**Java Directories**

A directory is Java File Class that contains a list of other files and directories. When we create a File object that is a directory, the isDiretory() method will return true. In this case, you can call list() on that object to extract the list of other files and directories inside. It has two forms. The first is shown here**:**

**package** day\_20;

**import** java.io.File;

**public** **class** DirectoriesDemo {

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

String dirname = "/Users/sabniss/Desktop/java-training/java-training/src";

File f1 = **new** File(dirname); // LINE A

System.***out***.println(f1.getAbsolutePath()); // LINE B

**if**(f1.isDirectory()) // LINE C

{

System.***out***.println("Directory of " + dirname);

String s[] = f1.list(); // LINE D

**for**(**int** i = 0; i < s.length; i++)

{

File f = **new** File(dirname + "/" + s[i]);

**if**(f.isDirectory())

{

System.***out***.println(s[i] + " is a directory");

}

**else**

System.***out***.println(s[i] + " is a file");

}

}

**else**

System.***out***.println(dirname + " is not a directory");

}

}

**Java Input Stream Class**

InputStream is an abstract superclass of all classes representing an input stream of bytes. The subclassess of InputStream are AudioInputStream, ByteArrayInputStream, FileInputStream, FilterInputStream, ObjectInputStream, PipedInputStream, SequenceInputStream, StringBufferInputStream. These are used to read data in bytes.

**public** **class** InputStreamDemo {

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

File f = **new** File("/Users/sabniss/Desktop/java-training/java-training/src/day\_20/sample1.txt"); // LINE A

InputStream is = **null**;

**if** (f.exists()) // LINE C

{

is = **new** FileInputStream(f); // LINE B

is.close(); // LINE G

System.***out***.println("File exists.");

// LINE D

**int** i = 0;

// LINE E

**while** ((i = is.read()) != -1) {

System.***out***.print((**char**) i); // LINE F

}

} **else** {

System.***out***.println("File not found.");

}

}

}

**Java OutputStream Class**

OutputStream is an abstract superclass of all classes representing an output stream of bytes. The subclass which inherit from OutputStream are ByteArrayOutputStream, FileOutputStream, FilterOutputStream, ObjectOutputStream, OutputStream, PipedOutputStream. These OutputStream sub classes are used to write bytes of data.

**public** **class** OutputStreamDemo {

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

File f = **new** File("/Users/sabniss/Desktop/java-training/java-training/src/day\_20/sample1.txt"); // LINE A

OutputStream os = **new** FileOutputStream(f); // LINE B

**if**(f.exists()) // LINE C

{

System.***out***.println("File exists.");

// LINE D

**byte** b[] = {'i',' ','a','m',' ','f','i','l','e','O','n','e','.'};

// Writing into file fileOne

os.write(b);

}

**else**

System.***out***.println("File not found.");

os.close(); // LINE F

// Reading from fileOne

InputStream is = **new** FileInputStream(f);

**int** i = 0;

**while**((i = is.read()) != -1)

{

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

}

is.close();

}

}

**Buffered Reader**

BufferedReader reads text from a character-input stream, to provide buffering for the efficient reading of characters, arrays and lines.

* The buffer size may be specified, or the default size can be used.
* Each read request causes a corresponding read request to character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as [Java FileReader](http://java.meritcampus.com/core-java-topics/java-filereader-or-filereader-in-java) and InputStreamReaders. For example,

BufferedReader in = new BufferedReader(new FileReader("F:\\test.txt"));

this will buffer the input from the test.txt file.

**public** **class** BufferedReaderExample {

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

**try** {

BufferedReader br = **new** BufferedReader(**new** FileReader("/Users/sabniss/Desktop/java-training/java-training/src/day\_20/sample1.txt"));

System.***out***.println((**char**) br.read()); // LINE A

System.***out***.println((**char**) br.read());

**boolean** ready = **false**;

ready = br.ready(); // LINE C

System.***out***.println("Buffered reader is ready : " + ready);

br.skip(5); // LINE F

System.***out***.println("skipped 2 characters");

System.***out***.println((**char**) br.read());

br.close(); // LINE G

br.read(); // LINE H

} **catch** (IOException e) {

System.***out***.println("You cannot read file because buffered Reader is closed.");

e.printStackTrace();

}

}

}

**Buffered Writer**

BufferedWriter writes text to a character-output stream, to provide buffering for the efficient writing of single characters, arrays and strings.

* The buffer size may be specified, or the default size may be used.
* A Writer sends its output immediately to the underlying character or byte stream. It is advisable to wrap a BufferedWriter around any Writer whose Write() operations may be costly, such as [Java FileWriter](http://java.meritcampus.com/core-java-topics/java-filewriter-or-filewriter-in-java) and OutputStreamWriters. For example,

**public** **class** BufferedWriterExample {

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

**try** {

String content = "This is the content to write into file.";

File file = **new** File("/Users/sabniss/Desktop/java-training/java-training/src/day\_20/sample1.txt");

**if** (!file.exists()) {

file.createNewFile();

}

FileWriter fw = **new** FileWriter(file.getAbsoluteFile());

BufferedWriter bw = **new** BufferedWriter(fw);

bw.write(content); // LINE A

bw.close();

System.***out***.println("Done");

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

**Serialization**

Serialization is the process of storing object data in the form of bytes. The data converted can be stored in a file or transmitted over network. This is very useful and important to save users data and retrieve at a later time. Serialization can be achieved using two interfaces - Serializable and Externalizable.

A class can be serialized if we simply implement the Serializable interface. For e.g., the Student class below can be serialized since it implements Serializable interface.

class Student implements Serializable  
{  
    String name;  
    int rollNumber;  
    char section;  
}

We need to understand the following important points about Serializable.

* Any class which extends from Student will automatically be Serializable. For e.g., the Engineer class below is Serializable since Student implements Serializable interface.

class Engineer extends Student  
{  
    String branch;  
}

* If a class contains objects of other classes, then those classes should also implement Serializable interface, otherwise the containing class can not be serialized. For e.g., Student class contains Address and if Address does not implement Serializable interface, then it throws a NotSerializableException if we try to serialize an object of Student class.

/\*  
// WON'T WORK SINCE NOT IMPLEMENTING Serializable INTERFACE  
class Address  
{  
    String doorNumber;  
    String line1;  
}  
\*/  
class Address implements Serializable  
{  
    String doorNumber;  
    String line1;  
}  
class Student implements Serializable  
{  
    String name;  
    int rollNumber;  
    char section;  
    Address address; // ANY CLASS INCLUDED SHOULD ALSO IMPLEMENT Serializable  
}

* Any variables which are declared as static or transient will not be serialized. Which means the value of those variables will be lost if we serialize and de-serialize. For e.g., below the values of allStudentsCount and percentage will be lost and they will set to their default values 0 and 0.0 after the serialization/de-serialization.

class Student implements Serializable  
{  
    static int allStudentsCount; // WON'T BE SERIALIZED  
    String name;  
    int rollNumber;  
    char section;  
    transient double percentage; // WON'T BE SERIALIZED  
}  
  
Student.allStudentsCount = 25;  
Student ravi = new Student();  
ravi.name = "Ravi";  
ravi.rollNumber = 35;  
ravi.section = 'B';  
ravi.percentage = 79.34;  
  
// AFTER SERIALIZATION AND DESERIALIZATION THE VALUES OF allStudentsCount AND percentage WILL BE LOST



**Java Serialization Process**

As discussed in Serialization In Java , the process of saving and restoring the object data is called Serialization. The interfaces Serialization In Java and Externalizable In Java with Example makes an object serializable. This process is used for storing data to files or for sending data over network.

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**import** java.io.ObjectOutputStream;

**import** java.io.Serializable;

**public** **class** SerializableTest {

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

// SERIALIZATION - START

Student hemanth = **new** Student();

hemanth.name = "Prakash";

hemanth.rollNumber = 25;

hemanth.section = 'C';

FileOutputStream fileOutputStream = **new** FileOutputStream(

"/Users/sabniss/Desktop/java-training/java-training/src/day\_20/serialization.txt");

ObjectOutputStream objectOutputStream = **new** ObjectOutputStream(fileOutputStream);

objectOutputStream.writeObject(hemanth);

objectOutputStream.flush();

objectOutputStream.close();

}

}

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

String name = "";

**int** rollNumber = 0;

**char** section = 'A'; // LINE C

}

**public** **class** SerializableRead {

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

FileInputStream fileInputStream = **new** FileInputStream("/Users/sabniss/Desktop/java-training/java-training/src/day\_20/serialization.txt");

ObjectInputStream objectInputStream = **new** ObjectInputStream(fileInputStream);

Student retreiveStudent = (Student) objectInputStream.readObject();

objectInputStream.close();

System.***out***.println(retreiveStudent.name);

System.***out***.println(retreiveStudent.section);

}

}