Scientific Programming Using Object-Oriented Languages
Module 4: Input & Output

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.

Input and output

- Computer program is pointless without output!
  - Any effect on the outside world is a kind of output.
  - Often a graphical display but may also be writing to a file, controlling a machine…
- Input is also often required
  - from a user at a keyboard;
  - from a file on disk (perhaps written by another program);
  - from external equipment, e.g. radiation monitor.
- The outside world is messy compared to the controlled environment inside the computer.
  - Even the other parts of the computer may not be as predictable as the Java Runtime Environment (JRE).
  - Different character sets, date formats, file systems,,,
- The JRE tries to make things easier
  - Package java.io contains a lot of utilities to make sure the same Java programs can run on many different computers.
  - Other packages, e.g. java.net, enable wider communication.

Text- and data-based I/O

- Most i/o in this course is in the form of text.
  - Even numbers are printed to the screen or to files in the form of text.
  - We will also see how to read text from the keyboard, files or URLs, and interpret it as numeric values, e.g. convert the string “3.141” to the `double` value 3.141.
- We will also introduce data-based i/o
  - Store and transfer data in binary form, as stored in memory.
  - Uses less space than same data in text form.
  - Can be processed by computer without time-consuming parsing.
  - Can’t easily be read or written by humans.
  - Different conventions used by different computer systems.
  - Text-based formats, e.g. XML, often used even for communication between different computers, and for storage.

Dealing with text

- In Java, textual characters represented by 16-bit `char` values, using UTF-16 encoding of Unicode.
- Most computers actually store text using 8-bit bytes, using an encoding selected for use with a restricted set of languages.
- The methods in the package `java.io` automatically convert between characters and bytes, using the appropriate encoding for the machine running the program.
- It is possible to select different *locales* to deal with other character sets, encodings etc.
  - Not covered in this course!

Byte and character streams

- Input and output in Java uses streams
  - Byte streams used for all input and output.
  - For text, character streams act as *wrappers*, dealing with the necessary conversions.
- Byte streams are represented by `InputStream` and `OutputStream` classes
  - e.g. `FileInputStream`
- Character streams are represented by Reader and Writer classes
  - e.g. `FileReader`, `BufferedReader`, `PrintWriter`
- Different classes
  - deal with different types of i/o (e.g. using keyboard, file, network);
  - provide different features, e.g. buffering.

Printing to the screen

- Have already seen `System.out.println`
- Can also get more control over format of numbers etc.:

```
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);
```

- For historical reasons, `System.out` is not a `PrintWriter` but a `PrintStream`, and is a byte stream not a character stream.
- However, it does a similar job so we will use it for writing to the screen, while using `PrintWriter` for writing to files.

Reading from the keyboard

- Like `System.out`, `System.in` predates the generally used `Reader` classes: it is an `InputStream`.
- It provides basic methods to read a character at a time from the keyboard, as illustrated in the notes.
- We will almost always wrap it with a `InputStreamReader`, and then with a `BufferedReader`, so we can read a line at a time.
- Like many I/O methods, `BufferedReader.readLine()` can throw an `IOException`, so we have to **`try`** and **`catch`**.

Reading from the keyboard: example

```
InputStreamReader r = new InputStreamReader(System.in);
BufferedReader b = new BufferedReader(b);

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());
}
```

Reading from a file

```
// 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);
   }
}
```

- Similar but use `FileReader` instead of `InputStreamReader`.
- Buffered reading is important here:
  - each disk access takes at least a certain time, the disk *latency*;
  - this is slow compared to the CPU;
  - when you ask for the first character from a `BufferedReader`, it reads a larger chunk of the file into a *buffer* in memory;
  - subsequent reads only have to access memory, which is fast, until the buffer is empty.

Writing to a 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);
}
pw.close();
```

- `FileWriter` and `BufferedWriter` like `FileReader` and `BufferedReader` in reverse.
- `PrintWriter` provides methods to print different types of value, e.g. integers here, like `System.out`.
- `BufferedWriter` only writes to memory until the buffer is full, or we flush it.
- Closing the stream causes the buffer to be flushed to disk automatically. Always close an output stream after use!

Reading from the web

- The package java.net provides methods for network access.
- We will use it to read a file from a web server.

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

Processing text

- Often need to read text and process it in some way
  - e.g. read numbers from a file and add them together;
  - first need to read text, as already seen;
  - then split input into tokens, e.g. individual numbers;
  - then parse these tokens, e.g. convert to numeric data type.
- Various ways to tokenize input; we will use Scanner class.

```
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!
    }
}
```

Data-based I/O

- Use `DataOutputStream` and `DataInputStream`.
- These have methods like `writeDouble`/`readDouble` to deal with each primitive data type.
- See examples in notes.
- We can even read and write objects of other classes using the `ObjectOutputStream` and `ObjectInputStream` classes:
  - *object-based I/O*;
  - need to *serialize* objects;
  - beyond the scope of this course.