Q.1(a)i) (1-mark for each feature)

OOP IS SIMPLE: Java is easy to write and understand. Java programmer doesn't have to work with pointers and doesn't need to manage memory explicitly. Java eliminated the use of pointers for security reasons. Its virtual machine is able to handle the memory management and so removes the occupied memory automatically when it is no longer referenced. Programmer can now concentrate on the required application logic rather than wasting time and logic with these managements.

OOP IS DISTRIBUTED: Java supports network programming to communicate with remote objects distributed over the network. Java provides libraries like RMI and CORBA to develop network applications.

OOP IS INTERPRETED: Java is called interpreted language because its byte code, which is generated after compiling the source code, is interpreted by the JVM and converts to machine dependent code which is also called native code.

OOP IS ROBUST AND SECURE: Java applications are reliable in various ways. It offers compile time checking to detect early the causes of bugs, run time checking, eliminates the use of pointers which can cause memory corruption or unwanted access of memory, garbage collection management to free the unused memories automatically and exception handling to handle the situation at the time of occurrence of any error and a lot more.

Q.1 (a)ii) (definition 1-mark, Example 3-marks)
A thread is a similar to a program that has a single low of control. A thread is known as an execution context or a lightweight process. Every thread has a beginning, a body and an end. However, thread is not a program, but runs within a program

Example:-

```
class Ascending extends Thread
{
public  void run()
{
System.out.println("\n\t Ascending order of numbers::" );
for(int i=1;i<=10 ;i++)
{
System.out.println("\t"+i);
}
}
}
Class Decending extends Thread
{
Public void run()
{
 System.out.println("\n\t Decending order of numbers::" );
for(int i=10;i>=1;i--)
{
System.out.println("\t"+i);
}
}
}
```

```
class Threadtest1
{
Public  static void main(String args[])
{

System.out.println("\n\t One Thread Printing  numbers in Ascending order is started::" );
new Ascending().start();

System.out.println(" Second Thread Printing  numbers in Decending order is started::" );
new Decending().start();
}
}
```

Q.1 (a) iii) (Brief Explanation 4-marks)

Java supports the features portability. Java programs can be easily moved from one computer system to another, anywhere and anytime. Java compiler on the other hand does not compile Java source files into native machine language, instead it compiles the source code into bytecodes. These byte codes are platform independent i.e. in other words specific to the Java Virtual Machine specification. This enables platform independent compilation. When the bytecode compiled programs are executed thru the Java interpreter, it converts those bytecodes into native machine code and executes them thru the JVM which is specific to host environment it is running on. This enables platform specific execution.

Java ensures portability in two ways.

(i) First, java compiler generates bytecode instructions that can be implemented on any machine.
(ii) Secondly, the size of the primitive data types is machine independent.

Q.1 (a) iv)(Explanation 2-marks, Example 2-marks)

Interface is also known as kind of a class. So interface contain methods and variable but with major difference the interface defines only abstract method and final fields. This means interface that interface do not specify any code to implement those method and data fields contain only constants.

Variable of interface are explicitly declared final and static (as constant) it means that the implementing class cannot change them. They must be initialized with constat value. The entire variable is implicitly public if the interface. Itself, is declared as public

Method declaration contains only a list of methods without any body statement and ends with semicolon. The methods are, essentially, abstract methods; there can be no default implementation of any method specified within an interface. Each class that includes an interface must implement all of the methods.

EX:

```
interface student
{
static final int rollno =077
static final string name="Pawan";
void display();
}
```

The code for the method is not included in the interface and method declaration ends with semicolon. The class that implements this interface must define the code for the method.

Q.1 (b) i) (2-marks for each)

Final variable:

A final variable can only be initialized once, either via an initializer or an assignment statement. It does not need to be initialized at the point of declaration: this is called a "blank final" variable. A blank final instance variable of a class must be definitely assigned at the end of every constructor of the class in which it is declared; similarly, a blank final static variable must be definitely assigned in a static initializer of the class in which it is declared; otherwise, a compile-time error occurs in both cases.(Note: If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references is still mutable, if it was originally mutable.)

Unlike the value of a constant, the value of a final variable is not necessarily known at compile time.

```
public class Sphere {
   // pi is a universal constant, about as constant as anything can be.
   public static final double PI = 3.141592653589793;

   public final double radius;
   public final double xpos;
   public final double ypos;
   public final double zpos;

   Sphere(double x, double y, double z, double r) {
       radius = r;
       xpos = x;
       ypos = y;
       zpos = z;
   }

   [...]
}
```

Final methods:

A final method can't be overridden by subclasses. This is used to prevent unexpected behaviour from a subclass altering a method that may be crucial to the function or consistency of the class.

```
public class MyClass {
public void myMethod() {...}
public final void myFinalMethod() {...}
}

public class AnotherClass  extends MyClass {
public void myMethod() {...}           // Ok
public final void myFinalMethod() {...} // forbidden
}
```

A common misconception is that declaring a class or method final improves efficiency by allowing the compiler to directly insert the method inline wherever it is called. In fact the compiler is unable to do this because the method is loaded at runtime and might not be the same version as the one that was just compiled. Only the runtime environment and JIT compiler have the information about exactly which classes have been loaded, and are able to make better decisions about when to inline, whether or not the method is final.

Abstract method:-

An abstract class is a class that is declared abstract—it may or may not include abstract methods. Abstract classes cannot be instantiated, but they can be subclassed.
An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon), like this:
abstract void moveTo(double deltaX, double deltaY);
If a class includes abstract methods, the class itself must be declared abstract, as in:

```
public abstract class GraphicObject {
  // declare fields
  // declare non-abstract methods
  abstract void draw();
}
```

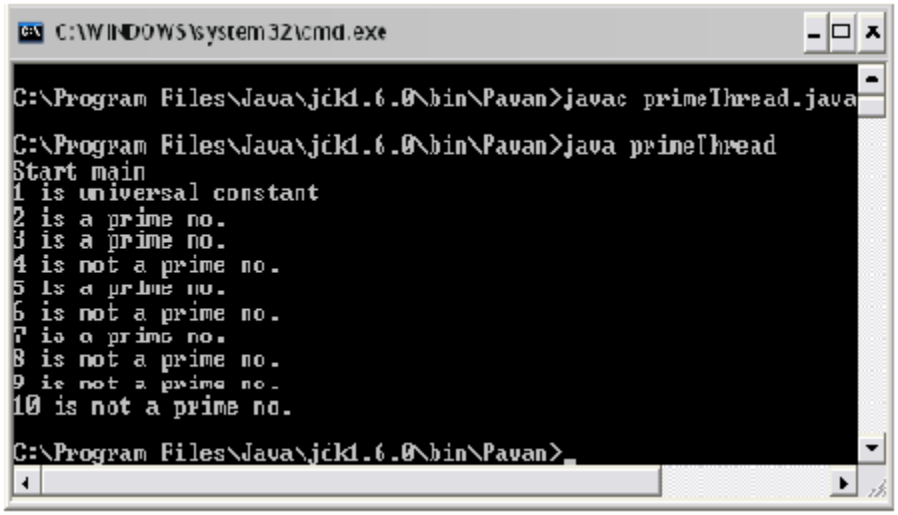
When an abstract class is subclassed, the subclass usually provides implementations for all of the abstract methods in its parent class. However, if it does not, the subclass must also be declared abstract.

Q.1(b)ii) (Program 5-marks, Output 1-mark)

```
class prime extends Thread
{
public void run()
{
int i=0,j=0;
for(i=2;i<11;i++){
for(j=2;j<i;j++)
{
if(i%j==0)
j=i+1;}
if(i==j)
System.out.println(i+" is a prime no.");
try{
sleep(500);
}
catch(Exception e){}}}}
class non_prime extends Thread
{
public void run()
{
int i=0,j=0;
for(i=2;i<11;i++){
for(j=2;j<i;j++){
if(i%j==0)
j=i+1;}
if(i!=j)
System.out.println(i+" is not a prime no.");
try
{
sleep(500);
}
```

```
catch(Exception e){}}}}
class primeThread
{
public static void main(String args[])
{
prime p=new prime();
non_prime n=new non_prime();
System.out.println("Start main");
System.out.println("1 is universal constant");
p.start();
n.start();
}
}
```

OUTPUT:-



Q.2 (a)(2-marks for each access modifiers)

Visibility control in Java is implemented using Access Modifiers. An Access Modifier is a key word in java that determines what level of access or visibility a particular java variable/method or class has. There are 4 basic access modifiers in java. They are:

1. Visible to the package. the default. No modifiers are needed.
2. Visible to the class only (private).
3. Visible to the world (public).
4. Visible to the package and all subclasses (protected).

Default Access Modifier - No keyword:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method etc.
A variable or method declared without any access control modifier is available to any other class in the same package. The default modifier cannot be used for methods, fields in an interface.

Example:

Variables and methods can be declared without any modifiers, as in the following examples:

```
String version = "1.5.1";

boolean processOrder() {
  return true;
}
```

Private Access Modifier - private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself.
Private access modifier is the most restrictive access level. Class and interfaces cannot be private.
Variables that are declared private can be accessed outside the class if public getter methods are present in the class.
Using the private modifier is the main way that an object encapsulates itself and hide data from the outside world.

Example:

The following class uses private access control:

```
class Logger {
  private String format;
  public String getFormat() {
    return this.format;
  }
  public void setFormat(String format) {
    this.format = format;
  }
}
```

Here, the format variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So to make this variable available to the outside world, we defined two public methods:getFormat(), which returns the value of format, and setFormat(String), which sets its value.

Public Access Modifier - public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.
However if the public class we are trying to access is in a different package, then the public class still need to be imported.
Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example:

The following function uses public access control:

```
public static void main(String[] arguments) {
  // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.
The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.
Protected access gives the subclass a chance to use the helper method or variable, while preventing a no related class from trying to use it.

Example:

The following parent class uses protected access control, to allow its child class override*openSpeaker()* method:

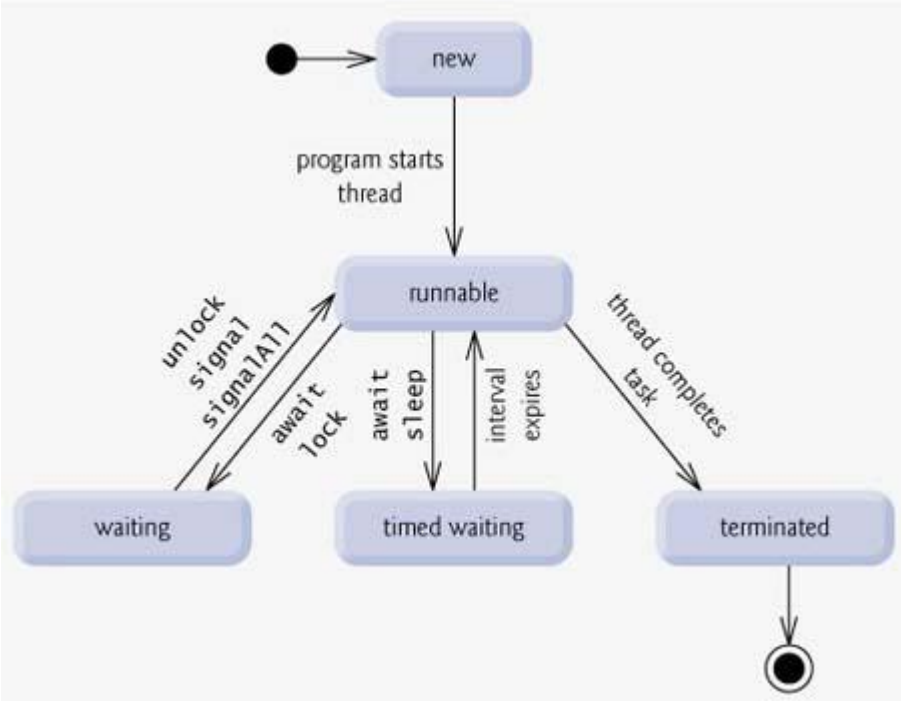
```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}

class StreamingAudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}
```

Here if we define openSpeaker() method as private then it would not be accessible from any other class other than AudioPlayer. If we define it as public then it would become accessible to all the outside world. But our intension is to expose this method to its subclass only, thats why we used protected modifier.

Q.2 (b) (Diagram 2-marks, Description 6-marks)

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.



Above mentioned stages are explained here:

- **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.

Q.2 (c) (Diagram 2-marks, Description 6-marks)
In Java, thread scheduler can use the thread **priorities** in the form of **integer value** to each of its thread to determine the execution schedule of threads. Thread gets the **ready-to-run** state according to their priorities. The **thread scheduler** provides the CPU time to thread of highest priority during ready-to-run state.
Priorities are integer values from 1 (lowest priority given by the constant **Thread.MIN_PRIORITY**) to 10 (highest priority given by the constant **Thread.MAX_PRIORITY**). The default priority is 5(**Thread.NORM_PRIORITY**).

The methods that are used to set the priority of thread shown as

Method	**Description**
setPriority()	This is method is used to set the priority of thread.
getPriority()	This methodetg is used to get the priority of thread.

When a Java thread is created, it inherits its priority from the thread that created it. At any given time, when multiple threads are ready to be executed, the runtime system chooses the runnable thread with the highest priority for execution. In Java runtime system, **preemptive scheduling** algorithm is applied. If at the execution time a thread with a higher priority and all other threads are runnable then the runtime system chooses the new**higher priority** thread for execution. On the other hand, if two threads of the same priority are waiting to be executed by the CPU then the **round-robin** algorithm is applied in which the scheduler chooses one of them to run according to their round of **time-slice**.

Thread Scheduler

In the implementation of threading scheduler usually applies one of the two following strategies:

Preemptive scheduling ? If the new thread has a higher priority then current running thread leaves the runnable state and higher priority thread enter to the runnable state.

Time-Sliced (Round-Robin) Scheduling ? A running thread is allowed to be execute for the fixed time, after completion the time, current thread indicates to the another thread to enter it in the runnable state.

You can also set a thread's priority at any time after its creation using the **setPriority** method. Lets see, how to set and get the priority of a thread.

Example:

```
class MyThread1 extends Thread{
  MyThread1(String s){
  super(s);
  start(); }
  public void run(){
  for(int i=0;i<3;i++){
  Thread cur=Thread.currentThread();
  cur.setPriority(Thread.MIN_PRIORITY);
  int p=cur.getPriority();
  System.out.println("Thread Name  :"+Thread.currentThread().getName());
  System.out.println("Thread Priority  :"+cur);}}}
```

```
class MyThread2 extends Thread{
  MyThread2(String s){
  super(s);
  start(); }
public void run(){
  for(int i=0;i<3;i++){
  Thread cur=Thread.currentThread();
  cur.setPriority(Thread.MAX_PRIORITY);
  int p=cur.getPriority();
  System.out.println("Thread Name  :"+Thread.currentThread().getName());
  System.out.println("Thread Priority  :"+cur);}}}
public class ThreadPriority1{
  public static void main(String args[]){
  MyThread1 m1=new MyThread1("My Thread 1");
  MyThread2 m2=new MyThread2("My Thread 2");
  }
}
```

Q.3 (a)(Any four 1 –mark each)

- **Compiled and Interpreted**

Java compiler translates source code into what is known as bytecode instructions. Bytecodes are not machine instructions and therefore, in the second stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program. We can thus say that java is both a compiled and interpreted language.

- **Platform Independent and Portable**

Java programs can be easily moved from one computer system to another, anywhere and anytime. Change and upgrades in operating systems. Processors and system resources will not force any changes in Java Programs.

- **Object-Oriented**

Almost everything in java is an object. All program code and date reside within objects and classes. Java comes with an extensive set of classes, arranged in package that we can use in our programs by inheritance.

- **Robust and Secure**

Java is robust language. It provides many safeguards to ensure reliable code. It has strict compile time and run time checking for data types. It is designed as a garbage collection language relieving the programmers virtually all memory management problems. Java also incorporates the concept of exception handling which captures series errors and eliminates any risk of crashing the system.

- **Simple, small and Familiar**

Java is a small and simple language. Java does not use pointers, preprocessor header files, goto statement and many others. It also eliminates operator overloading and multiple inheritances.

- **Distributed**

Java is designed as a distributed language for creating applications on networks. It has the ability to share both data and programs. Java applications can open and access remote objects on internet as easily as they can do in a local system. This enables multiple programmers at multiple remote locations to collaborate and work together on a single project.

- **Multithreaded and interactive**

Multithreaded means handling multiple tasks simultaneously. Java supports multithreaded programs.

- **High Performance**

Java performance is impressive for an interpreted language, mainly due to the use of intermediate bytecode.

- **Dynamic and Extensible**

Java is a dynamic language. Java is capable of dynamically linking in new class libraries, method, and objects. Java can alco determine the type of class through a query, making it possible to either dynamically link or abort the program, depending on the response.

- **Ease of Development**

Java 2 standard Edition (J2SE) 5.0 supports feature, such as Generic, Enhanced for loop, Authoboxing or unboxing, typesafe Enums, Varargs, static import and annotation. These features reduce the work of the programmer by shifting the reponseibility of creating the reusable code to the compiler. The resulting source code is free from bugs because the errors made by the compiler are less when compared to those made by programmers.

- **Scalability and Performance**

J2SE 5.0 assures a significant increase in scalability and performance by improving the startup time and reducing the amount of memory used in Java 2 runtime environment.

- **Monitoring and Manageability**

Java Supports a number of APIs, suchas JVM Monitoring and Management API, Sun Management Platform Extension, Logging, Monitoring and Management interface, and Java Management Extension(JXM) to monitor and Manage Java applications. Java provides tools, such as Jconsole, jsp, jstat, and jstatd to make use of monitoring and management facilities for example, GUI based tool called jconsole is used to monitor the JVM.

- **Desktop client**

It provides an improved Swing look and feel called Ocean. This feature is mainly used for developing graphics applications that require OpenGL hardware acceleration.

- **Core XML Support**

Adds powerful XML feature to the Java Platform, Java Contains some special packages for interface, to instantiate Simple API for XML (SAX) and Document Object Model (DOM) parsers to parse an XML document, transform the content of an XML document, and validate an XML document against the Schema.

- **Supplementary Character Support**

Java adds the 32-bit supplementary character support as part of the Unicode 4.0 support. The supplementary characters are encoded with UTF-16 values to generate a different character called, surrogate codepoint.

- **JDBC RowSet**

Java Supports JDBC RowSet to send data in a tabular formatbetween the remote components of a distrusted enterprise application. JDBC RowSet contains CachedRowSet and WebRowSet objects. The CacheRowSet object is a JavaBean component which acts like a container. This object contains a number of rows of data, which are retrieved from the database. The data stored in the rows of data that are retrieved from the database can be synchronized later. The WebRowSet object can operate without being connected to the database or data source. The WebRowSet object uses XML format to read and write the rowset.

Q.3 (b)(Casting 3-marks, Needs 1-mark)

There is a need to store a value of one type into a variable or another type. In such situations, we must cast the value to be stored by preceding it with the type name in parentheses. The syntax is:
Type variable = (type) variable;
The process of converting one data type to another is called casting. Examples:
Int m=50;
Byte n=(byte)m;
Long count=(long)m;

Casting into smaller type may result in a loss of data.

Casts that Result in No Loss of Information	
From	TO
byte	Short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

Type casting is used when the conversion is between two incompatible types.

Q.3 (c) **(Two marks for interface, two marks for difference, Any two differences)**

An interface is basically a kind of class. Like classes, interface contain methods and variables but with a major difference. The difference is that interface defines only abstract methods and final field. This means that interfaces do not specify and code to implements these methods and data fields contain only constants. Therefore, it is the responsibility of the class that implements an interface to define the code for implementation of these methods.

Difference between interface and class

1. Interface does not contain instance variables. Hence object of interface can't be created.

2. Interfaces r developed to support multiple inheritance...

3. The methods present in interfaces are abstract. The code for these methods must be written in the class that implements the interface.

4. The access specifiers public, private, protected r possible with classes. But the interface uses only one specifier public.

5. Interfaces define only final fields. There is no such restriction on the class.

Q.3 (d) (2mark for create, 2marks for access)

Creating our own package involves following steps:

1. Declare the package at the beginning of a file using the form

Package packagename

2. Define the class that is to be put in the package and declare it public.
3. Create a subdirectory under the directory where the main source files are stored.
4. Store the listing as the classname.java file in the subdirectory created.
5. Compile the file. This creates. Class file in the subdirectory.

Accessing a package

1. The import statement can be used to search a list of packages for a particular class. The general form of import statement for searching a class is as follows:

Import package1 [.package2][.package3].classname;

Here package1 is the name of the top level package, package2 is the name of package that is inside the package 1, and so on. We can have number of packages in a package hierarchy. Finally, the explicit classname is specified.

Note that the statement must end with a semicolon (;). The import statement should appear before any class definitions in a source file. Multiple import statements are allowed.

2. We can also use another approach as follows:

Import packagename.;*

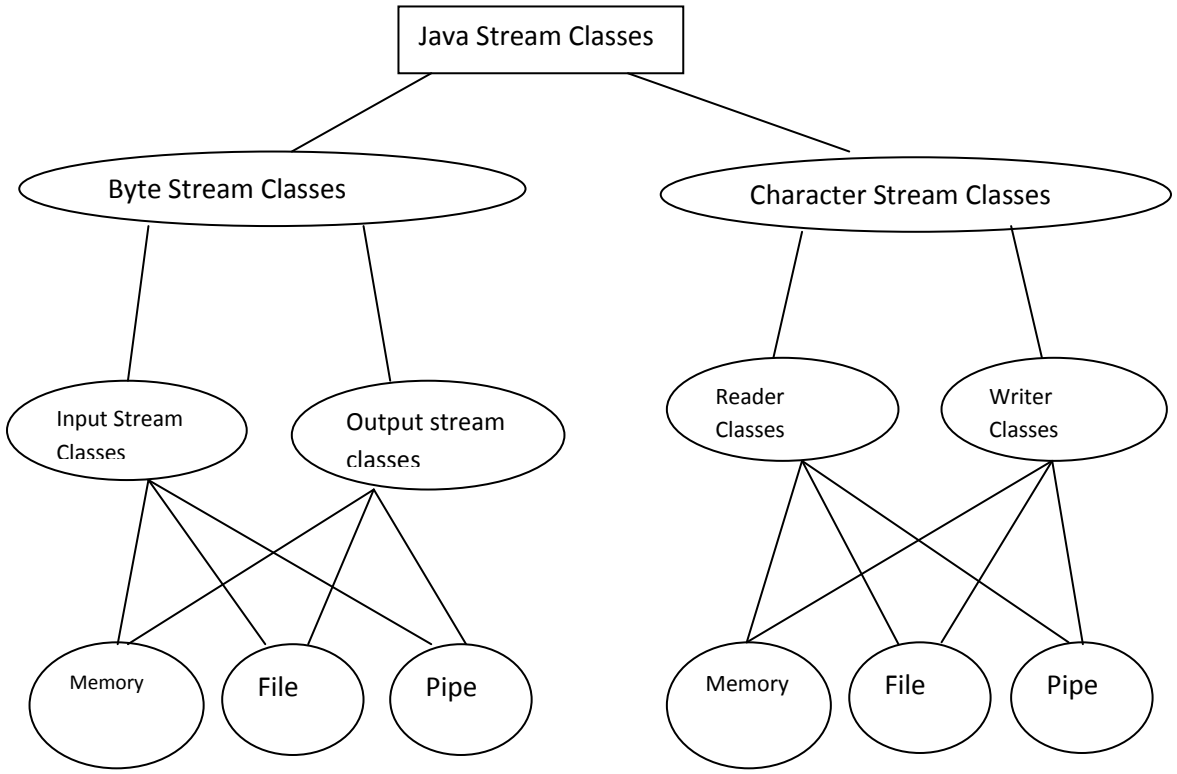
Here packagename may denote a single package or a hierarchy of packages as mentioned earlier. The star(*) indicates that the compiler should search this entire package hierarchy when it encounters a class name. this implies that we can access all classes contained in the above package directly.

Q.3 (e) (Diagram optional) (Stream classes 1 mark, Input stream class 1 1/2marks, output stream class 1 1/2marks)

The java. Io package contain a large number of stream classes that provide capabilities for processing all types of data. These classes may be categorized into two groups based on the data type on which they operate.

1. Byte stream classes that provide support for handling I/O operations on bytes.
2. Character stream classes that provide support for managing I/O operations on characters.

These two groups may further be classified based on their purpose. Show in fig how stream classes are grouped based on their functions. Byte stream and character stream classes contain specialized classes to deal with input and output operations independently on various types of devices.



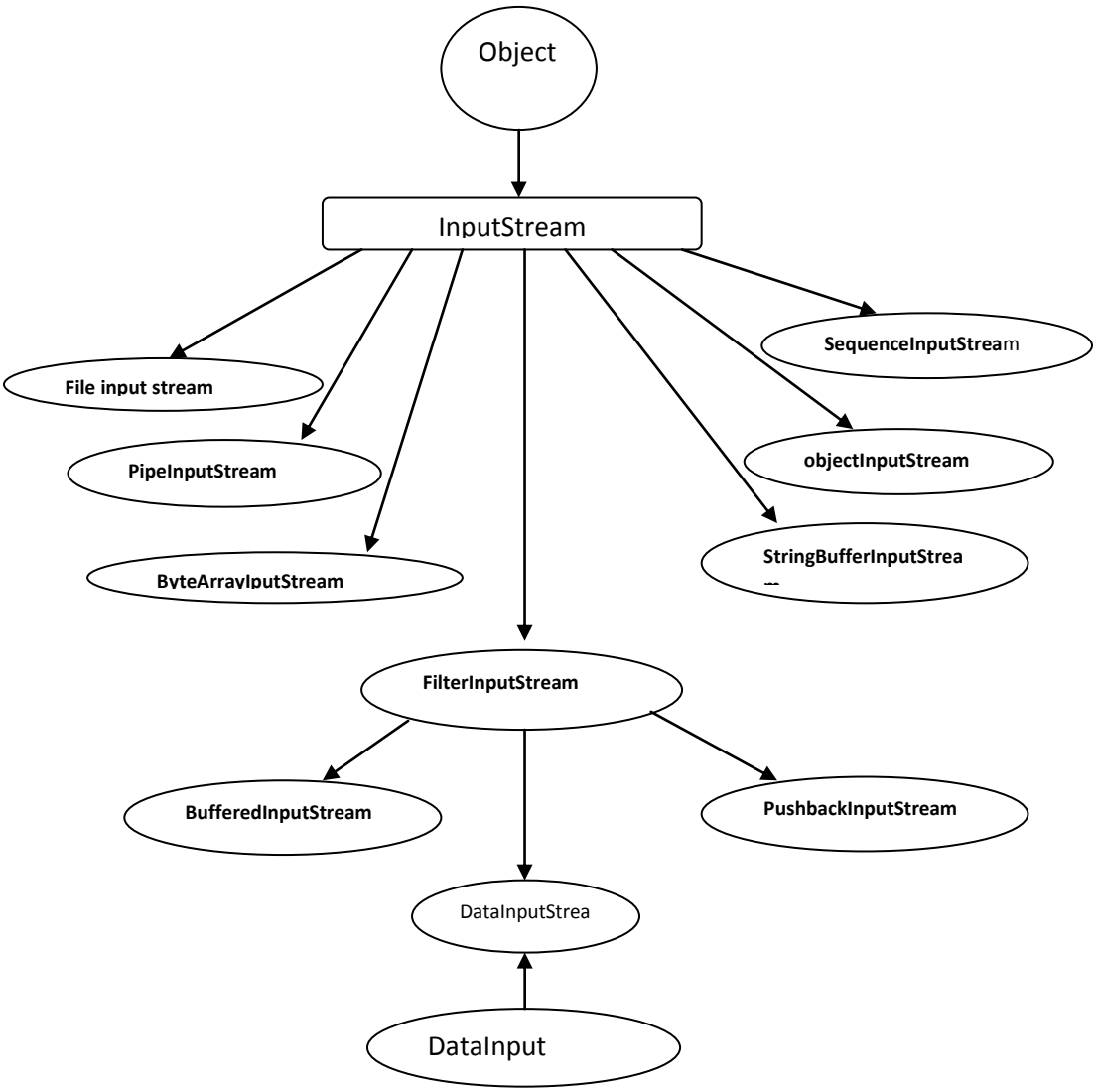
Byte Stream Classes

Byte stream classes have been designed to provide functional features for creating and manipulating streams and files for reading and writing bytes. Since the streams are unidirectional, they can transmit bytes in only one direction and, therefore, Java provides two kinds of byte stream classes: input stream classes and output stream classes.

Input Stream Classes

Input stream classes that are used to read 8-bit bytes include a super class known as InputStream and number of subclasses for supporting various input-related functions. The super class InputStream is an abstract class, and, therefore we cannot create instances of this class. Rather, we must use the subclass that inherits from this class. The InputStream class defines methods for performing input functions such as

- Reading bytes
- Closing Stream
- Marking positions in stream
- Skipping ahead in a stream
- Finding the number of bytes in a stream

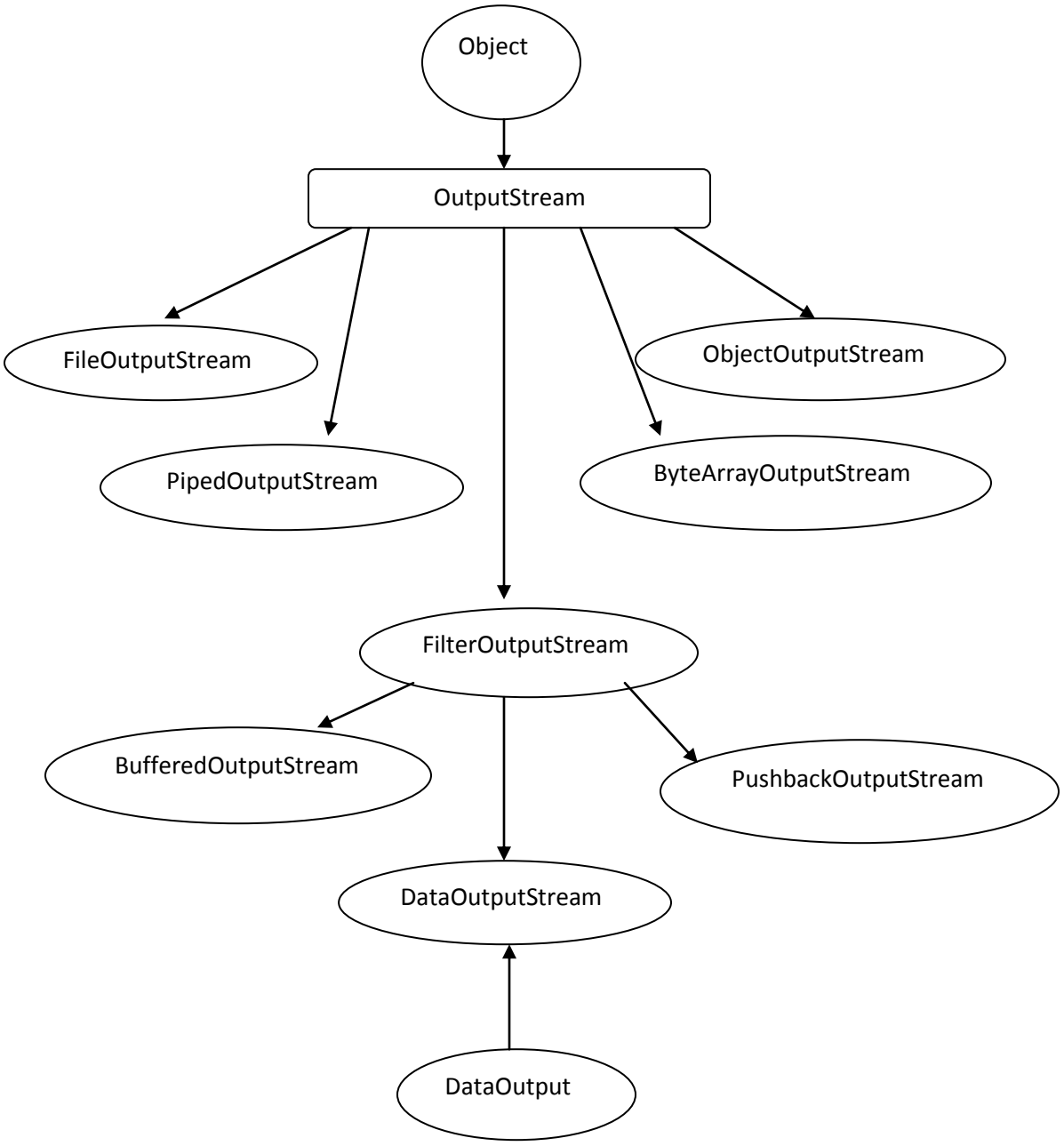


Output Stream Classes

Output Stream classes are derived from the base class OutputStream as shown in fig . like Inputstream, the OutputStream is an abstract class and therefore we cannot instantiate it. The several subclasses of the OutputStream can be used for performing the output operations.

The OutputStream includes methods that are designed to perform the following tasks:

- Writing bytes
- Closing Stream
- Flushing Stream



Q. 4 (a)i) (Brief explanation 4-marks)

Applets are not full-featured application programs. They are usually written to accomplish a small task or a component of a task. Since they are usually designed for use on the Internet, they impose certain limitation and restrictions in their design.

- Applet do not use the main() method for initiating the execution of the code. Applets, when loaded, automatically call certain methods of Applet class to start and execute the applet code.
- Unlike stand-alone applications, applets cannot be run independently. They are run from inside a web page using a special feature known as HTML tag.
- Applets cannot read from or write to the files in the local computer.
- Applets cannot communicate with other server on the network.
- Applets cannot run any program from the local computer.
- Applets are restricted from using libraries from languages such as C or C++.(Remember, Java language supports this feature through native methods).

Q.4 (a)ii) (Brief explanation 4-marks)

Serialization is the process of writing the state of an object to a byte stream. This is useful when you want to save the state of your program to persistent storage area, such as a file. At a later time, you may restore these objects by using the process of deserialization.
Serialization is also needed to implement remote Method Invocation (RMI). RMI allows a java object on one machine to invoke a method of a java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serialized the object and transmits it. The receiving machine desterilizes it.

Q.4 (a)iii) Write a program to create and sort all integer array. (Correct Program 4-marks)

```
class bubble

{

        public static void main(String args[])

        {

        int a[]={10,25,5,20,50,45,40,30,35,55};

        int i=0;

        int j=0;

        int temp=0;

        int l=a.length;

        for(i=0;i<l;i++)

        {
```

```
for(j=(i+1);j<l;j++)
                {
                        if(a[i]>a[j])
                        {
                        temp=a[i];
                        a[i]=a[j];
                        a[j]=temp;
                        }
                }
        }
System.out.println("Ascending order of numbers:");
for(i=0;i<l;i++)
System.out.println(""+a[i]);
}
}
```

Any other suitable program based on the same concept may also be considered.

Q.4 (a)iv) (Correct Program.4-marks)

```
 import java.lang.*;
 import java.io.*;
Class Employee
{
  int emp_id;
  String emp_name;
  float basic_slary;
Employee(int a, String s, float f)
{
  emp_id;
  emp_name=s;
  basic_salary=f;
}
  Void display()
{
   float da=basic_salary*15/100;
    float hra =basic_salary*10/100;
   float gross_salary =basic_salary+da+hra;
   system.out.println(“Emp_id=”+emp_id+”Gross salary=”+gross_salary);
}
```

```
public static void main(String args[])throws IOException
{
BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
System.out.println("Enter emp_id");
  int x=Integer.parseInt(in.readLine());
System.out.println("enter emp_name:");
String n=in.readLine();
System.out.println("Enter basic_salary:");
  float  f=Float.parseFloat(in.readLine());
Employee e = new  Employee(x,n,f)
e.display();
}
}
```

Any other suitable program based on the same concept may also be considered.)

Q.4 (b)i) (1 1/2 for overloading, 1 1/2 for overriding, 1 1/2 for each example.)

Overloading

In Java, it is possible to create methods that have the same name, but different parameter lists and different definitions. This is called as method Overriding. Method overriding is used when objects are required to perform similar tasks but using different input parameters. When we call a method in an object, java matches up the method name first and then number and type of parameters to decide which one of the definitions to execute. This process is known as polymorphism.

Example:

```
Class room

{

float length;

floatbreath;

Room(float x, float y)

{

length =x;

breath=y;

}

Room (float x)

{

length =breath=x;

}
```

```
int area()
{
return (length*breath);
}
}
```

Overriding

There may be occasions when we want an object to respond to the same method but have different behavior when that method is called. This is possible by defining a method in the subclass that has the same name, same arguments and same return type as a method in the superclass. Then, when that method is called, the method defined in the subclass is invoked and executed instead of the one in the superclass. This is known as overriding.

```
class Super
{
int x;
Super (int x)
 {
this.x=x;
}
void display()
{
System.out.println("Super x="+x);
}
}
Class Sub extends Super
{
int y;
Sub( int x, int y)
{
super (x);
this.y=y;
}
```

```
void display()
{
System.out.println("Super x=" +x);
System.out.println(Sub y=" +y);
}
}
class overrideTest
{
public static void main(String arg[])
{
Sub s1= new Sub(100,200);
s1.display();
}
  }
```

Q.4 (b)ii

(any form of inheritance may be considered)(6 marks)

```
 class Room
{
int lenght;
int breadth;
Room(int x, int y)
{
lenght=x;
breadth=y;
}
int area()
{
return(lenght*breadth);
}
}
```

```
class BedRoom extends Room
{
int height;
BedRoom(int x, int y, int z)
{
super(x,y);
height=z;
}
int volume()
{
return(lenght*breadth*height);
}
}
class InherTest
{
public static void main (String args[])
{
BedRoom room1=new BedRoom(14,12,10);
int area1=room1.area();
int volume1=room1.volume();
System.out.println("Area1="+area1);
System.out.println("Volume="+volume1);
}
}
```

(Any other suitable program based on the same concept may also be considered.)

Q.5 (a)

Java provides a mechanism for partitioning for class name space into more manageable chunks. This mechanism is package. The package is both naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class member that are only exposed to other member of the same package. Package can be used to group classes which are needed in one application as well as their visibility from outside world can be controlled.

Syntax to create package is

```
package packagename;

class <classname>

{

…….

……..

}
```

The class inside package can be imported in any other java code with 'import' as;

Example:

```
package  mypack

public  class  box

{

int l, b, h;

public box()

{

l=5;

b=6;

h=7;

}

public void volume()

{

int v=l*b*h;
```

System .out.println("Volume:"+v);

}

}

This class can be imported in another java file.

import mypack;

class test

{

public static void main(String arg[])

box B= new box();

B.volume();

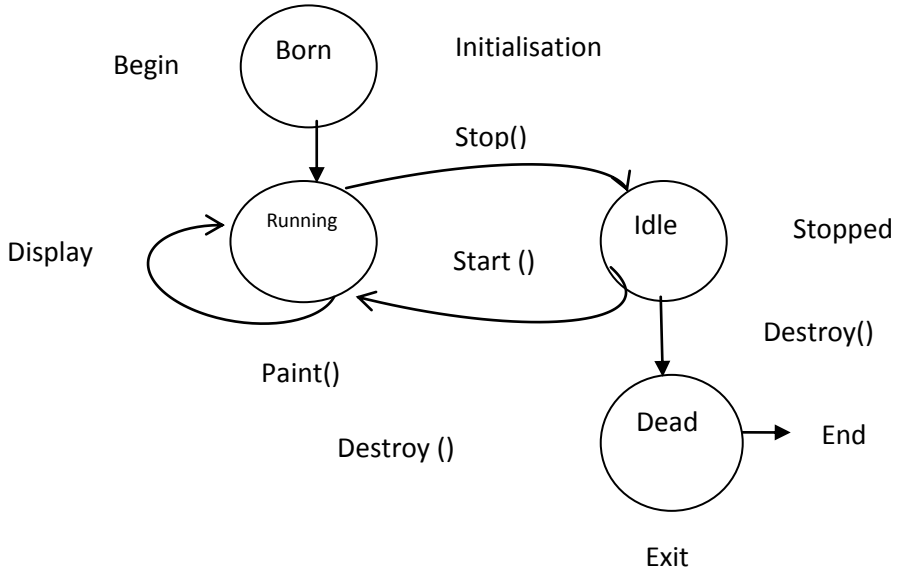
}

}

This class file named box should be copied inside directory mypack which reflects the same name as package name. Then only it can be used in another java code.

Q.5 (b) (Diagram 2-marks, Explanation 6-marks)

The life cycle of applet includes following state:
1. Born or initialization states
2. Running state
3. Idle state
4. Dead or destroyed state



1. Initialization State: Applet enters the initialization state when it is first loaded. This is achieved by init() method of Applet class. In this method initial values can be set, images & fonts can be loaded. Colors can be set.
Syntax of init() is

```
public void init()

{

(Action)

}
```

2. Running State: Applet enters running state when the system calls start() method of Applet class. This occurs automatically after the applet is initialized. It can be invoked also if the applet is stopped. Unlike init(), start() can be called more than once,

Syntax of start() is

```
public void start()

{

 (Action)

}
```

3. Idle or stopped state: An applet becomes idle when it s stopped from running. Stopping occurs automatically when we leave the page containing the currently running applet. We can also do so by calling stop() method explicitly.

Syntax of stop() is

```
public void stop()

{

(Action)

}
```

4. Dead state: An applet is said to be dead when it is removed from memory. This occurs automatically by invoking destroy() method, when we quit from the browser. Like initialization, destroying occurs only once in the applet's life cycle. destroy() can be overridden to clean up resources created by applet.

Syntax of destroy() is

```
public void destroy()

{

(Action)

}
```

5. Display State: Applet moves to display state whenever it has to perform some output operation on the screen. This happens immediately after the applet enters into the running state. The paint() method is called to accomplish this task. Almost every applet will have paint() method.

Syntax of paint() is

public void paint(Graphics g)

{

(display statements)

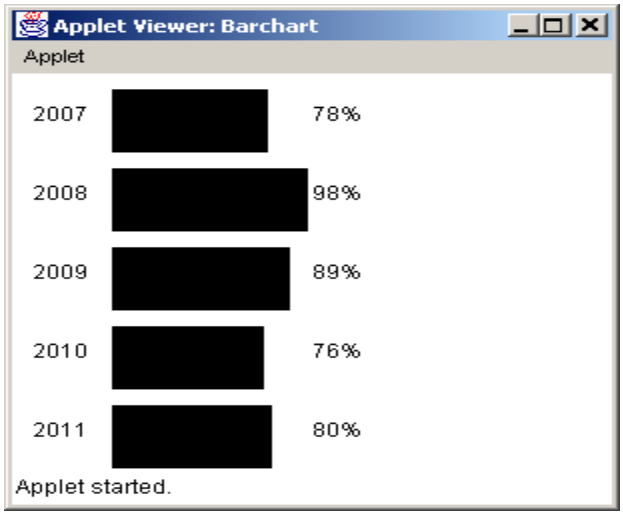
}

Q.5 (c)(Logic 4-marks, Syntax 4-marks)

```
import java.awt.*;
import java.applet.*;
public class Barchart extends Applet
{
        String label[];
        int value[];
        public void init()
        {
                try
                {
                label= new String[5];
                value= new int[5];
                label[0]=getParameter("yr1");
                label[1]=getParameter("yr2");
                label[2]=getParameter("yr3");
                label[3]=getParameter("yr4");
                label[4]=getParameter("yr5");
                value[0]=Integer.parseInt(getParameter("per1"));
                value[1]=Integer.parseInt(getParameter("per2"));
                value[2]=Integer.parseInt(getParameter("per3"));
                value[3]=Integer.parseInt(getParameter("per4"));
                value[4]=Integer.parseInt(getParameter("per5"));
                }
                catch(Exception e)
                {}
        }
```

```
public void paint(Graphics g)
        {
                for(int i=0;i<5;i++)
                {
                 g.setColor(Color.black);
                 g.drawString(label[i],10,i*50+30);
                 g.fillRect(50,i*50+10,value[i],40);
                String s=Integer.toString(value[i])+"%";
                 g.drawString(s,150,i*50+30);
                }
        }
}
/*
<applet code=Barchart width=300 height=250>
<param name="per1" value="78">
<param name="per2" value="98">
<param name="per3" value="89">
<param name="per4" value="76">
<param name="per5" value="80">
<param name="yr1" value="2007">
<param name="yr2" value="2008">
<param name="yr3" value="2009">
<param name="yr4" value="2010">
<param name="yr5" value="2011">
</applet>
*/
```

Output :



** Any other logic for drawing barchart can also be considered.

Q.6 (a)(Type of Errors & explanation 1-marks, throw, throws & finally 1-mark each)

Errors are broadly classified into two categories:-

1. Compile time errors

2. Runtime errors

Compile time errors: All syntax errors will be detected and displayed by java compiler and therefore these errors are known as compile time errors.

Runtime errors: Sometimes a program may compile successfully creating the .class file but may not run properly. Such programs may produce wrong results due to wrong logic or may terminate due to errors such as stack overflow. When such errors are encountered java typically generates an error message and aborts the program.

throw: If your program needs to throw an exception explicitly, it can be done using 'throw' statement. General form of throw statement is:

throw new Throwable subclass;

'throw' statement is mainly used in case there is user defined exception raised. Throwable instance must be an object of the type Throwable or a subclass of Throwable.

The flow of exception stops immediately after the 'throw' statement; any subsequent statements are not executed. The nearest enclosing 'try' block is inspected to see if it has a catch statement that matches the type of exception. If it does find a match, control is transferred to that statement. If not matching catch is found, then the default exception handler halts the program and prints the built in error message.

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

Type method-name (parameter list) throws exception list

{

// body of method

}

Here exception list can be separated by a comma.

finally: It can be used to handle an exception which is not caught by any of the previous catch statements. finally block can be used to handle any statement generated by try block. It may be added immediately after try or after last catch block.

Syntax

try OR	try
{	{
……	……
}	}
finally	catch()
{	{
….	….
}	}
catch()	finally()
{	{
…..	…..
}	}

Q.6 (b) (Differences 3-marks, Uses-1 mark)

Sr. No	String	StringBuffer
1.	String is a major class	StringBuffer is a peer class of String
2.	Length is fixed	Length is flexible
3.	Contents of object cannot be modified	Contents of can be modified
4.	Object can be created by assigning String constants enclosed in double quotes.	Objects can be created by calling constructor of StringBuffer class using ‘new’
5.	Ex:- String s=”abc”;	Ex:- StringBuffer s=new StringBuffer (“abc”);

Note: Any three differences

String: It is the most commonly used class which can be used to store string constants. String constants can be any characters from keyboard enclosed in double quotes. Strings can be used when data need not be altered.

StringBuffer: It is a peer class of string which can be modified in terms of length & contents.

Q.6 (c) (Explanation-2-marks, Example 2-marks)

In order to run a java applet in a browser it is necessary to have a webpage that references that applet. A web page is basically created with the help of HTML tags . HTML page contains different tags for different functionalities. To add an applet to HTML file, a pair of <Applet>……..</Applet> tags should be included in the body section of HTML page.

Minimum required attributes of <applet> tags are:

*Code:- specifies 'class' file of java applet.

*Height & Width:- specifies display area of applet output. It is in the form of pixels.

Other optimal attributes are:

Code base:- url where code resides

align – specifies alignment

Hspace, Vspace – Specifies left, right, top & bottom blank spaces.

Alt:- Specifies text when applet is not supported by browser.

There is one more tag called as <param> which can be included inside <applet>, if a parameter is to be sent from HTML page to an applet. It can be collected with getParameter() in an applet.

<Param Name="parameter name" value = "value">

Example:-

```
import java.awt.*;

import java.applet.*;

public class applettest extends Applet

{

String str;

public void init()

{

str=getParameter ("param1");

}

public void paint(Graphics g)

{

g.drawString("Hello" +str, 100,100);

}
```

HTML file

<HTML>

<Body>

<Applet Code=applettest width=200 Height=200>

<param name=”param1”,value=”ABC”>

</Applet>

</Body>

</HTML>

- An example can be without <param> tag also. Any other example can be considered.

Q.6 (d)(Logic 2-marks, Syntax 2-marks)

Program can be done by taking input from keyboard or can directly store value inside a variable.

In this program input is taken from keyboard using Buffered Reader.

```
// program to check whether the given number is divisible by 5 or not.
import java.io.*;
class test
{
        public static void main(String args[])
        {
        BufferedReader br=new BufferedReader( new InputStreamReader(System.in));
        int n;
                try
                {
                System.out.println("Enter a number :");
                n= Integer.parseInt(br.readLine());
                        if(n%5==0)
                        System.out.println(n + " is divisible by 5 ");
                        else
                        System.out.println(n + " is not divisible by 5 ");
                }
                catch(IOException e)
                {
                System.out.println("I/O error");
                }
        }
}
```

WINTER – 12 EXAMINATION

Q.6(e) (declaration 2-marks, Example 2-marks)

Use of Vector:- Java does not support the concept of variable arguments to a function. The feature can be achieved in java through the use of Vector class contained in java.util package. This class can be used to create a generic dynamic array known as Vector that can hold objects of any type & number. The objects do not have to be homogeneous. Arrays can be easily implemented as Vectors.

Declaration of vector object:-

Vector v1=new Vector(); → without size

Vector v1=new Vector(3); → with size

Example:-

```
//program to add 5 elements to a Vector and display them.
import  java.util.*;
class test1
{
        public static void main(String args[])
        {
        Vector v=new Vector();
        int l=args.length;
                for(int i=0;i<5;i++)
                {
                v.addElement(args[i]);
                }
                int size=v.size();
                System.out.println("Elements in vector are:");
                for(int i=0;i<size;i++)
                {
                System.out.println(v.elementAt(i));
                }
        }
}
```

** Any other example showing declaration and use of Vector can also be considered.