Java I/O

**Java I/O** (Input and Output) is used *to process the input* and *produce the output*.

Java uses the concept of a stream to make I/O operation fast. The java.io package contains all the classes required for input and output operations.

We can perform **file handling in Java** by Java I/O API.

Stream

A stream is a sequence of data. In Java, a stream is composed of bytes. It's called a stream because it is like a stream of water that continues to flow..2K

**Byte Oriented Stream and Character Oriented Stream:**

A byte stream access the file byte by byte. A byte stream is suitable for any kind of file, however not quite appropriate for text files. For example, if the file is using a unicode encoding and a character is represented with two bytes, the byte stream will treat these separately and you will need to do the conversion yourself.

A character stream will read a file character by character. A character stream needs to be given the file's encoding in order to work properly.

mer for the Riot Balance Team - Best of LoL Streams #371

In Java, 3 streams are created for us automatically. All these streams are attached with the console.

**1) System.out:**standard output stream

**2) System.in:**standard input stream

**3) System.err:**standard error stream

Let's see the code to print **output and an error** message to the console.

System.out.println("simple message");

System.err.println("error message");

Let's see the code to get **input** from console.

1. **int** i=System.in.read();//returns ASCII code of 1st character
2. System.out.println((**char**)i);//will print the character

OutputStream vs InputStream

The explanation of OutputStream and InputStream classes are given below:

OutputStream

Java application uses an output stream to write data to a destination; it may be a file, an array, peripheral device or socket.

InputStream

Java application uses an input stream to read data from a source; it may be a file, an array, peripheral device or socket.

Let's understand the working of Java OutputStream and InputStream by the figure given below.

Java IO

OutputStream class

OutputStream class is an abstract class. It is the superclass of all classes representing an output stream of bytes. An output stream accepts output bytes and sends them to some sink.

Useful methods of OutputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public void write(int)throws IOException | is used to write a byte to the current output stream. |
| 2) public void write(byte[])throws IOException | is used to write an array of byte to the current output stream. |
| 3) public void flush()throws IOException | flushes the current output stream. |
| 4) public void close()throws IOException | is used to close the current output stream. |

OutputStream Hierarchy

Java output stream hierarchy

InputStream class

InputStream class is an abstract class. It is the superclass of all classes representing an input stream of bytes.

Useful methods of InputStream

|  |  |
| --- | --- |
| **Method** | **Description** |
| 1) public abstract int read()throws IOException | reads the next byte of data from the input stream. It returns -1 at the end of the file. |
| 2) public int available()throws IOException | returns an estimate of the number of bytes that can be read from the current input stream. |
| 3) public void close()throws IOException | is used to close the current input stream. |

InputStream Hierarchy

Java input stream hierarchy

# Java FileOutputStream Class

Java FileOutputStream is an output stream used for writing data to a [file](https://www.javatpoint.com/java-file-class).

If you have to write primitive values into a file, use FileOutputStream class. You can write byte-oriented as well as character-oriented data through FileOutputStream class. But, for character-oriented data, it is preferred to use [FileWriter](https://www.javatpoint.com/java-filterwriter-class) than FileOutputStream.

Example

**import** java.io.FileOutputStream;

**public** **class** FileOutputStreamExample {

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

**try**{

             FileOutputStream fout=**new** FileOutputStream("D:\\testout.txt");

             fout.write(65);

             fout.close();

             System.out.println("success...");

            }**catch**(Exception e){System.out.println(e);}

      }

}

# Java FileInputStream Class

Java FileInputStream class obtains input bytes from a [file](https://www.javatpoint.com/java-file-class). It is used for reading byte-oriented data (streams of raw bytes) such as image data, audio, video etc. You can also read character-stream data. But, for reading streams of characters, it is recommended to use [FileReader](https://www.javatpoint.com/java-filereader-class) class.

Ex:

**import** java.io.FileInputStream;

**public** **class** DataStreamExample {

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

**try**{

            FileInputStream fin=**new** FileInputStream("D:\\testout.txt");

**int** i=fin.read();

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

            fin.close();

          }**catch**(Exception e){System.out.println(e);}

         }

        }

# Serialization and Deserialization

**Serialization in Java** is a mechanism of *writing the state of an object into a byte-stream*. It is mainly used in Hibernate, RMI, JPA, EJB and JMS technologies.

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 in a platform and deserialize in different platform.

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

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

Difference between JDK, JRE, and JVM

Advantages of Java Serialization

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



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.

It must be implemented by the class whose object you want to persist.

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

Let's see the example given below:

**import** java.io.Serializable;

**public** **class** Student **implements** Serializable{

**int** id;

 String name;

**public** Student(**int** id, String name) {

**this**.id = id;

**this**.name = name;

 }

}

In the above example, Student class implements Serializable interface. Now its objects can be converted into stream.

ObjectOutputStream class

The ObjectOutputStream class is used to write primitive data types, and Java objects to an OutputStream. Only objects that support the java.io.Serializable interface can be written to streams.

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Example of Java Serialization

In this example, we are going to serialize the object of Student class. 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.

**import** java.io.\*;

**class** Persist{

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

**try**{

  //Creating the object

  Student s1 =**new** Student(211,"ravi");

  //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);}

 }

}

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.

**import** java.io.\*;

**class** Depersist{

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

**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);

  //closing the stream

  in.close();

  }**catch**(Exception e){System.out.println(e);}

 }

}

211 ravi

In Java, **Enumerations or Java Enum** serve the purpose of representing a group of named constants in a programming language. Java Enums are used when we know all possible values at compile time, such as choices on a menu, rounding modes, command-line flags, etc. The set of constants in an enum type doesn’t need to stay fixed for all time.

## What is Enumeration or Enum in Java?

A Java enumeration is a class type. Although we don’t need to instantiate an enum using **new,**it has the same capabilities as other classes. This fact makes Java enumeration a very powerful tool. Just like classes, you can give them constructors, add instance variables and methods, and even implement interfaces.

#### Enum Example:

## ****Declaration of enum in Java****

Enum declaration can be done outside a Class or inside a Class but not inside a Method.

### Declaration outside the class

|  |
| --- |
| // A simple enum example where enum is declared  // outside any class (Note enum keyword instead of  // class keyword)    enum Color {      RED,      GREEN,      BLUE;  }    public class Test {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

**Output**

RED

### 2. Declaration inside a class

* Java

|  |
| --- |
| // enum declaration inside a class.    public class Test {      enum Color {          RED,          GREEN,          BLUE;      }        // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);      }  } |

**Output**

RED

* The first line inside the enum should be a list of constants and then other things like methods, variables, and constructors.
* According to [Java naming conventions](http://www.oracle.com/technetwork/java/codeconventions-135099.html), it is recommended that we name constant with all capital letters

## Properties of Enum in Java

There are certain properties followed by Enum as mentioned below:

* Every enum is internally implemented by using Class.
* Every enum constant represents an **object** of type enum.
* Enum type can be passed as an argument to **switch** statements.
* Every enum constant is always implicitly **public static final**. Since it is **static**, we can access it by using the enum Name. Since it is **final**, we can’t create child enums.
* We can declare the **main() method** inside the enum. Hence we can invoke the enum directly from the Command Prompt.

**Below is the implementation of the above properties:**

* Java

|  |
| --- |
| // A Java program to demonstrate working on enum  // in switch case (Filename Test. Java)    import java.util.Scanner;    // An Enum class  enum Day {      SUNDAY,      MONDAY,      TUESDAY,      WEDNESDAY,      THURSDAY,      FRIDAY,      SATURDAY;  }    // Driver class that contains an object of "day" and  // main().  public class Test {      Day day;        // Constructor      public Test(Day day) { this.day = day; }        // Prints a line about Day using switch      public void dayIsLike()      {          switch (day) {          case MONDAY:              System.out.println("Mondays are bad.");              break;          case FRIDAY:              System.out.println("Fridays are better.");              break;          case SATURDAY:          case SUNDAY:              System.out.println("Weekends are best.");              break;          default:              System.out.println("Midweek days are so-so.");              break;          }      }        // Driver method      public static void main(String[] args)      {          String str = "MONDAY";          Test t1 = new Test(Day.valueOf(str));          t1.dayIsLike();      }  } |

**Output**

Mondays are bad.

**Loop through Enum**

We can iterate over the Enum using values( ) and loop. values() function returns an array of Enum values as constants using which we can iterate over the values.

**Below is the implementation of the loop through Enum:**

* Java

|  |
| --- |
| // Java Program to Print all the values  // inside the enum using for loop  import java.io.\*;    // Enum Declared  enum Color {      RED,      GREEN,      BLUE;  }    // Driver Class  class GFG {        // Main Function      public static void main(String[] args)      {          // Iterating over all the values in          // enum using for loop          for (Color var\_1 : Color.values()) {              System.out.println(var\_1);          }      }  } |

**Output**

RED

GREEN

BLUE

**Enum and Methods**

Enum can contain both **concrete** methods and **abstract** methods. If an enum class has an abstract method, then each instance of the enum class must implement it.

* Java

|  |
| --- |
| // Java program to demonstrate that enums can have  // constructor and concrete methods.    // An enum (Note enum keyword inplace of class keyword)  enum Color {      RED,      GREEN,      BLUE;        // enum constructor called separately for each      // constant      private Color()      {          System.out.println("Constructor called for : "                             + this.toString());      }        public void colorInfo()      {          System.out.println("Universal Color");      }  }    public class Test {      // Driver method      public static void main(String[] args)      {          Color c1 = Color.RED;          System.out.println(c1);          c1.colorInfo();      }  } |

**Output**

Constructor called for : RED

Constructor called for : GREEN

Constructor called for : BLUE

RED

Universal Color