### I/0

An **I/O Stream** represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

**IO streams** are clean way to deal with input/output without having to every part of your code the physical.

A **stream** is a sequence of data.

**Streams** support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

A program **uses an input stream to read data from a source**, one item at a time.

A program **uses an output stream to write data to a destination**, one item at time.

**Byte Streams**

Programs **use byte streams to perform input and output of 8-bit bytes**. **All byte stream classes are descended from InputStream and OutputStream.**

**Byte stream is used to read and write byte data.**

**Character Streams**

The Java platform **stores character values using Unicode conventions**. **Character stream I/O automatically translates this internal format to and from the local character set**. In Western locales, the local character set is usually an 8-bit superset of ASCII.

**Character Stream used to read and write character data.**

For most applications, I/O with character streams is no more complicated than I/O with byte streams**. Input and output done with stream classes automatically translates to and from the local character set.** A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

**All character stream classes are descended from Reader and Writer.**

**Character streams are often "wrappers" for byte streams**. **The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.**

**Data Streams**

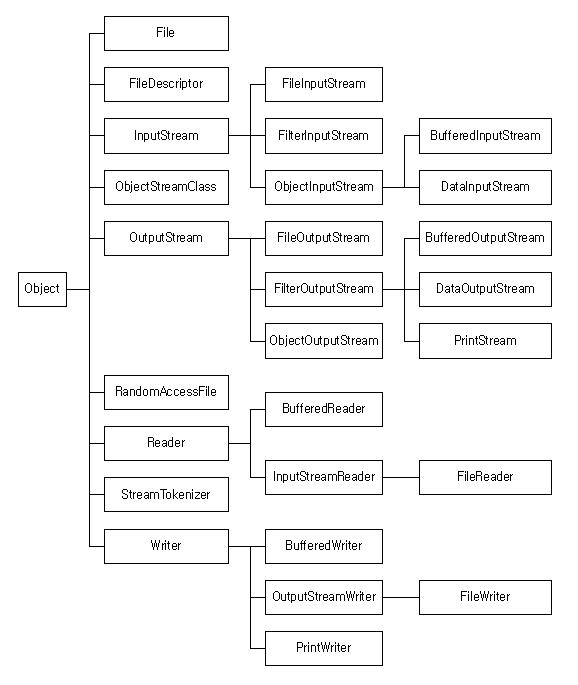
Data streams **support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values.** All data streams implement either the DataInput interface or the DataOutput interface.

**Primitive Data Stream has two interface classes DataInput and DataOutput that represents input and output stream of primitive data type.**

**Object Streams**

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

**Object Stream has two interface ObjectInput and ObjectOutput that represents input stream and output stream of object data type.**



|  |
| --- |
| **Byte stream** is used to read and write byte data.  **Character Stream** used to read and write character data.  **DataStream** supports primitive data type.  **ObjectStream** supports objects. |
| **InputStream** are **used to read data from various input devices like keyboard, file, network** etc.  **OutputStream** are **used to write the data to various output devices like monitor, file, network** etc. |
| **FileInputStream** is meant for **reading streams of raw bytes**.  **FileOutputStream** is meant for **writing streams of raw bytes**. |
| **BufferedInputStream** is used to read data from the stream. It internally uses buffer mechanism to make performance fast.  **BufferedOutputStream** writes data to a buffer and the native output API is called only when the buffer is full. |
|  |
| **Reader** is an abstract class and is the super class for all the character input stream.  **Writer** is an abstract class and is the super class for all the character output stream. |
| **FileReader** is meant for reading stream of characters.  **FileWriter** is meant for writing stream of characters. |
| **BufferedReader read data line by line and also writes byte by byte**  **BufferedWriter writes data line by line and also read byte by byte** |
|  |
| **DataInputStream** class allows an **application to read primitive data from the input stream in a machine independent way.**  **DataOutputStream** class allows an application to write primitive java data types to the OutputStream in machine independent way. |
| **ObjectOutputStream** write java objects to an output stream.  **ObjectInputStream** reads the java object from an output stream. |
|  |
| **Serialization** is the process of saving an object state to file called serialization.  **Deserialization is the** process of reading a state of an object from a file is called de-serialization.  **Externalizable** allows saving/restoring partial object i.e. 2 or 3 member variables of an object out of total objects. |

**InputStream** are **used to read data from various input devices like keyboard, file, network** etc.

**OutputStream** are **used to write the data to various output devices like monitor, file, network** etc.

Input Stream is an abstract class and it is the super class of all the input byte stream.

**FileInputStream** is meant for **reading streams of raw bytes**.

**FileOutputStream** is meant for **writing streams of raw bytes**.

int read() - It **reads byte by byte from the source file**.

The read returns int value and **is used to check end of the file as -1 is returned**.

int read(byte []b) - reads upto b.length bytes of data from this input stream into array of bytes.

FileInputStream fin = new FileInputStream("file1.txt");

FileOutputStream fout = new FileOutputStream("file2.txt");

|  |  |
| --- | --- |
| int b;  while((b=fin.read()) != -1){  fout.write(b); //storing in file  }  while((b=fin.read()) != -1){  System.out.println((char)b); // printing in console  } | byte[] buffer = new byte[fin.isAvailable()];  fin.read(buffer);  fout.write(buffer); //storing in file  for(byte bt:buffer){  System.out.print((char)bt); // printing in console  } |
| BufferedReader br = new BufferedReader(fin);  String content;  while((content=br.readLine())!=null){  System.out.print(content); //Print the content line by line  } |

|  |  |
| --- | --- |
| String strContent = "This is the text message";  byte[] content = strContent.getBytes();  fout.write(content); //storing in file | String strContent = "This is the text message";  byte[] content = strContent.getBytes();  ByteArrayInputStream bin = new ByteArrayInputStream(content);  int c;  while((c=bin.read())!=-1){  System.out.println((char)c); //Print the content  } |
| String strContent = "This is the text message";  byte[] content = strContent.getBytes();  ByteArrayOutputStream bout = new ByteArrayOutputStream();  bout.write(content);  bout.writeTo(fout); //storing in file |

**BufferedInputStream**

BufferedInputStream is used to read data from the stream. It internally uses buffer mechanism to make performance fast.

When BufferedInputStream is created, an internal buffer array is created and stored in buffer using which read operation gives good performance as the content is readily available in the buffer.

**BufferedOutputStream** writes data to a buffer and the native output API is called only when the buffer is full.

**flush() method is valid only output stream.**

Since the characters or bytes are stored in buffer before writing into file. so, **flushing the buffer is required sometimes without waiting for it to fill.**

There is an explicit flush() method for doing that. In most buffered system implementation even close() method internally calls flush() method before closing the stream.

**BufferedInputStream** and **BufferedOutputStream** are used to wrap unbuffered byte stream to create buffered byte stream.

**BufferedReader** and **BufferedWriter** are used to wrap unbuffered character stream to create buffered character stream.

**import** java.io.ByteArrayInputStream;

**import** java.io.ByteArrayOutputStream;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**public** **class** ByteStreamExample {

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

//Input is file and output is file

FileInputStream fin = **new** FileInputStream("File1.txt");

FileOutputStream fout = **new** FileOutputStream("File2.txt");

**int** b;

**while**((b=fin.read())!= -1) { // read byte by byte till end of file returns -1

fout.write(b); // Writing to file output

System.***out***.print((**char**)b); // Writing into console output

}

fout.close();

fin.close();

//Input is String ; Output is File

String str = "This is the content";

**byte** []content = str.getBytes();

FileOutputStream fouts = **new** FileOutputStream("StringFile.txt");

fouts.write(content);

fouts.close();

//Stores the string into file using ByteArrayOutputStream

String strb = "This is the content strB";

**byte** [] bcontent = strb.getBytes();

ByteArrayOutputStream bout = **new** ByteArrayOutputStream();

bout.write(bcontent);

FileOutputStream foutb = **new** FileOutputStream("CharacterFile.txt");

bout.writeTo(foutb);

foutb.close();

//Stores the string into file using ByteArrayInputStream

String strc ="This is the content for strc";

**byte** []contentc = strc.getBytes();

ByteArrayInputStream bis = **new** ByteArrayInputStream(contentc);

FileOutputStream foutc = **new** FileOutputStream("CharFile.txt");

**int** bc;

System.***out***.println("Byte through bis console output");

**while**((bc=bis.read())!=-1) {

foutc.write(bc); // Write the content to file

System.***out***.print((**char**)bc); // print the content in the console

}

foutc.close();

//read from file using byte and write into file.

FileInputStream fis = **new** FileInputStream("File1.txt");

**byte** []buffer = **new** **byte**[fis.available()];

fis.read(buffer);

FileOutputStream fos = **new** FileOutputStream("File3.txt");

fos.write(buffer); //Writing byte data into file

System.***out***.println("\n Byte to console output");

**for**(**byte** by:buffer) {

System.***out***.print((**char**)by);

}

fos.close();

fis.close();

}

}

**Reader** is an abstract class and is the super class for all the character input stream.

**Writer** is an abstract class and is the super class for all the character output stream.

**FileReader** is meant for reading stream of characters.

**FileWriter** is meant for writing stream of characters.

int read() - It is used to return a character in ASCII form. It returns -1 at the end of file.

**File Reader reads the data from file.**

**Buffered Reader reads the file line by line.**

|  |  |
| --- | --- |
| FileReader fr = new FileReader("file1.txt");  FileWriter fw = new FileWriter("file2.txt");  int c;  while((c=fr.read())!=-1){  fw.write(c); //storing in file  }  while((c=fr.read())!=-1){  System.out.print((char)c); // printing in console  } | BufferedReader br = new BufferedReader(new FileReader("file1.txt"));  String contentLine;  while((contentline=br.readLine())!= null){  System.out.print(contentLine);  }  int num=0;  char ch;  while((num=br.read())!=-1){  ch = (char)num;  System.out.print(ch);  } |
| String content ="This is test content";  fw.write(content); | String content ="This is test content";  BufferedWriter bw = new BufferedWriter(new FileWriter("file1.txt"));  bw.write(content); |

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**public** **class** CharacterStreamExample {

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

//Read from file and Write to file

FileReader fr = **new** FileReader("File1.txt");

FileWriter fw = **new** FileWriter("File2.txt");

**int** c;

**while**((c=fr.read())!=-1) {

fw.write(c);

System.***out***.print((**char**)c);

}

fw.close();

fr.close();

//Read from String and write to file

String str ="This is the content from writer";

FileWriter fwa = **new** FileWriter("File3.txt");

fwa.write(str);

fwa.close();

// read from file using Buffered Reader and write to file

BufferedReader br = **new** BufferedReader(**new** FileReader("File3.txt"));

FileWriter fwb = **new** FileWriter("File4.txt");

**int** cbr;

**while**((cbr=br.read())!=-1) {

fwb.write(cbr);

System.***out***.print((**char**)cbr);

}

br.close();

fwb.close();

//read from file using Buffered Reader and

//write to file using buffered Reader - readLine

BufferedReader bra = **new** BufferedReader(**new** FileReader("File1.txt"));

FileWriter fwc = **new** FileWriter("File5.txt");

String contentLine = bra.readLine();

**while**(contentLine!=**null**) {

fwc.write(contentLine);

contentLine=bra.readLine();

}

bra.close();

fwc.close();

}

}

**DataInputStream** class allows an **application to read primitive data from the input stream in a machine independent way.**

readInt(), readLong(), readUTF()

**DataOutputStream** class allows an **application to write primitive java data types to the OutputStream in machine independent way.**

writeInt(), writerLong(), writeUTF()

Java application generally uses the DataOutputStream to write data that can be later be read by a DataInputStream.

**flush() is used to flushes the data output stream.**

**ObjectOutputStream**

ObjectOutputStream **writes java objects to an output stream.**

ObjectOutputStream oos = new ObjectOutputStream();

Person person = new Person();

oos.write(person);

**ObjectInputStream**

ObjectInputStream **reads the java object from an output stream.**

Person person = (Person)ois.read();

**import** java.io.DataInputStream;

**import** java.io.DataOutputStream;

**import** java.io.FileInputStream;

**import** java.io.FileOutputStream;

**public** **class** DataInputStreamExample {

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

DataOutputStream dos = **new** DataOutputStream(**new** FileOutputStream("Data.out1"));

dos.writeDouble(1.1);

dos.writeBoolean(**true**);

dos.writeInt(1000);

dos.writeChar('G');

//dos.writeChars("Welcome");

dos.writeUTF("guru");

dos.writeByte('g');

dos.writeBytes("gurunathaan");

DataInputStream dis = **new** DataInputStream(**new** FileInputStream("Data.out1"));

System.***out***.println("dis.readDouble"+dis.readDouble());

System.***out***.println("dis.readBoolean"+dis.readBoolean());

System.***out***.println("dis.readInt"+dis.readInt());

System.***out***.println("dis.readChar"+dis.readChar());

//System.out.println("dis.readLine"+dis.readChar());

System.***out***.println("dis.readUTF"+dis.readUTF());

System.***out***.println("dis.readByte"+dis.readByte());

System.***out***.println("dis.readByte"+dis.~~readLine~~());

}

}

**Output :**

dis.readDouble1.1

dis.readBooleantrue

dis.readInt1000

dis.readCharG

dis.readUTFguru

dis.readByte103

dis.readBytegurunathaan

**Serialization**

Serialization **is the process of saving an object state to file** called serialization.

But practically, it is **the process of converting and storing the object's state from heap memory** (in byte stream) **to file supported form** (in binary format).

* **An object said to be serializable only if corresponding class implements java.io. serializable interface.**
* using ObjectOutputStream, FileOutputStream
* oos.writeObject(customer);

**Deserialization**

Deserialization is **the process of reading state of an object from a file** is called de-serialization.

But practically, it is **the process of converting and restoring an object's state into heap memory** **from file supported form** (which is in binary format).

using ObjectInputStream, FileInputStream

oos.readObject();

**Transient Keyword**

Transient is **applicable only for variable.**

we can **stop persisting specific variable by declaring transient keyword**.

During serialization, **JVM ignores the original value of transient variable and saves default value to file**.

**Whenever we encounter transient keyword, it means that not to serialize**.

**static variable in Serialization**

static variable **does not participate in serialization process**.

Reason: **Static variable is not part of objects state**.

**Final variable in Serialization**

Non-final, non-transient variables will be serialized(directly as values).

**Final, transient variables won't be serialized.**

**Transient variable will be assigned to default value.**

**import** java.io.Serializable;

**public** **class** Person **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 6249250551300108951L;

**int** id;

**static** String *name*;

**transient** String deptId;

**final** String address;

// ID1

// Namenull

// Deptnull

// AddressTKM

**public** Person(**int** id, String name, String deptId, String address) {

**super**();

**this**.id = id;

**this**.*name* = name;

**this**.deptId = deptId;

**this**.address = address;

}

}

**import** java.io.FileOutputStream;

**import** java.io.ObjectOutputStream;

**public** **class** PersonSerialization {

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

Person p = **new** Person(1,"guru","IT","TKM");

FileOutputStream fos = **new** FileOutputStream("serial.text");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(p);

oos.flush();

oos.close();

System.***out***.println("Person Object is save to serial.text");

}

}

**import** java.io.FileInputStream;

**import** java.io.ObjectInputStream;

**public** **class** PersonDeSerialization {

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

FileInputStream fis = **new** FileInputStream("serial.text");

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

Person p = (Person) ois.readObject();

System.***out***.println("ID"+p.id); //1

System.***out***.println("Name"+p.*name*); //null

System.***out***.println("Dept"+p.deptId); //null

System.***out***.println("Address"+p.address); //TKM

}

}

**Externalizable (java.io.Externalizable)**

Externalizable **allows saving/restoring partial object i.e. 2 or 3 member variables of an object out of total objects**.

**Externalizable is best suitable,** **when partial object or few member variables of an object needs to be serialized to file** **storage** otherwise still serializable interface is good option for saving the total object.

Externalizable interface is sub interface of serializable interface.

**writeExternal(ObjectOutput out)** method, **programmer has to explicitly write logic for saving only those required variables to file storage.**

out.writeInt(id);

out.writeObject(name);

**readExternal(ObjectInput in)** method, **programmer has to explicitly write logic for restoring object from file storage.**

int tempId = in.readInt();

String tempname = in.readObject();

**public no argument constructor is very must in externalizable otherwise InvalidCastException is thrown**.

**Transient modifier is not required in Externalizable**.

**import** java.io.Externalizable;

**import** java.io.IOException;

**import** java.io.ObjectInput;

**import** java.io.ObjectOutput;

**public** **class** Person **implements** Externalizable{

**int** id;

String name;

String deptId;

String address;

**public** Person() {

}

**public** Person(**int** id, String name, String deptId, String address) {

**super**();

**this**.id = id;

**this**.name = name;

**this**.deptId = deptId;

**this**.address = address;

}

**public** **void** writeExternal(ObjectOutput out) **throws** IOException {

out.writeInt(id);

out.writeObject(name);

}

**public** **void** readExternal(ObjectInput in) **throws** IOException, ClassNotFoundException {

**int** tempId = in.readInt();

String tempName = (String)in.readObject();

id = tempId;

name = tempName;

}

}

**import** java.io.FileOutputStream;

**import** java.io.ObjectOutputStream;

**public** **class** PersonSerialization {

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

Person p = **new** Person(1,"guru","IT","TKM");

FileOutputStream fos = **new** FileOutputStream("external.text");

ObjectOutputStream oos = **new** ObjectOutputStream(fos);

oos.writeObject(p);

oos.flush();

oos.close();

System.***out***.println("Person Object is save to serial.text");

}

}

**import** java.io.FileInputStream;

**import** java.io.ObjectInputStream;

**public** **class** PersonDeSerialization {

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

FileInputStream fis = **new** FileInputStream("external.text");

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

Person p = (Person) ois.readObject();

System.***out***.println("ID"+p.id);//1

System.***out***.println("Name"+p.name); //guru

System.***out***.println("Dept"+p.deptId);//null

System.***out***.println("Address"+p.address);//null

}

}

**Serialization with Aggregation**

Rule 1: **All classes that needs to be serialized** must implement java.io. serializable interface.

Rule 2: All reference classes inside a serializable class must also be serializable.

Rule 3: If any of the classes is not implementing java.io. serializable in the serialization process, then **the JVM throws** **NotSerializableException**.

class Customer implements Serializable {

String name;

Address address;

Phone phone;

}

class Address implements Serializable {}

class Phone implements Serializable {}

**Serialization with inheritance**

- **If superclass implements Serializable interface**, then **all sub-classes is also serializable by default**.

- While serializing sub-class whose super class does not implement serializable interface, then during serialization process inheriting instance variable of **non-serializable super class will be stored to default value ignoring their original value.**