Lab Notes Week 3
DT228/4 Distributed Systems

External data representation and marshalling

Marshalling and Unmarshalling

- The information stored in running programs is represented as data structures
  - for example, by sets of interconnected objects
  - whereas the information in message consists of sequences of bytes.

- Irrespective of the form of communication used, the data structures must be
  - Flattened, converted to a sequence of bytes before transmission
  - and rebuilt on arrival.

- The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. The representation of floating-point numbers also differs between architectures.

Marshalling and Unmarshalling

- There are two variants for the ordering of integers: the so-called big-endian order, in which the most significant byte comes first; and little-endian order, in which it comes last.

- Another issue is the set of codes used to represent characters:
  - for example, the majority of applications on systems such as UNIX use ASCII character coding, taking *one* byte per character, whereas
  - the Unicode standard allows for the representation of texts in many different languages and takes *two* bytes per character.

External data representation and marshalling

- One of the following methods can be used to enable any two computers to exchange binary data values:
  - The values are converted to an agreed external format before transmission and converted to the local form on receipt;
    - if the two computers are known to be the same type, the conversion to external format can be omitted.
  - The values are transmitted in the sender's format, together with an indication of the format used, and the recipient converts the values if necessary.

- An agreed standard for the representation of data structures and primitive values is called an *external data representation*.

Marshalling and Unmarshalling

- *Marshalling* is the process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
  - Marshalling consists of the translation of structured data items and primitive values into an external data representation.

- *Unmarshalling* is the process of disassembling them on arrival to produce an equivalent collection of data items at the destination.
  - Unmarshalling consists of the generation of primitive values from their external data representation and the rebuilding of the data structures.

External data representation and marshalling

Approaches:

- CORBA's common data representation
  - which is concerned with an external representation for the structured and primitive types that can be passed as the arguments and results of remote method invocations in CORBA. It can be used by a variety of programming languages

- Java's object serialization
  - which is concerned with the flattening and external data representation of any single object or tree of objects that may need to be transmitted in a message or stored on a disk. It is for use only by Java.

- XML (Extensible Markup Language)
  - which defines a textual format for representing structured data. It was originally intended for documents containing textual self-describing structured data – for example documents accessible on the Web – but it is now also used to represent the data sent in messages exchanged by clients and servers in web services.

External data representation and marshalling

- Two other techniques for external data representation:
  - Google uses an approach called *protocol buffers* to capture representations of both stored and transmitted data
  - JSON (JavaScript Object Notation) is an approach to external data representation [www.json.org].
- Protocol buffers and JSON represent a step towards more lightweight approaches to data representation (when compared, for example, to XML).

JAVA SERIALISATION

Serialization

- **Serialization**: objects of any class that implements the *java.io.Serializable* interface may be transferred to and from disc files as whole objects, with no need for decomposition of those objects.

- The *Serializable* interface is a marker to tell Java that objects of this class may be transferred on an object stream to and from files.

- Implementation of the *Serializable* interface involves no implementation of methods. The programmer only has to ensure that the class to be used includes the declaration '*implements Serializable* ' in its header line.

java.io.ObjectOutputStream

- Class *java.io.ObjectOutputStream* is used to save entire object directly to disc.
- For output:
  - wrap an object of class ObjectOutputStream around an object of class java.io.FileOutputStream, which itself is wrapped around a *java.io.File* object or file name.

```
ObjectOutputStream ostream =
    new ObjectOutputStream (
          new FileOutputStream("personnel.dat"));
```

java.io.ObjectInputStream

- Class *java.io.ObjectInputStream* is used to read them back from disc.
- For input:
  - wrap an object *ObjectInputStream* object around a *jav.io.FileInputStream* object, which in turn is wrapped around a *java.io.File* object or file name.

```
ObjectInputStrem istream =
   new ObjectInputStrem(
         new FileInputStrem("personnel.dat"));
```

Writing/reading

- Methods *writeObject* (of *ObjectOutputStream*) and *readObject* (of *ObjectInputStream*) are then used for the output and input respectively.
- These methods write/read objects of class Object, therefore any objects read back from file must be **typecast** into their original class before we try to use them.

```
Personnel person = (Personnel)istream.readObject();
```

writeObject() method

```
final void writeObject(Object obj)
                        throws IOException
```

Write the specified object to the ObjectOutputStream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are written.

readObject() method

```
final Object readObject()
    throws IOException,ClassNotFoundException
```

Read an object from the ObjectInputStream. The class of the object, the signature of the class, and the values of the non-transient and non-static fields of the class and all of its supertypes are read.

Possible exceptions

- *Exceptions that needs to be cared about:*
  - *java.io.IOException*
  - *java.io.ClassNotFoundException*
  - *java.io.EOFException*
- In order to detect end of file, the java relies on catching the *EOFException.* This is not ideal, as:
  - Exception handling is designed to cater for exceptional and erroneous situations that we do not expect to happen if all goes well and processing proceeds as planned.
  - However, we are going to be using exception handling to detect something that we do not only know will happen eventually, but also are dependant upon happening if processing is to reach a successful conclusion.

EOF handling example

```
...
try{
   do{
   Personnel p=(Personnel)istream.readObject();//Typecast.
   …
   }while (true);
   }
catch (EOFException e){
   System.out.println("\n\n*** End of file ***\n");
   istream.close();
}
…
```

Transient fields

- Not every piece of program state can, or should be, serialized. Some things, like *FileDescriptor* objects, are inherently platform-specific or virtual-machine-dependent. If a *FileDescriptor* were serialized, it would have no meaning when deserialized in a different virtual machine.
- Even when an object is serializable, it may not make sense for it to serialize all of its state. To tell the serialization mechanism that a field should not be saved, simply declare it `transient`.

```
Protected transient short last_x, last_y
//temporary field for mouse position
```

Transient fields (I)

- A field may not be transient – but for some reason it cannot be successfully serialized.
- Example:
  - A custom AWTR component that computes its preferred size based on the size of text it displays. Because fonts have slight size variations from platform to platform

ArrayLists

- An object of class *java.util.ArrayList* is like an array, but can dynamically increase or decrease its size according to an application's changing storage requirements.
- It can hold only references to objects, not values or primitive types
- An individual *ArrayList* can hold only references to instances of a single, specified class.

ArrayLists Constructors

```
public ArrayList()
```

- Constructs an empty list with an initial capacity of ten
- The class of elements that may be held in an ArrayList structure is specified in angle brackets after the collection class name, both in the declaration of the ArrayList and in its creation.

```
ArrayList<String> stringArray = new ArrayList<String>();
```

Or shortened:

```
ArrayList<String> stringArray = new ArrayList<>();
```

java.util.ArrayLists

- Objects are added to the end of an ArrayList via method `add` and then referenced/retrieved via method `get`, which takes a single argument that specifies the object's position within the ArrayList.

Example:

```
String name1 = "Jones";
nameList.add(Name1);
String name2 = nameList.get(0);
```

Example 2: Addinf to a specific position

```
nameList.add(2, "Smith");
```

//added to 3rd position

ArrayLists and Serialisation

- More efficient to save a single ArrayList to disc than is to save a series of individual objects
- Placing a series of objects into a single `ArrayList` is a very neat way of packaging and transferring our objects.
- Having some form of **random** access, via the ArrayList class's *get* method. Without this, we have the considerable disadvantage of being restricted to serial access only.

- See Example `ArrayListSerialise.java`
- It creates three objects of class *Personnel* and then uses the add method of class ArrayList to place the objects into a ArrayList. It then employs a 'for-each' loop and 'get' method of class Personnel to retrieve the data members of the three objects.

Personnel class

```
class Personnel implements Serializable {
        private long payrollNum;
        private String surname;
        private String firstNames;

        public Personnel(long payNum,String sName,

        String fNames){
                payrollNum = payNum;
                surname = sName;
                firstNames = fNames;
        }
        public long getPayNum(){
                return payrollNum;
        }

        public String getSurname(){
                return surname;}
        public String getFirstNames(){
                return firstNames;
        }
        public void setSurname(String sName){
                surname = sName;
        }
}
```

ArrayListSerialise class

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

public class ArrayListSerialise {
public static void main(String[] args) throws IOException, ClassNotFoundException {
    ObjectOutputStream outStream = new ObjectOutputStream(new
FileOutputStream("personnelList.dat"));
            ArrayList<Personnel> staffListOut = new ArrayList<>();
                    ArrayList<Personnel> staffListIn =  new ArrayList<>();

            Personnel[] staff
                    {new Personnel(123456,"Smith", "John"),
                    new Personnel(234567,"Jones", "Sally Ann"),
                    new Personnel(999999,"Black", "James Paul")};

                    for (int i=0; i<staff.length; i++)
                            staffListOut.add(staff[i]);
                    outStream.writeObject(staffListOut);
                    outStream.close();
                    ObjectInputStream inStream = new ObjectInputStream(new
FileInputStream("personnelList.dat"));

                    int staffCount = 0;
```

ArrayListSerialise class (Cont)

```
try{
        staffListIn =  (ArrayList<Personnel>)inStream.readObject();
        //The compiler will issue a warning for the
        //above line, but ignore this!

        for (Personnel person:staffListIn) {
                staffCount++;
                System.out.println( "\nStaff member " + staffCount);

                System.out.println("Payroll number: "+ person.getPayNum());
                System.out.println("Surname: "          + person.getSurname());
                System.out.println("First names: "      + person.getFirstNames());
                }
        System.out.println("\n");
}
catch (EOFException eofEx){
        System.out.println("\n\n*** End of file ***\n");
        inStream.close();
        }
}
}
```

Vectors

- An object of class *java.util.Vector* is like an array, but can dynamically increase or decrease its size according to an application's changing storage requirements

- Like an array, it contains components that can be accessed using an integer index. However, the size of a Vector can grow or shrink as needed to accommodate adding and removing items after the Vector has been created.

java.util.Vector (I)

- Constructor overloading allows us to specify the initial size and, if we wish, the amount by which a Vector's size will increase once it becomes full.

- Vector method *elementAt*:

```
public Object elementAt(int index)
```

Returns the component at the specified index.

Vectors versus ArrayLists

- The ArrayList is faster, but it is **not** trade safe
- Vector **is** thread safe
- If thread-safety is important to your program, you should use a Vector. If not, then you should should use an ArrayList.

FILE HANDLING

Serial access files

- Serial access files
  - Files in which data is stored in physically adjacent locations, often in no particular order, with each new item of data being added to the end of the file.

- Often misnamed sequential files.
  - A sequential file is a serial file in which the data is stored in a particular order, e.g. in account number order. A sequential file is a serial file, but a serial file is not necessarily a sequential file.

Serial access files (I)

- Binary:
  - a compact format determined by the architecture of the particular computers on which the file is to be used.
  - stores data more efficiently (than text).

- Text
  - human-readable format, using ASCII.
  - more convenient for humans.

Text Files

- Require a *java.io.FileReader* object for input and a *java.io.FileWriter* object for output.
- Constructors for these objects are overloaded and may take either of the following arguments:
  - a string (holding the filename)
  - a File object (constructed from the filename).

Constructor examples

- public FileReader(String fileName)
  throws FileNotFoundException

```
1.  FileReader fileIn = new FileReader("accounts.dat");

2.  FileWriter fileOut = new FileWriter("results.dat");


3.  String fileName = "dataFile.txt";
    ---------------------------------
    FileReader input = new FileReader(fileName);
```

Constructor examples (I)

- public FileReader(File file)
  throws FileNotFoundException

- Creates a new *FileReader*, given the *java.io.File* object to read from.

```
//File objects created first…
File inFile = new File(“input.txt”);
File outFile = newFile(“output.txt”);
// Now the FileReader and FileWriter objects…
FileReader in = new FileReader(inFile);
FileWriter out = new FileWriter(outFile);
```

Constructor examples (II)

- If a string lateral is used, the full pathname may be included, but double backslashes are then required in place of single backslashes. E.g.

```
FileWriter outFile = new
    FileWriter(c:\\data\\results.dat”);
```

- A single backslash cannot be used – it would indicated an escape sequence (resenting on of the invisible characters). E.g., tab.
- File names:
  - Can be called anything;
  - Common practice to denote files by a suffix `.txt` (for text) or `.dat` (for data)

java.io.File

- An abstract representation of file and directory pathnames.
-
- Useful for checking whether an input file actually exist. Programs that depend upon the existence of such a file in order to carry out their processing *must* make use of this method.

java.io. BufferedReader and java.io.BufferedWriter

- As FileReader and FileWriter do not provide sufficient functionality or flexibility for reading and writing data from and to files, we need to wrap a BufferedReader around a File Reader object, and a PrintWriter object around a FileWriter object.

```
BufferedReader input =
    new BufferedReader(new FileREader(“in.txt”));

PrintWriter output =
    new PrintWriter(new FileWriter(“out.txt"));
```

After this we can make use of readLine(), print() and println() methods.

readLine() method

- *BufferedReader* method *readLine*():

```
public String readLine()
                throws IOException
```

Read a line of text. A line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a linefeed.

```
BufferedReader in =
    new BufferedReader(new FileREader("infile.dat"));
String input = in.readLine();
System.out.println("Value read: " + input);
```

print(), println()

- There are several `print`() and `println`() methods in the `PrintWriter` class:

```
File out = new File("outFile.dat");
PrintWriter output = new PrintWriter(new FileWriter(out));
out.println("Test output");
```

- Numeric values will automatically be converted into strings when output by `print` and `println`. When reading such data back, the wrapper class Integer, Double and Float (as appropriate) will need to be used with their corresponding *parse* methods (`parseInt`, `parseDouble` and `parseFloat`) if any arithmetic or formatting is to be performed on the data.

```
BufferedReader in =
     new BufferedReader(new FileREader("infile.dat"));
String input = in.readLine();
int num = Integer.parseInt(input);
```

`close`() method

- When the processing of a file has been completed, the file should be close via the `close`() method, which is a member of both `BufferedReader` and `PrintWriter`.

  E.g.:

  ```
  in.close()
  ```

- Especially important for output files, to insure that the file buffer has been emptied and all data written to the file. Since file output is buffered, it is not until the output buffer is full that data will normally be written to disc. If a program crash occurs, then any data still in the buffer will not have been written to disc.

Appending to files

- If you wish to *append* data to a serial file, use the following constructor:

```
public FileWriter(File file, boolean append)
        throws IOException
```

  which constructs a *FileWriter* object given a File object. If the second argument is true, then bytes will be written to the end of the file rather than the beginning

- If you construct file with this constructor:

```
public FileWriter(File file)
         throws IOException
```

  if the file already existed, its initial contents will have been **overwritten** (which may or may not have been your intention)

Flushing

- There may be a significant delay between consecutive file output operations while awaiting input from the user, it is good programming practice to use method `flush`() to empty the file output buffer.

  e.g.

  ```
  output.flush()
  ```

- java.io.Writer.flush():

  Flush the stream. If the stream has saved any characters from the various write() methods in a buffer, write them immediately to their intended destination. Then, if that destination is another character or byte stream, flush it. Thus one `flush`() invocation will flush all the buffers in a chain of Writers and OutputStreams.

End of file handling

- Programming languages differ in detecting an end-of-file (EOF) situation:
  - with some, a program will crash if an attempt is made to read beyond the end of a file;
  - with others (Java), you must attempt to read beyond the end of the file on order the EOF to be detected. With Java, we keep reading until the string has a *null* reference.

```
textMark = input.readLine();
    while (textMark != null){
          ...
          textMark = input.readLine();
    }
```

Random access files

- Serial access files disadvantages:
  - not possible to go directly to a specific record (need to physically read past all the preceding records).
  - not possible to modify records within an existing file (the whole file would have to be recreated).
- Random access files, also called direct access files overcome both of these problems, but have other disadvantages:
  - all the logical records in a particular file must be of the same length.
  - a given string field must be of the same length for all records on the file.
  - numeric data is not in human-readable form.
- Random access files speed and flexibility greatly outweighs the above disadvantages.

java.io.RandomAccessFile

- To create a random access file in Java, we create a *RandomAccessFile* object which support both reading and writing to a random access file.

- If the random access file is created in read/write mode, then output operations are also available; output operations write bytes starting at the file pointer and advance the file pointer past the bytes written.

- The methods():
  - *readInt*, *readLong*, *readFloat*, *readDouble*
  - *writeInt*, *writeLong*, *writeFloat*, *writeDouble*.

Some File methods

- public boolean **canRead**().
  - Tests whether the application can read the file denoted by this abstract pathname. Returns true if the file specified by this abstract pathname exists *and* can be read by the application; false otherwise .
- public boolean **canWrite**()
  - Tests whether the application can modify to the file denoted by this abstract pathname. Returns true if the file system actually contains a file denoted by this abstract pathname *and* the application is allowed to write to the file; false otherwise.
- public boolean **exists**()
  - Tests whether the file or directory denoted by this abstract pathname exists. public boolean **isDirectory**()
  - Tests whether the file denoted by this abstract pathname is a directory.
- public boolean **isDirectory**()
  - Tests whether the file denoted by this abstract pathname is a directory.
- public boolean **isFile**()
  - Tests whether the file denoted by this abstract pathname is a normal file. A file is *normal* if it is not a directory and, in addition, satisfies other system-dependent criteria. Any non-directory file created by a Java application is guaranteed to be a normal file.

Redirection

- The standard input stream *System.in* is associated with the keyboard, whereas the standard output stream *System.out* is associated with the VDU.
- Use ‘<‘ to specify the new source of input.
- Use ‘>’ to specify the new output destination.

  E.g.

  ```
  java ReadData < payroll.dat
  ```

  Program *ReadData* begins execution as normal. But, wheneverit executes readLine statement, it will now take its input from the next available line of text in file *payroll.dat.*

  ```
  java WriteData > results.dat
  ```

  Program *WriteData* directs the output of any print and printl statement to file *results.dat*.

Redirection (I)

- Both input and output with same program can be redirected:

```
java ProcessData < readings.dat > results.dat
```

All *readLines* of *ProcessData* program will read from file readings.dat, while all *prints* and *println*s will send output to file *results.dat*.

Command line parameters

- The *java* command allows us to supply values in addition to the name of the program to be executed. These values are called **command line parameters** and are values that the program may make use of.

- Such values are received by method *main* as an array of *Strings.* If this argument is called *arg*, then the elements may be referred to as *arg[0], arg[1], arg[2*]….

  (Java program called Copy.class copies the contents of one file to another)

  ```
  java Copy source.dat dest.dat
  ```

Command line parameters (I)

```
public class Copy{
   public static void main(String[] arg) throws IOException
   {
   //First check that 2 file names have been supplied...
    if (arg.length < 2){
    System.out.println( "You must supply TWO file names.");
    System.out.println("Syntax:");
    System.out.println("java Copy <source> <destination>");
              return;
    }
     …
   // The rest of the code
}
}
```

Interfaces and Anonymous Classes

- *Interfaces* are special types of classes that cannot be instantiated into objects.
  - They are provided only for inheritance i.e. in order to use an interface you must extend it.

- An *anonymous* class is a class that implements (effectively extends to form a class) an interface without requiring its own class definition and class name.
- They are used when the programmer needs to implement an interface to create a class that will only ever be used once.

Interface

- An interface is completely abstract i.e. its methods do not have a body, but must be defined by subclasses Interfaces, therefore cannot be instantiated into objects. They are used to define contracts for classes.

```
interface Being {
    public int getAge();
}
```

- You cannot instantiate an interface, e.g.

```
Being b = new Being();
```

- You can instantiate classes which extend an interface.

```
Person p = new Person("Pat", 55);
```

where

```
class Person implements Being { etc…
```

Interface implementation

```
class Person implements Being {
	private String name;
	private int age;

	public Person(String n, int a) {
		name = n;
		age = a;
	}
	public String getName() {
		return name;
	}
	public int getAge() {
		return age;
	}
	public String toString() {
		return "Person (" + name + ", " + age + ")";
	}
}
```

Interface example

```
public class InterfaceSample {
	public InterfaceSample() {
	   // You can instantiate classes which extend an interface
	    Person p = new Person("Pat", 55);
	    // You can also extend an interface on the fly, by creating an anonymous class which overrides
	// the required method. This approach is used when you are extending an interface to create a class
	    // that you will use only once. If you would reuse the class, it should be properly  defined.
		Being b = new Being () {
			public int getAge() {
				return 0;
			}
			public String toString() {
				return "This is just some instance of a daft anonymous class";
			}
		};
		System.out.println(p.getAge());
		System.out.println(b.getAge());
		System.out.println(p);
		System.out.println(b);
	}
	public static void main(String args[]) {
		new InterfaceSample();
	}
}
```

References

- Chapter 4, *Introduction to Network Programming in Java* by Jan Graba
- Chapter 4: Coulouris, Dollimore and Kindberg,
  Distributed Systems: Concepts and Design
- http://docs.oracle.com/javase/7/docs/platform/serialization/spec/serialTOC.html