UNIT-2

Packages Packages and interfaces are two of the basic components of a Java program. In general, a Java source file can contain any (or all) of the following four internal parts:

* A single package statement (optional)
* Any number of import statements (optional)
* A single public class declaration (required)
* Any number of classes private to the package (optional)

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. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package.

Defining a Package:

* Creating 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. While the default package is fine for short, sample programs, it is inadequate for real applications. Most of the time, you will define a package for your code.

Creating a Package:

* The general form of the **package** statement:
  package *pkg*; Here, *pkg* is the name of the package.

* For example, the following statement creates a package called **MyPackage**.
  package MyPackage;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called My**Package**. Remember that case is significant, and the directory name must match the package name exactly.

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. 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.
For example, a package declared as **package java.awt.image;** needs to be stored in **java/awt/image**, **java\awt\image**,
Or
java:awt:image
on your
UNIX, Windows, or Macintosh file system, respectively. Be sure to choose your package names carefully.

* You cannot rename a package without renaming the directory in which the classes are stored.

Finding Packages and CLASSPATH:

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.
For example, consider the following package specification.
package MyPack;

In order for a program to find **MyPack**, one of two things must be true. Either the program is executed from a directory immediately above **MyPack**, or **CLASSPATH** must be set to include the path to **MyPack**.

The first alternative is the easiest (and doesn‘t require a change to **CLASSPATH**), but the second alternative lets your program find **MyPack** no matter what directory the program is in.

Create the package directories below your current development directory, put the **.class** files into the appropriate directories and then execute the programs from the development directory.

Example:
```
// A simple package
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();
}
}
```

Save this file as **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. Executing the **AccountBalance** class, using the following command line:

java MyPack.AccountBalance

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

java AccountBalance

AccountBalance must be qualified with its package name.

Access Protection:

Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java‘s smallest unit of abstraction. Because of the interplay between classes and packages, 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

Table: Class member access

	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

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

Example:

* This is file **Protection.java**:

```
package p1;
public class Protection
{
        int n = 1;
        private int n_pri = 2;
        protected int n_pro = 3;
        public int n_pub = 4;
        public Protection()
        {
        System.out.println("base constructor");
        System.out.println("n = " + n);
        System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

* This is file **Derived.java**:

```
    package p1;
    class Derived extends Protection
    {
            Derived()
            {
            System.out.println("derived constructor");
            System.out.println("n = " + n);
            // class only
            // System.out.println("n_pri = " + n_pri);
            System.out.println("n_pro = " + n_pro);
            System.out.println("n_pub = " + n_pub);
            }
    }
```

* This is file **SamePackage.java**:

```
    package p1;
    class SamePackage
    {
    SamePackage()
            {
            Protection p = new Protection();
            System.out.println("same package constructor");
            System.out.println("n = " + p.n);
            // class only
            // System.out.println("n_pri = " + p.n_pri);
            System.out.println("n_pro = " + p.n_pro);
            System.out.println("n_pub = " + p.n_pub);
            }
    }
```

* This is file **Protection2.java**:

```
    package p2;
    class Protection2 extends p1.Protection
    {
            Protection2()
            {
```

```
        System.out.println("derived other package constructor");
        // class or package only
        // System.out.println("n = " + n);
        // class only
        // System.out.println("n_pri = " + n_pri);
        System.out.println("n_pro = " + n_pro);
        System.out.println("n_pub = " + n_pub);
        }
}
```

* This is file **OtherPackage.java**:

```
package p2;
class OtherPackage
{
        OtherPackage()
        {
        p1.Protection p = new p1.Protection();
        System.out.println("other package constructor");
        // class or package only
        // System.out.println("n = " + p.n);
        // class only
        // System.out.println("n_pri = " + p.n_pri);
        // class, subclass or package only
        // System.out.println("n_pro = " + p.n_pro);
        System.out.println("n_pub = " + p.n_pub);
        }
}
```

* If you wish to try these two packages, here are two test files you can use. The one for package **p1** is shown here:

```
// Demo package p1.
package p1;
// Instantiate the various classes in p1.
public class Demo
{
        public static void main(String args[])
        {
        Protection ob1 = new Protection();
        Derived ob2 = new Derived();
        SamePackage ob3 = new SamePackage();
        }
}
```

* The test file for **p2** is shown next:

```
// Demo package p2.
```

```
package p2;
// Instantiate the various classes in p2.
public class Demo
 {
        public static void main(String args[])
        {
        Protection2 ob1 = new Protection2();
        OtherPackage ob2 = new OtherPackage();
        }
}
```

Importing Packages:

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 convenient.

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 (**.**). There is no practical limit on the depth of a package hierarchy, except that imposed by the file system. Finally, you specify either an explicit *classname* or a star (*****), which indicates that the Java compiler should import the entire package.

This code fragment shows both forms in use:

import java.util.Date;
import java.io.*;

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.

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

when a package is imported, only those items within the package declared as **public** will be available to non-subclasses in the importing code.

For example, if you want the **Balance** class of the package **MyPack** shown earlier to be available as a stand-alone class for general use outside of **MyPack**, then you will need to declare it as **public** and put it into its own file, as shown here:

```
package MyPack;

/* Now, the Balance class, its constructor, and its show() method are public. This
means that they can be used by non-subclass code outside their package.*/
public class Balance
{
        String name;
        double bal;
        public Balance(String n, double b)
        {
        name = n;
        bal = b;
        }
        public void show()
        {
        if(bal<0)
        System.out.print("--> ");
        System.out.println(name + ": $" + bal);
        }
}
```

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

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

Interfaces

Using the keyword interface, you can fully abstract a class' interface from its implementation.That is, using interface, you can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance varia bles, and their methods are declared without any body. In practice, this means that you can define interfaces which don‘t make assumptions about how they are implemented. 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. Each class can determine the details of its own implementation. 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. This requirement by itself makes for a static and no extensible classing environment. Inevitably in a system like this, functionality gets pushed up higher and higher in the class hierarchy so that the mechanisms will be available to more and more subclasses. Interfaces are designed to avoid this problem. They disconnect the definition of a method or set of methods from the inheritance hierarchy. 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.

Interfaces add most of the functionality that is required for many applications which would normally resort to using multiple inheritance in a language such as C++.

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.

An example of an interface definition. It declares a simple interface which contains one method called **callback( )** that takes a single integer parameter.

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

<u>Implementing Interfaces:</u>

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 implements the interface:

*

```
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.

Example: class that implements the Callback interface shown earlier.

```
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.
For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
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:

We can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to. This is one of the key features of interfaces. The method to be executed is looked up dynamically at run time, allowing classes to be created later than the code which calls methods on them. The calling code can dispatch through an interface without having to know anything about the ―callee.‖ This process is similar to using a superclass reference to access a subclass object.

Because dynamic lookup of a method at run time incurs a significant overhead when compared with the normal method invocation in Java, you should be careful not to use interfaces casually in performance-critical code.

The following example calls the callback( ) method via an interface reference variable:

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

Output:
```
callback called with 42.
```

Notice that variable c is declared to be of the interface type Callback, yet it was assigned an instance of Client. Although c can be used to access the callback( ) method, it cannot access any other members of the Client class. An interface reference variable only has knowledge of the methods declared by its interface declaration.

Thus, c could not be used to access nonIfaceMeth( ) since it is defined by Client but not Callback.

The preceding example shows, mechanically, how an interface reference variable can access an implementation object, it does not demonstrate the polymorphic power of such a reference. To sample this usage, first create the second implementation of Callback, shown here:

```
// Another implementation of Callback.

class AnotherClient implements Callback
{
public void callback(int p)
        {
        System.out.println("Another version of callback");
        System.out.println("p squared is " + (p*p));
        }
}
class TestIface2
{
        public static void main(String args[])
        {
        Callback c = new Client();
        AnotherClient ob = new AnotherClient();
        c.callback(42);
        c = ob; // c now refers to AnotherClient object
        c.callback(42);
        }
}
```
Output:
```
callback called with 42
Another version of callback
p squared is 1764
```

As you can see, the version of callback( ) that is called is determined by the type of object that c refers to at run time.

If a class includes an interface but does not fully implement the methods defined by that interface, then that class must be declared as abstract.

For example:

```
abstract class Incomplete implements Callback
{
        int a, b;
        void show()
        {
        System.out.println(a + " " + b);
        }
}
```

Here, the class Incomplete does not implement callback( ) and must be declared as abstract. Any class that inherits Incomplete must implement callback( ) or be declared abstract itself.

Applying Interfaces:

We define a stack interface, leaving it to each implementation to define the specifics. Let‘s look at two examples.
First, here is the interface that defines an integer stack. Put this in a file called IntStack.java. This interface will be used by both stack implementations.
Example:

```
// Define an integer stack interface.
interface IntStack
{
        void push(int item); // store an item
        int pop(); // retrieve an item
}
```

The following program creates a class called FixedStack that implements a fixed-length version of an integer stack:
Example:

```
// An implementation of IntStack that uses fixed storage.
class FixedStack implements IntStack
{
        private int stck[];
        private int tos;
        FixedStack(int size)
        stck = new int[size];
        tos = -1;
}
// Push an item onto the stack
public void push(int item)
{
        if(tos==stck.length-1) // use length member
```

```
            System.out.println("Stack is full.");
            else
            stck[++tos] = item;
}

// Pop an item from the stack
public int pop()
{
            if(tos < 0)
            {
            System.out.println("Stack underflow.");
            return 0;
            }
            else
            return stck[tos--];
            }
}
class IFTest
 {
            public static void main(String args[])
            {
            FixedStack mystack1 = new FixedStack(5);
            FixedStack mystack2 = new FixedStack(8);
            // push some numbers onto the stack
            for(int i=0; i<5; i++) mystack1.push(i);
            for(int i=0; i<8; i++) mystack2.push(i);
            // pop those numbers off the stack
            System.out.println("Stack in mystack1:");
            for(int i=0; i<5; i++)
            System.out.println(mystack1.pop());
            System.out.println("Stack in mystack2:");
            for(int i=0; i<8; i++)
            System.out.println(mystack2.pop());
            }
}
```

Following is another implementation of IntStack that creates a dynamic stack by use of the same interface definition. In this implementation, each stack is constructed with an initial length. If this initial length is exceeded, then the stack is increased in size. Each time more room is needed, the size of the stack is doubled.

Example:

```
// Implement a "growable" stack.
class DynStack implements IntStack
```

```
{
private int stck[];
private int tos;
// allocate and initialize stack
DynStack(int size)
{
stck = new int[size];
tos = -1;
}
// Push an item onto the stack
public void push(int item)
{
// if stack is full, allocate a larger stack
if(tos==stck.length-1)
{
int temp[] = new int[stck.length * 2]; // double size
for(int i=0; i<stck.length; i++)
temp[i] = stck[i];
stck = temp;
stck[++tos] = item;
}
else
stck[++tos] = item;
}
// Pop an item from the stack
public int pop()
 {
if(tos < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[tos--];
}
}
class IFTest2
{
public static void main(String args[])
 {
DynStack mystack1 = new DynStack(5);
DynStack mystack2 = new DynStack(8);
// these loops cause each stack to grow
for(int i=0; i<12; i++) mystack1.push(i);
for(int i=0; i<20; i++) mystack2.push(i);
System.out.println("Stack in mystack1:");
for(int i=0; i<12; i++)
System.out.println(mystack1.pop());
System.out.println("Stack in mystack2:");
for(int i=0; i<20; i++)
System.out.println(mystack2.pop());
}
}
```

The following class uses both the FixedStack and DynStack implementations.It does so through an interface reference. This means that calls to push( ) and pop( )are resolved at run time rather than at compile time.

```
/* Create an interface variable and access stacks through it.
*/
class IFTest3
{
public static void main(String args[])
{
IntStack mystack; // create an interface reference variable
DynStack ds = new DynStack(5);
FixedStack fs = new FixedStack(8);
mystack = ds; // load dynamic stack
// push some numbers onto the stack
for(int i=0; i<12; i++)
mystack.push(i);
mystack = fs; // load fixed stack
for(int i=0; i<8; i++)
mystack.push(i);
mystack = ds;
System.out.println("Values in dynamic stack:");
for(int i=0; i<12; i++)
System.out.println(mystack.pop());
mystack = fs;
System.out.println("Values in fixed stack:");
for(int i=0; i<8; i++)
System.out.println(mystack.pop());
}
}
```

In this program, mystack is a reference to the IntStack interface. Thus, when it refers to ds, it uses the versions of push( ) and pop( ) defined by the DynStack implementation. When it refers to fs, it uses the versions of push( ) and pop( ) defined by FixedStack.

These determinations are made at run time. Accessing multiple implementations of an interface through an interface reference variable is the most powerful way that Java achieves run-time polymorphism.

Variables in Interfaces:

Interfaces can be used to import shared constants into multiple classes by simply declaring an interface that contains variables which are initialized to the desired values. When we include that interface in a class (that is, when you ―implement‖ the interface), all of those variable names will be in scope as constants. This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations. If an interface contains no methods, then any class that includes such an interface doesn‘t actually implement anything. It is as if that class were importing the constant variables into the class name space as final variables

Example:

```
import java.util.Random;
interface SharedConstants
 {
int NO = 0;
int YES = 1;
int MAYBE = 2;
int LATER = 3;
int SOON = 4;
int NEVER = 5;
}
class Question implements SharedConstants
 {
Random rand = new Random();
int ask()
 {
int prob = (int) (100 * rand.nextDouble());
if (prob < 30)
return NO; // 30%
else if (prob < 60)
return YES; // 30%
else if (prob < 75)
return LATER; // 15%
else if (prob < 98)
return SOON; // 13%
else
return NEVER; // 2%
}
}
class AskMe implements SharedConstants
{
static void answer(int result)
 {
switch(result)
 {
case NO:
```

```
System.out.println("No");
break;
case YES:
System.out.println("Yes");
break;
case MAYBE:
System.out.println("Maybe");
break;
case LATER:
System.out.println("Later");
break;
case SOON:
System.out.println("Soon");
break;
case NEVER:
System.out.println("Never");
break;
}
}
public static void main(String args[])
{
Question q = new Question();
answer(q.ask());
answer(q.ask());
answer(q.ask());
answer(q.ask());
}
}
```

Output:

```
Later
Soon
No
Yes
```

This program makes use of one of Java‘s standard classes: Random. This class provides pseudorandom numbers. It contains several methods which allow you to obtain random numbers in the form required by your program. In this example, the method nextDouble( ) is used. It returns random numbers in the range 0.0 to 1.0.

In this sample program, the two classes, Question and AskMe, both implement the SharedConstants interface where NO, YES, MAYBE, SOON, LATER, and NEVER are defined. Inside each class, the code refers to these constants as if each class had defined or inherited them directly.

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.

Example:

```
// One interface can extend another.
interface A
 {
void meth1();
void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A
{
void meth3();
}
// this class must implement all of A and B
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();
}
}
```

Any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

IO STREAMS

I/O Streams

Stream:
Stream is an Abstract representation of data connected to some I/O device.

I/O in java is categorized into two types:
- Byte Streams
- Character Streams

Byte Streams

Byte Streams deals with writing or reading of data to or from the I/O device as bytes.

Ex. Working with Primitive Data Types,objects.

Base Abstract classes for Byte streams are:
- InputStream
- OutputStream

Character Streams

Character Streams deals with writing or reading of data to or from the I/O device as characters.

Ex. Working with text data (strings)

Base Abstract classes for Character streams are:
- Reader
- Writer

Methods of InputStream

int available() : It is used to return the estimated number of bytes that can be read from the input stream.

void close() : It is used to closes the stream.

void mark(int numBytes) : used to mark the current location.

boolean markSupported() : returns true or false which tells whether the invoking stream supports marking fecility or not.

int read() : It is used to read the byte of data from the input stream.

int read(byte buf[]) : It is used to read up to **b.length** bytes of data from the input stream.

int read(byte buf[], int offset, int len) : It is used to read up to **len** bytes of data from the input stream and stores at buf at the specified offset.

void reset() : used to reset the file hanled to the marked location.

long skip(long numbytes) : It is used to skip over and discards x bytes of data from the input stream.

Methods of OutputStream

void write(int b): It is used to write a byte of data to the input stream.

void write(byte buf[]): It is used to write b.length bytes of data to the output stream.

void write(byte buf[], int offset, int numbytes): It is used to write len bytes of data to the output stream from buf with the specified offset.

void flush(): clears the output buffer.

void close(): It is used to closes the stream.

Byte Stream Classes

FileInputStream,FileOutputStream
BufferedInputStream, BufferedOutputStream
DataInputStream,DataOutputStream
ObjectInputStream,ObjectOutputStream
SequenceInputStream
PrintStream
ByteArrayInputStream, ByteArrayOutputStream
ZipInputSteam,ZipOutputStream

PipedInputStream,PipedOutputStream

FileInputStream

Extends 'InputStream' class

Creates an InputStream that we can use to read bytes from a file.

Constructors
FileInputStream(String filepath)
FileInputStream(String fileobj)

Obtaining Input Stream Reference for a file

```
FileInputStream fis;

fis = new FileInputStream("ex1.txt");
```

(or)

```
File file = new File("ex1.txt");

FileInputStream fis = new FileInputStream(file);
```

Reading a byte at a time from the file until the end of File

```
FileInputStream fis = new FileInputStream("ex1.txt");

int i=fis.read()

while(i!=-1)
{
	System.out.println((char)i);
	i=fis.read();
}

fis.close();
```

Reading a byte array from the file

```
FileInputStream fis = new FileInputStream("ex1.txt");

byte x[] = new byte[40];

int n = fis.read(x);

System.out.println("No. of bytes read = " +n );
System.out.println(“Data is : “+ new String(x,0,n));
```

Note:

Most of the methods that operate on I/O devices throws IOException (checked exception) . So put all that I/O code in try block and process the exception.

mark(), reset() are not overridden, any attempt to call reset() on FileInputStream throws IOException.

Program1:

```
//demonstrates how to read a byte at a time from the file and display on the monitor.

import java.io.*;

class FileRead1
{
  public static void main(String[] args) throws IOException,FileNotFoundException
  {
    int ch;

    FileInputStream fis=null;

    //Attach file name to FileInputStream
    fis = new FileInputStream("ex1.txt");

     System.out.println("Contents of the file....");

   //Read characters from fis into ch,and display it on the screen,repeat this till end of
the file is reached

   while(true)
   {
      ch=fis.read();
      if(ch==-1)
          break;
     System.out.print((char)ch);
   }

   //close the file handle.
   fis.close();
}
}
```

Program2:

//To demonstrate how to Read the content of the given file byte by byte and display on the monitor.(user Scanner class to get file name from user)

```
import java.io.*;
import java.util.*;

class FileRead2
{
  public static void main(String[] args)throws IOException
  {
    int ch;
    Scanner s=new Scanner(System.in);
    FileInputStream fis=null;
    System.out.println("Enter the file name (ex1.txt or ex2.txt)");
    String fname=s.next();
    //Attach file name to FileInputStream
     fis = new FileInputStream(fname);
     System.out.println("Contents of the file....");

   //Read characters from fis into ch,and display it on the screen,repeat this till end of
the file is        reached
   while(true)
   {
      ch=fis.read();
      if(ch==-1)
        break;
      System.out.print((char)ch);
     }
    fis.close();
 }
}
```

<u>Program3:</u>

/* Read the content of the file ex2.txt, count and display number of characters,words and lines present in the File.

create the following text file (dont press enter after the third line)

<u>ex2.txt</u>
shiva rama krishna
younus shariff
tejaswita

Solution:*/

```
import java.io.*;
import java.util.*;

class CountCWL
{
  public static void main(String[] args)throws IOException
  {
    int ch,ccount=0,wcount=0,lcount=0;
    FileInputStream fis=null;

    //Attach file name to FileInputStream
    fis = new FileInputStream("ex2.txt");

   //Read characters from fis into ch and check whether the character is a new line
character
   // or space character or any other character.
   while((ch=fis.read())!=-1)
   {
      ccount++;
       if((char)ch==' ')
       wcount++;

      if((char)ch=='\n')
         {
          lcount++;
           wcount++;
         }
     }
  fis.close();

  System.out.println("Number of characters in the file:"+ccount);
  System.out.println("Number of words in the file:"+wcount);
  System.out.println("Number of lines in the file:"+lcount);
   }
}
```

input: no input

output:

```
Number of characters in the file:46
Number of words in the file:6
Number of lines in the file:3
```

FileOutputStream

Creates an output stream that you can use to write bytes to a file.

Constructors

```
FileOutputStream(String  pathname);
FileOutputStream(File fileObj);
FileOutputStream(String  pathname, boolean append);
FileOutputStream(File fileObj, boolean append);
```

All can throw FileNotFoundException.

Writing to a file

```
FileOutputStream fos =  new FileOutputStream(“ex1.txt”);

String s = “Hello kmit”;

byte b[] = s.getBytes();

for(i=0;i<b.length;i++)
fos.write(b[i]);
```

Example_Program:

//Retrieve characters from a string variable and write them a byte at a time onto a file.

```
import java.util.*;
import java.io.*;
class FileWrite1
{
 public static void main(String[] args) throws IOException
 {
   int ch;int ch2;
   //Attch file name to FileOutputStream
   FileOutputStream fos=new FileOutputStream("fil1.txt");

   String s="hello welcome to kmit";

   byte b[]=s.getBytes();

   for(int i=0;i<b.length;i++)
      fos.write(b[i]);

   fos.close();

    /*
   //displaying the content of newly creted file
   FileInputStream fis=new FileInputStream("file19.txt");
   System.out.println("The content of the file file1.txt:");
   while(true)
    {
      ch2=fis.read();
      if(ch2==-1)
         break;
      System.out.print((char)ch2);
    }
   fis.close();*/
 }
}
```

Example_program:

```
/*Copy the content of the file ex1.txt to abc.txt a byte at a time.Display the content of
abc.txt file.

create the following file

ex1.txt
shivaram

Solution:
*/

import java.io.*;
class FileCopy1
{
  public static void main(String[] args)throws IOException
   {
    int ch;

    FileInputStream fis=null,fis2=null;
    FileOutputStream fos=null;

    //Attach file name to FileInputStream
    fis = new FileInputStream("ex1.txt");
      //Attach file name to FileOutputStream
   fos=new FileOutputStream("abc.txt");

   //Read characters from fis into ch,convert to uppercase and write onto fos,repeat
this till end  of the file is reached
   while(true)
   {
    ch=fis.read();

     if(ch==-1)
       break;
     fos.write(ch);
   }

   fis.close();
   fos.close();

   fis2=new FileInputStream("abc.txt");

  //  System.out.println("The content of abc.txt file is:");

   //Read characters from fis2 into ch and write onto the monitor,repeat this till end of
the file is reached
   while(true)
   {
      ch=fis2.read();

     if(ch==-1)
        break;
      System.out.print((char)ch);

   }

   fis2.close();
}
}
```

input: no input

output:
```
SHIVARAMê
```

Filtere Byte Streams

Filtered streams are simply wrappers around underlying input or output streams that transparently provide some extended level of functionality.

EX: Buffering, raw data translation, character translation.

The Filtered byte streams are
    FilterInputStream
    FilterOutputStream

BufferedInputStream

Extends FilterInputStream

Buffering I/O is a performance optimization.

BuferedInputStream allows us to wrap any input stream to improve performance.

Low level system can read blocks of data from the disk or network and store the results in the buffer.

Even if we are reading the data byte at a time out of input stream, we will be accessing the fast memory most of the time.

Constructors

BuferedInputStream(InputStream is)

BufferedInputStream(InputStream is, int bufsize);

Supports mark() and reset(). These are very useful for pattern matching.

Creating Buffered InputStream for File

```
FileInputStream fis =  new FileInputStream("ex1.txt");

BufferedInputStream bis =  new BufferedInputStream(fis);
```

Reading Data from File

```
int i= bis.read();
while(i!=-1)
{
        System.out.println((char)i);
        i=bis.read();
}
bis.close();
```

Example_program:

//To demonstrate how to read the content of the file using BufferedInputStream class.

```
import java.io.*;

class FileRead3
{
  public static void main(String[] args) throws IOException
  {
    int ch;

    FileInputStream fis=null;
    BufferedInputStream bis=null;
    //Attach file name to FileInputStream
    fis = new FileInputStream("ex1.txt");

    bis=new BufferedInputStream(fis);
     System.out.println("Contents of the file....");
```

```
   //Read characters from fis into ch,and display it on the screen,repeat this till end of
the file is reached

   while(true)
   {
      ch=bis.read();
      if(ch==-1)
          break;
     System.out.print((char)ch);
   }

   //close the file handle.
   bis.close();
}
}
```

BufferedOutputStream

Extends FilterOutputStream.

Improves performance by reducing the number of times the system actually writes the data to the destination output stream.

Constructors

BufferedOutputStream(OutputStream os);

BufferedOutputStream(OutputStream os, int bufsize);

Writing to a File using Buffered Streams

```
FileOuputSream  fos =  new FileOuputSream("ex1.txt");

BufferedOutputSream bos =  new BufferedOutputStream(fos);

String s = "Hello kmit";
Byte buf[] = s.getBytes();

for(int i=0; i<buf.length;i++)
     bos..write(buf[i]);
```

Example Program:

/*Read one character at a time from the file ex1.txt, convert to uppercase and then write onto the file abc2.txt. Display the content of abc2.txt file.

[use BufferedInputStream and BufferedOutputStream classes].

create the following file

<u>ex1.txt</u>
shivaram

Solution:
*/

```
import java.io.*;
class FileCopy2
{
  public static void main(String[] args)throws IOException
  {
    int ch;

    FileInputStream fis=null,fis2=null;
    FileOutputStream fos=null;
    BufferedInputStream bis=null,bis2=null;
    BufferedOutputStream bos=null;
    //Attach file name to FileInputStream
    fis = new FileInputStream("ex1.txt");
    bis=new BufferedInputStream(fis);
      //Attach file name to FileOutputStream
   fos=new FileOutputStream("abc2.txt");
   bos=new BufferedOutputStream(fos);
   //Read characters from bis into ch,convert to uppercase and write onto bos,repeat
this till end  of the file is reached
   while(true)
   {
    ch=bis.read();

     if(ch==-1)
       break;
     ch=ch-32;
     bos.write(ch);
   }

   bis.close();
   bos.close();

   fis2=new FileInputStream("abc2.txt");
   bis2=new BufferedInputStream(fis2);
 //  System.out.println("The content of abc.txt file is:");
```

```
   //Read characters from fis2 into ch and write onto the monitor,repeat this till end of
the file is reached
   while(true)
   {
      ch=bis2.read();

     if(ch==-1)
       break;
      System.out.print((char)ch);
   }
  bis2.close();
}
}
```

input: no input
output:
```
SHIVARAMê
```

DataOutputStream

Extends FilterOutputStream, implements DataOutput interface.

Enable us to write any primitive data type value at a time .

Constructor :

DataOutputStream(OuputStream os)

Writing to a file using DataOutputStream

```
FileOutputStream fis =  new FileOutputStream("ex1.txt");
DataOutputStream dos = new DataOutputStream(fos);
dos.writeInt(432);
dos.writeDouble(43.123);
dos.writeBoolean(true);
dos.close();
```

DataInputStream

Enables to read primitive data type values from the underlying input stream.

It extends FilterInputStream and implements DataInput interface

Constructor :

DataInputStream(InputStream is);

Reading from file a File using DataInputStream

```
FileInputStream fis =new FileInputStream(“ex1.txt”);

DataInputStream dis =new DataInputStream(fis);

int x=dis.readInt()
double y=dis.readDouble()
boolean z=dis.readBoolean()

System.out.println(x);
System.out.println(y);
System.out.println(z);
```

Example_Program:

```
//To demonstrate how to read characters from keyboard/user(using DataInputStream)
and write onto a File.
import java.util.*;
import java.io.*;
class FileWrite2
{
 public static void main(String[] args) throws IOException
 {
   int ch;char ch2;
   //Attch file name to FileOutputStream
   FileOutputStream fos=new FileOutputStream("fil2.txt");
   DataInputStream dis=new DataInputStream(System.in);
   System.out.println("enter characters,# to stop");
  while((ch=(char)dis.read())!='#')
   {
     fos.write(ch);
   }

   fos.close();
   /*
   //displaying the content of newly creted file
   FileInputStream fis=new FileInputStream("file19.txt");
   System.out.println("The content of the file file1.txt:");
   while(true)
```

```
    {
      ch2=fis.read();
      if(ch2==-1)
        break;
      System.out.print((char)ch2);
    }
   fis.close();*/
 }
}
```

Example_Program:

//To demonstrate how to read integer values from keyboar and write them onto a file(DataOutputStream).

```
import java.io.*;
import java.util.*;
class FileWrite3
{
 public static void main(String[] args) throws IOException
 {
   int i,x;
   //Attach keyboard to DataInputStream
   Scanner s=new Scanner(System.in);
   //Attch file name to FileOutputStream
   FileOutputStream fos=new FileOutputStream("file1.dat");
   DataOutputStream dos=new DataOutputStream(fos);

   //reading 5 integers from keyboard & writing them onto a file.
   for(i=1;i<=5;i++)
    {
     System.out.println("enter the number:");
     x=s.nextInt();
     dos.writeInt(x);
    }
    dos.close();
}
}
```

Example_Program:

//To demonstrate how to retrieve all integer values present in the file(using DataInputStream) and display on the monitor.

```
import java.io.*;
import java.util.*;
class FileRead4
{
 public static void main(String[] args) throws IOException
 {
   int x;
   //retieving the integer values from the file and displaying on console
   FileInputStream fis=new FileInputStream("file4.dat");
   DataInputStream dis2=new DataInputStream(fis);
   while(dis2.available()>0)
    {
      x=dis2.readInt();
      System.out.println(x);
    }
   dis2.close();
}
}
```

Example_Program:

```
/*(Q)Read int values from the file 'file1.dat' and write all even numbers into the file
'even.dat', odd numbers into the file 'odd.dat'.Display the contents of the file 'odd.dat'.

@admin:
ensure the file file1.dat is existing.

The contents of the file(11,22,33,44,55) should not be manually created.has to be
written to the file using DataInputStream class writeInt().

solution:
*/
import java.io.*;
import java.util.*;

class FileRW
{
 public static void main(String[] args) throws IOException
 {
   int i,x;
   FileOutputStream fos1=new FileOutputStream("even.dat");
   DataOutputStream dos1=new DataOutputStream(fos1);

  FileOutputStream fos2=new FileOutputStream("odd.dat");
  DataOutputStream dos2=new DataOutputStream(fos2);

 //retieving the integer values from the file
```

```
   FileInputStream fis=new FileInputStream("file1.dat");
   DataInputStream dis=new DataInputStream(fis);
   while(dis.available()>0)
    {
     x=dis.readInt();
     if(x%2==0)
        dos1.writeInt(x);
     else
        dos2.writeInt(x);
    }
   dis.close();
   dos1.close();
   dos2.close();

   System.out.println("the odd numbers from odd.dat file:");
   FileInputStream fis2=new FileInputStream("odd.dat");
   DataInputStream dis2=new DataInputStream(fis2);
   while(dis2.available()>0)
     {
     x=dis2.readInt();
     System.out.print(x+" ");
     }
   dis2.close();
   }
}
```

output:

```
the odd numbers from odd.dat file:
11 33 55
```

Serialization

Process of writing state of an object to a byte stream.

Saving the object in a persistant storage.

Only the objects that implement serializable interface can be serialized.

The serializable interface has no methods, it only indicates that the objects of this can be serialized.

If a class is Serializable, all of its sub classes are also serializable.

Transient attributes will not be serialized.

Static attributes also will not be serialized.

If the object to be serialized has references to other objects, which in turn have reference to still more objects. This set of objects and the relation ships among them form a directed graph.

There may also be circular references within this object graph ie object x may contain reference to object y, and y may have references to back to x.

Objects may also contain references to themselves.

The object serialization and deserialization facilities have been designed to work correctly in these scenarios.

If you attempt to serialize an object at the top of the object graph, all of the referenced objects are recursively located and serialized.

At deserialization all these objects and their references are corectly restored.

It is legal to serialize an object of type that has a super type that does not implement Serializable interface.

At the time of deserialization jre invokes the constructors of those base classes which do not implement Serializable interface.

An object Serialized on one JVM can be successfully deserialized on another JVM.

ObjectOutputStream

Extends OutputStream class, implements ObjectOutput interface.

Enables us to write entire object at once.

Constructors

ObjectOutputStream(OutputStream os) throws IOException;

Important method

void WriteObject(Object obj);

ObjectInputStream

Extends InputStream class, implements ObjectInput interface

Enables us to read entire object at once.

Constructors

ObjectInputStream(InputStream is) throws IOException;

Important method

Object readObject();

Example

```
public class Employee implements Serializable
{
        String name;
        int id;
        public Employee(String name,int id)
        {
                this.name=name;
                this.id=id;
        }
        public String toString()
        {
                return "NAME:" +name+" ID:" +id;
        }
}
```

Writing Object into a File

```
FileOutputStream fos = new FileOutputStream("ex1.dat");
ObjectOutputStream oos = new ObjectOutputStream(fos);

Employee e1 = new Employee("Shiva",263);

oos.writeObject(e1);

oos.close();
```

Retrieving Object from the file

```
FileInputStream fis = new FileInputStream("ex1.dat");

ObjectInputStream ois = new ObjectInputStream(fis);
```

```
Employee e =(Employee) ois.readObject();

System.out.println(e);

ois.close();
```

Example_Program1:

```
//To demonstrate how to write object onto a File.

import java.io.*;
import java.util.*;

public class Student implements Serializable
{
   public String rno;
   public String name;

   public Student(String rno,String name)
   {
     this.rno=rno;
     this.name=name;
   }
   public String toString()
   {
      return "ROLL NO="+rno+" NAME="+name;
   }
}

class FileWrite4
{
   public static void main(String args[]) throws
     IOException,ClassNotFoundException
   {
      FileOutputStream fos=new FileOutputStream("student.dat");
      ObjectOutputStream oos=new ObjectOutputStream(fos);

      Student s1=new Student("15BD1A0501","shiva");
      oos.writeObject(s1);

      Student s2=new Student("15BD1A0502","Younus");
      oos.writeObject(s2);
```

```
    Student s3=new Student("15BD1A0503","kiran");
    oos.writeObject(s3);

    oos.close();
}
}
```

Example_Program:

```
//To demonstrate how to retrieve and display all the objects present in the file.

import java.io.*;
import java.util.*;

class FileRead5
{
  public static void main(String args[]) throws
   IOException,ClassNotFoundException
  {
    FileInputStream fis=new FileInputStream("student.dat");
    ObjectInputStream ois=new ObjectInputStream(fis);
    Student s=null;
    while(true)
      {
       try
       {
       s=(Student)ois.readObject();
       System.out.println(s);
       }
       catch(EOFException e)
       {
         break;
       }
      }
   }
}
```

Example_Program:

/*(Q)The Serializable class Student is with 2 attributes rno(String), name(String).The details(rno,name) of 3 students with roll numbers 15BD1A0501, 15BD1A0502, 15BD1A0503 is saved in the file 'student.dat'.Display the name of the student for the given roll number.

Student.java

```
import java.io.*;

public class Student implements Serializable
{
  public String rno;
  public String name;

  public Student(String rno,String name)
  {
   this.rno=rno;
   this.name=name;
  }
  public String toString()
  {
    return "ROLL NO="+rno+" NAME="+name;
  }
}
```

<u>ObjectIO.java</u>

```
import java.io.*;
import java.util.*;

class FileRead6
{
  public static void main(String args[]) throws
   IOException,ClassNotFoundException
  {
    /*
    FileOutputStream fos=new FileOutputStream("student.dat");
    ObjectOutputStream oos=new ObjectOutputStream(fos);

    Student s1=new Student("15BD1A0501","shiva");
    oos.writeObject(s1);
    Student s2=new Student("15BD1A0502","Younus");
    oos.writeObject(s2);
    Student s3=new Student("15BD1A0503","kiran");
    oos.writeObject(s3);
    oos.close();*/

    Scanner s=new Scanner(System.in);
    System.out.println("enter the student roll number:");
    String no=s.next();
    String sname="";
```

```
        FileInputStream fis=new FileInputStream("student.dat");
        ObjectInputStream ois=new ObjectInputStream(fis);
        Student ss;
        int found=0;
        while(true)
          {
            try
            {
            ss=(Student)ois.readObject();

            if(no.equals(ss.rno))
              {
               found=1;
               sname=ss.name;
               break;
               }
            }
            catch(EOFException e)
            {
              break;
            }
          }

        ois.close();
        if(found==0)
           System.out.println("No such student found");
        else
           System.out.println("STUDENT NAME="+sname);
       }
}
```

input1:

```
enter the student roll number:
15BD1A0503
```

output1:

```
STUDENT NAME=kiran
```

input2

```
enter the student roll number:
15BD1A0504
```

output2:

```
No such student found
```

SequenceInputStream

Used to concatenate multiple InputStreams. Extends from InputStream class.

Constructors

```
SequenceInputStream(InputStream is1,InputStream is2);

SequenceInputStream(Enumeration e);
```

Operationally the class fulfills read request from the first InputStream until it runs out and then switch over to the next one.

Creating Enumeration of InputStreams

```
FileInputStream fis1 =  new FileInputStream("ex1.txt");

FileInputStream fis2 =  new FileInputStream("ex2.txt");

FileInputStream fis3 =  new FileInputStream("ex3.txt");

Vector v = new Vector();

v.add(fis1);
v.add(fis2);
v.add(fis3);

Enumeration e = v.elements();
```

Constructing SequenceInputStream & Reading from all the three files

```
SequenceInputStream sis = new SequenceInputStream(e);

int i = sis.read();
while(i!=-1)
{
        System.out.println((char)i);
        i=sis.read();
}
sis.close();
```

Example_Program:

```
// To demonstrate how to concatenate multiple input streams

import java.io.*;
import java.util.*;
```

```
class FileRead7
{
public static void main(String args[]) throws IOException
{

FileInputStream fis1=new FileInputStream("seq1.txt");
FileInputStream fis2=new FileInputStream("seq2.txt");
FileInputStream fis3=new FileInputStream("seq3.txt");

Vector v=new Vector();
v.add(fis1);
v.add(fis2);
v.add(fis3);

Enumeration files=v.elements();

SequenceInputStream sis=new SequenceInputStream(files);
int i;
System.out.println("The content of the three files :");
while(true)
{
  i=sis.read();
  if(i==-1)
    break;
  System.out.print((char)i);
}
sis.close();
}
}
```

PrintStream

Extends FilterOutputStream.

Provides all output capabilities which we have used with System.out.

Supports print(),println() for all types including objects.

jdk5 added printf(),format() which allows to specify precise format of data.

Constructors

PrintStream(OutputStream os)

PrintStream(OutputStream os, boolean flushOnNewline);

PrintStream(File outputfile)

PrintStream(String outputfilename);

Note:
With the last constructor, the file with the given outputfilename will automatically created.

If a file exists with the given name, it will be destroyed

Can flush the output data automatically so no need call flush() explicitly.

Its methods do not throw IOExeption.

Because System.out is of type PrintStream class, we can call printf().

Examples:

```
System.out.println("Hello world");

System.out.printf("x=%d y=%f ",10,20.3f);

System.out.printf("x=%,.2f",123456.123);
```

Output:
```
Hello world
x=10 y=20.300000
x=1,234,567.12
```

Example_Program:

```
import java.io.*;

class PrintStreamDemo
{
public static void main(String[] args) throws IOException
   {

System.out.println("hello world");

System.out.printf("%f%n",1234567.3456);
System.out.printf("%,f%n",1234567.3456);
System.out.printf("%10.2f%n",1234567.3456);
PrintStream out = new PrintStream(System.out);
double d = 0.1/0.3;
String s = "shiva";

out.format("NAME=%s VAL=%7.3f%n",s,d);

int x=10,y=20;
out.printf("%2$d %3$s %1$d%n", x, y, "is bigger than");
   }
}
```

Other ByteStream Classes

PipedInputStream & PipedOutputStream

Used for inter thread communication.

PipedInputStream
Used by the thread which reads the data from the pipe.

PipedOutputStream
Used by the thread which writes the data to the pipe.

ZipInputStream & ZipOutputStream

ZipInputStream
Enable us to extract the file entries from the zip file.

ZipOutputStream
Helps us to create a zip file by adding file entries to it.

ByteArrayInputStream & ByteArrayOutputStream

ByteArrayInputStream
Used when the data source is a byte Array variable.

ByteArrayOutputStream
Used when the data is to be written onto a byte Array variable.

File Class

Abstract representation of File & Directory pathnames.

Not used to actually read or write data.

Work at higher level for performing the following:

- Making new empty files
- Searching for files
- Deleting files
- Making directories

constructors

File(String pathname)

File(File Parent, String child)

File(String parent,String child)

With Constuctor1,if the file exists with the given filename(pathname) then a new file object is created to that existing filename.

If the filename do not exist, it just creates a File object representing that new file name. Does not physically creates a new file.

You have to call explicitly createNewFile() method on that File object to physically create a file on the hard disc.

With Constuctor2,the First parameter represents the directory(parent), second parameter represent the filename(child).

If the file exists with the given filename(child) and in the given parent directory, then the File Constructor creates a new File object representing that file.

If child(filename) is null, the File constructor throws an exception at runtime.

If the parent is null, the File object is created. If you call createNewFile() method to physically create a file, the new file is created in the current folder.

If the parent does not exists, the File object is created. If you call createNewFile() method to physically create a file, throws an exception at runtime.

Operations performed on a File

isFile()
length()
exists()
lastModified()
getName()
createNewFile()
getparent()
renameTo()
GetAbsolutePath()
delete()
canRead()
isDirectory()
canWrite()

<u>Operations performed on a Directory (File)</u>

isDirectory()
list()
listFiles()
listFiles(Filename Filter obj)
mkdir()
mkdirs()
createNewFile()
renameTo()
delete()

FilenameFilter

String[] list(FilenameFilter FFObj)

Argument for this method must be object of the class that implements FilenameFilter interface .

It has only one method

boolean accept(File directory,String filename)

This method is called automatically called once for each file in the directory.

Any file will be added to the returned list of files by the list() only when the accept() for that file entry returns true.

<u>Retrieving only HTML files from the directory</u>

public class OnlyExt implements FilenameFilter

```
{
String ext;

public onlyExt(String ext)
   {
  this.ext=“.”+ext;
   }

 public boolean accept(File dir,String name)
    {
        return name.endsWith(ext);
     }
}

File f1 = new File(“/Java”);

FilenameFilter ff = new onlyExt(“html”);

String fnames[] = f1.list(ff);

for(i=0;i<fnames.length;i++)
        System.out.println(fnames[i]);
```

<u>Example_Program:</u>

```
// To demonstrate how retrieve information about a file.
import java.io.*;

class File1
{
public static void main(String args[])
{
File f1=new File("nnn3/demo1.java");
System.out.println("NAME: "+f1.getName());
System.out.println("PATH: "+f1.getPath());
System.out.println("ABSOLUTE PATH: "+f1.getAbsolutePath());
System.out.println("PARENT: "+f1.getParent());
System.out.println("EXISTS: "+f1.exists());
System.out.println("LENGTH: "+f1.length());
System.out.println("READABLE: "+f1.canRead());
System.out.println("WRITABLE: "+f1.canWrite());
System.out.println("IS DIRECTORY: "+f1.isDirectory());
System.out.println("IS FILE: "+f1.isFile());
System.out.println("LAST MODIFIED ON: "+f1.lastModified());
}
}
```

Example_Program:

```
// To demonstrate how to work with directory files.

import java.io.*;

class File2
{
public static void main(String args[])
{
String dirname="nnn3";
File f1=new File(dirname);
if(f1.isDirectory())
{
String s[]=f1.list();
for(int i=0;i<s.length;i++)
 {
   File f2=new File(dirname+"/"+s[i]);
   if(f2.isDirectory())
     System.out.println(s[i]+" is Directory");
   else
     System.out.println(s[i]+" is File");
 }
}
else
System.out.println(dirname +" is Not a Directory");
}
}
```

Example_Program:

```
// To demonstrate how to filter directory listing.
import java.io.*;

class OnlyExt implements FilenameFilter
{
  String ext;

  public OnlyExt(String ext)
  {
   this.ext="."+ext;
  }
  public boolean accept(File dir,String fname)
  {
   return fname.endsWith(ext);
  }
}

class File3
{
public static void main(String args[])
{
  File f=new File("nnn3");
  OnlyExt e=new OnlyExt("java");
  String s[]=f.list(e);
  for(int i=0;i<s.length;i++)
     System.out.println(s[i]);
}
}
```

Character Streams

Character Streams deals with writing or reading of data to or from the I/O device as characters.

Ex. Working with text data (strings)

Base Abstract classes for Character streams are:
- Reader
- Writer

Methods of Reader class

```
int read()
int read(char buf[])
int read(char buf[], int offset, int numchars);
boolean marksupported();
void mark(int numchars)
void reset()
long skip(long numchars)
void close()
```

Methods of Writer class

```
void write(int ch)
void write(char buf[])
void write(char buf[], int offset, int numchars);
void write(String str)
void write(String str, int offset, int numchars);
Writer append(char ch);
void flush()
void close()
```

Character Streams classes:

FileReader, FileWriter
BufferedReader, BufferedWriter
PrintWriter
CharacterArrayReader , CharacterArrayWriter

FileReader

Creates a Reader that we can use to read the contents of a file.

Constructors:

FileReader(String filepath) throws FileNotFoundException

FileReader(File obj) throws FileNotFoundException

Reading from a file

```
FileReader fr = new FileReader("ex1.txt");

int c = fr.read();

while(c!=-1)
{
        System.out.println((char)c);
        c=fr.read();
}
```

Example_Program:

```
// To demonstrate FileReader.
import java.io.*;

class FileRead8
{
  public static void main(String[] args) throws IOException,FileNotFoundException
  {
    int ch;
```

```
     FileReader fr=null;

     //Attach file name to FileReader
     fr = new FileReader("ex1.txt");

      System.out.println("Contents of the file....");

    //Read characters from fr into ch,and display it on the screen,repeat this till end of
the file is reached

    while(true)
    {
       ch=fr.read();
       if(ch==-1)
           break;
      System.out.print((char)ch);
    }

    //close the file handle.
    fr.close();
}
}
```

FileWriter

Creates a writer that we can use to write to a file.

Constructors:

FileWriter(String pathname) throws IOException

FileWriter(File fileobj)

FileWriter(String pathname, boolean append)

FileWriter(File fileobj,boolean append)

Writing to a file

String s1 = “Hello kmit”;

char c[] = new char[s1.length()];

s1.getchars(0,s1.length(),c,0);

```
FileWriter fw = new FileWriter("ex1.txt");

for(i=0;i<c.length;i++)
        fw.write(c[i]);
```

Example_Program:
// To demonstrate FileWriter, Read characters from keyboard and write them onto a file.

```
import java.util.*;
import java.io.*;
class FileWrite5
{
 public static void main(String[] args) throws IOException
 {
   char ch;int ch2;
   Scanner s=new Scanner(System.in);
   //Attch file name to FileWriter
   FileWriter fw=new FileWriter("file10.txt");

   System.out.println("enter a Line of characters,# to stop");
   String data=s.nextLine();
   System.out.println("input="+data);
   for(int i=0;i<data.length();i++)
     fw.write(data.charAt(i));
   fw.close();

   //displaying the content of newly creted file
   FileReader fr=new FileReader("file10.txt");
   System.out.println("The content of the file file1.txt:");
   while(true)
    {
      ch2=fr.read();
      if(ch2==-1)
        break;
      System.out.print((char)ch2);
    }
   fr.close();
 }
}
```

BufferedReader

Improves performance by buffering the input .

Constructors:

BufferedReader(Reader reader)

BufferedReader(Reader reader, int bufsize);

Reading from file using BufferedReader

```
FileReader fr = new FileReader(“ex1.txt”);

BufferedReader br = new BufferedReader(fr);
String s;

while((s=br.readLine())!=null)
{
        System.out.println(s);
}
```

Example Program:

```
//To demonstrate BufferedReader to read a line at a time.

import java.io.*;

class FileRead9
{
public static void main(String args[]) throws IOException
{
  FileReader fr=new FileReader("ex22.txt");
  BufferedReader br=new BufferedReader(fr);
  String s;
  System.out.println("The File Content is:");
  while((s=br.readLine())!=null)
     System.out.println(s);
  br.close();
}
}
```

BufferedWriter

Is a writer that buffers the output.

Using BufferedWriter can improve performance by reducing the number of times data is actually physically written to the output device.

Constructors:

BufferedWriter(Writer writer)
BufferedWriter (Writer writer, int bufsize);

The method NewLine() helps us to write line separator onto the stream

PrintWriter

Character oriented version of PrintStream.

Constructors:

PrintWriter(Writer writer)

PrintWriter (Writer writer, boolean flushon newline);

PrintWriter(File outputfile);

PrintWriter(String outputfilename);

Example Program:
```
// To demonstrate how to write formatted data to the file.
import java.io.*;

class PrintWriterDemo2
{
public static void main(String[] args) throws IOException
{
FileWriter fw = new FileWriter("format.txt");
PrintWriter pw = new PrintWriter(fw);
double r = Math.random();
int x = 1, y = 2;
pw.format("The number %4.2f is between %d and %d%n", r, x, y);
pw.printf("%2$d %3$s %1$d%n", x, y, "is bigger than");
pw.flush();
}
}
```

Console Class

Console is a physical device with a keyboard and a display.

Makes it easy to accept input from command line, both echoed and non echoed (password).

Makes it easy to write formatted output to command line.

Reading a String

```
String name;

Console c = System.console();

name = c.readLine("Enter %s name", "employee");

c.format("Hello %s", name);
```

Reading Password

```
Console c = System.console();

char  pw[] = c.readPassword("Enter password: ");

for(i=0;i<pw.length;i++)
        System.out.println(pw[i]);
```

Method in Console class

```
String readLine()

String readLine(String fmt, Object args);

char[] readPassword()

char[] readPassword(String fmt, Object args);

Console printf(String fmt,Object args);

Console format(String fmt,Object args);
```

Example_Program:

```
// To demonstrate how to write formatted data on monitor.
import java.io.*;
class ConsoleDemo
```

```
{
public static void main(String args[]) throws IOException{
Console console = System.console();
if(console == null)
{
throw new IOException("Console not available");
}

String formula = "Formula = ";
double radius = 2.0;

console.format("%10s%12.10f * %3$4.2f * %3$4.2f%n",
formula, Math.PI, radius);

double area = Math.PI * radius * radius;

console.format("%10s%16.13f%n", "Result = ", area);
}
}
```

Example Program:

```
// To demonstrate how to read lines,password from console.
import java.io.*;

class ConsoleDemo2
{
public static void main(String args[])throws IOException
{
Console console = System.console();

String userprompt = "Enter username:";
String passprompt = "Password:";

String username = console.readLine("%18s ", userprompt);

char [] password = console.readPassword("%18s ", passprompt);

System.out.println(username);
System.out.println(password);
}
}
```

RandomAccessFile

This class is not derived from either InputStream or OutputStream.

Implemented DataInput & DataOutput interfaces.

We can position the file pointer at a required position for read/write operations.

Constructors:

```
RandomAccessFile(File fileobj,String access)
                        throws FileNotFoundException;

RandomAccessFile(String filename,String access)
                        throws FileNotFoundException;
```

Access mode

r - for read

w - for write

rw - for read and write

rws - Any change to the file data or metadata will be immediately written to the physical device.

rwd - Any change to the file data will be immediately written to the physical device.

Positioning File Pointer

```
void seek(long newpos)
                        throws IOException;
```

It moves the file pointer by number of bytes specified by the argument ‘newpos’ from the beginning of the file.

RandomAccessFile class has implemented all the standard input & output methods which we can use for read/write with Random access files.

Autoboxing

It is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent wrapper whenever an object of that type is needed.

Auto Unboxing

It is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed.
Autoboxing and Auto Unboxing in Assignment statements

Ex1 Autoboxing

```
    Integer iOb=100;
```

Ex2 Auto-Unboxing

```
    int i=iOb;
```

<u>With method parameters & Return values</u>

```
class Ex1
{
  static int meth1(Integer v)
  {  return v;   }

  public static void main(String args[])
  {
       Integer Iob = meth1(100);
       System.out.println(iOb);
  }
}
```

<u>With Expressions</u>

```
Integer iOb =100, iOb2;
 ++ iOb;
System.out.println(iOb);
iOb2 = iOb +(iOb/3);
System.out.println(iOb2);
```

<u>note:</u>

Wrapper objects are immutable, if you update the value, a new object will be created with that value.

This allows us to mix different types of numberic objects in an expression.

Example**:**

```
Integer iOb=100;
Double dOb=98.6;
dOb=dOb+iOb;
System.out.println(dOb);
```

Enables us to have objects also as a control variable in the switch statements.

Example:

```
        Integer iOb2=10;
         switch(iOb2)
        {
           case 10:
           case 20:
         }
```

With Boolean & Character values

Example 1

```
Boolean b1=true;
if(b1)
System.out.println("b1 is true");
```

Example2

```
character ch='x';
char ch2=ch;
System.out.println(ch2);
```

note:

When an object whose value is null (or not initialized) needs to be unboxed, it throws NullPointerException.

Example

```
Integer x=null;
Int y=20;
Int result= x+ y;
```

Program1:

```
class demo1
{
public static void main(String args[]){
   Integer intObject1 = 2300;          //autoboxing
        Integer intObject2 = intObject1;    //shallow copy
        if(intObject1 == intObject2)
            System.out.println("Same object !");
        else
            System.out.println("Different objects !");
}
}
```

Program2:

```
class demo2
{
public static void main(String args[]){
        Integer intObject3 = 2300;          //autoboxing
        Integer intObject4 = 2300;          //autoboxing
        //compare using ==
        if(intObject3 == intObject4)
            System.out.println("Same object !");
        else
            System.out.println("Different objects !");
}
}
```

Program3:

```
class demo3
{
public static void main(String args[]){
        Integer intObject1 =2300;    //autoboxing
    Integer intObject2 = 2300;    //autoboxing
```

```
      if(intObject1.equals(intObject2))
         System.out.println("Equal values");
       else
         System.out.println("Different values");
}
}
```

Program4:

```
class demo4
{
public static void main(String args[]){
   Integer i1 = 2300;          //autoboxing
      int i2 = 2300;          //autoboxing
      //compare using ==
      if(i1 == i2)
         System.out.println("Equal Values!");
      else
         System.out.println("Different Values");
 }
}
```

Program5:

```
class demo5
{
public static void main(String args[]){
      Integer intObject3 = 23;          //autoboxing
      Integer intObject4 = 23;          //autoboxing
      //compare using ==
      if(intObject3 == intObject4)
         System.out.println("Same Values");
      else
         System.out.println("Different Values");
}
}
```

Program6:

```
class demo6
{
public static void main(String args[]){
   Integer intObject1 = 2300;          //autoboxing
        Integer intObject2 = intObject1;     //shallow copy
        //increment the value
        intObject1++;
        //because it is immutable, you get a new object
        System.out.println(intObject1);
        if(intObject1 == intObject2)
            System.out.println("Same object");
        else
            System.out.println("Different objects");
}
}
```

Program7:

```
class demo7
{
public static void main(String args[]){
 Integer i = null;
 Integer j=0;
 int k = i+j;
 System.out.println(k);
}
}
```

Enumerations

Is a list of named constants

In C language, they are simply list of named integer constants.

In Java, Enumeration defines a class type.

Enumeration can have constructors, methods and instance variables.

<u>Example</u>

```
enum Apple
{
    Jonathan,GoldenDel,RedDel,Cortland
}
```

Jonathan, GoldenDel, RedDel, Cortland are enumeration constants.Their type is Apple type.

Implicitly declared as public, final, static members.

We can not change the access specifier of these constants.

Need not instantiate(can't instantiate), can treat like primitive type.

```
Apple ap;
ap = Apple.ReDel;
System.out.println(ap);
```

<u>**Can use in if condition** as</u>

```
  if(ap==Apple.GoldenDel)
  {
  -------------
  --------------
  }
```

<u>In switch statement</u>

```
Apple ap1;
ap1=Apple.RedDel;

switch(ap1)
{
   case Jonathan:
    case GoldenDel:
}
```

Methods

1)public static enum-type[] **values**()

Returns an array that contains a list of enumeration constants.

Example :

```
  Apple a[] = Apple.values()
  for(Apple a1 :a)
    System.out.println(a1);
```

2)public static enum-type[] **valueOf**(String str)

Returns an enum constant whose value corresponds to the string passed.

Example :

```
  Apple ap;
  ap=Apple.valueOf("RedDel");
  System.out.println(ap);
```

Note:

Each Enumeration constant is an object of its enumeration type.

The constructor will be called when each enumeration constant is created.

Each enum constant has its own copy of instance variable.

Default value '0'.

Constructors, instance var, .. etc all can be specified only after the declaration enumeration constants.

Example

```
enum Apple
{
  Jonathan(10), GoldenDel(9), RedDel(12);
  private int price;
  Apple() { price=-1; }
  Apple(int p) { price =p; }
  int getPrice() {  return price;  }
}
class Demo
  {
          public static void main(String args[])
          {
             Apple ap;
             System.out.println(Apple.GoldenDel.getPrice() );
             for(Apple a : Apple.values())
                     System.out.println(a.getPrice());
      }
  }
```

3)final int **ordinal**()

You can obtain a value (ordinal value) that indicates an enumeration constant position in the list of constants.

Ordinal value begins with zero.

4)final int **compareTo**(enum-type e)

Compares the ordinal values of two enum constants of the same enumeration type.

Returns **-ve value** if invoking constants ordinal value is less than the arguments ordinal value.

Returns **+ve** value if invoking constants ordinal value is greater than the arguments ordinal value.

Returns **zero** if ordinal values of both are same.

5)boolean **equals**(enum-type e)

Returns **true** only if both enum constants refer to the same enumeration constant (having same ordinal value is not enough).

<u>Restrictions</u>

An Enumearation can't inherit another class.

An Enumeration can't be extended.

An Enumeration can implement interfaces.

All enumerations automatically inherit java.lang.Enum