Top of Form

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

#### Stream

A stream can be defined as a sequence of data. There are two kinds of Streams −

* InPutStream − The InputStream is used to read data from a source.
* OutPutStream − The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one −

#### Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes are, FileInputStream and FileOutputStream. Following is an example which makes use of these two classes to copy an input file into an output file −

Example

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();

}

}

}

}

Now let's have a file input.txt with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

#### Character Streams

Java Byte streams are used to perform input and output of 8-bit bytes, whereas Java Character streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently used classes are, FileReader and FileWriter. Though internally FileReader uses FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file −

Example

import java.io.\*;

public class CopyFile {

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

FileReader in = null;

FileWriter out = null;

try {

in = new FileReader("input.txt");

out = new FileWriter("output.txt");

int c;

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

out.write(c);

}

}finally {

if (in != null) {

in.close();

}

if (out != null) {

out.close();

}

}

}

}

Now let's have a file input.txt with the following content −

This is test for copy file.

As a next step, compile the above program and execute it, which will result in creating output.txt file with the same content as we have in input.txt. So let's put the above code in CopyFile.java file and do the following −

$javac CopyFile.java

$java CopyFile

#### Standard Streams

All the programming languages provide support for standard I/O where the user's program can take input from a keyboard and then produce an output on the computer screen. If you are aware of C or C++ programming languages, then you must be aware of three standard devices STDIN, STDOUT and STDERR. Similarly, Java provides the following three standard streams −

* Standard Input − This is used to feed the data to user's program and usually a keyboard is used as standard input stream and represented as System.in.
* Standard Output − This is used to output the data produced by the user's program and usually a computer screen is used for standard output stream and represented as System.out.
* Standard Error − This is used to output the error data produced by the user's program and usually a computer screen is used for standard error stream and represented as System.err.

Following is a simple program, which creates InputStreamReader to read standard input stream until the user types a "q" −

Example

[Live Demo](http://tpcg.io/lVH2u1)

import java.io.\*;

public class ReadConsole {

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

InputStreamReader cin = null;

try {

cin = new InputStreamReader(System.in);

System.out.println("Enter characters, 'q' to quit.");

char c;

do {

c = (char) cin.read();

System.out.print(c);

} while(c != 'q');

}finally {

if (cin != null) {

cin.close();

}

}

}

}

Let's keep the above code in ReadConsole.java file and try to compile and execute it as shown in the following program. This program continues to read and output the same character until we press 'q' −

$javac ReadConsole.java

$java ReadConsole

Enter characters, 'q' to quit.

1

1

e

e

q

q

#### 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 in this tutorial.

#### 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.

There are other important input streams available, for more detail you can refer to the following links −

* [ByteArrayInputStream](https://www.tutorialspoint.com/java/java_bytearrayinputstream.htm)
* [DataInputStream](https://www.tutorialspoint.com/java/java_datainputstream.htm)

#### 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.

Bottom of Form

Top of Form

There are other important output streams available, for more detail you can refer to the following links −

* [ByteArrayOutputStream](https://www.tutorialspoint.com/java/java_bytearrayoutputstream.htm)
* [DataOutputStream](https://www.tutorialspoint.com/java/java_dataoutputstream.htm)

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.

Bottom of Form

Top of Form

#### 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.

Bottom of Form

Top of Form

Clone() method in Java

Object cloning refers to creation of exact copy of an object. It creates a new instance of the class of current object and initializes all its fields with exactly the contents of the corresponding fields of this object.

**Using Assignment Operator to create copy of reference variable**

In Java, there is no operator to create copy of an object. Unlike C++, in Java, if we use assignment operator then it will create a copy of reference variable and not the object. This can be explained by taking an example. Following program demonstrates the same.

// Java program to demonstrate that assignment

// operator only creates a new reference to same

// object.

import java.io.\*;

// A test class whose objects are cloned

class Test

{

int x, y;

Test()

{

x = 10;

y = 20;

}

}

// Driver Class

class Main

{

public static void main(String[] args)

{

Test ob1 = new Test();

System.out.println(ob1.x + " " + ob1.y);

// Creating a new reference variable ob2

// pointing to same address as ob1

Test ob2 = ob1;

// Any change made in ob2 will be reflected

// in ob1

ob2.x = 100;

System.out.println(ob1.x+" "+ob1.y);

System.out.println(ob2.x+" "+ob2.y);

}

}

**Creating a copy using clone() method**

The class whose object’s copy is to be made must have a public clone method in it or in one of its parent class.

* Every class that implements clone() should call super.clone() to obtain the cloned object reference.
* The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when clone method is called on that class’s object.
* Syntax:
* protected Object clone() throws CloneNotSupportedException

**Usage of clone() method -Shallow Copy**

// A Java program to demonstrate shallow copy

// using clone()

import java.util.ArrayList;

// An object reference of this class is

// contained by Test2

class Test

{

int x, y;

}

// Contains a reference of Test and implements

// clone with shallow copy.

class Test2 implements Cloneable

{

int a;

int b;

Test c = new Test();

public Object clone() throws

CloneNotSupportedException

{

return super.clone();

}

}

// Driver class

public class Main

{

public static void main(String args[]) throws

CloneNotSupportedException

{

Test2 t1 = new Test2();

t1.a = 10;

t1.b = 20;

t1.c.x = 30;

t1.c.y = 40;

Test2 t2 = (Test2)t1.clone();

// Creating a copy of object t1 and passing

// it to t2

t2.a = 100;

// Change in primitive type of t2 will not

// be reflected in t1 field

t2.c.x = 300;

// Change in object type field will be

// reflected in both t2 and t1(shallow copy)

System.out.println(t1.a + " " + t1.b + " " +

t1.c.x + " " + t1.c.y);

System.out.println(t2.a + " " + t2.b + " " +

t2.c.x + " " + t2.c.y);

}

}

**Output:**

10 20 300 40

100 20 300 40

In the above example, t1.clone returns the shallow copy of the object t1. To obtain a deep copy of the object certain modifications have to be made in clone method after obtaining the copy.

**Deep Copy vs Shallow Copy**

* **Shallow copy** is method of copying an object and is followed by default in cloning. In this method the fields of an old object X are copied to the new object Y. While copying the object type field the reference is copied to Y i.e object Y will point to same location as pointed out by X. If the field value is a primitive type it copies the value of the primitive type.
* Therefore, any changes made in referenced objects in object X or Y will be reflected in other object.

Shallow copies are cheap and simple to make. In above example, we created a shallow copy of object.

**Usage of clone() method – Deep Copy**

* If we want to create a deep copy of object X and place it in a new object Y then new copy of any referenced objects fields are created and these references are placed in object Y. This means any changes made in referenced object fields in object X or Y will be reflected only in that object and not in the other. In below example, we create a deep copy of object.
* A deep copy copies all fields, and makes copies of dynamically allocated memory pointed to by the fields. A deep copy occurs when an object is copied along with the objects to which it refers.

// A Java program to demonstrate deep copy

// using clone()

import java.util.ArrayList;

// An object reference of this class is

// contained by Test2

class Test

{

int x, y;

}

// Contains a reference of Test and implements

// clone with deep copy.

class Test2 implements Cloneable

{

int a, b;

Test c = new Test();

public Object clone() throws

CloneNotSupportedException

{

// Assign the shallow copy to new reference variable t

Test2 t = (Test2)super.clone();

t.c = new Test();

// Create a new object for the field c

// and assign it to shallow copy obtained,

// to make it a deep copy

return t;

}

}

public class Main

{

public static void main(String args[]) throws

CloneNotSupportedException

{

Test2 t1 = new Test2();

t1.a = 10;

t1.b = 20;

t1.c.x = 30;

t1.c.y = 40;

Test2 t3 = (Test2)t1.clone();

t3.a = 100;

// Change in primitive type of t2 will not

// be reflected in t1 field

t3.c.x = 300;

// Change in object type field of t2 will not

// be reflected in t1(deep copy)

System.out.println(t1.a + " " + t1.b + " " +

t1.c.x + " " + t1.c.y);

System.out.println(t3.a + " " + t3.b + " " +

t3.c.x + " " + t3.c.y);

}

}

**Output:**

10 20 30 40

100 20 300 0

In the above example, we can see that a new object for Test class has been assigned to copy object that will be returned in clone method.Due to this t3 will obtain a deep copy of the object t1. So any changes made in ‘c’ object fields by t3 ,will not be reflected in t1.

**Advantages of clone method:**

* If we use assignment operator to assign an object reference to another reference variable then it will point to same address location of the old object and no new copy of the object will be created. Due to this any changes in reference variable will be reflected in original object.
* If we use copy constructor, then we have to copy all of the data over explicitly i.e. we have to reassign all the fields of the class in constructor explicitly. But in clone method this work of creating a new copy is done by the method itself.So to avoid extra processing we use object cloning.

Bottom of Form

Top of Form

**Reference Links:**

https://www.javatpoint.com/java-io

https://www.tutorialspoint.com/java/java\_files\_io.htm

https://docs.oracle.com/javase/tutorial/essential/io/streams.html

https://www.programiz.com/java-programming/io-streams

http://tutorials.jenkov.com/java-io/streams.html

https://www.studytonight.com/java/java-io-stream.php

https://www.tutorialspoint.com/java/java\_serialization.htm

**Video Links:**

https://youtu.be/3YRahx2ltSg

https://youtu.be/mq-f7zPZ7b8

https://youtu.be/6B6vp0jZnb0

Bottom of Form