Contents

0 Introduction

A fundamental operation of computer applications is that of reading from and writing to files. This allows us to create permanent records of our activity on the disk and to reopen it later for further editing or use. We will concern ourselves chiefly with two major types of files: text files and files that serialize, or pack away, objects which can later be deserialized, or unpacked.

We will begin, in this chapter, with text files, and do a case study of creating a simple text editor that opens text files for editing in a simple GUI. Along the way we shall meet a new and very useful widget, the `JFileChooser`. Common to both of these streams is the `File` class; we shall begin our exposition with that. This will lay the groundwork for the case study in Chapter 10, which is a full-featured text editor.

1 The File Class and Paths

During this chapter we will become acquainted with a new package, `java.io`, which deals with file input/output operations, or file IO, for short.

The File class belongs to this package; it is used to represent locations in your file system. The File class does not play a role in the actual reading or writing of data to a file. Instances of the `File` class can point at files or directories stored on your system. There are other objects that handle the actual mechanics of file IO.

Let us explore this class and see what it does. Open a DrJava interactive session, and the Java API guide. Let us begin by looking at the Field Summary in the guide. It features four fields. Really, they harbor two pieces of data, the path separator and the separator character. This character can be yielded up as a character or a string. Hence the existence of four constants. Notice that these constants are static, so we can call them by class name.

```
> import java.io.File;
> File.pathSeparator
":"
> File.pathSeparatorChar
':'
> File.separator
"/"
> File.separatorChar
'/'
>
```

The purpose of the separator character is to separate files in a path. This character is a on Windoze systems, and it is a / on a UNIX system. The notion of path is common to all operating systems. Recall a path consists of a sequence of directories followed by a directory or file. The separator character can be expanded to "and then into." Only the last item in a path can be a regular file; all others must be directories.

For example the path

```
animals/mammals/tapir.html
```

specifies a file tapir.html that lives inside of directory mammals, which in turn lives inside of directory animals. In Windoze, this path is specified by

```
animals\mammals\tapir.html
```

Common to the command line interfaces of Windoze and UNIX is the notion of `search path`. In UNIX if you enter the command `ls`, it not in your `cwd`. Therefore UNIX checks your search path, which is a list of paths to directories for the presence of `ls`. It checks this list in order; if it finds the command in some directory, it immediately executes it. Since `ls` lives in `/bin`, the directory `/bin` must be in your path for `ls` to run. Fortunately, this is done for you by default. Windoze also has a search path mechanism that works in an identical way. Let us show the path on both systems. First on UNIX, we see the path by entering `echo $PATH`at the command prompt.

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
$
```

In Windoze, bring up a console window by going to the run menu item in the Start menu, and type `cmd` into the text slot. Then, in this little black window, type `PATH` at the prompt.

```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\John Morrison>PATH
PATH=C:\WINDOWS\system32;C:\WINDOWS;C:\WINDOWS\System32\Wbem
```

In both systems, there is an environment variable containing the search path. In UNIX this is `$PATH`, in Windoze it is `PATH`. You will also notice that the search path is a list of absolute paths. Absolute paths in Windoze begin with a drive letter such as `C:`; in UNIX they begin with a `/`. Observe that the directory `/bin` is listed on the path of this machine, so ls there is found and run. If you type an command that does not live in your path, you will get an error message, as seen here.

```
$ ontv
bash: ontv: command not found
$
```

We sent UNIX on a wild goose chase and we get rewarded with a nastygram. If you are running a MAC you will see the same things. In Windoze, you will see this, informing us it has been sent on a similarly futile wild goose chase.

```
C:\Documents and Settings\John Morrison>ontv
'ontv' is not recognized as an internal or external command,
operable program or batch file.
```

We see that the path separator, which is : in UNIX and ; in Windoze, separates directories in the search path.

2 Constructors and Methods

We shall most commonly use the constructor

```
public File(String pathname)
```

to create instances of `File` objects. Such an object may point at a directory or a regular file. This is not surprising to UNIX users, since they know that directories are just special files that contain an index to their contents. You may use an absolute path name, or a relative path name. A relative path name will be relative to the `cwd` of the java program when it is running on the user's machine.

The `exists()` method can be used to see if a given file already exists on the user's system. Here we show a brief example.

```
> f = new File(".");
> f.exists()
true
> g = new File("someFileThatDoesNotExist")
> g.exists()
false
>
```

We began by making f point at the Java program's `cwd`. Naturally, this must exist. We then deliberately chose a file that does not exist in the program's cwd, and we see that exists() discerns its nonexistence. This method can be very useful for performing file IO; you can use it to avoid clobbering an existing (valuable) file on the (hapless) user's system.

The `canRead()`, `canWrite()`, and `canExecute()` methods are self-explanatory. They come in handy: you can check to see if you have permission for gaining access to a file prior to charging forth. This can save the throwing of an exception.

The methods `getPath()` and `getAbsolutePath()` will return the string representation of the path to a file from the program's `cwd`. You should make a directory, place a few unvaluable files in it, and experiment with the methods in the API. You can remove files, make new directories, and do all manner of file management with this class. Do the simple exercises below in a program or in an interactive DrJava session.

Programming Exercises

1. Make a new `File` object and use it to determine the absolute path of your `cwd`.
2. Perform ls -l on your cwd. Try resetting permission bits on one of your files by creating a file object. Verify what you did using ls -l, and by using methods from the `File` class.
3. Make a `File` object in the interactive prompt and change its `cwd` to various places. Check for existence and nonexistence of various files. See if you can determine what permissions you have for the files. Can you check if a file is a directory?
4. See if you can write a program that takes a file or directory as command line argument and which imitates the action of the UNIX command `ls`.

3 A Simple Case Study: Copying a File

During this section you will see how to read from and write to a text file. We shall emulate the action of the cp command in UNIX. The usage for our program will be

```
$ java Copy donorFile recipientFile
```

and its action will be to copy the contents of the donor file into the recipient file. It will clobber any recipient file that already exists, just as the UNIX `cp` command does. You may enter this command in the DrJava interactions pane to execute it as well as at the command prompt.

Let us begin by creating the class `Copy`. We will make our method be a static method inside of this class. Here is the start.

```
public class Copy
{
    public static void copy(String donor, String recipient)
    {
    }
}
```

This file compiles happily. Now, inside of the `copy` method, add this code.

```
    donorFile = new File(donor);
    recipientFile = new File(recipient);
```

Also, add this import statement at the top of the program.

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

to avoid angry yellow. The resulting code will compile. Our `File` objects just point to paths in the file system, which might or might not exist. Next, let us open the donor file for reading and the recipient file for writing. To do so, we use a `FileReader` as follows.

```
FileReader fr = new FileReader(donorFile);
```

and a `FileWriter` as follows.

```
FileWriter fw = new FileWriter(donorFile);
```

Take a trip to the API guide for the `FileReader` class. We are using the constructors

```
FileReader(File file)
```

and

```
FileWriter(File file)
```

to create the code above.

Next, we will read each line from the donor file, then write them to the recipient file. Adding in the appropriate exception handling yields the following class. Notice how the `copy` method passes the buck and forces the caller to handle any exception it generates.

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

public class Copy
{
    public static void copy(File donorFile, File recipientFile)
       throws IOException
    {
        FileReader fr = new FileReader(donorFile);
        FileWriter fw = new FileWriter(recipientFile);
    }
    public static void main(String[] args)
    {
        try
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

3.1 A Programming Idiom

We now need to read the contents of the donor file. We shall read it, a line at at time, and in turn write each line to the donor. You might think, "How do I iterate through a file? It was so easy in Python with a `for` loop!" Here is an idiom that does exactly that.

```
int ch;
while( (ch != fr.read()) != -1)
{
    //Process each character of the file.
}
```

What we are doing here is to write each character of the donor to the recipient in the loop. We next close each file so they are properly saved and so system resource are liberated.

```
int ch;
while( (ch = fr.readLine()) != -1)
{
    fr.write(ch);
}
fw.close();
fr.close();
```

Putting it all together we get this class.

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

public class Copy
{
    public static void copy(File donorFile, File recipientFile)
        throws IOException
    {
        FileReader fr = new FileReader(donorFile);
        FileWriter fw = new FileWriter(recipientFile);
        int ch;
        while( (ch = fr.read()) != -1 )
        {
            fw.write(ch);
        }
        fw.close();
        fr.close();
    }
    public static void main(String[] args)
    {
        try
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

Now run this at the command line as follows.

```
$ java Copy donorFile recipientFile
```

3.2 Buffered FileIO

When you read a character from a file using a `FileReader`, you are making a request of the operating system to see in that file and fetch that character. This is a fairly costly proceeding. There are times when you are programming with devices such as terminals when you want to do this. However, with what we are doing with text files, this sort of character-by-character retrieval is unnecessary and wasteful of system resources.

We make our application much faster by using *buffered* fileIO. A buffer is simply a temporary storage space. Your refrigerator acts as a buffer. Periodically, you go to the grocery store, fetch what you need and store it in the 'fridge. This saves time and money since you do not have to go to the store every time you need a food item. Your fridge is, effectively, a food intake buffer. You also likely have a recycling bin in the garage. You place recyclables in the bin, which is either periodically collected or which you periodically empty at the recycling center as it fills. Buffers ease the transfer of stuff.

Java has two standard library classes for buffered fileIO, `java.io.BufferedReader` and `java.io.BufferedWriter`. These fetch bytes from a file one disk sector (usually 4K) at a time, and then you can read from the buffer. When the buffer empties, another request is made to the operating system to refill it, until you come to the end of the file. All of this happens behind the scenes, so you need not worry about it.

Here is a code snippet that creates a buffered reader.

```
BufferedReader bf = new BufferedReader(new FileReader(someFile));
```

You pass the buffered reader a file reader that is connected to some file. As the buffered reader reads from the file, it can throw various IOExceptions. Said exceptions must be handled.

Analogously, a buffered writer is created as follows.

```
BufferedWriter bw = new BufferedWriter(new FileWriter(someFile));
```

We will now create a new class for copying files that uses buffered reading and writing.

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.FileNotFoundException;
import java.io.IOException;

public class BufferedCopy
{
    public static void copy(File donorFile, File recipientFile)
       throws IOException
    {
        String line;
        BufferedReader fr = new BufferedReader(new FileReader(donorFile));
        BufferedWriter fw = new BufferedWriter(new FileWriter(recipientFile));
        while( (line = fr.readLine()) != null)
        {
            fw.write(line + "\n");
        }
        fr.close();
        fw.close();
    }
    public static void main(String[] args)
    {
        try
        {
            copy(new File(args[0]), new File(args[1]));
        }
        catch(FileNotFoundException ex)
        {
            System.err.println(args[0] + " not found.");
        }
        catch(IOException ex)
        {
            System.err.println("IOException!");
        }
    }
}
```

Notice that when using `write` we had to furnish an end-of-line character. This is because the `readLine()` method strips off the end-of-line character. This was not a worry in unbuffered IO since all characters, including end-of-line characters are extracted from the donor and put to the recipient.

We show a performance comparison. You can see that the buffered version is quite a bit faster. We created a file called `megazero.txt` which contains 12500 lines, each containing 79 zeroes and one newline character. We remove the recipient file in between the two tests to ensure "fairness."

```
$ time(java Copy megazero.txt foo.txt)
real    0m0.552s
user    0m0.640s
sys 0m0.060s
$ rm foo.txt
$ time(java BufferedCopy megazero.txt foo.txt)
real    0m0.225s
user    0m0.260s
sys 0m0.040s
$
```

Programming Exercises Now it's time to write some programs and practice what you have seen.

1. Write a program called `Cat.java` that takes a list of regular files as arguments and which puts them to `stdout` *in seratum*.
2. Write a program called `Ls.java` that lists the files in the directory passed it as a command–line argument in long format.
3. Write a program called `Grep.java` that takes a string and a filename as command–line arguments, and which prints all lines containing the string in the first command-line argument.