sign up log in tour help

x Dismiss

Join the Stack Overflow Community

Stack Overflow is a community of 6.7 million programmers, just like you, helping each other. Join them; it only takes a minute:

Sign up

Reading a plain text file in Java

It seems there are different ways to read and write data of files in Java.

I want to read ASCII data from a file. What are the possible ways and their differences?

java file-io ascii

edited Jan 15 '16 at 13:12 — Palec — 5,295 4 25 47

asked Jan 17 '11 at 18:29 — Tim the Enchanter — 3,276 4 12 17

18 I also disagree with closing as "not constructive". Fortunately, this could well be closed as **duplicate**. Good answers e.g. in How to create a String from the contents of a file?, What is simplest way to read a file into String?, What are the simplest classes for reading files? – Jonik Dec 29 '13 at 13:35

Without loops: {{{ Scanner sc = new Scanner(file, "UTF-8"); sc.useDelimiter("$^"); // regex matching nothing String text = sc.next(); sc.close(); }}} – Aivar Apr 28 '14 at 18:58

1 it's so interesting that there is nothing like "read()" in python , to read the whole file to a string – kommradHomer Oct 21 '14 at 8:44

2 I'm pretty sure this could be closed as "too broad". – QPaysTaxes Jan 8 '15 at 19:24

This is the simplest way to do this: mkyong.com/java/… – dellasavia Mar 17 '15 at 13:47

20 Answers

ASCII is a TEXT file so you would use Readers for reading. Java also supports reading from a binary file using InputStreams. If the files being read are huge then you would want to use a BufferedReader on top of a FileReader to improve read performance.

Go through this article on how to use a Reader

I'd also recommend you download and read this wonderful (yet free) book called **Thinking In Java**

In Java 7:

new String(Files.readAllBytes(...)) or Files.readAllLines(...)

In Java 8:

Files.lines(..).forEach(...)

edited Aug 27 '16 at 17:43 — gsamaras — 21.7k 17 40 84

answered Jan 17 '11 at 18:31 — Aravind R. Yarram — 51.5k 22 148 237

9 Picking a Reader really depends on what you need the content of the file for. If the file is small(ish) and you need it all, it's faster (benchmarked by us: 1.8-2x) to just use a FileReader and read everything (or at least large enough chunks). If you're processing it line by line then go for the BufferedReader. – Vlad Aug 27 '13 at 13:45

2 Will the line order be preserved when using "Files.lines(..).forEach(…)". My understanding is that the order will be arbitrary after this operation. – Daniil Shevelev Sep 14 '14 at 18:49

12 `Files.lines(…).forEach(…)` does not preserve order of lines but is executed in parallel, @Dash. If the order is important, you can use `Files.lines(…).forEachOrdered(…)` , which should preserve the order (did not verify though). – Palec Feb 15 '15 at 22:45

1 @Palec this is interesting, but can you quote from the docs where it says that `Files.lines(...).forEach(...)` is executed in parallel? I thought this was only the case when you explicitly make the stream parallel using `Files.lines(...).parallel().forEach(...)` . – Klitos Kyriacou Nov 9 '15 at 14:03

1 My original formulation is not bulletproof, @KlitosKyriacou. The point is that `forEach` does not guarantee

any order and the reason is easy parallelization. If order is to be preserved, use `forEachOrdered` . – Palec Nov 10 '15 at 5:48

Did you find this question interesting? Try our newsletter

Sign up for our newsletter and get our top new questions delivered to your inbox (see an example).

My favorite way to read a small file is to use a BufferedReader and a StringBuilder. It is very simple and to the point (though not particularly effective, but good enough for most cases):

```
BufferedReader br = new BufferedReader(new FileReader("file.txt"));
try {
    StringBuilder sb = new StringBuilder();
    String line = br.readLine();

    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = br.readLine();
    }
    String everything = sb.toString();
} finally {
    br.close();
}
```

Some has pointed out that after Java 7 you should use try-with-resources (i.e. auto close) features:

```
try(BufferedReader br = new BufferedReader(new FileReader("file.txt"))) {
    StringBuilder sb = new StringBuilder();
    String line = br.readLine();

    while (line != null) {
        sb.append(line);
        sb.append(System.lineSeparator());
        line = br.readLine();
    }
    String everything = sb.toString();
}
```

When I read strings like this, I usually want to do some string handling per line anyways, so then I go for this implementation.

Though if I want to actually just read a file into a String, I always use Apache Commons IO with the class IOUtils.toString() method. You can have a look at the source here:

http://www.docjar.com/html/api/org/apache/commons/io/IOUtils.java.html

```
FileInputStream inputStream = new FileInputStream("foo.txt");
try {
    String everything = IOUtils.toString(inputStream);
} finally {
    inputStream.close();
}
```

And even simpler with Java 7:

```
try(FileInputStream inputStream = new FileInputStream("foo.txt")) {
    String everything = IOUtils.toString(inputStream);
    // do something with everything string
}
```

edited Jan 4 at 9:43 — gtonic — 1,307 11 25

answered Jan 17 '11 at 18:42 — Knubo — 6,402 3 12 20

5 I've made a small adjustment to stop adding a newline (\n) if the last line is reached. `code` while (line != null) { sb.append(line); line = br.readLine(); // Only add new line when curline is NOT the last line.. if(line != null) { sb.append("\n"); } } `code` – Ramon Fincken Apr 16 '13 at 11:07

1 I feel it is better to explicitly give the encoding option and use the FileInputStream rather than directly the FileReader ? See this question too stackoverflow.com/questions/696626/… `reader = new InputStreamReader(new FileInputStream("<filePath>", "UTF-8"); – Alex Punnen Jun 4 '13 at 6:43

1 Similar to Apache Common IO IOUtils#toString() is sun.misc.IOUtils#readFully(), which is included in the Sun/Oracle JREs. – gb96 Jul 5 '13 at 0:55

3 For performance always call sb.append('\n') in preference to sb.append("\n") as a char is appended to the StringBuilder faster than a String – gb96 Jul 5 '13 at 0:58

3 there is no need to use readers directly and also no need for ioutils. java7 has built in methods to read an entire file/all lines: See docs.oracle.com/javase/7/docs/api/java/nio/file/… and docs.oracle.com/javase/7/docs/api/java/nio/file/… – kritzikratzi Mar 23 '14 at 18:48

The easiest way is to use the `Scanner` class in Java and the FileReader object. Simple example:

```
Scanner in = new Scanner(new FileReader("filename.txt"));
```

`Scanner` has several methods for reading in strings, numbers, etc... You can look for more information on this on the Java documentation page.

edited Jul 19 '14 at 16:00

Peter Mortensen
10.8k 13 75 109

answered Jan 17 '11 at 18:35

Jesus Ramos
18.5k 7 42 72

21 while (in.hasNext()) { System.out.println (in.next()); } – gnB Apr 18 '14 at 20:15

not so efficient like BufferedReader – Sazzad Hissain Khan Aug 13 '14 at 3:54

11 @Hissain But much easier to use than `BufferedReader` – Jesus Ramos Aug 13 '14 at 4:38

1 Must Surround it with try Catch – Rahal Kanishka Jun 27 '16 at 12:03

Here's another way to do it without using external libraries:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public String readFile(String filename)
{
    String content = null;
    File file = new File(filename); //for ex foo.txt
    FileReader reader = null;
    try {
        reader = new FileReader(file);
        char[] chars = new char[(int) file.length()];
        reader.read(chars);
        content = new String(chars);
        reader.close();
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if(reader !=null){reader.close();}
    }
    return content;
}
```

edited Jul 30 '15 at 11:08

Victor Polevoy
4,278 1 19 54

answered May 22 '12 at 21:02

Grimy
503 4 2

34 Close the reader in finally – PhiLho May 28 '13 at 13:50

8 or use "try-with-resources" *try(FileReader reader = new FileReader(file))* – Hernán Eche Jan 16 '14 at 13:04

3 I noticed the file.length(), How well does this work with utf-16 files? – Wayne Jan 30 '14 at 3:02

3 This technique assumes that read() fills the buffer; that the number of chars equals the number of bytes; that the number of bytes fits into memory; and that the number of bytes fits into an integer. -1 – EJP Aug 28 '14 at 10:01

2 This code is just simply wrong. – Stefan Reich Sep 17 '15 at 18:56

Here is a simple solution:

```
String content;

content = new String(Files.readAllBytes(Paths.get("sample.txt")));
```

answered Jan 29 '15 at 16:24

Nery Jr
2,099 14 19

5 This is mentioned in the accepted answer. – Palec Jan 30 '15 at 18:06

The methods within `org.apache.commons.io.FileUtils` may also be very handy, e.g.:

```
/**
 * Reads the contents of a file line by line to a List
 * of Strings using the default encoding for the VM.
 */
static List readLines(File file)
```

edited Feb 15 '15 at 12:00

Palec
5,295 4 25 47

answered Jan 17 '11 at 18:46

Claude
317 1 9

Or if you prefer **Guava** (a more modern, actively maintained library), it has similar utilities in its Files class. Simple examples in this answer. – Jonik Dec 29 '13 at 13:26

or you simply use the built in method to get all lines: docs.oracle.com/javase/7/docs/api/java/nio/file/… –

kritzikratzi Mar 23 '14 at 18:50

What do you want to do with the text? Is the file small enough to fit into memory? I would try to find the simplest way to handle the file for your needs. The FileUtils library is very handle for this.

```
for(String line: FileUtils.readLines("my-text-file"))
    System.out.println(line);
```

answered Jan 17 '11 at 22:33

Peter Lawrey
368k 42 428 752

1 Where did you get FileUtils? – toc777 Jul 20 '11 at 13:02

2 it's also built into java7: docs.oracle.com/javase/7/docs/api/java/nio/file/… – kritzikratzi Mar 23 '14 at 18:51

@PeterLawrey probably means `org.apache.commons.io.FileUtils` . Google link may change content over time, as the most widespread meaning shifts, but this matches his query and looks correct. – Palec Feb 15 '15 at 11:42

1 Unfortunately, nowadays there is no `readLines(String)` and `readLines(File)` is deprecated in favor of `readLines(File, Charset)` . The encoding can be supplied also as a string. – Palec Feb 15 '15 at 11:46

A marginally older answer suggesting the same method – Palec Feb 15 '15 at 11:56

Probably not as fast as with buffered I/O, but quite terse:

```
String content;
try (Scanner scanner = new Scanner(textFile).useDelimiter("\\Z")) {
    content = scanner.next();
}
```

The `\Z` pattern tells the `Scanner` that the delimiter is EOF.

edited Jan 14 at 14:29

Peter Mortensen
10.8k 13 75 109

answered Dec 31 '14 at 13:00

David Soroko
2,734 13 21

A very related, already existing answer is by Jesus Ramos. – Palec Jan 2 '15 at 8:59

1 This won't work if file is empty. – vitaut Aug 12 '15 at 18:57

1 True, should be: `if(scanner.hasNext()) content = scanner.next();` – David Soroko Aug 13 '15 at 21:14

1 This fails for me on Android 4.4. Only 1024 bytes are read. YMMV. – Roger Keays Nov 24 '16 at 10:11

I had to benchmark the different ways. I shall comment on my findings but, in short, the fastest way is to use a plain old BufferedInputStream over a FileInputStream. If many files must be read then three threads will reduce the total execution time to roughly half, but adding more threads will progressively degrade performance until making it take three times longer to complete with twenty threads than with just one thread.

The assumption is that you must read a file and do something meaningful with its contents. In the examples here is reading lines from a log and count the ones which contain values that exceed a certain threshold. So I am assuming that the one-liner Java 8 `Files.lines(Paths.get("/path/to/file.txt")).map(line -> line.split(";"))` is not an option.

I tested on Java 1.8, Windows 7 and both SSD and HDD drives.

I wrote six different implementations:

rawParse: Use BufferedInputStream over a FileInputStream and then cut lines reading byte by byte. This outperformed any other single-thread approach, but it may be very inconvenient for non-ASCII files.

lineReaderParse: Use a BufferedReader over a FileReader, read line by line, split lines by calling String.split(). This is approximatedly 20% slower that rawParse.

lineReaderParseParallel: This is the same as lineReaderParse, but it uses several threads. This is the fastest option overall in all cases.

nioFilesParse: Use java.nio.files.Files.lines()

nioAsyncParse: Use an AsynchronousFileChannel with a completion handler and a thread pool.

nioMemoryMappedParse: Use a memory-mapped file. This is really a bad idea yielding execution times at least three times longer than any other implementation.

These are the average times for reading 204 files of 4 MB each on an quad-core i7 and SSD drive. The files are generated on the fly to avoid disk caching.

```
rawParse                11.10 sec
lineReaderParse         13.86 sec
lineReaderParseParallel  6.00 sec
nioFilesParse           13.52 sec
nioAsyncParse           16.06 sec
nioMemoryMappedParse    37.68 sec
```

I found a difference smaller than I expected between running on an SSD or an HDD drive being the SSD approximately 15% faster. This may be because the files are generated on an unfragmented HDD and they are read sequentially, therefore the spinning drive can perform nearly as an SSD.

I was surprised by the low performance of the nioAsyncParse implementation. Either I have implemented something in the wrong way or the multi-thread implementation using NIO and a completion handler performs the same (or even worse) than a single-thread implementation with the java.io API. Moreover the asynchronous parse with a CompletionHandler is much longer in lines of code and tricky to implement correctly than a straight implementation on old streams.

Now the six implementations followed by a class containing them all plus a parametrizable main() method that allows to play with the number of files, file size and concurrency degree. Note that the size of the files varies plus minus 20%. This is to avoid any effect due to all the files being of exactly the same size.

rawParse

```
public void rawParse(final String targetDir, final int numberOfFiles) throws IOException,
ParseException {
    overrunCount = 0;
    final int dl = (int) ';';
    StringBuffer lineBuffer = new StringBuffer(1024);
    for (int f=0; f<numberOfFiles; f++) {
        File fl = new File(targetDir+filenamePreffix+String.valueOf(f)+".txt");
        FileInputStream fin = new FileInputStream(fl);
        BufferedInputStream bin = new BufferedInputStream(fin);
        int character;
        while((character=bin.read())!=-1) {
            if (character==dl) {

                // Here is where something is done with each line
                doSomethingWithRawLine(lineBuffer.toString());
                lineBuffer.setLength(0);
            }
            else {
                lineBuffer.append((char) character);
            }
        }
        bin.close();
        fin.close();
    }
}

public final void doSomethingWithRawLine(String line) throws ParseException {
    // What to do for each line
    int fieldNumber = 0;
    final int len = line.length();
    StringBuffer fieldBuffer = new StringBuffer(256);
    for (int charPos=0; charPos<len; charPos++) {
        char c = line.charAt(charPos);
        if (c==DL0) {
            String fieldValue = fieldBuffer.toString();
            if (fieldValue.length()>0) {
                switch (fieldNumber) {
                    case 0:
                        Date dt = fmt.parse(fieldValue);
                        fieldNumber++;
                        break;
                    case 1:
                        double d = Double.parseDouble(fieldValue);
                        fieldNumber++;
                        break;
                    case 2:
                        int t = Integer.parseInt(fieldValue);
                        fieldNumber++;
                        break;
                    case 3:
                        if (fieldValue.equals("overrun"))
                            overrunCount++;
                        break;
                }
            }
            fieldBuffer.setLength(0);
        }
        else {
            fieldBuffer.append(c);
        }
    }
}
```

lineReaderParse

```
public void lineReaderParse(final String targetDir, final int numberOfFiles) throws
IOException, ParseException {
    String line;
    for (int f=0; f<numberOfFiles; f++) {
        File fl = new File(targetDir+filenamePreffix+String.valueOf(f)+".txt");
        FileReader frd = new FileReader(fl);
        BufferedReader brd = new BufferedReader(frd);

        while ((line=brd.readLine())!=null)
```

```
            doSomethingWithLine(line);
        brd.close();
        frd.close();
    }
}

public final void doSomethingWithLine(String line) throws ParseException {
    // Example of what to do for each line
    String[] fields = line.split(";");
    Date dt = fmt.parse(fields[0]);
    double d = Double.parseDouble(fields[1]);
    int t = Integer.parseInt(fields[2]);
    if (fields[3].equals("overrun"))
        overrunCount++;
}
```

lineReaderParseParallel

```
public void lineReaderParseParallel(final String targetDir, final int numberOfFiles, final
int degreeOfParalelism) throws IOException, ParseException, InterruptedException {
    Thread[] pool = new Thread[degreeOfParalelism];
    int batchSize = numberOfFiles / degreeOfParalelism;
    for (int b=0; b<degreeOfParalelism; b++) {
        pool[b] = new LineReaderParseThread(targetDir, b*batchSize,
b*batchSize+b*batchSize);
        pool[b].start();
    }
    for (int b=0; b<degreeOfParalelism; b++)
        pool[b].join();
}

class LineReaderParseThread extends Thread {

    private String targetDir;
    private int fileFrom;
    private int fileTo;
    private DateFormat fmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
    private int overrunCounter = 0;

    public LineReaderParseThread(String targetDir, int fileFrom, int fileTo) {
        this.targetDir = targetDir;
        this.fileFrom = fileFrom;
        this.fileTo = fileTo;
    }

    private void doSomethingWithTheLine(String line) throws ParseException {
        String[] fields = line.split(DL);
        Date dt = fmt.parse(fields[0]);
        double d = Double.parseDouble(fields[1]);
        int t = Integer.parseInt(fields[2]);
        if (fields[3].equals("overrun"))
            overrunCounter++;
    }

    @Override
    public void run() {
        String line;
        for (int f=fileFrom; f<fileTo; f++) {
            File fl = new File(targetDir+filenamePreffix+String.valueOf(f)+".txt");
            try {
            FileReader frd = new FileReader(fl);
            BufferedReader brd = new BufferedReader(frd);
            while ((line=brd.readLine())!=null) {
                doSomethingWithTheLine(line);
            }
            brd.close();
            frd.close();
            } catch (IOException | ParseException ioe) { }
        }
    }
}
```

nioFilesParse

```
public void nioFilesParse(final String targetDir, final int numberOfFiles) throws
IOException, ParseException {
    for (int f=0; f<numberOfFiles; f++) {
        Path ph = Paths.get(targetDir+filenamePreffix+String.valueOf(f)+".txt");
        Consumer<String> action = new LineConsumer();
        Stream<String> lines = Files.lines(ph);
        lines.forEach(action);
        lines.close();
    }
}


class LineConsumer implements Consumer<String> {

    @Override
    public void accept(String line) {

        // What to do for each line
        String[] fields = line.split(DL);
        if (fields.length>1) {
            try {
                Date dt = fmt.parse(fields[0]);
            }
            catch (ParseException e) {
            }
            double d = Double.parseDouble(fields[1]);
            int t = Integer.parseInt(fields[2]);
            if (fields[3].equals("overrun"))
                overrunCount++;
```

```
        }
    }
}
```

nioAsyncParse

```
public void nioAsyncParse(final String targetDir, final int numberOfFiles, final int
numberOfThreads, final int bufferSize) throws IOException, ParseException,
InterruptedException {
    ScheduledThreadPoolExecutor pool = new ScheduledThreadPoolExecutor(numberOfThreads);
    ConcurrentLinkedQueue<ByteBuffer> byteBuffers = new ConcurrentLinkedQueue<ByteBuffer>
();

    for (int b=0; b<numberOfThreads; b++)
        byteBuffers.add(ByteBuffer.allocate(bufferSize));

    for (int f=0; f<numberOfFiles; f++) {
        consumerThreads.acquire();
        String fileName = targetDir+filenamePreffix+String.valueOf(f)+".txt";
        AsynchronousFileChannel channel =
AsynchronousFileChannel.open(Paths.get(fileName), EnumSet.of(StandardOpenOption.READ),
pool);
        BufferConsumer consumer = new BufferConsumer(byteBuffers, fileName, bufferSize);
        channel.read(consumer.buffer(), 0l, channel, consumer);
    }
    consumerThreads.acquire(numberOfThreads);
}


class BufferConsumer implements CompletionHandler<Integer, AsynchronousFileChannel> {

        private ConcurrentLinkedQueue<ByteBuffer> buffers;
        private ByteBuffer bytes;
        private String file;
        private StringBuffer chars;
        private int limit;
        private long position;
        private DateFormat frmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

        public BufferConsumer(ConcurrentLinkedQueue<ByteBuffer> byteBuffers, String
fileName, int bufferSize) {
            buffers = byteBuffers;
            bytes = buffers.poll();
            if (bytes==null)
                bytes = ByteBuffer.allocate(bufferSize);

            file = fileName;
            chars = new StringBuffer(bufferSize);
            frmt = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
            limit = bufferSize;
            position = 0l;
        }

        public ByteBuffer buffer() {
            return bytes;
        }

        @Override
        public synchronized void completed(Integer result, AsynchronousFileChannel
channel) {

            if (result!=-1) {
                bytes.flip();
                final int len = bytes.limit();
                int i = 0;
                try {
                    for (i = 0; i < len; i++) {
                        byte by = bytes.get();
                        if (by=='\n') {
                            // ***
                            // The code used to process the line goes here
                            chars.setLength(0);
                        }
                        else {
                                chars.append((char) by);
                        }
                    }
                }
                catch (Exception x) {
                    System.out.println(
                        "Caught exception " + x.getClass().getName() + " " +
x.getMessage() +
                        " i=" + String.valueOf(i) + ", limit=" + String.valueOf(len) +
                        ", position="+String.valueOf(position));
                }

                if (len==limit) {
                    bytes.clear();
                    position += len;
                    channel.read(bytes, position, channel, this);
                }
                else {
                    try {
                        channel.close();
                    }
                    catch (IOException e) {
                    }
                    consumerThreads.release();
                    bytes.clear();
                    buffers.add(bytes);
                }
            }
            else {
```

```
                try {
                    channel.close();
                }
                catch (IOException e) {
                }
                consumerThreads.release();
                bytes.clear();
                buffers.add(bytes);
            }
        }

        @Override
        public void failed(Throwable e, AsynchronousFileChannel channel) {
        }
};
```

FULL RUNNABLE IMPLEMENTATION OF ALL CASES

https://github.com/sergiomt/javaiobenchmark/blob/master/FileReadBenchmark.java

edited Jan 14 at 14:49

Peter Mortensen
10.8k 13 75 109

answered Nov 14 '16 at 20:20

Sergio Montoro
2,041 7 16

I don't see it mentioned yet in the other answers so far. But if "Best" means speed, then the new Java I/O (NIO) might provide the fastest preformance, but not always the easiest to figure out for someone learning.

http://download.oracle.com/javase/tutorial/essential/io/file.html

answered Jan 17 '11 at 19:45

jzd
20.1k 7 37 69

Using BufferedReader:

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

BufferedReader br;
try {
    br = new BufferedReader(new FileReader("/fileToRead.txt"));
    try {
        String x;
        while ( (x = br.readLine()) != null ) {
            // Printing out each line in the file
            System.out.println(x);
        }
    }
    catch (IOException e) {
        e.printStackTrace();
    }
}
catch (FileNotFoundException e) {
    System.out.println(e);
    e.printStackTrace();
}
```

edited Jan 14 at 14:35

Peter Mortensen
10.8k 13 75 109

answered Dec 26 '15 at 20:17

Alice
71 7

This is basically the exact same as Jesus Ramos' answer, except with **File** instead of **FileReader** plus iteration to step through the contents of the file.

```
Scanner in = new Scanner(new File("filename.txt"));

while (in.hasNext()) { // Iterates each line in the file
    String line = in.nextLine();
    // Do something with line
}

in.close(); // Don't forget to close resource leaks
```

... throws `FileNotFoundException`

edited Jan 14 at 14:35

Peter Mortensen
10.8k 13 75 109

answered Jul 19 '16 at 5:13

ThisClark
4,101 4 20 38

File vs FileReader: With a FileReader, the file must exist and operating system permissions must permit access. With a File, it is possible to test those permissions or check if the file is a directory. File has useful functions: isFile(), isDirectory(), listFiles(), canExecute(), canRead(), canWrite(), exists(), mkdir(), delete(). File.createTempFile() writes to the system default temp directory. This method will return a file object that can be used to open FileOutputStream objects, etc. source – ThisClark Nov 22 '16 at 15:13

Below is a one-liner of doing it in the Java 8 way. Assuming `text.txt` file is in the root of the project directory of the Eclipse.

```
Files.lines(Paths.get("text.txt")).collect(Collectors.toList());
```

edited Jan 14 at 14:52

Peter Mortensen
10.8k 13 75 109

answered Nov 15 '16 at 17:07

Zeus
3,115 2 18 35

The most simple way to read data from a file in Java is making use of the **File** class to read the file and the **Scanner** class to read the content of the file.

```
public static void main(String args[])throws Exception
{
   File f = new File("input.txt");
   takeInputIn2DArray(f);
}

public static void takeInputIn2DArray(File f) throws Exception
{
    Scanner s = new Scanner(f);
    int a[][] = new int[20][20];
    for(int i=0; i<20; i++)
    {
        for(int j=0; j<20; j++)
        {
            a[i][j] = s.nextInt();
        }
    }
}
```

PS: Don't forget to import java.util.*; for Scanner to work.

edited Jan 14 at 14:30

Peter Mortensen
10.8k 13 75 109

answered Feb 2 '15 at 14:48

anadir47
31 4

For JSF-based Maven web applications, just use ClassLoader and the `Resources` folder to read in any file you want:

1. Put any file you want to read in the Resources folder.
2. Put the Apache Commons IO dependency into your POM:

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
</dependency>
```

3. Use the code below to read it (e.g. below is reading in a .json file):

```
String metadata = null;
FileInputStream inputStream;
try {

    ClassLoader loader = Thread.currentThread().getContextClassLoader();
    inputStream = (FileInputStream) loader
            .getResourceAsStream("/metadata.json");
    metadata = IOUtils.toString(inputStream);
    inputStream.close();
}
catch (FileNotFoundException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
catch (IOException e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
return metadata;
```

You can do the same for text files, .properties files, XSD schemas, etc.

edited Jan 14 at 14:33

Peter Mortensen
10.8k 13 75 109

answered Apr 11 '15 at 8:03

fuzzyanalysis
1,660 2 24 42

works great. thanks! – Punter Vicky Nov 1 '16 at 23:54

Here are the three working and tested methods:

Using `BufferedReader`

```
package io;
import java.io.*;
public class ReadFromFile2 {
    public static void main(String[] args)throws Exception {
        File file = new File("C:\\Users\\pankaj\\Desktop\\test.java");
        BufferedReader br = new BufferedReader(new FileReader(file));
        String st;
        while((st=br.readLine()) != null){
            System.out.println(st);
        }
    }
}
```

Using `Scanner`

```
package io;

import java.io.File;
import java.util.Scanner;

public class ReadFromFileUsingScanner {
    public static void main(String[] args) throws Exception {
        File file = new File("C:\\Users\\pankaj\\Desktop\\test.java");
        Scanner sc = new Scanner(file);
        while(sc.hasNextLine()){
            System.out.println(sc.nextLine());
        }
    }
}
```

Using `FileReader`

```
package io;
import java.io.*;
public class ReadingFromFile {

    public static void main(String[] args) throws Exception {
        FileReader fr = new FileReader("C:\\Users\\pankaj\\Desktop\\test.java");
        int i;
        while ((i=fr.read()) != -1){
            System.out.print((char) i);
        }
    }
}
```

Read the entire file without a loop using the `Scanner` class

```
package io;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadingEntireFileWithoutLoop {

    public static void main(String[] args) throws FileNotFoundException {
        File file = new File("C:\\Users\\pankaj\\Desktop\\test.java");
        Scanner sc = new Scanner(file);
        sc.useDelimiter("\\Z");
        System.out.println(sc.next());
    }
}
```

edited Jan 14 at 14:54

Peter Mortensen
10.8k 13 75 109

answered Jan 10 at 18:52

pankaj
99 11

Here is a simple solution:

```
String content = new String(java.nio.file.Files.readAllBytes(
    java.nio.file.Paths.get("sample.txt")));
```

answered Nov 22 '16 at 7:21

KIM Taegyoon
776 8 10

Use Java kiss if this is about simplicity of structure:

```
import static kiss.API.*;

class App {
  void run() {
    String line;
    try (Close in = inOpen("file.dat")) {
      while ((line = readLine()) != null) {
        println(line);
      }
    }
  }
}
```

edited Jan 14 at 14:38

Peter Mortensen
10.8k 13 75 109

answered Aug 13 '16 at 7:28

Warren MacEvoy
321 3 8

Guava provides a one-liner for this:

```
import com.google.common.base.Charsets;
import com.google.common.io.Files;

String contents = Files.toString(filePath, Charsets.UTF_8);
```

edited Jan 14 at 14:38

Peter Mortensen
10.8k 13 75 109

answered Oct 12 '16 at 7:06

rahul mehra
1

This code I programmed is much faster for very large files:

```
public String readDoc(File f) {
    String text = "";
    int read, N = 1024 * 1024;
    char[] buffer = new char[N];

    try {
        FileReader fr = new FileReader(f);
        BufferedReader br = new BufferedReader(fr);

        while(true) {
            read = br.read(buffer, 0, N);
            text += new String(buffer, 0, read);

            if(read < N) {
                break;
            }
        }
    } catch(Exception ex) {
        ex.printStackTrace();
    }

    return text;
}
```

edited Oct 20 '12 at 17:29

MikkoP
1,381 7 30 67

answered May 21 '12 at 23:19

Juan Carlos Kuri Pinto
686 5 11

7 Much faster, I doubt it, if you use simple string concatenation instead of a StringBuilder... – PhiLho May 28 '13 at 13:41

4 I think the main speed gain is from reading in 1MB (1024 * 1024) blocks. However you could do the same simply by passing 1024 * 1024 as second arg to BufferedReader constructor. – gb96 Jul 5 '13 at 0:50

3 i don't believe this is tested at all. using `+=` in this way gives you quadratic (!) complexity for a task that should be linear complexity. this will start to crawl for files over a few mb. to get around this you should either keep the textblocks in a list<string> or use the aforementioned stringbuilder. – kritzikratzi Mar 23 '14 at 18:55

4 Much faster than what? It most certainly is *not* faster than appending to a StringBuffer. -1 – EJP Aug 28 '14 at 10:02

2 @EJP It probably is, if you're talking about `StringBuffer`s. **Never use them.** `StringBuilder`s are *much* better. – bcsb1001 Jan 4 '15 at 19:03

---

protected by Samuel Liew Nov 28 '16 at 3:36

Thank you for your interest in this question. Because it has attracted low-quality or spam answers that had to be removed, posting an answer now requires 10 reputation on this site (the association bonus does not count).

Would you like to answer one of these unanswered questions instead?