Java & OOSE 2

Laboratory Sheet 6

This Lab Sheet contains material primarily based on Lectures 11 – 12.

The deadline for Moodle submission is 17:00 on Thursday 3 November.

Aims and objectives

- Using `ArrayList`s and the Collections framework in Java
- Reading and making use of online documentation of Java library classes
- Designing classes and consolidation of earlier object-oriented concepts
- Implementing interfaces in classes
- Using classes inside a package

Set up

1. Download Laboratory6.zip from Moodle and launch Eclipse.
2. In Eclipse, select File → Import … to launch the import wizard – this will be used to import the starter projects into your workspace. Then select General → Existing Projects into Workspace and click Next.
3. Choose Select archive file and then browse to the location where you downloaded Laboratory6.zip. Select Laboratory6.zip and click Open.
4. You should now see a new project in the Projects window: Submission6_1. Ensure that the checkbox beside this project is selected, and then press Finish.
5. In the *Package Explorer view* (on the left of the Eclipse window), you should now see an icon representing the new project (in addition to any projects that were there before).

Please refer to the lab sheet from Laboratory 1 for additional details on how to open and run the code in the new project. Note that, as in previous labs, you will be creating some new classes as part of the assignment; refer to the lab sheet for Laboratory 4 for details on creating classes in Eclipse if necessary.

Submission material

Preparatory work for these programming exercises, prior to your scheduled lab session, is expected and essential to enable you to submit satisfactory solutions.

Unit Tests: I have supplied some simple test cases to check that your code is working properly, in the package knapsack.test. **Note that the tests may not test all aspects of the program's behaviour – passing all of the tests does not mean that your code is perfect (although failing a test will definitely indicate a problem to be investigated).** You should also read through the specification and make sure that you have implemented everything correctly before submitting.

Submission

You are to design and implement a set of classes that will select a set of items with a limited total weight that has the highest possible total value. As a concrete example, consider a set of four items as follows:

1. Weight 3, value 6
2. Weight 10, value 7
3. Weight 4, value 7
4. Weight 3, value 10
5. Weight 5, value 9

If the goal is to pack these items into a knapsack with total capacity 12, then the best choice is to include items 3, 4, and 5 – this produces a weight of 12 (the limit) and a total value of 26. No other combination of items can do better in this case.

You will implement three possible algorithms for carrying out the knapsack packing:

The ***simple*** strategy goes through the list of items in the order presented, including each item as long as it does not exceed the capacity. Items are skipped if their addition would exceed the capacity. For example, with the list above, this strategy will include items 1, 3, and 4, producing a total weight of 10 and a total value of 23.

The ***brute-force*** strategy considers all possible combinations of items and chooses the set that best satisfies the problem. For the above example, this strategy would choose the optimal items 3, 4, and 5, with a weight of 12 and a total value of 26.

The ***sorting*** strategy implements addresses limitations of the other two strategies above. As the simple strategy always considers the items in the order they are first presented, it can miss possible better solutions (as in the above example). On the other hand, while the brute-force strategy will always find the best solution, it does so by enumerating and testing all of the possibilities; in general, if there are N items in the list, this will involve processing 2^N possibilities, which could be an extremely large amount. For the sorting strategy, we first sort the list of items by decreasing **unit price** – that is, put the items with the highest value of (value/weight) at the start. Once the list is sorted, this strategy proceeds as in the simple strategy – since the items at the start of the list provide more "bang for your buck", this strategy often comes up with a better solution than the simple strategy, but is not guaranteed to find the optimal solution.

For the above example, the sorting strategy would first re-order the list as (4, 1, 5, 3, 2), and the final selection would then include items 4, 1, and 5 for a total weight of 11 and a total value of 25.

Packages

A general note: all classes created as part of this lab should be put in the `knapsack` package, rather than the default package as we have done previously.

Representing an Item

You should create a class `Item` to represent an item. An item should incorporate three private `int` fields: one for the weight, one for the value, and one for a unique identifier. The constructor should take two `int` parameter specifying the weight and value of the item, and should set those two fields and also give the item a unique identifier. You should also implement getter methods for the weight, value, and ID, along with an overridden toString() method that includes the value of all fields in the output.

You can use the provided TestItem class to test the behavior of your Item class. Note that the testCompareTo() test will fail until you have implemented the modifications described later in this lab sheet, but you should be sure that the rest of the tests pass before continuing with the assignment.

PackingStrategy

The knapsack.PackingStrategy interface specifies the methods needed to represent the knapsack packing strategies outlined on the previous page. It includes two methods:

- String getName() – should return a string representing the particular packing strategy implemented
- List<Item> packItems(int capacity, List<Item> items) – implements the given packing strategy on the given problem instance and returns the items selected by the strategy

Note that java.util.List is the Java Collections interface implemented by java.util.ArrayList (and others). In general, we will use the List type rather than the ArrayList type for all variables and parameters in this lab.

SimpleStrategy

The first packing strategy to implement is the ***simple*** strategy described above. You should create a class knapsack.SimpleStrategy implementing PackingStrategy, with a packItems() method that implements the simple strategy of iterating through the list of items and including each only if it does not cause the capacity to be exceeded.

BruteForceStrategy

You should implement the brute-force strategy above in a class knapsack.BruteForceStrategy. You can make use of the provided class knapsack.Utils and its included static method powerSet() to create the "power-set" of the item list – in other words, this method returns a list of all the possible sub-lists of its input list. So for the list [1, 2, 3], it would return the list-of-lists [[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]] – note that this includes both the empty list and the full set of items, as well as all other subsets.

Your BruteForceStrategy.packItems() method should compute the power-set of its input list, and then go through every item in the resulting list to see if (a) its total weight is within the capacity, and (b) if the total value is greater than the best value found so far. At the end, it should return the list consisting of the items that provide the best total value.

SortingStrategy

This final strategy class should implement the sorting strategy specified above. As part of this, you will need to modify your Item class to make it implement the generic Comparable interface – in other words, the class header should be changed to look like this:

```
public class Item implements Comparable<Item> { … }
```

As part of implementing the Comparable interface, you will need to add a method with the following signature:

```
public int compareTo (Item otherItem) { … }
```

This method should compare the current item to the other one in terms of **unit price** (i.e., value divided by weight). The return value should be:

- Less than zero if the current item's unit price is **less than** that of the other item

- Equal to zero if the current items' unit price is **identical to** that of the other item
- Greater than zero if the current item's unit price is **greater than** that of the other item

Don't forget to treat the unit price as a floating-point value – i.e., watch out for integer division! You can use the TestItem test class to test the behavior of your compareTo method as well.

Once you have implemented comparison of Items, you can implement the sorting strategy described above. Here are some building blocks that will be useful (all documentation of the Collections class can be found at https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html)

- You can use the Collections.sort() method to sort a list of items into ascending order using the default comparator.
    - If you want the list to be sorted in the opposite order (as is the case here), you can make use of either Collections.reverse() or Collections.reverseOrder().
- You should be sure not to modify the input list of items while carrying out the packing process (the test cases will check this) – instead, you should first make a copy of the list and then do all sorting/etc operations on the copy. You can copy a list either by using an appropriate ArrayList constructor or by using the addAll() method – see ArrayList documentation at http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Testing your code

The TestItem and TestPackingStrategies classes provide JUnit test cases for your code. You can also use the KnapsackMain class's main method to do more testing; this method first runs the example given at the start of the lab sheet, and then generates a random problem instance and runs all packing strategies on that. If you want to use this tester before you have implemented all packing strategies, you can comment out the lines where the un-implemented classes are constructed to test the ones that are working.

Note that this program also prints out the run-time for all algorithms. Try increasing the value of numItems at line 40 and see how the run-time of the brute-force algorithm increases dramatically – in fact, at least on my computer, if you increase the number of items much over 20, the program runs out of memory and crashes before even completing the brute-force search.

Submission

You should submit your work before the deadline no matter whether the programs are fully working or not.

When you are ready to submit, go to the JOOSE2 moodle site. Click on Laboratory 6 Submission. Click 'Add Submission'. Open Windows Explorer and browse to the folder that contains your Java source code and drag *only* the four Java files Item.java, SimpleStrategy.java, BruteForceStrategy.java, and SortingStrategy.java into the drag-n-drop area on the moodle submission page. **Your markers only want to read your java files, not your class files.** Then click the blue save changes button. Check the four .java files are uploaded to the system. Then click submit assignment and fill in the non-plagiarism declaration. Your tutor will inspect your file and return feedback to you via moodle.

Outline Mark Scheme

Here is a rough mark scheme that your tutor will use when assessing your work.

A: Excellent attempt. Code compiles and runs. Code passes TestItem and TestPackingStrategy unit tests. Correct output for main method.
B: Very good attempt. Code compiles and runs, although may not give correct answers in all cases. SimpleStrategy and/or BruteForceStrategy algorithm implementation is correct.
C: Good attempt at solution. Code may not compile, but you have attempted to implement all specified classes.
D: Minimal attempt at solution. Code may not compile, but you have attempted at least Item and one of the PackingStrategy subclasses.

Within these general guidelines, the tutors are also checking to see whether:

- you use sensible variable names and method names
- you write appropriate comments (Javadoc or similar) for classes and 'interesting' methods
- you write clean and efficient control flow structures (e.g. for each loops) in your code.
- you include an `@author` tag in each source code file
- you are using packages correctly

Possible extensions

The above is all that we expect you to do on this assignment, and is all that you are required to submit. This section describes some possible extensions to this work that should be attempted only by those who are interested – this work is not required, and will not be assessed.

This lab makes use of an instance of a well-known problem in computer algorithms called the **0/1 knapsack problem** – more information is available, for example, at https://en.wikipedia.org/wiki/Knapsack_problem#0.2F1_knapsack_problem.

As stated on that page, the best known algorithm for this problem makes use of **dynamic programming**, and the algorithm is described on that Wikipedia page as well. If you want, you could try implementing a dynamic programming solution to the current problem.

Another possible programming challenge would be to produce a better way of creating the power-set for the brute-force method: at the moment, the whole power set is stored in memory and used, but a more memory efficient implementation would generate the elements of the power set only as they are needed. This would not save on run time, but would allow larger instances to be evaluated without running out of memory.