UNIT-III

Java - Files and I/O

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

- **InPutStream** − The InputStream is used to read data from a source.
- **OutPutStream** − The OutputStream is used for writing data to a destination.

Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

```
import java.io.*;

public class CopyFile {

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

    FileInputStream in = null;

    FileOutputStream out = null;

    try {

      in = new FileInputStream("input.txt");

      out = new FileOutputStream("output.txt");

      int c;

      while ((c = in.read()) != -1) {
```

```
        out.write(c);
      }
    }finally {
      if (in != null) {
        in.close();
      }
      if (out != null) {
        out.close();
      }}}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

```
import java.io.*;

public class CopyFile {

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

```
FileReader in = null;
FileWriter out = null;
try {
   in = new FileReader("input.txt");
   out = new FileWriter("output.txt");
   int c;
   while ((c = in.read()) != -1) {
      out.write(c);}
}finally {
   if (in != null) {
      in.close();}
   if (out != null) {
      out.close();
   }} }}
```

Now let's have a file **input.txt** with the following content −

```
This is test for copy file.
```

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

```
$javac CopyFile.java
$java CopyFile
```

Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. Java provides the following three standard streams −

- **Standard Input** − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as **System.in**.

- **Standard Output** − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as **System.out**.
- **Standard Error** − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as **System.err**.

Following is a simple program, which creates **InputStreamReader** to read standard input stream until the user types a "

Example

```
import java.io.*;

public class ReadConsole {
   public static void main(String args[]) throws IOException {
      InputStreamReader cin = null;
      try {
         cin = new InputStreamReader(System.in);
         System.out.println("Enter characters, 'q' to quit.");
         char c;
         do {
            c = (char) cin.read();
            System.out.print(c);
         } while(c != 'q');
      }finally {
if (cin != null) {
            cin.close();
         } } }}
```

This program continues to read and output the same character until we press 'q' −

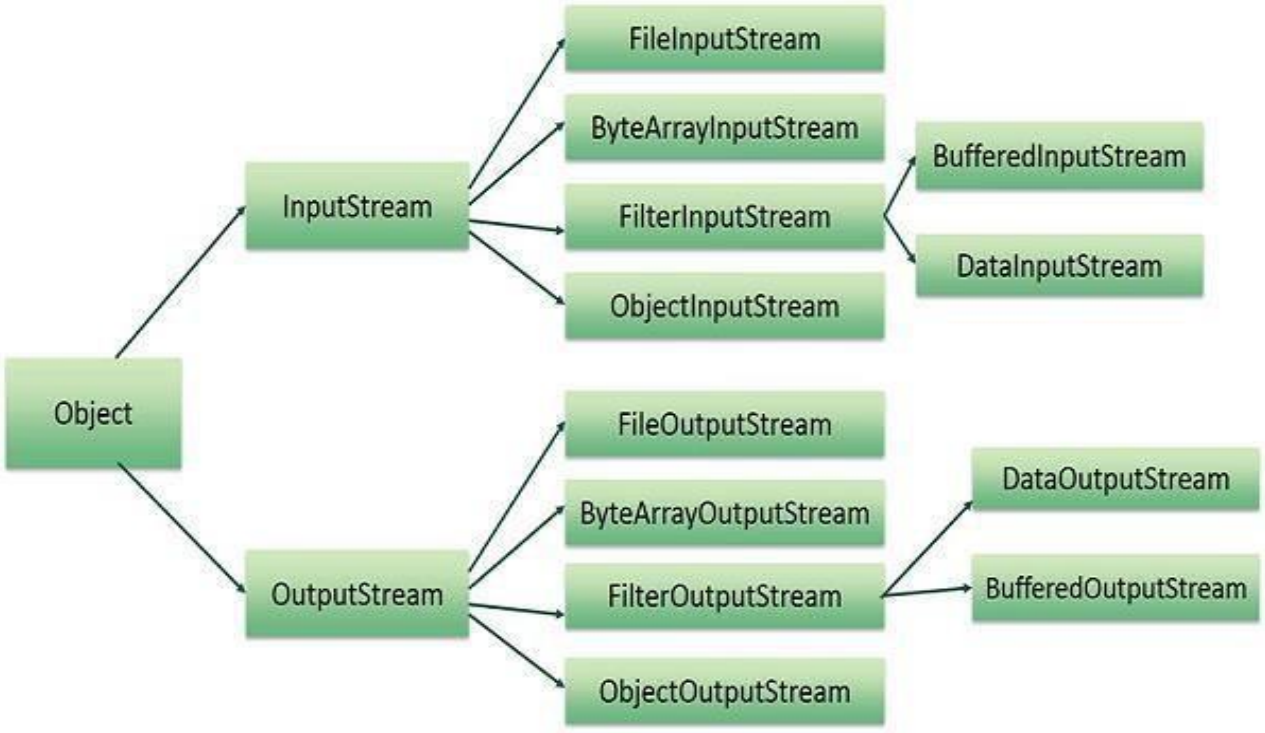
```
$javac ReadConsole.java
$java ReadConsole
```

```
Enter characters, 'q' to quit.
1
1
e
e
q
q
```

Reading and Writing Files

As described earlier, a stream can be defined as a sequence of data. The **InputStream** is used to read data from a source and the **OutputStream** is used for writing data to a destination.

Here is a hierarchy of classes to deal with Input and Output streams.



The two important streams are **FileInputStream** and **FileOutputStream**

FileInputStream

This stream is used for reading data from the files. Objects can be created using the keyword **new** and there are several types of constructors available.

Following constructor takes a file name as a string to create an input stream object to read the file −

```
InputStream f = new FileInputStream("C:/java/hello");
```

Following constructor takes a file object to create an input stream object to read the file. First we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
InputStream f = new FileInputStream(f);
```

Once you have *InputStream* object in hand, then there is a list of helper methods which can be used to read to stream or to do other operations on the stream.

- ByteArrayInputStream
- DataInputStream

FileOutputStream

FileOutputStream is used to create a file and write data into it. The stream would create a file, if it doesn't already exist, before opening it for output.

Here are two constructors which can be used to create a FileOutputStream object.

Following constructor takes a file name as a string to create an input stream object to write the file −

```
OutputStream f = new FileOutputStream("C:/java/hello")
```

Following constructor takes a file object to create an output stream object to write the file. First, we create a file object using File() method as follows −

```
File f = new File("C:/java/hello");
OutputStream f = new FileOutputStream(f);
```

Once you have *OutputStream* object in hand, then there is a list of helper methods, which can be used to write to stream or to do other operations on the stream.

- ByteArrayOutputStream
- DataOutputStream

Example

Following is the example to demonstrate InputStream and OutputStream −

```
import java.io.*;

public class fileStreamTest {

  public static void main(String args[]) {

    try {
```

```
      byte bWrite [] = {11,21,3,40,5};
      OutputStream os = new FileOutputStream("test.txt");
      for(int x = 0; x < bWrite.length ; x++) {
        os.write( bWrite[x] ); // writes the bytes}
      os.close();
      InputStream is = new FileInputStream("test.txt");
      int size = is.available();
      for(int i = 0; i < size; i++) {
        System.out.print((char)is.read() + " ");  }
      is.close();
    } catch (IOException e) {
      System.out.print("Exception");
    }      }}
```

Java.io.RandomAccessFile Class

The **Java.io.RandomAccessFile** class file behaves like a large array of bytes stored in the file system.Instances of this class support both reading and writing to a random access file.

Class declaration

Following is the declaration for **Java.io.RandomAccessFile** class −

```
public class RandomAccessFile
  extends Object
    implements DataOutput, DataInput, Closeable
```

Class constructors

S.N.	Constructor & Description
1	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, the file specified by the File argument.

2	**RandomAccessFile(File file, String mode)** This creates a random access file stream to read from, and optionally to write to, a file with the specified name.

Methodsinherited

This class inherits methods from the following classes −

- Java.io.Object

Java.io.File Class in Java

The File class is Java's representation of a file or directory path name. Because file and directory names have different formats on different platforms, a simple string is not adequate to name them. The File class contains several methods for working with the path name, deleting and renaming files, creating new directories, listing the contents of a directory, and determining several common attributes of files and directories.

- It is an abstract representation of file and directory pathnames.
- A pathname, whether abstract or in string form can be either absolute or relative. The parent of an abstract pathname may be obtained by invoking the getParent() method of this class.
- First of all, we should create the File class object by passing the filename or directory name to it. A file system may implement restrictions to certain operations on the actual file-system object, such as reading, writing, and executing. These restrictions are collectively known as access permissions.
- Instances of the File class are immutable; that is, once created, the abstract pathname represented by a File object will never change.

How to create a File Object?

A File object is created by passing in a String that represents the name of a file, or a String or another File object. For example,

```
File a = new File("/usr/local/bin/geeks");
```

defines an abstract file name for the geeks file in directory /usr/local/bin. This is an absolute abstract file name.

Program to check if a file or directory physically exist or not.

```
// In this program, we accepts a file or directory name from
// command line arguments. Then the program will check if
// that file or directory physically exist or not and
// it displays the property of that file or directory.
*import java.io.File;

// Displaying file property
class fileProperty
{
   public static void main(String[] args) {
```

```
        //accept file name or directory name through
        command line argsString fname =args[0];

        //pass the filename or directory
        name to File objectFile f = new
        File(fname);

        //apply File class methods on File
        object 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.getPare
        nt()); System.out.println("Exists
        :"+f.exists());
        if(f.exists())
        {
           System.out.println("Is
           writeable:"+f.canWrite());
           System.out.println("Is
           readable"+f.canRead());
           System.out.println("Is a
           directory:"+f.isDirectory());
           System.out.println("File Size in
           bytes "+f.length());
        }
     }
}
```

Output:

```
File name :file.txt

Path: file.txt

Absolute path:C:\Users\akki\IdeaProjects\codewriting\src\file.txt

Parent:null

Exists :true

Is writeable:true

Is readable:true

Is a directory:false

File Size in bytes 20
```