Reading files in Java is the cause for a lot of confusion. There are multiple ways of accomplishing the same task and it's often not clear which file reading method is best to use. Something that's quick and dirty for a small example file might not be the best method to use when you need to read a very large file. Something that worked in an earlier Java version, might not be the preferred method anymore.

This article aims to be the definitive guide for reading files in Java 7, 8 and 9. I'm going to cover all the ways you can read files in Java. Too often, you'll read an article that tells you one way to read a file, only to discover later there are other ways to do that. I'm actually going to cover 15 different ways to read a file in Java. I'm going to cover reading files in multiple ways with the core Java libraries as well as two third party libraries.

But that's not all – what good is knowing how to do something in multiple ways if you don't know which way is best for your situation?

I also put each of these methods to a real performance test and document the results. That way, you will have some hard data to know the performance metrics of each method.

Table of Contents

Methodology
JDK Versions
Java File Reading Libraries
Closing File Resources
File Location
Encoding
Download Code
Code Quality and Code Encapsulation
Exception Handling
Future Updates
File Reading Methods
Classic I/O – Reading Text
1a) FileReader – Default Encoding
1b) FileReader – Explicit Encoding (InputStreamReader)
2a) BufferedReader – Default Encoding
2b) BufferedReader – Explicit Encoding
Classic I/O – Reading Bytes
1) FileInputStream
2) BufferedInputStream
New I/O – Reading Text
1a) Files.readAllLines() – Default Encoding
1b) Files.readAllLines() – Explicit Encoding
2a) Files.lines() – Default Encoding
2b) Files.lines() – Explicit Encoding
3a) Scanner – Default Encoding
3b) Scanner – Explicit Encoding
New I/O – Reading Bytes
Files.readAllBytes()
3rd Party I/O – Reading Text
Commons – FileUtils.readLines()
Guava – Files.readLines()
Performance Testing
Dev Setup
Data Files
Performance Summary
The Winners
The Losers
Performance Rankings
Conclusion

Methodology

JDK Versions

Java code samples don’t live in isolation, especially when it comes to Java I/O, as the API keeps evolving. All code for this article has been tested on:

Java SE 7 (jdk1.7.0_80)
Java SE 8 (jdk1.8.0_162)
Java SE 9 (jdk-9.0.4)

When there is an incompatibility, it will be stated in that section. Otherwise, the code works unaltered for different Java versions. The main incompatibility is the use of lambda expressions which was introduced in Java 8.

Java File Reading Libraries

There are multiple ways of reading from files in Java. This article aims to be a comprehensive collection of all the different methods. I will cover:

java.io.FileReader.read()
java.io.BufferedReader.readLine()
java.io.FileInputStream.read()
java.io.BufferedInputStream.read()
java.nio.file.Files.readAllBytes()
java.nio.file.Files.readAllLines()
java.nio.file.Files.lines()
java.util.Scanner.nextLine()
org.apache.commons.io.FileUtils.readLines() – Apache Commons
com.google.common.io.Files.readLines() – Google Guava

Closing File Resources

Prior to JDK7, when opening a file in Java, all file resources would need to be manually closed using a try-catch-finally block. JDK7 introduced the try-with-resources statement, which simplifies the process of closing streams. You no longer need to write explicit code to close streams because the JVM will automatically close the stream for you, whether an exception occurred or not. All examples used in this article use the try-with-resources statement for importing, loading, parsing and closing files.

File Location

All examples will read test files from C:\temp.

Encoding

Character encoding is not explicitly saved with text files so Java makes assumptions about the encoding when reading files. Usually, the assumption is correct but sometimes you want to be explicit when instructing your programs to read from files. When encoding isn't correct, you'll see funny characters appear when reading files.

All examples for reading text files use two encoding variations:
Default system encoding where no encoding is specified and explicitly setting the encoding to UTF-8.

Download Code

All code files are available from Github.

Code Quality and Code Encapsulation

There is a difference between writing code for your personal or work project and writing code to explain and teach concepts.

If I was writing this code for my own project, I would use proper object-oriented principles like encapsulation, abstraction, polymorphism, etc. But I wanted to make each example stand alone and easily understood, which meant that some of the code has been copied from one example to the next. I did this on purpose because I didn't want the reader to have to figure out all the encapsulation and object structures I so cleverly created. That would take away from the examples.

For the same reason, I chose NOT to write these example with a unit testing framework like JUnit or TestNG because that's not the purpose of this article. That would add another library for the reader to understand that has nothing to do with reading files in Java. That's why all the example are written inline inside the main method, without extra methods or classes.

My main purpose is to make the examples as easy to understand as possible and I believe that having extra unit testing and encapsulation code will not help with this. That doesn't mean that's how I would encourage you to write your own personal code. It's just the way I chose to write the examples in this article to make them easier to understand.

Exception Handling

All examples declare any checked exceptions in the throwing method declaration.

The purpose of this article is to show all the different ways to read from files in Java – it's not meant to show how to handle exceptions, which will be very specific to your situation.

So instead of creating unhelpful try catch blocks that just print exception stack traces and clutter up the code, all example will declare any checked exception in the calling method. This will make the code cleaner and easier to understand without sacrificing any functionality.

Future Updates

As Java file reading evolves, I will be updating this article with any required changes.

File Reading Methods

I organized the file reading methods into three groups:

> Classic I/O classes that have been part of Java since before JDK 1.7. This includes the java.io and java.util packages.
> New Java I/O classes that have been part of Java since JDK1.7. This covers the java.nio.file.Files class.
> Third party I/O classes from the Apache Commons and Google Guava projects.

Classic I/O – Reading Text

1a) FileReader – Default Encoding

FileReader reads in one character at a time, without any buffering. It's meant for reading text files. It uses the default character encoding on your system, so I have provided examples for both the default case, as well as specifying the encoding explicitly.

```
1   import java.io.FileReader;
2   import java.io.IOException;
3
4   public class ReadFile_FileReader_Read {
5     public static void main(String [] pArgs) throws IOException {
6       String fileName = "c:\\temp\\sample-10KB.txt";
7
8       try (FileReader fileReader = new FileReader(fileName)) {
9         int singleCharInt;
10        char singleChar;
11        while((singleCharInt = fileReader.read()) != -1) {
12          singleChar = (char) singleCharInt;
```

```
13
14          //display one character at a time
15          System.out.print(singleChar);
16        }
17      }
18    }
19  }
```

1b) FileReader – Explicit Encoding (InputStreamReader)

It's actually not possible to set the encoding explicitly on a FileReader so you have to use the parent class, InputStreamReader and wrap it around a FileInputStream:

```
1   import java.io.FileInputStream;
2   import java.io.IOException;
3   import java.io.InputStreamReader;
4
5   public class ReadFile_FileReader_Read_Encoding {
6     public static void main(String [] pArgs) throws IOException {
7       String fileName = "c:\\temp\\sample-10KB.txt";
8       FileInputStream fileInputStream = new FileInputStream(fileName);
9
10      //specify UTF-8 encoding explicitly
11      try (InputStreamReader inputStreamReader =
12        new InputStreamReader(fileInputStream, "UTF-8")) {
13
14        int singleCharInt;
15        char singleChar;
16        while((singleCharInt = inputStreamReader.read()) != -1) {
17          singleChar = (char) singleCharInt;
18          System.out.print(singleChar); //display one character at a time
19        }
20      }
21    }
22  }
```

2a) BufferedReader – Default Encoding

BufferedReader reads an entire line at a time, instead of one character at a time like FileReader. It's meant for reading text files.

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

public class ReadFile_BufferedReader_ReadLine {
  public static void main(String [] args) throws IOException {
    String fileName = "c:\\temp\\sample-10KB.txt";
    FileReader fileReader = new FileReader(fileName);

    try (BufferedReader bufferedReader = new BufferedReader(fileReader))
      String line;
      while((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
}
```

2b) BufferedReader – Explicit Encoding

In a similar way to how we set encoding explicitly for FileReader, we need to create FileInputStream, wrap it inside InputStreamReader with an explicit encoding and pass that to BufferedReader:

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

public class ReadFile_BufferedReader_ReadLine_Encoding {
  public static void main(String [] args) throws IOException {
    String fileName = "c:\\temp\\sample-10KB.txt";

    FileInputStream fileInputStream = new FileInputStream(fileName);

    //specify UTF-8 encoding explicitly

  InputStreamReader inputStreamReader = new InputStreamReader(fileInputSt
8");

    try (BufferedReader bufferedReader = new BufferedReader(inputStreamRe
      String line;
      while((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
      }
    }
  }
}
```

Classic I/O – Reading Bytes

1) FileInputStream

FileInputStream reads in one byte at a time, without any buffering. While it's meant for reading binary files such as images or audio files, it can still be used to read text file. It's similar to reading with FileReader in that you're reading one character at a time as an integer and you need to cast that int to a char to see the ASCII value.

By default, it uses the default character encoding on your system, so I have provided examples for both the default case, as well as specifying the encoding explicitly.

```
1  import java.io.File;
2  import java.io.FileInputStream;
3  import java.io.FileNotFoundException;
4  import java.io.IOException;
5
6  public class ReadFile_FileInputStream_Read {
7    public static void main(String [] pArgs) throws FileNotFoundException,
8      String fileName = "c:\\temp\\sample-10KB.txt";
9      File file = new File(fileName);
10
11     try (FileInputStream fileInputStream = new FileInputStream(file)) {
12       int singleCharInt;
13       char singleChar;
14
15       while((singleCharInt = fileInputStream.read()) != -1) {
16         singleChar = (char) singleCharInt;
17         System.out.print(singleChar);
18       }
19     }
20   }
21 }
```

2) BufferedInputStream

BufferedInputStream reads a set of bytes all at once into an internal byte array buffer. The buffer size can be set explicitly or use the default, which is what we'll demonstrate in our

example. The default buffer size appears to be 8KB but I have not explicitly verified this. All performance tests used the default buffer size so it will automatically re-size the buffer when it needs to.

```
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class ReadFile_BufferedInputStream_Read {
  public static void main(String [] pArgs) throws FileNotFoundException,
    String fileName = "c:\\temp\\sample-10KB.txt";
    File file = new File(fileName);
    FileInputStream fileInputStream = new FileInputStream(file);


  try (BufferedInputStream bufferedInputStream = new BufferedInputStream(
      int singleCharInt;
      char singleChar;
      while((singleCharInt = bufferedInputStream.read()) != -1) {
        singleChar = (char) singleCharInt;
        System.out.print(singleChar);
      }
    }
  }
}
```

New I/O – Reading Text

1a) Files.readAllLines() – Default Encoding

The Files class is part of the new Java I/O classes introduced in jdk1.7. It only has static utility methods for working with files and directories.

The readAllLines() method that uses the default character encoding was introduced in jdk1.8 so this example will not work in Java 7.

```
import java.io.File;
import java.io.IOException;
import java.nio.file.Files;
import java.util.List;

```

```
6  public class ReadFile_Files_ReadAllLines {
7    public static void main(String [] pArgs) throws IOException {
8      String fileName = "c:\\temp\\sample-10KB.txt";
9      File file = new File(fileName);
10
11     List  fileLinesList = Files.readAllLines(file.toPath());
12
13     for(String line : fileLinesList) {
14       System.out.println(line);
15     }
16   }
17 }
```

1b) Files.readAllLines() – Explicit Encoding

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.nio.charset.StandardCharsets;
4  import java.nio.file.Files;
5  import java.util.List;
6
7  public class ReadFile_Files_ReadAllLines_Encoding {
8    public static void main(String [] pArgs) throws IOException {
9      String fileName = "c:\\temp\\sample-10KB.txt";
10     File file = new File(fileName);
11
12     //use UTF-8 encoding
13     List  fileLinesList = Files.readAllLines(file.toPath(),
14 StandardCharsets.UTF_8);
15
16     for(String line : fileLinesList) {
17       System.out.println(line);
18     }
19   }
   }
```

2a) Files.lines() – Default Encoding

This code was tested to work in Java 8 and 9. Java 7 didn't run because of the lack of support for lambda expressions.

```
1  import java.io.File;
```

```
import java.io.IOException;
import java.nio.file.Files;
import java.util.stream.Stream;

public class ReadFile_Files_Lines {
  public static void main(String[] pArgs) throws IOException {
    String fileName = "c:\\temp\\sample-10KB.txt";
    File file = new File(fileName);

    try (Stream linesStream = Files.lines(file.toPath())) {
      linesStream.forEach(line -> {
        System.out.println(line);
      });
    }
  }
}
```

2b) Files.lines() – Explicit Encoding

Just like in the previous example, this code was tested and works in Java 8 and 9 but not in Java 7.

```
import java.io.File;
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.util.stream.Stream;

public class ReadFile_Files_Lines_Encoding {
  public static void main(String[] pArgs) throws IOException {
    String fileName = "c:\\temp\\sample-10KB.txt";
    File file = new File(fileName);

    try (Stream linesStream = Files.lines(file.toPath(),
StandardCharsets.UTF_8)) {
      linesStream.forEach(line -> {
        System.out.println(line);
      });
    }
  }
}
```

3a) Scanner – Default Encoding

The Scanner class was introduced in jdk1.7 and can be used to read from files or from the console (user input).

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  public class ReadFile_Scanner_NextLine {
6    public static void main(String [] pArgs) throws FileNotFoundException {
7      String fileName = "c:\\temp\\sample-10KB.txt";
8      File file = new File(fileName);
9
10     try (Scanner scanner = new Scanner(file)) {
11       String line;
12       boolean hasNextLine = false;
13       while(hasNextLine = scanner.hasNextLine()) {
14         line = scanner.nextLine();
15         System.out.println(line);
16       }
17     }
18   }
19 }
```

3b) Scanner – Explicit Encoding

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.util.Scanner;
4
5  public class ReadFile_Scanner_NextLine_Encoding {
6    public static void main(String [] pArgs) throws FileNotFoundException {
7      String fileName = "c:\\temp\\sample-10KB.txt";
8      File file = new File(fileName);
9
10     //use UTF-8 encoding
11     try (Scanner scanner = new Scanner(file, "UTF-8")) {
12       String line;
13       boolean hasNextLine = false;
14       while(hasNextLine = scanner.hasNextLine()) {
15         line = scanner.nextLine();
16         System.out.println(line);
17       }
18     }
19   }
20 }
```

New I/O – Reading Bytes

Files.readAllBytes()

Even though the documentation for this method states that "it is not intended for reading in large files" I found this to be the absolute best performing file reading method, even on files as large as 1GB.

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.nio.file.Files;
4
5  public class ReadFile_Files_ReadAllBytes {
6    public static void main(String [] pArgs) throws IOException {
7      String fileName = "c:\\temp\\sample-10KB.txt";
8      File file = new File(fileName);
9
10     byte [] fileBytes = Files.readAllBytes(file.toPath());
11     char singleChar;
12     for(byte b : fileBytes) {
13       singleChar = (char) b;
14       System.out.print(singleChar);
15     }
16   }
17 }
```

3rd Party I/O – Reading Text

Commons – FileUtils.readLines()

Apache Commons IO is an open source Java library that comes with utility classes for reading and writing text and binary files. I listed it in this article because it can be used instead of the built in Java libraries. The class we're using is FileUtils.

For this article, version 2.6 was used which is compatible with JDK 1.7+

Note that you need to explicitly specify the encoding and that method for using the default encoding has been deprecated.

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.List;
4
5  import org.apache.commons.io.FileUtils;
6
7  public class ReadFile_Commons_FileUtils_ReadLines {
8    public static void main(String [] pArgs) throws IOException {
9      String fileName = "c:\\temp\\sample-10KB.txt";
10     File file = new File(fileName);
11
12     List fileLinesList = FileUtils.readLines(file, "UTF-8");
13
14     for(String line : fileLinesList) {
15       System.out.println(line);
16     }
17   }
18 }
```

Guava – Files.readLines()

Google Guava is an open source library that comes with utility classes for common tasks like collections handling, cache management, IO operations, string processing.

I listed it in this article because it can be used instead of the built in Java libraries and I wanted to compare its performance with the Java built in libraries.

For this article, version 23.0 was used.

I'm not going to examine all the different ways to read files with Guava, since this article is not meant for that. For a more detailed look at all the different ways to read and write files with Guava, have a look at Baeldung's in depth article.

When reading a file, Guava requires that the character encoding be set explicitly, just like Apache Commons.

Compatibility note: This code was tested successfully on Java 8 and 9. I couldn't get it to work on Java 7 and kept getting "Unsupported major.minor version 52.0" error. Guava has

a separate API doc for Java 7 which uses a slightly different version of the Files.readLine() method. I thought I could get it to work but I kept getting that error.

```
1  import java.io.File;
2  import java.io.IOException;
3  import java.util.List;
4
5  import com.google.common.base.Charsets;
6  import com.google.common.io.Files;
7
8  public class ReadFile_Guava_Files_ReadLines {
9    public static void main(String[] args) throws IOException {
10     String fileName = "c:\\temp\\sample-10KB.txt";
11     File file = new File(fileName);
12
13     List fileLinesList = Files.readLines(file, Charsets.UTF_8);
14
15     for(String line : fileLinesList) {
16       System.out.println(line);
17     }
18   }
19 }
```