HD Task 11.1 High Distinction Project

By Paul Watts, 569887 October, 2020

Having not done any serious Programming in almost 40 years, and never having attended University, taking on a Bachelor of Information Technology and Communications degree, and having a Programming unit as my first subject, has been quite the challenge.

While frustrating at times, due to the inherent challenges of Programming in general, the KIT101 Programming Fundamentals course has reignited my passion for Software Development and given me immense satisfaction as I've worked through and mastered the progressively harder Programming Portfolio tasks.

Learning Java, and the principles of Object Oriented and Functional Programming, has been a revelation and while I've only scratched the surface of Java in this course it has kindled my desire to learn a lot more.

While there are an enormous numbers of areas I wish to explore further one that I did explore to complete my HD Programming Project is the ability to have dynamic Arrays to remove the restriction of fixed sized Arrays in my DN level program.

Also, in all of the Portfolio tasks, data was never persisisted, so all of the data disappeared when the program ended.

My HD Programming task involves working with Products similar to what you would encounter in a store such as JB Hi-Fi. The Distinction level Program used an Array of Product objects, with Arrays in Java being a fixed size determined at initialisiation, and the data disappearing once the program ended or crashed.

Not only is this a poor way to treat useful data it was also annoying to have to continually enter Products in order to test the program.

To achieve both my goals for the HD program, I learnt a lot, mostly the hard way, about the Java Collections Framework, and Java Serialized Objects .

By hard way I mean not fully understanding how to use these libraries and methods in my program but knowing the end result I wanted and persevering until I solved the nearly endless stream of problems and bugs until I had them working.

On multiple occassions there were a few hours spent to produce one line of code, e.g. `Collections.sort(products, new ProductSkuSorter()); // Sort into sku order`.

While still only having a basic understanding of the above topics the purpose of this Tutorial is to assist my fellow novice Programmers in going beyond what we have been taught in KIT101 and putting these very useful Java Libraries to work.

After much research, and trial error with files types such as .CSV files, I zeroed in on the Java Collections framework and Java Serialized objects (American spelling as per the Documentation).

The Collections Framework is an extremely useful set of Java Libraries with the most basic interface implemented using `import java.util.Collections` with a number of inherited classes that can be imported for more specialised functionality.

A *collection* in this use case is defined as an object that represents a group of objects and a *collections framework* is a unifed architecture for representing and manipulating collections.

The immediate benefits of using the Collections Framework was solving the fixed size Array problem by using an ArrayList which is a resizable-array implemention of the List Interface . I experimented with using a Linked List which would have also suited my purposes but while there were pros and cons for each List type, mainly to do with performance, I settled on an ArrayList. Something to note here is that these Lists permit all elements, including null, which can cause problems if you allow null objects to be loaded inadvertently.

You implement an ArrayList using `import java.util.ArrayList;` and in my case, for products, instantiated by `products = new ArrayList<Product>();`, with Product my own specialised data Object class.

You can allocate an initial size for the ArrayList by including a size in the `()` but unlike an Array it will automatically expand when it reaches this size. I believe the default size is 10 elements and will then expand by 10 each time it resizes.

For efficiency I set the size to 100 which will expand by 100 each time it resizes. An ArrayList will be slower expanding than a LinkedList but the performance difference is fairly negligible unless there are a large number of elements.

The Java HashMap is another useful data construct and while not used in this Project is one that I can see I will be using for future Projects.

Unlike an Array, to use a Product object in my program, using an ArrayList, it had to be "cast" from it's generic object type, inherited from the List interface, back into the Product object inherited from the Product class.

I encountered many problems before I understood this simple fact.

This was achieved by `product = (Product) products.get(valid); // Cast the ArrayList object into Product` where `valid` is a valid index position in the ArrayList of products.

This "casting" concept is important also for our next topic which is Java Serialized Objects

With appropriate runtime error handling and exceptions, File input and output is relatively easy to implement in Java, especially using Serialized objects.

Coming from my misspent youth of procedural programming and tightly coupled record file definitions, serialization in Java was a very pleasant revelation and fascinating concept.

Simply put, to serialize an object means to convert its state to a byte stream so that the byte stream can be reverted back into a copy of the object. These serialized objects can be written to a file and then loaded again and deserialized back into the desired object, in my case, products.

Perfect for my purposes, where I wished to persist objects rather than have them disappear when the program ended.

The starting point was to implement the Serializable interface by `import java.io.Serializable;` in my Product class and declaring it Serializable by `public class Product implements Serializable;`. From there it is basically a "black box" that does all the heavy lifting in Serialization, which is the beauty of using the Java libraries.

To use these objects for Input and Output we implement the Java IO library by the statement `import java.io.*;` which provides classes for system input and output through data streams, serialization and the file system.

As mentioned previously it is important that proper error handling is implemented through Java Exceptions `try` and `catch` when implementing these IO classes and in fact there were compile errors (not just runtime errors) when certain errors were not being caught.

The expected errors to catch included `FileNotFoundException` if the file does not exist and `EOFException` when hitting the end of the file when reading from a File.

Remembering that serialized objects are being written and read, we need to cast them to our Product class in order to use them. An example statement in my method that reads these objects is `product = (Product) objectinputstream.readObject(); // Explicitly cast the Product`

Interestingly the Object Oriented Paradigm is evident with File operations (as with all things in Java) as we intstaniate new objects which inherit the attributes and behaviours of the java.IO.* classes to perform the work we need. Examples statements from my reading method `private void bulkLoadProduct() {}` are:

```
FileInputStream streamIn = new FileInputStream("products.ser");
 objectinputstream = new ObjectInputStream(streamIn);
```

Similary objects are instaniated in my writing method `private void bulkSaveProduct() {}`:

```
FileOutputStream writeData = new FileOutputStream("products.ser");
ObjectOutputStream writeStream = new ObjectOutputStream(writeData);
```

The more I have become accustomed to Object Oriented Programming in KIT101 the more I can see the power of this concept and how it is being adopted not just in Java but in new languages such as Dart where everything is an object (even primitive data types).

There are many benefits from working with serialized objects, compared to tightly coupled record definitions, which require code changes when the record definitions change.

For example I was able to add a new Enum to my Product class, a Product Category, with zero changes to my file IO methods.

There are also many advantages in using List interface methods over Arrays, in addition to the fact that they can grow dynamically.

For example there was no need to keep track of the next index position to add an element as this was automatically handled by `products.add(product);`. This was something I needed to explicitly handle in my DN level program and something that led to a number of errors initially.

While I only used a fraction of the power of the Collections Framework in my program it was enough to think of use cases on how I will use it more extensively in future projects.

Finally, sorting and searching can be reduced to a single line of code using this Framework.

For example whereas I had a whole section of code to perform an Insertion sort in my DN level program this was reduced to a single line of code in my HD program by `Collections.sort(products, new ProductSkuSorter()); // Sort into sku order` .

Sorts can be accomplished in different orders, e.g. ascending or descending and by sorting by different properties of the desired class by using Comparators or a Comparable Interface.

Similaly a custom written binary search can be replaced by a Collections method binary search in a single line of code.

I hope this Tutorial has encouraged novice programmers to investigate these topics further and to continue to explore the incredible power of the Java Libraries.