**Java File I/O** [https://docs.oracle.com/javase/tutorial/essential/io/streams.html]

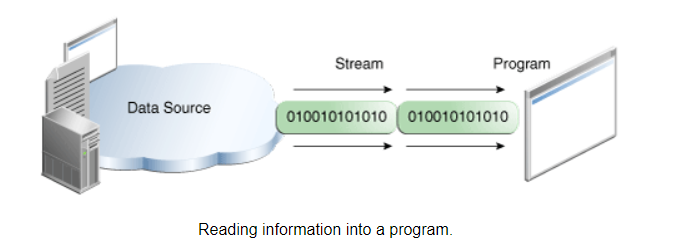
* **I/O Streams** 
  + **Byte Streams** - handle I/O of raw binary data
  + **Character Streams** - handle I/O of character data, automatically handling translation to and from the local character set
  + **Buffered Streams** - optimize input and output by reducing the number of calls to the native API
  + **Scanning and Formatting** - allow programs to read and write formatted text
  + **I/O from Command Line** describes the standard streams and Console object
  + **Data Streams** - handle binary I/O of primitive data type and String values
  + **Object Streams** - handle I/O of objects
* **File I/O**
  + What is Path? examines concept of a path on a file system
  + The Path class - introduces the cornerstone class of the java.nio.file package
  + Path Operations
  + File Operations
  + Checking a File or Directory
  + Copying File or Directory
  + Deleting File or Directory
  + Moving File or Directory
  + Managing Metadata
  + Reading, Writing and Creating Files
  + Random Access Files
  + Creating and Reading Directories

**I/O Streams**

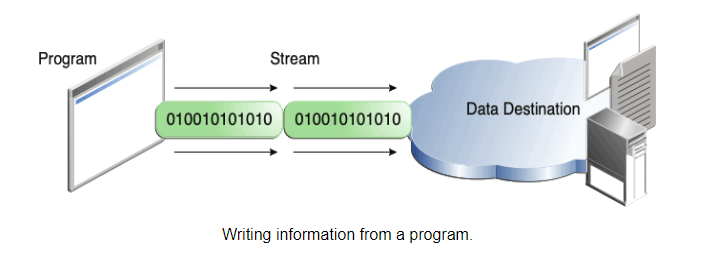
An I/O Stream represents an input source or an output destination. A stream can represent many different kinds of sources and destinations, including disk files, devices, other programs, and memory arrays.

Streams support many different kinds of data, including simple bytes, primitive data types, localized characters, and objects. Some streams simply pass on data; others manipulate and transform the data in useful ways.

No matter how they work internally, all streams present the same simple model to programs that use them: A stream is a sequence of data. A program uses an input stream to read data from a source, one item at a time:



A program uses an output stream to write data to a destination, one item at time:



In this lesson, we'll see streams that can handle all kinds of data, from primitive values to advanced objects.

The data source and data destination pictured above can be anything that holds, generates, or consumes data. Obviously this includes disk files, but a source or destination can also be another program, a peripheral device, a network socket, or an array.

In the next section, we'll use the most basic kind of streams, byte streams, to demonstrate the common operations of Stream I/O. For sample input, we'll use the example file xanadu.txt, which contains the following verse:

In Xanadu did Kubla Khan

A stately pleasure-dome decree:

Where Alph, the sacred river, ran

Through caverns measureless to man

Down to a sunless sea.

**Byte Streams**

Programs use byte streams to perform input and output of 8-bit bytes. All byte stream classes are descended from *InputStream* and *OutputStream*.

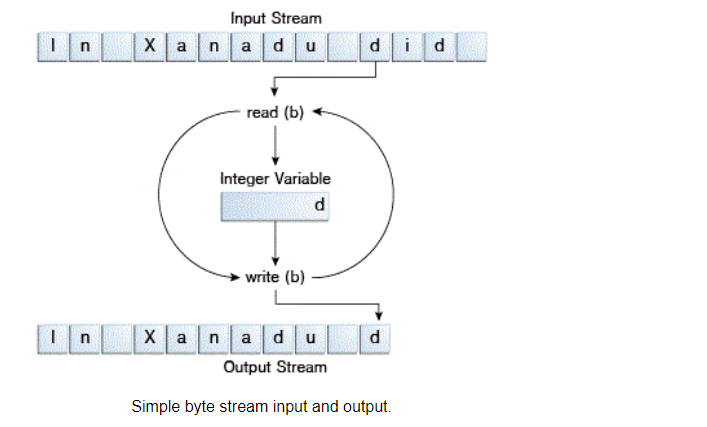
There are many byte stream classes. To demonstrate how byte streams work, we'll focus on the file I/O byte streams, FileInputStream and FileOutputStream. Other kinds of byte streams are used in much the same way; they differ mainly in the way they are constructed.

**Using Byte Streams**

We'll explore FileInputStream and FileOutputStream by examining an example program named CopyBytes, which uses byte streams to copy xanadu.txt, one byte at a time.

| import java.io.FileInputStream;  import java.io.FileOutputStream;  import java.io.IOException;  public class CopyBytes {  public static void main(String[] args) throws IOException {  FileInputStream in = null;  FileOutputStream out = null;  try {  in = new FileInputStream("xanadu.txt");  out = new FileOutputStream("outagain.txt");  int c;  while ((c = in.read()) != -1) {  out.write(c);  }  } finally {  if (in != null) {  in.close();  }  if (out != null) {  out.close();  }  }  }  } |
| --- |

CopyBytes spends most of its time in a simple loop that reads the input stream and writes the output stream, one byte at a time, as shown in the following figure.



**Always Close Streams**

Closing a stream when it's no longer needed is very important — so important that CopyBytes uses a finally block to guarantee that both streams will be closed even if an error occurs. This practice helps avoid serious resource leaks.

One possible error is that CopyBytes was unable to open one or both files. When that happens, the stream variable corresponding to the file never changes from its initial null value. That's why CopyBytes makes sure that each stream variable contains an object reference before invoking close.

**When Not to Use Byte Streams**

CopyBytes seems like a normal program, but it actually represents a kind of low-level I/O that you should avoid. Since xanadu.txt contains character data, the best approach is to use character streams, as discussed in the next section. There are also streams for more complicated data types. Byte streams should only be used for the most primitive I/O.

So why talk about byte streams? Because all other stream types are built on byte streams.

**Character Streams**

The Java platform stores character values using *Unicode conventions*. Character stream I/O automatically translates this internal format to and from the local character set. In Western locales, the local character set is usually an 8-bit superset of ASCII.

For most applications, I/O with character streams is no more complicated than I/O with byte streams. Input and output done with stream classes automatically translates to and from the local character set. A program that uses character streams in place of byte streams automatically adapts to the local character set and is ready for internationalization — all without extra effort by the programmer.

If internationalization isn't a priority, you can simply use the character stream classes without paying much attention to character set issues. Later, if internationalization becomes a priority, your program can be adapted without extensive recoding.

**Using Character Streams**

All character stream classes are descended from Reader and Writer. As with byte streams, there are character stream classes that specialize in file I/O: FileReader and FileWriter. The CopyCharacters example illustrates these classes.

| import java.io.FileReader;  import java.io.FileWriter;  import java.io.IOException;  public class CopyCharacters {  public static void main(String[] args) throws IOException {  FileReader inputStream = null;  FileWriter outputStream = null;  try {  inputStream = new FileReader("xanadu.txt");  outputStream = new FileWriter("characteroutput.txt");  int c;  while ((c = inputStream.read()) != -1) {  outputStream.write(c);  }  } finally {  if (inputStream != null) {  inputStream.close();  }  if (outputStream != null) {  outputStream.close();  }  }  }  } |
| --- |

CopyCharacters is very similar to CopyBytes. The most important difference is that CopyCharacters uses *FileReader and FileWriter* for input and output in place of FileInputStream and FileOutputStream. Notice that both CopyBytes and CopyCharacters use an int variable to read to and write from. However, in CopyCharacters, the int variable holds a character value in its last 16 bits; in CopyBytes, the int variable holds a byte value in its last 8 bits.

**Character Streams that Use Byte Streams**

Character streams are often "wrappers" for byte streams. The character stream uses the byte stream to perform the physical I/O, while the character stream handles translation between characters and bytes. FileReader, for example, uses FileInputStream, while FileWriter uses FileOutputStream.

There are two general-purpose byte-to-character "bridge" streams: *InputStreamReader and OutputStreamWriter*. Use them to create character streams when there are no prepackaged character stream classes that meet your needs.

**Line-Oriented I/O**

Character I/O usually occurs in bigger units than single characters. One common unit is the line: a string of characters with a line terminator at the end. A line terminator can be a carriage-return/line-feed sequence ("\r\n"), a single carriage-return ("\r"), or a single line-feed ("\n"). Supporting all possible line terminators allows programs to read text files created on any of the widely used operating systems.

Let's modify the CopyCharacters example to use line-oriented I/O. To do this, we have to use two classes we haven't seen before, BufferedReader and PrintWriter. We'll explore these classes in greater depth in Buffered I/O and Formatting. Right now, we're just interested in their support for line-oriented I/O.

The CopyLines example invokes BufferedReader.readLine and PrintWriter.println to do input and output one line at a time.

| import java.io.FileReader;  import java.io.FileWriter;  import java.io.BufferedReader;  import java.io.PrintWriter;  import java.io.IOException;  public class CopyLines {  public static void main(String[] args) throws IOException {  BufferedReader inputStream = null;  PrintWriter outputStream = null;  try {  inputStream = new BufferedReader(new FileReader("xanadu.txt"));  outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));  String l;  while ((l = inputStream.readLine()) != null) {  outputStream.println(l);  }  } finally {  if (inputStream != null) {  inputStream.close();  }  if (outputStream != null) {  outputStream.close();  }  }  }  } |
| --- |

Invoking readLine returns a line of text with the line. CopyLines outputs each line using println, which appends the line terminator for the current operating system. This might not be the same line terminator that was used in the input file.

**Buffered Streams**

Most of the examples we've seen so far use unbuffered I/O. This means each read or write request is handled directly by the underlying OS. This can make a program much less efficient, since each such request often triggers disk access, network activity, or some other operation that is relatively expensive.

To reduce this kind of overhead, the Java platform implements buffered I/O streams. Buffered input streams read data from a memory area known as a buffer; the native input API is called only when the buffer is empty. Similarly, buffered output streams write data to a buffer, and the native output API is called only when the buffer is full.

A program can convert an unbuffered stream into a buffered stream using the wrapping idiom we've used several times now, where the unbuffered stream object is passed to the constructor for a buffered stream class. Here's how you might modify the constructor invocations in the CopyCharacters example to use buffered I/O:

| inputStream = new BufferedReader(new FileReader("xanadu.txt"));  outputStream = new BufferedWriter(new FileWriter("characteroutput.txt")); |
| --- |

There are four buffered stream classes used to wrap unbuffered streams: *BufferedInputStream* and *BufferedOutputStream* create buffered byte streams, while *BufferedReader* and *BufferedWriter* create buffered character streams.

**Flushing Buffered Streams**

It often makes sense to write out a buffer at critical points, without waiting for it to fill. This is known as flushing the buffer.

Some buffered output classes support autoflush, specified by an optional constructor argument. When autoflush is enabled, certain key events cause the buffer to be flushed. For example, an autoflush PrintWriter object flushes the buffer on every invocation of println or format. See Formatting for more on these methods.

To flush a stream manually, invoke its flush method. The flush method is valid on any output stream, but has no effect unless the stream is buffered.

**Scanning and Formatting**

Programming I/O often involves translating to and from the neatly formatted data humans like to work with. To assist you with these chores, the Java platform provides two APIs. The scanner API breaks input into individual tokens associated with bits of data. The formatting API assembles data into nicely formatted, human-readable form.

**Scanning**

Objects of type Scanner are useful for breaking down formatted input into tokens and translating individual tokens according to their data type.

**Breaking inputs into tokens**

By default, a scanner uses white space to separate tokens. (White space characters include blanks, tabs, and line terminators.To see how scanning works, let's look at ScanXan, a program that reads the individual words in xanadu.txt and prints them out, one per line.

| import java.io.\*;  import java.util.Scanner;  public class ScanXan {  public static void main(String[] args) throws IOException {  Scanner s = null;  try {  s = new Scanner(new BufferedReader(new FileReader("xanadu.txt")));  while (s.hasNext()) {  System.out.println(s.next());  }  } finally {  if (s != null) {  s.close();  }  }  }  } |
| --- |

Notice that ScanXan invokes Scanner's close method when it is done with the scanner object. Even though a scanner is not a stream, you need to close it to indicate that you're done with its underlying stream.

The output of ScanXan looks like this:

| In  Xanadu  did  Kubla  Khan  A  stately  pleasure-dome  ... |
| --- |

**Formatting**

Stream objects that implement formatting are instances of either PrintWriter, a character stream class, or PrintStream, a byte stream class.

| Note: The only PrintStream objects you are likely to need are System.out and System.err. (See I/O from the Command Line for more on these objects.) When you need to create a formatted output stream, instantiate PrintWriter, not PrintStream. |
| --- |

Like all byte and character stream objects, instances of PrintStream and PrintWriter implement a standard set of write methods for simple byte and character output. In addition, both PrintStream and PrintWriter implement the same set of methods for converting internal data into formatted output. Two levels of formatting are provided:

* print and println format individual values in a standard way.
* format formats almost any number of values based on a format string, with many options for precise formatting.

**The print and println Methods**

Invoking print or println outputs a single value after converting the value using the appropriate toString method. We can see this in the Root example:

| public class Root {  public static void main(String[] args) {  int i = 2;  double r = Math.sqrt(i);    System.out.print("The square root of ");  System.out.print(i);  System.out.print(" is ");  System.out.print(r);  System.out.println(".");  i = 5;  r = Math.sqrt(i);  System.out.println("The square root of " + i + " is " + r + ".");  }  } |
| --- |

Here is the output of Root:

The square root of 2 is 1.4142135623730951.

The square root of 5 is 2.23606797749979.

The i and r variables are formatted twice: the first time using code in an overload of print, the second time by conversion code automatically generated by the Java compiler, which also utilizes toString. You can format any value this way, but you don't have much control over the results.

**The format Method**

The format method formats multiple arguments based on a format string. The format string consists of static text embedded with format specifiers; except for the format specifiers, the format string is output unchanged.

Format strings support many features. In this tutorial, we'll just cover some basics. For a complete description, see format string syntax in the API specification.

The Root2 example formats two values with a single format invocation:

| public class Root2 {  public static void main(String[] args) {  int i = 2;  double r = Math.sqrt(i);    System.out.format("The square root of %d is %f.%n", i, r);  }  } |
| --- |

Here is the output:

The square root of 2 is 1.414214.

Like the three used in this example, all format specifiers begin with a % and end with a 1- or 2-character conversion that specifies the kind of formatted output being generated. The three conversions used here are:

* d formats an integer value as a decimal value.
* f formats a floating point value as a decimal value.
* n outputs a platform-specific line terminator.

Here are some other conversions:

* x formats an integer as a hexadecimal value.
* s formats any value as a string.
* tB formats an integer as a locale-specific month name.

There are many other conversions. [Refer : https://docs.oracle.com/javase/tutorial/essential/io/formatting.html]

**I/O from Command Line**

A program is often run from the command line and interacts with the user in the command line environment. The Java platform supports this kind of interaction in two ways: through the Standard Streams and through the Console.

**Standard Streams**

Standard Streams are a feature of many operating systems. By default, they read input from the keyboard and write output to the display. They also support I/O on files and between programs, but that feature is controlled by the command line interpreter, not the program.

The Java platform supports three Standard Streams: Standard Input, accessed through System.in; Standard Output, accessed through System.out; and Standard Error, accessed through System.err. These objects are defined automatically and do not need to be opened. Standard Output and Standard Error are both for output; having error output separately allows the user to divert regular output to a file and still be able to read error messages. For more information, refer to the documentation for your command line interpreter.

You might expect the Standard Streams to be character streams, but, for historical reasons, they are byte streams. System.out and System.err are defined as PrintStream objects. Although it is technically a byte stream, PrintStream utilizes an internal character stream object to emulate many of the features of character streams.

By contrast, System.in is a byte stream with no character stream features. To use Standard Input as a character stream, wrap System.in in InputStreamReader.

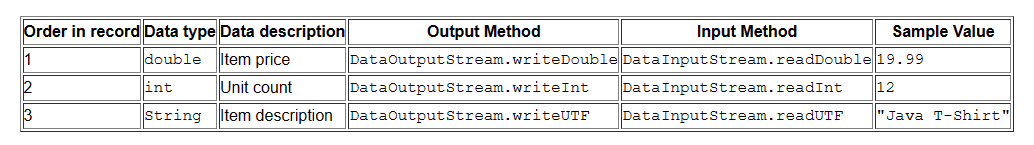
| InputStreamReader cin = new InputStreamReader(System.in); |
| --- |

**The Console** [You can read at : <https://docs.oracle.com/javase/tutorial/essential/io/cl.html>]

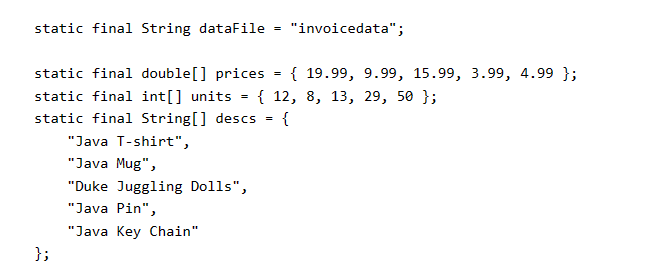
**Data Streams**

Data streams support binary I/O of primitive data type values (boolean, char, byte, short, int, long, float, and double) as well as String values. All data streams implement either the ***DataInput*** interface or the ***DataOutput*** interface. This section focuses on the most widely-used implementations of these interfaces, DataInputStream and DataOutputStream.

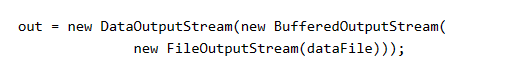
The DataStreams example demonstrates data streams by writing out a set of data records, and then reading them in again. Each record consists of three values related to an item on an invoice, as shown in the following table:



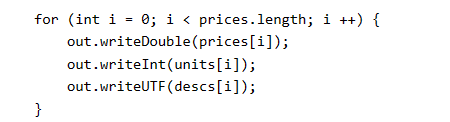
Lets examine crucial code in DataStreams. First the program defines some constants the name of the data file and the data that will be written to it:



The DataStreams opens an output stream. Since a DataOutputStream can only be created as a wrapper for an existing byte stream object, DataStreams provides a buffered file output byte stream.

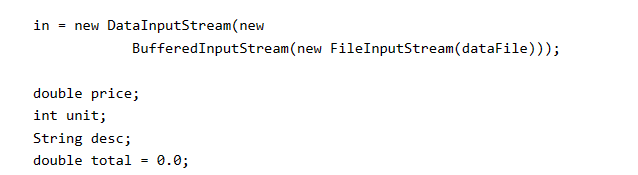


DataStreams writes out the records and closes the output stream

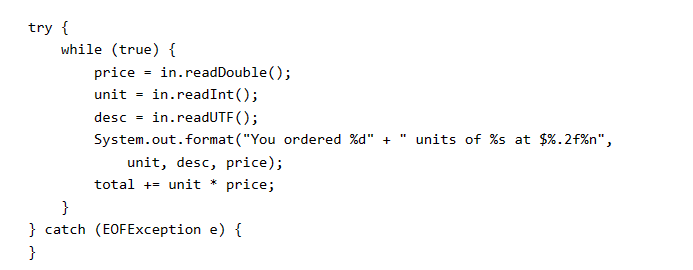


The writeUTF() method writes out String values in a modified form of UTF-8. This is a variable-width character encoding that only needs a single byte for common Western characters.

Now, DataStreams reads the data back again. First, it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream muse constructed as a wrapper for a byte stream



Now DataStreams can read each record in the stream, reporting on the data it encounters:



Notice that DataStreams detects the end-of-file condition by catching EOFException, instead of testing for an invalid return value. All implementations of DataInput methods use EOFException instead of return values.

Also notice that each specialized write in DataStreams is exactly matched by the corresponding specialized read. It is up to the programmer to make sure that output types and input types are matched in this way: The input stream consists of simple binary data, with nothing to indicate the type of individual values, or where they begin in the stream.

DataStreams uses one very bad programming technique: it uses floating point numbers to represent monetary values. In general, floating point is bad for precise values. It's particularly bad for decimal fractions, because common values (such as 0.1) do not have a binary representation.

The correct type to use for currency values is **java.math.BigDecimal.** Unfortunately, BigDecimal is an object type, so it won't work with data streams. However, BigDecimal will work with object streams, which are covered in the next section.

**Object Streams**

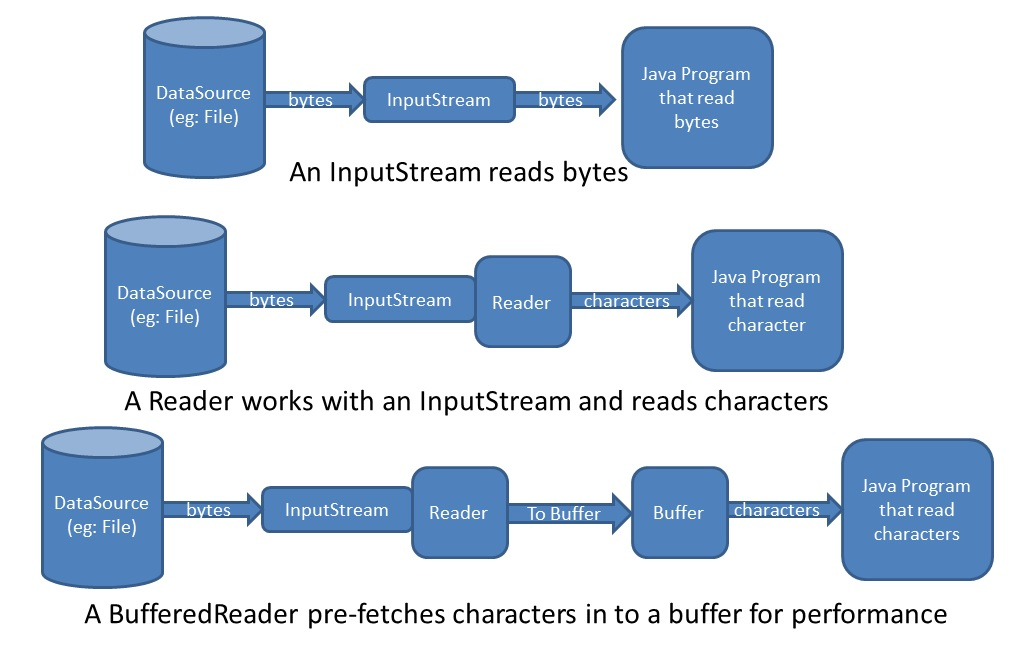
Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface Serializable.

The object stream classes are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are subinterface of DataInput and DataOutput. That means, that all the primitive data I/O methods covered in DataStreams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values.

| /\*  \* Copyright (c) 1995, 2008, Oracle and/or its affiliates. All rights reserved.  \*  \* Redistribution and use in source and binary forms, with or without  \* modification, are permitted provided that the following conditions  \* are met:  \*  \* - Redistributions of source code must retain the above copyright  \* notice, this list of conditions and the following disclaimer.  \*  \* - Redistributions in binary form must reproduce the above copyright  \* notice, this list of conditions and the following disclaimer in the  \* documentation and/or other materials provided with the distribution.  \*  \* - Neither the name of Oracle or the names of its  \* contributors may be used to endorse or promote products derived  \* from this software without specific prior written permission.  \*  \* THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS  \* IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,  \* THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR  \* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR  \* CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,  \* EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,  \* PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR  \* PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF  \* LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING  \* NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS  \* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.  \*/  import java.io.\*;  import java.math.BigDecimal;  import java.util.Calendar;  public class ObjectStreams {  static final String dataFile = "invoicedata";  static final BigDecimal[] prices = {  new BigDecimal("19.99"),  new BigDecimal("9.99"),  new BigDecimal("15.99"),  new BigDecimal("3.99"),  new BigDecimal("4.99") };  static final int[] units = { 12, 8, 13, 29, 50 };  static final String[] descs = { "Java T-shirt",  "Java Mug",  "Duke Juggling Dolls",  "Java Pin",  "Java Key Chain" };  public static void main(String[] args)  throws IOException, ClassNotFoundException {    ObjectOutputStream out = null;  try {  out = new ObjectOutputStream(new  BufferedOutputStream(new FileOutputStream(dataFile)));  out.writeObject(Calendar.getInstance());  for (int i = 0; i < prices.length; i ++) {  out.writeObject(prices[i]);  out.writeInt(units[i]);  out.writeUTF(descs[i]);  }  } finally {  out.close();  }  ObjectInputStream in = null;  try {  in = new ObjectInputStream(new  BufferedInputStream(new FileInputStream(dataFile)));  Calendar date = null;  BigDecimal price;  int unit;  String desc;  BigDecimal total = new BigDecimal(0);  date = (Calendar) in.readObject();  System.out.format ("On %tA, %<tB %<te, %<tY:%n", date);  try {  while (true) {  price = (BigDecimal) in.readObject();  unit = in.readInt();  desc = in.readUTF();  System.out.format("You ordered %d units of %s at $%.2f%n",  unit, desc, price);  total = total.add(price.multiply(new BigDecimal(unit)));  }  } catch (EOFException e) {}  System.out.format("For a TOTAL of: $%.2f%n", total);  } finally {  in.close();  }  }  } |
| --- |

Read: [Object Streams (The Java™ Tutorials > Essential Java Classes > Basic I/O) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/io/objectstreams.html)

Note:



*Streams, Readers, Writers, BufferedReader, BufferedWriter* - these are the terminologies you will deal with in Java. These are the classes provided in Java to operate with input and output. It is really worth knowing how these are related and how it is used.

***Streams*** - used to deal with byte level data.

***Reader/Writer*** - Use to deal with character level. It supports various character encoding also. (FileReader and FileWriter)

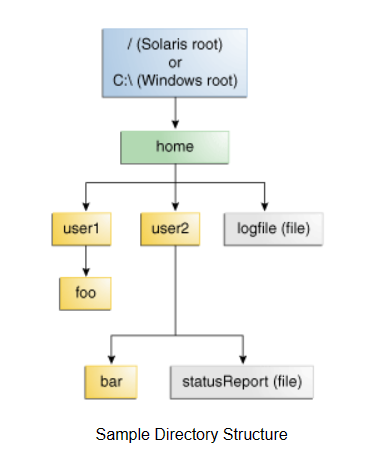
***BufferedReader/BufferedWriter*** - To increase performance. Data to be read will be buffered in the memory for quick access.

**File I/O**

A file system stores and organizes files on some form of media, generally one or more hard drives, in such a way that they can be easily retrieved. Most of file systems in use today store the files in a tree (or hierarchical) structure. At the top of the tree is one (or more) root nodes.Under the root node, there are files and directories (folders). Each directory can contain files and subdirectories, which in turn can contain files and subdirectories, and so on, potentially to an almost limitless depth.

What is a Path?

The following figure shows a sample directory tree containing a single root node. Microsoft Windows support multiple root nodes. Each root node maps to a volume, such as C:\ or D:\. The Solaris OS supports a single root node, which is denoted by the slash character , /.



A file is identified by its path through the file system, beginning from the root node. For example, the statusReport file, in the above figure is described by the following notation in the Solaris OS:

/home/sally/statusReport

In Microsoft Windows, statusReport is described by the following notation:

C:\home\sally\statusReport

The character used to separate the directory names (also called as delimiter) is specific to the file system. The Solaris OS uses the forward slash (/), and Microsoft Windows uses the back slash (\).

**Relative or Absolute?**

A path is either relative or absolute. An ***absolute path*** always contains the root element and the complete directory list required to locate the file. For example, /home/sally/statusReport is an absolute path. All of the information needed to locate the file is contained in the path string.

A **relative path** needs to be combined with another path in order to access a file. For example, joe/foo is a relative path. Without more information, a program cannot reliably locate the joe/foo directory in the file system.

**The Path class**

The Path class is a programmatic representation of a path in the file system. A Path object contains the file name and directory list used to construct the path, and is used to examine, locate and manipulate files.

**Path Operations**

The Path class includes various methods that can be used to obtain information about the path, access elements of the path, convert the path to other forms, or extract portions of a path. There are also methods for matching the path string and methods for removing redundancies in a path. This lesson addresses these Path methods, sometimes called syntactic operations, because they operate on the path itself and don't access the file system.

Topics

* Creating a Path
* Retrieving Information about a Path
* Removing Redundancies from a Path
* Converting a Path
* Joining two Paths
* Creating a Path between two paths
* Comparing two Paths

Read: [Path Operations (The Java™ Tutorials > Essential Java Classes > Basic I/O) (oracle.com)](https://docs.oracle.com/javase/tutorial/essential/io/pathOps.html)

File Operations

Checking a File or Directory

Deleting a File or Directory

Copying a File or Directory

Moving a File or Directory

Reading, Writing and Creating Files

Random Access Files