

CS1Ah Lecture Note 29

Streams and Exceptions

We saw back in Lecture Note 9 how to design and implement our own Java classes. An object such as a `Student4` object contains related fields such as `surname`, `forename` and `matricno`. A particular object instance provides a name (such as `JoSmith`) for these fields.

When an object is created, it is stored in computer memory. Computer memory is separated into a *stack* and a *heap*. A stack allows direct access only to the top (like a stack of plates) and it is the most recently stored information which is most easily accessible. Stacks are used for storing local information, such as the local variables or formal parameters of a method. A heap in contrast allows access to all parts (like a heap of sand). Heaps are used for storing global information, such as objects, which might be returned from a method invocation and so cannot be considered to be only of local interest.

When a computer program finishes executing, the information which it built up in memory is no longer accessible. If we wish to make a permanent record of the objects which we have created then we need to write these to a *file* stored on a computer disk. A file is one kind of *stream*. Characters typed on a computer keyboard make up another kind of stream (an *input stream*) and strings written to the screen are a third (an *output stream*). The word “stream” is used for these because it indicates that information will be accessed in sequence, starting at the start and going on to the end, without jumping randomly about. The other form of access is provided in Java by a `RandomAccessFile`.

29.1 Object serialization

The conversion of an object from its representation in memory into a form suitable for storing on disk is not as simple as one might at first think. All objects have a `toString()` method which provides one form of conversion from an in-memory representation to a storable representation but there is no guarantee that this process is reversible. Storing objects in files on disk is only useful if we can later retrieve them from these files and re-build their in-memory form for further processing so the requirement for reversibility of the storage process is an inescapable one.

The process of converting an object held in memory into a form suitable for storing on disk is known as *serialization*. Some objects are so complex that they cannot be serialized at all. Simple objects such as a `String` or an `Integer` can be serialized. We previously created a `Student4` class whose fields were three strings. Because all

of its fields are serializable you might expect that the Java language would *infer* that `Student4` was serializable. Some languages infer information in this way, but Java does not. A class is not serializable unless all of its fields are serializable and it itself is explicitly marked as being serializable.

We modify the `Student4` class to create a serializable version, `Student`. This involves changing just line 1, to tag `Student` as being serializable, and line 7, to change the name of the constructor.

```
1  class Student implements java.io.Serializable {
2      private String surname;
3      private String forename;
4      private String matricno;
5
6      // the following method is the constructor
7      public Student( String surname,
8                      String forename, String matricno) {
9          this.surname = surname;
10         this.forename = forename;
11         if (matricno.length() != 7) {
12             this.matricno = null;
13         } else {
14             this.matricno = matricno;
15         }
16     }
17     // these methods can be included as before:
18     // getSurname(), getForename(), getMatricno()
19
20     public String toString() {
21         return surname.toUpperCase() + ", " +
22             forename + " (" + matricno + ")";
23     }
24 }
```

29.2 Exceptions

Streams allow communication from a program to a storage device such as a disk. There are circumstances in which this communication might not be permitted. Shared computer systems divide up resources such as disks, allocating to each person using the system a *quota* of disk storage. These quotas are assigned to individuals and it would not be fair for one user to write a file into another's file space, thereby using up some of their quota. To enforce this policy of fairness an operating system such as Linux will set access permissions to prevent one user writing files into another's file space unless they have previously been given permission to do so.

Even on a single user system it might not be possible to write an output file. Disks have a fixed storage capacity. When the disk is full it will not be possible to write more data to it.

Similar remarks apply when reading an input file. The file might not exist, or it might not be readable (perhaps it is in someone else's private file space).

These exceptional situations are handled in Java as *exceptions*. Knowing that there is the possibility of failure, we *try* to execute a block of statements and we *catch* an exception if it occurs, executing another block of statements in that case. In the Java language the keywords **try** and **catch** are used to mark off the relevant blocks of statements. Different exceptions are *thrown* to indicate different kinds of errors. The divide by zero error which we discussed in Lecture Note 5 throws an `ArithmeticException` whereas file-related errors throw an `IOException` (an "input/output" exception).

If exceptions do not occur then the **try** block completes and the statements in the **catch** block are not executed. If it was not possible to write to the file `Students.dat` then the program would fail at line 7 with a `FileNotFoundException` (a kind of `IOException`) and continue execution at line 17, which prints a trace of the stack of method invocations which led to the point where the exception was thrown.

```

1  import java.io.*;
2  class StudentOutput {
3      public static void main(String[] args) {
4          try {
5              /* Create an output stream of student records */
6              String name = "Students.dat";
7              FileOutputStream file = new FileOutputStream(name);
8              ObjectOutputStream out = new ObjectOutputStream(file);
9
10             Student JoSmith = new Student("Smith", "Jo", "0123456");
11             Student TimSmith = new Student("Smith", "Tim", "0123457");
12
13             out.writeObject(JoSmith);
14             out.writeObject(TimSmith);
15             out.close();
16         } catch (IOException e) {
17             e.printStackTrace();
18         }
19     }
20 }

```

Having successfully opened the file at line 7 we then define at line 8 that we wish to write objects to this file (and not, for example, values of simple types such as `int`).

Two student records are created in lines 10 and 11. The first is one for Jo Smith and the second is one for Tim Smith. These are written to the file in lines 13 and 14. At line 15 the file is closed so that no further objects can be written to it. Closing a file is as important as opening it because operating systems impose a limit on the number of files which a program can have open at the same time.

Having created the file named `Students.dat` we might wish to inspect its contents. However, its contents are stored in a form which is convenient for processing by a Java program, not in a form which is convenient for us to read. The file contains some strings of characters which we can make sense of but it also contains some

additional characters whose meaning is unknown to us. These additional characters in the file are effectively a form of punctuation, separating out fields of the objects and separating one object from another. These punctuation characters have no sensible character representation so instead of trying to view the `Students.dat` file directly on the screen or print it to a printer we use a program to convert the file into a printable form. I used the Linux “octal dump” command `od -An -c Students.dat` to inspect the contents of the file in this way. The result is shown below.

```

254 355 \0 005 s r \0 \a S t u d e n t X
I 306 n 346 205 360 D 002 \0 003 L \0 \b f o r
e n a m e t \0 022 L j a v a / l a
n g / S t r i n g ; L \0 \b m a t
r i c n o q \0 ~ \0 001 L \0 \a s u r
n a m e q \0 ~ \0 001 x p t \0 002 J o
t \0 \a 0 1 2 3 4 5 6 t \0 005 S m i
t h s q \0 ~ \0 \0 t \0 003 T i m t \0
\a 0 1 2 3 4 5 7 q \0 ~ \0 005

```

(If one uses instead the command `strings Students.dat` it will pick out words which occur in the file, including `Student` and `java/lang/String`.) The contents of the file might seem confusing to us but we can write a simple program to read the file and print a report of all the objects which it finds.

29.3 Stream input

When we come to read in the contents of the file, they are read as objects. Although we only wrote `Student` objects to the output stream with the previous program it would have been possible for us to write other classes of objects there also (perhaps `Director`, `Course` or `Programme` objects). When we read objects in from an `ObjectInputStream` we must *cast* them to a particular class before we can invoke their methods (see Lecture Note 5). However, cast operations can fail. Attempting to cast a `Director` object into a `Student` object would have an undefined outcome. This too then throws an exception, a `ClassCastException`. Since we are generating a report from the file we should not simply ignore objects in the file which we do not expect to find. We can print a message in the report stating that we found an “invalid record” in the file and continue on to the next student data record. Another exceptional case occurs if we cannot find the relevant class for an object which we have read from the input stream. This is a `ClassNotFoundException`. Because there are often several possible ways in which a block of statements can fail, the grammar for the `try` statement allows more than one `catch` block to follow. In the `StudentReport` program there are two `try` statements each of which has two `catch` blocks.

Because we have in mind here a particular file which we know contains only two objects, Java’s approach to file input will seem a little strange. We attempt to keep reading from the file forever. When we reach the end of the file it is treated as an exceptional case, an `EOFException` (an “end of file” exception). We can catch this exception and print out the final line of the report, stating that we have reached the end of the report. This is done on line 14. After this we know that we do not wish to

try reading another object from the file because there are none remaining. However, we are still inside a loop statement which wishes to continue forever (the **while** loop which begins on line 9 and ends on line 19). The **break** statement on line 15 causes an exit from a **while** loop just as it causes an exit from a **switch** statement (see Lecture Note 6). After executing the **break** statement on line 15 we exit the loop and execute the statement which immediately follows it, the method invocation on line 20 which closes the input stream.

```

1  import java.io.*;
2  class StudentReport {
3      public static void main(String[] args) {
4          try {
5              /* Read the file of student records */
6              String name = "Students.dat";
7              FileInputStream file = new FileInputStream(name);
8              ObjectInputStream in = new ObjectInputStream(file);
9              while (true) {
10                 try {
11                     Student s = (Student) in.readObject();
12                     System.out.println(s);
13                 } catch (EOFException e) {
14                     System.out.println("End of report");
15                     break;
16                 } catch (ClassCastException e) {
17                     System.out.println("Invalid record");
18                 }
19             }
20             in.close();
21         } catch (IOException e) {
22             e.printStackTrace();
23         } catch (ClassNotFoundException e) {
24             e.printStackTrace();
25         }
26     }
27 }

```

Breaking out of a loop is described as a *transfer of control*. It interrupts the normal flow of program control which proceeds from executing one statement to the next. Another form of transfer of control is caused by throwing an exception. This interrupts the execution of a block of statements which we are trying to execute and transfers control to the block of statements which catches the exception. This observation suggests another way to express the `StudentReport` program. Instead of catching the `EOFException` within the **while** loop and then breaking out of it, we can instead catch the end of file exception outside the loop, so there is no need to use a **break** statement. In this version of the program the outermost statement is a **try .. catch** (lines 4 to 26). Inside that is another (lines 9 to 20) and inside that is a **while** loop (lines 10 to 17). The innermost statement is yet another **try .. catch** (lines 11 to 16).

```

1  import java.io.*;
2  class StudentReport2 {
3      public static void main(String[] args) {
4          try {
5              /* Read the file of student records */
6              String name = "Students.dat";
7              FileInputStream file = new FileInputStream(name);
8              ObjectInputStream in = new ObjectInputStream(file);
9              try {
10                 while (true) {
11                     try {
12                         Student s = (Student) in.readObject();
13                         System.out.println(s);
14                     } catch (ClassCastException e) {
15                         System.out.println("Invalid record");
16                     }
17                 }
18             } catch (EOFException e) {
19                 System.out.println("End of report");
20             }
21             in.close();
22         } catch (IOException e) {
23             e.printStackTrace();
24         } catch (ClassNotFoundException e) {
25             e.printStackTrace();
26         }
27     }
28 }

```

When executed both programs produce the same short report:

```

SMITH, Jo (0123456)
SMITH, Tim (0123457)
End of report

```

29.4 Exception specification

In order to structure the `main()` method of `StudentReport2` a bit more, we define a method `read()` which takes as parameter an object input stream and reads from it until the end of the stream is reached. Now, `main()` only has to create the object stream `in` and then call `read(in)`.

However, both `main()` and `read()` can throw an `IOException`: `main()` if `Students.dat` is not found, and `read()` if primitive data appears in the stream instead of an object. Assume that in both cases we wish to abort the computation and print the stack trace. We can do so by including the same `catch` block in both `main()` and `read()`, but this is clearly redundant. To solve this problem, Java offers the keyword

throws, which allows to declare that a method may throw an exception. We use it in line 3 of the `StudentReport3` class shown below. The compiler understands that any `IOException` thrown by `read()` will be caught by a **catch** block after the **try** block from which `read()` was invoked. The compiler does not report an error, even though `read()` does not contain any **catch** clause for `IOExceptions`.

```

1  import java.io.*;
2  class StudentReport3 {
3      static void read(ObjectInputStream in) throws IOException {
4          try {
5              while (true) {
6                  try {
7                      Student s = (Student) in.readObject();
8                      System.out.println(s);
9                  } catch (ClassCastException e) {
10                     System.out.println("Invalid record");
11                 }
12             }
13         } catch (EOFException e) {
14             System.out.println("End of report");
15         }
16         catch (ClassNotFoundException e) {
17             e.printStackTrace();
18         }
19     }
20     public static void main(String[] args) {
21         try {
22             /* Read the file of student records */
23             String name = "Students.dat";
24             FileInputStream file = new FileInputStream(name);
25             ObjectInputStream in = new ObjectInputStream(file);
26             read(in);
27             in.close();
28         } catch (IOException e) {
29             e.printStackTrace();
30         }
31     }
32 }

```

29.5 Runtime exceptions

As a rule, for each exception that a method may throw, the programmer must either handle the exception by means of a **catch** block in the method, or explicitly declare the exception using **throws**. Forgetting to do so results in an error at compile time. This is an interesting advantage of Java over C++, which only catches violations of exception specifications at run time

However, there is a special case. The designers of Java decided that certain exceptions that may occur almost anywhere do not have to be declared; it is implicitly assumed that any method can throw them. From a practical point of view, requiring all such exceptions to be declared would have led to cluttered code.

Runtime exceptions almost always point to a bug in the program, and so they are often not caught. They are grouped together in the class `RuntimeException`. `ArithmeticException` and `ClassCastException` are subclasses of `RuntimeException`. Other subclasses are `IndexOutOfBoundsException`, thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range, `NullPointerException`, thrown when an application attempts to use `null` in a case where an object is required, and `IllegalArgumentException`, thrown to indicate that the application has attempted to convert a string to one of the numeric types, but that the string does not have the appropriate format.

Exceptions like `ClassNotFoundException`, `EOFException`, or its superclass `IOException` are not runtime exceptions. If they are not declared or handled, then the compiler reports an error.

Stephen Gilmore.

Javier Esparza, 2003/01/03 11:06:52.