<http://www.java-tips.org/java-se-tips/java.util/scanning-text-with-java.util.scanner-3.html>

# Java BufferedReader vs. Scanner class?

BufferedReader is the "old" way and dates back to at least JDK 1.1 (which is when I started using Java).  
  
Scanner was introduced relatively recently (JDK 5, maybe?) to handle very commonly repeated tasks (read an integer from a file, or from the keyboard) that are kind of clumsy with BufferedReader.  
  
If you only read input one line at a time, there's not a lot of difference between the two. Where Scanner shines is when you want nextInt() or nextFloat() or something like that.  
  
See the discussion here:  
[http://www.java-tips.org/java-se-tips/ja…](http://www.java-tips.org/java-se-tips/java.util/scanning-text-with-java.util.scanner-3.html)  
... for more details.

<http://www.java-tips.org/java-se-tips/java.util/scanning-text-with-java.util.scanner-3.html>

|  |  |
| --- | --- |
| **Scanning text with java.util.Scanner** | [E-mail](http://www.java-tips.org/index2.php?option=com_content&task=emailform&id=271&itemid=61) |

Top of Form

User Rating:http://www.java-tips.org/images/M_images/rating_star.pnghttp://www.java-tips.org/images/M_images/rating_star.pnghttp://www.java-tips.org/images/M_images/rating_star.pnghttp://www.java-tips.org/images/M_images/rating_star.pnghttp://www.java-tips.org/images/M_images/rating_star_blank.png / 405   
PoorBest 

Bottom of Form

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| This Tech Tip reprinted with permission by [java.sun.com](http://java.sun.com/)  J2SE 5.0 adds classes and methods that can make every day tasks easier to perform. In this tip you will see how the newly added java.util.Scanner class makes it easier to read and parse strings and primitive types using [regular expressions](http://www.java-tips.org/).  Before the J2SE 5.0 release, you probably wrote code such as the following TextReader class to read text from a file:   |  | | --- | | **import**java.io.BufferedReader;    **import**java.io.FileReader;    **import**java.io.IOException;    **import**java.io.File;     **public class**TextReader {      **private static void**readFile(String fileName) {        **try**{          File file = **new**File(fileName);          FileReader reader = **new**FileReader(file);          BufferedReader in = **new**BufferedReader(reader);          String string;          **while**((string = in.readLine()) != **null**) {            System.out.println(string);          }          in.close();        } **catch**(IOException e) {          e.printStackTrace();        }      }       **public static void**main(String[] args) {        **if**(args.length != 1) {          System.err.println("usage: java TextReader "            + "file location");          System.exit(0);        }        readFile(args[0]);      }    } |   The basic approach in classes like this is to create a File object that corresponds to the actual file on the hard drive. The class then creates a FileReader associated with the file and then a BufferedReader from the FileReader. It then uses the BufferedFile reader to read the file one line at a time.  To view the TextReader class in action, you need to create a document for the class to read and parse. To create the document, save the following two lines of text in a file named TextSample.txt in the same directory as TextReader:  Here is a small text file that you will  use to test java.util.scanner.  Compile TextReader. Then run it by entering the following:  java TextReader TextSample.txt  You should see the original file echoed back to you in standard output.  You can simplify the code in TextReader by using java.util.Scanner, a class that parses primitive types and strings:   |  | | --- | | **import**java.io.File;    **import**java.io.FileNotFoundException;    **import**java.util.Scanner;     **public class**TextScanner {       **private static void**readFile(String fileName) {        **try**{          File file = **new**File(fileName);          Scanner scanner = **new**Scanner(file);          **while**(scanner.hasNext()) {            System.out.println(scanner.next());          }          scanner.close();        } **catch**(FileNotFoundException e) {          e.printStackTrace();        }      }       **public static void**main(String[] args) {        **if**(args.length != 1) {          System.err.println("usage: java TextScanner1"            + "file location");          System.exit(0);        }        readFile(args[0]);      }    } |   Compile TextScanner. Then run it as follows:  java TextScanner TextSample.txt  You should get the following output:  Here  is  a  small  text  file  that  you  will  use  to  test  java.util.scanner.  TextScanner creates a Scanner object from the File. The Scanner breaks the contents of the File into tokens using a delimiter pattern, By default the delimiter pattern is whitespace. TextScanner then calls the hasNext() method in Scanner. This method returns true if another token exists in the Scanner's input, which is the case until it reaches the end of the file. The next() method returns a String that represents the next token. So until it reaches the end of the file, TextScanner prints the String returned by next() on a separate line.  You can change the delimeter that is used to tokenize the input, through the useDelimiter method of Scanner. You can pass in a String or a java.util.regex.Pattern to the method. See the JavaDocs page for Pattern for information on what patterns are appropriate. For example, you can read the input one line at a time by using the newline character (\n) as a delimiter. Here is the revised readFile() method for TextScanner that uses a newline character as the delimiter:   |  | | --- | | **private static void**readFile(String fileName) {      **try**{        Scanner scanner = **new**Scanner(**new**File(fileName));        scanner.useDelimiter          (System.getProperty("line.separator"));         **while**(scanner.hasNext()) {          System.out.println(scanner.next());        scanner.close();      } **catch**(FileNotFoundException e) {        e.printStackTrace();      }    } |   Note that there are other options for detecting the end of a line. You could, for example, test for lines that end with a newline character or that end with a carriage return and a newline character. You can do that using the regular expression "\r\n|\n". The JavaDocs for java.util.regex.Pattern shows other possible line terminators, so a more complete check would use the expression "\r\n|[\r\n\u2028\u2029\u0085]". You can also use the hasNextLine() and nextLine() methods from the Scanner class. In any case, with the revised TextScanner, the output should match the contents and layout of TextSample.txt. In other words, you should see the following:  Here is a small text file that you will  use to test java.util.scanner.  A simple change of the pattern in the delimiter used by the Scanner gives you a great deal of power and flexibility. For example, if you specify the following delimiter:  scanner.useDelimiter("\\z");  it reads in the entire file at once. This is similar to the trick suggested by Pat Niemeyer in his java.net blog. You can read in the entire contents of a web page without creating several intermediate objects. The code for the following class, WebPageScanner, reads in the current contents of the java.net homepage:   |  | | --- | | **import**java.net.URL;    **import**java.net.URLConnection;    **import**java.io.IOException;    **import**java.util.Scanner;     **public class**WebPageScanner {      **public static void**main(String[] args) {        **try**{          URLConnection connection =            **new**URL("http://java.net").openConnection();          String text = **new**Scanner(            connection.getInputStream()).            useDelimiter("\\Z").next();        } **catch**(IOException e) {          e.printStackTrace();        }      }    } |   You can handle more than Strings with the Scanner class. You can also use Scanner to parse data that consists of primitives. To illustrate this, save the following three lines in a file named Employee.data (in the same directory as TextSample):  Joe, 38, true  Kay, 27, true  Lou, 33, false  You could still treat this as one large String and perform the conversions after parsing the String. Instead, you can parse this file in two steps. This is illustrated in the following class, DataScanner:   |  | | --- | | **import**java.util.Scanner;    **import**java.io.File;    **import**java.io.FileNotFoundException;     **public class**DataScanner {       **private static void**readFile(String fileName) {        **try**{          Scanner scanner =            **new**Scanner(**new**File(fileName));          scanner.useDelimiter            (System.getProperty("line.separator"));           **while**(scanner.hasNext()) {            parseLine(scanner.next());          }          scanner.close();        } **catch**(FileNotFoundException e) {          e.printStackTrace();        }      }       **private static void**parseLine(String line) {        Scanner lineScanner = **new**Scanner(line);       lineScanner.useDelimiter("\\s\*,\\s\*");        String name = lineScanner.next();        **int**age = lineScanner.nextInt();        **boolean**isCertified = lineScanner.nextBoolean();        System.out.println("It is " + isCertified +          " that " + name + ", age "          + age + ", is certified.");      }       **public static void**main(String[] args) {        **if**(args.length != 1) {          System.err.println("usage: java TextScanner2"            + "file location");          System.exit(0);        }        readFile(args[0]);      }    } |   The outer Scanner object in DataScanner reads a file, one line at a time. The readFile() method passes each line to a second scanner. The second scanner parses the comma delimited data and discards the whitespace on either side of the comma. There are variants of the hasNext() and next() methods which enable you to test whether or not the next token is of a specified type and to attempt to treat the next token as an instance of that type. For example, nextBoolean() attempts to treat the next token as a boolean and tries to match it to either the String "true" or the String "false". If the match cannot be made, a java.util.InputMismatchException is thrown. The parseLine() method of DataScanner shows how each line is parsed into a String, an int, and a boolean.  Compile DataScanner. Then run it as follows:  java DataScanner Employee.data  You should get the following output:  It is true that Joe, age 38, is certified.  It is true that Kay, age 27, is certified.  It is false that Lou, age 33, is certified.  You might be tempted to use just the comma as a delimiter. In other words you might try this:  lineScanner.useDelimiter(",");  This will result in an InputMismatchException. That's because an extra space will be included in the token that you are trying to convert to a boolean, and the space does not match either "true" or "false". As is the case with all applications of regular expressions, the underlying power requires that you take extra care in constructing your patterns.  For more information on Scanner, see the [formal documentation](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Scanner.html).  Copyright (c) 2004-2005 Sun Microsystems, Inc. All Rights Reserved. |

<http://stackoverflow.com/questions/2231369/scanner-vs-bufferedreader>

Scanner is used for parsing tokens from the contents of the stream while BufferedReader just reads the stream and does not do any special parsing.

In currently latest JDK6 release/build, the [Scanner](http://java.sun.com/javase/6/docs/api/java/util/Scanner.html) has a littler buffer (1KB char buffer) as opposed to the [BufferedReader](http://java.sun.com/javase/6/docs/api/java/io/BufferedReader.html) (8KB byte buffer), but it's more than enough.

See this [link](http://en.allexperts.com/q/Java-1046/2009/2/Difference-Scanner-Method-Buffered.htm), following is quoted from there:

A BufferedReader is a simple class meant to efficiently read from the underling stream. Generally, each read request made of a Reader like a FileReader causes a corresponding read request to be made to underlying stream. Each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient. Efficiency is improved appreciably if a Reader is warped in a BufferedReader.

BufferedReader is synchronized, so read operations on a BufferedReader can safely be done from multiple threads.

A scanner on the other hand has a lot more cheese built into it; it can do all that a BufferedReader can do and at the same level of efficiency as well. However, in addition a Scanner can parse the underlying stream for primitive types and strings using regular expressions. It can also tokenize the underlying stream with the delimiter of your choice. It can also do forward scanning of the underlying stream disregarding the delimiter!

A scanner however is not thread safe, it has to be externally synchronized.

The choice of using a BufferedReader or a Scanner depends on the code you are writing, if you are writing a simple log reader Buffered reader is adequate. However if you are writing an XML parser Scanner is the more natural choice.

Even while reading the input, if want to accept user input line by line and say just add it to a file, a BufferedReader is good enough. On the other hand if you want to accept user input as a command with multiple options, and then intend to perform different operations based on the command and options specified, a Scanner will suit better.

I suggest to use BufferedReader for reading text. Scanner hides IOException while BufferedReader throws it immediately.

<http://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html>

**Class BufferedReader**

* [java.lang.Object](http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html)
  + [java.io.Reader](http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html)
    - java.io.BufferedReader
* **All Implemented Interfaces:**

[Closeable](http://docs.oracle.com/javase/7/docs/api/java/io/Closeable.html), [AutoCloseable](http://docs.oracle.com/javase/7/docs/api/java/lang/AutoCloseable.html" \o "interface in java.lang), [Readable](http://docs.oracle.com/javase/7/docs/api/java/lang/Readable.html)

**Direct Known Subclasses:**

[LineNumberReader](http://docs.oracle.com/javase/7/docs/api/java/io/LineNumberReader.html)

public class **BufferedReader**

extends [Reader](http://docs.oracle.com/javase/7/docs/api/java/io/Reader.html)

Reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.

The buffer size may be specified, or the default size may be used. The default is large enough for most purposes.

In general, each read request made of a Reader causes a corresponding read request to be made of the underlying character or byte stream. It is therefore advisable to wrap a BufferedReader around any Reader whose read() operations may be costly, such as FileReaders and InputStreamReaders. For example,

BufferedReader in

= new BufferedReader(new FileReader("foo.in"));

will buffer the input from the specified file. Without buffering, each invocation of read() or readLine() could cause bytes to be read from the file, converted into characters, and then returned, which can be very inefficient.

Programs that use DataInputStreams for textual input can be localized by replacing each DataInputStream with an appropriate BufferedReader.

**Since:**

JDK1.1

**See Also:**

[FileReader](http://docs.oracle.com/javase/7/docs/api/java/io/FileReader.html), [InputStreamReader](http://docs.oracle.com/javase/7/docs/api/java/io/InputStreamReader.html), [Files.newBufferedReader(java.nio.file.Path, java.nio.charset.Charset)](http://docs.oracle.com/javase/7/docs/api/java/nio/file/Files.html#newBufferedReader(java.nio.file.Path, java.nio.charset.Charset))

<http://stackoverflow.com/questions/1747040/difference-between-java-io-printwriter-and-java-io-bufferedwriter>

# [Difference between java.io.PrintWriter and java.io.BufferedWriter?](http://stackoverflow.com/questions/1747040/difference-between-java-io-printwriter-and-java-io-bufferedwriter)

File file = new File("blah.txt");

FileWriter fileWriter = new FileWriter(file);

BufferedWriter bWriter = new BufferedWriter(fileWriter);

The main reason to use the PrintWriter is to get access to the printXXX methods (like println(int)). You can essentially use a PrintWriter to write to a file just like you would use System.out to write to the console.

A BufferedWriter is an efficient way to write to a file (or anything else) as it will buffer the characters in Java memory before (probably, depending on the implementation) dropping to C to do the writing to the file.

There is no such concept as a "PrintReader" the closest you will come is probably java.util.Scanner.

PrintWriter gives more methods (println), but the most important (and worrying) difference to be aware of is that it swallows exceptions.

You can call checkError later on to see whether any errors have occurred, but typically you'd usePrintWriter for things like writing to the console - or in "quick 'n dirty" apps where you don't want to be bothered by exceptions (and where long-term reliability isn't an issue).

I'm not sure why the "extra formatting abilities" and "don't swallow exceptions" aspects are bundled into the same class - formatting is obviously useful in many places where you don't want exceptions to be swallowed. It would be nice to see BufferedWriter get the same abilities at some point...