Topic 8

Java Streams, Serialization and Files

Learning Outcomes

- **Describe** how streams are used for Input/Output (I/O) in Java
- **Explain** the pros and cons of text files and **demonstrate** their application in Java code
- **Explain** the pros and cons of binary files and **demonstrate** their application in Java code
- **Explain** the pros and cons of Random Access files and **demonstrate** their application in Java code
- **Describe** the Object Serialisation mechanism in Java and **explain** the pros and cons of this approach
- **Implement** Object Serialisation in Java using appropriate classes and interfaces
- **Describe and Document Diagrammatically** the structure of the Decorator pattern and how it is used with Streams in Java

Files Overview

- In the absence of files we either have to input all the details through the keyboard every time/or hard code them!
  - Since RAM is volatile and any changes are not reflected when we run the program again
- We can instead write all the program data (e.g. students, accounts) to a file, allowing it to be read back and recreated
- In this lecture we will deal mainly with text files – though binary files are more efficient if human readability is not required
- Java also provides convenient ways to serialize (write) Java objects to files – example code covered later in lecture

Files : General Approach

Before you work with files you should first:

- Decide whether a file is the most appropriate choice e.g. is a relational database or persistence engine such as an object-relational manager (ORM) more appropriate
- If using a formal structured format such as XML or JSON then consider whether you should be using a library/API instead of parsing the file directly (e.g. see `javax.xml.parsers` package in API)
- Identify the data values (e.g. instance variables) that need to be written out (often different for each class that has persistent data)
- Devise a file format which effectively represents the details for each different data type as an identifiable record in the file (including differentiating between different record types)

Files : General Approach (contd.)

- Decide when you will read/update/write data
  - e.g. start/end of program
  - every time a change is made
  - may rewrite entire file or just the changes/new data (e.g. appending or random access)
- Implement a file writing mechanism based on which of the above approaches you choose
- Implement a file reading mechanism which reads in the data from the file and reconstructs the corresponding objects/data to recreate the original program state

- Ok let's take a look at our first simple example `sadi.topic8.files.FileToConsole`

Streams

- A stream is a mechanism that allows data to flow between a computer program and I/O devices.
- We have already used two streams which are pre-created and described in API docs
  - **System.out** output stream connected to screen
  - **System.in** input stream connected to keyboard
- More details on OO design of Java streams later in lecture!

Using Scanner class

- You have probably already used the Scanner class for reading input from the console
- The Scanner class can also be used for reading from a file
- To read from a file, you can configure a Scanner class to use a File class as in:

```
Scanner input = new Scanner(new File("marks.txt"));
```

- Scanner breaks the input into tokens based on delimiter characters or regular expressions (see next slide)
- The `sadi.topic8.files.ReadMarksTextFile` example uses a text file which is read and displayed in the screen

- **NOTE**: I have made some changes to the project source code compared to these notes so you might like to compare :)

Some tips for Scanner

- Check the constructors to see what types of source input Scanner can handle
  - It is pretty flexible including streams, strings, files etc.!
- If the input has an expected format you can just call `nextXYZ()` methods accordingly and catch exceptions if unexpected input occurs
- If the input is unknown or can vary then can use the `hasNextXYZ()` methods to see what is available and handle accordingly
- Supports regular expression syntax from the `java.util.regex.Pattern` class
  - See methods `useDelimeter(..)` and `findInLine(..)`

Copying One File to Another

```
import java.io.*;
import java.util.*;
public class FileToFile  {
   public static void main(String[] args)
                              throws IOException  {
       // open file for reading with a Scanner
       Scanner sc = new Scanner("source.txt"));

       // open file for writing with a PrintWriter
       PrintWriter pw = new PrintWriter (
                      new FileWriter ("dest.txt")));

       String inputLine;

       while (sc.hasNextLine())  {  // still more input
          inputLine = sc.nextLine();
          pw.println(inputLine);
       }
       pw.close(); // must do this to write data out properly!
   }// main
}
```

Create a FileWriter object and decorate it with a PrintWriter object.

Using println() of PrintWriter.

Numbers and other types of data in text files

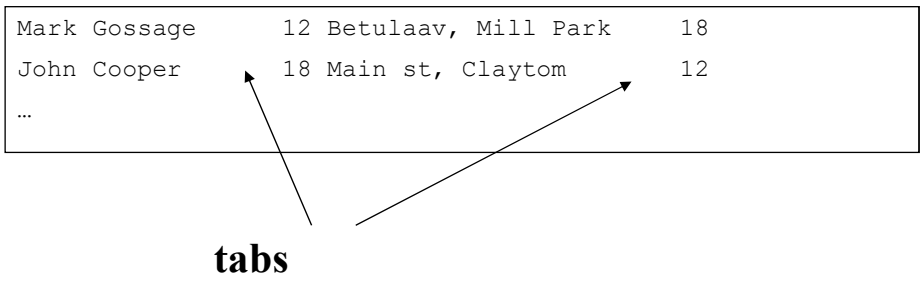
```
// writes numbers, booleans to a file as text
import java.io.*;
public class BinaryToTextFile  {
   public static void main(String[] args)
      throws IOException  {

     //  create new buffered output writer
     PrintWriter pw = new PrintWriter(
                     new FileWriter ("dest.txt")));

     boolean flag = true;
     int anInt = 17;
     double aDouble = 123.45;

     pw.println(flag);
     pw.println(anInt);
     pw.println(aDouble);

     pw.close();
   }  // main
}
```

Overloaded println() method

Writing records with multiple fields

```
import java.io.*; import java.util.*;
public class WriteStudentInfo{
   public static void main(String[] args) throws IOException
    {
        String name, address, fileName;
        int age;
        System.out.println("Enter the name of the file : ");
        Scanner sc = new Scanner(System.in);
        fileName = console.nextLine();
        PrintWriter pw = new PrintWriter(new FileWriter(fileName));
        do {
            System.out.println("Enter name : "); name = sc.nextLine();
            System.out.println("Enter address:");address = sc.nextLine();
            System.out.println("Enter age : ");  age = sc.readInt();

            sc.nextLine();
            pw.println("" + name + "\t" + address + "\t" + age);
            System.out.println("Continue  Y/N ?");
        } while ( sc.nextLine().charAt(0) == 'Y');

        pw.close();  // must close PrintWriter to write data out properly!
        sc.close();
   }
}
```


Format of the output file and Reading the values back



To reading these fields back we use a StringTokenizer class, whose constructor takes a String and a delimeter ("\t")

```
StringTokenizer inReader =
      new StringTokenizer(line,"\t");
```

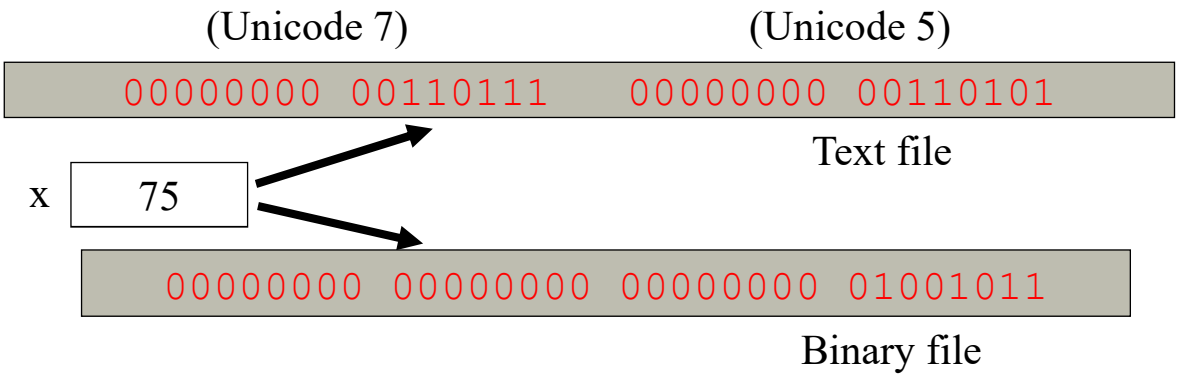
```
import java.io.*;  import java.util.*;
public class ReadStudentsInfo
{  public static void main(String[] args)  throws IOException
   {  String line;    String name;
      String address;
      String fileName;   int age;
      System.out.println("Name of file to read from : ");
      Scanner sc = new Scanner(System.in);
      fileName = sc.nextLine();
      Scanner fileSc = new Scanner(new FileReader(fileName));
      System.out.println("name \t address \t age");
      while ( fileSc.hasNextLine() )
      {  line = fileSc.nextLine();
         StringTokenizer inReader = new StringTokenizer(line,"\t");
         if (inReader.countTokens() != 3)
            throw new IOException("Invalid Input Format");
         else
         {  name = inReader.nextToken();
            address = inReader.nextToken();
            age = Integer.parseInt(inReader.nextToken());
            System.out.println(name+"\t"+address + "\t" + age);
         }
      }
      fileSc.close();
   }
}
```


Binary and Text Files

Contents of binary files are to be treated as binary digits.

- Humans cannot easily read binary files
- Binary files cannot be created/edited with text editors
- More efficient to process

In contrast contents of text files can be created/edited by text editors and are human readable.



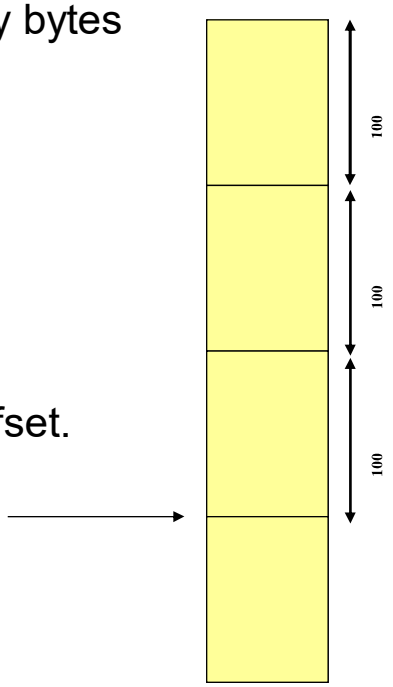
Random Access Files

If each record is 100 bytes how many bytes should I skip to read 4th record ?

To read nth record ?

Use the seek() method of RandomAccessFile specifying the offset.

Why is it useful ?



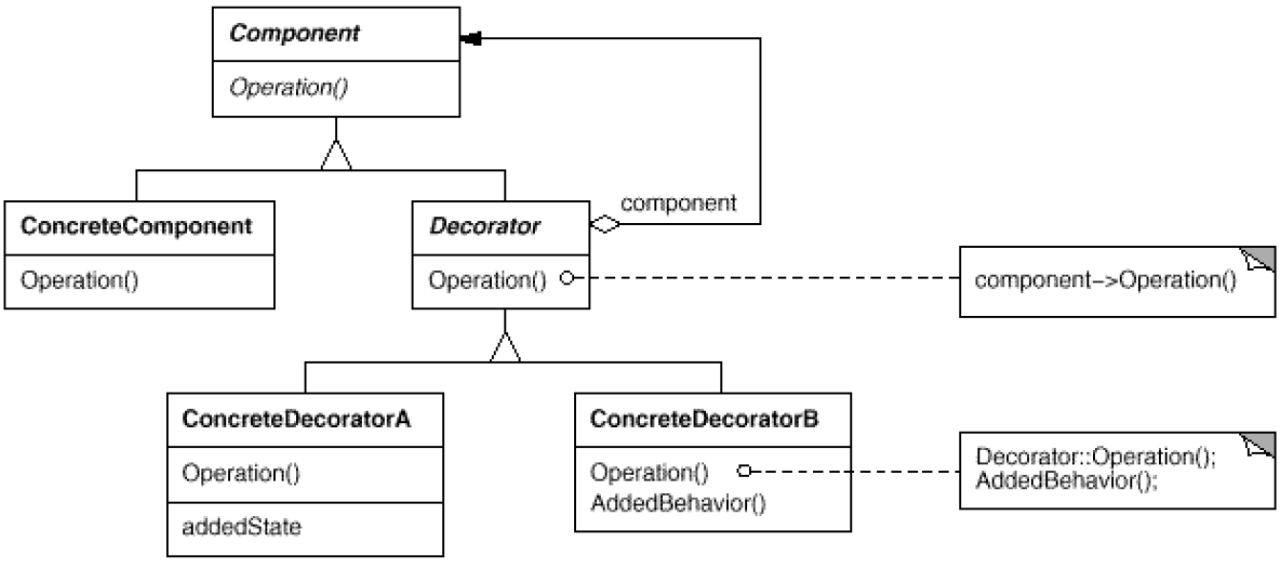
RandomAccessFile Example

```
import java.io.*;
public class RFile  {
   public static void main(String[] args) throws IOException  {
         RandomAccessFile rafile = new RandomAccessFile("test.dat","rw");
         // writing 3 int elements to array
         for (int i = 1; i <= 3; i++)
             rafile.writeInt(i);// each int occupies 4 bytes in the file

         //  reading and updating values
         rafile.seek( rafile.getFilePointer() - 4 * 1);
         int x=rafile.readInt();
         x = x + 20;
         rafile.seek( rafile.getFilePointer() - 4 * 1);
         rafile.writeInt(x);

         rafile.seek(0);
         for (int i=1; i<=3; i++)
             System.out.println(" " + rafile.readInt());
   }  // main
}
```

Streams in Java (More Details!)

- A stream is a continuous collection of bytes that is read/written from/to a source and a destination
  - the abstract class `java.io.OutputStream` accepts bytes and writes them to a data sink (destination)
  - the abstract class `java.io.InputStream` reads bytes from a data source
- In Java streams are used for most common input/output (I/O) based operations such as
  - devices, files, sockets (streams over a network), object serialization, pipes between threads etc.
- By using streams a considerable amount of reuse is obtained in implementing these different functionalities
  - makes it easier for the programmer since common operations are used for the different tasks (polymorphically)
- Streams in Java are implemented using the classic Decorator pattern (see next slide)

Decorator Pattern



From: Erich Gamma et al., *Design patterns : elements of reusable object-oriented software* , Addison-Wesley, 1995

Decorator Pattern

- **Component**: defines one or more operations that can have specialised responsibilities added to them dynamically as an alternative to inheritance
  - e.g. `java.io.InputStream` and `java.io.OutputStream`
- **ConcreteCompenent**: defines a concrete object to which additional responsibilities can be dynamically added at runtime
  - e.g. `java.io.FileOutputStream`
- **Decorator**: maintains a reference to a Component object and defines an interface that conforms to **Component's** interface
  - Java does not explicitly use this class in its streams implementation
- **ConcreteDecorator**: adds responsibilities to a Component which is usually given as a constructor parameter
  - e.g. `new ObjectOutputStream(new FileOutputStream("myFile.txt"))`
- The decorator pattern provides flexibility of many combinations of the different concrete classes without excessive inheritance combinations

Streams and Decorator Pattern (continued)

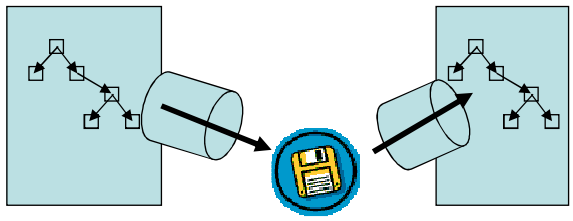
- See also `java.io.Reader` and `java.io.Writer` and its subclasses
- character based rather than byte based and thus supports extended character sets such as UNICODE UTF-16 etc.
- e.g. `java.io.StringWriter, java.io.PrintWriter`
- some versions e.g. `PrintWriter` can decorate an `java.io.OutputStream` directly else use an `InputStreamReader` or `OutputStreamWriter` to convert
- e.g. `new PrintWriter(new FileOutputStream("myfile.dat"));`
- e.g. `new BufferedReader(new InputStreamReader(socket.getInputStream()));`
- can decorate other Writers
- e.g. `new PrintWriter(new StringWriter());`

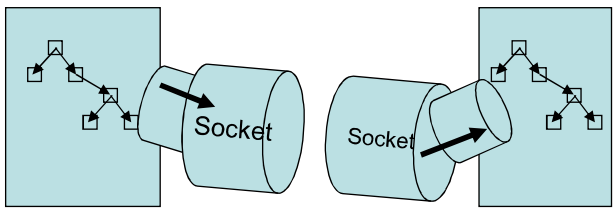
Streams and Decorator Pattern (continued)

- Sometimes you need to do a bit of work to find what you are looking for!
- For example `FileOutputStream` and `FileWriter` constructors have a `boolean` parameter for whether to append to the file
- see API docs (especially constructors) to see which combinations are possible
- http://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html

Object Streams

- Java provides the `java.io.ObjectOutputStream` class that can write Java types (ranging from primitives to entire object graphs) to a stream (e.g. a complete Collection of complex types)
- To read back the object graph the `java.io.ObjectInputStream` class is provided
- Classes must implement `java.io.Serializable` in order to be serialized





Writing/Reading Objects to file

Writing an object to the file student.dat

```
Student s = ...
ObjectOutputStream out = new ObjectOutputStream(new
   FileOutputStream("student.dat"));
out.writeObject(s);
```

Reading the object from the file student.dat

```
Student s = …
ObjectInputStream in = new ObjectInputStream(new
   FileInputStream("student.dat"));
Student s = (Student) in.readObjet();
```

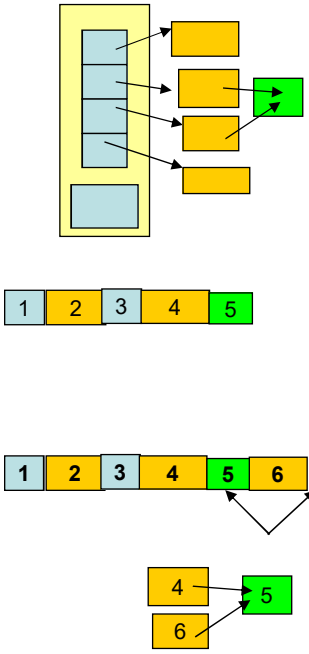

Serialization

- What kind of objects can be written to an object stream ?
  - Objects whose classes implement the `java.io.Serializable` interface
- What are the methods in `Serializable` ?
  - This interface has no methods and is called a tagging interface since its purpose is only to id the class as having a certain property
- Why all classes are not made `Serializable`?
  - A programmer may not want objects containing confidential data to be serialized
  - Part of the object may hold temporary values that are meaningless once the program execution is complete.
  - Attributes that should not be serialized should include the qualifier `transient`
  - many standard library classes such as String etc. are Serializable

```
class Student implements Serializable
{
...
}
```

How Serialization Works

- When an object is serialized all the objects referred by that object are also serialized i.e. the complete object graph
  - The data represented by the complex graph must be flattened to a byte stream i.e. serialized
- Each object is given a serial number since graphs can have multiple or circular references and if the same object is written twice only the serial number is written the second time
- When objects are read back from the stream, duplicate numbers are restored as references to the same object
- IMPORTANT: If the object is subsequently modified but sent to the same stream we need to reset() the stream otherwise we get the old value
- see http://www.javaspecialists.eu/archive/Issue088.html



Sample Program that uses Serialization

- The sample program consists of three main classes Student, Committee and ArrayList
- The Student and Committee classes implement `java.io.Serializable` and so does ArrayList (see API docs to confirm this)
- The first program creates a few student objects and serializes the List containing them to the file students.dat
- The second program reconstructs all the objects and displays all the student details.
- See sample code package `sadi.topic8.serialization`