PHAS3459: Scientific Programming Using Object Oriented Languages

Module 4: Input and Output

Dr. B. Waugh and Dr. S. Jolly
Department of Physics & Astronomy, UCL

Contents

1. **Aims of Module 4**
2. **Input and Output (I/O)**
3. **Text-Based and Data-Based I/O**
4. **The Screen and Keyboard**
5. **Files and URLs**
6. **Closing Resources**
7. **Processing Text**
8. **Data-Based I/O**
9. **Summary**

1. Aims of Module 4

- Understand the concepts of:
  - text-based and data-based I/O;
  - buffered I/O.
- Be able to read data:
  - from the keyboard;
  - from a file on disk;
  - from a file on a web server.
- Be able to write data:
  - to the screen;
  - to a file on disk.
- Be able to filter and tokenize data.
- Be able to convert data from `String` format to numeric data types.
- Be able to handle the exceptions associated with I/O.

2. Input and Output (I/O)

Any effort that goes into making a computer program do anything will be wasted if the results cannot be passed in some way from the computer running the problem to the rest of the world. This may involve displaying text or graphics on a screen, printing them on paper or sending a signal to make a machine perform some physical action. Some results may also be stored on a hard drive or other storage device so that they can be accessed and used later by another or the same computer program.

In many cases the program may also need to receive information from the outside world in order to fulfil its function. It may read the results of another program from a file on disk, react to a person pressing keys on a keyboard (or a rat moving levers in a cage) or receive data from a piece of equipment such as a flow meter or a particle detector.

In this module we will deal with a limited range of input and output mechanisms, mostly using the keyboard and screen of the computer, the disk drive of the computer (or a disk somewhere in a file server) and files that we can access on web servers via the Internet.

Even in these cases there are complications that make a contrast with the controlled environment of the Java Virtual Machine. Different operating systems and computers have different ways of representing the same information, and the same raw information can be represented in different ways. A byte somewhere in a file or in the memory of a computer may represent a text character or an integer number. It may also combine with other bytes to make up a character from a larger set, or an integer or real number. If it represents part of a larger data type then the most significant bits could be carried by the first byte in a group or the last. It may represent different characters to software using different alphabets, and the same character in the same alphabet may be represented in different ways on different systems. Even the end of a line in a simple text file is represented as a *line feed* character in Linux, a *carriage return* character in Mac OS 9, and both (carriage return followed by line feed) in Windows.

Fortunately Java comes with a package `java.io` containing classes and methods that deal with most of these problems for you. These methods can read bytes from a file and automatically convert them to other data types in the appropriate way for the computer system in question. In order to use the classes from this package you need to import them into your classes by adding the following line below the `package` line at the top of each source file:

```
import java.io.*;
```

There is another package, `java.nio`, which provides more flexible and higher-performance input and output at the cost of being less easy to understand and use. We will not use it in this course.

3. Text-Based and Data-Based I/O

Text, as distinct from other forms of data, is generally intended to be readable by humans as well as by computers, whether it is written in English, Japanese or Java. Different forms of text may be written using different sets of characters, and different *encodings* may be used to convert these characters into bytes in a computer disk or memory. Java uses a system called Unicode, and more specifically an encoding called UTF-16, to represent characters as 16-bit character variables, as introduced in Module 1.

However, most computer systems are set up to be used with encodings that use 8-bit *bytes*, usually only representing a smaller set of characters sufficient for a small number of languages. The methods available in the `java.io` package automatically convert between Unicode and these other encodings as required. The classes that do this are known as *character streams* or as `Reader` and `Writer` classes. Different versions of these classes exist for reading and writing data from and to different sources and destinations.

When dealing with numeric data it often makes sense to store and transfer it in a similar binary form to that used to store variables in the computer's memory. The computer can carry out arithmetic operations directly on numbers in this form. They also require less space in memory or on disk than the same data in text form. A `double` variable can store a real number to a precision of approximately 15 significant figures, while storing the number in the form of a text string would typically require eight bits for each character — including the digits, the decimal point and any exponent and minus sign — about twice as much space.

However, when data needs to be entered or read by a human there is obviously an advantage to using text form. Since different computer systems also use different binary representations of the same number, it can also be useful to use text as a way of transferring data between computers or programs. This is often done using various forms of XML, or simply space- or comma-separated lists of values in a file.

In this module you will find out how to convert numeric data between these two forms, while most of the input and output will be in text form.

4. The Screen and Keyboard

The programs you have seen and written so far have presented their output to the screen in the form of text. The screen can also display graphics and this will be covered later in the course. In this module you will write programs that also accept input from the keyboard, which is of course a source of text-based input.

To print information to the screen you have already been using the `System.out` object, a static member of the `System` class. `System.out` is an instance of a class called `PrintStream`, which behaves in many ways like a `Writer` class (a character stream) but with some differences for historical reasons: `System.out` has been part of Java for longer than the `Reader` and `Writer` classes. The modern equivalent of `PrintStream` is the `PrintWriter` class, which you will use when writing text to files. However, for writing to the screen we may as well stick with `System.out`.

The main functionality that `PrintStream` and `PrintWriter` both add to the plain character stream classes, which can only output a series of individual characters, is the ability to convert various types of data values to text strings using the overloaded methods `print` and `println`. You should by now be so familiar with this that no examples are needed!

The `print` and `println` methods print each value in a standard way, which may not always be exactly what we want. For example, a `double` is typically printed with 16 digits, which may be more than we really need. Java provides a relatively simple way to gain more control of the format of the output, for example to print a value to a chosen number of decimal places, using the `format` method of the `PrintWriter` and `PrintStream` classes[1]. The following example illustrates how the same value can be formatted in different ways.

```
System.out.println("pi = "+Math.PI);
System.out.format("pi = %f%n", Math.PI);
System.out.format("pi = %e%n", Math.PI);
System.out.format("pi = %10.3f%n", Math.PI);
System.out.format("pi = %.3f; e = %.3f", Math.PI, Math.E);
System.out.format("; each is stored using %d bits%n", Double.SIZE);
```

The first argument of the `format` method is a *format string*, which comprises the text to be printed but with a *format specifier* in place of each value to be included in the output. The format string is followed by the list of corresponding values. Each format specifier starts with the `%` character followed by various pieces of information specifying what format to use, optionally including the width (total number of characters to print) and precision (number of digits after the decimal point) to print. In the above example, `%10.3f` means a floating point number with a width of 10 and 3 decimal places. A new line is indicated by `%n`. For more details of all the formats and options available see the API documentation of the class `java.util.Formatter`.

To read input from the keyboard we use the object `System.in`. Like `System.out`, `System.in` has been in the Java language for longer than the modern `Reader` classes and is in fact an instance of the class `InputStream`. We can use it to get data a byte at a time:

```
public class KeyboardTest {
  public static void main(String[] args) {
    int CARRIAGE_RETURN = 10;
    int av;
    System.out.println("Type something please!");
    try {
      do {
        av = System.in.read();
        char avc = (char) av;
        System.out.println(av+": "+avc);
      }
      while (av != CARRIAGE_RETURN);
    }
    catch (java.io.IOException e) {
      System.out.println("Problem: "+e.getMessage());
    }
    System.out.println("Thank you!");
  }
}
```

Note that, like many input/output operations, the `read` method of the `InputStream` class throws an exception of type `IOException` and therefore needs to be enclosed in a `try` block.

It usually makes more sense to wrap `System.in` with a `Reader` object (more specifically an `InputStreamReader`) to convert the bytes to characters using the appropriate text encoding. It is even more convenient to use a `BufferedReader` to read the input a line at a time, thus automatically taking care of any required backspaces while entering text:

```
InputStreamReader r = new InputStreamReader(System.in);
BufferedReader b = new BufferedReader(r);
System.out.println("Please type something!");
try {
  String s = b.readLine();
  System.out.println("You typed: "+s);
}
catch (IOException e) {
  System.out.println("Problem: "+e.getMessage());
}
```

---

[1] This method was added in Java version 1.5. In earlier versions it was necessary to use the class `java.text.Format` and its subclasses such as `DecimalFormat`. ↩

---

5. Files and URLs

A `FileReader` object can be used to read characters from a file in much the same ways as we used a `InputStreamReader` above to read characters (via `System.in`) from the keyboard.

Once more, it makes sense to wrap the basic `Reader` object with a `BufferedReader`. Every operation that reads from or writes to a disk takes at least a certain time, called the *latency*.

Thus it is more efficient to read a large chunk of data into memory in one go than to fetch it a byte or character at a time. A `BufferedReader` does this for you behind the scenes. When you ask for the first character from a `BufferedReader` it actually reads many characters from the input stream and stores them in an area of memory called a *buffer*. The second and subsequent characters only have to be read from memory, which takes place much faster, until the entire contents of the buffer have been read and the buffer is refilled.

```
// Print contents of text
// file to the screen
public static void printFile(String fileName)
throws IOException {
  FileReader fr = new FileReader(fileName);
  BufferedReader br = new BufferedReader(fr);
  String line;
  while ((line=br.readLine()) != null) {
    System.out.println(line);
  }
}
```

Writing to a file is similar, but this time we use a `FileWriter` and wrap it with a `BufferedWriter`. The latter acts like a `BufferedReader` in reverse. You can write characters to it but it will initially just store them in a buffer in memory. Only when the buffer is full, or the buffer is *flushed*, will the accumulated data be passed on, in this case to a file on disk. Flushing can be carried out using the `flush` method but will also happen automatically when the `BufferedWriter` is closed. It is important to close output streams when you have finished writing to them to make sure all the data makes it to its destination. This will be covered in the following section.

In the following example we use a `PrintWriter` to add a further layer around the `BufferedWriter`. This (as discussed earlier) enables us to print numbers as well as strings to the file:

```
FileWriter f = new FileWriter(fileName);
BufferedWriter b = new BufferedWriter(f);
PrintWriter pw = new PrintWriter(b);
pw.println("Some numbers");
for (int i=0; i<10; i++) {
  pw.println(i+" "+i*0.1);
}
```

Reading from a file on a web server is also made simple by the Java API. This time we need to import the package `java.net`:

```
import java.net.*;
```

To read the contents of a file over the web, we need to create a `URL` object and use its `openStream` method, then read from the resulting `InputStream` using an `InputStreamReader`. Then we can wrap the `InputStreamReader` with a `BufferedReader` as before. It may seem rather convoluted, but when you have done it once you can always refer back to your code when you need to do it again.

```
URL u = new URL(webAddress);
InputStream is = u.openStream();
InputStreamReader isr = new InputStreamReader(is);
BufferedReader b = new BufferedReader(isr);
String line;
while ((line=b.readLine()) != null) {
  System.out.println(line);
}
```

6. Closing Resources

The code examples dealing with files in the previous section will all cause Eclipse to warn you of a possible *resource leak* because we are not closing the files after using them. A *resource* here simply means an object that must be closed when the program has finished with it, and a file is an example of this.

The simplest way to get rid of the warning is to call the `close()` method:

```
FileWriter f = new FileWriter(fileName);
BufferedWriter b = new BufferedWriter(f);
PrintWriter pw = new PrintWriter(b);
pw.println("Some numbers");
for (int i=0; i<10; i++) {
  pw.println(i+" "+i*0.1);
}
pw.close();
```

When the `PrintWriter` is closed, it calls the `close` method of the `BufferedWriter`, which in turn does the same to the `FileWriter`, so it is not necessary to close all of these objects separately.

Unfortunately this code is not guaranteed to work correctly. In earlier versions of Java, ensuring that resources were properly closed was quite difficult, and even expert programmers frequently made mistakes. The problems arise from the fact that any of the steps dealing with the file, including opening and closing it, can throw an exception. Careful use of `try` and `finally` blocks was required to handle all the possible ways the process could go wrong.

Since the release of Java 7, this is made much easier by the *try-with-resources* statement. We can simply tell the compiler at the beginning of a `try` block which resources we need to use within the block, by listing them between parentheses, and it will do its best to ensure the resource is closed at the end of the block. We still have to be prepared for the situation where the `close()` statement itself throws an exception, and declare or catch this.

```
// Print contents of text
// file to the screen
public static void printFile(String fileName)
throws IOException {
  try (
    FileReader fr = new FileReader(fileName);
    BufferedReader br = new BufferedReader(fr);
  ) {
    String line;
    while ((line=br.readLine()) != null) {
      System.out.println(line);
    }
  }
}
```

7. Processing Text

Now that we know how to read text from various sources, we can start to look at ways we can deal with that text, including reading numeric data from it.

The first task in dealing with input in text form is generally to break the text into individual *tokens*, which may be words, numbers or other groups of characters. There are several ways of doing this in Java but probably the simplest uses the `Scanner` class in the package `java.util`[1].

As an example, if we have a file containing several numbers per line and we want to find the total, we could do something like this:

```
try (
  BufferedReader r = new BufferedReader(new FileReader(file));
  Scanner s = new Scanner(r);
) {
  double sum = 0;
  while (s.hasNext()) {
    String token = s.next();
    try {
      double x = Double.parseDouble(token);
      sum += x;
    } catch (NumberFormatException e) {
      // Ignore anything that is not a number!
    }
  }
}
```

By default the `Scanner` class looks for tokens separated by *whitespace*, i.e. any sequence of *whitespace characters*, which include space, tab, new-line and a few others. It is possible to use different *delimiters*, e.g. to look for values separated by commas. In this case you would need to look up the method `Scanner.useDelimiter()` and the class `java.util.regex.Pattern`.

You may recognize the method `Double.parseDouble()` from the Module 2 notes, where we mentioned briefly the use of the numeric *wrapper classes*. It takes a string as input, e.g. the string `"3.14"`, and tries to interpret it as a decimal number, converting it into a value of

type `double`. If the string cannot be interpreted as a decimal number (e.g. `"hello"` or `"twenty"`) the method throws a `NumberFormatException`.

---

[2] Other methods make use of the method `String.split()` or of the package `java.util.regex`, in both cases requiring some knowledge of *regular expressions*. For more about regular expressions, see the lesson *Regular Expressions* in the Sun Java Tutorials *Essential Classes* trail, linked from the *Java Resources* page on the course home page. The older `StringTokenizer` class used in earlier versions of Java is still available but its use in new code is discouraged. ↩

---

8. Data-Based I/O

So far we have dealt only with text-based input and output, even where the actual text we were dealing with comprised a list of numbers. As mentioned earlier, the other way of storing numeric data uses binary representation and can save both disk space and computation time.

The classes `DataOutputStream` and `DataInputStream` provide the necessary methods to convert between numbers (`int`, `double` etc.) and the streams of bytes that are dealt with by the *byte stream* classes such as `OutputStream`, `InputStream`, `FileOutputStream` etc.

To write some double-precision values to a binary file we might use something like this, using a *buffered* output stream to make our access to the disk file more efficient:

```
try (
  FileOutputStream f = new FileOutputStream(fileName);
  BufferedOutputStream b = new BufferedOutputStream(f);
  DataOutputStream d = new DataOutputStream(b);
) {
  for (int i=0; i<100; i++) {
    d.writeDouble(i*0.1);
  }
}
```

To read in some double-precision values from a binary file and add them all together we could use this:

```
try (
  FileInputStream f = new FileInputStream(fileName);
  BufferedInputStream b = new BufferedInputStream(f);
  DataInputStream d = new DataInputStream(b);
) {
  double sum = 0;
  while (true) {
    try {
      double x = d.readDouble();
      sum += x;
    } catch (EOFException e) {
      break;
    }
  }
  return sum;
}
```

Note that if we don't know how many values we expect to read in then, as in this example, we can simply loop until the `readDouble` method throws an `EOFException`, indicating the "End of File" (EOF).

9. Summary

Input and output are an essential aspect of any computer program that interacts in any way with the outside world, but they introduce various complications. This module has provided a brief tour of the basics of I/O in Java, but there is much more in the Java API than has been touched on here.