaultMutableTreeNode("A");
    top.add(a);
    DefaultMutableTreeNode a1 = new DefaultMutableTreeNode("A1");
    a.add(a1);
    DefaultMutableTreeNode a2 = new DefaultMutableTreeNode("A2");
    a.add(a2);
    DefaultMutableTreeNode b = new DefaultMutableTreeNode("B");
    top.add(b);
    DefaultMutableTreeNode b1 = new DefaultMutableTreeNode("B1");
    b.add(b1);
    DefaultMutableTreeNode b2 = new DefaultMutableTreeNode("B2");
    b.add(b2);
    DefaultMutableTreeNode b3 = new DefaultMutableTreeNode("B3");
    b.add(b3);
    tree = new JTree(top);
    int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
    int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
    JScrollPane jsp = new JScrollPane(tree, v, h);

    contentPane.add(jsp, BorderLayout.CENTER);
    jtf = new JTextField("", 20);
    contentPane.add(jtf, BorderLayout.SOUTH);
    tree.addMouseListener(new MouseAdapter() {
    public void mouseClicked(MouseEvent me) {
    doMouseClicked(me);
    }
    });
    }
    void doMouseClicked(MouseEvent me)
    {
        TreePath tp = tree.getPathForLocation(me.getX(), me.getY());
        if(tp != null)
        jtf.setText(tp.toString());
        else
        jtf.setText("");
    }
}
```

Output:



Table

- A table is a component that displays rows and columns of data.
- You can drag thecursor 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**.
- One of its constructors is shown here:
  JTable(Object data[ ][ ], Object colHeads[ ])

Here, data is a two-dimensional array of the information to be presented, and colHeads is a one-dimensional array with the column headings.

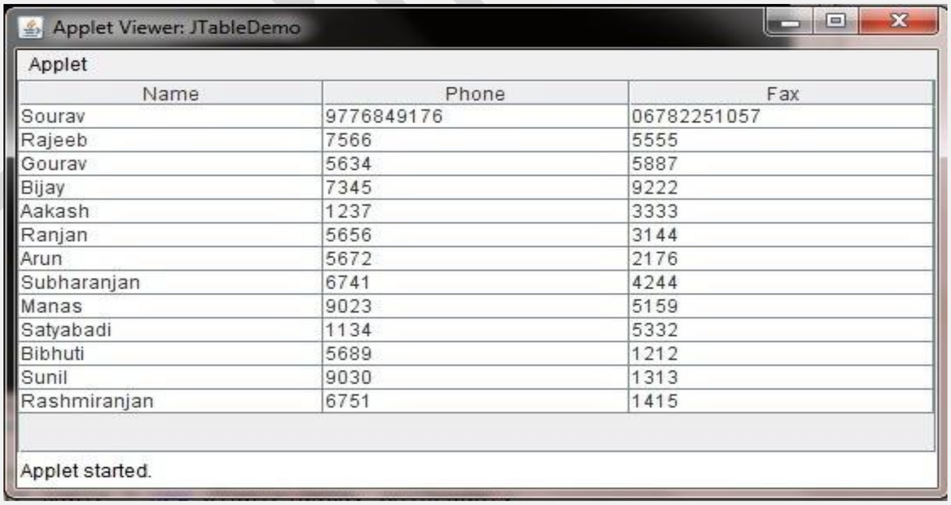
Here are the steps for using a table in an applet:

1. Create a **JTable** object.
2. Create a **JScrollPane** object. (The arguments to the constructor specify the table
and the policies for vertical and horizontal scroll bars.)
3. Add the table to the scroll pane.
4. Add the scroll pane to the content pane of the applet.

```
import java.awt.*;
import javax.swing.*;
/*<applet code="JTableDemo" width=400 height=200></applet>*/
public class JTableDemo extends JApplet
 {
```

```
public void init() {
Container contentPane = getContentPane();
contentPane.setLayout(new BorderLayout());
final String[] colHeads = { "Name", "Phone", "Fax" };
final Object[][] data = {
{ "Sourav", "9776849176", "06782251057" },
{ "Rajeeb", "7566", "5555" },
{ "Gourav", "5634", "5887" },
{ "Bijay", "7345", "9222" },
{ "Aakash", "1237", "3333" },
{ "Ranjan", "5656", "3144" },
{ "Arun", "5672", "2176" },
{ "Subharanjan", "6741", "4244" },
{ "Manas", "9023", "5159" },
{ "Satyabadi", "1134", "5332" },
{ "Bibhuti", "5689", "1212" },
{ "Sunil", "9030", "1313" },
{ "Rashmiranjan", "6751", "1415" }
};
JTable table = new JTable(data, colHeads);
int v = ScrollPaneConstants.VERTICAL_SCROLLBAR_AS_NEEDED;
int h = ScrollPaneConstants.HORIZONTAL_SCROLLBAR_AS_NEEDED;
JScrollPane jsp = new JScrollPane(table, v, h);
contentPane.add(jsp, BorderLayout.CENTER);
}
}
```

Output:



Name	Phone	Fax
Sourav	9776849176	06782251057
Rajeeb	7566	5555
Gourav	5634	5887
Bijay	7345	9222
Aakash	1237	3333
Ranjan	5656	3144
Arun	5672	2176
Subharanjan	6741	4244
Manas	9023	5159
Satyabadi	1134	5332
Bibhuti	5689	1212
Sunil	9030	1313
Rashmiranjan	6751	1415

Applet started.

Assignment 14

Short Type Questions

1. What is JApplet?
2. What is combo box?

Long Type Questions

1. Explain table in swing with an example.
2. Explain tree in swing with an example.
3. Discuss scroll pane in swing.
4. Differentiate between swing and applet.

Java.lang

Java.lang provides classes that are fundamental to the design of the Java programming language. The most important classes are Object, which is the root of the class hierarchy, and Class, instances of which represent classes at run time.

Frequently it is necessary to represent a value of primitive type as if it were an object. The wrapper classes Boolean, Character, Integer, Long, Float, and Double serve this purpose. An object of type Double, for example, contains a field whose type is double, representing that value in such a way that a reference to it can be stored in a variable of reference type. These classes also provide a number of methods for converting among primitive values, as well as supporting such standard methods as equals and hash Code. The Void class is a non-instantiable class that holds a reference to a Class object representing the primitive type void.

The class Math provides commonly used mathematical functions such as sine, cosine, and square root. The classes String and StringBuffer similarly provide commonly used operations on character strings.

Wrapper class

Wrapper class is a wrapper around a primitive data type. It represents primitive data types in their corresponding class instances e.g. a boolean data type can be represented as a Boolean class instance. All of the primitive wrapper classes in Java are immutable i.e. once assigned a value to a wrapper class instance cannot be changed further. Wrapper Classes are used broadly with Collection classes in the java.util package and with the classes in the java.lang.reflect reflection package. Following table lists the primitive types and the corresponding wrapper classes:

Primitive	Wrapper
boolean	java.lang.Boolean
byte	java.lang.Byte
char	java.lang.Character
double	java.lang.Double
float	java.lang.Float
int	java.lang.Integer
long	java.lang.Long
short	java.lang.Short
void	java.lang.Void

Using clone( ) and the Cloneable Interface

The clone( ) method generates a duplicate copy of the object on which it is called. Only classes that implement the Cloneable interface can be cloned. The Cloneable interface defines no members. It is used to indicate that a class allows a bitwise copy of an object (that is, a clone) to be made. If you try to call clone( ) on a class that does not implement Cloneable, a CloneNotSupportedException is thrown. When a clone is made, the constructor for the object being cloned is not called. A clone is simply an exact copy of the original.

Cloning is a potentially dangerous action, because it can cause unintended side effects. For example, if the object being cloned contains a reference variable called obRef, then when the clone is made, obRef in the clone will refer to the same object as does obRef in the original. If the clone makes a change to the contents of the object referred to by obRef, then it will be changed for the original object, too. Here is another example. If an object opens an I/O stream and is then cloned, two objects will be capable of operating on the same stream. Further, if one of these objects closes the stream, the other object might still attempt to write to it, causing an error. Because cloning can cause problems, clone( ) is declared as protected inside Object. This means that it must either be called from within a method defined by the class that implements Cloneable, or it must be explicitly overridden by that class so that it is public.

Program 15.1 **Demonstration of clone() and cloneable interface.**

Solution:

```
class TestClone implements Cloneable
{
        int a;
        double b;
        TestClone cloneTest()
        {
        try
        {
        return (TestClone) super.clone();
        }
        catch(CloneNotSupportedException e)
        {
        System.out.println("Cloning not allowed.");
        return this;
        }
        }
}
class CloneDemo
{
        public static void main(String args[])
        {
        TestClone x1 = new TestClone();
        TestClone x2;
        x1.a = 10;
        x1.b = 20.98;
        x2 = x1.cloneTest(); // clone x1
        System.out.println("x1: " + x1.a + " " + x1.b);
        System.out.println("x2: " + x2.a + " " + x2.b);
        }
}
```

Output:

```
X1: 10   20:98
X2:10    20.98
```

Thread, ThreadGroup, and Runnable

The Runnable interface and the Thread and ThreadGroup classes support multithreaded programming. Each is examined next.

The Runnable Interface

The Runnable interface must be implemented by any class that will initiate a separate thread of execution. Runnable only defines one abstract method, called run( ), which is the entry point to the thread. It is defined like this:
abstract void run( )
Threads that you create must implement this method.

Thread

Thread creates a new thread of execution. It defines the following commonly used constructors:
Thread( )
Thread(Runnable threadOb)
Thread(Runnable threadOb, StringthreadName)
Thread(String threadName)
Thread(ThreadGroup groupOb, Runnable threadOb)
Thread(ThreadGroup groupOb, Runnable threadOb, String threadName)
Thread(ThreadGroup groupOb, String threadName)
threadOb is an instance of a class that implements the Runnable interface and defines where execution of the thread will begin. The name of the thread is specified by threadName. When a name is not

specified, one is created by the Java Virtual Machine. groupOb specifies the thread group to which the new thread will belong. When no thread
group is specified, the new thread belongs to the same group as the parent thread. The following constants are defined by Thread:
MAX_PRIORITY
MIN_PRIORITY
NORM_PRIORITY

ThreadGroup

ThreadGroup creates a group of threads. It defines these two constructors:

ThreadGroup(String groupName)

ThreadGroup(ThreadGroup parentOb, String groupName)

For both forms, groupName specifies the name of the thread group. The first version creates a new group that has the current thread as its parent. In the second form, the parent is specified by parentOb.

Assignment 15

Short Type Questions

1. Define wrapper class.
2. What is the use of thread priority?

Long Type Questions

1. What are thread priorities? Explain.
2. Discuss the use of clone and clonable interface with an example.

BPUT 5th Semester Examination 2012

1. Answer all the questions :
   - (a) Can java run on any machine? What is needed to run java on a computer?
   - (b) What is the statement to display a string on the console? What is the statement to display the message “Hello world” in a message dialog box?
   - (c) What is y after the following switch statement is executed?

```
X=3 ; y=3 ;
switch (x+3){
        case 6: y=1;
        default: y+=1;
}
```

   - (d) Is it possible to declare a class as both abstract and final? Explain your answer.
   - (e) What are the steps to add a class to a package?
   - (f) Explain the difference between method overloading and method overriding.
   - (g) What are the states of java Applet life cycle?
   - (h) What is the difference between doGet() and doPost()?
   - (i) Why doesn’t a JComboBox send out change events?
   - (j) State two ways to create threads in Java.

2. (a) What do you mean by object oriented programming techniques? Explain how Java language facilitate better structured programming design by using class and object constructors over traditional languages like C and C++.

   (b) What is Synchronization? Why is it important? Give one example.

3. (a) Write a Java applet program to display Fibonacci series of first 10 terms inside a label.

   (b) What is JDBC? State its application. How many types of JDBC Drives are present and what are they?

4. (a) What do you mean by RMI? How will you pass parameters in RMI? With suitable diagram discuss the different layers of RMI.

   (b) Demonstrate how to create and throw chained exceptions using a programming example.

5. (a) Describe the <applet> HTML tag. How do you pass parameters to an applet? Explain with one example.

   (b) State true or false with proper justification:

   i. A subclass is a subset of a superclass.
   ii. When invoking a constructor from a subclass, its superclass’s no-arg constructor is always invoked.
   iii. You can override a private method defined in a superclass.
   iv. You can override a static method defined in a superclass.

6. (a) State the differences between Swing and AWT. Explain important components and container in AWT.
   (b) What are the advantages of RMI over socket-level programming?

7. (a) What is the layout manager used in JToolBar? Can you change the layout manager? Explain your answer.
   (b) Write a Java program to find out factorial of a number using I/O Exception.

8. (a) Write short notes on any two:
   - a) TCP/IP Server Sockets
   - b) this keyword
   - c) Runtime memory management
   - d) Object cloning.