I/O

1 Input/Output (I/O)

Computer programs are just processors of data. They take some data as input, they return some data as output. Up to now, all the data that our little programs have processed was either provided by a human user or already included in the program. In the real world, however, the most usual sources of data are computers; either the computer where the program is running (local files), a computer in the vicinity (a database server), or a remote computer (remote resources through the net).

We are going to learn how to use the first of these three sources of information: how to read from the local disk and how to write to the local disk. Next week we will learn a bit about network programming, and next term we will learn about database access.

1.1 The basics

1.1.1 File systems

Computers have different levels of memory, but there is a clear separation between primary and secondary memory. Primary memory (usually referred to as simply "memory" or "RAM") faster but requires electricity to run; if the computer is switched off, the contents of the memory are lost. Secondary memory (usually referred to as "disk") is slower, but its contents are persistent, i.e. they are still there when the computer is switched off and on again. Secondary memory is usually implemented in the form of a hard disk or so-called flash memories. It is usually referred to as "disk", regardless of the actual technology used for it.

Data is stored in secondary memory in files[1] The file system is a subsystem of the operating systems that keeps all your files in place, and provides a way of accessing them. Usually this is done by means of hierarchies: there is one folder/directory at the top level of the hierarchy (called the *root*), which contains some files and subdirectories, each of these subdirectories may contain files and/or subdirectories, etc. In order to access a file `taxForm2012.odt` inside a folder `taxes`, inside a folder `MyStuff`, you may access it with a route like:

```
/home/john/MyStuff/taxes/taxForm2012.odt                    (unix)
c:\Documents and Settings\john\MyStuff\taxes\taxForm2012.odt (windows)
./MyStuff/taxes/taxForm2012.odt
MyStuff/taxes/taxForm2012.odt
```

[1] This name, as many others, comes from an ancient era where files, archives, directories, and folders were physical objects that contained documents or data.

Different operating systems use different conventions for the root of the tree of subdirectories and the separation of levels in the hierarchy. Unix systems (e.g. Linux, MacOS/X) use a single root ("/") for the whole filesystem, and separate directories using a slash ("/"). Windows use different roots, one per physical device or partition (C:, D:, etc) and separate directories with a backslash ("\").

File routes that start at the root are called *absolute* and those that do not are called *relative* (because they are relative to the current directory). Absolute routes in Unix always start with "/", but in Windows they can start in different ways (e.g. "`C:\`", "`D:\`", "\", etc) because there are many roots. Relative routes may start with a dot (meaning "current directory") but they usually start with the name of a file or directory.

1.1.2 The I/O Process

The process of reading from / writing to an external source always follows the same sequence of steps:

1. Open the resource (e.g. file). If it cannot be opened, finish.
2. Read from and/or write to the resource.
3. Close the resource.

This process is the basis of all interaction with external data sources, including files, databases, and remote resources[2]. It is *very important to close the resource* (file, database connection, remote connection / socket) at the end. Otherwise, the resource may not be used by other programs (or even by your same program in some situations). In a way, closing a data source is like releasing memory that you no longer used. Unfortunately, it is not possible to create something like a garbage collector for data sources, so programmers have to close their data sources manually.

1.2 File names in Java and the File class

File names are represented in Java by the class `File` in package `java.io`. It is important to note that Java uses the Unix tradition of considering (almost) everything a file: source code, letters, and spreadsheets are files, but so are directories too. Therefore, an object of class `File` can represent the name of an actual file or the name of a directory.

This class implements a lot of methods that are useful when interacting with the local disk. Some of the most commonly used include: `createNewFile()`, `delete()`, `exists()`, `isDirectory()`, `isFile()`, `length()`, `list()` (lists all files in a directory), and `mkdir()` (creates a directory). There are many other useful methods, as you can see on the JavaDoc of class `File`[3], and most of them have quite self-descriptive name.

Using the `File` class to get a pointer to a file (or directory) on disk is really easy:

```
String filename = "filename.txt";
File file = new File(filename);
```

[2] As we will see, in the case of network resources this process is usually hidden from the programmer.

[3] As you have done many times in the past weeks, you can find the JavaDoc of this class by searching for "java file". Usually the first link will be the documentation of the class.

The name of the file (`filename`) can be any valid route, e.g. `/home/john/file.odt`. However, there are two things that a Java developer should keep in mind to make sure that Java programs can run on any computer:

Slash as separator You should use slash ("/") as a separator, as it works in both Unix and windows systems. For extra security, you can use the static final field `File.separator` that always has the right value for the operating system the Java Virtual Machine is running on. Do not use names like `.\myFile.txt`; use either `./myFile.txt` or —to make sure it works in all computers—:

```
"." + File.separator + "myFile.txt"
```

Relative filenames: Your filenames should always be *relative*, rather than absolute. Absolute routes will not work among different operating systems. For example, `myFile.txt` and `./myStuff/myFile.txt` are fine, but `c:\myStuff\file.txt` is not.

1.3 Reading from and writing to files

Once we have a pointer to a file (through its name) we can try opening it as we said in Section 1.1.2. Note that the file may or may not exist. The `File` object is only a reference to the name. If we want to know whether the file actually exists, we can call the `exists()` method.

`File` objects cannot be directly opened in Java. Instead, an input source of data (a `Reader`) can be created with a file and it is this source that is opened (and closed at the end) to read. Conversely, an output sink of data (a `Writer`) can be created with a file, opened, written to, and closed.

1.3.1 Readers

`Reader` is the abstract class from which all other reader classes descend, and it defines methods to read characters, either one by one or many in one go. There are three main reader classes, each of them with a different purpose:

FileReader: The class for reading from text files[4] (including common formats like CSV and XML (and XML-based formats like OpenDocument).

BufferedReader: A general-purpose reader that reads characters from a character input stream, buffering them for efficiency. This means that you do not need to read character by character. It provides useful methods like `readLine()`[5].

StringReader: A string-based reader that uses a string as the input source of data. Can be useful if the data is already stored in a String instead of being read from a file or network connection.

[4] For binary files you use a FileInputStream.

[5] Although the name is the same, and the functionality is quite similar, this method has nothing to with the readLine() method of class Console, the one that we use when we execute System.console().readLine().

Opening a data source A `Reader` is automatically open at creation (with `new`). There is no need to explicitly open it with a method call.

The usual idiom to create an input source combines a `FileReader` (to read from a `File`) with a `BufferedReader` (to have access to buffered data input and the `readLine()` method). If a `FileReader` is directly used (without buffering), the performance of the application may suffer.

```
File file = new File("here/be/the/route/to/the/file");
BufferedReader in = new BufferedReader(new FileReader(file));
```

Creation of a `FileReader` may throw an `FileNotFoundException`, and this is a checked exception. This means that the code above must be placed inside a `try/catch` construct or the method must declare that it `throws` this exception.

Reading from the source Once the data source is opened, we can read from it line by line using `readLine()`. When we reach the end of the file, `readLine()` returns `null`.

Usually text files are read line by line, and something is done for every line. This looks like a situation to use a `while` loop, as in this example:

```
String line = in.readLine());
while (line != null) {
    // do things here with the line...
    line = in.readLine();
}
```

This repeats the `readLine()` call, so we may think of using a `do...while()` loop:

```
do {
    String line = in.readLine());
    // do things here with the line...
} while (line != null) {
```

... but this is wrong. Sooner or later the end of the file will be reached and a `NullPointerException` will be thrown. So we need to use a `while` loop, but we can write it a bit neater (although it seems confusing at first):

```
String line;
while ((line = in.readLine()) != null) {
    // do things here with the line...
}
```

This is the usual idiom for reading lines from text files in Java. Although mixing assignment and conditional clauses in the same line is usually discouraged because it can be confusing, this idiom is so common that everybody uses it without any confusion.

Closing the source When we have finished reading from the file, we must close the data source. This is easily done with the `close()` method.

```
in.close();
```

Alert readers may have noticed that this poses a small problem. Look at the following code:

```
01    File file = new File("file.csv");
02    try {
03        BufferedReader in = new BufferedReader(new FileReader(file));
04        String line;
05        while ((line = in.readLine()) != null) {
06            // ... do things with the data here
07        }
08        in.close();
09    } catch (FileNotFoundException ex) {
10        System.out.println("File " + file + " does not exist.");
11    } catch (IOException ex) {
12        ex.printStackTrace();
13    }
```

If a `FileNotFoundException` is thrown on line 03 nothing bad happens; it is dealt with at the appropriate catch block[6]. But what happens if an `IOException` is thrown at line 05 or inside the loop? The execution flow will jump to the catch clause on line 11, skipping line 08 and the data source will never be closed!

- We cannot move line 08 out of the `try` block because `in` is defined inside that scope.
- We cannot move line 08 to the `catch` block because that means `in` will not be closed if no exceptions occur, which is even worse.
- Another possibility would be to declare `in` out of the `try` block and initialise it inside, then closing it outside. However, this risks a `NullPointerException` if it is never initialised (e.g. because the file does not exist or is not readable) before `in.close()` is called.
- Finally, a fourth possibility is to duplicate the `close()` call, which is clearly suboptimal.

We just need a way to make sure that a data source is always closed.

This is what `finally` is for. Every `try` can have a `finally` block that is always executed after the `try` block; if an exception is thrown and caught, the `finally` method will be executed after the code in the `catch` block is executed. The resulting code would look like this:

```
File file = new File("file.csv");
BufferedReader in = null;
try {
    in = new BufferedReader(new FileReader(file));
```

[6] If you look at the Java Doc, you will see that FileNotFoundException is a subclass of IOException, so this catch block must come before the other one.

```
        String line;
        while ((line = in.readLine()) != null) {
            // ... do things with the data here
        }
    } catch (FileNotFoundException ex) {
        System.out.println("File " + file + " does not exist.");
    } catch (IOException ex) {
        ex.printStackTrace();
    } finally {
        try {
            if (in != null) {
                in.close();
            }
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
```

There are several things worth noticing. First, the BufferedReader must be declared before the `try/catch/finally` blocks to be visible in all scopes. Second, the method `close()` of readers can throw an `IOException`, so it must be called inside another `try/catch` construct. The code is a bit complicated, but we can make it much clearer by using a private method to hide some of the details:

```
// ... in some method...
    File file = new File("file.csv");
    BufferedReader in = null;
    try {
        in = new BufferedReader(new FileReader(file));
        String line;
        while ((line = in.readLine()) != null) {
            // ... do things with the data here
        }
    } catch (FileNotFoundException ex) {
        System.out.println("File " + file + " does not exist.");
    } catch (IOException ex) {
        ex.printStackTrace();
    } finally {
        closeReader(in);
    }
// ...
private void closeReader(Reader reader) {
    try {
        if (reader != null) {
            reader.close();
        }
    } catch (IOException ex) {
```

```
            ex.printStackTrace();
        }
    }
```

1.3.2 Writers

Class `Writer` is the dual opposite of `Reader`, and the class from where all writers descend. Writers are used to write on files on disk.

Most of the reader classes have an equivalent writer class. There is a `Writer`, a `BufferedWriter`, a `FileWriter`, and a `StringWriter`. They are used mostly in the same way, although there is one important difference: there is no `writeLine()` method. Apart from this, they are used in a very similar way to how readers have been used in the preceding section.

There is an additional class that has no dual reading counterpart but is very convenient to use: `PrintWriter`. This class provides a constructor to build an object from a `File`, and it also provides convenient methods for writing into files, including: `append(char)`, `print(String)`, `printf(...)` (similar to C's), and `println(String)`. On top of that, closing a `PrintWriter` does not throw an `IOException` so there is no need for an additional `try/catch` inside a `finally` block. The code below shows a typical use of `PrintWriter`:

```
File file = new File("file.csv");
try {
    PrintWriter out = new PrintWriter(file);
    out.write(...);
} catch (FileNotFoundException ex) {
    // This happens if file does not exist and cannot be created,
    // or if it exists, but is not writable
    System.out.println("Cannot write to file " + file + ".");
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    out.close()
}
```

Flushing When you read from a file, your program will not continue until the read operation finishes. This is not exactly the case with writing operations. When you execute one of the `write()` suite of methods, this only puts the information in some structure of the operating system. This may or may not be put immediately on disk, depending in different factors.

This may produce strange effects, like data being lost because the program crashed after the `write()` call but before the data was actually committed to disk. If you want to ensure that data has been written on disk, you can use the method `flush()`.

Closing a data sink using `close()` automatically flushes it before closing to prevent data loss.

1.4 Binary files

Text files contain text. They contain only printable characters; they can be seen as a big string on disk. You are encouraged to use only text files: they are simpler, they are clearer, and they allow

you to examine them when there are bugs in your code that corrupt data. In the preceding notes we have seen how to interact with text files.

Old applications sometimes use *binary* files. Binary files contain non-printable characters and cannot be read by people. Compiled class files are binary files. Try opening a `.class` file in your favourite text editor to get a feel of what they look like (hint: gibberish). Old applications use binary files usually because they use less space on disk, and disk (and memory!) space was limited in old times.

In order to work with binary files, the classes `InputStream` and `OutputStream` are used, instead of `Reader` and `Writer`. If you ever need to work with binary files in your future career as a programmer, you are prepared at this point to read the documentation directly from the Java Doc. The main ideas are the same: open resource, read and write, close resource. There is also a class `RandomAccessFile` that allows you to read from and write to arbitrary positions in a file, instead of using it as a stream.

Unless you are dealing with old legacy code, you should always use plain text in your programs. Memory and disk space are much cheaper[7] than sleepless nights trying to debug binary data with an hexadecimal editor.

[7] If you are really concerned about space, use text files compressed with ZIP —this is what LibreOffice does—. Look at package *java.util.zip* where you will find many classes that take care of the compressing / decompressing process.