## Input & Output (I/O) java.io Package

Programs read inputs from data sources (e.g., keyboard, file, network, memory buffer, or another program) and write outputs to data sinks (e.g., display console, file, network, memory buffer, or another program).

In Java standard I/O, inputs and outputs are handled by the so-called **streams**. A stream is a sequential and contiguous one-way flow of data (just like water or oil flows through the pipe). It is important to mention that Java does not differentiate between the various types of data sources or sinks (e.g., file or network) in stream I/O. They are all treated as a sequential flow of data.

Input and output streams can be established from/to any data source/sink, such as files, network, keyboard/console or another program. The Java program receives data from a source by opening an input stream, and sends data to a sink by opening an output stream.

**Stream I/O operations involve three steps:**

1. Open an input/output stream associated with a physical device (e.g., file, network, console/keyboard), by constructing an appropriate I/O stream instance.
2. Read from the opened input stream until "end-of-stream" encountered, or *write* to the opened output stream (and optionally flush the buffered output).
3. Close the input/output stream.

As a consequence, Java needs to differentiate between byte-based I/O for processing raw bytes or binarydata, and character-based I/O for processing texts made up of characters.

A diagram of a program

Description automatically generated

A Stream presents a uniform, easy-to-use, object oriented interface between the program and input/output devices. A Stream in java is a path along which data flows. It has a source (of data) and a destination (for that data). Both the source and destination may a physical devices or programs or other streams in the same program

The term stream refers to a sequence of values from any input source of data (keyboard, file, port, and so on), or to any output destination for data (screen, file, port, and so on). The actual physical source or destination must be supplied as an argument to an appropriate constructor of an appropriate class, during the construction of the actual object that will do the physical reading or writing of data.

A diagram of a computer program

Description automatically generated

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 diagram of a stream

Description automatically generated

Reading information into a program.

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

A diagram of a stream

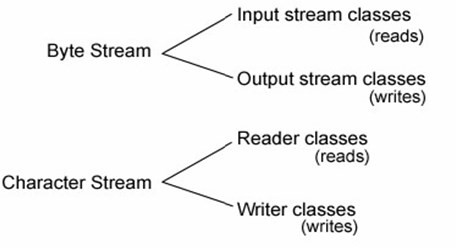
Description automatically generated

Writing information from a program.

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.

## I/O Streams

The classes in java.io package may be categorized into two groups based on the data type on which they operate



* Byte Stream classes – provides support for handling I/O operations on bytes
* Character stream Class - provides support for managing I/O operations on characters

These two groups may be further classified bases based on their purpose.

Byte Streams and Character Stream classes contains specialized classes to deal with input and output operations on various types of devices

## Keyboard Input Using DataInputStream

DataInputStream dis = new DataInputStream(System.in);

What is "System.in"?

" **in**" is an object of " **InputStream**" class defined in System class (like "out" is an object of PrintStream class defined in System class). It is declared as static and final object. The **in** object is connected implicitly to the " **standard input stream**" of the underlying OS

|  |
| --- |
| **Keyboard Input using DataInputStream** |
| import java.io.\*;      public class JavaIODemo  {  public static void main(String[] args)throws IOException  {  //Scanner scan = new Scanner(System.in);  DataInputStream dis = new DataInputStream(System.in);  String str;  System.out.print("\n\t Enter a string :");  str = dis.readLine();    int age;  System.out.println("\n\t Enter age :"); age = Integer.parseInt(dis.readLine());    System.out.println("\n\t Str = " + str);  System.out.println("\n\t Age = " + age);  }  } |

Program N0.

137

The **readLine()** method of **DataInputStream** reads a line at a time and returns as a string, irrespective of what the line contains. Depending on the input value, the string is to be parsed into an **int** or **double** etc.

## Keyboard input using InputStreamReader

By wrapping the **System.in** (standard input stream) in an **InputStreamReader** which is wrapped in a BufferedReader, we can read input from the user in the command line. Here’s an example:

**InputStreamReader** class can be used to read data from keyboard.It performs two tasks:

1. connects to input stream of keyboard
2. converts the byte-oriented stream into character-oriented stream

**BufferedReader** class can be used to read data line by line by readLine() method.

InputStreamReader r = new InputStreamReader(System.in)

BufferedReader br = new BufferedReader(r)

A diagram of a system

Description automatically generated

* The **System.in** is a byte stream and cannot be chained to BufferedReader as BufferedReader is a character stream
* **Note :** (this problem, we did not face with DataInputStream as DataInputStream and System.in are both byte streams)
* The byte stream **System.in** should be converted (wrapped) into a character stream and then passed to BufferedReader constructor. This is done by **InputStreamReader**.

|  |
| --- |
| **Keyboard Input using InputStreamReader** |
| import java.io.\*;      public class JavaIODemo1\_138  {  public static void main(String[] args)throws IOException  {  InputStreamReader in = new InputStreamReader(System.in); BufferedReader br = new BufferedReader(in);    String str;  System.out.print("\n\t Enter a string :");  str = br.readLine();    int age;  System.out.println("\n\t Enter age :"); age = Integer.parseInt(br.readLine());    System.out.println("\n\t Str = " + str);  System.out.println("\n\t Age = " + age);  }  } |

Program N0.

138

* InputStreamReader, is neither an input stream nor a reader. It is not a carrier of data ▪ It is simply a wrapper around input stream to give a reader functionality.
* The InputStreamReader is used to link an input stream with character stream on reading-side

**Advantages:** The input is buffered for efficient reading.

**Drawbacks:** The wrapping code is hard to remember.

## Keyboard Input Using Scanner Class

The main purpose of the **Scanner** class (available since Java 1.5) is to parse primitive types and strings using regular expressions, however it is also can be used to read input from the user in the command line. Here’s an example:

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your nationality: ");

String nationality = scanner.nextLine();

System.out.print("Enter your age: "); int age = scanner.nextInt(); **Advantages:**

* Convenient methods for parsing primitives (nextInt(), nextFloat(), …) from the tokenized input.
* Regular expressions can be used to find tokens.

**Drawbacks:**

• The reading methods are not synchronized.

## Keyboard input using Console class

The Java Console class is used to get input from console. It provides methods to read text and password. If you read password using Console class, it will not be displayed to the user. System class provides a static method console() that returns the unique instance of Console class.

The java.io.Console class is attached with system console internally. The Console class is introduced since 1.5.

### Methods of Console class

|  |  |
| --- | --- |
| **Method** | **Description** |
| **public String readLine()** | It is used to read a single line of text from the console. |
| **public String readLine(String fmt,Object... args)** | It provides a formatted prompt then reads the single line of text from the console. |
| **public char[] readPassword()** | It is used to read password that is not being displayed on the console. |
| **public char[] readPassword(String fmt,Object... args)** | it provides a formatted prompt then reads the password that is not being displayed on the console |

**Advantages:**

* Reading password without echoing the entered characters.
* Reading methods are synchronized.
* Format string syntax can be used.

**Drawbacks:**

* Does not work in non-interactive environment (such as in an IDE).

|  |
| --- |
| **Keyboard Input using Console class** |
| import java.io.\*;    public class ReadStringTest  {  public static void main(String args[])  {  Console c=System.console();    System.out.print("Enter your name: ");  String n=c.readLine();    System.out.print("Enter password: "); char[] ch=c.readPassword();  String pass=String.valueOf(ch);//converting char array into string    System.out.println("Welcome "+n);  System.out.println("Password is: "+pass);  }  } |

Program N0.

138

java.io.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

### Methods of PrintStream class

* public void print(DataType value): it prints the specified value.
* public void println(DataType value): it prints the specified value and terminates the line

|  |
| --- |
| **Write Ouput on Console Screen using PrintStream** |
| import java.io.\*;    public class PrintStreamDemo  {  public static void main(String args[])  {  PrintStream pout = new PrintStream(System.out);  pout.println("Helloworld");  pout.print("This is PrintStream Example");    }  } |

Program N0.

138

## File and Directory

### Class java.io.File

The class java.io.File can represents either a **file** or a **directory**.

A **path string** is used to locate a **file** or a **directory**. Unfortunately, path strings are system dependent, e.g., "c:\myproject\java\Hello.java" in Windows or "/myproject/java/Hello.java" in Unix/Mac.

* Windows use back-slash '\' as the directory separator; while Unixes/Mac use forward-slash '/'.
* Windows use semi-colon ';' as path separator to separate a list of paths; while Unixes/Mac use colon ':'.
* Windows use "\r\n" as line delimiter for text file; while Unixes use "\n" and Mac uses "\r".
* The "c:\" or "\" is called the *root*. Windows supports multiple roots, each maps to a drive (e.g., "c:\", "d:\"). Unixes/Mac has a single root ("\")

A path could be *absolute* (beginning from the root) or *relative* (which is relative to a reference directory). Special notations "." and ".." denote the current directory and the parent directory, respectively.

The java.io.File class maintains these system-dependent properties, for you to write programs that are portable:

* **Directory Separator**: in static fields File.separator (as String) and File.separatorChar. [They failed to follow the Java naming convention for constants adopted since JDK 1.2.] As mentioned, Windows use backslash '\'; while Unixes/Mac use forward slash '/'.
* **Path Separator:** in static fields File.pathSeparator (as String) and File.pathSeparatorChar. As mentioned, Windows use semi-colon ';' to separate a list of paths; while Unixes/Mac use colon ':'.

You can construct a File instance with a path string or URI, as follows. Take note that the physical file/directory may or may not exist. A file URL takes the form of file://..., e.g., file:///d:/docs/programming/java/test.html.

public File(String pathString)

public File(String parent, String child) public File(File parent, String child)

**// Constructs a File instance based on the given path string.**

public File(URI uri)

**// Constructs a File instance by converting from the given file-URI "file://...."**

For examples,

File file = new File("in.txt"); **// A file relative to the current working directory**

File file = new File("d:\\myproject\\java\\Hello.java"); **// A file with absolute path** File dir = new File("c:\\temp"); **// A directory**

For applications that you intend to distribute as JAR files, you should use the URL class to reference the resources, as it can reference disk files as well as JAR'ed files , for example,

java.net.URL url = this.getClass().getResource("icon.png");

## Methods of java.io.File class

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| public boolean **exists**() | Tests if this file/directory exists. |
| public long **length**() | Returns the length of this file. |
| public boolean **isDirectory**() | Tests if this instance is a directory. |
| public boolean **isFile**() | Tests if this instance is a file. |
| public boolean **canRead**() | Tests if this file is readable. |
| public boolean **canWrite**() | Tests if this file is writable. |
| public boolean **delete**() | Deletes this file/directory. |
| public void **deleteOnExit**() | Deletes this file/directory when the program terminates. |
| public boolean **renameTo**(File *dest*) | Renames this file. |
| public boolean **mkdir**() | Makes (Creates) this directory. |
| public Boolean **createNewFile**() | Crates new file |

## List Directory

For a directory, you can use the following methods to list its contents: public String[] **list**() // List the contents of this directory in a String-array public File[] **listFiles**() // List the contents of this directory in a File-array

### Example: The following program recursively lists the contents of a given directory

import java.io.File;

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

File dir = new File("d:\\myproject\\test"); // Escape sequence needed for '\' listRecursive(dir);

}

public static void listRecursive(File dir) { if (dir.isDirectory()) {

File[] items = dir.listFiles(); for (File item : items) {

System.out.println(item.getAbsoluteFile()); if (item.isDirectory())

listRecursive(item); // Recursive call

}

}

}

}

## List Directory with Filter

You can apply a filter to list() and listFiles(), to list only files that meet a certain criteria.

public String[] **list**(FilenameFilter *filter*) public File[] **listFiles**(FilenameFilter *filter*) public File[] **listFiles**(FileFilter *filter*)

**The interface java.io.FilenameFilter declares one abstract method:**

public boolean **accept**(File *dirName*, String *fileName*)

The list() and listFiles() methods does a *call-back* to accept() for each of the file/sub-directory produced. You can program your filtering criteria in accept(). Those files/sub-directories that result in a false return will be excluded.

Example: The following program lists only files that meet a certain filtering criteria.

**// List files that end with ".java"**

import java.io.File; import java.io.FilenameFilter; public class **ListDirectoryWithFilter** { public static void main(String[] args) {

File dir = new File("."); // current working directory if (dir.isDirectory()) {

// List only files that meet the filtering criteria

// programmed in accept() method of FilenameFilter.

String[] files = dir.list(**new FilenameFilter() { public boolean accept(File dir, String file) {**

**return file.endsWith(".java");**

**}**

**});** // an anonymous inner class as FilenameFilter for (String file : files) {

System.out.println(file);

}

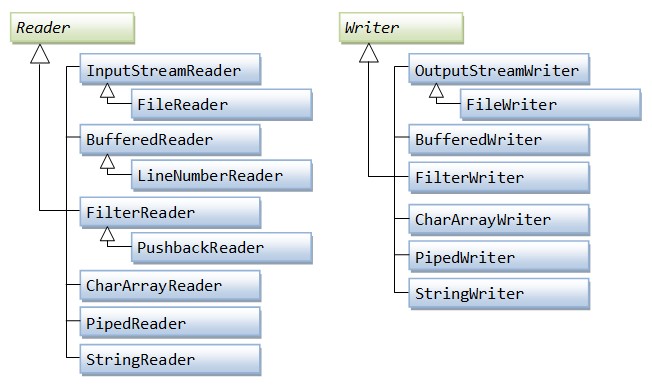
}

}

}

## Character-Based I/O

Some important Charcter stream classes.



|  |  |
| --- | --- |
| **Stream class** | **Description** |
| Reader | Abstract class that define character stream input |
| Writer | Abstract class that define character stream output |
| BufferedReader | Handles buffered input stream. |
| BufferedWriter | Handles buffered output stream. |
| FileReader | Input stream that reads from file. |
| FileWriter | Output stream that writes to file. |
| InputStreamReader | Input stream that translate byte to character |
| OutputStreamWriter | Output stream that translate character to byte. |
| PrintWriter | Output Stream that contain print() and println() method. |

## Abstract superclass Reader and Writer

The abstract superclass Reader operates on char. It declares an abstract method read() to read one character from the input source. read() returns the character as an int between 0 to 65535 (a char in Java can be treated as an unsigned 16-bit integer); or -1 if end-of-stream is detected; or throws an IOException if I/O error occurs. There are also two variations of read() to read a block of characters into char-array public abstract int read() throws IOException

public int read(char[] chars, int offset, int length) throws IOException public int read(char[] chars) throws IOException

**Writer** is the abstract class for writing character streams. It implements the following fundamental methods:

* write(int): writes a single character.
* write(char[]): writes an array of characters.
* write(String): writes a string. ▪ close(): closes the stream.

## OutputStreamWriter, FileWriter, BufferedWriter

OutputStreamWriter is a bridge from byte streams to character streams. Characters are encoded into bytes using a specified charset. The charset can be default character encoding of the operating system, or can be specified explicitly when creating an OutputStreamWriter.

FileWriter is a convenient class for writing text files using the default character encoding of the operating system.

BufferedWriter writes text to a character stream with efficiency (characters, arrays and strings are buffered to avoid frequently writing to the underlying stream) and provides a convenient method for writing a line separator: newLine().

The following diagram show relationship of these writer classes in the java.io package:

A black text on a white background

Description automatically generated

### Character Encoding and Charset

When constructing a reader or writer object, the default character encoding of the operating system is used (e.g.

Cp1252 on Windows):

FileReader reader = new FileReader("MyFile.txt");

FileWriter writer = new FileWriter("YourFile.txt");

So if we want to use a specific charset, use an **InputStreamReader** or **OutputStreamWriter** instead. For example:

InputStreamReader reader = new InputStreamReader(

new FileInputStream("MyFile.txt"), "UTF-16");

That creates a new reader with the Unicode character encoding UTF-16.

And the following statement constructs a writer with the UTF-8 encoding:

OutputStreamWriter writer = new OutputStreamWriter(

new FileOutputStream("YourFile.txt"), "UTF-8");

In case we want to use a **BufferedReader**, just wrap the **InputStreamReader** inside, for example:

InputStreamReader reader = new InputStreamReader( new FileInputStream("MyFile.txt"), "UTF-16"); BufferedReader bufReader = new BufferedReader(reader);

And for a **BufferedWriter** example:

OutputStreamWriter writer = new OutputStreamWriter(

new FileOutputStream("YourFile.txt"), "UTF-8");

BufferedWriter bufWriter = new BufferedWriter(writer);

## Reading from Text File Example

The following small program reads every single character from the file MyFile.txt and prints all the characters to the output console:

import java.io.FileReader; import java.io.IOException;

public class TextFileReadingExample1 { public static void main(String[] args) { try {

FileReader reader = new FileReader("MyFile.txt"); int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

The following example reads a text file with assumption that the encoding is UTF-16:

import java.io.FileInputStream; import java.io.IOException; import java.io.InputStreamReader;

public class TextFileReadingExample2 { public static void main(String[] args) { try {

FileInputStream inputStream = new FileInputStream("MyFile.txt"); InputStreamReader reader = new InputStreamReader(inputStream, "UTF-16"); int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

## Buffered I/O Character-Streams - BufferedReader & BufferedWriter

BufferedReader and BufferedWriter can be stacked on top of FileReader/FileWriter or other character streams to perform buffered I/O, instead of character-by-character. BufferedReader provides a new method readLine(), which reads a line and returns a String

And the following example uses a **BufferedReader** to read a text file line by line (this is the most efficient and preferred way):

import java.io.BufferedReader; import java.io.FileReader; import java.io.IOException;

public class TextFileReadingExample3 { public static void main(String[] args) { try {

FileReader reader = new FileReader("MyFile.txt");

BufferedReader bufferedReader = new BufferedReader(reader);

String line;

while ((line = bufferedReader.readLine()) != null) {

System.out.println(line);

}

reader.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

## Writing to Text File Example

In the following example, a **FileWriter** is used to write two words “Hello World” and “Good Bye!” to a file named MyFile.txt:

import java.io.FileWriter; import java.io.IOException;

public class TextFileWritingExample1 { public static void main(String[] args) { try {

FileWriter writer = new FileWriter("MyFile.txt", true); writer.write("Hello World");

writer.write("\r\n"); // write new line writer.write("Good Bye!"); writer.close(); } catch (IOException e) {

e.printStackTrace();

}

} }

Note that, a writer uses default character encoding of the operating system by default. It also creates a new file if not exits, or overwrites the existing one. If you want to append text to an existing file, pass a boolean flag of true to constructor of the writer class:

FileWriter writer = new FileWriter("MyFile.txt", true);

The following example uses a **BufferedReader** that wraps a **FileReader** to append text to an existing file:

import java.io.BufferedWriter;

import java.io.FileWriter; import java.io.IOException; public class TextFileWritingExample2 { public static void main(String[] args) { try {

FileWriter writer = new FileWriter("MyFile.txt", true);

BufferedWriter bufferedWriter = new BufferedWriter(writer);

bufferedWriter.write("Hello World"); bufferedWriter.newLine();

bufferedWriter.write("See You Again!");

bufferedWriter.close(); } catch (IOException e) {

e.printStackTrace();

}

} }

This is the preferred way to write to text file because the **BufferedReader** provides efficient way for writing character streams.

## InputStreamReader and OutputStreamWriter

As mentioned, Java internally stores characters (char type) in 16-bit UCS-2 character set. But the external data source/sink could store characters in other character set (e.g., US-ASCII, ISO-8859-x, UTF-8, UTF-16, and many others), in fixed length of 8-bit or 16-bit, or in variable length of 1 to 4 bytes. The FileReader/FileWriter introduced earlier uses the default charset for decoding/encoding, resulted in non-portable programs.

InputStreamReader and OutputStreamWriter are considered to be byte-to-character "bridge" streams.

You can choose the character set in the InputStreamReader's constructor: public **InputStreamReader**(InputStream *in*) // Use default charset public **InputStreamReader**(InputStream *in*, String *charsetName*) throws UnsupportedEncodingException public **InputStreamReader**(InputStream *in*, Charset *cs*)

You can list the available charsets via static method java.nio.charset.Charset.availableCharsets(). The commonly-used Charset names supported by Java are:

* "US-ASCII": 7-bit ASCII (aka ISO646-US)
* "ISO-8859-1": Latin-1
* "UTF-8": Most commonly-used encoding scheme for Unicode
* "UTF-16BE": Big-endian (big byte first) (big-endian is usually the default)
* "UTF-16LE": Little-endian (little byte first)
* "UTF-16": with a 2-byte BOM (Byte-Order-Mark) to specify the byte order. FE FF indicates big-endian, FF FE indicates little-endian.

As the InputStreamReader/OutputStreamWriter often needs to read/write in multiple bytes, it is best to wrap it with a BufferedReader/BufferedWriter.

And the following example specifies specific character encoding (UTF-16) when writing to the file:

import java.io.BufferedWriter; import java.io.FileOutputStream; import java.io.IOException; import java.io.OutputStreamWriter; public class TextFileWritingExample3 { public static void main(String[] args) { try {

FileOutputStream outputStream = new FileOutputStream

("MyFile.txt");

OutputStreamWriter outputStreamWriter = new OutputStreamWriter

(outputStream, "UTF-16");

BufferedWriter bufferedWriter = new BufferedWriter

(outputStreamWriter);

bufferedWriter.write("Hello world!"); bufferedWriter.newLine();

bufferedWriter.write("Welcome to my program");

bufferedWriter.close(); } catch (IOException e) {

e.printStackTrace();

}

}

}

This program writes some Unicode string to the specified text file.

**NOTE:** From Java 7, you can use try-with-resources statement to simplify the code of opening and closing the reader/writer. For example:

try (FileReader reader = new FileReader("MyFile.txt")) { int character;

while ((character = reader.read()) != -1) {

System.out.print((char) character);

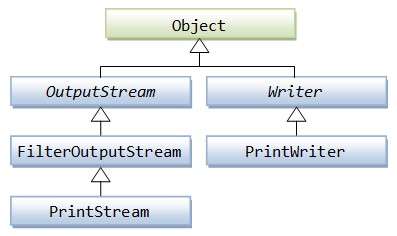
}

} catch (IOException e) {

e.printStackTrace();

}

## java.io.PrintStream & java.io.PrintWriter



The byte-based java.io.printSteam supports convenient printing methods such as print() and println() for printing primitives and text string. Primitives are converted to their string representation for printing. The printf() and format() were introduced in JDK 1.5 for formatting output with former specifiers. printf() and format() are identical.

A PrintStream never throws an IOException. Instead, it sets an internal flag which can be checked via the checkError() method. A PrintStream can also be created to flush the output automatically. That is,

the flush()method is automatically invoked after a byte array is written, one of the println() methods is invoked, or after a newline ('\n') is written.

The standard output and error streams (System.out and System.err) belong to PrintStream.

All characters printed by a PrintStream are converted into bytes using the default character encoding. The PrintWriter class should be used in situations that require writing characters rather than bytes.

The character-stream PrintWriter is similar to PrintStream, except that it write in characters instead of bytes. The PrintWriter also supports all the convenient printing methods print(), println(), printf() and format(). It never throws an IOException and can optionally be created to support automatic flushing

## InputStream and OutputStream

A screenshot of a computer program

Description automatically generated

Some important Byte stream classes.

|  |  |
| --- | --- |
| **Stream class** | **Description** |
| InputStream | Abstract class that describe stream input. |
| OutputStream | Abstract class that describe stream output. |
| PrintStream | Output Stream that contain print() and println() method |
| BufferedInputStream | Used for Buffered Input Stream. |
| BufferedOutputStream | Used for Buffered Output Stream. |
| DataInputStream | Contains method for reading java standard datatype |
| DataOutputStream | An output stream that contain method for writing java standard data type |
| FileInputStream | Input stream that reads from a file |
| FileOutputStream | Output stream that write to a file. |

## InputStream class

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

### Commonly used methods of InputStream class

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

## 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.

### Commonly used methods of OutputStream class

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

### Reading from an InputStream

The abstract superclass InputStream declares an abstract method read() to read one data-byte from the input source:

public abstract int **read**() throws IOException **The read() method:**

▪ returns the input byte read as an int in the range of 0 to 255, or ▪ returns -1 if "end of stream" condition is detected, or ▪ throws an IOException if it encounters an I/O error.

The read() method returns an int instead of a byte, because it uses -1 to indicate end-of-stream.

The read() method blocks until a byte is available, an I/O error occurs, or the "end-of-stream" is detected. The term "block" means that the method (and the program) will be suspended. The program will resume only when the method returns.

Two variations of read() methods are implemented in the InputStream for reading a block of bytes into a byte-array. It returns the number of bytes read, or -1 if "end-of-stream" encounters.

public int **read**(byte[] *bytes*, int *offset*, int *length*) throws IOException

// Read "length" number of bytes, store in bytes array starting from offset of index.

public int **read**(byte[] *bytes*) throws IOException

// Same as read(bytes, 0, bytes.length)

### Writing to an OutputStream

Similar to the input counterpart, the abstract superclass OutputStream declares an abstract method write() to write a data-byte to the output sink. write() takes an int. The least-significant byte of the int argument is written out; the upper 3 bytes are discarded. It throws an IOException if I/O error occurs (e.g., output stream has been closed).

public void abstract void **write**(int *unsignedByte*) throws IOException

Similar to the read(), two variations of the write() method to write a block of bytes from a byte-array are implemented:

public void **write**(byte[] *bytes*, int *offset*, int *length*) throws IOException // Write "*length*" number of bytes, from the *bytes* array starting from offset of *index*.

public void **write**(byte[] *bytes*) throws IOException

// Same as write(*bytes*, 0, *bytes*.length)

### Opening & Closing I/O Streams

You open an I/O stream by constructing an instance of the stream. Both the InputStream and the OutputStream provides a close() method to close the stream, which performs the necessary clean-up operations to free up the system resources.

public void **close**() throws IOException // close this Stream

It is a good practice to explicitly close the I/O stream, by running close() in the finally clause of try-catchfinally to free up the system resources immediately when the stream is no longer needed. This could prevent serious resource leaks. Unfortunately, the close() method also throws aIOException, and needs to be enclosed in a nested try-catch statement, as follows. This makes the codes somehow ugly.

FileInputStream in = null;

...... try {

in = new FileInputStream(...); // Open stream

......

......

} catch (IOException ex) { ex.printStackTrace();

} finally { // always close the I/O streams try {

if (in != null) **in.close()**; } catch (IOException ex) { ex.printStackTrace();

}

}

## FileInputStream & FileOutputStream

FileInputStream and FileOutputStream are concrete implementations to the abstract classes InputStream and OutputStream, to support I/O from disk files.

### FileOutputStream class

Java FileOutputStream is an output stream for writing data to a file.

If you have to write primitive values then use FileOutputStream.Instead, for character-oriented data, prefer FileWriter.But you can write byte-oriented as well as character-oriented dat

|  |
| --- |
| **File OutputStream Demo** |
| import java.io.\*;  public class FileOutputStreamDemo  {  public static void main(String args[])  { try  {  FileOutputStream fout=new FileOutputStream("abc.txt"); String s="Sachin Tendulkar is my favourite player"; byte b[]=s.getBytes();//converting string into byte array fout.write(b); fout.close();  System.out.println("success...");  }  catch(Exception e)  {  System.out.println(e);  }  }  } |

Program N0.

### FileInputStream class

Java FileInputStream class obtains input bytes from a file.It is used for reading streams of raw bytes such as image data. For reading streams of characters, consider using FileReader.

It should be used to read byte-oriented data for example to read image, audio, video etc

|  |
| --- |
| **File InputStream Demo** |
| import java.io.\*; public class SimpleRead  {  public static void main(String args[])  {  try  {  FileInputStream fin=new FileInputStream("abc.txt"); int i=0;  while((i=fin.read())!=-1)  {  System.out.println((char)i);  }  fin.close();  }  catch(IOException e)  {  System.out.println(e);  }  }  } |

Program N0.

## BufferedInputStream & BufferedOutputStream

The I/O streams are often layered or chained with other I/O streams, for purposes such as buffering, filtering, or dataformat conversion (between raw bytes and primitive types). For example, we can layer a BufferedInputStream to a FileInputStream for buffered input,

A diagram of a block of buffering

Description automatically generated

The read()/write() method in InputStream/OutputStream are designed to read/write a single byte of data on each call. This is grossly inefficient, as each call is handled by the underlying operating system (which may trigger a disk access, or other expensive operations). Buffering, which reads/writes a block of bytes from the external device into/from a memory buffer in a single I/O operation, is commonly applied to speed up the I/O.

FileInputStream/FileOutputStream is not buffered. It is often chained to

a BufferedInputStream or BufferedOutputStream, which provides the buffering. To chain the streams together, simply pass an instance of one stream into the constructor of another stream. For example, the following codes chain a FileInputStream to a BufferedInputStream, and finally, a DataInputStream:

FileInputStream fileIn = new FileInputStream("in.dat");

BufferedInputStream bufferIn = new BufferedInputStream(fileIn);

DataInputStream dataIn = new DataInputStream(bufferIn);

// or

DataInputStream in = new DataInputStream( new BufferedInputStream( new FileInputStream("in.dat")));

|  |
| --- |
| **Copying a file byte-by-byte without Buffering.** |
| import java.io.\*;  public class **FileCopyNoBuffer** { // Pre-JDK 7 public static void main(String[] args) {  String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  FileInputStream in = null; FileOutputStream out = null; long startTime, elapsedTime; // for speed benchmarking  // Print file length  **File fileIn = new File(inFileStr);**  System.out.println("File size is " + **fileIn.length()** + " bytes");  try { **in = new FileInputStream(inFileStr)**; |

Program N0.

|  |
| --- |
| **out = new FileOutputStream(outFileStr);**  startTime = System.nanoTime(); int byteRead;  // Read a raw byte, returns an int of 0 to 255.  while ((**byteRead = in.read()**) != -1) {  // Write the least-significant byte of int, drop the upper 3 bytes **out.write(byteRead);**  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is" + (elapsedTime/1000000.0)+  " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the I/O streams try { if (in != null) **in.close()**; if (out != null) **out.close()**; } catch (IOException ex) { ex.printStackTrace();  }  }  }  } |

This example copies a file by reading a byte from the input file and writing it to the output file. It uses FileInputStream and FileOutputStream directly without buffering. Notice that most the I/O methods "throws" IOException, which must be caught or declared to be thrown. The methodclose() is programmed inside the finally clause. It is guaranteed to be run after try or catch. However, method close() also throws an IOException, and therefore must be enclosed inside a nested try-catch block, which makes the codes a little ugly.

We used System.nanoTime(), which was introduced in JDK 1.5, for a more accurate measure of the elapsed time, instead of the legacy not-so-precise System.currentTimeMillis(). The output shows that it took about 4 seconds to copy a 400KB file.

|  |
| --- |
| **Copying a file with a Programmer-Managed Buffer.** |
| import java.io.\*;  public class **FileCopyUserBuffer** { // Pre-JDK 7 public static void main(String[] args) { String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  FileInputStream in = null; FileOutputStream out = null;  long startTime, elapsedTime; // for speed benchmarking  // Check file length  File fileIn = new File(inFileStr);  System.out.println("File size is " + fileIn.length() + " bytes");  try {  in = new FileInputStream(inFileStr); out = new FileOutputStream(outFileStr); startTime = System.nanoTime();  **byte[] byteBuf = new byte[4096];** // 4K byte-buffer int numBytesRead;  while ((numBytesRead = **in.read(byteBuf)**) != -1) { |

Program N0. 138

|  |
| --- |
| **out.write(byteBuf, 0, numBytesRead)**;  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the streams try {  if (in != null) in.close(); if (out != null) out.close();  } catch (IOException ex) { ex.printStackTrace(); }  }  }  } |

This example again uses FileInputStream and FileOutputStream directly. However, instead of reading/writing one byte at a time, it reads/writes a 4KB block. This program took only 3 millisecond - a more than 1000 times speed-up compared with the previous example.

Larger buffer size, up to a certain limit, generally improves the I/O performance. However, there is a trade-off between speed-up the the memory usage. For file copying, a large buffer is certainly recommended. But for reading just a few bytes from a file, large buffer simply wastes the memory.

**re-write the program using JDK 1.7, and try on various buffer size on a much bigger file of 26MB.**

import java.io.\*;

public class **FileCopyUserBufferLoopJDK7** { public static void main(String[] args) { String inFileStr = "test-in.jpg"; String outFileStr = "test-out.jpg"; long startTime, elapsedTime; // for speed benchmarking

// Check file length

File fileIn = new File(inFileStr);

System.out.println("File size is " + fileIn.length() + " bytes");

int[] bufSizeKB = {1, 2, 4, 8, 16, 32, 64, 256, 1024}; // in KB int bufSize; // in bytes

for (int run = 0; run < bufSizeKB.length; ++run) { bufSize = bufSizeKB[run] \* 1024; try (FileInputStream in = new FileInputStream(inFileStr); FileOutputStream out = new FileOutputStream(outFileStr)) { startTime = System.nanoTime(); byte[] byteBuf = new byte[bufSize]; int numBytesRead; while ((numBytesRead = in.read(byteBuf)) != -1) { out.write(byteBuf, 0, numBytesRead);

} elapsedTime = System.nanoTime() - startTime;

System.out.printf("%4dKB: %6.2fmsec%n", bufSizeKB[run], (elapsedTime / 1000000.0));

//System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");

} catch (IOException ex) { ex.printStackTrace();

}

}

}

}

### Copying File with Buffered Streams

|  |
| --- |
| **Copying a file with Buffered Streams.** |
| import java.io.\*;  public class **FileCopyBufferedStream** { // Pre-JDK 7  public static void main(String[] args) { String inFileStr = "test-in.jpg";  String outFileStr = "test-out.jpg";  **BufferedInputStream in** = null; **BufferedOutputStream out** = null;  long startTime, elapsedTime; // for speed benchmarking  // Check file length  File fileIn = new File(inFileStr);  System.out.println("File size is " + fileIn.length() + " bytes");  try {  **in = new BufferedInputStream(new FileInputStream(inFileStr)); out = new BufferedOutputStream(new FileOutputStream(outFileStr));** startTime = System.nanoTime(); int byteRead;  while ((byteRead = **in.read()**) != -1) { // Read byte-by-byte from buffer  **out.write(byteRead)**;  }  elapsedTime = System.nanoTime() - startTime;  System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");  } catch (IOException ex) { ex.printStackTrace();  } finally { // always close the streams try {  if (in != null) in.close(); if (out != null) out.close();  } catch (IOException ex) { ex.printStackTrace(); }  }  }  } |

Program N0.

In this example, the **FileInputStream** with **BufferedInputStream**, **FileOutputStream** with **BufferedOutputStream**, and read/write byte-by-byte. The JRE decides on the buffer size. The program took 62 milliseconds, about 60 times speed-up compared with example 1, but slower than the programmer-managed buffer.

The JDK 1.7 version of the above example is as follows: import java.io.\*;

public class FileCopyBufferedStreamJDK7 { public static void main(String[] args) { String inFileStr = "test-in.jpg"; String outFileStr = "test-out.jpg";

long startTime, elapsedTime; // for speed benchmarking

// Check file length

File fileIn = new File(inFileStr);

System.out.println("File size is " + fileIn.length() + " bytes");

**try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(inFileStr));**

**BufferedOutputStream out = new BufferedOutputStream(new FileOutputStream(outFileStr)))**

{ startTime = System.nanoTime(); int byteRead; while ((byteRead = in.read()) != -1) { out.write(byteRead);

} elapsedTime = System.nanoTime() - startTime;

System.out.println("Elapsed Time is " + (elapsedTime / 1000000.0) + " msec");

} catch (IOException ex) { ex.printStackTrace();

}

}

}

Formatted Data-Streams:

## DataInputStream & DataOutputStream

The DataInputStream and DataOutputStream can be stacked on top of any InputStream and OutputStream to parse the raw bytes to perform I/O operations in the desired data format, such as int and double.

To use DataInputStream for formatted input, you can chain up the input streams as follows:

DataInputStream in = new DataInputStream( new BufferedInputStream( new FileInputStream("in.dat")));

DataInputStream implements DataInput interface, which provides methods to read formatted primitive data and String, such as:

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | | **Description** | |
| public final int **readInt**() throws IOExcpetion; | | Read 4 bytes and convert into int | |
| public final double **readDoube**() throws IOExcpetion; | | Read 8 bytes and convert into double | |
| public final byte **readByte**() throws IOExcpetion; | |  | |
| public final char **readChar**() throws IOExcpetion; | |  | |
| public final short **readShort**() throws IOExcpetion; | |  | |
| public final long **readLong**() throws IOExcpetion; | |  | |
| public final boolean **readBoolean**() throws IOExcpetion; | | Read 1 byte. Convert to false if zero | |
| public final float **readFloat**() throws IOExcpetion; | |  | |
|  | |  | |
| public final int **readUnsignedByte**() throws IOExcpetion; | | Read 1 byte in [0, 255] upcast to int | |
| public final int **readUnsignedShort**() throws IOExcpetion; | | Read 2 bytes in [0, 65535], same as char, upcast to int | |
|  | public final void **readFully**(byte[] *b*, int *off*, int *len*) throws IOException; | |  | |
|  | public final void **readFully**(byte[] *b*) throws IOException; | |  | |
|  |  | |  | |
|  | public final String **readLine**() throws IOException; | | Read a line (until newline), convert each byte into a char - no unicode support | |
|  | public final String **readUTF**() throws IOException; | | read a UTF-encoded string with first two bytes indicating its UTF bytes length | |
|  |  | |  | |
|  | public final int **skipBytes**(int n) | | Skip a number of bytes | |

Similarly, you can stack the DataOutputStream as follows:

DataOutputStream out = new DataOutputStream( new BufferedOutputStream( new FileOutputStream("out.dat")));

A screen shot of a computer

Description automatically generated DataOutputStream implements DataOutput interface, which provides methods to write formatted primitive data and String. For examples,

|  |  |
| --- | --- |
| **Method** | **Description** |
| public final void **writeInt**(int *i*) throws IOExcpetion; | Write the int as 4 bytes |
| public final void **writeFloat**(float *f*) throws IOExcpetion; |  |
| public final void **writeDoube**(double *d*) throws IOExcpetion; | Write the double as 8 bytes |
| public final void **writeByte**(int *b*) throws IOExcpetion; | least-significant byte |
| public final void **writeShort**(int *s*) throws IOExcpetion; | two lower bytes |
| public final void **writeLong**(long *l*) throws IOExcpetion; |  |
| public final void **writeBoolean**(boolean *b*) throws IOExcpetion; |  |
| public final void **writeChar**(int *i*) throws IOExcpetion; |  |
|  |  |
| public final void **writeByte**s(String *str*) throws IOExcpetion; | least-significant byte of each char |
| public final void **writeChars**(String *str*) throws IOExcpetion; | Write String as UCS-2 16-bit char, Big-endian (big byte first) |
| public final void **writeUTF**(String *str*) throws IOException; | Write String as UTF, with first two bytes indicating UTF bytes length |
| public final void **write**(byte[] *b*, int *off*, int *len*) throws IOException  public final void **write**(byte[] *b*) throws IOException |  |
| public final void **write**(int *b*) throws IOException | Write the least-significant byte |

Example: The following program writes some primitives to a disk file. It then reads the raw bytes to check how the primitives were stored. Finally, it reads the data as primitives.

import java.io.\*;

public class TestDataIOStream {

public static void main(String[] args) {

String filename = "data-out.dat";

String message = "Hi,您好!";

// Write primitives to an output file

try (DataOutputStream out = new DataOutputStream( new BufferedOutputStream(

new FileOutputStream(filename)))) { out.writeByte(127); out.writeShort(0xFFFF); // -1 out.writeInt(0xABCD); out.writeLong(0x1234\_5678); // JDK 7 syntax out.writeFloat(11.22f); out.writeDouble(55.66); out.writeBoolean(true); out.writeBoolean(false);

for (int i = 0; i < message.length(); ++i) { out.writeChar(message.charAt(i));

}

out.writeChars(message); out.writeBytes(message); out.flush();

} catch (IOException ex) { ex.printStackTrace();

}

// Read raw bytes and print in Hex try (BufferedInputStream in = new BufferedInputStream( new FileInputStream(filename))) { int inByte;

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

System.out.printf("%02X ", inByte); // Print Hex codes }

System.out.printf("%n%n"); } catch (IOException ex) { ex.printStackTrace();

}

// Read primitives try (DataInputStream in = new DataInputStream( new BufferedInputStream(

new FileInputStream(filename)))) { System.out.println("byte: " + in.readByte());

System.out.println("short: " + in.readShort());

System.out.println("int: " + in.readInt());

System.out.println("long: " + in.readLong());

System.out.println("float: " + in.readFloat());

System.out.println("double: " + in.readDouble());

System.out.println("boolean: " + in.readBoolean()); System.out.println("boolean: " + in.readBoolean()); System.out.print("char: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print(in.readChar());

}

System.out.println();

System.out.print("chars: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print(in.readChar());

}

System.out.println();

System.out.print("bytes: ");

for (int i = 0; i < message.length(); ++i) {

System.out.print((char)in.readByte());

}

System.out.println(); } catch (IOException ex) { ex.printStackTrace();

}

}

}

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](https://docs.oracle.com/javase/8/docs/api/java/io/DataInput.html) interface or the [DataOutput](https://docs.oracle.com/javase/8/docs/api/java/io/DataOutput.html) interface. This section focuses on the most widely-used implementations of these interfaces, [DataInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/DataInputStream.html) and

[DataOutputStream.](https://docs.oracle.com/javase/8/docs/api/java/io/DataOutputStream.html)

The [DataStreams](https://docs.oracle.com/javase/tutorial/essential/io/examples/DataStreams.java) 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:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Order in record** | **Data type** | **Data description** | **Output Method** | **Input Method** | **Sample Value** |
| 1 | Double | Item price | DataOutputStream.writeDouble | DataInputStream.readDouble | 19.99 |
| 2 | Int | Unit count | DataOutputStream.writeInt | DataInputStream.readInt | 12 |
| 3 | String | Item description | DataOutputStream.writeUTF | DataInputStream.readUTF | "Java T-  Shirt" |

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

static final String dataFile = "invoicedata";

static final double[] prices = { 19.99, 9.99, 15.99, 3.99, 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" };

Then 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.

out = new DataOutputStream(new BufferedOutputStream( new FileOutputStream(dataFile)));

DataStreams writes out the records and closes the output stream.

for (int i = 0; i < prices.length; i ++) { out.writeDouble(prices[i]); out.writeInt(units[i]); out.writeUTF(descs[i]);

}

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 in again. First it must provide an input stream, and variables to hold the input data. Like DataOutputStream, DataInputStream must be constructed as a wrapper for a byte stream.

in = new DataInputStream(new

BufferedInputStream(new FileInputStream(dataFile))); double price; int unit; String desc; double total = 0.0;

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

try {

while (true) {

price = in.readDouble(); unit = in.readInt(); desc = in.readUTF();

System.out.format("You ordered %d" + " units of %s at $%.2f%n", unit, desc, price); total += unit \* price;

}

} catch (EOFException e) { }

Notice that DataStreams detects an end-of-file condition by catching [EOFException,](https://docs.oracle.com/javase/8/docs/api/java/io/EOFException.html) 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.

## Object Streams and Object Serialization

Data streams (DataInputStream and DataOutputStream) allow you to read and write primitive data (such as int, double) and String, rather than individual bytes. Just as data streams support I/O of primitive data types, object streams support I/O of objects

The object stream classes are [ObjectInputStream](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInputStream.html) and [ObjectOutputStream.](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutputStream.html) These classes implement [ObjectInput](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectInput.html) and [ObjectOutput,](https://docs.oracle.com/javase/8/docs/api/java/io/ObjectOutput.html) which are subinterfaces of DataInput and DataOutput.

Most of the core Java classes implement Serializable, such as all the wrapper classes, collection classes, and GUI classes. In fact, the only core Java classes that do not implement Serializable are ones that should not be serialized. Arrays of primitives or serializable objects are themselves serializable. transient & static

* static fields are not serialized, as it belongs to the class instead of the particular instance to be serialized.
* To prevent certain fields from being serialized, mark them using the keyword **transient**. This could cut down the amount of data traffic.
* The writeObject() method writes out the class of the object, the class signature, and values of non-

static and non-transient fields

## Object streams

(ObjectInputStream and ObjectOutputStream) go one step further to allow you to read and write entire objects (such as Date, ArrayList or any custom objects).

Object serialization is the process of representing a "particular state of an object" in a serialized bit-stream, so that the bit stream can be written out to an external device (such as a disk file or network). The bit-stream can later be reconstructed to recover the state of that object. Object serialization is necessary to save a state of an object into a disk file for persistence or sent the object across the network for applications such as Web Services, Distributed-object applications, and Remote Method Invocation (RMI).

In Java, object that requires to be serialized must

implement java.io.Serializable or java.io.Externalizable interface. Serializable interface is

an empty interface (or *tagged* interface) with nothing declared. Its purpose is simply to declare that particular object is serializable.

## ObjectInputStream & ObjectOutputStream

We use object streams to read and write Java objects in binary format. ObjectInputStreamand ObjectOutputStream are the main object stream classes provided by the Java File I/O API.

The ObjectOutputStream class implements the ObjectOutput interface that defines a method for writing an object to an output stream:

* writeObject(Object): writes an object to the underlying storage or stream. This method throws IOException if an I/O error occurs.

The process of writing an object to an output stream is called **Serialization**

The ObjectOutput interface extends from the DataOutput interface, which means an ObjectOutputStream inherits all behaviors of writing primitive types and Strings like a DataOutputStream.

Likewise, the ObjectInputStream class implements the ObjectInput interface that defines a method for reading an object from an input stream:

* **readObject():** reads and returns an object. This method throws ClassNotFoundException if the class of the serialized object cannot be found, and throws IOException if an I/O error occurs.

The process of reconstructing an object from an input stream is called **deserialization**.

The ObjectInput interface extends from the DataInput interface, which means an ObjectInputStream also has behaviors of reading primitive types and Strings like a DataInputStream.

Which kinds of object are eligible for serialization?

Note that only objects of classes that implement the [java.io.](http://java.io/)**Serializable** interface can be written to and read from an output/input stream. Serializable is a marker interface, which doesn’t define any methods. Only objects that are marked ‘serializable’ can be used with ObjectOutputStream and ObjectInputStream.

Most classes in Java (including Date and primitive wrappers Integer, Double, Long, etc) implement the Serializable interface. We have to implement this interface for our custom classes only, such as the Student class we see in the previous email about data streams.

If we attempt to write an object of a non-serializable class, we will get a java.io.NotSerializableException.

import java.util.Date; import java.io.Serializable;

public class Student implements Serializable { private String name; private Date birthday;

private boolean gender; // true is male, false is female private int age; private float grade; public Student() {

}

public Student(String name, Date birthday, boolean gender, int age, float grade) {  [this.name](http://this.name/) = name; this.birthday = birthday; this.gender = gender; this.age = age; this.grade = grade;

}

// getters and setters go here... }

As you can see, we make this class implements the Serializable interface and add another member variable call birthday which is of type java.util.Date.

Rewrite the StudentRecordWriter program to use an ObjectOutputStream like this:

import java.util.\*; import java.text.\*; import java.io.\*;

public class StudentRecordWriter { public static void main(String[] args) { if (args.length < 1) {

System.out.println("Please provide output file");

System.exit(0);

}

String outputFile = args[0];

DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy"); try (

ObjectOutputStream objectOutput

= new ObjectOutputStream(new FileOutputStream(outputFile)); ) {

List listStudent = new ArrayList<>();

listStudent.add(new Student("Alice", dateFormat.parse("02-15-1993"), false, 23, 80.5f));

listStudent.add(new Student("Brian", dateFormat.parse("10-03-1994"), true, 22, 95.0f));

listStudent.add(new Student("Carol", dateFormat.parse("08-22-1995"), false, 21, 79.8f)); for (Student student : listStudent) { objectOutput.writeObject(student);

}

} catch (IOException | ParseException ex) { ex.printStackTrace();

}

}

}

Run this program via command line:

java StudentRecordWriter Student.db

And rewrite the StudentRecordReader program like this:

import java.text.\*; import java.io.\*;

public class StudentRecordReader {

public static void main(String[] args) { if (args.length < 1) {

System.out.println("Please provide input file");

System.exit(0);

}

String inputFile = args[0];

DateFormat dateFormat = new SimpleDateFormat("MM-dd-yyyy"); try (

ObjectInputStream objectInput

= new ObjectInputStream(new FileInputStream(inputFile));

){

while (true) {

Student student = (Student) objectInput.readObject();

System.out.print(student.getName() + "\t");

System.out.print(dateFormat.format(student.getBirthday()) + "\t");

System.out.print(student.getGender() + "\t");

System.out.print(student.getAge() + "\t");

System.out.println(student.getGrade());

}

} catch (EOFException eof) {

System.out.println("Reached end of file"); } catch (IOException | ClassNotFoundException ex) { ex.printStackTrace();

}

}

}

Run this program via command line:

java StudentRecordReader Student.db

Output:

Alice 02-15-1993 false 23 80.5

Brian 10-03-1994 true 22 95.0 Carol 08-22-1995 false 21 79.8

Reached end of file

## java.io.Externalizable Interface

The Serializable has a sub-interface called Externalizable, which you could used if you want to customize the way a class is serialized. Since Externalizable extends Serializable, it is also a Serializable and you could invoke readObject() and writeObject().

Externalizable declares two abstract methods:

void **writeExternal**(ObjectOutput *out*) throws IOException

void **readExternal**(ObjectInput *in*) throws IOException, ClassNotFoundException

ObjectOutput and ObjectInput are interfaces that are implemented by ObjectOutputStream and ObjectInputStream, which define

the writeObject() and readObject() methods, respectively. When an instance of Externalizable is passed to an ObjectOutputStream, the default serialization procedure is bypassed; instead, the stream calls the instance's writeExternal() method. Similarly, when an ObjectInputStream reads a Exteranlizabled instance, it uses readExternal() to reconstruct the instance.

Externalizable is useful if you want complete control on how your objects shall be serialized/deserialized. For example, you could encrypt sensitive data before the object is serialized

## Output and Input of Complex Objects

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class like Calendar, which just encapsulates primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

This is demonstrated in the following figure, where writeObject is invoked to write a single object named **a**. This object contains references to objects **b** and **c**, while **b** contains references to **d** and **e**. Invoking writeobject(a) writes not just **a**, but all the objects necessary to reconstitute **a**, so the other four objects in this web are written also. When **a** is read back by readObject, the other four objects are read back as well, and all the original object references are preserved.

A screenshot of a computer

Description automatically generated

I/O of multiple referred-to objects

You might wonder what happens if two objects on the same stream both contain references to a single object. Will they both refer to a single object when they're read back? The answer is "yes." A stream can only contain one copy of an object, though it can contain any number of references to it. Thus if you explicitly write an object to a stream twice, you're really writing only the reference twice. For example, if the following code writes an object ob twice to a stream:

Object ob = new Object(); out.writeObject(ob); out.writeObject(ob);

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

Object ob1 = in.readObject();

Object ob2 = in.readObject();

This results in two variables, ob1 and ob2, that are references to a single object.

## Random Access Files

All the I/O streams covered so far are one-way streams. That is, they are either read-only input stream or write-only output stream. Furthermore, they are all sequential-access (or serial) streams, meant for reading and writing data sequentially. Nonetheless, it is sometimes necessary to read a file record directly as well as modifying existing records or inserting new records. The class RandomAccessFile provides supports for non-sequential, direct (or random) access to a disk file. RandomAccessFile is a two-way stream, supporting both input and output operations in the same stream.

RandomAccessFile can be treated as a huge byte array. You can use a file pointer (of type long), similar to array index, to access individual byte or group of bytes in primitive types (such as int and double). The file pointer is located at 0 when the file is opened. It advances automatically for every read and write operation by the number of bytes processed.

A computer screen shot of a pointer

Description automatically generated

In constructing a RandomAccessFile, you can use flags 'r' or 'rw' to indicate whether the file is "read-only" or "readwrite" access, e.g.,

RandomAccessFile f1 = new **RandomAccessFile**("filename", "r");

RandomAccessFile f2 = new **RandomAccessFile**("filename", "rw");

The following methods are available:

|  |  |
| --- | --- |
| **Method** | **Description** |
| public void seek(long pos) throws IOException | Positions the file pointer for subsequent read/write operation |
| public int skipBytes(int numBytes) throws IOException | Moves the file pointer forward by the specified number of bytes |
| public long getFilePointer() throws IOException | Gets the position of the current file pointer, in bytes, from the beginning of the file. |
| public long length() throws IOException | Returns the length of this file |

RandomAccessFile does not inherit from InputStream or OutputStream. However, it

implements DataInput and DataOutput interfaces (similar to DataInputStream andDataOutputStream). Therefore, you can use various methods to read/write primitive types to the file, e.g.,

public int **readInt**() throws IOException; public double **readDouble**() throws IOException; public void **writeInt**(int i) throws IOException; public void **writeDouble**(double d) throws IOException;

**Example:** Read and write records from a RandomAccessFile. (A student file consists of student record of name (String) and id (int)).