UNIT IV

Packages and Interfaces : Defining, Creating and Accessing a Package, Understanding CLASSPATH, importing packages, differences between classes and interfaces, defining an interface, implementing interface, applying interfaces, variables in interface and extending interfaces.
Exploring packages – Java.io

PACKAGES

- ✓ A unique name had to be used for each class to avoid name collisions. The name you choose for a class will be reasonably unique and not collide with class names chosen by other programmers.
- ✓ Java provides a mechanism for partitioning the class name space into more manageable chunks. This mechanism is the package.
- ✓ The package is both a naming and a visibility control mechanism. W can define classes inside a package that are not accessible by code outside that package. We can also define class members that are only exposed to other members of the same package.

A package is the set of classes and interface that provides name space management and access protection.

Defining a Package

To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package.
The **package** statement defines a name space in which classes are stored.
If you omit the **package** statement, the class names are put into the default package, which has no name.

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

package *pkg*;

Here, *pkg* is the name of the package.

For example, package MyPackage;

Java uses file system directories to store packages. More than one file can include the same **package** statement. The **package** statement simply specifies to which package the classes defined in a file belong. It does not exclude other classes in other files from being part of that same package. Most real-world packages are spread across many files. You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.

The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system.

For example,   package java.awt.image;

Finding Packages and CLASSPATH

- ✓ Packages are mirrored by directories. This raises an important question: How does the Java run-time system know where to look for packages that you create? The answer has two parts.
- ✓ First, by default, the Java run-time system uses the current working directory as its starting point. Thus, if your package is in the current directory, or a subdirectory of the current directory, it will be found.
- ✓ Second, you can specify a directory path or paths by setting the **CLASSPATH** environmental variable.

Setting up Java Environment:

After installing the JDK, we need to set at least one environment variable in order to able to compile and run Java programs. A PATH environment variable enables the operating system to find the JDK executables when our working directory is not the JDK's binary directory.

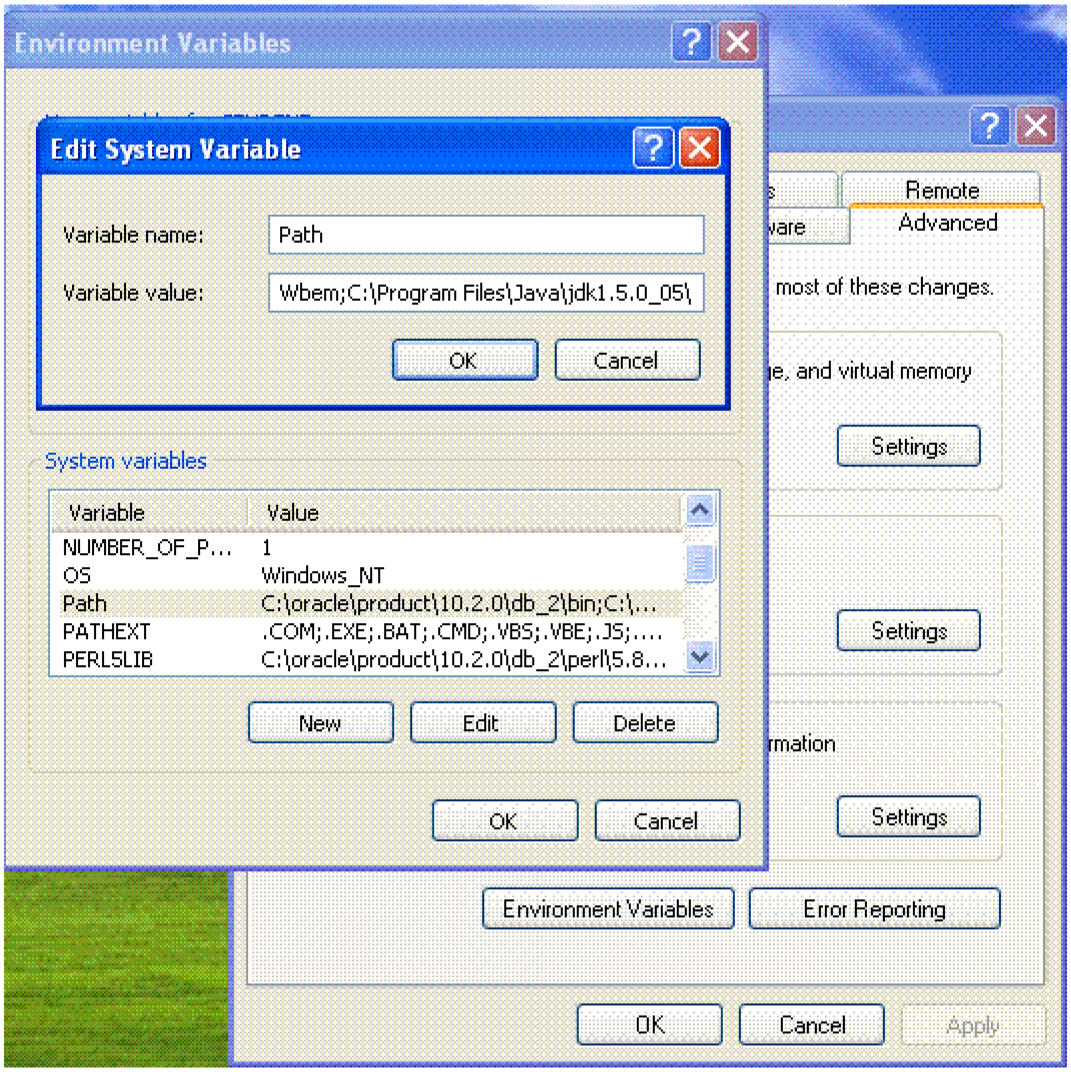
· **Setting environment variables from a command prompt:** If we set the variables from a command prompt, they will only hold for that session. To set the PATH from a command prompt:

```
set PATH=C:\Program Files\Java\jdk1.5.0_05\bin;
```

Setting environment variables as system variables:

If we set the variables as system variables they will hold continuously.
o Right-click on *My Computer*
o Choose *Properties*
o Select the *Advanced* tab
o Click the *Environment Variables* button at the bottom
o In system variables tab, select path (system variable) and click on edit button
o A window with variable name path and its value will be displayed.

o Don’t disturb the default path value that is appearing and just append (add) to that path at the end:
;C:\ProgramFiles\Java\ jdk1.5.0_05\bin;
o Finally press OK button.



Variable name: Path
Variable value: Wbem;C:\Program Files\Java\jdk1.5.0_05\

Variable	Value
NUMBER_OF_P...	1
OS	Windows_NT
Path	C:\oracle\product\10.2.0\db_2\bin;C:\...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PERL5LIB	C:\oracle\product\10.2.0\db_2\perl\5.8...

most of these changes.
e, and virtual memory
rmation

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

class AccountBalance {
public static void main(String args[]) {
Balance current[] = new Balance[3];
current[0] = new Balance("K. J. Fielding", 123.23);
current[1] = new Balance("Will Tell", 157.02);
current[2] = new Balance("Tom Jackson", -12.33);
for(int i=0; i<3; i++) current[i].show();
}
}
```

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

```
java MyPack.AccountBalance
```

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

```
java AccountBalance
```

Access Protection

Java addresses four categories of visibility for class members:
■ Subclasses in the same package
■ Non-subclasses in the same package
■ Subclasses in different packages
■ Classes that are neither in the same package nor subclasses

The three access specifiers, **private**, **public**, and **protected**, provide a variety of ways to produce the many levels of access required by these categories.

Anything declared **public** can be accessed from anywhere.
Anything declared **private** cannot be seen outside of its class.

When a member does not have an explicit access specification, it is visible to subclasses as well as to other classes in the same package. This is the **default** access.
If you want to allow an element to be seen outside your current package, but only to classes that subclass your class directly, then declare that element **protected**.

A class has only two possible access levels: default and public. When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package.

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

Importing Packages

- ✓ Given that packages exist and are a good mechanism for compartmentalizing diverse classes from each other, it is easy to see why all of the built-in Java classes are stored in packages. There are no core Java classes in the unnamed default package; all of the standard classes are stored in some named package. Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.
- ✓ The **import** statement is a convenience to the programmer and is not technically needed to write a complete Java program. If you are going to refer to a few dozen classes in your application, however, the **import** statement will save a lot of typing.

- ✓ In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions.

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

```
import pkg1[.pkg2].(classname|*);
```

Here, *pkg1* is the name of a top-level package, and *pkg2* is the name of a subordinate package inside the outer package separated by a dot (**.**).Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs.

```
import java.util.*;
class MyDate extends Date {
}
```

The same example without the **import** statement looks like this:

```
class MyDate extends java.util.Date {
}
```

Example:

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

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

INTERFACES

- ✓ Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.
- ✓ Once it is defined, any number of classes can implement an **interface**. Also, one class can implement any number of interfaces.
- ✓ To implement an interface, a class must create the complete set of methods defined by the interface.
- ✓ By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism.
- ✓ Interfaces are designed to support dynamic method resolution at run time. Normally, in order for a method to be called from one class to another, both classes need to be present at compile time so the Java compiler can check to ensure that the method signatures are compatible.
- ✓ Since interfaces are in a different hierarchy from classes, it is possible for classes that are unrelated in terms of the class hierarchy to implement the same interface. This is where the real power of interfaces is realized.

A programmer uses an abstract class when there are some common features shared by all the objects. A programmer writes an interface when all the features have different implementations for different objects. Interfaces are written when the programmer wants to leave the implementation to third party vendors. An interface is a specification of method prototypes. All the methods in an interface are abstract methods.

· An interface is a specification of method prototypes.
· An interface contains zero or more abstract methods.
· All the methods of interface are public, abstract by default.
· An interface may contain variables which are by default public static final.
· Once an interface is written any third party vendor can implement it.

· All the methods of the interface should be implemented in its implementation classes.
· If any one of the method is not implemented, then that implementation class should be declared as abstract.
· We cannot create an object to an interface.
· We can create a reference variable to an interface.
· An interface cannot implement another interface.
· An interface can extend another interface.
· A class can implement multiple interfaces.

Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;       }
```

- ✓ Here, *access* is either **public** or not used. When no access specifier is included, then default access results, and the interface is only available to other members of the package in which it is declared.
- ✓ When it is declared as **public**, the interface can be used by any other code. *name* is the name of the interface, and can be any valid identifier. Notice that the methods which are declared have no bodies. They end with a semicolon after the parameter list. They are, essentially, abstract methods; there can be no default implementation of any method specified within an interface.
- ✓ Each class that includes an interface must implement all of the methods.
- ✓ Variables can be declared inside of interface declarations. They are implicitly **final** and **static**, meaning they cannot be changed by the implementing class. They must also be initialized with a constant value.
- ✓ All methods and variables are implicitly **public** if the interface, itself, is declared as **public**.

```
interface Callback {
void callback(int param);
}
```

Implementing Interfaces

Once an **interface** has been defined, one or more classes can implement that interface. To implement an interface, include the **implements** clause in a class definition, and then create the methods defined by the interface.

The general form of a class that includes the **implements** clause looks like this:

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

Here, *access* is either **public** or not used. If a class implements more than one interface, the interfaces are separated with a comma. If a class implements two interfaces that declare the same method, then the same method will be used by clients of either interface. The methods that implement an interface must be declared **public**. Also, the type signature of the implementing method must match exactly the type signature specified in the **interface** definition.

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}
}
```

Notice that **callback( )** is declared using the **public** access specifier. *When you implement an interface method, it must be declared as **public***. It is both permissible and common for classes that implement interfaces to define additional members of their own.

```
class Client implements Callback {
public void callback(int p) {
System.out.println("callback called with " + p);
}

void nonIfaceMeth() {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References

```
class TestIface {
public static void main(String args[]) {
Callback c = new Client();
c.callback(42);
}
}
```

INTERFACES CAN BE EXTENDED

One interface can inherit another by use of the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain. Following is an example:

```
// One interface can extend another.
interface A {
void meth1();
void meth2();
}

interface B extends A {
void meth3();
}

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

VARIABLES IN INTERFACE

The variables that are declared within an interface are implicitly **static and final**.

```
public interface A
{
int x=10;
}
class B implements A
```

```
{
int i;
B()
{
i=x+1;
}

void show()
{
System.out.println(i);
}
}

class Demo
{
public static void main(String args[])
{
B b=new B();
b.show();
System.out.println(B.x);
}
}
```

MULTIPLE INHERITANCE USING INTERFACES

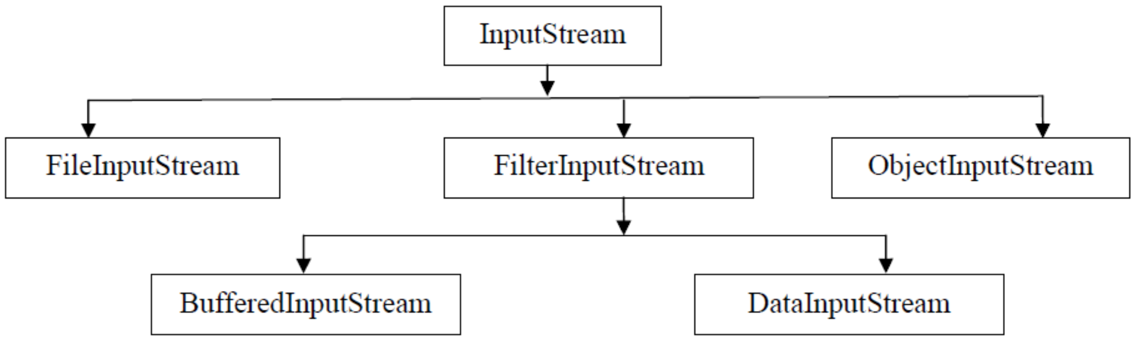
Java does not support multiple inheritance. But multiple inheritance can be achieved by using interfaces.

```
interface Father
{ double PROPERTY = 10000;
double HEIGHT = 5.6;}
interface Mother
{ double PROPERTY = 30000;
double HEIGHT = 5.4;}

class MyClass implements Father, Mother
{
void show()
{
 System.out.println("Total property is :" +(Father.PROPERTY+Mother.PROPERTY));
System.out.println ("Average height is :" + (Father.HEIGHT + Mother.HEIGHT)/2 );
}
}

class InterfaceDemo
{
 public static void main(String args[])
{
 MyClass ob1 = new MyClass();
ob1.show();
}
}
```

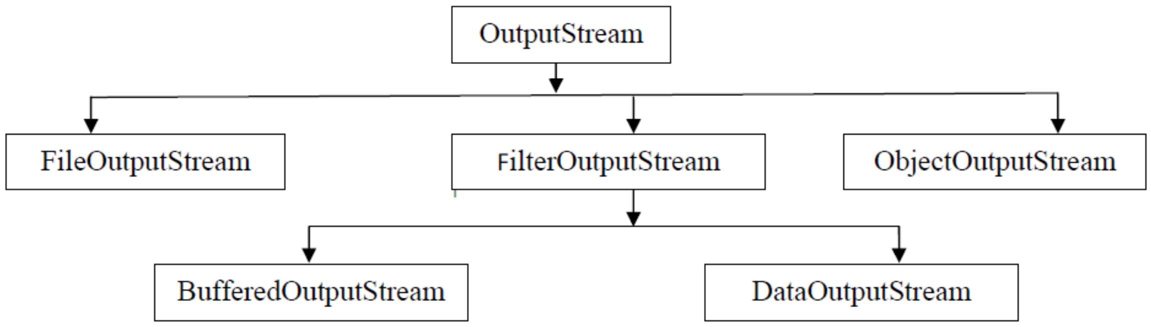
EXPLORING java.io

A Stream represents flow of data from one place to another place. Input Streams reads or accepts data. Output Streams sends or writes data to some other place. All streams are represented as classes in java.io package. The main advantage of using stream concept is to achieve hardware independence. This is because we need not change the stream in our program even though we change the hardware. Streams are of two types in Java:

· **Byte Streams:** Handle data in the form of bits and bytes. Byte streams are used to handle anycharacters (text), images, audio and video files. For example, to store an image file (.gif or .jpg), we should go for a byte stream. To handle data in the form of 'bytes' the abstract classes: InputStream and OutputStream are used. The important classes of byte streams are:



Byte Stream Classes for Reading Data

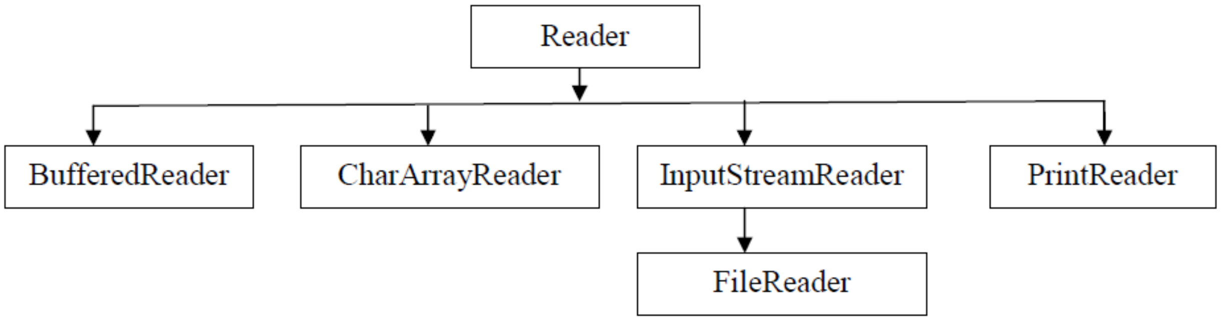


Byte Stream Classes for Writing Data

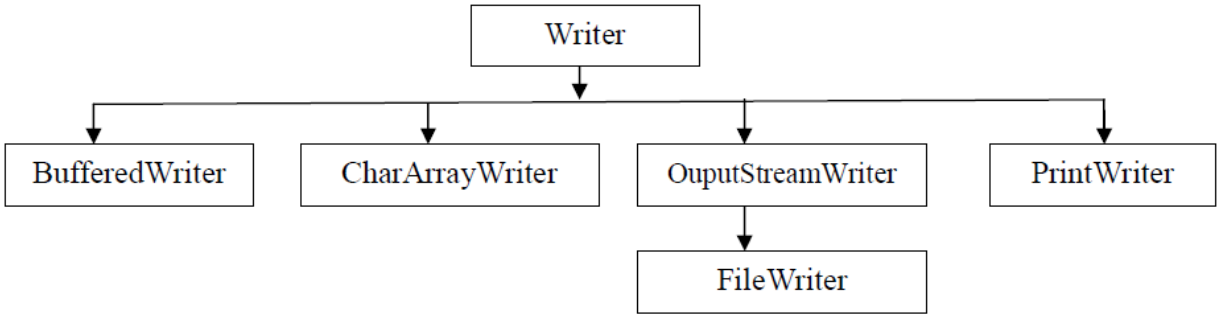
o FileInputStream/FileOutputStream: They handle data to be read or written to disk files.

o FilterInputStream/FilterOutputStream: They read data from one stream and write it to another stream.
o ObjectInputStream/ObjectOutputStream: They handle storage of objects and primitive data.

· **Character or Text Streams:** Handle data in the form of characters. Character or text streams can always store and retrieve data in the form of characters (or text) only. It means text streams are more suitable for handling text files like the ones we create in Notepad. They are not suitable to handle the images, audio or video files. To handle data in the form of 'text' the abstract classes: Reader and Writer are used. The important classes of character streams are:



Text Stream Classes for Reading Data



Text Stream Classes for Writing Data

o BufferedReader/BufferedWriter: - Handles characters (text) by buffering them. They provide efficiency.
o CharArrayReader/CharArrayWriter: - Handles array of characters.
o InputStreamReader/OutputStreamWriter: - They are bridge between byte streams and character streams. Reader reads bytes and then decodes them into 16-bit Unicode characters. Writer decodes characters into bytes and then writes.
o PrintReader/PrintWriter: - Handle printing of characters on the screen.

File: A file represents organized collection of data. Data is stored permanently in the file. Once data is stored in the form of a file we can use it in different programs.

Program 1: Write a program to read data from the keyboard and write it to a text file using byte stream classes.

```
import java.io.*;
class Create1
{ public static void main(String args[]) throws IOException
{ //attach keyboard to DataInputStream
DataInputStream dis = new DataInputStream (System.in);
//attach the file to FileOutputStream
FileOutputStream fout = new FileOutputStream ("myfile");
//read data from DataInputStream and write into FileOutputStream
char ch; System.out.println ("Enter @ at end : " ) ;
while( (ch = (char) dis.read() ) != '@' )
fout.write (ch);
fout.close ();
}
}
```

Program 2: Write a program to improve the efficiency of writing data into a file using BufferedOutputStream.

```
//Creating a text file using byte stream classes
import java.io.*;
class Create2
{
 public static void main(String args[]) throws IOException
{
DataInputStream dis = new DataInputStream (System.in);

//attach file to FileOutputStream, if we use true then it will open in append mode

FileOutputStream fout = new FileOutputStream ("myfile", true);
BufferedOutputStream bout = new BufferedOutputStream (fout, 1024);

//Buffer size is declared as 1024 otherwise default buffer size of 512 bytes is used.
//read data from DataInputStream and write into FileOutputStream

char ch;
System.out.println ("Enter @ at end : " ) ;
while ( (ch = (char) dis.read() ) != '@' )
bout.write (ch);
bout.close ();
fout.close ();
}
}
```

Program 3: Write a program to read data from *myfile* using FileInputStream.
//Reading a text file using byte stream classes

```
import java.io.*;
class Read1
{
 public static void main (String args[]) throws IOException
{
//attach the file to FileInputStream

FileInputStream fin = new FileInputStream ("myfile");

//read data from FileInputStream and display it on the monitor
int ch;
while ( (ch = fin.read() ) != -1 )
System.out.print ((char) ch);
fin.close ();
}
}
```

Program 4: Write a program to improve the efficiency while reading data from a file using BufferedInputStream.

```
//Reading a text file using byte stream classes

import java.io.*;
class Read2
{ public static void main(String args[]) throws IOException
{
 //attach the file to FileInputStream

FileInputStream fin = new FileInputStream ("myfile");
BufferedInputStream bin = new BufferedInputStream (fin);

//read data from FileInputStream and display it on the monitor

int ch;
while ( (ch = bin.read() ) != -1 )
System.out.print ( (char) ch);
fin.close ();
}
}
```

Program 5: Write a program to create a text file using character or text stream classes.

```
//Creating a text file using character (text) stream classes

import java.io.*;
class Create3
{
 public static void main(String args[]) throws IOException
{
 String str = "This is an Institute" + "\n You are a student"; // take a String

//Connect a file to FileWriter

FileWriter fw = new FileWriter ("textfile");

//read chars from str and send to fw

for (int i = 0; i<str.length () ; i++)
fw.write (str.charAt (i) );
fw.close ();
}
}
```

Program 6: Write a program to read a text file using character or text stream classes.

```
//Reading data from file using character (text) stream classes

import java.io.*;
class Read3

{
 public static void main(String args[]) throws IOException
{
 //attach file to FileReader
FileReader fr = new FileReader ("textfile");
//read data from fr and display
int ch;
while ((ch = fr.read()) != -1)
System.out.print ((char) ch);
//close the file
fr.close ();
}
}
```

Serialization of objects:

· Serialization is the process of storing object contents into a file. The class whose objects are stored in the file should implement ''Serializable' interface of java.io package.
· Serializable interface is an empty interface without any members and methods, such an interface is called 'marking interface' or 'tagging interface'.
· Marking interface is useful to mark the objects of a class for a special purpose. For example, 'Serializable' interface marks the class objects as 'serializable' so that they can

be written into a file. If serializable interface is not implemented by the class, then writing that class objects into a file will lead to NotSerializableException.
· static and transient variables cannot be serialized.
· De-serialization is the process of reading back the objects from a file.

Program 7: Write a program to create Employ class whose objects is to be stored into a file.

```
//Employ information
import java.io.*;
import java.util.*;
class Employ implements Serializable
{
private int id;
private String name;
private float sal;
private Date doj;
Employ (int i, String n, float s, Date d)
{ id = i;
name = n;
sal = s;
doj = d;
}
void display ()
{
System.out.println (id+ "\t" + name + "\t" + sal + "\t" + doj);
}
static Employ getData() throws IOException
{ BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
System.out.print ("Enter employ id : ");
int id = Integer.parseInt(br.readLine());
System.out.print ("Enter employ name : ");
String name = br.readLine ();
System.out.print ("Enter employ salary : " );
float sal = Float.parseFloat(br.readLine ());
Date d = new Date ();
Employ e = new Employ (id, name, sal, d);
return e;
}
}
```

Program 8: Write a program to show serialization of objects.

```
//ObjectOutputStream is used to store objects to a file
import java.io.*;
import java.util.*;
class StoreObj
{
```

```
public static void main (String args[]) throws IOException
{
 BufferedReader br = new BufferedReader (new InputStreamReader (System.in));
FileOutputStream fos = new FileOutputStream ("objfile");
ObjectOutputStream oos = new ObjectOutputStream ( fos );
System.out.print ("Enter how many objects : ");
int n = Integer.parseInt(br.readLine () );
for(int i = 0;i<n;i++)
{
Employ e1 = Employ.getData ();
oos.writeObject (e1);
}
oos.close ();
fos.close ();
}
}
```

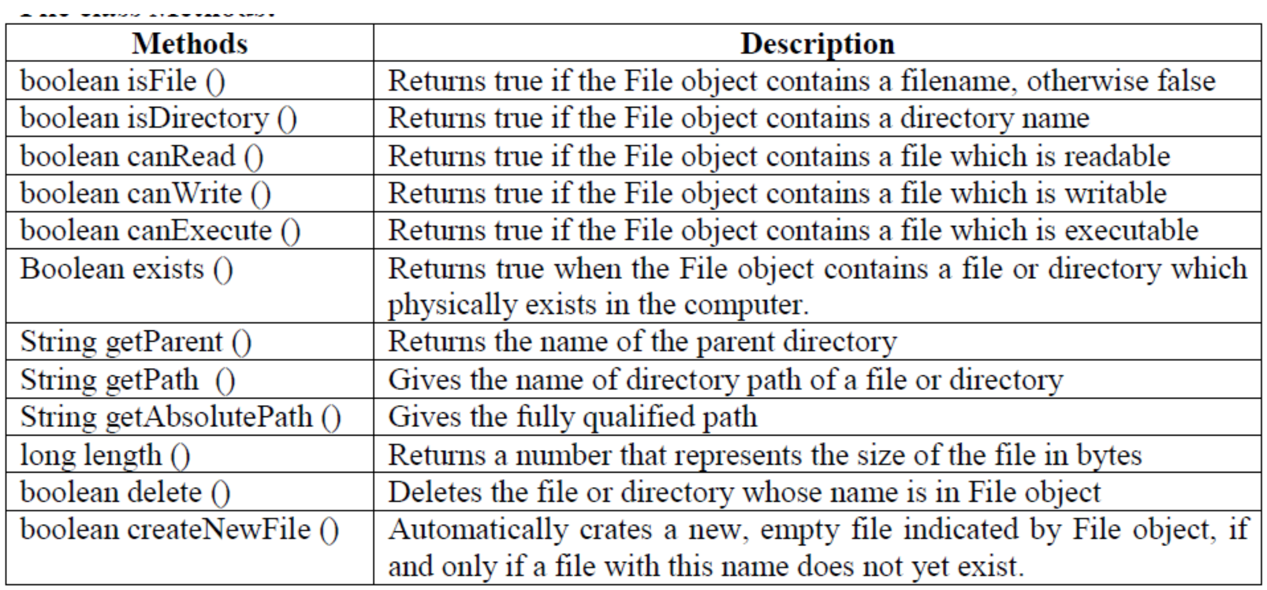
Program 9: Write a program showing deserialization of objects.

```
//ObjectInputStream is used to read objects from a file
import java.io.*;
class ObjRead
{
 public static void main(String args[]) throws Exception
{
FileInputStream fis = new FileInputStream ("objfile");
ObjectInputStream ois = new ObjectInputStream (fis);
try
{
 Employ e;
while ( (e = (Employ) ois.readObject() ) != null)
e.display ();
}
catch(EOFException ee)
{
System.out.println ("End of file Reached...");
}
finally
{ ois.close ();
fis.close ();
}
}
}
```

File Class: File class of java.io package provides some methods to know the properties of a file or a directory. We can create the File class object by passing the filename or directory name to it.

· File obj = new File (filename);
· File obj = new File (directoryname);
· File obj = new File ("path", filename);
· File obj = new File ("path", directoryname);

File class Methods:



Methods	Description
boolean isFile ()	Returns true if the File object contains a filename, otherwise false
boolean isDirectory ()	Returns true if the File object contains a directory name
boolean canRead ()	Returns true if the File object contains a file which is readable
boolean canWrite ()	Returns true if the File object contains a file which is writable
boolean canExecute ()	Returns true if the File object contains a file which is executable
Boolean exists ()	Returns true when the File object contains a file or directory which physically exists in the computer.
String getParent ()	Returns the name of the parent directory
String getPath ()	Gives the name of directory path of a file or directory
String getAbsolutePath ()	Gives the fully qualified path
long length ()	Returns a number that represents the size of the file in bytes
boolean delete ()	Deletes the file or directory whose name is in File object
boolean createNewFile ()	Automatically crates a new, empty file indicated by File object, if and only if a file with this name does not yet exist.

Program 10: Write a program that uses File class methods.

```
//Displaying file properties
import java.io.*;
class FileProp
{
public static void main(String args[])
{
String fname = args [0];
File f = new File (fname);
System.out.println ("File name: " + f.getname ());
System.out.println ("Path:"+ f.getPath ());
System.out.println ("Absolute Path:"+ f.getAbsolutePath ());
System.out.println ("Parent:"+ f.getParent ());
System.out.println ("Exists:"+ f.exists ());
if ( f.exists() )
{
System.out.println ("Is writable: "+ f.canWrite ());
System.out.println ("Is readable: "+ f.canRead ());
System.out.println ("Is executable: "+ f.canExecute ());
System.out.println ("Is directory: "+ f.isDirectory ());
System.out.println ("File size in bytes: "+ f.length ());
} } }
```

INPUT AND OUTPUT

A stream represents flow of data from one place to other place. Streams are of two types in java. Input streams which are used to accept or receive data. Output streams are used to display or write data. Streams are represented as classes in java.io package.

· **System.in:** This represents InputStream object, which by default represents standard input device that is keyboard.
· **System.out:** This represents PrintStream object, which by default represents standard output device that is monitor.
· **System.err:** This field also represents PrintStream object, which by default represents monitor. System.out is used to display normal messages and results whereas System.err is used to display error messages.

To accept data from the keyboard:

· Connect the keyboard to an input stream object. Here, we can use InputStreamReader that can read data from the keyboard.

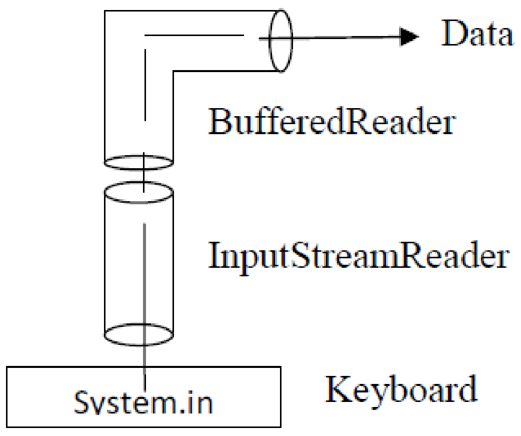
InputSteamReader obj = new InputStreamReader (System.in);

· Connect InputStreamReader to BufferReader, which is another input type of stream. We are using BufferedReader as it has got methods to read data properly, coming from the stream.

BufferedReader br = new BufferedReader (obj);

The above two steps can be combined and rewritten in a single statement as:
BufferedReader br = new BufferedReader (new InputStreamReader (System.in));

· Now, we can read the data coming from the keyboard using read () and readLine () methods available in BufferedReader class.



Accepting a Single Character from the Keyboard:

· Create a BufferedReader class object (br).
· Then read a single character from the keyboard using read() method as:
char ch = (char) br.read();

· The read method reads a single character from the keyboard but it returns its ASCII number,which is an integer. Since, this integer number cannot be stored into character type variable ch, we should convert it into char type by writing (char) before the method. int data type is converted into char data type, converting one data type into another data type is called type casting.

Accepting a String from Keyboard:

· Create a BufferedReader class object (br).
· Then read a string from the keyboard using readLine() method as:
String str = br.readLine ();

· readLine () method accepts a string from keyboard and returns the string into str. In this case, casting is not needed since readLine () is taking a string and returning the same data type.

Accepting an Integer value from Keyboard:

· First, we should accept the integer number from the keyboard as a string, using readLine ()
as: String str = br.readLine ();
· Now, the number is in str, i.e. in form of a string. This should be converted into an int by using parseInt () method, method of Integer class as:
int n = Integer.parseInt (str);

If needed, the above two statements can be combined and written as:
int n = Integer.parseInt (br.readLine());

· parseInt () is a static method in Integer class, so it can be called using class name as Integer.parseInt ().

· We are not using casting to convert String type into int type. The reason is String is a class and int is a fundamental data type. Converting a class type into a fundamental data type is not possible by using casting. It is possible by using the method Integer.parseInt().

Accepting a Float value from Keyboard:

· We can accept a float value from the keyboard with the help of the following statement:
float n = Float.parseFloat (br.readLine());

· We are accepting a float value in the form of a string using br.readLine () and then passing the string to Float.parseFloat () to convert it into float. parseFloat () is a static method in Float class.

Accepting a Double value from Keyboard:

· We can accept a double value from the keyboard with the help of the following statement:
double n = Double.parseDouble (br.readLine());

· We are accepting a double value in the form of a string using br.readLine () and then passing the string to Double.parseDouble () to convert it into double. parseDouble () is a static method in Double class.

Accepting Other Types of Values:

· To accept a byte value: byte n = Byte.parseByte (br.readLine ());
· To accept a short value: short n = Short.parseShort (br.readLine ());
· To accept a long value: long n = Long.parseLong (br.readLine ());
· To accept a boolean value: boolean x = Boolean.parseBoolean (br.readLine ());
If read () / readLine () method could not accept values due to some reason (like insufficient memory or illegal character), then it gives rise to a runtime error which is called by the name IOException, where IO stands for Input/Output and Exception represents runtime error.But we do not know how to handle this exception, in Java we can use throws command to throw the exception without handling it by writing:

throws IOException at the side of the method where read ()/ readLine () is used.

Program 1: Write a program to accept and display student details.

```
// Accepting and displaying student details.
import java.io.*;
class StudentDemo
{ public static void main(String args[]) throws IOException
{ // Create BufferedReader object to accept data
BufferedReader br =new BufferedReader (new InputStreamReader (System.in));
//Accept student details
System.out.print ("Enter roll number: ");
int rno = Integer.parseInt (br.readLine());
System.out.print (“Enter Gender (M/F): “);
char gender = (char)br.read();
br.skip (2);
System.out.print ("Enter Student name: ");
String name = br.readLine ()
System.out.println ("Roll No.: " + rno);
System.out.println ("Gender: " + gender);
System.out.println ("Name: " + name);}}
```