

UNIT-IV

PACKAGES AND INTERFACES

DEFINING PACKAGE:

This chapter examines two of Java's most innovative features: packages and interfaces.

Packages are containers for classes that are used to keep the class name space compartmentalized. Packages are stored in a hierarchical manner and are explicitly imported into new class definitions.

A package is a container of classes and interfaces. A package represents a directory that contains related group of classes and interfaces. For example, when we write statements like:

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

Here we are importing classes of java.io package. Here, java is a directory name and io is another sub directory within it. The '*' represents all the classes and interfaces of that io sub directory. We can create our own packages called user-defined packages or extend the available packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

USE: Packages provide reusability.

ADVANTAGES OF PACKAGES:

- Packages are useful to arrange related classes and interfaces into a group. This makes all the classes and interfaces performing the same task to put together in the same package. For example, in Java, all the classes and interfaces which perform input and output operations are stored in java.io. package.
- Packages hide the classes and interfaces in a separate sub directory, so that accidental deletion of classes and interfaces will not take place.
- The classes and interfaces of a package are isolated from the classes and interfaces of another package. This means that we can use same names for classes of two different packages. For example, there is a Date class in java.util package and also there is another Date class in java.sql package.
- A group of package called a library. The classes and interfaces of a package are like books in a library and can be reused several times. This reusability nature of packages makes programming easy. Just think, the packages in Java are created by JavaSoft people only once, and millions of programmers all over the world are daily by using them in various programs.

Different Types of Packages:

There are two types of packages in Java. They are:

- Built-in packages
- User-defined packages

Built-in packages:

These are the packages which are already available in Java language. These packages provide all most all necessary classes, interfaces and methods for the programmer to perform any task in his programs. Since, Java has an extensive library of packages, a programmer need not think about logic for doing any task. For everything, there is a method available in Java and that method can be used by the programmer without developing the logic on his own. This makes the programming easy. Here, we introduce some of the important packages of Java SE:

Java.lang: lang stands for language. This package got primary classes and interfaces essential for developing a basic Java program. It consists of wrapper classes(Integer, Character, Float etc), which are useful to convert primitive data types into objects. There are classes like String, StringBuffer, StringBuilder classes to handle strings. There is a Thread class to create various individual processes. Runtime and System classes are also present in java.lang package which contain methods to execute an application and find the total memory and free memory available in JVM.

Java.util: util stands for utility. This package contains useful classes and interfaces like Stack, LinkedList, Hashtable, Vector, Arrays, etc. These classes are collections. There are also classes for handling Date and Time operations.

Java.io: io stands for input and output. This package contains streams. A stream represents flow of data from one place to another place. Streams are useful to store data in the form of files and also to perform input-output related tasks.

Java.awt: awt stands for abstract window toolkit. This helps to develop GUI(Graphical user Interfaces) where programs with colorful screens, paintings and images etc., can be developed. It consists of an important sub package, java.awt.event, which is useful to provide action for components like push buttons, radio buttons, menus etc.

Javax.swing: this package helps to develop GUI like java.awt. The 'x' in javax represents that it is an extended package which means it is a package developed from another package by adding new features to it. In fact, javax.swing is an extended package of java.awt.

Java.net: net stands for network. Client-Server programming can be done by using this package. Classes related to obtaining authentication for network, creating sockets at client and server to establish communication between them are also available in java.net package.

Java.applet: applets are programs which come from a server into a client and get executed on the client machine on a network. Applet class of this package is useful to create and use applets.

Java.text: this package has two important classes, DateFormat to format dates and times, and NumberFormat which is useful to format numeric values.

Java.sql: sql stands structured query language. This package helps to connect to databases like Oracle or Sybase, retrieve the data from them and use it in a Java program.

Package	Primary Function
java.awt.font	Represents various types of fonts.
java.awt.geom	Allows you to work with geometric shapes.
java.awt.im	Allows input of Japanese, Chinese, and Korean characters to text editing components.
java.awt.im.spi	Supports alternative input devices.
java.awt.image	Processes images.
java.awt.image.renderable	Supports rendering-independent images.
java.awt.print	Supports general print capabilities.
java.beans	Allows you to build software components.
java.beans.beancontext	Provides an execution environment for Beans.
java.io	Inputs and outputs data.
java.lang	Provides core functionality.
java.lang.annotation	Supports annotations (metadata).
java.lang.instrument	Supports program instrumentation.
java.lang.management	Supports management of the execution environment.
java.lang.ref	Enables some interaction with the garbage collector.
java.lang.reflect	Analyzes code at run time.
java.math	Handles large integers and decimal numbers.
java.net	Supports networking.
java.nio	Top-level package for the NIO classes. Encapsulates buffers.
java.nio.channels	Encapsulates channels, which are used by the NIO system.
java.nio.channels.spi	Supports service providers for channels.
java.nio.charset	Encapsulates character sets.
java.nio.charset.spi	Supports service providers for character sets.

Package	Primary Function
java.text	Formats, searches, and manipulates text.
java.text.spi	Supports service providers for text formatting classes in java.text . (Added by Java SE 6.)
java.util	Contains common utilities.
java.util.concurrent	Supports the concurrent utilities.
java.util.concurrent.atomic	Supports atomic (that is, indivisible) operations on variables without the use of locks.
java.util.concurrent.locks	Supports synchronization locks.
java.util.jar	Creates and reads JAR files.
java.util.logging	Supports logging of information related to a program's execution.
java.util.prefs	Encapsulates information relating to user preference.
java.util.regex	Supports regular expression processing.
java.util.spi	Supports service providers for the utility classes in java.util . (Added by Java SE 6.)
java.util.zip	Reads and writes compressed and uncompressed ZIP files.

java.rmi	Provides remote method invocation.
java.rmi.activation	Activates persistent objects.
java.rmi.dgc	Manages distributed garbage collection.
java.rmi.registry	Maps names to remote object references.
java.rmi.server	Supports remote method invocation.
java.security	Handles certificates, keys, digests, signatures, and other security functions.
java.security.acl	Manages access control lists.
java.security.cert	Parses and manages certificates.
java.security.interfaces	Defines interfaces for DSA (Digital Signature Algorithm) keys.
java.security.spec	Specifies keys and algorithm parameters.
java.sql	Communicates with a SQL (Structured Query Language) database.

Package	Primary Function
java.applet	Supports construction of applets.
java.awt	Provides capabilities for graphical user interfaces.
java.awt.color	Supports color spaces and profiles.
java.awt.datatransfer	Transfers data to and from the system clipboard.
java.awt.dnd	Supports drag-and-drop operations.
java.awt.event	Handles events.

javax.swing	javax.swing.border	javax.swing.colorchooser
javax.swing.event	javax.swing.filechooser	javax.swing.plaf
javax.swing.plaf.basic	javax.swing.plaf.metal	javax.swing.plaf.multi
javax.swing.plaf.synth	javax.swing.table	javax.swing.text
javax.swing.text.html	javax.swing.text.html.parser	javax.swing.text.rtf
javax.swing.tree	javax.swing.undo	

User-Defined packages:

Just like the built in packages shown earlier, the users of the Java language can also create their own packages. They are called user-defined packages. User-defined packages can also be imported into other classes and used exactly in the same way as the Built-in packages.

CREATING AND IMPORTING PACKAGES:

```
package packagename; //to create a package
```

```
package packagename.subpackagename;//to create a sub package within a package.
```

e.g.: package pack;

- The first statement in the program must be package statement while creating a package.
- While creating a package except instance variables, declare all the members and the class itself as public then only the public members are available outside the package to other programs.

Program 1: Write a program to create a package pack with Addition class.

```
//creating a package
```

```
package pack;
```

```
public class Addition
```

```
{ private double d1,d2;
```

```
public Addition(double a,double b)
```

```
{ d1 = a;
```

```
    d2 = b;
```

```
}
```

```
public void sum()
```

```
{ System.out.println ("Sum of two given numbers is : " + (d1+d2) );
```

```
}
```

```
}
```

Compiling the above program:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Addition.java
D:\JQR>
```

The `-d` option tells the Java compiler to create a separate directory and place the `.class` file in that directory (package). The `(.)` dot after `-d` indicates that the package should be created in the current directory. So, our package `pack` with `Addition` class is ready.

Program 2: Write a program to use the `Addition` class of package `pack`.

```
//Using the package pack
```

```
import pack.Addition;
```

```
class Use
```

```
{ public static void main(String args[])
  { Addition ob1 = new Addition(10,20);
    ob1.sum();
  }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Use.java
D:\JQR>java Use
Sum of two given numbers is : 30.0
D:\JQR>
```

Program 3: Write a program to add one more class `Subtraction` to the same package `pack`.

```
//Adding one more class to package pack:
```

```
package pack;
```

```
public class Subtraction
```

```
{ private double d1,d2;
  public Subtraction(double a, double b)
  { d1 = a;
```

```

    d2 = b;
}
public void difference()
{ System.out.println ("Sum of two given numbers is : " + (d1 - d2) );
}
}
}

```

Compiling the above program:



The screenshot shows a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The current directory is "D:\JQR". The command entered is "javac -d . Subtraction.java". The prompt shows the command has been executed successfully, with the prompt returning to "D:\JQR>".

Program 4: Write a program to access all the classes in the package pack.

//To import all the classes and interfaces in a class using import pack.*;

```
import pack.*;
```

```
class Use
```

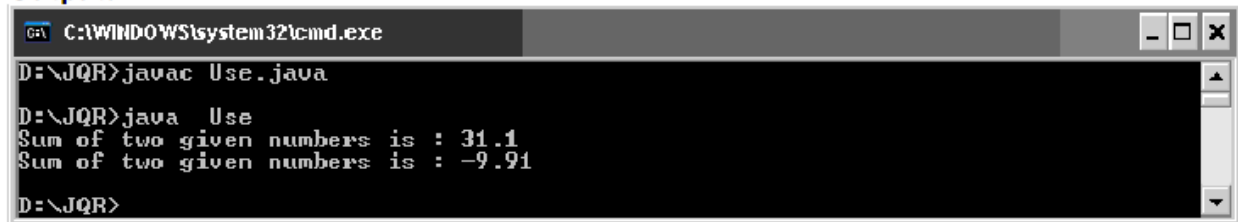
```

{ public static void main(String args[])
{ Addition ob1 = new Addition(10.5,20.6);
    ob1.sum();
    Subtraction ob2 = new Subtraction(30.2,40.11);
    ob2.difference();
}
}

```

In this case, please be sure that any of the Addition.java and Subtraction.java programs will not exist in the current directory. Delete them from the current directory as they cause confusion for the Java compiler. The compiler looks for byte code in Addition.java and Subtraction.java files and there it gets no byte code and hence it flags some errors.

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Use.java
D:\JQR>java Use
Sum of two given numbers is : 31.1
Sum of two given numbers is : -9.91
D:\JQR>
```

UNDERSTANDING CLASSPATH:

If the package pack is available in different directory, in that case the compiler should be given information regarding the package location by mentioning the directory name of the package in the classpath.

The **CLASSPATH** is an environment variable that tells the Java compiler where to look for class files to import.

If our package exists in e:\sub then we need to set class path as follows:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>set CLASSPATH=e:\sub; .;%CLASSPATH%
D:\JQR>
```

We are setting the classpath to e:\sub directory and current directory (.) and %CLASSPATH% means retain the already available classpath as it is.

Creating Sub package in a package: We can create sub package in a package in the format:

package packagename.subpackagename;

e.g.: package pack1.pack2;

Here, we are creating pack2 subpackage which is created inside pack1 package. To use the classes and interfaces of pack2, we can write import statement as:

```
import pack1.pack2;
```

Program 5: Program to show how to create a subpackage in a package.

```
//Creating a subpackage in a package
```

```
package pack1.pack2;
```

```
public class Sample
```

```
{ public void show ()
```

```
{
```

```
System.out.println ("Hello Java Learners");
```

```
}
```


}

Compiling the above program:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . Sample.java
D:\JQR>
```



CAUTION: The javac compiler always looks for files in the current directory, but the java virtual machine launcher only looks into the current directory if the "." directory is on the class path. If you have no class path set, this is not a problem—the default class path consists of the "." directory. But if you have set the class path and forgot to include the "." directory, your programs will compile without error, but they won't run.

ACCESSING A PACKAGES:

Access Specifier: Specifies the scope of the data members, class and methods.

- private members of the class are available with in the class only. The scope of private members of the class is "CLASS SCOPE".
- public members of the class are available anywhere . The scope of public members of the class is "GLOBAL SCOPE".
- default members of the class are available with in the class, outside the class and in its sub class of same package. It is not available outside the package. So the scope of default members of the class is "PACKAGE SCOPE".
- protected members of the class are available with in the class, outside the class and in its sub class of same package and also available to subclasses in different package also.

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

Program 6: Write a program to create class A with different access specifiers.

```
//create a package same
package same;

public class A
{
    private int a=1;

    public int b = 2;

    protected int c = 3;
```

```
int d = 4;
}
```

Compiling the above program:

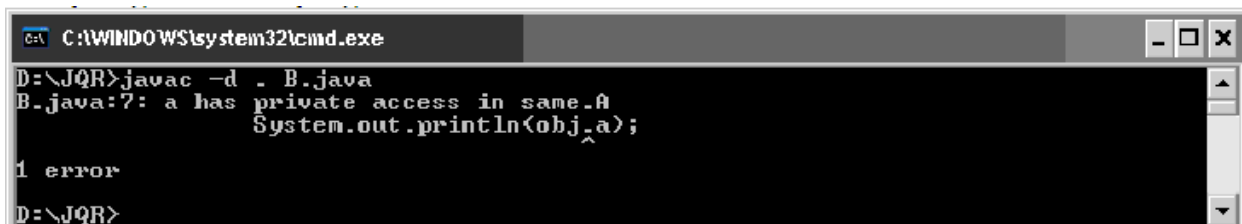


```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . A.java
D:\JQR>
```

Program 7: Write a program for creating class B in the same package.

```
//class B of same package
package same;
import same.A;
public class B
{
public static void main(String args[])
{
A obj = new A();
    System.out.println(obj.a);
    System.out.println(obj.b);
    System.out.println(obj.c);
    System.out.println(obj.d);
}
}
```

Compiling the above program:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . B.java
B.java:7: a has private access in same.A
    System.out.println(obj.a);
                        ^
1 error
D:\JQR>
```

Program 8: Write a program for creating class C of another package.

```

package another;

import same.A;

public class C extends A
{
    public static void main(String args[])
    {
        C obj = new C();

        System.out.println(obj.a);

        System.out.println(obj.b);

        System.out.println(obj.c);

        System.out.println(obj.d);

    }
}

```

Compiling the above program:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac -d . C.java
C.java:8: a has private access in same.A
      System.out.println(obj.a);
                        ^
C.java:11: d is not public in same.A; cannot be accessed from outside package
      System.out.println(obj.d);
                        ^
2 errors
D:\JQR>

```

DIFFERENCES BETWEEN CLASSES AND INTERFACES:

Classes	Interfaces
Classes have instances as variables and methods with body	Interfaces have instances as abstract methods and final constants variables.
Inheritance goes with extends keyword	Inheritance goes with implements keywords.
The variables can have any access specifier.	The Variables should be public, static, final
Multiple inheritance is not possible	It is possible
Classes are created by putting the keyword class prior to classname.	Interfaces are created by putting the keyword interface prior to interfacename(super class).
Classes contain any type of methods. Classes may or may not provide the abstractions.	Interfaces contain mostly abstract methods. Interfaces are exhibit the fully abstractions

DEFINING AN INTERFACE AND IMPLEMENTING AN INTERFACE :

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.

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;  
  
}
```

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:

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

```
}
```

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.

Partial Implementations

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.

Nested Interfaces

An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface. A nested interface can be declared as public, private, or protected. This differs from a top-level interface, which must either be declared as public or use the default access level, as previously described. When a nested interface is used outside of its enclosing scope, it must be qualified by the name of the class or interface of which it is a member. Thus, outside of the class or interface in which a nested interface is declared, its name must be fully qualified.

Here is an example that demonstrates a nested interface:

```
// A nested interface example.  
  
// This class contains a member interface.
```

```

class A {
// this is a nested interface
public interface NestedIF {
boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
public boolean isNotNegative(int x) {
return x < 0 ? false : true;
}
}
class NestedIFDemo {
public static void main(String args[]) {
// use a nested interface reference
A.NestedIF nif = new B();
if(nif.isNotNegative(10))
System.out.println("10 is not negative");
if(nif.isNotNegative(-12))
System.out.println("this won't be displayed");
}
}

```

Notice that A defines a member interface called NestedIF and that it is declared public. Next, B implements the nested interface by specifying implements

A.NestedIF

Notice that the name is fully qualified by the enclosing class' name. Inside the main() method, an A.NestedIF reference called nif is created, and it is assigned a reference to a B object. Because B implements A.NestedIF, this is legal.

Program 1: Write an example program for interface

```
interface Shape
```

```
{ void area ();
```

```
void volume ();
```

```
double pi = 3.14;
```

```
}
```

```
class Circle implements Shape
```

```
{ double r;
```

```
Circle (double radius)
```

```
{ r = radius;
```

```
}
```

```
public void area ()
```

```
{ System.out.println ("Area of a circle is : " + pi*r*r);
```

```
}
```

```
public void volume ()
```

```
{ System.out.println ("Volume of a circle is : " + 2*pi*r);
```

```
}
```

```
}
```

```
class Rectangle implements Shape
```

```
{ double l,b;
```

```
Rectangle (double length, double breadth)
```

```
{ l = length;
```

```
b = breadth;
```

```
}
```

```
public void area ()
```

```
{ System.out.println ("Area of a Rectangle is : " + l*b );
```

```
}
```

```

public void volume ()
{ System.out.println ("Volume of a Rectangle is : " + 2*(l+b));
}
}

class InterfaceDemo
{ public static void main (String args[])
{ Circle ob1 = new Circle (10.2);

ob1.area ();

ob1.volume ();

Rectangle ob2 = new Rectangle (12.6, 23.55);

ob2.area ();

ob2.volume ();

}
}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac InterfaceDemo.java
D:\JQR>java InterfaceDemo
Area of a circle is : 326.68559999999997
Volume of a circle is : 64.056
Area of a Rectangle is : 296.73
Volume of a Rectangle is : 72.3
D:\JQR>

```

APPLYING INTERFACE:

To understand the power of interfaces, let's look at a more practical example. In earlier chapters, you developed a class called Stack that implemented a simple fixed-size stack. However, there are many ways to implement a stack. For example, the stack can be of a fixed size or it can be "growable." The stack can also be held in an array, a linked list, a binary tree, and so on. No matter how the stack is implemented, the interface to the stack remains the same. That is, the methods push() and pop() define the interface to the stack independently of the details of the implementation. Because the interface to a stack is separate from its implementation, it is easy to 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.

```
// Define an integer stack interface.
```

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

Applications are:

- Abstractions
- Multiple Inheritance

VARIABLES IN INTERFACES:

You can use interfaces to import shared **constants** into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you 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 fields into the class name space as **final** variables. The next example uses this technique to implement an automated “decision maker”:

```
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());
}
}

```

Notice that this program makes use of one of Java's standard classes: Random. This class provides pseudorandom numbers. It contains several methods that 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. Here is the output of a sample run of this program. Note that the results are different each time it is run.

```

Later
Soon
No
Yes

```

EXTENDING INTERFACES:

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 one or more interfaces..
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();

}

}

```

As an experiment, you might want to try removing the implementation for meth1() in MyClass. This will cause a compile-time error. As stated earlier, any class that implements an interface must implement all methods defined by that interface, including any that are inherited from other interfaces.

Although the examples we've included in this book do not make frequent use of packages or interfaces, both of these tools are an important part of the Java programming environment.

Virtually all real programs that you write in Java will be contained within packages. A number will probably implement interfaces as well. It is important, therefore, that you be comfortable with their usage.

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:

Streams

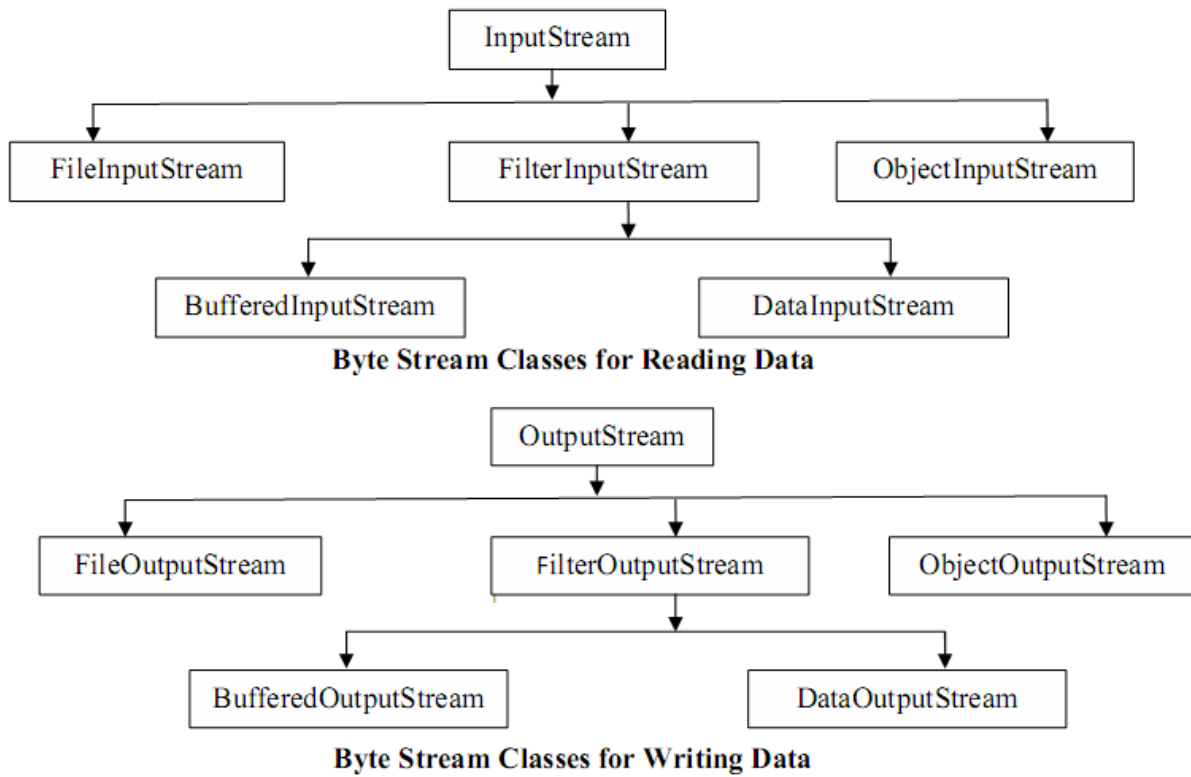
Java programs perform I/O through streams. A stream is an abstraction that either produces or consumes information. A stream is linked to a physical device by the Java I/O system.

All streams behave in the same manner, even if the actual physical devices to which they are linked differ. Thus, the same I/O classes and methods can be applied to any type of device. This means that an input stream can abstract many different kinds of input: from a disk file, a keyboard, or a network socket. Likewise, an output stream may refer to the console, a disk file, or a network connection.

Streams are a clean way to deal with input/output without having every part of your code understand the difference between a keyboard and a network, for example. Java implements streams within class hierarchies defined in the java.io package.

- **Byte Streams:** Handle data in the form of bits and bytes. Byte streams are used to handle any characters (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:



Stream Class	Meaning
BufferedInputStream	Buffered input stream
BufferedOutputStream	Buffered output stream
ByteArrayInputStream	Input stream that reads from a byte array
ByteArrayOutputStream	Output stream that writes to a byte array
DataInputStream	An input stream that contains methods for reading the Java standard data types
DataOutputStream	An output stream that contains methods for writing the Java standard data types
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that writes to a file
FilterInputStream	Implements InputStream
FilterOutputStream	Implements OutputStream
InputStream	Abstract class that describes stream input
ObjectInputStream	Input stream for objects
ObjectOutputStream	Output stream for objects
OutputStream	Abstract class that describes stream output
PipedInputStream	Input pipe
PipedOutputStream	Output pipe
PrintStream	Output stream that contains print() and println()
PushbackInputStream	Input stream that supports one-byte "unget," which returns a byte to the input stream
RandomAccessFile	Supports random access file I/O
SequenceInputStream	Input stream that is a combination of two or more input streams that will be read sequentially, one after the other

TABLE 13-1 The Byte Stream Classes

Stream Class	Meaning
BufferedReader	Buffered input character stream
BufferedWriter	Buffered output character stream
CharArrayReader	Input stream that reads from a character array
CharArrayWriter	Output stream that writes to a character array
FileReader	Input stream that reads from a file
FileWriter	Output stream that writes to a file
FilterReader	Filtered reader
FilterWriter	Filtered writer

TABLE 13-2 The Character Stream I/O Classes

Stream Class	Meaning
InputStreamReader	Input stream that translates bytes to characters
LineNumberReader	Input stream that counts lines
OutputStreamWriter	Output stream that translates characters to bytes
PipedReader	Input pipe
PipedWriter	Output pipe
PrintWriter	Output stream that contains print() and println()
PushbackReader	Input stream that allows characters to be returned to the input stream
Reader	Abstract class that describes character stream input
StringReader	Input stream that reads from a string
StringWriter	Output stream that writes to a string
Writer	Abstract class that describes character stream output

TABLE 13-2 The Character Stream I/O Classes *(continued)*

- **BufferedReader/BufferedWriter:** - Handles characters (text) by buffering them. They provide efficiency.
- **CharArrayReader/CharArrayWriter:** - Handles array of characters.
- **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.
- **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.

//Creating 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 ();

}

}

```

Output:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Createl.java
D:\JQR>java Createl
Enter @ at end :
I am writing first line
I am writing second line
@
D:\JQR>

```

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

{ //attach keyboard to DataInputStream

    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 ();
}
}

```

OUTPUT:

```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create2.java
D:\JQR>java Create2
Enter @ at end :
This is new line in my file
^
D:\JQR>type myfile
I am writing first line
I am writing second line
This is new line in my file
D:\JQR>

```

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 ();
    }
}

```

```
}  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
D:\JQR>javac Read1.java  
D:\JQR>java Read1  
I am writing first line  
I am writing second line  
This is new line in my file  
D:\JQR>
```

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 ();
```

```
}
```

```
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Read2.java
D:\JQR>java Read2
I am writing first line
I am writing second line
This is new line in my file
D:\JQR>
```

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 ();

    }
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Create3.java
D:\JQR>java Create3
D:\JQR>type textfile
This is an Institute
You are a student
D:\JQR>
```

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 ();
}
}

```

Output:



```

C:\WINDOWS\system32\cmd.exe
D:\JQR>javac Read3.java
D:\JQR>java Read3
This is an Institute
You are a student
D:\JQR>

```

Note: Use `BufferedReader` and `BufferedWriter` to improve the efficiency of the above two programs.

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;  
}  
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe  
D:\JQR>javac Employ.java  
D:\JQR>
```

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 ();  
}  
}
```

Output:

```
G:\ C:\WINDOWS\system32\cmd.exe
D:\JQR>javac StoreObj.java
D:\JQR>java StoreObj
Enter how many objects : 2
Enter employ id : 1
Enter employ name : Ravi
Enter employ salary : 10000
Enter employ id : 2
Enter employ name : Chandra
Enter employ salary : 20000
D:\JQR>
```

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 ();
```

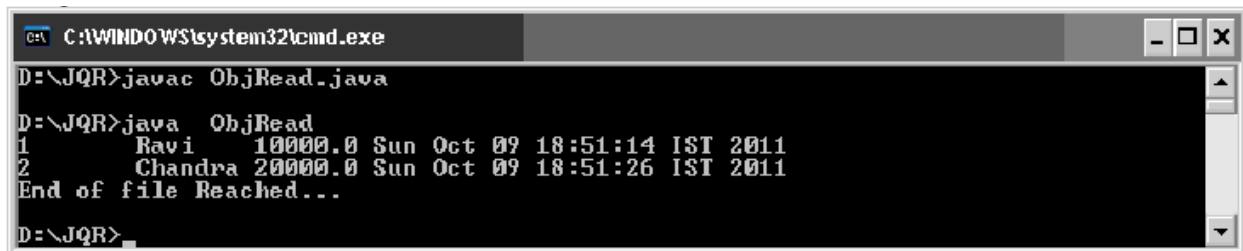
```
}
```

```
}
```



```
}
```

Output:



```
C:\WINDOWS\system32\cmd.exe
D:\JQR>javac ObjRead.java
D:\JQR>java ObjRead
1 Ravi 10000.0 Sun Oct 09 18:51:14 IST 2011
2 Chandra 20000.0 Sun Oct 09 18:51:26 IST 2011
End of file Reached...
D:\JQR>
```

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 ());  
}  
}  
}
```

Ouput:



```
C:\WINDOWS\system32\cmd.exe  
D:\JQR>javac FileProp.java  
D:\JQR>java FileProp myfile  
File name: myfile  
Path:myfile  
Absolute Path:D:\JQR\myfile  
Parent:null  
Exists:true  
Is writable: true  
Is readable: true  
Is executable: true  
Is directory: false  
File size in bytes: 80  
D:\JQR>
```