Design of Information Structures - Lab 1

January 12, 2019

Introduction

All the laboratories will follow the same format. You will spend two hours working through a worksheet. To gain credit for the labs, which carry 10% module credit, you will be asked to discuss and demonstrate your results.

Note that:

- The lab worksheets are always available to do outside the supervised sessions.
- Each lab consists of a number of practical exercises, all of which are required for marks to be attained. In particular, you should try to accomplish the 3 labelled milestones.
- Each lab is worth 2% of the final course.
- Attendance is compulsory and will be monitored on Tabula. If you are unable to attend a session you must inform a tutor in advance.

The tutors are there to help you think through any issues — but they won't write the programs for you!

The normal rules of assessment apply: if you can't make a laboratory or miss one, you must have a good reason. If the problem is medical then a doctor's note must be presented as evidence. You should attempt to make up for lab absences in your own time.

Although you will be allowed to submit work in the lab a week after it was given, we **strongly** advise you to have completed it before entering the next lab. For example, the work for this lab (lab 1) can be submitted at the start of the lab next week (lab 2), but do not attempt to complete work for lab 1 *during* lab 2. Marks will not be awarded beyond the next lab, meaning lab 1 will not be marked during labs 3, 4 and 5.

Finally, please attempt to use the terminal/command prompt over file browsers during these labs. When a lab tutor asks "what directory are you in", or "which files are in the current directory", show them the outputs of `pwd` and `ls`.

Laboratory 1

In this first Design of Information Structures Laboratory, we will look at some basic data structures: arrays and array lists. These are some basic building blocks for several other data structures we will cover later in the course.

We will start by:

- making a work space for the laboratories
- copying the relevant files for this laboratory
- compiling and running the first example Java program.

Even if you think you know what you are doing, take your time. Follow carefully, step-by-step.

Exercise 1) Log on to the system, if you haven't done so already and open up a terminal window.

Exercise 2) In your home directory, make a new directory called `cs126` and change to that directory:

```
$ mkdir cs126
$ cd cs126
```

Exercise 3) Now make 5 new directories for this laboratory and the ones to come:

```
$ mkdir lab1 lab2 lab3 lab4 lab5
```

Exercise 4) Today, we will do all our work in `lab1`. So change to that directory:

```
$ cd lab1
```

You can find out what directory you are in by entering the pwd (print current directory) command e.g.

```
$ pwd
/dcs/18/u1812345/cs126/lab1
```

where the prefix to your home directory (`/dcs/18/u1812345`) will depend on who you are. If you get lost then using `cd` on its own returns you back to your home directory.

Exercise 5) Now copy the files for this laboratory from the cs126 web site (download file `lab1.zip` into the `lab1` directory and unzip it):

```
$ unzip lab1.zip
```

Exercise 6) List the files by using `ls`:

```
$ ls
BlueChair  StringArray.java  WordList.java ...
```

0.1 Linux Commands

Just as a quick reminder, these are some of the Linux commands you may need to use.

`cd`	Return to home directory
`cd` *dir*	Change to given directory
`cd ..`	Change to parent directory denoted by ".."
`mkdir` *dir*	Make a new directory
`pwd`	Print current (working) directory path
`cp` *src destn*	Copy file from source file *src* to directory or file *destn*
`cp` *src* .	Copy file from source file *src* to current directory denoted by single dot "."
`ls`	List files in current directory
`ls -l`	List files in long format (file permissions, modifications, etc.)
`more` *file*	Display the contents of given file
cmd `| more`	Pipe the output of *cmd* through more
`man` *cmd*	Get Linux help from system manual *cmd* (try `man man`)

Reading words from a text file

Exercise 7) In addition to the Java files, you should also have files called `BlueChair` and `coapm` in your directory. Look at either file — you'll see that it's the words to a song. Not surprisingly, there's a lot of repetition, especially when the chorus is repeated.

Exercise 8) Now open `Words.java`, which is a program to read in a given file and scan it word-by-word. The main method looks like this:

```
public static void main(String args[])
```

```
{
    try
    {
      // try some data input
    }
    catch (IOException e)
    {
      // print exception if there is one
    }
}
```

The `WordReader` class opens up a file given by the first argument to the program, reads the words and prints them out, each on a new line.

Exercise 9) Compile `Words.java` and run it as follows

```
$ javac Words.java
$ java Words coapm
EARTH
WHAT
A
SUFFERATION
YEAH
SUFFERER
I
WOULD
RATHER
...
```

You might want to pipe the output through less to stop it scrolling off the screen.

```
$ java Words coapm | less
```

So as expected each word of the song is printed out on a new line. To exit less, press 'q'.

No IDE?

Ok, so you've gathered by now that there's no fancy IDE (Integrated Development Environment), like NetBeans, Java Studio, or Eclipse, to help you with the programming in these labs. To make life a little easier for yourself, if you haven't already done so, you might like to arrange your desktop to have just 3 things open and visible.

- A browser displaying the lab worksheet.

- You favourite program editors (`kwrite/kate/atom` are good as they do syntax highlighting and enables multiple files to be opened.)
- A terminal window in which to enter compile commands like
  `javac Words.java`
  and to run the programs and see results, like
  `java Words BlueChair | more`
  Remember that you can get quickly to the previous command by using the up-arrow key or *ctrl-p*. You can search for previous commands using *ctrl-r*

You can then easily cycle round the windows by (1) reading and assessing what you have to do from the worksheet, (2) opening the relevant java files, making and saving changes etc., (3) compiling and seeing the results.

Exercise 10) Look again at the `Words.java` program, especially at the `while()` loop. Can you make sense of this statement:

```
(str = in.readWord()) != null
```

Why does it work and what does it do?

Exercise 11) Now, you've not been told how the `WordReader` class works, but to use it all you need to know is that

1. The class constructor

```
public WordReader(String filename)
```

takes 1 String argument, which is the name of the file to be read.

2. The class method

```
public String readWord()
```

reads the next word in the file, or returns a null string if the end-of-file is reached.

3. The file is closed by the method

```
public void close()
```

We shan't be concerned about the exception handling of `WordReader`.

So you can see, that it's possible to use a class by only knowing its functionality (what it does), rather than its implementation (how it does it).

It's time to do something useful with our `WordReader` class and start using ADTs called *associations*.

Simple text processing

We can now use the `WordReader` class to write programs for text processing. In the remainder of this laboratory, we will look at how basic data structures can be used to develop a program to count the frequency of unique words in a given piece of text, a bit like a dictionary.

To build this program we need to be able to

- read separate words from a given file (using `WordReader`),
- keep a list of unique words in that file, and
- keep a count how often the word is used

Arrays

An array is a collection of elements of a given type (e.g. `int or String`) which can be indexed. You should already be familiar with arrays in Java.

For example, an array can be declared, initialized and its values set by

```
int[] a; // Declaration of an array called a

a = new int[10]; // Initialization of a to an array of 10 ints.

for (int i=0;i<10;i++) {
    a[i] = i*2; // Set value position i of array to i*2
}
```

We can allocate an array of any type. An array of four strings can be allocated by

```
String[] s =  new String[4]; // array of 4 strings
String name = "Freddy"

s[0] = "Hello";
s[1] = name;
// element 2 left to default value (null)
s[3] = null; // element 4 set to null
```

Note this array declaration only says that we want to hold 4 references to values of String object. The first element (pos 0) of the array is set to “Hello” and the second (pos 1) to the value of name (which is “Freddy”. The third element (pos 2) is not assigned and takes a value of null, and the final element (pos 3) is set to null.

Exercise 12) Look at program `ArrayTest.java`. The last instruction in the main method is

```
System.out.println(words);
```

What does this print out? Modify it to print out each element of the array on a new line. Compile and run the program to check that this works.

Milestone 1) You have now completed Milestone 1. You should get this signed off by your lab tutor.

Exercise 13) What happens when you try an access an element outside the index range of the array?

Exercise 14) The problem with arrays is that once they are allocated, it is hard to extend them and it is cumbersome to insert an element in the middle.

For example, having added the first 4 words into words, how would you

(i) add new words on to the end of the array “We are the eggmen”
(ii) insert the word “red" between “the” and “eggman”
(iii) delete the word “am” from second element

How would a second temporary array help perform operations (i) and (ii)?

A Primer on Java generics

When we make data structures, we sometimes know that they will be used for one purpose only. In `StringList.java` is an implementation of an array list that is built only for Strings. If, later, we wish to store other types of objects such as `Cat` objects or `ints`, we will have to edit this code and re-compile it. A better solution would be to have data structures which could store any data type we needed. We could store everything in arrays of Objects, but in doing so we would loose type information, and this can lead to some very nasty bugs.

Java generics overcomes this issue by allowing the data structure to become multi-purpose. They allow us to tell the compiler what we intend to store in a data structure. For the ArrayList data structure in Java takes a generic argument:

```
// An array list that will store String objects
ArrayList<String> astringlist = new ArrayList<>();
// An array list that will store Integer (int) objects
ArrayList<Integer> anintlist = new ArrayList<>();
```

With this code (particularly the bits inside `<>`), it knows that `anstringlist` will store String objects. It also knows that `anintlist` will store Integer objects.

If we attempt to put another type into either of these lists, such as a `Cat`, an error will be thrown.

Generics allows us to build multi-purpose data structures that are capable of storing many types. In `MyArrayList.java` is the start of an implementation of a generic array list. We can use this list in much the same way as the above example, and we can store all types of objects in it without editing or re-compiling this code.

For more information (outside of this lab) on Java generics please see the tutorial at:
https://docs.oracle.com/javase/tutorial/extra/generics/index.html

ArrayLists in Java

Arrays in Java are dynamic in the sense that we can specify their size when we create them during runtime, but they cannot grow with the demands of the program and it is hard to perform insertions and deletions because you have to shift existing elements.

The `java.util package` contains a class called `ArrayList` which can be used whenever you need a useful extensible array.

Exercise 15) Look at the package documentation on `java.util.ArrayList` at:
https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html (there is a link off the course web page to all the standard Java packages).

Exercise 16) Look at file `ArrayListTest.java`. It uses `java.util.ArrayList` instead of arrays. This time, the print statement works as we might have expected, because ArrayList implements a `toString()` method. What does this mean? You should use a search engine to find the Oracle documentation on `toString()` methods to find out more.

Exercise 17) Make changes to this program to implement the operations (i)-(iii) indicated in the previous section on arrays. You will need to know which methods to use to insert an element at a given point and to delete an element at a given point.

Confirm the result by printing out the contents of the array list after each operation (i)-(iii).

Exercise 18) Look at the file `IList.java` files. It contains the interface for a list, which we will now implement. The methods it defines are operations that we would expect a list to have, including `add()`, `get(int)`, `size()`, and others. The comments above the methods explain what each operation does, and the expected return values should be.

The `MyArrayList.java` is the start of this implementation, and we will now finish the methods marked `//INCOMPLETE`.

Exercise 19) For the `add()` method you will have to keep track of how many elements have been filled and if the new element causes this to go past the length of the current array, you will have to allocate a new array and copy the existing and new elements to it.

What strategy will you use extend the array capacity:

(i) increase by 1 element each time?
(ii) increase by n elements each time (n>1)?
(iii) increase by n*length each time e.g. for n = 2 it would double capacity each time?

What are the advantages and disadvantages of each of the extension strategies (i)-(iii)?

Exercise 20) Test out your `add()` method using the `ArrayListTest` program (remember to change the `ArrayList` to `MyArrayList` at the start of main()). You may want to comment out your answers to (b) above.

Exercise 21) Implement the `contains()` method, and test it in the `ArrayListTest` program by printing the results of `words.contains("eggman")` and `words.contains("asdf")`.

Milestone 2) You have now completed Milestone 2. You should get this signed off by your lab tutor.

Sets

Look at the `ISet.java` file, an interface for a set implementation where only unique elements are stored. In Set.java is a skeleton implementation of ISet, which you will now complete using the `MyArrayList` class.

Exercise 22) In `Set.java` the implementation of `add()` is incomplete. It should check the current list of words and only insert the word if it is not already in the list (i.e., unique). Modify this method so the post-condition is satisfied.

Exercise 23) Complete the `toString()` method and write a main method to test your Set implementation. You may wish to look at `Words.java`, and edit this file to create a Set object to add words to as they are read from the file.

Milestone 3) You have now completed Milestone 3. You should get this signed off by your lab tutor.

Exercise 24) Write some code to print out a random 10 words from a set. Again, you may wish to add this to the `Words.java` file

Exercise 25) How would you use your MyArrayList, and ideas from your Set implementation, to keep a frequency count for each word? The `ICounter.java`, `Counter.java`, and `Count.java` are relevant here. `Counter.java` is the start of an implementation for a counts collection, you will now complete its methods marked with `//INCOMPLETE.`

The `getCount` method is not yet implemented, while the `add()` method is missing two lines in the `if` statements. The `size()` method is currently implemented incorrectly, and you should fix it to match the interface description.

Exercise 26) Use this Counter class and modify your random words code from part (c) to select a word based on its likelihood, i.e., the probability of a word is its frequency divided by the total number of words examined. Hence, print out a random sequence of 10 words based on their likelihoods of use.