IO Stream

Java performs I/O through **Streams**. A Stream is linked to a physical layer by java I/O system to make input and output operation in java. In general, a stream means continuous flow of data. Streams are clean way to deal with input/output without having every part of your code understand the physical.

Java encapsulates Stream under **java.io** package. Java defines two types of streams. They are,

1. **Byte Stream :** It provides a convenient means for handling input and output of byte.
2. **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode and therefore can be internationalized.

Byte Stream Classes

Byte stream is defined by using two abstract class at the top of hierarchy, they are InputStream and OutputStream.

These two abstract classes have several concrete classes that handle various devices such as disk files, network connection etc.

Some important Byte stream classes.

Stream class	Description
BufferedInputStream	Used for Buffered Input Stream.
BufferedOutputStream	Used for Buffered Output Stream.
DataInputStream	Contains method for reading java standard datatype
DataOutputStream	An output stream that contain method for writing java standard data type
FileInputStream	Input stream that reads from a file
FileOutputStream	Output stream that write to a file.
InputStream	Abstract class that describe stream input.
OutputStream	Abstract class that describe stream output.
PrintStream	Output Stream that contain `print()` and `println()` method

These classes define several key methods. Two most important are

1. `read()` : reads byte of data.
2. `write()` : Writes byte of data.

Character Stream Classes

Character stream is also defined by using two abstract class at the top of hierarchy, they are Reader and Writer.

These two abstract classes have several concrete classes that handle unicode character.

Some important Charcter stream classes.

Stream class	Description
BufferedReader	Handles buffered input stream.
BufferedWriter	Handles buffered output stream.
FileReader	Input stream that reads from file.
FileWriter	Output stream that writes to file.
InputStreamReader	Input stream that translate byte to character
OutputStreamReader	Output stream that translate character to byte.
PrintWriter	Output Stream that contain `print()` and `println()` method.
Reader	Abstract class that define character stream input
Writer	Abstract class that define character stream output

Reading Console Input

We use the object of BufferedReader class to take inputs from the keyboard.

Reading Characters

`read()` method is used with BufferedReader object to read characters. As this function returns integer type value has we need to use typecasting to convert it into **char** type.

```
int read() throws IOException
```

Below is a simple example explaining character input.

```
class CharRead
{
 public static void main( String args[])
 {
  BufferedReader br = new Bufferedreader(new InputstreamReader(System.in));
  char c = (char)br.read();        //Reading character
 }
}
```

Reading Strings

To read string we have to use `readLine()` function with BufferedReader class's object.

```
String readLine() throws IOException
```

Program to take String input from Keyboard in Java

```
import java.io.*;
class MyInput
{
 public static void main(String[] args)
```

```
 {
  String text;
  InputStreamReader isr = new InputStreamReader(System.in);
  BufferedReader br = new BufferedReader(isr);
  text = br.readLine();          //Reading String
  System.out.println(text);
 }
}
```

Program to read from a file using BufferedReader class

```
import java. Io *;
class ReadTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   BufferedReader br = new BufferedReader(new FileReader(fl)) ;
   String str;
   while ((str=br.readLine())!=null)
   {
    System.out.println(str);
   }
   br.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
}
```

Program to write to a File using FileWriter class

```
import java. Io *;
class WriteTest
{
 public static void main(String[] args)
 {
  try
  {
   File fl = new File("d:/myfile.txt");
   String str="Write this string to my file";
   FileWriter fw = new FileWriter(fl) ;
   fw.write(str);
   fw.close();
   fl.close();
  }
  catch (IOException  e)
  { e.printStackTrace(); }
 }
}
```

Java BufferedOutputStream and BufferedInputStream

Java BufferedOutputStream class

Java BufferedOutputStream class uses an internal buffer to store data. It adds more efficiency than to write data directly into a stream. So, it makes the performance fast.

Example of BufferedOutputStream class:

In this example, we are writing the textual information in the BufferedOutputStream object which is connected to the FileOutputStream object. The flush() flushes the data of one stream and send it into another. It is required if you have connected the one stream with another.

```
1.  import java.io.*;
2.  class Test{
3.   public static void main(String args[])throws Exception{
4.     FileOutputStream fout=new FileOutputStream("f1.txt");
5.     BufferedOutputStream bout=new BufferedOutputStream(fout);
6.     String s="Sachin is my favourite player";
7.     byte b[]=s.getBytes();
8.     bout.write(b);
9.
10.   bout.flush();
11.   bout.close();
12.   fout.close();
13.   System.out.println("success");
14. }
15.}
```

Output:

```
success...
```

Java BufferedInputStream class

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

Example of Java BufferedInputStream

Let's see the simple example to read data of file using BufferedInputStream.

```
1.  import java.io.*;
2.  class SimpleRead{
3.   public static void main(String args[]){
4.    try{
5.      FileInputStream fin=new FileInputStream("f1.txt");
6.      BufferedInputStream bin=new BufferedInputStream(fin);
7.      int i;
8.      while((i=bin.read())!=-1){
9.       System.out.println((char)i);
10.    }
11.    bin.close();
12.    fin.close();
13.  }catch(Exception e){system.out.println(e);}
14. }
15.}
```

Output:

```
Sachin is my favourite player
```

java.io.PrintStream class

The PrintStream class provides methods to write data to another stream. The PrintStream class automatically flushes the data so there is no need to call flush() method. Moreover, its methods don't throw IOException.

Commonly used methods of PrintStream class

There are many methods in PrintStream class. Let's see commonly used methods of PrintStream class:

- **public void print(boolean b):** it prints the specified boolean value.
- **public void print(char c):** it prints the specified char value.
- **public void print(char[] c):** it prints the specified character array values.
- **public void print(int i):** it prints the specified int value.
- **public void print(long l):** it prints the specified long value.
- **public void print(float f):** it prints the specified float value.
- **public void print(double d):** it prints the specified double value.
- **public void print(String s):** it prints the specified string value.
- **public void print(Object obj):** it prints the specified object value.
- **public void println(boolean b):** it prints the specified boolean value and terminates the line.
- **public void println(char c):** it prints the specified char value and terminates the line.
- **public void println(char[] c):** it prints the specified character array values and terminates the line.
- **public void println(int i):** it prints the specified int value and terminates the line.
- **public void println(long l):** it prints the specified long value and terminates the line.
- **public void println(float f):** it prints the specified float value and terminates the line.
- **public void println(double d):** it prints the specified double value and terminates the line.
- **public void println(String s):** it prints the specified string value and terminates the line./li>
- **public void println(Object obj):** it prints the specified object value and terminates the line.
- **public void println():** it terminates the line only.
- **public void printf(Object format, Object... args):** it writes the formatted string to the current stream.
- **public void printf(Locale l, Object format, Object... args):** it writes the formatted string to the current stream.
- **public void format(Object format, Object... args):** it writes the formatted string to the current stream using specified format.
- **public void format(Locale l, Object format, Object... args):** it writes the formatted string to the current stream using specified format.

Example of java.io.PrintStream class

In this example, we are simply printing integer and string values.

```
1. import java.io.*;
2. class PrintStreamTest{
3.  public static void main(String args[])throws Exception{
4.    FileOutputStream fout=new FileOutputStream("mfile.txt");
5.    PrintStream pout=new PrintStream(fout);
6.    pout.println(1900);
7.    pout.println("Hello Java");
8.    pout.println("Welcome to Java");
9.    pout.close();
10.   fout.close();
11. }
12.}
```

Example of printf() method of java.io.PrintStream class:

Let's see the simple example of printing integer value by format specifier.

```
1. class PrintStreamTest{
2.  public static void main(String args[]){
3.    int a=10;
4.    System.out.printf("%d",a);//Note, out is the object of PrintStream class
5.
6.  }
7. }
```

```
Output:10
```

Random Access Files

Using a random access file, we can read from a file as well as write to the file.

Reading and writing using the file input and output streams are a sequential process.

Using a random access file, we can read or write at any position within the file.

An object of the RandomAccessFile class can do the random file access. We can read/write bytes and all primitive types values to a file.

RandomAccessFile can work with strings using its readUTF() and writeUTF() methods.

The RandomAccessFile class is not in the class hierarchy of the InputStream and OutputStream classes.

Mode

A random access file can be created in four different access modes. The access mode value is a string. They are listed as follows:

Mode	Meaning
"r"	The file is opened in a read-only mode.
"rw"	The file is opened in a read-write mode. The file is created if it does not exist.
"rws"	The file is opened in a read-write mode. Any modifications to the file's content and its metadata are written to the storage device immediately.
"rwd"	The file is opened in a read-write mode. Any modifications to the file's content are written to the storage device immediately.

Read and Write

We create an instance of the RandomAccessFile class by specifying the file name and the access mode.

```
RandomAccessFile  raf = new RandomAccessFile("randomtest.txt", "rw");
```

A random access file has a file pointer that moves forward when we read data from it or write data to it.

The file pointer is a cursor where our next read or write will start.

Its value indicates the distance of the cursor from the beginning of the file in bytes.

We can get the value of file pointer by using its **getFilePointer**() method.

When we create an object of the RandomAccessFile class, the file pointer is set to zero.

We can set the file pointer at a specific location in the file using the seek() method.

The length() method of a RandomAccessFile returns the current length of the file. We can extend or truncate a file by using its **setLength**() method.

Example

The following code shows how to read and write Files Using a RandomAccessFile Object.

```
import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
// w ww  .j av a  2s.  c  om
public class Main {
  public static void main(String[] args) throws IOException {
    String fileName = "randomaccessfile.txt";
    File fileObject = new File(fileName);

    if (!fileObject.exists()) {
      initialWrite(fileName);
    }
    readFile(fileName);
    readFile(fileName);
  }

  public static void readFile(String fileName) throws IOException {
    RandomAccessFile raf = new RandomAccessFile(fileName, "rw");
    int counter = raf.readInt();
    String msg = raf.readUTF();

    System.out.println(counter);
    System.out.println(msg);
    incrementReadCounter(raf);
    raf.close();
  }

  public static void incrementReadCounter(RandomAccessFile raf)
      throws IOException {
    long currentPosition = raf.getFilePointer();
    raf.seek(0);
    int counter = raf.readInt();
    counter++;
    raf.seek(0);
    raf.writeInt(counter);
    raf.seek(currentPosition);
  }

  public static void initialWrite(String fileName) throws IOException {
    RandomAccessFile raf = new RandomAccessFile(fileName, "rw");
    raf.writeInt(0);
    raf.writeUTF("Hello world!");
    raf.close();
  }
}
```

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.

Class methods

S.N.	Method & Description
1	void close() This method Closes this random access file stream and releases any system resources associated with the stream.
2	FileChannel getChannel() This method returns the unique FileChannel object associated with this file.
3	FileDescriptor getFD() This method returns the opaque file descriptor object associated with this stream.
4	long getFilePointer() This method returns the current offset in this file.
5	long length() This method returns the length of this file.
6	int read() This method reads a byte of data from this file.
7	int read(byte[] b) This method reads up to *b.length* bytes of data from this file into an array of bytes.

8 int read(byte[] b, int off, int len)
This method reads up to *len* bytes of data from this file into an array of bytes.

9 boolean readBoolean()
This method reads a boolean from this file.

10 byte readByte()
This method reads a signed eight-bit value from this file.

11 char readChar()
This method reads a character from this file.

12 double readDouble()
This method reads a double from this file.

13 float readFloat()
This method reads a float from this file.

14 void readFully(byte[] b)
This method reads b.length bytes from this file into the byte array, starting at the current file pointer.

15 void readFully(byte[] b, int off, int len)
This method reads exactly len bytes from this file into the byte array, starting at the current file pointer.

16 int readInt()
This method reads a signed 32-bit integer from this file.

17 String readLine()
This method reads the next line of text from this file.

18 long readLong()
This method reads a signed 64-bit integer from this file.

19 short readShort()
This method reads a signed 16-bit number from this file.

20 int readUnsignedByte()
This method reads an unsigned eight-bit number from this file.

21 int readUnsignedShort()
This method reads an unsigned 16-bit number from this file.

22 String readUTF()
This method reads in a string from this file.

23 void seek(long pos)
This method sets the file-pointer offset, measured from the beginning of this file, at which the next read or write occurs.

24 void setLength(long newLength)
This method sets the length of this file.

25 int skipBytes(int n)
This method attempts to skip over n bytes of input discarding the skipped bytes.

26 void write(byte[] b)
This method writes b.length bytes from the specified byte array to this file, starting at the current file pointer.

27 void write(byte[] b, int off, int len)
This method writes len bytes from the specified byte array starting at offset off to this file.

28 void write(int b)
This method writes the specified byte to this file.

29 void writeBoolean(boolean v)
This method writes a boolean to the file as a one-byte value.

30 void writeByte(int v)
This method writes a byte to the file as a one-byte value.

31 void writeBytes(String s)
This method writes the string to the file as a sequence of bytes.

32 void writeChar(int v)
This method writes a char to the file as a two-byte value, high byte first.

33 void writeChars(String s)
This method writes a string to the file as a sequence of characters.

34 void writeDouble(double v)
This method converts the double argument to a long using the doubleToLongBits method in class Double, and then writes that long value to the file as an eight-byte quantity, high byte first.

35 void writeFloat(float v)
This method converts the float argument to an int using the floatToIntBits method in class Float, and then writes that int value to the file as a four-byte quantity, high byte first.

36 void writeInt(int v)
This method writes an int to the file as four bytes, high byte first.

37 void writeLong(long v)
This method writes a long to the file as eight bytes, high byte first.

38 void writeShort(int v)
This method writes a short to the file as two bytes, high byte first.

39 void writeUTF(String str)
This method writes a string to the file using modified UTF-8 encoding in a machine-independent manner.

Random Access File

```
import java.io.IOException;
import java.io.RandomAccessFile;

public class RandomAccessFileExample {

   public static void main(String[] args) {
      try {
         String filePath = "/Users/pankaj/source.txt";
         System.out.println(new String(readCharsFromFile(filePath, 1, 5)));

         writeData(filePath, "Data", 5);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }

   private static void writeData(String filePath, String data, int seek) throws IOException {
      RandomAccessFile file = new RandomAccessFile(filePath, "rw");
      file.seek(seek);
      file.write(data.getBytes());
      file.close();
   }

   private static byte[] readCharsFromFile(String filePath, int seek, int chars) throws
IOException {
      RandomAccessFile file = new RandomAccessFile(filePath, "r");
      file.seek(seek);
      byte[] bytes = new byte[chars];
      file.read(bytes);
      file.close();
      return bytes;
   }

}
```

Copy content of one file to another

```
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();
      }
    }
  }
}
```

Java Serialization

Java provides a mechanism, called object serialization where an object can be represented as a sequence of bytes that includes the object's data as well as information about the object's type and the types of data stored in the object.

After a serialized object has been written into a file, it can be read from the file and deserialized that is, the type information and bytes that represent the object and its data can be used to recreate the object in memory.

Most impressive is that the entire process is JVM independent, meaning an object can be serialized on one platform and deserialized on an entirely different platform.

Classes **ObjectInputStream** and **ObjectOutputStream** are high-level streams that contain the methods for serializing and deserializing an object.

The ObjectOutputStream class contains many write methods for writing various data types, but one method in particular stands out −

public final void writeObject(Object x) throws IOException

The above method serializes an Object and sends it to the output stream. Similarly, the ObjectInputStream class contains the following method for deserializing an object −

public final Object readObject() throws IOException, ClassNotFoundException

This method retrieves the next Object out of the stream and deserializes it. The return value is Object, so you will need to cast it to its appropriate data type.

To demonstrate how serialization works in Java, I am going to use the Employee class that we discussed early on in the book. Suppose that we have the following Employee class, which implements the Serializable interface −

Example

```
public class Employee implements java.io.Serializable {
   public String name;
   public String address;
   public transient int SSN;
   public int number;

   public void mailCheck() {
      System.out.println("Mailing a check to " + name + " " + address);
   }
}
```

Notice that for a class to be serialized successfully, two conditions must be met −

- The class must implement the java.io.Serializable interface.
- All of the fields in the class must be serializable. If a field is not serializable, it must be marked **transient**.

If you are curious to know if a Java Standard Class is serializable or not, check the documentation for the class. The test is simple: If the class implements java.io.Serializable, then it is serializable; otherwise, it's not.

Serializing an Object

The ObjectOutputStream class is used to serialize an Object. The following SerializeDemo program instantiates an Employee object and serializes it to a file.

When the program is done executing, a file named employee.ser is created. The program does not generate any output, but study the code and try to determine what the program is doing.

Note − When serializing an object to a file, the standard convention in Java is to give the file a **.ser** extension.

Example

```
import java.io.*;
public class SerializeDemo {

   public static void main(String [] args) {
      Employee e = new Employee();
      e.name = "Reyan Ali";
      e.address = "Phokka Kuan, Ambehta Peer";
      e.SSN = 11122333;
      e.number = 101;

      try {
         FileOutputStream fileOut =
         new FileOutputStream("/tmp/employee.ser");
         ObjectOutputStream out = new ObjectOutputStream(fileOut);
         out.writeObject(e);
         out.close();
         fileOut.close();
         System.out.printf("Serialized data is saved in /tmp/employee.ser");
      }catch(IOException i) {
         i.printStackTrace();
      }
   }
}
```

Deserializing an Object

The following DeserializeDemo program deserializes the Employee object created in the SerializeDemo program. Study the program and try to determine its output −

Example

```
import java.io.*;
public class DeserializeDemo {

   public static void main(String [] args) {
      Employee e = null;
```

```
    try {
      FileInputStream fileIn = new FileInputStream("/tmp/employee.ser");
      ObjectInputStream in = new ObjectInputStream(fileIn);
      e = (Employee) in.readObject();
      in.close();
      fileIn.close();
    }catch(IOException i) {
      i.printStackTrace();
      return;
    }catch(ClassNotFoundException c) {
      System.out.println("Employee class not found");
      c.printStackTrace();
      return;
    }

    System.out.println("Deserialized Employee...");
    System.out.println("Name: " + e.name);
    System.out.println("Address: " + e.address);
    System.out.println("SSN: " + e.SSN);
    System.out.println("Number: " + e.number);
  }
}
```

This will produce the following result −

Output

```
Deserialized Employee...
Name: Reyan Ali
Address:Phokka Kuan, Ambehta Peer
SSN: 0
Number:101
```

Here are following important points to be noted −

- The try/catch block tries to catch a ClassNotFoundException, which is declared by the readObject() method. For a JVM to be able to deserialize an object, it must be able to find the bytecode for the class. If the JVM can't find a class during the deserialization of an object, it throws a ClassNotFoundException.
- Notice that the return value of readObject() is cast to an Employee reference.
- The value of the SSN field was 11122333 when the object was serialized, but because the field is transient, this value was not sent to the output stream. The SSN field of the deserialized Employee object is 0.

Multithreaded server

Following example demonstrates how to create a multithreaded server by using ssock.accept() method of Socket class and MultiThreadServer(socketname) method of ServerSocket class.

```
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

public class MultiThreadServer implements Runnable {
  Socket csocket;
  MultiThreadServer(Socket csocket) {
    this.csocket = csocket;
  }

  public static void main(String args[])
  throws Exception {
    ServerSocket ssock = new ServerSocket(1234);
    System.out.println("Listening");
    while (true) {
      Socket sock = ssock.accept();
      System.out.println("Connected");
      new Thread(new MultiThreadServer(sock)).start();
    }
  }
  public void run() {
    try {
      PrintStream pstream = new PrintStream
      (csocket.getOutputStream());
      for (int i = 100; i >= 0; i--) {
        pstream.println(i +
        " bottles of beer on the wall");
      }
      pstream.close();
      csocket.close();
    }
    catch (IOException e) {
      System.out.println(e);
    }
  }
}
```

Result:

The above code sample will produce the following result.

```
Listening
Connected
```