[Skip to content](https://www.amitph.com/java-read-write-large-files-efficiently/#content)

[](https://www.amitph.com/)

* [**Home**](https://www.amitph.com/)
* [**Spring**](https://www.amitph.com/spring/)
* [**Java**](https://www.amitph.com/java/)
* [**Java 8**](https://www.amitph.com/java-8/)
* [**About**](https://www.amitph.com/about/)

[**Home**](https://www.amitph.com/)**»**[**Java**](https://www.amitph.com/java-tutorials/)**» Java Large Files – Efficient Processing**

**Java Large Files – Efficient Processing**

**BY**[**AMIT PHALTANKAR**](https://www.amitph.com/about/)

Guide to **Optimal ways of Java Large Files Processing** to avoid *OutOfMemoryError*. Compare between the **fasted and the most memory efficient ways to read and write files**.

**Tutorial Contents**

* [Overview](https://www.amitph.com/java-read-write-large-files-efficiently/#Overview)
* [Fasted way of Java Large File Processing](https://www.amitph.com/java-read-write-large-files-efficiently/#Fasted_way_of_Java_Large_File_Processing)
* [Using BufferedReader and Java Streams](https://www.amitph.com/java-read-write-large-files-efficiently/#Using_BufferedReader_and_Java_Streams)
* [Java Scanner](https://www.amitph.com/java-read-write-large-files-efficiently/#Java_Scanner)
* [Using FileChannel](https://www.amitph.com/java-read-write-large-files-efficiently/#Using_FileChannel)
* [Process Large File In Chunks (BufferdInputStream)](https://www.amitph.com/java-read-write-large-files-efficiently/#Process_Large_File_In_Chunks_BufferdInputStream)
* [Most Efficient Way of Java Large File Processing](https://www.amitph.com/java-read-write-large-files-efficiently/#Most_Efficient_Way_of_Java_Large_File_Processing)
  + [In Memory Transfer](https://www.amitph.com/java-read-write-large-files-efficiently/#In_Memory_Transfer)
  + [Java Stream and Scanner](https://www.amitph.com/java-read-write-large-files-efficiently/#Java_Stream_and_Scanner)
  + [FileChannel and Chunk Transfer using BufferedInputStream](https://www.amitph.com/java-read-write-large-files-efficiently/#FileChannel_and_Chunk_Transfer_using_BufferedInputStream)
  + [Conclusion](https://www.amitph.com/java-read-write-large-files-efficiently/#Conclusion)
* [Summary](https://www.amitph.com/java-read-write-large-files-efficiently/#Summary)

**Overview**

This tutorial discusses **different ways processing large files in Java** and also How to avoid Java *OutOfMemoryException* while transferring or processing large files. Java File IO and Java NIO provide various ways of dealing with files. However, large file handling is challenging because we need to find a right balance between the speed and memory utilisation.

In this article we will use different ways read a very large file from one place and copy it to another. While doing so, we will monitor the time it takes and the memory it consumes. Finally, we will discuss their performances and find the **most efficient way Java Large File Processing**.

We will write examples to transfer large files by using Java Streams, using Java Scanners, using Java File Channels, and then by using Java BufferedInputStream. However, to begin with we will discuss the fastest way of file transfer.

**Fasted way of Java Large File Processing**

This section covers the **fasted way of reading and writing large files** in java. However, **a faster way doesn’t mean a better way,** and we are going to discuss that soon.

*When we use a Java IO to read a file or to write a file, the slowest part of the process is when the file contents are actually transferred between the hard disk and the JVM memory. Thus, to make File IO faster we can reduce the number of times the data transfer happens. And, the easiest way of doing this is to transfer everything in one go*.

For example, **using Files.readAllBytes()**

**byte**[] bytes = Files.readAllBytes(sourcePath);

Code language: Java (java)

Or, **using Files.readAllLines()**.

List<String> lines = Files.readAllLines(sourcePath);

Code language: Java (java)

In the first snippet, the entire content of the file is copied into a byte array, which is held in memory. Similarly, in the second snippet the entire content of a text file is read as a *List* of string and it is held in memory too.

Next method reads **byte[]** from a source file and write those **bytes[]** on the target file.

**private** **void** **copyByUsingByteArray**() **throws** IOException {

Path sourcePath = Path.of(source);

Path targetPath = Path.of(target);

**byte**[] bytes = Files.readAllBytes(sourcePath);

Files.write(targetPath, bytes, StandardOpenOption.CREATE);

}

Code language: Java (java)

By using this method, we will process a 667 MB File to read it from source and write to the target. In order to observe the memory footprint, we are running this method in a separate thread. Also, while the copy happens in the thread, the parent thread, on fixed intervals prints the amount of free memory (in MB).

Source File Size 667

Memory used: 9

Memory used: 676

Memory used: 676

total time 1803

The transfer finished really fast however it consumed a lot of memory. This solution is impractical when you are copying such large files or processing multiple such files simultaneously.

**Using BufferedReader and Java Streams**

Now, we will test the performance of the Java **Streams to process a very large file**. To do that, we will use *BufferedReader*, which provides a Stream of strings read from the file.

Next is an **example of using Java Stream provided by BufferedReader to process a very very large file (10GB)**.

**private** **void** **copyUsingJavaStreams**() **throws** IOException {

**try** (

InputStream inputStream = **new** FileInputStream(source);

BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(inputStream));

FileWriter fileWriter = **new** FileWriter(target, **true**);

PrintWriter printWriter = **new** PrintWriter(**new** BufferedWriter(fileWriter));

Stream<String> linesStream = bufferedReader.lines();

) {

linesStream

.forEach(printWriter::println);

}

}

Code language: Java (java)

Now, we will test the method that **uses *BufferedReader* to read a 10GB file**.

Source File Size 10471

Memory used: 9

Memory used: 112

Memory used: 71

Memory used: 17

Memory used: 124

Memory used: 76

Memory used: 28

Memory used: 69

Memory used: 35

Memory used: 47

total time 42025

The**Java Streams are lazy and that is why they provide optimal performance**. That means, while each line from the stream is being written to the target, the next ones are efficiently read from the source. This is evident with the memory logs, as we see the highest memory consumption was less than 125MB and the Garbage Collector doing its job in between. Although, it performed better on the memory, but it took around 42 seconds to finish the file processing.

**Java Scanner**

Java Scanner is used to scan through a file, and it supports streaming the content without exhausting large amount of memory.

Next is an example of **using Java Scanner to copy a 10GB file**.

**private** **void** **copyUsingScanner**() **throws** IOException {

**try** (

InputStream inputStream = **new** FileInputStream(source);

Scanner scanner = **new** Scanner(inputStream, StandardCharsets.UTF\_8);

FileWriter fileWriter = **new** FileWriter(target, **true**);

PrintWriter printWriter = **new** PrintWriter(**new** BufferedWriter(fileWriter));

) {

**while** (scanner.hasNext()) {

printWriter.println(scanner.next());

}

}

Code language: Java (java)

**Output:**

Source File Size 10471

Memory used: 9

Memory used: 8

Memory used: 9

Memory used: 110

Memory used: 27

Memory used: 176

Memory used: 44

Memory used: 13

Memory used: 74

Memory used: 17

Memory used: 184

Memory used: 35

total time 660054

Although, scanner has used almost the same amount of memory, the performance is extremely slow. It took around 11 minutes to copy a 10GB file from one location to other.

**Using FileChannel**

Next, we will cover an example of using **Java FileChannels to transfer a very large amount of data from one file to other**.

**private** **void** **copyUsingChannel**() **throws** IOException {

**try** (

FileChannel inputChannel = **new** FileInputStream(source).getChannel();

FileChannel outputChannel = **new** FileOutputStream(target).getChannel();

) {

ByteBuffer buffer = ByteBuffer.allocateDirect(4 \* 1024);

**while** (inputChannel.read(buffer) != -1) {

buffer.flip();

outputChannel.write(buffer);

buffer.clear();

}

}

}

Code language: Java (java)

Here, we are using a buffer of **(4 \* 1024)** size.

Source File Size 10471

Memory used: 9

Memory used: 10

Memory used: 10

Memory used: 10

total time 21403

From the output it is clear that, this is so far the fastest and **most memory efficient way of processing large files**.

**Process Large File In Chunks (BufferdInputStream)**

Finally, we will have a look at the traditional way of processing large amount of data in Java IO. We will use *BufferedInputStream* stream with the same size buffer as we used for *FileChannels*, and analyse the results.

Next is an example of **Reading and Writing Large Files in Chunks using Java BufferedInputStream**.

**private** **void** **copyUsingChunks**() **throws** IOException {

**try** (

InputStream inputStream = **new** FileInputStream(source);

BufferedInputStream bufferedInputStream = **new** BufferedInputStream(inputStream);

OutputStream outputStream = **new** FileOutputStream(target);

) {

**byte**[] buffer = **new** **byte**[4 \* 1024];

**int** read;

**while** ((read = bufferedInputStream.read(buffer, 0, buffer.length)) != -1) {

outputStream.write(buffer, 0, read);

}

}

}

Code language: Java (java)

**Output:**

Source File Size 10471

Memory used: 9

Memory used: 10

Memory used: 10

Memory used: 10

total time 20581

And, the performance we see is similar to the Scanner. Which is because, we used the buffer of same size.

**Most Efficient Way of Java Large File Processing**

We have tried a various ways of reading and writing very large files in Java. In this section we will discuss their performance and understand which one is the **optimal way of large file handling in Java**.

**In Memory Transfer**

As stated earlier, the in memory transfer is a fast way of data transfer. However, holding entire content of a file in memory, for example *byte[]* or *List<String>* is not practical with very large files. It can easily exhaust all available memory when a file is very large, or the application is serving multiple such requests simultaneously.

**Java Stream and Scanner**

In the Java Stream example of processing large files, we generated Stream of lines using *BufferedReader*, which produced a descent result. Similarly, example Java FileScanner to transfer large files turned out better on the memory. However, both of these transfer were really slow.

**FileChannel and Chunk Transfer using BufferedInputStream**

We have also seen examples of using *FileChannel* and *BufferedInputStream* to read and write very large files. At the base of both examples, we used a buffer of a fixed size. Both of these ways demonstrated better performance in terms of speed and low memory consumption.

Moreover, we can still improve the performance of these two ways by using larger buffers. Because, larger buffers means lesser interactions with underlying files. However, larger buffers also means larger consumption of memory. To prove that we will rerun both of these examples with a buffer size of 1048576 (or 1MB).

**BufferedInputStream**

We will modify the buffer size.

**byte**[] buffer = **new** **byte**[1048576];

Code language: Java (java)

And, the output we get:

Source File Size 10471

Memory used: 9

Memory used: 12

Memory used: 12

Memory used: 12

total time 11390

**FileChannel**

Similarly, we will increase the ByteBuffer value in the FileChannel Example.

ByteBuffer buffer = ByteBuffer.allocateDirect(1048576);

Code language: Java (java)

And the result looks like this:

Source File Size 10471

Memory used: 9

Memory used: 10

Memory used: 10

Memory used: 10

total time 11431

From both of the outputs above we can see a performance improvement, with a slightly more impact on the memory.

**Conclusion**

The conclusion of this long practical comparison is that the best way of transferring a very large amount of data using Java IO is by using buffer. Copying the file in chunks helps to limit the amount of consumed memory consumed by the file content.

Both the *FileChannel* and *BufferedInputStream* performed head to head in our tests. The advantage of using *BufferedInputStream* or *FileChannel* to read large files is that they have a configurable buffer. Thus, based on the nature of the server load, and size of the file we can control the buffer size and eventually find an optimal and the most efficient way to read large files in Java IO.

**Summary**

In this long and practical oriented tutorial we discussed **Java Large File Processing**. We began by understanding that we can speed up large file reads at the cost of memory consumption. Or Keep the memory utilisation to minimal by slowing down the processing.

Also, we practically tested these ways, which included using Java *Streams*, Java *Scanner*, Java *FileChannel*, and Java *BufferedInputStream* to transfer a 10GB file and analysed their performance. Finally, we concluded that the ***BufferedInputStream* and the *FileChannel* are the optimal and most efficient ways to read and write very large files in Java IO**. They offer excellent control to optimise the large file handling in Java. For more on Java, please visit: [**Java Tutorials**](https://www.amitph.com/java/).

[Java Core](https://www.amitph.com/tag/java-core/)[Java IO](https://www.amitph.com/tag/java-io/)

[](https://www.amitph.com/)

**TOPICS**

* [Java](https://www.amitph.com/java-tutorials/)
* [Scala](https://www.amitph.com/scala-tutorials/)
* [Spring](https://www.amitph.com/spring-tutorials/)

**RESOURCES**

* [About](https://www.amitph.com/about/)
* [Contact us](https://www.amitph.com/contact/)
* [Java Tutorials](https://www.amitph.com/java/)
* [Spring Tutorials](https://www.amitph.com/spring/)
* [Java Streams and Lambdas](https://www.amitph.com/java-8/)
* [Learning Scala](https://www.amitph.com/scala/)
* [Home](https://www.amitph.com/)
* [MongoDB Fundamentals – Book](https://www.amitph.com/mongodb-fundamentals-book/)

**LATEST**

* [Spring AOP Guide](https://www.amitph.com/spring-aop-guide/)
* [Spring Boot Exit Codes Examples with Exception Mapping](https://www.amitph.com/spring-boot-exit-codes/)
* [Validations with @ConfigurationProperties in Spring Boot](https://www.amitph.com/spring-boot-configuration-properties-validation/)
* [YAML to Java List of Objects in Spring Boot](https://www.amitph.com/spring-boot-yaml-to-list/)
* [YAML to Map with Spring Boot](https://www.amitph.com/spring-boot-yaml-to-map/)