

# *Notes from Well House Consultants*

*These notes are written by Well House Consultants and distributed under their Open Training Notes License. If a copy of this license is not supplied at the end of these notes, please visit*

*<http://www.wellho.net/net/whcotnl.html>  
for details.*

### 1.1 Well House Consultants

Well House Consultants provides niche training, primarily but not exclusively in Open Source programming languages. We offer public courses at our training centre and private courses at your offices. We also make some of our training notes available under our "Open Training Notes" license, such as we're doing in this document here.

### 1.2 Open Training Notes License

With an "Open Training Notes License", for which we make no charge, you're allowed to print, use and distribute these notes provided that you retain the complete and unaltered license agreement with them, including our copyright statement. This means that you can learn from the notes, and have others learn from them too.

You are NOT allowed to charge (directly or indirectly) for the copying or distribution of these notes, nor are you allowed to charge for presentations making any use of them.

### 1.3 Courses presented by the author

If you would like us to attend a course (Java, Perl, Python, PHP, Tcl/Tk, MySQL or Linux) presented by the author of these notes, please see our public course schedule at

<http://www.wellho.net/course/index.html>

If you have a group of 4 or more trainees who require the same course at the same time, it will cost you less to have us run a private course for you. Please visit our onsite training page at

<http://www.wellho.net/course/otc.html>

which will give you details and costing information

### 1.4 Contact Details

Well House Consultants may be found online at

<http://www.wellho.net>

[graham@wellho.net](mailto:graham@wellho.net)

technical contact

[lisa@wellho.net](mailto:lisa@wellho.net)

administration contact

Our full postal address is

404 The Spa

Melksham

Wiltshire

UK SN12 6QL

Phone +44 (0) 1225 708225

Fax +44 (0) 1225 707126

# *More Input and Output*

This module covers reading and writing from and to files, other processes, network connections, keyboards and screens. Also how to find out if an object exists on your file system and if it does, to get more information about it.

|                                       |    |
|---------------------------------------|----|
| <i>Overview</i> .....                 | 4  |
| <i>The Wellreader class</i> .....     | 4  |
| <i>Input/output from basics</i> ..... | 4  |
| <i>Streams</i> .....                  | 5  |
| <i>Writing to a file</i> .....        | 5  |
| <i>Formatted output</i> .....         | 8  |
| <i>Reading from a file</i> .....      | 12 |

## 2.1 Overview

For output so far, we have used the `System.out.print` method and its brother `System.out.println`. There are many more ways of doing output, including a huge selection in the `java.io` package.

For input, we have used the `wellreader` class specially provided for this course. You may wonder why we haven't even shown you one standard input class (yes, there are many).

- Input is very error prone
- If you ask for a number, your user is very likely to provide text, or an incorrect format, or an end-of-file

And so all the input routines provided as a standard part of the Java libraries include exceptions. You need to `try` and `catch`, if necessary, in almost every circumstance!

However, you are now familiar with exceptions, so let's have a look at the `wellreader` class ...

## 2.2 The Wellreader class

```
// Well House Consultants©2004.

import java.util.*;
import java.io.*;

/** Reading from the keyboard - my own class*/

public class wellreader
{
    public static float read_number()
    {
        BufferedReader standard = new BufferedReader(
            new InputStreamReader(System.in));
        try{
            String inline = standard.readLine();
            Float from_user = Float.valueOf(inline);
            return from_user.floatValue();
        }
        catch (Exception e)
        {
            return (0.0f);
        }
    }
}
```

Hmm. There are still a number of other new things there, though you will have worked out by now that if the `wellreader` class has any problem reading a number, it simply returns a zero!

## 2.3 Input/output from basics

You may want to input and output to and from a number of sources and destinations. Output could be to ...

- Text to the current window
- A device, such as a tape drive or printer
- The input channel of another process on the same system
- The input channel of a process on another system

And the output could be either direct from your application or to your shell, which in turn could output to most of those devices.

Some choice! How are we going to achieve the flexibility?

There are common characteristics in the way you'll wish to handle output to all these devices:

- You'll want to squirt characters, sometimes one at a time, sometimes in larger spurts
- For the sake of efficiency, you'll want the system to take the squirts and spurts and form them into larger chunks which are written to the device. In the case of a disc, for example, sector by sector

Do you feel a class of object coming on? Yes, all our squirts and spurts get processed through a **stream**.

## 2.4 Streams

A similar logic can be followed through for an input stream and how that works. But streams do not apply to:

- Graphics / windows output
- Input relating to the graphics (mouse movements, etc)

You'll be introduced to the classes we use to handle graphics and events in the last section of this course, and learn a lot more about them on the *Java Advanced* course!

There are, in fact, two types of stream Classes available in Java:

- **byte streams**. Used for handling data accessed by (and only by) Java programs, using unicones
- **character streams**. Used for handling data accessed by other applications (perhaps as well as by Java applications)

For character streams, we use:

**OutputStream** for writing

**InputStream** for reading

but both are abstract classes, declaring a basic set of operations available on all streams, and so cannot be instantiated directly.

Similarly, for byte streams we use:

**Writer** for writing

**Reader** for reading

both of which (again) are abstract classes.

## 2.5 Writing to a file

The final type of class you'll need will be the *File* class which (you can guess!) holds information about and provides methods to check and access files.

We'll come back to *File* objects soon, but let's start with a short example of writing to a file using a byte stream.

Example *broughton.java*. Same as *whitley.java* except ...

```
// And print out cards!

FileWriter demofile = new FileWriter("abc.123");

for (int k=0;k<cardcount;k++)
{
System.out.println(play[k].getcardname());
demofile.write("card number "+(k+1)+" is ");
demofile.write(play[k].getcardname()+"\n");
}
demofile.close();

}
}
```

We created an instance of object *FileWriter*, which is one of the derived classes of the *Writer* abstract class.

```
FileWriter demofile = new FileWriter("abc.123");
```

Our *Filewriter* object is in fact implemented as a file on the *file* system, which is opened by the successful calling of the constructor method since this is a character rather than a byte stream.

In other words, after this line has been executed, a file called *abc.123* has been created which we can write to via the *demofile* object. Let's go on and do so!!

We can write to the file using the *write* method:

```
demofile.write("card number "+(k+1)+" is ");
demofile.write(play[k].getcardname()+"\n");
```

which is, in fact, defined in the superclass. In other words, although we are writing to file here, the method would work just as well on other output streams such as:

- *StringWriter*
- *CharArrayWriter*
- *PipeWriter*

which write to a character string, a character array, and a pipe to another process, respectively.

The method also works to output streams:

- *BufferedWriter*
- *FilterWriter*

which are actually extra streams that fit between the application and the file, buffering (for efficiency) or filtering / converting data.

OK, this is getting a bit exotic, but it does give you some idea of the flexibility! We'll come back to that later.

And when we have finished writing to the file, we close it with

```
demofile.close();
```

Let's look at some more mundane things associated with files, the bread and butter into which we can put our fancy kangaroo meat and sweetcorn filling, later.

The areas we should look at include:

- Ensuring that we can create the file and that it creates correctly and that there is enough disc space
- Ensuring that we are not overwriting an existing file
- Ensuring that the write operations work
- More flexibility in how we write to the file, and indeed on how we write to the screen! To come up with tabular data, for example, would be a nightmare in what you have seen so far!

Let's start with the opening of the file and checking it.

Actually, in our *broughton.java* example we did have to make one consideration of error conditions. If you compile it with just the open/write/close sequence added from the previous example, you'll get the error message:

```
seal% javac broughton.java
```

```
broughton.java:29: Exception java.io.IOException must be caught, or it must be declared in the throws clause of this method.
```

```
    FileWriter demofile = new FileWriter("abc.123");
```

```
        ^
```

```
1 error
```

```
seal%
```

And in our example, the author cheated by just passing up the exception to the calling class, changing:

```
public static void main(String[] args)
to:
public static void main(String[] args)
                                throws IOException
```

However, rather than give a **String** argument to *FileWriter*, it would be much more robust and flexible for us to use the *File* Class directly. Let's see an example:

```
File demofile = new File(infile);

if (! demofile.exists())
{
FileWriter demo = new FileWriter(demofile);
demo.write
("// File created by Java Class winsley\n");
demo.close();
}
else
{
System.out.print
("Sorry - file already exists\n");
long how_big = demofile.length();
System.out.println("length "+how_big);
}
}
```

We have created an object of class *File* which we have then examined with the **exists** method to ensure that we do not overwrite an existing file.

Note that creating a *File* object does not itself create or open a file. The file is still created/opened by the *FileWriter* method.

We can use our *File* object to learn more about our file using other class methods. In our example, we have chosen to report on the length of the file.

```
seal% java winsley
File creation
Please enter the name of a filefred
seal% java winsley
File creation
Please enter the name of a filefred
Sorry - file already exists
length 38
seal% ls -l fred
-rw-r--r--  1 graham  1999  38 Jan 11 01:50 fred
seal%
```

Tests that return boolean results on file objects include:

```
exists()
isDirectory()
isFile()
isAbsolute()
canRead()
canWrite()
```

Tests that return other information include:

```
getName()      string - eliminates the path
getPath()      string - full path and name
length()       long - file size in bytes
lastModified() long - milliseconds from 1.1.70
```

Other useful methods in the *File* class include:

- An overridden **equals** method which returns "true" if two file objects point to one and the same file, even if objects themselves are different
- A **list** method that will return an array of strings containing the names of all the entries in a directory<sup>1</sup>
- Methods for file system manipulation including **mkdir**, **makedirs**, **renameTo** and **delete**

## 2.6 Formatted output

There is no "printf" as in C or Perl. Instead, you'll use:

- Classes such as **DecimalFormat** which we saw earlier to define the format

```
DecimalFormat display_temp =
    new DecimalFormat("###0.00 degrees;###0.00 deg below");
```

- Methods such as **format** to turn the value into a **String**
- The regular **print** or **println** to actually output it to the stream

Let's see an example converting temperatures from centigrade to fahrenheit, formatting the temperatures differently. Main code:

```
// Well House Consultants 2004.
import java.io.*;
import java.text.*;
/** Formatted printing */
public class bratton

{
    public static void main(String[] args)
        throws IOException

    {
        String inline;
        File demofile = new File("abc.txt");

        if (demofile.exists())
        {
            System.out.print
                ("Sorry - file already exists\n");
            System.exit(1);
        }

        FileWriter demo = new FileWriter(demofile);
        demo.write
            ("// File created by Java Class bratton\n");
        formatted_table(demo, -50.0, 100.0, 150);
        demo.close();
    }
}
```

<sup>1</sup> it is also possible to specify a filter

The `formatted_table` function is where the new stuff is ...

```
private static void formatted_table
    (FileWriter destiny,
     double lower, double upper, int steps)
    throws IOException
{double temper,faren;
  String tstring;
  DecimalFormat centFormat = new DecimalFormat(
"##,##0.00 degrees C;###0.00 deg'. C below");
  DecimalFormat farFormat = new DecimalFormat(
"###0.0# degrees F");
  farFormat.setPositivePrefix("+");
```

A temperature is really an object, so we create formatting objects which we will associate with the temperatures.

The `farFormat` object is set up as follows:

```
"###0.0# degrees F"
```

Indicating ...

- A series of digits which may be omitted if there are leading zeros (signified by the #)
- A digit (a zero being shown if everything is leading zeros)
- A decimal point
- Up to two places after the decimal point, at least one being shown
- The literal text " degrees F"

Other attributes of our `DecimalFormat` object can be set; in this case we have chosen to set the positive number prefix:

```
farFormat.setPositivePrefix("+");
```

Here are some examples of what is printed when we apply the format (we'll show you the actual code in a page or two):

```
-58.0 degrees F
-56.2 degrees F
-54.39 degrees F
-0.39 degrees F
+1.4 degrees F
+32.0 degrees F
+212.0 degrees F
```

To show you further facilities of the `DecimalFormat` object, we have set up a different object for the centigrade temperature:

```
DecimalFormat centFormat = new DecimalFormat
("##,##0.00 degrees C;###0.00 deg'. C below");
```

The ";" in the format indicates that we are providing separate formats for positive and negative numbers; nothing so simple as an automatic minus sign as we had in the last example!

Positive numbers, described first:

- Show a single leading zero if they are less than one degree
- Have a comma separator every two digits (yes, we know the usual thing would be every three digits)
- Always show exactly two figures after the decimal point, even if they are both zero

Negative numbers, by contrast:

- Have no comma separators
- Have differing text including a fullstop, which we "escape" with a quote to take away any special significance
- Do not have a minus sign

Here are some examples of applying the centigrade format:

```
50.00 deg. C below
1.00 deg. C below
0.00 degrees C
1.00 degrees C
10.00 degrees C
1,00.00 degrees C
```

Finally, this code applies the appropriate `DecimalFormat` to the temperatures, producing a `String` object which is then printed out:

```
for (temper=lower;
    temper<=upper;
    temper += (upper-lower)/steps)
{
    tstring = centFormat.format(temper);
    destiny.write(tstring);
    faren = temper / 5.0 * 9.0 + 32.0;
    tstring = farFormat.format(faren);
    destiny.write(" converts to "+tstring+"\n");
}
```

Here's part of the output file to complete the example:

```
// File created by Java Class bratton
50.00 deg. C below converts to -58.0 degrees F
49.00 deg. C below converts to -56.2 degrees F
48.00 deg. C below converts to -54.39 degrees F
47.00 deg. C below converts to -52.6 degrees F
[snip]
20.00 deg. C below converts to -4.0 degrees F
19.00 deg. C below converts to -2.19 degrees F
18.00 deg. C below converts to -0.39 degrees F
17.00 deg. C below converts to +1.4 degrees F
16.00 deg. C below converts to +3.19 degrees F
[snip]
3.00 deg. C below converts to +26.6 degrees F
2.00 deg. C below converts to +28.4 degrees F
1.00 deg. C below converts to +30.2 degrees F
0.00 degrees C converts to +32.0 degrees F
1.00 degrees C converts to +33.79 degrees F
2.00 degrees C converts to +35.6 degrees F
3.00 degrees C converts to +37.4 degrees F
[snip]
97.00 degrees C converts to +206.6 degrees F
98.00 degrees C converts to +208.4 degrees F
99.00 degrees C converts to +210.2 degrees F
1,00.00 degrees C converts to +212.0 degrees F
```

## Exercise

Extend your program from the previous example to read four integer values and write them out to a file. If the file already exists, flag that fact before you even start, and exit.

### Our example answer is *dulcimer*

#### *Sample*

```
seal% java dulcimer
Output file already exists
seal% rm output.txt
seal% java dulcimer
give me number 1 please45
give me number 2 please3
give me number 3 pleasehello
Expecting an integer. Try again9
give me number 4 please+4
seal% cat output.txt
I got 45
I got 3
I got 9
I got -4
seal%
```

#### *For Advanced Students*

Replace our *wellreader.readLine()* method with a method you write/copy yourself!

- Improve the method as you copy it in by adding better exception handling
- Pass the exceptions back to your calling method

## 2.7 Reading from a file

We'll extend our previous example to say "if the file *does* exist, read and display the first few bytes" ...

```
File demofile = new File(infile);

if (! demofile.exists())
{
FileWriter demo = new FileWriter(demofile);
demo.write
("// File created by Java Class winsley\n");
demo.close();
}
else
{
FileReader demo = new FileReader(demofile);
int inchar = -1;
for (int k=0;k<10;k++)
    {
    inchar = demo.read();
    if (inchar >= 0) System.out.print(" "+inchar);
    }
if (inchar < 0)
    System.out.print(" End-of-file");
    System.out.print("\n");
    }
```

We use the same object of class *File*. After all, at this point we don't know whether we're going to be writing to or reading from the file in this particular example!

```
File demofile = new File(infile);
```

Once we know that the file **exists**, we choose to read it; we must open it for reading by creating a *FileReader* object for it.

```
FileReader demo = new FileReader(demofile);
```

The most basic method of reading a file is the **read** method (specified without arguments) which reads one character at a time and passes each back to us as an integer. A **-1** is returned if we have read to the end of the stream. Thus, to read up to 10 characters and display their integer values:

```
int inchar = -1;
for (int k=0;k<10;k++)
{
inchar = demo.read();
if (inchar >= 0) System.out.print(" "+inchar);
}
```

Let's confirm that in operation:

```
seal% java bradley
File creation or display
Please enter the name of a file:fred
 47 47 32 70 105 108 101 10 End-of-file
seal% java bradley
File creation or display
Please enter the name of a file:bradley.class
 202 254 186 190 0 3 0 45 0 112
seal%
```

And to check that really is the file content ...

```
seal% od -c fred
0000000  /  /      F  i  l  e  \n
0000010
seal% od -c bradley.class | head -3
0000000 312 376 272 276  \0 003  \0  -  \0  p  \b  \0  :  \b  \0  ;
0000020  \b  \0  <  \b  \0  H  \b  \0  M  \b  \0  R  \b  \0  x 007
0000040  \0  U 007  \0  [ 007  \0  \ 007  \0  ] 007  \0  ^ 007  \0
seal%
```

You can do anything reading character-by-character ... but you have to do it. All the data manipulation must be elsewhere in your Java program, be it in your main method or in other methods you call!

The `read` method does also support reading into a character array. It reads up to the length of the array and returns the number of characters actually read. Thus:

```
FileReader demo = new FileReader(demofile);
int char_count;
char [] buffr = new char [10];
char_count = demo.read(buffr);
System.out.print(buffr);
System.out.print("\nCount read = "+char_count+"\n");
```

Figure 1 Running public class bradford

```
seal% java bradford
File creation or display
Please enter the name of a file:fred
// File
Count read = 8
seal% java bradford
File creation or display
Please enter the name of a file:bradley.class
Ëp°¿-p
Count read = 10
seal%
```

Yes, that works.<sup>1</sup>

Some files may contain non-printables, but many will contain lines of text written by methods similar to `println`, for example, your `.java` files. In other words, the files

<sup>1</sup> although notice that it's dangerous to print out the character array because it may contain any number of non-printable characters

will contain ASCII text, separated into lines with a particular delimiter.

Our base `FileReader` class does not provide a method for reading line-by-line, but `BufferedReader` does:

```

BufferedReader demo = new BufferedReader(
    new FileReader(demofile));
for (int k=0;k<10;k++)
{
    String next_line = demo.readLine();
    System.out.println(next_line);
}

// Well House Consultants 2004.
import java.io.*;

/** File Manipulation */

public class farleigh
{
    public static void main(String[] args) throws IOException
    {
        String inline;

        System.out.println("File creation or display");
        System.out.print("Please enter the name of a file: ");

        try{
            BufferedReader standard = new BufferedReader(
                new InputStreamReader(System.in));
            inline = standard.readLine();
        }
        catch (Throwable e)
        {
            System.out.print("data entry error");
            System.out.println(e);
            return ;
        }
        File demofile = new File(inline);

        if (! demofile.exists())
        {
            FileWriter demo = new FileWriter(demofile);
            demo.write("// File created by Java Class winsley\n");
            demo.close();
        }
        else
        {
            BufferedReader demo = new BufferedReader(
                new FileReader(demofile));
            for (int k=0;k<10;k++)
            {
                String next_line = demo.readLine();
                System.out.println(next_line);
            }
        }
    }
}

```

Figure 2 Running public class farleigh

```
seal% java farleigh
File creation or display
Please enter the name of a file:farleigh.java
seal% java farleigh
File creation or display
Please enter the name of a file:fred
// File
null
null
null
null
null
null
null
null
null
null
null
null
seal%
```

Other topics ...

- Zip archives
- Tokenizers
- Writing and reading back objects
- Serialization

## Exercise

Take the example program `Javafgrep.java` from the course server and work out what it does. Modify it so that it always reads the data from a file named `stdcodes.xyz`, and further modify it to report on the total number of lines on the input file.

Write a Java program which creates a file called `"my.txt"`. Into that file, write the names of all the files in the current directory that end in `".java"`, and the first line of each of those file.

### *For Advanced Students*

Enhance the above program so that

- it will exit with an error message if `my.txt` already exists
- it will correctly flag `.java` files which are not readable
- it will correctly deal with empty `.java` files in the directory

# *License*

*These notes are distributed under the **Well House Consultants Open Training Notes License**. Basically, if you distribute it and use it for free, we'll let you have it for free. If you charge for its distribution of use, we'll charge.*

### 3.1 Open Training Notes License

Training notes distributed under the **Well House Consultants Open Training Notes License** (WHCOTNL) may be reproduced for any purpose PROVIDE THAT:

- This License statement is retained, unaltered (save for additions to the change log) and complete.
- No charge is made for the distribution, nor for the use or application thereof. This means that you can use them to run training sessions or as support material for those sessions, but you cannot then make a charge for those training sessions.
- Alterations to the content of the document are clearly marked as being such, and a log of amendments is added below this notice.
- These notes are provided "as is" with no warranty of fitness for purpose. Whilst every attempt has been made to ensure their accuracy, no liability can be accepted for any errors of the consequences thereof.

Copyright is retained by Well House Consultants Ltd, of 404, The Spa, Melksham, Wiltshire, UK, SN12 6QL - phone number +44 (1) 1225 708225. Email contact - Graham Ellis (graham@wellho.net).

Please send any amendments and corrections to these notes to the Copyright holder - under the spirit of the Open Distribution license, we will incorporate suitable changes into future releases for the use of the community.

If you are charged for this material, or for presentation of a course (Other than by Well House Consultants) using this material, please let us know. It is a violation of the license under which this notes are distributed for such a charge to be made, except by the Copyright Holder.

If you would like Well House Consultants to use this material to present a training course for your organisation, or if you wish to attend a public course is one is available, please contact us or see our web site - <http://www.wellho.net> - for further details.

Change log  
Original Version, Well House Consultants, 2004

Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_  
Updated by: \_\_\_\_\_ on \_\_\_\_\_

*License Ends.*