**IO Streams**

**PREREQUISITES**

The reader should have minimum knowledge of

* Basic concepts of Object Oriented programming
* Exception handling, its types and its importance
* Concept of file handling

**Points to remember**

**File**

**Purpose**

* The **java.io.File** class represents a file or a directory and defines a number of important methods for manipulating files and directories. Use the File class for typical operations such as creating, deleting, searching, and renaming files and directories; manuplating file's attributes; checking whether file exist, and so on.
* The File class does not actually provide for read/write to files. It simply provides an identifier of files and directories.

**Constructor of File Class:**

[**File**](http://download.oracle.com/javase/6/docs/api/java/io/File.html#File(java.lang.String))([String](http://download.oracle.com/javase/6/docs/api/java/lang/String.html) pathname)   
          Creates a new File instance by converting the given pathname string into an abstract

pathname.

* The File class includes several overloaded constructors: Refer to API

For example,

* Create a File object refers to a file named  myTest.txt in the current directory of the program that the JVM is running:

File f = new File("myTest.txt");

if (f.exists()) {

try {

FileReader fr = new FileReader(f);

fr.read();

...

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

}

} else { ... }

* The file myTest.txt may or may not exist in the current directory. You must use exits() method to determine whether file exists before you use File object that refers to a file for reading and writing operations, otherwise an FileNotFoundException exception will be thrown. In the above example, we checked file existing status before we read it.
* Create a File object refers to a file named  myTest.txt in the c:\testDir directory:

File f = new File("c:\\testDir", "myTest.txt");

if (f.exists()) {

;

} else {

try {

f.createNewFile();

FileWriter fr = new FileWriter(f);

....

} catch (IOException e) {

e.printStackTrace();

}

}

* The exists method returns false that means the c:\testDir\myTest.txt did not exist now but it does not have any additional information about whether the directory existed. If the directory does not exist on the specified storage, when you try to call createNewFile method it will throw an IOException exception.
* Create a File object refers to a file named  myTest.txt in the c:\testDir directory by using another way.

File dir = new File("c:\\testFile");

dir.mkdir(); //Create a directory

File f = new File(dir, "myTest.txt");

if (f.exists()) {

;

}

else {

try {

f.createNewFile();

FileWriter fr = new FileWriter(f);

...

} catch (IOException e) {

e.printStackTrace();

}

}

**Useful Methods:** File Class consists of many methods.Examples of few methods is demonstrated below. **For the complete lists of methods please refer to API**.

Examples:

* List all files in a directory

public static void main(String[] args) {

File directory = new File( "c:\\" );

String[] listing = directory.list();

for (String f: listing) {

System.out.println( f );

}

}

* Get File Information

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

File exam = new File( "myTest.txt" );

System.out.println( "Absolute Path: " + exam.getAbsolutePath() );

System.out.println( "Path: " + exam.getPath() );

System.out.println( "Name: " + exam.getName() );

System.out.println( "Parent: " + exam.getParent() );

System.out.println( "Can read: " + exam.canRead() );

System.out.println( "Can write: " + exam.canWrite() );

System.out.println( "Length: " + exam.length() );

System.out.println( "Modified?: " + exam.lastModified() );

System.out.println( "Modified!: " + new Date(exam.lastModified()) );

}

* Create a Temp file

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

// Create a temporary file to use during processing

File temp = File.createTempFile("app", ".tmp");

// Filename prefix and suffix

// Make sure file gets deleted when we're done with it temp.deleteOnExit();

}

**FileReader and FileWriter**

* The java.io package contains a collection of stream classes that support these algorithms for reading and writing. These classes are divided into two class hierarchies based on the data type (either characters or bytes) on which they operate.
* The Reader and Writer are the abstract superclasses for character streams in java.io package. The Reader provides the API and partial implementation for readers--streams that read 16-bit characters. And the Writer provides the API and partial implementation for writers--streams that write 16-bit characters.
* Remember that FileReader and FileWriter read and write 16-bit characters. However, most native file systems are based on 8-bit bytes. These streams encode the characters as they operate according to the default character-encoding scheme.

**java.io.FileReader**

* The **java.io.FileReader** class, which is subclass of java.io.Reader, is used to read character file. It is used to
* Read a single character.
* Read characters into a portion of an array.
* Read characters into an array. This method will block until some input is available, an I/O error occurs, or the end of the stream is reached.
* The constructors of FileReader class assume that the default character encoding and the default byte-buffer size are appropriate. To specify these values yourself, construct an InputStreamReader on a FileInputStream.
* FileReader is meant for reading streams of characters. For reading streams of raw bytes, consider using a FileInputStream.
* The FileReader Class Constructors:
* public FileReader(String fileName) creates a new FileReader, given the name of the file to read from. Imay throw a FileNotFoundException exception if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for readingt
* public FileReader(File file) creates a new FileReader, given the File to read from. It may throw a FileNotFoundException exception if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading
* public FileReader(FileDescriptor fd) creates a new FileReader, given the FileDescriptor to read from.
* The FileReader Reading Methods:
* public int read() reads a single character.The method returns the character read, as an integer in the range 0 to 65535 (0x00-0xffff), or -1 if the end of the stream has been reached. This method will block until a character is available, an I/O error occurs, or the end of the stream is reached. Subclasses that intend to support efficient single-character input should override this method. An IOException exception will be thrown if an I/O error occurs.
* **For the complete lists of methods please refer to API.**

**java.io.FileWriter**

* The **java.io.FileWriter** class, which is subclass of java.io.Writer, is used to write character file. It is used to
* Write a single character.
* Write a String.
* Write a portion of an array of characters.
* Write a portion of a String.
* The constructors of FileWriter class assume that the default character encoding and the default byte-buffer size are acceptable. To specify these values yourself, construct an OutputStreamWriter on a FileOutputStream.
* Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileWriter (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.
* Construct a FileWriter object from a given File object. An IOException will be thrown if the file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason:
* public FileWriter(File file) constructs a FileWriter object given a File object.
* public FileWriter(File file, boolean append) constructs a FileWriter object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
* Construct a FileWriter object from a given file name. An IOException will be thrown if the named file exists but is a directory rather than a regular file, does not exist but cannot be created, or cannot be opened for any other reason:
* public FileWritef(String filename) constructs a FileWriter object given a file name.
* public FileWriter(String filename,boolean append) constructs a FileWriter object given a file name. If the second argument is true, then bytes will be written to the end of the file rather than the beginning.
* Methods for writing content to a file:
* public void write(int c) writes a single character. The character to be written is contained in the 16 low-order bits of the given integer value; the 16 high-order bits are ignored.
* public void write(char[] cbuf) writes an array of characters.
* public void write(char[] cbuf, int off, int len) writes a portion of an array of characters.
* public void write(String str) writes a string.
* public void write(String str,int off, int len) writes a portion of a string.
* For writing a portion of characters/string, the off parameter value must in the range from 0 to length() of characters or string AND also (off + len) <= length(); otherwise a StringIndexOutOfBoundsException exception will be thrown.
* For all of the writing methods, an IOException exception will be thrown if an I/O error occurs .

**Examples**

* The following program uses FileReader and FileWriter to copy the contents of a

file named input.txt into a file called output.txt :

import java.io.\*;

public class CopyFile {

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

try {

//Create a FileReader object

File inputFile = new File("input.txt");

FileReader in = new FileReader(inputFile);

//Create a FileWriter object

File outputFile = new File("output.txt");

FileWriter out = new FileWriter(outputFile);

//Copy contents

int c;

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

out.write(c);

//Clear up

in.close();

out.close();

} catch(IOException e) {}

}

}

* This program is very simple. It opens a FileReader on input.txt and opens a FileWriter on output.txt. The program reads characters from the reader as long as there's more input in the input file. When the input runs out, the program closes both the reader and the writer.

**Summary**

* The FileReader and FileWriter provide fairly low-level mthods for reading and writing a text file. In practice, the FileReader is usually wrapped by BufferedReader and the FileWriter is usually wrapped by BufferedWriter or PrintWriter. These high-level classes provide more convenient ways to deal with data and also better performance.

**BufferedReader and BufferedWriter**

**java.io.BufferedReader**

* The java.io.BufferedReader reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
* The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.
* In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example:

BufferedReader in = new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

* Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.
* BufferedReader Constructors:
* public BufferedReader(Reader in) creates a buffering character-input stream that uses a default-sized input buffer.
* public BufferedReader(Reader in, int sz) creates a buffering character-input stream that uses an input buffer of the specified size. If sz is <= 0 then an IllegalArgumentException exception will be thrown.
* The constructor takes a Reader as the parameter, since FileReader is a Reader it is the correct type for the parameter:

BufferedReader br = new BufferedReader(new FileReader("foo.in"));

**Reading Methods**:

* public int read() throws IOException : Read a single character . Returns the character read, as an integer in the range 0 to 65535 (0x00-0xffff), or -1 if the end of the stream has been reached.
* **For the complete lists of methods please refer to API.**

**java.io.BufferedWriter**

* The java.io.BufferedWriter writes text to a character-output stream, buffering characters so as to provide for the efficient writing of single characters, arrays, and strings.
* The buffer size may be specified, or the default size may be accepted. The default is large enough for most purposes.
* In general, a Writer sends its output immediately to the underlying character or byte stream. Unless prompt output is required, it is advisable to wrap a BufferedWriter around any Writer whose write() operations may be costly, such as FileWriters and OutputStreamWriters. For example:

PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));

will buffer the PrintWriter's output to the file. Without buffering, each invocation of a print() method would cause characters to be converted into bytes that would then be written immediately to the file, which can be very inefficient.

* The BufferedWriter Class Constructors

public BufferedWriter(Writer out) creates a buffered character-output stream that uses a default-sized output buffer.

* public BufferedWriter(Writer out, int sz) - creates a new buffered character-output stream that uses an output buffer of the given size. The sz (Output-buffer size) must be a positive integer. If sz is <= 0 then an IllegalArgumentException exception will be thrown.

**Writing Methods:**

* public void write(int c) throws IOException : Writes a single character.
* **For the complete lists of methods please refer to API.**

**java.io.PrintWriter**

* Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.
* Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the println, printf, or format methods is invoked, rather than whenever a newline character happens to be output. These methods use the platform's own notion of line separator rather than the newline character.
* Methods in this class never throw I/O exceptions, although some of its constructors may. The client may inquire as to whether any errors have occurred by invoking checkError().
* The class PrintWriter is used to deal with the end-of-line problem and other frustrations of file output. PrintWriter provides many methods that are useful for "printing" characters. ("Printing" in this context means sending characters to an output destination, not usually a hard-copy printer.) The println() method uses the approprate line separator for the operating system the program is running on.
* Often a PrintWriter stream is connected to a BufferedWriter stream which is connected to a FileWriter. A further advantage of PrintWriter is that none of its methods (including println()) throw exceptions. Usually output is more reliable than input (because the program has control over its output), so exception handling is sometimes not needed.

**Examples**

* The following program uses BufferedReader and BufferedWriter to copy the contents of a file named input.txt into a file called output.txt :

import java.io.\*;

public class CopyFile {

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

try {

//Create a FileReader object

File inputFile = new File("input.txt");

BufferedReader br = new BufferedReader(new FileReader(inputFile));

//Create a FileWriter object

File outputFile = new File("output.txt");

BufferedWriter bw = new BufferedWriter(new FileWriter(outputFile));

//Copy contents

int c;

while ((c = br.read()) != -1)

bw.write(c);

//Clear up

br.close();

bw.close();

} catch(IOException e) {}

}

}

* This program is very simple. It opens a BufferedReader on input.txt and opens a BufferedWriter on output.txt. The program reads characters from the reader as long as there's more input in the input file. When the input runs out, the program closes both the reader and the writer.
* The following example code uses FileWriter and PrintWriter:

import java.io.\*;

class WriteTextFile

{

public static void main ( String[] args )

{

String fileName = "output.txt" ;

PrintWriter print = null;

try

{

print = new PrintWriter( new BufferedWriter( new FileWriter( fileName ) ) );

}

catch ( IOException e)

{

...

}

print.println( "Line1..........!" );

print.println( "Line2..........!" );

print.println( "Line3..........!" );

print.println( "Line4..........!" );

print.close();

}

}

**java.io.DataInputStream**

* The DataInputStream class is a subclass of FilterInputStream that provides methods for reading a variety of data types. The DataInputStream class implements the DataInput interface, so it defines methods for reading all of the primitive Java data types.
* A data input stream lets an application read primitive Java data types from an underlying input stream in a machine-independent way. An application uses a data output stream to write data that can later be read by a data input stream.
* DataInputStream, like other filtered input streams, must be attached to some other InputStream. You create a DataInputStream by passing a reference to an underlying InputStream to the constructor.
* public DataInputStream(InputStream in) creates a DataInputStream that uses the specified underlying InputStream.
* DataInputStream provides a high-level interface for reading data from a file. All of the reading methods are thrown an IOException if an I/O error occurs. All of the reading methods except for read from a byte[] array) are thrown an EOFException if this stream reaches the end before reading all the bytes.

Few Method Summary’s in DataInputStream Class:

* public final byte readByte() throws IOException reads and returns one input byte (a signed 8-bit byte). The byte is treated as a signed value in the range -128 through 127, inclusive. This method is suitable for reading the byte written by the writeByte method of interface DataOutput.
* **For the complete lists of methods please refer to API.**
* DataInputStream also provides a high-level interface for reading modified UTF-8 data format from a file. All of the reading methods are thrown an IOException if an I/O error occurs; If the bytes do not represent a valid modified UTF-8 encoding of a string, an UTFDataFormatException is thrown; If this input stream reaches the end before reading all the bytes, an EOFException is thrown.
* public final String readUTF() throws IOException reads in a string that has been encoded using a modified UTF-8 format. The general contract of readUTF is that it reads a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String.
* public static final String readUTF(DataInput in) throws IOException reads from the stream in a representation of a Unicode character string encoded in modified UTF-8 format; this string of characters is then returned as a String. The details of the modified UTF-8 representation are exactly the same as for the readUTF method of DataInput.
* Data must be read sequentially from a DataInputStream; you can skip forward, but you cannot move back.

**For the entire method Summary Please refer to API**

**DataOutputStream**

* The package java.io provides two set of class hierarchies - one for handling reading and writing of bytes, and another for handling reading and writing of characters. The abstract classes InputStream and OutputStream are the root of inheritance hierarchies handling reading and writing of bytes respectively.

**java.io.DataOutputStream**

* The DataOutputStream class is a subclass of the FilterOutputStream class that provides methods for writing a variety of data types to an OutputStream. The DataOutputStream class implements the DataOutput interface, so it defines methods for writing all of the primitive Java data types.
* DataOutputStream, like other filtered output streams, must be attached to some other OutputStream.You create a DataOutputStream by passing a reference to an underlying OutputStream to the constructor.
* public DataOutputStream(OutputStream out) creates a new data output stream to write data to the specified underlying output stream. The counter written is set to zero.
* The DataOutputStream class defines methods for writing primitive data types to an output stream in a machine-independent manner. Many of the methods of DataOutputStream write a single primitive data type, in binary format, to an underlying output stream. All of the writing methods throw an IOException if there is an I/O error occurs.
* public final void writeByte(int v) throws IOException writes out a byte to the underlying output stream as a 1-byte value. If no exception is thrown, the counter written is incremented by 1.
* **For the complete lists of methods please refer to API.**

**FileInputStream**

* The FileInputStream class represents a byte stream that reads data from a file in a file system. What files are available depends on the host environment. The file can be specified using a FileDescriptor, a File object, or a String that represents a pathname. All of the constructors can throw a SecurityException if the application does not have permission to read from the specified file:

Constructor Summary:

* public FileInputStream(File file) throws FileNotFoundException creates a FileInputStream by opening a connection to an actual file, the file named by the File object file in the file system. A new FileDescriptor object is created to represent this file connection.

**For the entire Constructor Summary Please Refer API.**

* FileInputStream provides a low-level interface for reading data from a file. You should wrap a FileInputStream with a DataInputStream if you need a higher-level interface that can handle reading strings and binary data. You should also think about wrapping a FileInputStream with a BufferedInputStream to increase reading efficiency.

**Method Summary**

* public int read() throws IOException reads a byte of data from this input stream. This method blocks if no input is yet available. If an I/O error occurs,an IOException will be thrown.
* public int available() throws IOException returns the number of bytes that can be read from this file input stream without blocking. Returns the number of bytes that can be read from this file input stream without blocking. An IOException is thrown if an I/O error occurs.

**For the entire Method Summary Please Refer API.**

* Data must be read sequentially from a FileInputStream; you can skip forward, but you cannot move back. If you need random access to file data, use the RandomAccessFile class instead.
* public long skip(long n) throws IOException skips over and discards n bytes of data from the input stream. Returns the actual number of bytes skipped. An IOException is thrown if n is negative, or if an I/O error occurs.
* Do not forget to close and release the resource you do not need to use.
* public void close() throws IOException closes this file input stream and releases any system resources associated with the stream. If this stream has an associated channel then the channel is closed as well. An IOException is thrown if an I/O error occurs.
* protected void finalize() throws IOException ensures that the close method of this file input stream is called when there are no more references to it. An IOException is thrown if an I/O error occurs.

FileInputStream is ment for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

**FileOutputStream**

* The FileOutputStream class represents a byte stream that writes data to a file. Whether or not a file is available or may be created depends upon the underlying platform. Some platforms, in particular, allow a file to be opened for writing by only one FileOutputStream (or other file-writing object) at a time. In such situations the constructors in this class will fail if the file involved is already open.
* The file can be specified using a FileDescriptor, a File object, or a String that represents a pathname. All of the constructors can throw a SecurityException if the application does not have permission to write to the specified file.

**Constructor Summary’s**

* public FileOutputStream(String name) throws FileNotFoundException creates an output file stream to write to the file with the specified name. A new FileDescriptor object is created to represent this file connection. First, if there is a security manager, its checkWrite method is called with name as its argument.
* public FileOutputStream(File file) throws FileNotFoundException creates a file output stream to write to the file represented by the specified File object. A new FileDescriptor object is created to represent this file connection. First, if there is a security manager, its checkWrite method is called with the path represented by the file argument as its argument.

**For the entire Constructor Summary Please Refer API.**

* FileOutputStream provides a low-level interface for writing data to a file. Wrap a FileOutputStream with a DataOutputStream or a PrintStream if you need a higher-level interface that can handle writing strings and binary data. You should also think about wrapping a FileOutputStream with a BufferedOutputStream to increase writing efficiency.
* public void write(int b) throws IOException Writes the specified byte to this file output stream. Implements the write method of OutputStream. An IOException is thrown if an I/O error occurs.
* **For the entire method Summary Please refer to API.**
* Data must be written sequentially to a FileOutputStream; you can either overwrite existing data or append data to the end of the file. If you need random access to file data, use the RandomAccessFile class instead.
* Do not forget to close and release the resource you opened:
* FileOutputStream is meant for writing streams of raw bytes such as image data. For writing streams of characters, consider using FileWriter .

**ObjectInputStream**

* An ObjectInputStream deserializes primitive data and objects previously written using an ObjectOutputStream. The objects and other data must have been written by an ObjectOutputStream. ObjectOutputStream and ObjectInputStream can provide the ability to save the state information of components on some persistent (permanent) storage medium so that it can be restored when the application is run later, when they are used in conjunction with FileInputStream and FileOutputStream. The classes can also be used with socket streams to pass objects across the network.
* The ObjectInputStream constructor requires an InputStream.The constructor calls readStreamHeader to read and verifies the header and version written by the corresponding ObjectOutputStream.writeStreamHeader method.
* public ObjectInputStream(InputStream in) throws IOException creates an ObjectInputStream that reads from the specified InputStream. A serialization stream header is read from the stream and verified. This constructor will block until the corresponding ObjectOutputStream has written and flushed the header. ObjectInputStream is used to recover those objects previously serialized by an ObjectOutputStream. Other uses include passing objects between hosts using a socket stream or for marshaling and unmarshaling arguments and parameters in a remote communication system.
* ObjectInputStream ensures that the types of all objects in the graph created from the stream match the classes present in the Java Virtual Machine. Classes are loaded as required using the standard mechanisms.
* Any objects that are instances of classes that implement the Serializable or Externalizable interfaces can be deserialized from an input stream.
* Reading an object is analogous to running the constructors of a new object. Memory is allocated for the object and initialized to zero (NULL). No-arg constructors are invoked for the non-serializable classes and then the fields of the serializable classes are restored from the stream starting with the serializable class closest to java.lang.Object and finishing with the object's most specific class. For example:

try {

// Serialize today's date to a file.

FileOutputStream f = new FileOutputStream("t.tmp");

ObjectOutputStream oos = new ObjectOutputStream(f);

oos.writeObject("Today");

oos.writeObject(new Date());

oos.flush();

oos.close();

//Deserialize today's date from a file

FileInputStream fis = new FileInputStream("t.tmp");

ObjectInputStream ois = new ObjectInputStream(fis);

int i = ois.readInt();

String today = (String) ois.readObject();

Date date = (Date) ois.readObject();

ois.close();

} catch (Exception e) {

....

}

Classes control how they are serialized by implementing either the java.io.Serializable or java.io.Externalizable interfaces.

* Implementing the Serializable interface allows object serialization to save and restore the entire state of the object and it allows classes to evolve between the time the stream is written and the time it is read. It automatically traverses references between objects, saving and restoring entire graphs.
* Classes that have transient instance variables may require special handling to reconstruct the values of these variables when objects are deserialized. Special handling may also be necessary to correctly deserialize objects that were serialized with a different version of their class than is in use when they are deserialized. Serializable classes that require special handling during the serialization and deserialization process should implement the following methods:

private void writeObject(java.io.ObjectOutputStream stream)

throws IOException;

private void readObject(java.io.ObjectInputStream stream)

throws IOException, ClassNotFoundException;

private void readObjectNoData()

throws ObjectStreamException;

* The writeObject() method is responsible for writing the state of the object for the particular class so that it can be restored by readObject(). The readObject() method registers an object validation callback by calling registerValidation() as its first action. The readObject method is responsible for reading and restoring the state of the object for its particular class using data written to the stream by the corresponding writeObject method.

**Examples**

* The following example intends to show that how the ObjectInputStream handles deserializing class whose super class is none-serializable class. You should pay attention about which constructor is called as well as transient and static variable.

**import java.io.\*;**

class NonSerializable {

private int i;

private static int j = 10;

private transient String str = "abc";

public NonSerializable()

{

System.out.println("I am in NonSerializable constructor!");

}

public void changeValues()

{

i = 100;

j = 200;

str = "def";

}

public void printOut() {

System.out.println("Class NonSerializable:");

System.out.println("i = " + i);

System.out.println("j = " + j);

System.out.println("str = " + str);

}

}

* public class ObjectInputStreamSample extends NonSerializable implements Serializable {

private int m;

private static int n= 10;

private transient String str = "hello";

private String str1;

public ObjectInputStreamSample()

{

System.out.println("I am in ObjectInputStreamSample constructor!");

}

public void changeValues(){

m = 100;

n = 200;

str = "transient";

str1 = "non-transient";

super.changeValues();

}

public void printOut() {

System.out.println("Class ObjectInputStreamSample:");

System.out.println("m = " + m);

System.out.println("n = " + n);

System.out.println("str = " + str);

System.out.println("str1 = " + str1);

super.printOut();

}

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

ObjectInputStreamSample a = new ObjectInputStreamSample();

a.changeValues();

System.out.println("Before serialization:");

a.printOut();

ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("myTest.ser"));

oos.writeObject(a);

oos.flush();

oos.close();

ObjectInputStream ois = new ObjectInputStream(new FileInputStream("myTest.ser"));

ObjectInputStreamSample b = (ObjectInputStreamSample)ois.readObject();

System.out.println("After serialization:");

b.printOut();

}

}

The output is

I am in NonSerializable constructor!

I am in ObjectInputStreamSample constructor!

Before serialization:

Class ObjectInputStreamSample:

m = 100

n = 200

str = transient

str1 = non-transient

Class NonSerializable:

i = 100

j = 200

str = def

I am in NonSerializable constructor!

After serialization:

Class ObjectInputStreamSample:

m = 100

n = 200

str = null

str1 = non-transient

Class NonSerializable:

i = 0

j = 200

str = abc

# ObjectOutputStream

* An ObjectOutputStream writes both primitive data types and graphs of Java objects to an underlying OutputStream. The objects and other data can then be read (reconstituted) using an ObjectInputStream. Persistent storage of objects can be accomplished by using a file for the stream. If the stream is a network socket stream, the objects can be reconstituted on another host or in another process.
* Only objects that support the java.io.Serializable or java.io.Externalizable interface can be serialized to an output streams.
* The method writeObject is used to write an object to the stream. Any object, including Strings and arrays, is written with writeObject. Multiple objects or primitives can be written to the stream. The objects MUST be read back from the corresponding ObjectInputStream with the SAME types and in the SAME order as they were written.
* Primitive data types can also be written to the stream using the appropriate methods from DataOutput. Strings can also be written using the writeUTF method.
* For example to write an object that can be read by the example in ObjectInputStream:

FileOutputStream fos = new FileOutputStream("test.tmp");

ObjectOutputStream oos = new ObjectOutputStream(fos);

oos.writeInt(12345);

oos.writeObject("Today");

oos.writeObject(new Date());

oos.close();

Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

private void readObject(java.io.ObjectInputStream stream)

throws IOException, ClassNotFoundException;

private void writeObject(java.io.ObjectOutputStream stream)

throws IOException

# Serializable

* Serializability of a class is enabled by the class implementing the java.io.Serializable interface. Classes that do not implement this interface will not have any of their state serialized or deserialized. All subtypes of a serializable class are themselves serializable. The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.
* To allow subtypes of non-serializable classes to be serialized, the subtype may assume responsibility for saving and restoring the state of the supertype's public, protected, and (if accessible) package fields. The subtype may assume this responsibility only if the class it extends has an accessible no-arg constructor to initialize the class's state. It is an error to declare a class Serializable if this is not the case. The error will be detected at runtime.
* During deserialization, the fields of non-serializable classes will be initialized using the public or protected no-arg constructor of the class. A no-arg constructor must be accessible to the subclass that is serializable. The fields of serializable subclasses will be restored from the stream.
* When traversing a graph, an object may be encountered that does not support the Serializable interface. In this case the NotSerializableException will be thrown and will identify the class of the non-serializable object.
* Classes that require special handling during the serialization and deserialization process must implement special methods with these exact signatures:

private void writeObject(java.io.ObjectOutputStream out)

throws IOException

private void readObject(java.io.ObjectInputStream in)

throws IOException, ClassNotFoundException;

* The writeObject method is responsible for writing the state of the object for its particular class so that the corresponding readObject method can restore it. The default mechanism for saving the Object's fields can be invoked by calling out.defaultWriteObject. The method does not need to concern itself with the state belonging to its superclasses or subclasses.
* The readObject method is responsible for reading from the stream and restoring the classes fields. It may call in.defaultReadObject to invoke the default mechanism for restoring the object's non-static and non-transient fields. The writeReplace Method
* Serializable or Externalizable classes that need to designate an alternative object to be used when writing an object to the stream should implement this special method with the exact signature:
* ANY-ACCESS-MODIFIER Object writeReplace() throws ObjectStreamException;

­

**Object Serialization**

* The key to writing an object is to represent its state in a serialized form sufficient to reconstruct the object as it is read. Thus reading and writing objects is a process called object serialization. Object serialization is essential to building all but the most transient applications. You can use object serialization in the following ways:
* Remote Method Invocation (RMI)--communication between objects via sockets
* Lightweight persistence--the archival of an object for use in a later invocation of the same program.
* We need a byte-coded representation of objects that can be stored in a file external to Java programs, so that the file can be read later and the objects can be reconstructed. Serialization provides a mechanism for saving and restoring objects.
* Serializing an object means to code it as an ordered series of bytes in such a way that it can be rebuilt (really a copy) from that byte stream. Deserialization generates a new live object graph out of the byte stream.
* The serialization mechanism needs to store enough information so that the original object can be recreated including all objects to which it refers (the object graph). When reconstructing an object from a streamWhen an object is retrieved from a stream, it is validated to ensure that it can be rebuilt as the intended object. Validation may fail if the class definition of the object has changed. Because the deserialization process will create new instances of the objects. Comparisons based on the "==" operator may no longer be valid.
* What is it saved:
* The class of the object.
* The class signature of the object.

**Serializing Objects**

* An object is serializable only if its class or its super class implements the EMPTY interface java.io.Serializable or the  Externalizable interface.
* Two stream classes in java.io, ObjectInputStream and ObjectOutputStream , allow the creation of streams for object serialization and methods that write to and read from these streams.
* Reconstructing an object from a stream requires that the object first be written to a stream. Writing objects to a stream is a straightforward process.Once you've written objects and primitive data types to a stream, you'll likely want to read them out again and reconstruct the objects. This is also straightforward. For example

class ScheduledItem implements Serializable {

private String name;

private Date date;

public ScheduledItem(String name, Date date) {

this.name = name; this.date = date;

}

public Date getDate() { return date;}

public String getName() {return name;}

public void setDate(Date date) {this.date = date;}

public void setName(String string) {name = string;}

}

public class SerializationSample {

public static void main(String[] args) {

try {

FileOutputStream out = new FileOutputStream("myCalendar.ser");

ObjectOutputStream oos = new ObjectOutputStream(out);

ScheduledItem item = new ScheduledItem("Weekly Meeting",

new Date(106,1,1,11,30));

oos.writeObject(item);

oos.flush();

FileInputStream in = new FileInputStream("myCalendar.ser");

ObjectInputStream ois = new ObjectInputStream(in);

item = (ScheduledItem)ois.readObject();

System.out.println("Name : " + item.getName());

System.out.println("Date : " + item.getDate());

} catch (FileNotFoundException e) {

e.printStackTrace();

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

* A class whose objects are to be saved must implement interface Serializable, with no methods, or the Externalizable interface, with two methods. Otherwise, runtime exception will be thrown. If we do not implements serializable interface for ScheduledItem, then we will get the exception as following:
* java.io.NotSerializableException: ScheduledItem

at java.io.ObjectOutputStream.writeObject0(Unknown Source)

at java.io.ObjectOutputStream.writeObject(Unknown Source)

at SerializationSample.main(SerializationSample.java:39)

* Normally, the serialization of instances of a class are handled by the defaultWriteObject method of ObjectOutputStream and the deserialization of any instance of the class with the defaultReadObject method in ObjectInputStream. You don't have to write any methods.
* In the above example, we used the writeObject and readObject methods. The ObjectOutputStream and ObjectInputStream also implement methods for writing and reading primitive data and Strings from the interfaces DataOutput and DataInput: such as writeChar(), readChar(), writeInt(), readInt(), writeDouble(), readDouble(), writeUTF(), readUTF() and so on.

**Notes**

* An object is serializable only if its class or its superclass implements the Serializable interface.
* An object is serializable (itself implements the Serializable interface) even if its superclass is not. However, the first superclass in the hierarchy of the serializable class, that does not implements Serializable interface, MUST have a no-arg constructor. If this is violated, readObject() will produce a java.io.InvalidClassException
* The no-arg constructor of every non-serializable superclass will run when an object is deserialized. However, the deserialized objects? Constructor does not run when it is deserialized.
* The class must be visible at the point of serialization.
* All primitive types are serializable.
* Transient fields (with transient modifier) are NOT serialized, (i.e., not saved or restored). A class that implements Serializable must mark transient fields of classes that do not support serialization (e.g., a file stream).
* Static fields (with static modifier) are not serialized.
* If member variables of a serializable object reference to a non-serializable object, the code will compile but a RuntimeException will be thrown.

**Example 7.1.1E(S) Level: 1**

Purpose

Using BufferedReader() and readLine() methods.

Code

*/\* The program reads and displays lines of text until we press enter key \*/*

1. **import** java.io.\*;
2. **public** **class** BufferStringInput {
3. *// Method for reading String inputs*
4. **public** String readString() {

*/\*\* Creating object for BufferedReader class-InputStreamReader converts bytes to char\*\*/*

1. BufferedReader buf = **new** BufferedReader(**new**

InputStreamReader(System.in));

*/\*\* buf**is a character-based stream that is linked to the console through*

*System.in\*\*/*

1. String str=null;
2. **try** {
3. str = buf.readLine();// method used to read string from keyboard
4. }**catch**(Exception e) {
5. System.out.print("Error…....");
6. }
7. **return** str;
8. }
9. **public** **static** **void** main(String args[]) {
10. BufferStringInput in = **new** BufferStringInput ();
11. *// Data accepted until press enter*
12. System.out.print("Enter data and press “Enter”: ");
13. *// readString().length() is used to get the length of the stirng*
14. System.out.println("The data you entered has

"+in.readString().length()+" characters.");

1. }
2. }

Output

C:\Documents and Settings\sv41104\Desktop>java BufferedStringInput

Enter data and press “Enter”: Satyam

The data you entered has 6 characters.

Check your understanding

1. Here, ……………………….r is the stream that is linked to the instance of BufferedReader that is being created. (see line no. 5)
2. reader is an abstract class. One of its subclasses is …………………., which converts bytes to characters.(see line no. 5)
3. buf is a character-based stream that is linked to the console through …………………. (see line no. 5)
4. which method is used hereto read a string from the keyboard? (see line no. 8)

 Check: Do you now understand

* + How to use bufferedReader() and readLine()
  + What is its purpose

**Example 7.1.2E(S) Level: 2**

Purpose

Using BufferedReader and readLine method

Code

1. *// The program reads different types of data and displays*
2. **import** java.io.\*;
3. **public** **class** BufferInput {
4. *// Method for reading String inputs.*
5. **public** String readString() {

*/\*\*Creating object for BufferedReader class*

*\*InputStreamReader converts bytes to char\*\*/*

1. BufferedReader buf = **new** BufferedReader(**new**

InputStreamReader(System.in));

*/\*\*buf**is a character-based stream that is linked to the console through*

*System.in\*\*/*

1. String str=null;
2. **try** {
3. str = buf.readLine(); *//Method used to read String*
4. } **catch**(Exception e) {
5. System.out.print("Error .....");
6. }
7. **return** str;
8. }
9. **public** **byte** readByte() {  *//Method for reading byte inputs*
10. **byte** by = 0;
11. **try** {
12. *// Convertig the string to byte type*

by = Byte.parseByte(readString());

1. } **catch**(NumberFormatException nfe) {
2. System.out.print("You inputted incorrect data! Try gain: ");
3. by = readByte();
4. }
5. **return** by;
6. }
7. **public** **int** readInt() { *//Method for reading integer inputs*
8. **int** in = 0;
9. **try** {
10. *// Converting string to integer type*

in = Integer.parseInt(readString());

1. } **catch**(NumberFormatException nfe) {
2. System.out.print("You inputted incorrect data! Try again: ");
3. in = readInt();
4. }
5. **return** in;
6. }
7. *// Method for reading single floating point number inputs*
8. **public** **float** readFloat() {
9. **float** flo = 0;
10. **try** {
11. flo = Float.parseFloat(readString()); //Converting string to

float type

1. } **catch**(NumberFormatException nfe) {
2. System.out.print("You inputted incorrect data!Try again: ");
3. flo = readFloat();
4. }
5. **return** flo;
6. }
7. **public** **static** **void** main(String args[]) {
8. BufferInput in = **new** BufferInput ();
9. System.out.print("Input a byte: ");
10. *// Displays the times 2 of inputted value*
11. System.out.println("The number you inputted times 2 equals

"+in.readByte()\*2);

1. System.out.print("Input an int: ");
2. System.out.println("The number you inputted times 2 equals

"+in.readInt()\*2);

1. System.out.print("Input a float: ");
2. System.out.println("The number you inputted times 2 equals

"+in.readFloat()\*2);

1. }
2. }

Output

C:\Documents and Settings\sv41104\Desktop>java BufferInput

Input a byte: 1

The number you inputted times 2 equals 2

Input an int: 85

The number you inputted times 2 equals 170

Input a float: 23.56

The number you inputted times 2 equals 47.12

Check your understanding

1. Here, can you tell how an object is created for BufferedReader? (see line no. 9)
2. Here, the oject is then used to read a String, converting String to byte, string to integer.

 Check: Do you now understand

* + How to use bufferedReader() and readLine()
  + What is its purpose

**Example 7.1.3E (S) Level: 1**

Purpose

Demonstrating Buffered Reader using File Reader

Code

*// the program reads the data from file and diplays*

1. **import** java.io.\*;
2. **public** **class** BufferedTextFile {
3. **public** **static** **void** main( String args[] ) throws IOException {
4. String line; *// for input*
5. String filename = "java.txt";
6. BufferedReader buf = **new** BufferedReader(**new** FileReader(

filename ) );

*/\*\*buf**is a character-based stream that is linked to the console through*

*filename\*\*/*

1. System.out.println( "Reading file....." );
2. *// reading the data in the file until line gets null*
3. **while** ( ( line = buf.readLine() ) != null ) {
4. System.out.println( line );
5. }
6. buf.close();
7. }
8. }

Output

C:\Documents and Settings\sv41104\Desktop>java BufferedTextFile

Reading from file…..

java team

surya

neel

abhi

Check your understanding

1. A file name is read by method ………………………… (see line no. 8)
2. Which object holds this file? (see line no. 8)
3. The data from this file is then read using ……………………….. (see line no. 11)

 Check: Do you now understand

* + How to use bufferedReader() using File Reader
  + What is its purpose