UNIT 2

PACKAGES, INTERFACES AND STREAM BASED I/O (java.io)

PART-A
SHORT QUESTIONS WITH SOLUTIONS

Q1. Define a Package? What is its use in java? Explain.

Answer :

(Model Paper-I, Q1(c) | April/May-18(R16), Q1(c))

Package

Package is a mechanism that organizes classes and interfaces. It groups them based on their functionality. It acts as a container for the classes. Packages are of two types namely Java API Package and user-defined packages. Java API consists of various packages which contain classes. Packages involve in access control mechanism of Java. A class which is define in a package will be private to the package only. It cannot be accessed by a code which is beyond the package.

Syntax

```
access_specifier interface interface_name
{
    //variables declaration;
    //methods declaration;
}
```

An interface is defined using the keyword "interface" followed by the interface name. The interface header defines scope and the interfaces that are extended. All the member declarations are included in the interface body.

Example

```
public interface student
{
    final int id = 20;
    final string name = John;
    void display( );
}
```

Q2. What is the significance of the CLASSPATH environment variable in creating/using a package?

Nov./Dec.-17(R16), Q1(d)

Answer :

'CLASSPATH' can be defined as a statement which gives path or location of the class which is being currently used. When a package is created by a programmer and if the programmer is trying to execute one of the classes in it, then it is important to know where the Java run-time system looks for the package.

In order to solve this problem, two concepts are being implemented which are as follows,

1. Execute in Same Package
2. Set the Appropriate Class Path.

Q3. What is an interface? Give example. Nov./Dec.-16(R13), Q1(d)

OR

Define interface.

Answer : Nov./Dec.-17(R13), Q1(c)

Interfaces

Interfaces are similar to that of classes. It is specifically used to implement the concept of multiple inheritance. It does not contain instance variables. Interface methods are abstract and contains information about name, parameters and return type. They do not have implementation details and are publicly by default. An interface is also allowed to contain methods and variables which are only abstract and final respectively. The class which use the interface are responsible for providing implementation code for these methods.

Example

```
Interface shape
{
public final static double pi = 3.14;
public abstract double computer (double x, double y);
}
class circle implements shape
{
public double computer (double x, double y)
{
return (pi * x * y);
}
}
```

Q4. Define stream.

Answer : Model Paper-I, Q1(d)

Stream is a flow of data between source (such as file on hard disk, keyboard etc.,) and destination (such as computer screen, file or socket) through a program. If the data flows from the source to a program it is called input stream. If the data flows from a program to destination, it is called output stream. They can be handled using the classes and interfaces provided by java.io package.

Q5. What are the methods available in the character streams?

Answer : Nov./Dec.-17(R16), Q1(c)

The methods of character streams are categorized into two classes namely reader and writer.

Methods of Reader Class

1. **int read( ):** This method is used to read the characters from input stream.
2. **int read(char[ ]c):** This method is used to read a set of characters from the input stream and store them in char array c.
3. **close( ):** This method is used to close the output stream and release the resources connected to it.

Methods of Writer Class

1. **Abstract void flush( ):** This method is used to flush the output stream by pumping out the buffered bytes that are to be written.
2. **void write(int c):** This method is used to write the characters to the output stream.
3. **void write(char[ ] arr):** This method is used to write a complete char array to the output stream.
4. **Abstract void close( ):** This method is used to close the output stream and release the resources connected to it.

Q6. List the class of reader.

Answer :

	Class	Description
1.	BufferedReader	It is used for reading the text from the character input stream and then buffering them for efficient reading.
	❖ LineNumberReader	It is a buffered character input stream used to keep track of the line numbers.
2.	CharArrayReader	It is used for implementing the character buffer which is used as character input stream.
3.	InputStreamReader	It is used to read bytes and decode them to characters by using a charset.
	❖ FileReader	It is used to read the character files.
4.	FilterReader	It is used to read the filtered character streams.
	❖ PushbackReader	It is a character stream reader that is used to allow the characters to be pushed back to the stream.
5.	PipedReader	It is a piped character input stream.
6.	StringReader	It is a character stream extracted from a string.

Q7. What is Console class? What is its use in java?

Answer : **Nov./Dec.-18(R16), Q1(c)**

The java.io.console class provides the methods to read a string from console and to write a string to the console. The functionality of the console is provided through the system.in and system.out. Therefore it is known as the convenience class. It is also used to implement the flushable interface. It uses simple methods to read the strings from console. The console objects are obtained/accessed by calling system console( ) instead of supplying constructors.

Static console console( )

The reference of the console will be returned if available. Otherwise it will automatically return a null value.

Incase of input errors one of the console method readLine( ) throws IOError. IOError is the subclass of Error and specifies the I/O errors of a program. It is not easy to catch an IOError. Since it is a type of failure that makes many users to suffer.

Q8. What is serializable interface?

Answer :

Objects of a class that implements Serializable interface can only be stored and retrieved i.e., a class that implements this interface is a serializable class. This interface does not define any methods. It is used to indicate that the objects of this class are serializable. Subclasses of a class that implements serializable interface are also serializable. Serialization does not store the variables that are declared as transient and static.

Q9. Define enumerated data type.

Model Paper-II, Q1(c)

Answer :

Enumerated Data Type

Enumerated data type can be defined as a data type which allows the user to create own data type and also define values to the variables of this type. It is a user-defined data type. An enumerated data type can be created by using a keyword 'enum'. The syntax of enumerated data type is as follows,

Syntax

```
enum datatype_name
{
    value1, value2, ...., valuen
};
```

Q10. What is the use of auto boxing in java? Explain.

Nov./Dec.-18(R16), Q1(d)

Answer :

Autoboxing

The automatic process of encapsulating or boxing a primitive type into its equivalent type wrapper (i.e., Double, Float, Long, Integer, Short, Byte, Character or Boolean), when its object is required is called autoboxing. In other words, autoboxing is an automatic conversion of primitive type into an object. It avoids wrapping up of primitive data types through manual or explicit construction objects. Instead, it allows to assign the value of a primitive type to the reference of a type wrapper.

Example

```
Integer itw = 100;                    //Autobox int to Integer
Instead of,
Integer itw = new Integer(100);       //Wrap into integer
```

Q11. Discuss in brief about generics.

Answer :

Generics

Generics is a concept in Java that enables a programmer to define and apply an algorithm irrespective of the data type. That is same algorithm can be applied multiple times on many data types without enforcing additional effort. Using generic it is possible to create classes, interfaces and method that can be executed in a type-safer manner independent of any specific type of data. Such methods/classes are referred as generic methods/classes. The major advantage of using generic is that it executes the program in type-safe mode. Moreover, generic expands the programmer ability in reusing the code.

Syntax

```
class className<Datatype.param 1, Datatype param 2,......Datatype param n)
{
//Statement
}
```

Now, the way of declaring reference to generic class is,

```
classname<datatype arg 1, datatype arg 2.....datatype arg n > variable name = new classname<datatype arg 1,
                                datatype arg 2.......datatype arg n>(cons arg 1, cons arg n);
```

Q12. How a generic method is created?

Answer :

Creating Generic Method

Method that is declared inside a generic class can be use the type parameters of the class, thereby making the method a "generic method". This method can be declared with at least one type parameter of the respective method itself. In addition to this, it is also possible to create a generic method which is defined within a non-generic class. The primary advantage of generic method is that, the method ensures type safety.

Syntax for Declaring Generic Method

```
<list_of_type parameters> return_type name_of_ method (list_of_parameters)
```

The list of type parameters are separated by comma and are placed before the return type.

Q13. List out the benefits of Stream oriented I/O.

Answer :

(Model Paper-II, Q1(d) | April/May-18(R16), Q1(d))

The various benefits of stream oriented I/O are as follows,

1. It provides a clean abstraction for complicated tasks.
2. User can also create the custom streaming interface that is fit to the data transfer requirements with the help of the content of filtered stream classes.
3. The working of Reader, Writer, InputStream and OutputStream classes will not be effected/changed even when the new classes are introduced further.

PART-B
ESSAY QUESTIONS WITH SOLUTIONS

2.1 PACKAGES

2.1.1 Defining a Package, CLASSPATH

Q14. Define a package. How can you create a package?

OR

Explain the process of defining and creating a package with suitable examples.

Nov./Dec.-17(R16), Q4(a)

Answer :

Package

Package is a mechanism that organizes classes and interfaces. It groups them based on their functionality. It acts as a container for the classes. Packages are of two types namely Java API Package and user-defined packages. Java API consists of various packages which contain classes. Packages involve in access control mechanism of Java. A class which is define in a package will be private to the package only. It cannot be accessed by a code which is beyond the package.

Creating a Package

A package is created or defined using 'package' keyword. The name of the package should be proceeded with keyword 'package' in its declaration. These packages are called user-defined packages.

Syntax

```
package package_name;
```

If the package statement is not used then the classname is included in the default package. Package are stored in directories that are contained in file system. All the classes of a file belong to the package contained in the file. Multiple packages can be created as follows,

```
package package1[.package2[.package3]];
```

Example

```
package mypack1;
public class Student
{
public static void main(String args[ ])
{
    System.out.println("Welcome to package program");
}
}
```

Output

```
c:\javapgm>javac mypack1/Student.java
c:\javapgm>java mypack1.Student
Welcome to package program
```

Q15. Discuss about CLASSPATH environment variables.

(Model Paper-I, Q4(a) | April/May-18(R16), Q5(a))

Answer :

CLASSPATH

'CLASSPATH' can be defined as a statement which gives path or location of the class which is being currently used. When a package is created by a programmer and if the programmer is trying to execute one of the classes in it, then it is important to know where the Java run-time system looks for the package. In order to solve this problem, two concepts are being implemented which are as follows,

1. **Execute in Same Package**

In this concept, the class must be stored in the same location of the package. The .class file also must be saved in the package and now if this program is executed then it gives a positive result.

Here, the file should be executed as,

```
java packagename.classname
```

2. **Set the Appropriate Class Path**

In this concept, once the class path is set then the program could be run independently without any hierarchy of top level packages.

Here, class path is the exact location of the package in which the class is available.

When the class path is set, the program could be run as,

java classname

Example

```
package cp;
import java.io.*;
class Test
{
      void display( )
      {
      System.out.println("WELCOME TO SIA PUBLISHERS AND DISTRIBUTORS");
      }
}
class Testcp
{
      public static void main(String args[ ])
      {
      Test t = new Test( );
      t.display( );
      }
}
```

Output

```
c:\javapgm>javac cp/Testcp.java
c:\javapgm>java cp.Testcp
WELCOME TO SIA PUBLISHERS AND DISTRIBUTORS
```

Here, 'cp' be a package.

'Test' and 'Testcp' are two classes of 'cp' package.

If you need to run Testcp class then type the following command,

java Testcp

This command line argument is correct only if class path is set. If the class path is not set, then the two classes Test and Testcp must be saved in cp package and .class file of Testcp class must be saved in cp package.

Now, the program can be run as,

java cp.Testcp

2.1.2 Access Protection, Importing Packages

Q16. Discuss the different level of access protection available in Java.

Answer :

(April/May-18(R16), Q2(b) | April/May-18(R16), Q5(b)

Java provides many levels of protections to the variables and methods to be visible which are present inside the classes, subclasses and packages. The classes and packages both encapsulates the name space, scope of variables and methods. A class consist of data and code whereas package consist of related classes and sub packages. Because of the interaction between classes and packages, Java offers the following four categories of visibility for accessing class members.

1. Subclasses in the same package
2. Non-sub classes in the same package
3. Sub classes in different packages
4. Classes that are neither in the same package nor in the subclass.

Many levels of access that are required by the above categories can be achieved by the following access specifiers,

1. Public
2. Private
3. Protected and
4. Default.

Classes/Access Specifier	Public	Private	Protected	Default (No Modifier)
Same class	Y	Y	Y	Y
Same package subclass	Y	N	Y	Y
Same package non-subclass	Y	N	Y	Y
Different package subclass	Y	N	Y	N
Different package non-subclass	Y	N	N	N

Table: Accessing Members of Class

In the above table,

Y – It denotes Yes

N – It denotes No.

Public

A member when declared as 'Public' can be accessed or used by the classes of the same package and also the classes of other packages. This member can be accessed by any package as it will not have any restrictions.

Private

A member when declared as 'Private' can only be accessed by the members of that class itself. This member will be restricted to the package in which it is declared.

Protected

A member when declared as 'Protected' will be restricted to the current package and as well as to the subclasses of the packages present in it.

Default (No Modifier)

A member when does not have any access specifier then it can said to be have default access. This member will be restricted to the package in which it is declared. So, this member will have public access within the same package and private access to other packages.

Q17. Explain how packages can be imported.

OR

Describe the process of importing and accessing a package with suitable examples.

Nov./Dec.-17(R16), Q5(b)

Answer :

Accessing a Package

Packages that are defined by the user can only be accessed using import statement. The members of a package such as class can be accessed from some other package by proceeding the classname with import keyword and package name. Syntax for accessing a class is as follows,

```
import package_name.classname
```

In the above statement, package_name is the name of the package and class name is the name of the class that is to be imported from that particular class. In case if the user want to import all the contents of a package then the following syntax is used.

```
import package_name .*;
```

The members declared as public and protected are only accessible in other packages. The private and protected members are not allowed to be accessed in other packages.

Example

```
package mypack;
class Student
{
    int sid=25;
}
import mypack.*;
class Details
{
    public static void main(String args[ ])
    {
        Student s=new Student( );
        System.out.println("Student ID:" +s.sid);
    }
}
```

Q18. What is the accessibility of a public method or field inside a non-public class or interface? Explain.

Nov./Dec.-17(R16), Q5(a)

Answer :

A public method or field defined inside a non public class is only accessible to the members of the package to which it belongs. They are accessible to some other access point i.e., a public interface or public superclass. The members of a class or interface are accessible only within the class or interface through which they are accessed. A protected or private class has the default package access so that the public methods of fields of that class are not visible out of the package.

In these two situations, the public class member has complete public access even if the class has package access. Initially, the public interface exposes the methods of package private class which implements the interface. Then the methods that implement the interface will be visible out of the package.

A parent class that is public containing public methods can also expose the methods of package-private subclass. If reference is used to access instance of subclass from the parent class then any public method of the parent class becomes accessible to all the classes. Through dynamic binding the actual method body that is invoked will belong to the package-private subclass.

2.2 INTERFACES

2.2.1 Defining an Interface, Implementing Interfaces

Q19. Discuss how interfaces are defined and implemented.

OR

How to design and implement an interface in Java? Give an example.

Answer :

Nov./Dec.-18(R16), Q4(a)

Defining an Interface

Interfaces are similar to that of classes. It is specifically used to implement the concept of multiple inheritance. It does not contain instance variables. Interface methods are abstract and contains information about name, parameters and return type. They do not have implementation details and are publicly by default. An interface is also allowed to contain methods and variables which are only abstract and final respectively. The class which use the interface are responsible for providing implementation code for these methods.

Syntax

```
access_specifier interface interface_name
{
    variables declaration;
    methods declaration;
}
```

An interface is defined using the keyword "interface" followed by the interface name. The interface header defines scope and the interfaces that are extended. All the member declarations are included in the interface body.

Example

```
public interface student
{
    final int id = 20;
    final string name = John;
    void display( );
}
```

Implementing an Interface

After an interface is defined it can be implemented by multiple interfaces. For this purpose, the class declaration must include implements clause followed by interface name. Then the class can implement the methods that are defined by the interface. A class can implement more than one interface at a time. Its declaration should include the interface names separated using comma's. The methods implementing the interface must be declared as public. The signature type of the implementing method must be same as the signature type in the interface definition.

Syntax

```
class class_names implements interface_name
{
    class body;
}
```

Example

```
interface student
{
    final int id = 20;
    final string name = John;
    void display( );
}
class DisplayDetails implements Student
{
    void display( )
    {
        System.out.println("id" +id + "Name" + name);
    }
}
class Demo
{
    public static void main(String args[ ])
    {
        Displaydetails d = new Displaydetails( );
        d.display( );
    }
}
```

Q20. What is an interface? What are the similarities between interfaces and classes?

Answer : (Model Paper-I, Q4(b) | April/May-18(R16), Q4(a))

Interface

For answer refer Unit-II, Q19, Topic: Defining an Interface.

Similarities between Interface and Class

1. Both interface and class can be inherited. Interface can be inherited by using "implements" keyword and class be inherited by "extends" keyword.
2. Interface and class do not have any default constructors.
3. Dynamic polymorphism can be possible with both interfaces and classes.
4. Both the interface and class can contain final, static variables and static method implementation.
5. User cannot create object for both interface and class.
6. The syntax of interface is similar to the class.
7. Interface and class both are used to create new reference type.

Q21. Write in detail about accessing implementations through interface references.

Answer :

Generally, object references are of class type. It is also possible to create object references of interface type. A variable is used as object reference. This variable refers to the instance of a class which implements the interface. When this reference is used to call a method, the exact version to be called depends upon the instance of the interface that is referred. The method that must be executed is identified dynamically at runtime. This will delay the creation of classes. The code which calls the methods on this class is executed first.

The caller does not need to have any knowledge about call. This process is similar to that of a superclass accessing a subclass object through reference.

Example

```
interface student
{
    void display( );
}
class Details implements student
{
    public void display( )
    {
        System.out.println("Hello");
    }
}
public class Demo
{
    public static void main(String args[ ])
    {
        student s= new Details( );
        s.display( );
    }
}
```

Describe the various forms of implementing interfaces. Give examples of java code for each case.

Nov./Dec.-16(R13), Q4(b)

OR

Give an example where interface can be used to support multiple inheritance.

(Refer Only Topic: Example (Full Implementation))

Answer :

Interface Implementations

(Nov./Dec.-18(R16), Q4(b) | Nov./Dec.-17(R16), Q4(b))

In Java, an interface can be implemented either fully or partially.

Full Implementation

When an interface is fully implemented, the implementing class defines all the methods of an interface. Thus, full implementation is nothing but the implementation of an interface.

After defining an interface any number of classes can implement that interface. To implement an interface 'implements' keyword is used by a class which provides implementation to methods defined in an interface.

General Form

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

Here, access specifier is either public or not used. A class can implement more than one interface and their names are separated with a comma. All the methods that are implemented from an interface should be declared as public so that they can be accessed by any other class and their names and return types should match with those methods specified in the interface definition.

An interface can extend any number of interfaces. By interfaces Java supports multiple inheritance.

public class Demo implements A, B, C

Where A, B, C are the 3 interfaces and each extends the other and the class Demo implements all these interfaces supporting multiple inheritance.

Example

```
interface Shape
{
    public final static double pi = 3.14;
    public abstract double compute(double x,double y);
}
class Circle implements Shape
{
    public double compute(double x, double y)
    {
        return(pi*x*y);
    }
}
class TestInterfaces2
{
    public static void main(String args[ ])
    {
        Circle c = new Circle( );
        double value=c.compute(20.5,10.0);
        System.out.println(value);
    }
}
```

Output

643.7

In the above example, Shape is an interface that defines pi value as final and a method compute( ), the body of this method given by the class Circle that implements the interface Shape and returns the value computed.

2. Partial Implementation

When an interface is partially implemented, the implementing class defines few or none of the methods of an interface. The class declares itself as abstract, so that its children implements all or remaining methods.

Example

```
interface Shape
{
    public static double pi = 3.14;
    public abstract double compute(double x, double y);
    public abstract void show( );
}
abstract class Circle implements Shape
{
    public double compute(double x, double y)
    {
        return(pi*x*y);
    }
    //The class does not implement show( );
}
class CircleDisplay extends Circle
{
    public void show( )
    {
    System.out.println("this is the show( ) in child class of Circle");
    }
}
class TestImplementation
{
    public static void main(String args[ ])
    {
        CircleDisplay c = new CircleDisplay( );
        double value = c.compute (20.5, 10.0);
        System.out.println(value);
        c.show( );
    }
}
```

Q23. Write a super class interface employee has name and id number. Write manager and labour derived from employee class. Manager class has member data function and qualification and manager allowance and rank. Labour class has member data daily wage, overtime and grade.

Answer :

```
interface Employee
{
    String name = " ";
    int id = 0;
    public abstract void getDetails( );
}
class Manager implements Employee
```

```
{
    String qualification;
    double allowance;
    String rank;
    String myname;
    int myid;
    public Manager(String name1, int id1, String qualification1, double allowance1, String rank1)
    {
        myname = name1;
        myid = id1;
        qualification = qualification1;
        allowance = allowance1;
        rank = rank1;
    }
    public void getDetails( )
    {
    System.out.println("Manager Particulars:");
    System.out.println("Name:" + myname );
    System.out.println("ID:" + myid);
    System.out.println("Qualification:" + qualification);
    System.out.println("Rank:" + rank);
    }
    public void getPayment( )
    {
        System.out.println("Payment Rs." + allowance);
    }
}
class Labour implements Employee
{
    double daily_wage;
    int overtime;
    int grade;
    String myname;
    int myid;
    public Labour(String name1, int id1, double daily_wage1, int overtime1, int grade1)
    {
        myname = name1;
        myid = id1;
        daily_wage = daily_wage1;
        overtime = overtime1;
        grade = grade1;
    }
    public void getDetails( )
```

```
        {
            System.out.println("\nLabour Particulars:");
            System.out.println("Name:"  + myname );
            System.out.println("ID:" + myid);
            System.out.println("Daily Wage Rs.:" + daily_wage);
            System.out.println("Grade:"  + grade);
        }
        public void getPayment( )
        {
            System.out.println("Payment Rs." + daily_wage * overtime);
        }
    }
    public class Info
    {
        public static void main(String args[ ])
        {
            Manager accountant = new Manager("S N Rao", 3519, "Accountant", 9999.99, "Managerial");
            accountant.getDetails( );
            accountant.getPayment( );
            Labour helper = new Labour("Sridhar", 5678, 56.50,  4, 3);
            helper.getDetails( );
            helper.getPayment( );
        }
    }
```

2.2.2 Nested Interfaces, Applying Interfaces

Q24. How interfaces are nested and applied? Explain.

Answer :

An interface when embedded within any other interface or a class will be declared as nested interface. It will be declared inside a class or an interface. These interfaces are used for grouping the related interfaces so that they can be maintained easily. They require to be referred to be referenced by the outer class or interface. Without this they cannot be accessed.

When the nested interface is declared inside an interface they should be public. When it is declared inside a class it can have any modifier. The outer interface can be either public or default access level. If the nested interface is used outside its scope that is out of the nesting interface or class then its name should be fully qualified. It should be qualified with the name of the interface or class that embeds it.

Example

```
import java.io.*;
import java.util.*;
interface Display
{
    interface Msg
    {
        void msg( );
    }
}
class nestedinterface implements Display.Msg
```

```
{
    public void msg( )
    {
        System.out.println("Hello");
    }
    public static void main(String[ ]args)
    {
        Display.Msg dm = new nestedinterface( );
        dm.msg( );
    }
}
```

2.2.3 Variables in Interfaces and Extending Interfaces

Q25. Explain the scope of variables in interfaces.

Answer :

Model Paper-II, Q4(a)

The variables defined in the interfaces are also known as constants. They are public, static and final by default. The public is only one which is allowed in the declaration of interface variable. The use of static and final for a variable indicates that the value of the variable cannot be changed through out the program. This variable can be accessed by making use of class name. An important point to note here is that, the variables need not be explicitly declared as static and final. This is because, both the keywords are default in declaration of interface variables.

The above concept can be illustrated with the help of a simple program,

```
import java.io.*;
interface details
{
public static final String name = 'Ravi';
public static final int id = 175;
void display( );
}
class Student implements details
{
public void display( )
{
System.out.println("The name of the student is:" + name);
System.out.println("The student Id is:" + id);
}
}
class College
{
public static void main(String args[ ])
{
Student s = new Student( );
s.display( );
}
}
```

Output

```
The name of the student is:Ravi
The student Id is: 175
```

For instance, if the student id is changed to 200 then the program will not be complied.

Example

```
class student implements details
{
    int id = 200;
    :
}
```

It is an invalid declaration.

Since, the id cannot be reassigned as it is declared static and final. Hence, it can be said that interface variables are final and static by default.

Q26. How interfaces can be extended? Explain. Nov./Dec.-17(R13), Q5(b)

OR

How can you extend one interface by the other interface? Discuss.

Answer : April/May-18(R16), Q4(b)

In Java, interfaces are extended using the "extends" keyword. The extending interface is called as subinterface and the interface that is being extended is called as superinterface. The subinterface can access all the members of its superinterface. This is done by using extends keyword in subinterface declaration. The class which implements the subinterface must provide the implementation for all the methods of both superinterface and subinterface.

Syntax

```
interface sub_interface_name extends super_interface
{
    body of sub_interface
}
```

Example

```
interface student
{
    set( );
}
interface Details extends student
{
    void display( );
}
class studentdetails implements Details
{
    set( )
    {
        int id = 20;
        string name = John;
    }
    void display( )
    {
        System.out.println("Id" + id + "Name" + name);
    }
}
class Demo
{
    public static void main(String[ ] args)
    {
        Details d = new Details( );
        d.set( );
        d.display( );
    }
}
```

Q27. What are the differences between an interface and class? Explain with suitable example.

Answer : Nov./Dec.-16(R13), Q4(a)

Differences between Interface and Class

	Class		Interface
1.	The definition of class include the methods, along with their code, despite the method is either abstract or non-abstract.	1.	The definition of interface include the methods that are abstract in nature. So, the methods does not contain any code and it can be written by the class while implementing the interface.
2.	The class members can either be constant or variable.	2.	The interface members can only be constant. The constant values are declared as final.
3.	The class can be used for the declaration of objects.	3.	The interface cannot be used for the declaration of objects. It can be used only for inheritance.
4.	The class can use access specifiers such as public, private or protected.	4.	The interface can only use the access specifier public.

5.	**Syntax**
	class class_name
	{
	declaration of variables;
	declaration of functions;
	declaration and definition of functions;
	}
		5.	**Syntax**
			access_specifier interface interface_name
			{
			variables declaration;
			methods declaration;
			}

Example

For answer refer Unit-II, Q22, Topic: Example (Partial Implementation).

2.3 STREAM BASED I/O (java.io)

2.3.1 The Stream Classes – Byte Streams and Character Streams

Q28. Explain the concept of streams and about stream classes and classification.

OR

Distinguish between Byte Stream Classes and Character Stream Classes.

(Refer Only Topics: Byte Streams, Character Streams)

Answer : Nov./Dec.-18(R16), Q5(b)

Stream

Stream is a flow of data between source (such as file on hard disk, keyboard etc.,) and destination (such as computer screen, file or socket) through a program. If the data flows from the source to a program it is called input stream. If the data flows from a program to destination, it is called output stream. They can be handled using the classes and interfaces provided by java.io package. The streams are classified into byte streams and character streams.

1. **Byte Streams**

The byte streams are used for reading and writing the individual bytes which represent the primitive data types. All the classes of byte stream are classified into inputStream and outputStream.

a) **InputStream**

It is the base class of all the input streams of Java IO API.

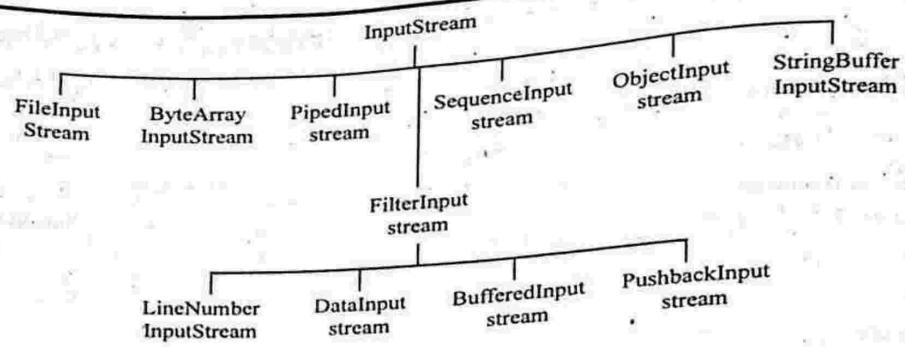

Figure: Classification of InputStream Class

	Class	Description
1.	FileInputStream	It is an InputStream used to read from a file.
2.	ByteArrayInputStream	It is used to read the data from byte arrays as streams and store them in a buffer.
3.	PipedInputStream	It is connected to pipedOutputStream. It is used to provide the data pipes that are to be written to the output stream.
4.	FilterInputStream	It contains and makes use of other input stream as the basic source of data. It provides additional functionality or transforms the data in its way.
	(a) LineNumberInputStream	It is used to keep track of the current line number.
	(b) DataInputStream	It is used to allow an application to read the Java primitive data types from underlying input stream.
	(c) BufferedInputStream	It is used to add the additional functionality to some other input stream. Additional functionality includes buffering the input and supporting mark and reset methods.
	(d) PushbackInputStream	It is used to add functionality such as push back or unread one byte to some other input stream.
5.	SequenceInputStream	It is used to represent the logical concatenation of some other input streams.
6.	ObjectInputStream	It is used to deserialize the primitive data and the objects that have been already written by object output stream.
7.	StringBufferInputStream	It is used by an application to create an input stream,which contains the bytes supplied by string contents.

(ii) OutputStream

It is the base class of all the output streams of java I/O API.

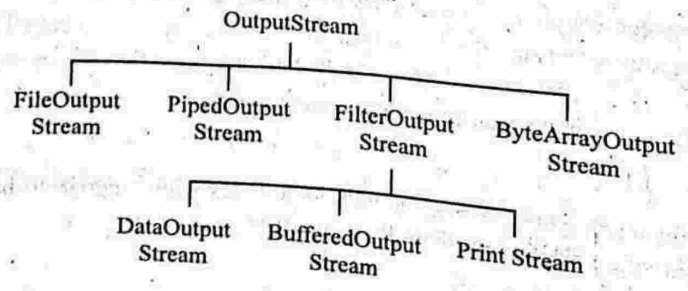

Figure: Classification of OutputStream Class

	Class	Description
1.	FileOutputStream	It is used for writing the data to a file or to file descriptor.
2.	PipedOutputStream	It is used by an object to send data through it.
3.	FilterOutputStream	It is the base class of the output stream classes that filter the output streams.
	(a) DataOutputStream	It is used to allow an application to write the primitive data types to output stream.
	(b) BufferedOutputStream	It is used to implement a buffered output stream to which an application can write bytes by calling the system only once.
	(c) PrintStream	It is used to add functionality (such as to print various types of data values) to some other output stream.
4.	ByteArrayOutputStream	This class is used to implement an output stream in which the data is written to a byteArray.

Character Streams

For answer refer Unit-II, Q29.

Q29. Explain character stream classes. **Model Paper-II, Q4(b)**

OR

What are the methods available in the Character Streams? Discuss.

(Refer Only Topic: Methods in Character Streams)

Answer : **Nov./Dec.-18(R16), Q5(a)**

Character Streams

Character streams are used for performing the input and the output operations on 16-bit unicode characters. All the classes are classified into reader and writer.

1. Reader

It is the base class of all the character input streams.

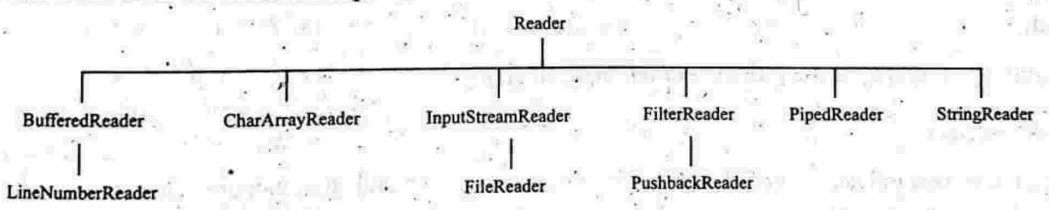

Figure: Classification of Reader Class

	Class	Description
1.	BufferedReader	It is used for reading the text from the character input stream and then buffering them for efficient reading.
	❖ LineNumberReader	It is a buffered character input stream used to keep track of the line numbers.
2.	CharArrayReader	It is used for implementing the character buffer which is used as character input stream.
3.	InputStreamReader	It is used to read bytes and decode them to characters by using a charset.
	❖ FileReader	It is used to read the character files.
4.	FilterReader	It is used to read the filtered character streams.
	❖ PushbackReader	It is a character stream reader that is used to allow the characters to be pushed back to the stream.
5.	PipedReader	It is a piped character input stream.
6.	StringReader	It is a character stream extracted from a string.

2. Writer

It is the base class of all the output streams.

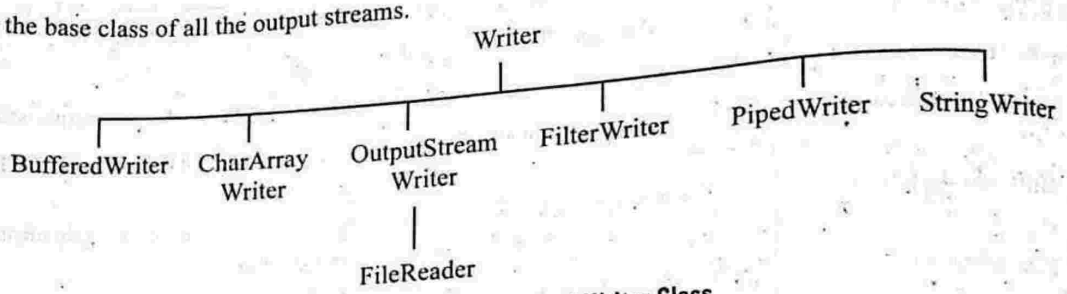

Figure: Classification of Writer Class

	Class	Description
1.	BufferedWriter	It is used to write text to character output stream by buffering the characters for efficient writing.
2.	CharArrayWriter	It is used to implement a character buffer that is used as an writer.
3.	OutputStreamWriter	It is used to encode the characters into bytes usign a charset.
	❖ FileWriter	It is used for writing character files.
4.	FilterWriter	It is used for writing filter character streams.
5.	PipedWriter	It is used as a piped character output stream.
6.	StringWriter	It is a character stream that extracts the output into a string buffer that is used for building a string.

Methods in Character Streams

The methods of character streams are categorized into two classes namely reader and writer.

Methods of Reader Class

1. **int read( )**

   This method is used to read the characters from input stream.

2. **int read(char[ ]c)**

   This method is used to read a set of characters from the input stream and store them in char array c.

3. **close( )**

   This method is used to close the output stream and release the resources connected to it.

Methods of Writer Class

1. **Abstract void flush( )**

   This method is used to flush the output stream by pumping out the buffered bytes that are to be written.

2. **void write(int c)**

   This method is used to write the characters to the output stream.

3. **void write(char[ ] arr)**

   This method is used to write a complete char array to the output stream.

4. **Abstract void close( )**

   This method is used to close the output stream and release the resources connected to it.

2.3.2 Reading Console Input and Writing Console Output

Q. Explain the process of reading console input using byte streams.

Answer :

The object which can be used to read console input using byte streams is system.in. As a matter of fact, an input can be read using both byte streams and character streams. However, byte streams can be used to manipulate simple programs, applications dealing with keyboard input and sample programs. The various methods present in InputStream class are as follows,

int available( ): This method can return the count of bytes presently available for reading.

void close( ): This method closes the input and if further attempts can be performed to read input, this method will generate an IOException.

void mark(int numBytes): This method places a mark at current input value in input stream and that mark can be valid until the specified numBytes are read.

boolean markSupported( ): This method returns true if the methods mark( ) or reset( ) can be supported by calling stream.

int read( ): This method returns an integer which represents the next input present and returns – 1, if end of stream is encountered.

int read(byte[ ] buffer): This method reads buffer.length bytes into buffer and returns the count of bytes read successfully. If end of stream is encountered, this method returns – 1.

int read(byte [ ] buffer, int offset, int numBytes): This method reads numBytes number of bytes into buffer by initiating at buffer [offset] and returns the count of bytes read successfully. If the input stream is completed, this method returns – 1.

void reset( ): This method resets the input pointer to the mark which can be set previously.

long skip(long numBytes): This method skips or ignores the numBytes, bytes while reading input and also returns the count of bytes that can be ignored.

These methods can be accessed by using the system.in object which can be created as an instance of InputStream class. The most commonly used method in Input Stream class is read( ) which can read the bytes.

The three ways in which read( ) method can be defined are as follows,

int read( ) throws IOException

int read(byte[ ] buffer) throws IOException

int read(byte[ ] buffer, int offset, int numBytes) throws IOException.

While performing read operation using system.in, if enter button is pressed, then it generates an end-of-stream. If an exception generated, this read( ) method can throw IOException.

Example Program

```
import java.io.*;
class ConsoleInputExample
{
    public static void main(String[ ] args) throws IOException
    {
    byte[ ]info = new byte[50];
    System.out.println("Enter a string");
    int val = System.in.read(info);
    System.out.println("The String is");
    int x;
    for (x = 0; x< val; x ++)
    System.out.print((char) info[x]);
    }
}
```

Q31. Describe about the concept of writing console output in byte streams.

Answer :

The console output can be written using byte stream. system.out. However, System.out can also write console output. The console output commonly uses two methods such as print( ) and println( ) which can be defined in printStream class. This class can be derived from another class called as OutputStream. The OutputStream class consists of various methods. They are as follows,

(i) **void close( ):** This method closes the output stream and if further attempts are performed to write data, this method generates an IOException.

(ii) **void flush( ):** This method can be used to flush or delete data from the output buffer.

(iii) **void write(byte[ ] buffer):** This method writes an array of bytes from buffer to output stream.

(iv) **void write(int b):** This method writes one character into output stream.

(v) **void write(byte[ ] buffer, int offset, int numBytes):** This method writes an array of numBytes bytes from buffer to output stream by starting at buffer[offset].

Among these methods, the OutputStream class uses the write( ) method to write console output. This method can be defined in three ways. They are as follows,

(a) void write(int x)

(b) void write(byte[ ] buffer)

(c) void write(byte[ ] buffer, int offset, int numBytes)

Example Program

```
class consoleOutputExample
{
    public static void main(String[ ] args)
    {
        int x;
        x = 'N';
        System.out.print("\n");
        System.out.println(x);
    }
}
```

2.3.3 File Class

Q32. Explain about file class.

Answer : Model Paper-I, Q5(a)

File Class

The file class interact directly with files and the file system, unlike other classes that operate on streams. An object of file class contains the information about a disk file such as type of file, permissions, created data, modified data, full directory path.

Constructors

- **File(String path):** This constructor creates the file object from the path name of the file.
- **File(String path, String fname):** This constructor creates file object from a path and file name.
- **File(File dirObject, String fname):** This constructor creates the file object from the given directory and file name.
- **File(URI uriobject):** This constructor creates file object from the file specified by uriObject.

Methods

File class defines the following methods to manipulate file information:

- **String getName( ):** Returns the name of the file or directory.
- **String getParent( ):** Returns the parent directory name

boolean exists( ): Returns true if this file exists otherwise returns false.

boolean isFile( ): Returns true if the invoking object is a file. Otherwise returns false. It returns false if file is not a normal file.

boolean isDirectory( ): Returns true if the file denoted by object is a directory. Otherwise returns false.

boolean renameTo(File newfname): Renames the file with newfname and returns true if the file is successfully renamed. Otherwise returns false.

boolean delete( ): Delete the file or the directory. Directory is deleted if and only if it is empty. It returns true if the file is deleted. Otherwise it returns false.

String[ ] list( ): Returns a string array consisting of a list of files and directories present in a directory denoted by file object.

boolean mkdir( ): Creates a directory with the name specified by file object and returns true if directory is created. Otherwise returns false.

String getAbsolutePath( ): Returns the absolute path of a file or directory denoted by file object.

boolean canRead( ): Determines whether a file has read permission or not. Returns true if the file is readable. Otherwise returns false.

boolean canWrite( ): Determines whether a file has write permission or not. Returns true if file is writable. Otherwise returns false.

mple

```
import java.io.*;
class UsingFile
{
public static void main(String args[]) throws IOException
{
String s;
File f = new File("c:/mydir/text.txt");
s = f.exists( )? "File exists" : "file does not exist";
System.out.println(s);
s = f.isFile( )? "Is a normal file" : "Is a special file";
System.out.println(s);
File f1 = new File("c:/mydir/subdir");
f1.mkdir( );
if(f1.isDirectory( ))
{
    System.out.println("Directory name:" +f1.getName( ));
    System.out.println("Directory contents:");
    String str[ ] = f1.list( );
    if(str.length==0)
    System.out.println("Directory is empty");
    else
    for(int i = 0; i<str.length; i++)
        System.out.println(str[i]);
    }
else
System.out.println("Is not a directory");
}
}
```

ut

```
File exists
Is a normal file
Directory name:subdir
Directory contents:
Directory is empty
```

2.3.4 Reading and Writing Files

Q33. Explain how an input can be read from file using byte streams.

Answer :

The file input can be read using the following steps,

Step1

Initially, the file must be opened by creating an instance to FileInputStream class. This can be done by using the following constructor,

Syntax

FileInputStream(Stream filename) throws FileNotFoundException

Here, filename is the name of required file which must be opened. If the specified file is not available, then FileNotFoundException can be thrown which can be called as a subclass of IOException.

Step2

Next, the opened file can be read using read( ) method. This method can be declared as follows,

Syntax

int read( ) throws IOException

Here, the read( ) method reads a single byte from the file and returns an integer value. If the data in file is completed, this method returns – 1. If an exception is generated, this method can throw an IOException. This read( ) method is similar with the read( ) method which can be used in reading console input

Step3

After the completion of reading data from file, the file must be closed. This can be done using close( ) method. The syntax to declare close( ) method is as follows,

void close( ) throws IOException

After closing the file, the system resources allocated to the file can be released and allows other file to use these resources. If the file cannot be closed after the completion of task, then the resources remain allocated to the file and these resources cannot be used by other files.

Example Program

```
import java.io.*;
class ReadInputExample
{
    public static void main(String[ ] args)
    {
        int x;
        FileInputStream objfis = null;
        if(args.length ! = 1)
        {
            System.out.println("The file is :");
            return;
        }
        try
        {
            File f = new File("C:/example/sia.txt");
            objtis = new FileInputStream(f);
            do
            {
                x = objfis.read( );
                if(x! = – 1)
                System.out.print((char)x);
            } while(x! = – 1);
```

```
            System.out.println("File read successfully");
        }
        catch(FileNotFoundException ex)
        {
            System.out.println("File does not exist");
        }
        catch(IOException ex)
        {
            System.out.println("I/OException raised");
        }
        finally
        {
            try
            {
                fin.close( );
            }
            catch(IOException ex)
            {
                System.out.println("Error while closing file");
            }
        }
    }
}
```

Q34. Discuss the concept of writing data into file using byte streams.

Answer :

The data can be written into file using byte streams. This can be done by satisfying the below steps,

Step1

Initially, the file must be opened for output. This can be done by creating an object to FileOutputStream class. The two constructors that can be used to declare FileOutputStream class are as follows,

Syntax

(a) FileOutputStream(String fileName) throws FileNotFoundException

(b) FileOutputStream(String fileName, boolean append) throws FileNotFoundException

Here, filename is the name of required file in which the data must be written. If append variable is initialized with true, then the new data can be written after the end of file. If it is false, then new data can be overwritten with existing content. If the name of the file similar with existing file, then the existing file can be deleted and a new file with specified name can be created. If the file cannot be created, then FileNotFoundException can be thrown.

Step2

Next, the data can be written into the file using write( ) method. The declaration of write( ) method is as follows,

Syntax

void write(int x) throws IOException

This method can write the bytes into file specified by x. Even though the variable x can be declared as integer, only the low order 8 bits can be written into the file. If an error is raised while writing data into file, an IOException can be thrown.

Step3

After the successful completion of writing data into file, then the file must be closed. This can be done using following method.

Syntax

void close( ) throws IOException

Here, the close( ) method release all system resources and permits other files to use these resources. This method ensures that the output in buffer is written.

Example Program

```
import java.io.*;
public class WriteFileDemo
{
    public static void main(String[ ] args)
    {
        FileOutputStream objfos = null;
        File f;
        String data = "This is text";
        try
        {
            f = new File("e:/example/newfile.txt");
            objfos = new FileInputStream(f);
            if(!f.exists( ))
            {
                f.createNewFile( );
            }
            byte[ ] data1 = data.getBytes( );
            objfos.write(data);
            objfos.flush( );
            objfos.close( );
            System.out.println("Data written successfully");
        }
        catch(I/OException ex)
        {
            ex.printStackTrace( );
        }
        finally
        {
            try
            {
                objfos.close( );
            }
        }
        catch(IOException ex)
        {
            System.out.println("File cannot be closed");
            ex.printStackTrace( );
        }
    }
}
```

2.3.5 Random Access File Operations

Q35. Discuss the concept of Random access file. Also explain its operations.

Answer :

Model Paper-II, Q5(a)

Random Access

The Random Access File can be defined as a class which can access the files called as random access files. It is encapsulated a RandomAccessFile class. The content present in random access file can be accessed in required order. The Random Access File class cannot be defined from Input Stream or Output Stream classes. However, Random Access File class can implement DataInput and DataOutput interfaces that can perform basic I/O operations. While accessing the random access file, file pointer can be used to represent the content of file. When the random access file is opened, the file pointer represents the beginning of the file and location is represented as 0. If read or write operations is performed on random access file, then the position of file pointer moves based on the number of bytes read or written. The Random Access File is defined in Java.io. Random Access File packages and this package can be inherited from Java.lang.object package. Random Access File class provides different modes to access the file. They are as follows,

(i) **r:** This mode used to read the file but not to write in the file.

(ii) **w:** This mode can be used only to write data into it.

(iii) **rws:** This mode can perform both read and write operations. It also writes all the updated changes of read and write operations of both metadata and data synchronously to the device.

(iv) **rwd:** This mode can often perform both read and write operations and also writes all the updates of data into the device synchronously.

Constructors of random AccesFile are as follows,

RandomAcessFile (File fo, String access) throws FileNotFoundException

RandomAccessFile(String fn, String access) throws FileNotFoundException.

Operations

On Random Access File, various operations can be performed using different methods. Some of them are as follows,

(a) **void close( ):** This method can be used to perform close operations on random-access-file.

(b) **filechannel getIsChannel( ):** This method is used to return the file channel corresponding to random access file.

(c) **filedescriptor getFD( ):** This method is used to return the file descriptor object on to the random access files.

(d) **long getFilePointer( ):** This method is used to return the current position of file pointer.

(e) **long length( ):** This method is used to get the size of the random access file.

(f) **int read( ):** This method is used to perform read operations on random access file.

(g) **int read(byte[ ]b):** This method is used to perform read operation on file. During operations an array of bytes with size b.length can be read from the file.

(h) **int read(byte[ ], int offset, int length):** This method can be used to read an array of bytes with specified length by starting at the position offset.

(i) **string readLine( ):** This method can read the next line from random access file.

(j) **void seek(long pos) Throws IOException:** This method can be used to set the position of file pointer using pos variable and performs next read or write operations.

(k) **void setLength(long newLength) throws IOException:** This method can be used to set the length of random access file using new length variable.

(l) **void write(byte[ ]b):** This method writes an array of bytes into file specified length and by starting at file pointer.

(m) **void write(byte[ ] b, int offset, int length):** This method write an array of bytes with specified length by starting at offset into the file.

Among all these methods, seek( ) method is commonly used in Random Access File class.

2.3.6 The Console Class

Q36. Discuss about the console class in java.

OR

Discuss about Console class. Nov./Dec.-17(R13), Q4(a)

Answer :

The java.io.console class provides the methods to read a string from console and to write a string to the console. The functionality of the console is provided through the system.in and system.out. Therefore it is known as the convenience class. It is also used to implement the flushable interface. It uses simple methods to read the strings from console. The console objects are obtained/accessed by calling system console( ) instead of supplying constructors.

Static console console( )

The reference of the console will be returned if available. Otherwise it will automatically return a null value.

Incase of input errors one of the console method readLine( ) throws IOError. IOError is the subclass of Error and specifies the I/O errors of a program. It is not easy to catch an IOError. Since it is a type of failure that makes many users to suffer.

The methods like readPassword( ) will read the password string without displaying it on the console arrays containing the user string and password should not be made zeroout. This will cut down the chance that a malicious software will be able to retrieve the password. Different methods that are defined by a console are as follows,

Method name	Description/purpose
Reader reader( )	It returns the reference to Reader object associated with the console.
PrintWriter writer( )	It returns the writer object associated with the console.
String readLine( )	It reads and returns the text from the input devices i.e., keyboard. A null will be returned at the end of the console input stream. Upon failure an IOError is thrown.
String readLine(String.Fmtstring.object args)	It shows the prompting formatted string specified by the fmtstring and args. It reads from and returns to the console null will be returned at the end of the string.
Console printf(String fmtstring, object args)	It writes the args to the console output by using the format mentioned by fmtstring.
Console format(String fmtstring, object...args)	It writes the formatted args/string to the console by using the specified format, say fmtstring.
void flush( )	It writes the buffered output to the console.
char[ ] readPassword( )	It reads a string but does not display on the console. A null will be returned at the end of the string.
char[ ] readPassword(string fmtstring, object...args)	It shows the prompting formatted string specified by the fmString and args. It reads the string that it does not display on the console. A null will be returned at the end of the string.

A sample program to illustrate the console class is as follows,

```
//console class
import java.io.console*;
class consoleEx
{
    public static void main(String args[ ])
    {
        string strng;
        console c;
```

```
        c = System.console( );
        if(c == null) return;
        strng = c.readLine("Enter the string:");
        c.printf("The string is: %s\n", strng);
    }
}
```

Output

```
Enter a string : Java console class
The string is : Java console class.
```

2.3.7 Serialization, Enumerations

Q37. What is serialization? Describe the interfaces and classes that support serialization.

Answer :

Serialization

Serialization is the process of storing object's state. Where an object's state is nothing but the values of the instance variable. The objects may have references to some other objects or to themselves forming a directed object graph. Serializing an object serializes all the other referenced objects in object graph recursively. Deserialization is the process of retrieving the object's state. Deserializing an object restores all of these objects and their references correctly.

Serialization is important if you want to store the state of your program in a file and then restore it at a later time using deserialization. Serialization is extensively used in Remote Method Invocation (RMI). In RMI the object of one class on one machine invokes the method of another class on a different machine. Here two sending machine serializes the object and transmits it. So that it could be reconstructed on a different machine in the same state as it was existing before on the original machine. The receiving machine deserializes the object to restore its state.

Interfaces and Classes that Support Serialization

The interfaces and classes that support serialization are described below.

(a) Serializable Interface

Objects of a class that implements Serializable interface can only be stored and retrieved i.e., a class that implements this interface is a serializable class. This interface does not define any methods. It is used to indicate that the objects of this class are serializable. Subclasses of a class that implements serializable interface are also serializable. Serialization does not store the variables that are declared as transient and static.

(b) Externalizable Interface

Externalizable interface gives the programmer control over saving and restoring the state of an object. This interface defines two methods.

```
void readExternal(ObjectInput in)
void writeExternal(ObjectOutput out)
```

Here 'in' is an object of byte stream to read the object from it 'out' is an object of byte stream to write the object to it. These two methods throws an IOException. The readExternal( ) method may also throw ClassNotFoundException.

(c) DataOutput Interface

The DataOutput interface defines several methods to convert data from any of the Java's primitive types to a series of bytes and write these bytes to a binary output stream. It also defines the methods that convert a string into a Java modified UTF-8 format and write these bytes to a binary output stream. The DataOutput interface defines the following methods.

For a byte or for a byte array it defines the overloaded versions of write( ) method.

For unicode strings, it defines writeBoolean( ), writeByte( ), writeBytes( ), writeChar( ) and writeChars( ) methods.

To display modified text for unicode, it defines writeDouble( ), writeFloat( ), writeInt( ), writeLong( ), writeShort( ) and writeUTF( ) methods.

(d) ObjectOutput Interface

The ObjectOutput interface is derived from DataOutput interface. This interface defines following methods to support object serialization. These methods throws an IOException on any error.

Methods

void close( ) – Closes the stream.

void flush( ) – Flushes the stream.

void write(byte buff[ ]) – Writes the contents byte array buff to the invoking stream.

void write(byte buff[ ], int index, int nBytes) – Writes the portion of byte array buff to the invoking stream.

void write(int b) – Writes a single byte.

void writeObject(Object object) – This is an important method used to serialize an object. This method writes an object "object" to the invoking stream.

(e) ObjectOutputStream Class

The ObjectOutputStream class is derived from the OutputStream class and ObjectOutput interface. This class is used for writing objects to a stream. It has the following constructor.

ObjectOutputStream(OutputStream out).

This constructor throws an IOException. It defines the following methods. All of the methods throws an IOException on any error.

void close( ) – Closes the stream.

void flush( ) – Flushes the stream.

void write(int b) – Writer a single byte.

void writeBoolean(boolean bool) – Writes a boolean.

void writeChar(int ch) – Writes a single character.

(f) ObjectInput Interface

The ObjectInput interface is derived from DataInput interface. This interface defines the following methods to support object serialization. These methods throws an IOException on any error.

int available( ) – Returns the number of bytes available for reading.

void close( ) – Closes the stream.

int read( ) – This method reads and returns the integer representation of the next available byte from input stream.

int read(byte buff[ ]) – This method reads buff.length bytes from the input stream into byte array buff[ ]. On success it returns the number of bytes actually read.

int read(byte buff[ ], int index, int nBytes) – This method reads nBytes bytes into byte array buff from the input stream. On success it returns the number of bytes actually read.

The read( ) method returns – 1 if the end of file is reached.

Object readObject( ) – This is an important method used to deserialize an object. This method reads an object from the input stream.

long skip(long nBytes) – Skips the specified number of bytes and returns actual number of bytes skipped.

(g) ObjectInputStream Class

The ObjectInputStream class is derived from the InputStream class and ObjectInput interface. This class is used for reading objects from a stream. It has the following constructor.

Constructors

ObjectInputStream(InputStream in)

This constructor may throw IOException and StreamCorrupted Exception.

ObjectInputStream class defines the following methods. All of these methods throws an IOException on error.

Methods

int available( ) – Returns the number of bytes available for reading.

void close( ) – Closes the stream.

int read( ) – Reads and returns the integer representation of next available byte.

boolean readBoolean( ) – Reads a boolean value.

char readChar( ) – Reads a single character.

final Object readObject( ) – Reads an object.

38. Define enumeration. Also, explain enumerated/enum data type with an example.

Answer :

Enumeration

An enumeration is basically an ordered list of named constants. In Java, enumeration also corresponds to a class type having its own constructors, methods and instance variables.

Enumerated Data Type

Enumerated data type can be defined as a data type which allows the user to create own data type and also define values of the variables of this type. It is a user-defined data type. An enumerated data type can be created by using a keyword 'enum'. The syntax of enumerated data type is as follows,

Syntax

```
enum datatype_name
{
    value1, value2, ...., valuen
};
```

Here, enum is a keyword, data type_name is the name of the data type depending on user requirement and value1, value2, ..value n are the possible values of this data type. These values are called as enumerators or enumeration constants.

Example

```
public enum month
{
    JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER,
    NOVEMBER, DECEMBER
}
```

Here month is name of enumerated data type and JANUARY, FEBRUARY, ... DECEMBER are called enumerators.

The procedure for declaring a variables of enum_type is same as that of the primitive types, as shown below,

Syntax

```
enum_type variable;
```

Example

```
Months m;
```

The variable of type enum_type can be assigned only those values as defined by the enumeration. For example, the value of *m* can be either Jan, Feb, Mar, ... or Dec. The values are assigned to the enum variables using dot operator.

Syntax

```
enum_var = enum_type.value;
```

Example

```
m = Months.Mar
```

be controlled using an enum value by making all the constant values of an enum as case statements.

```
switch(m)
{
    case Jan: :
    case Feb: :
    case Mar: :
        :
}
```

The enum constants used in the case statement need not be qualified by their enumeration type name as Months. Jan or Months.Feb. Because, the enum variable 'm' used in the switch expression provides the enum type for all its *case* constants.

The value of an enum constant can be displayed using a println( ) statement by specifying its qualified name. That is, enum_type.value.

Example

The statement,

System.out.println("Current Month:" + Month.Mar);

prints *March*

Program

```
public enum month
{
    January, February, March, April, May, June, July, August, September, October, November, December
};
public class EnumExample
{
    Month mnth;
    public EnumExample(Month mnth)
    {
        this.mnth = mnth;
    }
    public void Season( )
    {
        switch(mnth)
        {
            case: 'January':
                System.out.println("Winter");
                break;
            case: 'May':
                System.out.println("Summer");
                break;
```

```
            case: 'August':
                System.out.println("Rainy");
                break;
            default:
                System.out.println("Spring");
        }
    }
    public static void main(String[ ] args)
    {
        EnumExample firstmonth = new EnumExample(Month.January);
        firstmonth.Season( );
        EnumExample fifthmonth = new EnumExample(Month.May);
        fifthmonth.Season( );
        EnumExample eighthmonth = new EnumExample(Month.August);
        eighthmonth.Season( );
    }
}
```

2.3.8 Autoboxing, Generics

Q39. Define autoboxing and auto unboxing. Explain how they occur in methods and expressions.

Answer : Model Paper-I, Q5(b)

Autoboxing

The automatic process of encapsulating or boxing a primitive type into its equivalent type wrapper (i.e., Double, Float, Long, Integer, Short, Byte, Character or Boolean), when its object is required is called autoboxing. In other words, autoboxing is an automatic conversion of primitive type into an object. It avoids wrapping up of primitive data types through manual or explicit construction objects. Instead, it allows to assign the value of a primitive type to the reference of a type wrapper.

Example

```
Integer.itw = 100;                    //Autobox int to Integer
```

Instead of,

```
Integer itw = new Integer(100);       //Wrap into integer
```

Auto-Unboxing

The automatic process of extracting or unboxing the value of an encapsulated or a boxed object from a type wrapper, when its object is required is called auto-unboxing. In other words, auto-unboxing is an automatic conversion of an object into a primitive type. It avoids the extraction of value from the type wrapper through intValue( ) or getvalue( ) methods. Instead, the object reference of a type wrapper can be assigned to a primitive type variable.

Example

```
int ip = itw;     //Autounboxing integer to int
```

Instead of,

```
int ip = itw.intValue( ); //Unboxing integer to int
```

1. Autoboxing/Unboxing in Methods

When an argument is passed to or returned from a method, the autoboxing/unboxing occurs. Consider the following example program,

```
class AutoBox_Unbox
{
    static int meth(Integer m)
    {
        return m;
//type wrapper reference 'm' is autounboxed to int
    }
    public static void main(String args[ ])
    {
        Integer itw = meth(300);
        System.out.println(itw);
    }
}
```

Output

```
300
```

In the above program, the meth( ) method of Autobox_Unbox class takes an integer value and returns an int value by performing auto-unbox. In the main( ) method, the int value 300 is passed as a parameter to meth( ). This method, first autoboxes it into an integer and later returns m by performing auto-unbox. The returned int value is then assigned to an object of integer, which autoboxes the int again into an integer.

2. Autoboxing/Unboxing in Expressions

In expressions, auto unboxing is performed on numeric objects. The result of an expression sometimes requires reboxing.

Consider the following example,

```
class AutoBox_Unbox_Exp
{
    public static void main(String args[ ])
    {
        int i;
        Double dtw = 33.33;
        Integer itw1, itw2, isw;
        itw1 = 299;
    System.out.println("values of itw1:");
    System.out.println("Initially:", +itw1);
    ++itw1; //Auto unbox then rebox
    System.out.println("After increment", +itw1);
    itw2 = (itw1 × 10) + itw1;
        //Auto unbox then rebox
    System.out.println("Value of itw2:", + itw2);
```

```
        i = (itw1 × 10) + itw1;
            //Only autounbox, no rebox
        System.out.println("Value of i:", + i);
        dtw = itw1 + dtw;  //Auto unbox then rebox
        System.out.println("New value of dtw:", + dtw);
        isw = 1;
        switch(isw)          //Auto unbox
        {
            case 1:
                    System.out.println("First year");
                    break;
            case 2:
                    System.out.println("Second year");
                    break;
            default:
                    System.out.println("Not a student");
        }
    }
```

Output

```
Values of itw1
Initially: 299
After increment: 300
Value of itw2: 3300
Value of i: 3300
New value of dtw = 3333.33
First year
```

In the above example program, the increment operation on Integer object, itw1 is performed by first auto unboxing it into a int, then incrementing its value and later reboxing it back into an integer. The expression for computing the value of itw2, first auto unboxes itw1, then performs the computation and later reboxes the result to an integer.

The next expression for computing the value of *i*, first auto-unboxes itw1, then performs the computation. It does not requires reboxing because '*i*' is already an int. The expression for computing dtw involves objects of two different type wrappers double and integer. They both are unboxed first, then they are added and later the result is stored in dtw after reboxing it into a double. Next, the switch statement is controlled using an integer numeric object. This object is first unboxed and then one of the defined cases is selected based on its int value.

240. Explain the concept of generics and generic collections in Java.

Answer :

Generics(Parameterized Type)

Generics is a concept in Java that enables a programmer to define and apply an algorithm irrespective of the data type. That is same algorithm can be applied multiple times on many data types without enforcing additional effort. Using generic it is possible to create classes, interfaces and method that can be executed in a type-safer manner independent of any specific type of data. Such methods/classes are referred as generic methods/classes. The major advantage of using generic is that it executes the program in type-safe mode. Moreover, generic expands the programmer ability in reusing the code.

Syntax

```
class className<Datatype.param 1, Datatype param 2,.....Datatype param n)
{
:
}
```

Now, the way of declaring reference to generic class is,

```
classname<datatype arg 1, datatype arg 2.....datatype arg n > variable name = new classname<datatype arg 1,
datatype arg 2.....datatype arg n>(cons arg 1, cons arg n);
```

Generic Collection

Collection framework has been modified for supporting the concept of generic. This is because of the type safety capability that generic provides to collection framework.

Earlier pre-generic collection based code were used by programmer to store only the reference belonging to the type object. This required for a programmer to ensure that the reference of only object are stored in specific collection. If a programmer fails in this regard, than it result as an error. Moreover, the usage of pre-generic collection allow the programmer to manually type-cast the retrieved references into proper type, thereby making it inconvenient. Therefore to overcome this issue it is necessary to use the concept of generic, which enhance the usability and safety of collections. The two major improvement made by generic collections over pre-generic collection are,

1. It allows a programmer to store reference to object of proper type in the collection.
2. It doesn't allow the program to manually type cast a reference retrieved from a collection. Rather, it perform the casting automatically.

Example Program

```
import java.util.*;
class GenericExample
{
    public static void main(String args[ ])
    {
        ArrayLlist<String>arrlist = new ArrayList<string>( );
        arrlist.add("welcome");
        arrlist.add("to");
        arrlist.add("sia");
        arrlist.add("group");
        Iterator<String>obj = arrlist.iterator( );
        while(objI.hasNext( ))
        {
            String st = objI:next( );
            System.out.println(st+"is"+st.length( )+"chars long");
        }
    }
}
```

Here, the arrlist can store the references to objects of string data type. It is not necessary to cast the return value of next( ) method into string. The casting can be performed automatically. In this program, Iterator and ListIterator interfaces are used. These interfaces can also act as generic type. Therefore, the data type of argument must be accepted with the data type collections for which the iterator can be obtained. Mean while, type compatibility can be required at compile time. If any object type errors are occurred, they can be detected at compile time rather than throwing exceptions during run-time.

Q41. How to create a generic class with two type parameters? Explain.

Answer :

A generic class with two parameters can be created by placing comma between them i.e., the parameters are separated using a comma as shown below,

```
class Gen<T, X>
```

The following code illustrates the concept.

```
//Declaration of a generic class with two type parameters T and X
class GenericTwo<T, X>
{
```

```
    T 01;
    //Declaration of objects 01 and 02
    X 02;
    //Generic constructor with two objects of type T and X
    GenericTwo(T obj1, X obj2)
    {
        01 = obj1;
        02 = obj2;
    void displayTypes()
    //displaying the type of T and X
    {
    System.out.println("The datatype of type
    parameter T is" + 01.getClass( ).getName( ));
    System.out.println("The datatype of type
    parameter X is" + 02.getClass.getName( ));
    }
}
T get01( )
{
    return 01;   //return 01
}
X get02( )
{
    return 02;   //return 02
}
//Demonstration of GenericTwo class
class DemoGenericTwo
{
    public static void main(String args[ ])
    {
        //creating a 'GenericTwo' reference for double
        //and strings
        GenericTwo<Double, String> dsobj;
        dsobj = new GenericTwo<Double,
        String>(80.6, "Generics Example");
        dsobj.displayTypes( );
        //Displaying the types
        double i = dsobj.get01( );
        //storing and displaying the values
        System.out.println("The value is" +i);
        String str = dsobj.get02( );
        System.out.println("The value is" + str);
    }
}
```

Output

The datatype of type parameter T is java.lang.Double

The datatype of type parameter X is java.lang.String

The value is 80.6

The value is Generics Example

Initially, a generic class 'GenericTwo' with two type parameters T and X is declared. As it contains two type parameters, two type arguments must be passed at the time of object creation i.e., the first parameter **'Double'** is substituted for T and the second parameter **'String'** for **X**. However, same type of arguments can also be passed but doing so is unnecessary.

Q42. Give the general form of a generic class. Also, discuss the wildcard arguments.

Answer :

Syntax of a Generic Class

```
class name_of_class<list_of_type parameters>
{
    //Statements
}
```

Syntax for Declaring a Reference to a Generic Class

name_of_class<list_of_type argument> name_of_variable = new name_of_class<list_of_type argument>(list_of_constant arguments)

WildCard Arguments

WildCard argument is a feature of java generic using which it is possible to create a generic method that checks the specified condition irrespective of the type of data every object holds. This argument represents an unknown type and is denoted using "?" symbol.

Example

The following example demonstrates the usage of WildCard arguments.

```
class Avg_class< X extends number >
{
    X[ ] num_arr;
    Avg_class(X[ ]v)
    {
    num_arr = v;
    }
    double computeavg( )
    {
    double sum = 0.0;
    for (int i = 0; i<num_arr.length; i++)
    sum = sum + num_arr[i].doublevalue( );
    return sum/num_arr.length;
    }
    boolean same_Avg(Avg_class<?> s)
    {
    if(computeavg( ) = = s.computeavg ( ))
    return true;
    return false;
    }
}
```

```
class WildArg_Demo
{
public static void main(String args [ ])
{
Integer I[ ] = {2, 4, 6, 8, 10}
Avg_class<Integer> w1 = new Avg_class<Integer>( );
double d = w1.computeaverage( );
System.out.println ("The average of w1 is" + d);
Float f[ ] = {2.05, 3.123, 4.0F, 5.723};
Avg_class<Float>w2 = new Avg_class<Float>( );
double d1 = w2.computeaverage ( );
System.out.println("The average of w2 is" + d1);
System.out.println("The average of w1 and w2");
    if(w1.same_Avg(w2))
    System.out.println("The averages are same");
    else
System.out.println("The averages are different");
}
}
```

In the above program the generic method same_Avg( ) verifies whether the averages of two different data types are same. The statement Avg_class<?> matches any Avg_class object, thereby enabling comparison of averages of two different Avg_class objects.

243. Describe the process of creating a generic method.

Answer : **Model Paper-II, Q5(b)**

Creating a Generic Method

Method that is declared inside a generic class can be use the type parameters of the class, thereby making the method a "generic method". This method can be declared with at least one type parameter of the respective method itself. In addition to this, it is also possible to create a generic method which is defined within a non-generic class. The primary advantage of generic method is that, the method ensures type safety.

Syntax for Declaring Generic Method

<list_of_type parameters> return_type name_of_ method (list_of_parameters)

The list of type parameters are separated by comma and are placed before the return type.

Example

The following example demonstrates the creation of Generic method.

```
class Gen_method
{
  static < x, y extends x> boolean isMember(x a, y[ ] b)
{
    for(int i = 0; i < b.length; i ++)
    if(a.equals(b[i]))
    return true;
    return false;
}
```

```
public static void main(String args[ ])
{
String Str.arr[ ] = {"Alice", "Bob", "Racheal", "John"};
if(isMember("Bob", s))
    System.out.println("Bob is a member of string array");
if(!isMember("James", s))
    System.out.println("James is not a member of string array");
Integer num_arr[ ] = {2, 4, 6, 8, 10};
    if(isMember(6, num_arr))
        System.out.println("6 is member of number array");
    if(!isMember(12, num_arr))
        System.out.println("12 is not the member of number array")
}
}
```

In the above program, Gen_Method is a non-generic class in which a static generic method isMember( ) is declared. This method checks whether an object is present in the array or not.

Initially in the declaration statement of generic method, the list of parameters precedes the return type. Here, it must be ensured that the type of second argument (y) matches with the type of first argument (x) or any of its subclass. This is because type y is upperbounded by type x. The type-matching relationship enables the generic method "isMember" to be invoked with any of inter-compatible arguments. This method is invoked within the main ( ) without specifying the type arguments. The reason for not specifying the type arguments is that, these arguments are automatically recognized and types of X and Y are modified inaccordance to the type argument. For instance, consider the statement.

if (isMember ("Bob", str_arr))

In this statement, the type of first argument is "string" due to which the type of 'X' becomes string. The base type of second argument is also "string", thereby substituting string as a type for "y". Since there is no type mismatch (i.e., both the types of X and Y are equal), the statement is executed successfully and the statement "Bob is a member of str_arr" is printed.

However, if the same statement was declared in the following manner,

if (isMember("Bob", num_arr))

Then a type-mismatch error is generated at compile time because the type of "x" is different from type of "y".

Q44. Explain about generic interface.

Answer :

Creating Generic Interfaces

Generic interfaces are declared in a way similar to generic class.

Syntax for Declaring Generic Interfaces

interface name_of_interface <list_of_type parameters) ;

In the above syntax, list of type parameters is separated by comma.

Syntax for Declaring Type Arguments

Whenever a generic interface is implemented, the type arguments need to be specified in the following manner.

class name_of_class <list_of_type parameters>

implements name_of_interface <list_of_type arguments>

Example

The following example demonstrates the creation of generic interfaces.

```
interface  Max_Min <X extends Comparable <X>>
{
    X maximum ( );
    X minimum ( );
}
class MaxMinClass <X extends Comparable<X>> implements MaxMin<X>
{
    X[ ] values;
    MaxMin class (X [ ] v)
    {
        values = v;
    }
    public X minimum ( )
    {
        X p = values [0];
        for (int i = 1; i < values.length; i++)
        if (values [i].compareTo (p) < 0)
            p = values [i];
        return p;
    }
    public X maximum ( )
    {
        X p = values [0];
        for (int i = 1; i < values.length; i++)
        if (values[i].compareTo (p) > 0)
            p = values [i];
        return p;
    }
}
class Gen_interface
{
    public static void main (String args[ ])
    {
        Integer num_arr [ ] = {6, 12, 18, 24, 30};
        Character ch_arr [ ] = {'M', 'S', 'R', 'Y'};
        MaxMinclass<integer>c1 = new MaxMinClass <integer> (num_arr);
```

```
        System.out.println ("The maximum value in num_arr :" +c1.maximum());
        System.out.println ("The minimum value in num_arr :" +c1.minimum( ));
        MaxMinClass<character>c2 = new MaxMinClass<character>(ch_arr);
        System.out.println("The maximum value in ch_arr :" +c2.maximum( ));
        System.out.println("The minimum value in ch_arr :" +c2.minimum( ));
    }
}
```

In the above program, the declaration of generic interface consists of type parameter 'x', which is upper bounded to "comparable". It is an in-built interface defined in java.lang, this interface ensures that the above program can be invoked with comparable objects.

The statement

class MaxMinClass<X extends Comparable <X>> implements MaxMin<x>

Specifies that Max_Min is implemented by MaxMin class. Here, the type parameter 'X' is initially declared by MaxMinClass and then passed to Max_Min interface. It is necessary to ensure that MaxMinClass is also bounded to the same upperbound as Max_Min. After identifying the type parameter, it is passed to the interface. Generally, it can be said that a class which implements a generic interface must also be generic in order to avoid compilation errors.

For instance, the statement

class MaxMinClass implements Max_Min<X>.

Generates an error because the class doesnot declare type parameter due to which it is not possible to pass the parameter to Max_Min. However, this issue can be solved when the class implements a specific type of generic interface like the following.

class MaxMinClass implements <Integer>

Advantages of Generic Interface

1. Generic interface can be implemented for different data types.
2. Generic interface enables the user to apply constraints on the type of data for which it is possible to implement the interface.

Q45. Discuss about generic class hierarchies.

Answer :

Generic Class Hierarchies Nov./Dec.-17(R13), Q7(a)

A generic class can be one of the part of class hierarchy in the same non generic classes. So, it can be either subclass or superclass. The type arguments required by a generic superclass in generic class hierarchies must be passed through all the sub classes.

Generic Superclass

Consider the below example

```
class Generic <T>
{
T obj;
Generic (T x)
{
obj = x;
}
```

```
T getobj( )
{
return obj;
}
}
class Generic <T, V> extends Generic <T>
{
V obj2;
Generic 2(T x, V y)
{
super (x);
obj2 = y;
}
V getobj1( )
{
return y;
}
}
class GCH
{
public static void main (String args[ ])
{
Generic1 <String, Integer> a = new Generic1 <String, Integer > (" Result = :", 99);
System.out.print(a.getobj( ));
System.out.println(a.getobj1( ));
}
}
```

In the above example, Generic1 is the subclass that extends the superclass Generic. The subclass will pass the type parameter to superclass. Generic1 does not use the type parameter but adds its own parameters (i.e., V) if required.

Generic Subclass

A generic subclass can be child of any generic or non generic superclass. Consider the below example,

```
class Nongeneric
int n;
```

```
NonGeneric(int x)
{
n = x;
}
int get( )
{
return n;
}
}
class Generic <T> extends NonGeneric
{
T obj;
Generic(T a , int b)
{
super(b);
obj = a;
}
T getobj( )
{
return obj;
}
}
class Hierarchy
{
public static void main (string args[ ])
{
Generic <string> g = new Generic <String> ("Hello", 20);
System.out.print(g.getobj( ) + " ");
System.out.println(g. get( ));
}
}
```

In the above program Generic is the inheriting subclass of NonGeneric. Type arguments are not specified because NonGeneric is not Generic.

To avoid compatibility issues in generic hierarchy the instance of operator is used. The purpose of this object is to determine whether a particular object is an instance of the class. It is applied to the objects of generic classes. It returns true if the object belongs to particular type or when it can be cast to particular type. An instance of one generic class can be cast to another when both of them are compatible and contain same type arguments. Generic class hierarchies even allow the concept of method overriding to be applied to their methods.

EXERCISE QUESTIONS

What is the output of the following code snippet?

```
interface X
{
void display();
}
class Y
{
    public void display()
    {
        System.out.println("Welcome");
    }
}
class Z extends Y implements X
{
}
class BaseClass
{
    public static void main(String[] args)
    {
        X x = new Z();
        x.display();
    }
}
```

Give the reason for the compile time error in the following code snippet.

```
interface X
{
    void methodX();
}
class Y implements X
```

```
{
    void methodX()
    {
        System.out.println("Method X");
    }
}
```

3. **What are the advantages of Packages?**

4. **Write a java program to demonstrate the accessing members of a class.**

5. **Check whether the following declaration is valid or not?**

```
public interface Marker
{
}
```