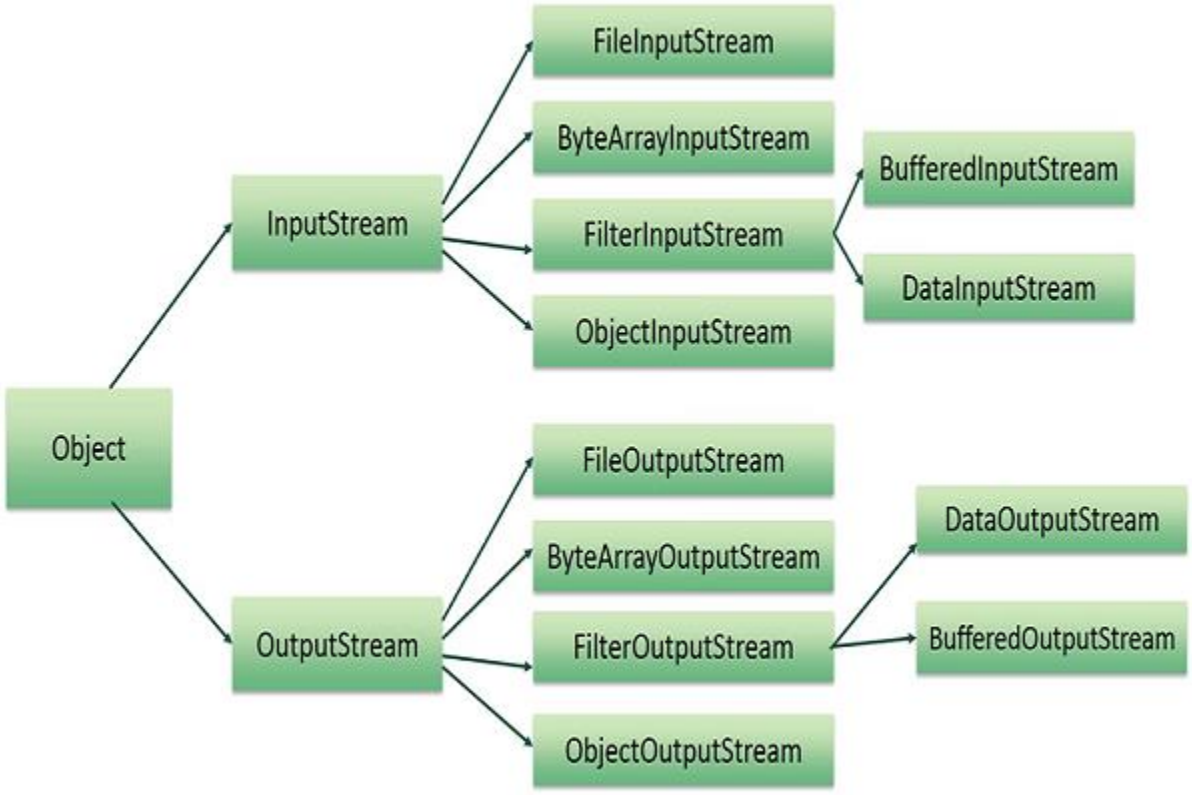
Reading and Writing Files

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

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



The two important streams are **FileInputStream** and **FileOutputStream**, which would be discussed here

FileInputStream

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

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

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

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

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

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

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public int read(int r)throws IOException{}** This method reads the specified byte of data from the InputStream. Returns an int. Returns the next byte of data and -1 will be returned if it's the end of the file.
4	**public int read(byte[] r) throws IOException{}** This method reads r.length bytes from the input stream into an array. Returns the total number of bytes read. If it is the end of the file, -1 will be returned.
5	**public int available() throws IOException{}** Gives the number of bytes that can be read from this file input stream. Returns an int.

FileOutputStream

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

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

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

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

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

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

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

Sr.No.	Method & Description
1	**public void close() throws IOException{}** This method closes the file output stream. Releases any system resources associated with the file. Throws an IOException.
2	**protected void finalize()throws IOException {}** This method cleans up the connection to the file. Ensures that the close method of this file output stream is called when there are no more references to this stream. Throws an IOException.
3	**public void write(int w)throws IOException{}** This methods writes the specified byte to the output stream.
4	**public void write(byte[] w)** Writes w.length bytes from the mentioned byte array to the OutputStream.

Example

Following is the example to demonstrate InputStream and OutputStream −

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

   public static void main(String args[]) {

      try {
         byte bWrite [] = {11,21,3,40,5};
         OutputStream os = new FileOutputStream("test.txt");
         for(int x = 0; x < bWrite.length ; x++) {
            os.write( bWrite[x] );   // writes the bytes
         }
         os.close();

         InputStream is = new FileInputStream("test.txt");
         int size = is.available();

         for(int i = 0; i < size; i++) {
            System.out.print((char)is.read() + "  ");
         }
         is.close();
      } catch (IOException e) {
         System.out.print("Exception");
      }
   }
}
```

The above code would create file test.txt and would write given numbers in binary format. Same would be the output on the stdout screen.

File Navigation and I/O

There are several other classes that we would be going through to get to know the basics of File Navigation and I/O.

- File Class
- FileReader Class
- FileWriter Class

Directories in Java

A directory is a File which can contain a list of other files and directories. You use **File** object to create directories, to list down files available in a directory. For complete detail, check a list of all the methods which you can call on File object and what are related to directories.

Creating Directories

There are two useful **File** utility methods, which can be used to create directories −

- The **mkdir( )** method creates a directory, returning true on success and false on failure. Failure indicates that the path specified in the File object already exists, or that the directory cannot be created because the entire path does not exist yet.
- The **mkdirs()** method creates both a directory and all the parents of the directory.

Following example creates "/tmp/user/java/bin" directory −

Example

```
import java.io.File;
public class CreateDir {

   public static void main(String args[]) {
      String dirname = "/tmp/user/java/bin";
      File d = new File(dirname);

      // Create directory now.
      d.mkdirs();
   }
}
```

Compile and execute the above code to create "/tmp/user/java/bin".

Note − Java automatically takes care of path separators on UNIX and Windows as per conventions. If you use a forward slash (/) on a Windows version of Java, the path will still resolve correctly.

Listing Directories

You can use **list( )** method provided by **File** object to list down all the files and directories available in a directory as follows −

Example

```
import java.io.File;
public class ReadDir {

   public static void main(String[] args) {
      File file = null;
      String[] paths;

      try {
         // create new file object
         file = new File("/tmp");

         // array of files and directory
         paths = file.list();

         // for each name in the path array
         for(String path:paths) {
            // prints filename and directory name
            System.out.println(path);
         }
      } catch (Exception e) {
         // if any error occurs
         e.printStackTrace();
      }
   }
}
```

This will produce the following result based on the directories and files available in your **/tmp** directory −

Output

```
test1.txt
test2.txt
ReadDir.java
ReadDir.class
```

Java 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.

Example of java PrintStream class

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

```
1.  package com.javatpoint;
2.
3.  import java.io.FileOutputStream;
4.  import java.io.PrintStream;
5.  public class PrintStreamTest{
6.   public static void main(String args[])throws Exception{
7.     FileOutputStream fout=new FileOutputStream("D:\\testout.txt ");
8.     PrintStream pout=new PrintStream(fout);
9.     pout.println(2016);
10.    pout.println("Hello Java");
11.    pout.println("Welcome to Java");
12.    pout.close();
13.    fout.close();
14.   System.out.println("Success?");
15.  }
16. }
17.
```

Output

```
Success...
```

The content of a text file **testout.txt** is set with the below data

```
2016
Hello Java
Welcome to Java
```

Example of printf() method using java PrintStream class:

Let's see the simple example of printing integer value by format specifier using **printf()** method of **java.io.PrintStream** class.

```
1.  class PrintStreamTest{
2.   public static void main(String args[]){
3.     int a=19;
4.     System.out.printf("%d",a); //Note: out is the object of printstream
5.   }
6.  }
```

Output

```
19
```

java PrintWriter class

Java PrintWriter class is the implementation of Writer class. It is used to print the formatted representation of objects to the text-output stream.

Class declaration

Let's see the declaration for Java.io.PrintWriter class:

```
1. public class PrintWriter extends Writer
```

Java PrintWriter Example

Let's see the simple example of writing the data on a **console** and in a **text file testout.txt** using Java PrintWriter class.

```
1. package com.javatpoint;
2.
3. import java.io.File;
4. import java.io.PrintWriter;
5. public class PrintWriterExample {
6.     public static void main(String[] args) throws Exception {
7.           //Data to write on Console using PrintWriter
8.       PrintWriter writer = new PrintWriter(System.out);
9.       writer.write("Javatpoint provides tutorials of all technology.");
10.  writer.flush();
11.      writer.close();
12. //Data to write in File using PrintWriter
13.      PrintWriter writer1 =null;
14.        writer1 = new PrintWriter(new File("D:\\testout.txt"));
15.        writer1.write("Like Java, Spring, Hibernate, Android, PHP etc.");

16.                  writer1.flush();
17.        writer1.close();
18.    }
19. }
```

Outpt

```
Javatpoint provides tutorials of all technology.
```

The content of a text file **testout.txt** is set with the data **Like Java, Spring, Hibernate, Android, PHP etc**

Java - RandomAccessFile

This class is used for reading and writing to random access file. A random access file behaves like a large array of bytes. There is a cursor implied to the array called file pointer, by moving the cursor we do the read write operations. If end-of-file is reached before the desired number of byte has been read than EOFException is thrown. It is a type of IOException.

Example

```
1.  import java.io.IOException;
2.  import java.io.RandomAccessFile;
3.
4.  public class RandomAccessFileExample {
5.     static final String FILEPATH ="myFile.TXT";
6.     public static void main(String[] args) {
7.        try {
8.           System.out.println(new String(readFromFile(FILEPATH, 0, 18)));
9.           writeToFile(FILEPATH, "I love my country and my people", 31);
10.       } catch (IOException e) {
11.          e.printStackTrace();
12.       }
13.    }
14.    private static byte[] readFromFile(String filePath, int position, int size)
15.          throws IOException {
16.       RandomAccessFile file = new RandomAccessFile(filePath, "r");  // file can be read but not
    written
17.       file.seek(position);
18.       byte[] bytes = new byte[size];
19.       file.read(bytes);
20.       file.close();
21.       return bytes;
22.    }
23.    private static void writeToFile(String filePath, String data, int position)
24.          throws IOException {
25.       RandomAccessFile file = new RandomAccessFile(filePath, "rw");  file is open in read-write
    mode
26.       file.seek(position);   //use to set current position of file pointer within file
27.       file.write(data.getBytes());
28.       file.close();
29.    }
30. }
```

The myFile.TXT contains text "This class is used for reading and writing to random access file."

after running the program it will contains

This class is used for reading I love my country and my people.

Serialization

Serialization in Java is a mechanism of *writing the state of an object into a byte-stream*

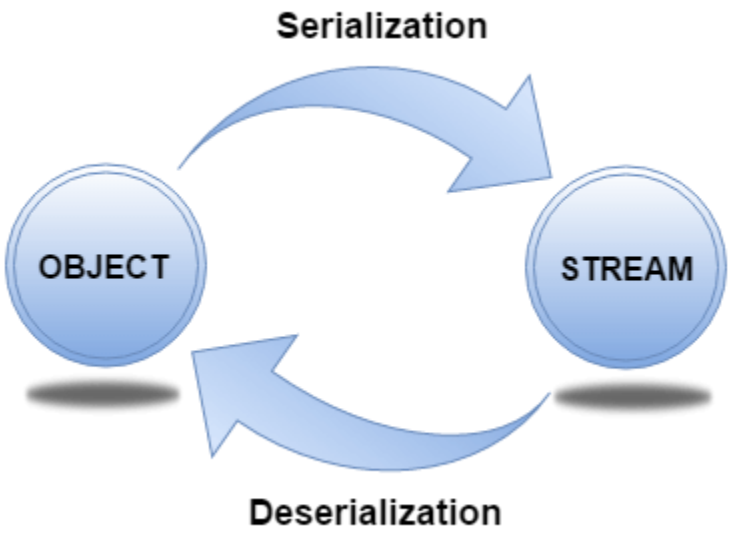
The reverse operation of serialization is called *deserialization* where byte-stream is converted into an object. The serialization and deserialization process is platform-independent, it means you can serialize an object on one platform and deserialize it on a different platform.

For serializing the object, we call the **writeObject()** method of *ObjectOutputStream* class, and for deserialization we call the **readObject()** method of *ObjectInputStream* class.

We must have to implement the *Serializable* interface for serializing the object.

Advantages of Java Serialization

It is mainly used to travel object's state on the network (that is known as marshalling).



java.io.Serializable interface

Serializable is a marker interface (has no data member and method). It is used to "mark" Java classes so that the objects of these classes may get a certain capability. The **Cloneable** and **Remote** are also marker interfaces.

The **Serializable** interface must be implemented by the class whose object needs to be persisted.

The String class and all the wrapper classes implement the *java.io.Serializable* interface by default.

Let's see the example given below:

Student.java

```
1. import java.io.Serializable;
2. public class Student implements Serializable{
3.  int id;
4.  String name;
5.  public Student(int id, String name) {
6.   this.id = id;
7.   this.name = name;
8.  }
9. }
```

In the above example, ***Student*** class implements Serializable interface. Now its objects can be converted into stream. The main class implementation of is showed in the next code.

Example of Java Serialization

In this example, we are going to serialize the object of ***Student*** class from above code. The writeObject() method of ObjectOutputStream class provides the functionality to serialize the object. We are saving the state of the object in the file named f.txt.

Persist.java

```
1. import java.io.*;
2. class Persist{
3.  public static void main(String args[]){
4.   try{
5.   //Creating the object
6.   Student s1 =new Student(211,"ravi");
7.   //Creating stream and writing the object
8.   FileOutputStream fout=new FileOutputStream("f.txt");
9.   ObjectOutputStream out=new ObjectOutputStream(fout);
10.  out.writeObject(s1);
11.  out.flush();
12.  //closing the stream
13.  out.close();
14.  System.out.println("success");
15.  }catch(Exception e){System.out.println(e);}
16. }
17. }
```

Output:

```
success
```

Example of Java Deserialization

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Deserialization is the process of reconstructing the object from the serialized state. It is the reverse operation of serialization. Let's see an example where we are reading the data from a deserialized object.

Depersist.java

```
1.  import java.io.*;
2.  class Depersist{
3.   public static void main(String args[]){
4.    try{
5.    //Creating stream to read the object
6.    ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
7.    Student s=(Student)in.readObject();
8.    //printing the data of the serialized object
9.    System.out.println(s.id+" "+s.name);
10.   //closing the stream
11.   in.close();
12.   }catch(Exception e){System.out.println(e);}
13.  }
14. }
```

Output:

```
211 ravi
```

download this example of deserialization

Java Serialization with Inheritance (IS-A Relationship)

If a class implements **Serializable interface** then all its sub classes will also be serializable. Let's see the example given below:

SerializeISA.java

```
1.  import java.io.Serializable;
2.  class Person implements Serializable{
3.   int id;
4.   String name;
```

```
    Person(int id, String name) {
     this.id = id;
     this.name = name;
    }
   }
class Student extends Person{
  String course;
  int fee;
  public Student(int id, String name, String course, int fee) {
   super(id,name);
   this.course=course;
   this.fee=fee;
  }
}
public class SerializeISA
{
  public static void main(String args[])
  {
     try{
   //Creating the object
   Student s1 =new Student(211,"ravi","Engineering",50000);
   //Creating stream and writing the object
   FileOutputStream fout=new FileOutputStream("f.txt");
   ObjectOutputStream out=new ObjectOutputStream(fout);
   out.writeObject(s1);
   out.flush();
   //closing the stream
   out.close();
   System.out.println("success");
   }catch(Exception e){System.out.println(e);}
   try{
   //Creating stream to read the object
   ObjectInputStream in=new ObjectInputStream(new FileInputStream("f.txt"));
   Student s=(Student)in.readObject();
   //printing the data of the serialized object
   System.out.println(s.id+" "+s.name+" "+s.course+" "+s.fee);
   //closing the stream
   in.close();
   }catch(Exception e){System.out.println(e);}
  }
}
```

Output:

```
success
211 ravi Engineering 50000
```

The SerializeISA class has serialized the Student class object that extends the Person class which is Serializable. Parent class properties are inherited to subclasses so if parent class is Serializable, subclass would also be.

.