File Input/Output in Java

Alberto Cortés <alcortes@it.uc3m.es>

March 31, 2013

Contents

Chapter 1

Introduction

1.1 Purpose of this document

This document is a brief introduction to Java file input/output.

It is targeted to first year students in the degree, with no previous experience in Java, beyond the basic knowledge about language syntax, its fundamental structures and the knowledge about object orientation typical of the first chapters of any Java programming introductory textbook.

It will briefly review the API and involved classes as well as some required operating systems and file systems. Appropriate examples complete the corresponding explanations.

Advanced concepts about file management, like channels, locks, pattern-matching, direct mapping of files to memory, multiplexing, selectors, random access, non-blocking operations, multithreading, or asynchronous access techniques are not covered in this document.

This document applies to Java version 1.7.0_17, thus the information relates to NIO.2 (*New I/O version 2*), the new file input/output system introduced in Java version 1.7 dated 2011-06-07.

Most of this document is based on The Java Tutorials, for Java version SE 7.

1.2 General context

Most of the file input/output support in Java is included in the `java.nio.file` package. Although such API includes many classes, only a few of them are actually access points to the API, which greatly simplifies its use.

1.3 File Systems

A file is an abstraction of the operating system for the generic storage of data. The part of the operating system responsible of file management is called “file system”.

Traditional file systems organise, store and name (identify with a name) the files stored in persistent storage devices, such as hard disks, USB solid state memories, DVDs…

Nowadays, well-known file systems are “ext3” for Linux operating systems, “NTFS” for systems based on Windows NT, such as Windows XP o Windows 7, and “ISO9660” y “UDF” for optical devices such as CDs y DVDs.

Although each file system offers its own vision of the data saved in a hard drive and manages and orders them its own way, all of them share some general aspects:

- Files are usually organised in hierarchical structures (trees) of directories. Such directories are file (and other directories) containers that allow organising the data in the drive

- The file name is related to its position in the directory tree that contains it, which allows not only identifying unambiguously each file but also finding it based on its name.

- The files usually have some associated metadata, such as its creation time, the time of its last modification, its owner ore the permissions that the different users have on it (read, write…). This turns the file system into a database of the contents stored in the drive that can be queried applying multiple search criteria.

Chapter 2

Paths: file names

A file is identified by its *path* through the file system.

For example, the *path* (its complete file name) of the file where I am writing this document is

```
/home/alcortes/room/current/Java-File-IO/tex/java-file-io.tex
```

In my computer there can be no other file with exactly the same path, though probably there are many files with the (short) *name* "java-file-io.tex".

Of course, the file path of a file changes depending on the file system used. For example, in Linux operating systems all the files are contained in the root directory or in directories hanging from it.

However, in NTFS-type file systems, the files are stored in a "volume" (identified by an alphabetic letter). Such volume can in turn contain files and directories. Traditionally, in the Windows environment the file names usually have an extension (typically a "." and three letters) that helps to identify the type of content stored in such file.

In both systems, all of the information needed to locate the file is contained in the path string. Let's see a couple of examples:

- Linux: `/home/alcortes/my_file`: In the root directory (`/`), there is a directory `/home/`, inside which there is a directory `/home/alcortes/` (-short- name "alcortes"), inside which there is a file `/home/alcortes/my_file`, with (short) name "my_file".
- Windows 7: `C:\Mis documentos\fichero.txt`: In the `C` volume, there is a directory `C:\Mis documentos\`, inside which there is a file `C:\Mis documentos\fichero.txt`, with (short name) "fichero.txt", which has the typical old Windows 3-letters extension ("txt").

As seen in the examples, paths have a very different format depending on the operating system. In Linux everything is under the root directory `/` and the character used to separate the directories an files (also called the `delimiter`) is `/`. Windows in turn supports multiple root nodes, each one mapping to a volume, and directories are separated with the `\` character.

It is also important to notice that each operating system has its own restrictions for other aspects of the path. For example, the FAT file system (used in MS-DOS systems) does not distinguish between uppercase and lowercase (it is case insensitive), thus the files `C:\FiChEro.TXT` and `C:\fichero.txt` are the same. It does not support file (short) names longer than 8 characters, nor characters not included in the ASCII standard (such as á, ñ, Å, ¿, α…). Most of these constraints have been solved in newer file systems.

2.1 Relative and absolute paths

Until now, we have only seen examples of absolute paths, which are those that allow identifying unambiguously a file and its location with no additional information.

A relative path specifies the path to a file from a reference directory, known as the *working directory*.

Let's see some examples:

- If the working directory is `/home/alcortes/`, the relative path `my_file` identifies the file `/home/alcortes/my_file`.
- If the working directory is `/home/alcortes/`, the relative path `my_project/my_file` identifies the file `/home/alcortes/my_project/my_file`.
- If the working directory is `/home/alcortes/`, the relative path
- Si el directorio de trabajo es `/home/alcortes/`, el path `../mcfp/my_file` identifies the file `/home/mcfp/my_file`. The `../` directory is an alias to represent the parent directory of the one it follows.
- If the working directory is `/home/alcortes/`, the relative path `./my_project/my_file` identifies the file `/home/alcortes/my_project/my_file`. The `./` directory is an alias of the current directory (the working directory).

The key for distinguishing a relative path from an absolute one consists on realising that the relative path does not start with the root of the file system (`/` in Linux, or with a volume letter in Windows, such as `C:\` for example).

2.2 Paths in Java

The `java.nio.file.Path` interface represents a path, and the classes implementing such interface can be used for locating files in the file system.

The simplest way of creating an object implementing the `Path` interface is using the `java.nio.file.Paths` class, that provides static methods that return `Path` objects based on a `String` representation of the desired path. For example:

```
Path p = Paths.get("/home/alcortes/my_file");
```

Of course, the files do not need to actually exist in the hard drive for creating the corresponding `Path` objects. The representation and management of paths in Java is not restricted by the actual existance of such files or directories in file system.

The `Path` interface declares many useful methods for managing paths, such as getting the (short) name of a file, getting its container directory, resolving relative paths, etc.

Notice that working with paths is totally unrelated to working with the content of the files they represent. For example, modifying the content of a file is an operation that is independent of its name or its location in the file system.

An instance of type `Path` reflects the naming system the subyacent. In consequence, path objects of different operating systems are not easily compared.

2.3 Example of use

PathExample.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;

class PathExample {
  public static void main(String args[]) {
    Path path = Paths.get("/home/alcortes/my_file");
    System.out.println("path = " + path);
    System.out.println("is absoute? = " + path.isAbsolute());
    System.out.println("file short name = " + path.getFileName());
    System.out.println("parent = " + path.getParent());
    System.out.println("uri = " + path.toUri());
  }
}
```

```
; javac PathExample.java
; java PathExample
path = /home/alcortes/my_file
is absoute? = true
file short name = my_file
parent = /home/alcortes
uri = file:///home/alcortes/my_file
```

Chapter 3

Files

The class `java.nio.file.Files` allows to manage files in a disk in Java.

This class has static methods to handle files, allowing us to create and delete files and directories, check if a file exists in the disk, check is permissions (reading, writting…), move a file to a different directory and reading and writing the contents of a file.

Let us see how to perform some of these simple operations first:

3.1 Checking for existence and permissions

ExistanceChecks.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;

class ExistanceChecks {
  public static void main(String args[]) {
    Path path = Paths.get("./ExistanceChecks.java");
    System.out.println("path = " + path);
    System.out.println("  exists = " + Files.exists(path));
    System.out.println("  readable = " + Files.isReadable(path));
    System.out.println("  writeable = " + Files.isWritable(path));
    System.out.println("  executeable = " + Files.isExecutable(path));

    path = Paths.get("/this_file_doesn't_exist");
    System.out.println("path = " + path);
    System.out.println("  exists = " + Files.exists(path));
    System.out.println("  readable = " + Files.isReadable(path));
    System.out.println("  writeable = " + Files.isWritable(path));
    System.out.println("  executeable = " + Files.isExecutable(path));
  }
}
```

```
path = ./ExistanceChecks.java
  exists = true
  readable = true
  writeable = true
  executeable = false
```

```
path = /this_file_doesn't_exist
  exists = false
  readable = false
  writeable = false
  executeable = false
```

3.2 Creating and deleting files

CreateOrDelete.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Creates a new file or delete it, if it already exists
class CreateOrDelete {
  private static void usage() {
    System.err.println("java CreateOrDelete <file>");
    System.err.println("  The <file> argument is required.");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      if (Files.exists(path))
        Files.delete(path);
      else
        Files.createFile(path);
    } catch (IOException e) {
      System.err.println(e);
      System.exit(1);
    }
  }
}
```

```
; java CreateOrDelete
java CreateOrDelete <file>
  The <file> argument is required.
; ls
CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete bla
; ls
bla  CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete bla
; ls
CreateOrDelete.class  CreateOrDelete.java
; java CreateOrDelete /root/bla
java.nio.file.AccessDeniedException: /root/bla
```

Chapter 4

Reading from a file

There are several ways to read a file in Java.

Small files can be read in a single invocation and stored in a byte array or a string.

For bigger files, this can be very inefficient in terms of memory usage, we would rather read them in chunks and process each chunk as they come by. This can be done by the means of a *buffer* (explained later in this document), which can also help in using the hard disk efficiently.

4.1 Character Sets (charsets)

A charset is an association between numbers and letters. The charset purpose is to assign a number to each letter, so you can store letters as their corresponding numbers, and retrieve them later by translating those numbers back to the original letters.

Charset are needed because a hard disk can only store (binary) numbers, not letters. Therefore, if we want to write a text file to disk, we must translate each of its letters to numbers, and store those numbers instead.

Some years ago, the most used charset was ASCII, shown bellow:

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL	32	SPACE	64	@	96	‘
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	’	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL
Dec	Char	Dec	Char	Dec	Char	Dec	Char

These means that if we want to store an ’a’ letter to disk using ASCII, we must store a 97 instead. Reading a 97 from disk, means we are reading a ’a;, if we use ASCII.

Please, note that the ASCII table do not include some characters from Spanish, as accented letters or the ’ñ’.

There are many charsets available, for example, until very recently ISO-8859-1 (latin-1) was commonly used in Spanish speaking countries and nowadays UTF-8 is widespread adopted.

When storing a text file in a disk, no information about the chosen charset is stored in it. This means that when trying to read it back, you can get the wrong characters if you do not use the same charset for reading that the one used for writing.

A nice solution to this problem is to use the default charset in Java.

The abstract class `java.nio.charset.Charset` is used in Java to control which charset we want to use in our programs. Being an abstract class, you can not instantiate it directly, but can name particular charsets whenever you need them. As an example, the following Java program will print some common charsets that can be used in Java and configure the JVM to use for different purposes.

CharsetExample.java

```java
import java.nio.charset.Charset;

class CharsetExample {
  public static void main(String args[]) {
    // find default charset
    System.out.println("Default Charset = " + Charset.defaultCharset());

    // Use Latin1 for file i/o instead of the default charset
    System.setProperty("file.encoding", "ISO-8859-1");
    System.out.println("file.encoding = " + System.getProperty("file.
         encoding"));

    // Example of directly using charset objects
    Charset ascii = Charset.forName("US-ASCII");
    System.out.println("Standard charset in old systems = " + ascii);
  }
}
```

```
; javac CharsetExample.java
; java CharsetExample
Default Charset = UTF-8
file.encoding = ISO-8859-1
Standard charset in old systems = US-ASCII
```

4.2 Reading to a byte array

This is quite simple, you name a file and you get a byte array filled with the contents of the file. Of course this can be very memory inefficient and besides, you will hardly need all the file at once.

It may be hard to process a text file in this way as the translation between bytes and characters is not a simple matter (remember charsets ?). You will have to build your own chars and lines by reading each member of the array and turning them into chars and end-of0line characters. As we will see later, there are better ways to do this.

But this way of reading files is very nice for small files (a couple of sentences) or for small binary files, where reading a byte at a time is the desired behaviour.

The following program writes to standard output the contents of file passed as its argument.

```
; javac Cat1.java
; java Cat1 /tmp/bla
ERROR: java.nio.file.NoSuchFileException: /tmp/bla
; java Cat1 Cat1.java
```

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// prints the contents of a file using an array of bytes
class Cat1 {
  private static void usage() {
    System.err.println("java Cat1 <file>");
    System.err.println("  A <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      byte[] content = Files.readAllBytes(path);
      for (int i=0; i<content.length; i++)
        System.out.print((char) content[i]);
      } catch (IOException e) {
        System.err.println("ERROR: " + e);
        System.exit(1);
    }
  }
}
```

The following program counts lines in a file.

CountLines1.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;

// Count (UNIX) lines in a file
class CountLines1 {
  private final static char UNIX_NEWLINE = '\n';

  private static void usage() {
    System.err.println("java CountLines <file>");
    System.err.println("  The <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    long count = 0;
    try {
      byte[] content = Files.readAllBytes(path);
      for (int i=0; i<content.length; i++)
        if ((char) content[i] == UNIX_NEWLINE)
          count++;
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
    System.out.println(count);
  }
}
```

```
; javac CountLines1.java
; java CountLines1 CountLines1.java
33
```

Lectura a una lista enlazada de `Strings`

The method `java.nio.file.Files.readAllLines()` can be very useful for certain text processing. As reading to a byte array, it reads all the file at once, which can be a problem. But it returns a linked list of strings, one per line, which can be very nice for small text files.

4.3 Buffers

A buffer is a data structure to manage chunks of data from a (bigger) collection of data.

Buffers are useful to prevent storing large data in memory at once, accessing such data in chunks instead.

They can also be useful for efficiency purposes, the applications no longer care about how is the best writing ratio to the hardware device, it just write to the buffer whenever it wants and the buffer takes care of writing to disk at the right moments for maximum disk efficiency.

4.4 Reading to a buffer

The class `java.io.BufferedReader` is used to read text files and process them. It allows to efficiently read chars, arrays or whole lines as strings.

Each read from a `BufferedReader` will return data from the corresponding file, in order, the `BufferedReader` keeps track of the amount of data previously read.

The method `readLine()` will read a whole line of text from the file and return it as an `String`.

The following program writes to standard output the contents of a file passed as it argument.

```
; javac Cat2.java
; java Cat2 Cat2.java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// prints the contents of a file using a BufferedReader
class Cat2 {
  private static void usage() {
    System.err.println("java Cat2 <file>");
    System.err.println("  A <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    try {
      BufferedReader reader =
        Files.newBufferedReader(path, Charset.defaultCharset());
      String line;
      while ( (line = reader.readLine()) != null )
        System.out.println(line);
      reader.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

This program is clearly more efficient than the previous one, *Cat1*, as we are using buffers now:

```
; time java Cat1 Cat2.java > /dev/null

real  0m0.101s
user  0m0.088s
sys 0m0.012s
; time java Cat2 Cat2.java > /dev/null

real  0m0.086s
user  0m0.064s
sys 0m0.012s
```

The following Java program count lines in a file using a buffer.

CountLines2.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// Count (UNIX) lines in a file
class CountLines2 {
  private static void usage() {
    System.err.println("java CountLines2 <file>");
    System.err.println("  A <file> argument is mandatory");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 1)
      usage();

    Path path = Paths.get(args[0]);
    long count = 0;
    try {
      BufferedReader reader =
        Files.newBufferedReader(path, Charset.defaultCharset());
      while ( reader.readLine() != null )
        count++;
      reader.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
    System.out.println(count);
  }
}
```

```
; javac CountLines2.java
; java CountLines2 CountLines2.java
34
```

Chapter 5

Writing Files

5.1 Access modes, the `OpenOptions` parameter

While writing files in Java we can restrict the permissions we have over to file, thanks to the operating system, for extra protection or easiest handle of the files.

As an example: if the user has read and write access over a file, but our Java application only wants to write in it, we can "open" the file in Java in the "writing" mode, that will restrict how the application use the file.

To this end and for some other extra functionalities, there are some access methods for files defined in Java, through the `OpenOptions` parameter. The easiest way of using this parameter is by using the members of **`enum`** `StandardOpenOptions`, that has the following values (among others):

- `WRITE`: allows writing to the file
- `APPEND`: start writing at the end of the file (keep the current contents)
- `CREATE_NEW`: create a new file or throw an exception if it already exists
- `CREATE`: create an new file or just open it if it already exists
- `TRUNCATE_EXISTING`: if the file exists and has contents, start writing from the beginning of the file, overwriting the old contents and deleting any remaining old content.

The following methods use this access modes, you can find what each method do if the access parameters are not explicitly used in its invocation by reading each method documentation.

5.2 Writing from a byte array

This is the simplest (and more limited) way of writing files. Use the method `java.nio.file.Files.write()` as in the following examples:

Cp1.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.file.StandardOpenOption;

// Copy a file
class Cp1 {
  private static void usage() {
    System.err.println("java Cp1 <input file> <output file>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path inputFile = Paths.get(args[0]);
    Path outputFile = Paths.get(args[1]);

    try {
      byte[] contents = Files.readAllBytes(inputFile);
      Files.write(outputFile, contents,
                  StandardOpenOption.WRITE,
                  StandardOpenOption.CREATE,
                  StandardOpenOption.TRUNCATE_EXISTING);
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac Cp1.java
; ls
Cp1.class  Cp1.java
; java Cp1 Cp1.class bla
; diff -sq Cp1.class bla
Files Cp1.class and bla are identical
```

5.3 Writing through buffers

As with reading files, writing files using buffers is more efficient.

The following Java program copies files, using buffers for reading and writing, one line at a time.

Cp2.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.nio.file.StandardOpenOption;

// Copy a file
class Cp2 {
  private static void usage() {
    System.err.println("java Cp2 <input file> <output file>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path input = Paths.get(args[0]);
    Path output = Paths.get(args[1]);

    try {
      BufferedReader inputReader =
        Files.newBufferedReader(input, Charset.defaultCharset());
      BufferedWriter outputWriter =
        Files.newBufferedWriter(output, Charset.defaultCharset(),
                                StandardOpenOption.WRITE,
                                StandardOpenOption.CREATE,
                                StandardOpenOption.TRUNCATE_EXISTING);

      String line;
      while ( (line = inputReader.readLine()) != null ) {
        outputWriter.write(line, 0, line.length());
        outputWriter.newLine();
      }

      inputReader.close();
      outputWriter.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac Cp2.java
; java Cp2 Cp2.class bla
ERROR: java.nio.charset.MalformedInputException: Input length = 1
; java Cp2 Cp2.java bla
; diff -sq Cp2.java bla
Files Cp2.java and bla are identical
```

5.4 More advanced examples

The following Java program, reads a file, ignores those lines with lowercase characters and prints to standard output the rest of them:

CopyUpperCase.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.nio.file.StandardOpenOption;

// Copy uppercase lines of file
class CopyUpperCase {
  private static void usage() {
    System.err.println("java CopyUpperCase <input file> <output file>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path input = Paths.get(args[0]);
    Path output = Paths.get(args[1]);

    try {
      BufferedReader inputReader =
        Files.newBufferedReader(input, Charset.defaultCharset());
      BufferedWriter outputWriter =
        Files.newBufferedWriter(output, Charset.defaultCharset(),
                                StandardOpenOption.WRITE,
                                StandardOpenOption.CREATE,
                                StandardOpenOption.TRUNCATE_EXISTING);

      String line;
      while ( (line = inputReader.readLine()) != null ) {
        if (line.equals(line.toUpperCase())) {
          outputWriter.write(line, 0, line.length());
          outputWriter.newLine();
        }
      }

      inputReader.close();
      outputWriter.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac CopyUpperCase.java
; cat test
This line has lowercase letters.
THIS LINE IS ALL IN UPPERCASE.
THIS LINE TOO.
this line is not in uppercase.
this LINE is NOT in uppercase.
THIS LAST LINE IS ALL IN UPPERCASE.
; java CopyUpperCase test output
; cat output
THIS LINE IS ALL IN UPPERCASE.
THIS LINE TOO.
THIS LAST LINE IS ALL IN UPPERCASE.
```

The following Java program prints the lines of a file that match a given pattern:

Grep.java

```
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedReader;

// Search for a text in a file
class Grep {
  private static void usage() {
    System.err.println("java Grep <input file> <pattern>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 2)
      usage();

    Path input = Paths.get(args[0]);

    try {
      BufferedReader inputReader =
        Files.newBufferedReader(input, Charset.defaultCharset());

      String line;
      long lineNumber = 1;
      while ( (line = inputReader.readLine()) != null ) {
        if (line.contains(args[1]))
          System.out.println(lineNumber + ": " + line);
        lineNumber++;
      }

      inputReader.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    }
  }
}
```

```
; javac Grep.java
; java Grep Grep.java import
1: import java.nio.file.Path;
2: import java.nio.file.Paths;
3: import java.nio.file.Files;
4: import java.io.IOException;
5: import java.nio.charset.Charset;
6: import java.io.BufferedReader;
; java Grep Grep.java line
25:       String line;
26:       long lineNumber = 1;
27:       while ( (line = inputReader.readLine()) != null ) {
28:         if (line.contains(args[1]))
29:           System.out.println(lineNumber + ": " + line);
30:         lineNumber++;
; java Grep Grep.java (
bash: syntax error near unexpected token `('
; java Grep Grep.java "("
10:   private static void usage() {
11:     System.err.println("java Grep <input file> <pattern>");
12:     System.exit(1);
15:   public static void main(String args[]) {
16:     if (args.length != 2)
17:       usage();
19:     Path input = Paths.get(args[0]);
23:         Files.newBufferedReader(input, Charset.defaultCharset());
27:       while ( (line = inputReader.readLine()) != null ) {
28:         if (line.contains(args[1]))
29:           System.out.println(lineNumber + ": " + line);
33:       inputReader.close();
34:     } catch (IOException e) {
35:       System.err.println("ERROR: " + e);
36:       System.exit(1);
```

Chapter 6

Binary Files

Reading short binary files can be performed by reading the whole file to a byte array, like in 4.2. But reading large binary files by such means is inefficient in terms memory usage.

You would like to use a buffer in such cases, like `BufferedReader` (4.4) does, but this class translates bytes to `String`s and characters, being quite difficult to translate them back to the original byte contents.

What you really need is a buffered access to the file without the extra feature of byte to string translation. A nice solution is to use `java.io.BufferedInputStream`, and its method `read(byte[] b, int off, int len)` that allows you to read efficiently the desired amount of bytes into an array.

Writing binary files has the same problem as reading, and there is also a `java.io.BufferedOutputStream` class to solve it. The method `write(byte[] b, int off, int len)` can be used to write to the buffer, delegating to the JVM and the operating system the complex task of writing the data efficiently to the disk.

The following program copies binary files (or any other file for that matter), by reading and writing bytes arrays of the desired size.

Dd.java

```java
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files;
import java.io.IOException;
import java.nio.charset.Charset;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.nio.file.StandardOpenOption;

// Copy files using binary buffers
class Dd {
  private static void usage() {
    System.err.println("java Dd <input file> <output file> <buffer size>");
    System.exit(1);
  }

  public static void main(String args[]) {
    if (args.length != 3)
      usage();

    Path inputPath = Paths.get(args[0]);
    Path outputPath = Paths.get(args[1]);

    try {
      int bufferSize =  Integer.parseInt(args[2]);
      if (bufferSize <= 0)
          throw new NumberFormatException(args[2] + " is not positive");

      BufferedInputStream input;
      BufferedOutputStream output;
      input  = new BufferedInputStream(
                    Files.newInputStream(inputPath,
                      StandardOpenOption.READ));
      output = new BufferedOutputStream(
                    Files.newOutputStream(outputPath,
                      StandardOpenOption.WRITE,
                      StandardOpenOption.CREATE,
                      StandardOpenOption.TRUNCATE_EXISTING));

      byte[] buffer = new byte[bufferSize];
      int bytesRead = input.read(buffer, 0, bufferSize);
      while ( bytesRead >= 0 ) {
        output.write(buffer, 0, bytesRead);
        bytesRead = input.read(buffer, 0, bufferSize);
      }

      input.close();
      output.close();
    } catch (IOException e) {
      System.err.println("ERROR: " + e);
      System.exit(1);
    } catch (NumberFormatException e) {
      System.err.println("ERROR: Bad number format: " + e);
      System.exit(1);
    }
  }
}
```

```
; # create a 10 MB file with random contents
; dd if=/dev/urandom of=/tmp/input bs=100000 count=100
100+0 records in
100+0 records out
10000000 bytes (10 MB) copied, 1.35976 s, 7.4 MB/s
;
;
; # copy the file using a 1024 bytes buffer
; java Dd /tmp/input /tmp/output 1024
; diff -sq /tmp/input /tmp/output
Files /tmp/input and /tmp/output are identical
;
;
; # copy the file one byte at a time, this is slow
; # even if we use buffers!!
; time java Dd /tmp/input /tmp/output 1

real  0m1.168s
user  0m1.100s
sys 0m0.060s
;
;
; # copy the file using a 1024 byte buffer, this is much faster
; # in user time, but can still be slow on real time
; time java Dd /tmp/input /tmp/output 1024

```

```
real  0m0.168s
user  0m0.120s
sys 0m0.032s
;
;
; # copy the file using a 1MBi byte buffer, this is waaay faster
; time java Dd /tmp/input /tmp/output 1048576

real  0m0.114s
user  0m0.068s
sys 0m0.044s
```