Software and Programming 2 (SP2)
2024/25: Coursework Instructions

1 Introduction

- **Submission Deadline: 14 November 2024, 14:00 UK time.**

There are **two** parts to the coursework for this module. The coursework assignments contribute to your overall module mark as follows:

- Part 1 is worth 50% of the coursework mark (i.e., 25% of the overall module mark);
- Part 2 is worth 50% of the coursework mark (i.e., 25% of the overall module mark).

The coursework is marked out of a total of 100. The code for this coursework assignment is made available to you on Moodle and on github classroom.
The parts of the coursework are further explained in Section 2 and Section 3 below.
Section 4 presents the deadlines and submission instructions.

2 Description of the work — Part One: CargoBox

2.1 `CargoBox` and `Item`

In this coursework part, we want to write a class `CargoBox`. A class like this might be used, for example, to represent a physical box to contain items from a warehouse for shipping. The instances of the `CargoBox` class can store and provide information about objects of a class `Item` that has already been written. The `Items` stored in our `CargoBox` objects are very simple: they have a name and a weight. A `CargoBox` object can then tell us thing like the average weight of its current items, the number of current items, and so on.

The following example:

```
CargoBox unit = new CargoBox();
unit.add(new Item("Bicycle", 9000));
unit.add(new Item("Scarf", 60));
System.out.println(unit.numberOfItems());
unit.add(new Item("Pen", 35));
System.out.println(unit.numberOfItems());
```

should print:

```
2
3
```

Here, the method `numberOfItems()` returns the number of items that have been added to the CargoBox so far. When we called `unit.numberOfItems()` for the first time, only the first two items, named `"Bicycle"` and `"Scarf"`, had been added to our storage unit, so the result was `2`. When we then called `unit.numberOfItems()` for the second time, the third item, named `"Pen"`, had also been added to our unit, so the result was `3`. Thus, the same method call on the same CargoBox object (e.g., `unit.numberOfItems()`) can have different results, depending on the state of the object.
In this coursework we do not want to analyse any `null` item references. So, the code snippet

```
CargoBox unit = new CargoBox();
unit.add(new Item("Tinned food", 400));
unit.add(null);
System.out.println(unit.numberOfItems());
```

should print:

```
1
```

It is *up to you* as the implementor of the class `CargoBox` whether the method add or the method `numberOfItems` deals with the `null` references that may occur as an argument of add. For the users of your class (who only care about the "behaviour" of its objects, i.e., what effects calling the methods on the objects have), this is an implementation detail that they need not know about. What they (mainly) care about is that your methods always give the right results.
The *public interface* of the class `CargoBox` is already present in the template code on Moodle in the form of headers for constructors and methods and documentation comments describing the desired behaviour. However, the current method *implementations* are "stubs" that allow the code to compile but not to work correctly. Thus, you will need to provide implementations for these methods that work correctly according to the documentation of the public interface of the class `CargoBox`. You will certainly also need one (or more) suitable `private` instance variables — also called fields or attributes — for the class `CargoBox`.

2.2 `Coursework1Main`

The file `Coursework1Main.java` is provided in the github classroom repository. This class makes use of some of the desired functionalities of the class `CargoBox` in the `main` method. You can (and should) test your implementation of `CargoBox` by running `Coursework1Main.main`. These tests provide further clarification for the behaviour that `CargoBox` is supposed to show. It is a requirement that your implementation of `CargoBox` compiles and works with the *unmodified* `Coursework1Main.java` and `Item.java`. You should expect that we will use the original versions of these files to test your implementation of the `CargoBox` class, not the ones that you may have modified!
The file `Coursework1Main.java` also contains a comment at the end with the output that its `main` method produces with our implementation of the `CargoBox` class.
Note, however, that the tests performed by `Coursework1Main` are not meant to be exhaustive — so even if `Coursework1Main` has the desired outputs, this does not automatically mean that your implementation is necessarily correct for all purposes. Thus, it is a good idea review your code and test it with further test cases before handing in your solution

2.3 Coding Requirements

- Only one of the two constructors should explicitly initialise all the instance variables of the class. The other constructor should then just call the first constructor via `this(...)` with suitable arguments.
- All instance variables should be `private`.
- Your implementations of the methods should not modify their arguments.
- None of the methods and none of the constructors in the class `CargoBox` may print on the screen (i.e., no `System.out.println`).
- Every method (including constructors) should have Javadoc comments.

  For Part One, comments for the required methods and constructors are already provided, but if you write additional methods or constructors, you should of course document them.
- Every class you create or modify should have your name in the Javadoc comment for the class itself, using the Javadoc tag `@author`.
- Write documentation comments also for the instance variables (attributes) of your classes. These are useful not only for you at the moment, but also for programmers who may later work on your code. This may well be *you* again, a few years down the line, wondering what you were thinking back in the days when you had written that code.

  (Remember — in this module we are training for "skyscraper-sized" software projects. So, even if your coursework might not consist of thousands of classes and might not be developed and maintained over several decades, **we want you to write code as if it were the case**.*)*

  The other programmers who will later have to modify your code need to know what they may assume about the values of the attributes, and at the same time also what

they must guarantee themselves so that the code in the class will still work correctly after *their* new method has run.

For example, it does not make sense for a `length` attribute to have a negative value. So, your instance methods can assume that `length` is not negative. But then, your (or someone else's!) instance methods in this class must also make sure that `length` is *still* non-negative after they have run (which is not a problem if they don't modify `length`).

- Your source code should be properly formatted.
- Code style: One aspect for Java projects is the use of the `this.` prefix before the name of an instance variable or method. In this coursework, follow a consistent style: either *always* write `this.` when possible for method calls and accesses to instance variables (so in your instance methods you would always write `this.foo()` and `this.bar`), or alternatively write `this.` only when this is *unavoidable*, because a local variable or formal parameter "shadows" an instance variable (then you would always try to write `foo()` and `bar` to access these instance methods and variables in the same class).

  (*Many large software projects impose such style guidelines on their contributors so that the code looks uniform and is easy to read for other project members.*)
- Reminder — use methods (and auxiliary method if required). Do not cram everything into one or two methods; instead, try to divide up the work into sensible parts with reasonable names. Every method should be short enough to see all at once on the screen.

  For the methods in this assignment, their length is probably not a problematic issue. However, do keep an eye on the other methods that you are writing for this coursework — perhaps one of them already does part of the work that you need in another one? Then you could just call that method instead of writing down its code twice. So instead of re-implementing a method of `CargoBox` (or `Item`, or ...) as part of another method, you should just *call* that method whenever it can do part of the work for your method.

2.4 Hints

- Think about the most suitable internal data representation for your `CargoBox` implementation. There are many correct choices, but some of them can make implementing the methods *significantly* easier than others. Take into consideration that when we construct our `CargoBox`, we do not know how many `Items` will be added to it in the future.

  (What Java classes have we already seen in SP2 for containers that can store an unbounded number of elements?)
- In case you are considering to extend (i.e., write a subclass of) a class from the Java API that may already have some of the needed functionality: this solution is very likely to be "more trouble than it's worth". Rather think about whether you can have an instance variable of that class to which you can "delegate" some of the `CargoBox` method calls.

- In this assignment, we are analysing objects of the class `Item`. The class `Item` provides you with a number of useful methods that you can call on `Item` objects.

  We are providing you with the implementation of the class `Item`. Instead of looking at the source code, you can also read up on the methods of the class `Item` in its *API documentation*. It is available in the repository in the file

  `doc/Item.html`

  You do not need to scrutinise the whole documentation of the class `Item` — just try to find method names that look potentially helpful for your task and then read up on what these particular methods do. You will most probably not need all methods in the class `Item`.

  (Reading the API documentation of other people's code that we do not want to modify, but only use *— instead of looking into the actual implementation — is fairly common, particularly when we are dealing with large code bases. In this coursework, we even have the source code of the class* `Item`*. However, in general the source code of the implementation may not even be* available *to us, e.g., because the authors of the code have not shared it with us. This may happen in particular in commercial settings. Then other programmers will just read the API documentation, which* is *available in most cases.)*

- Keep an eye on the way the methods are supposed to deal with `null` as an actual parameter (or as an *entry* of an array that is an actual parameter).

2.5 Marking Scheme: Part One — 50 marks

For this coursework part, we aim to award marks based on *automated testing*. This means that we will compile your file `CargoBox.java` together with the original `Item.java`, and we will run snippets of code similar to the ones in `Coursework1Main`. In particular, this means that your file `CargoBox.java` must compile together with the unchanged `Item.java`.
We will test all public constructors and methods that are present in the file `CargoBox.java` on Moodle.
30 marks will be allocated according to the proportion of tests passed. (i.e., 30 times the number of passed tests, divided by the total number of tests.)

$$30 \cdot \text{number of passed tests} / \text{number of tests}$$

To see whether you are "on the right track", we recommend that you run the `main` method of class `Coursework1Main` and to check whether your outputs are identical to the ones in the comment at the bottom of the file. However, we have not provided all of the tests that will be used for marking. You will need to ensure that your code works correctly according to the specification, in addition to passing the tests that we have provided. in `Coursework1Main.java`.
The remaining 20 marks for Part One are awarded for your coding style and github commit history.
Therefore, the overall marks for Part One are:

- Automated tests: **30 marks**
- The quality of your design, coding style, and comments. **10 marks**
- Evidence that you have regularly committed compiling code to your github classroom repository at appropriate points during the development of your solution. **10 marks**

3 Description of the work — Part Two: Headgear

3.1 Headgear

The first part of this coursework is about an object-oriented data structure for different kinds of *headgear*. An initial analysis has revealed the following properties and requirements.

- Every piece of headgear provides a method to compute its value in GBP (for simplicity represented as a `double` value).
- A crown is a piece of headgear which has the number of jewels as a property of interest. The value of a crown is determined by its number of jewels multiplied by a fixed "value factor" of 200000.
- There are several kinds of *protective* headgear. A commonality among them is that they have a "protection factor", expressed as a `double` value.
- Scooter helmets are protective headgear. A relevant property of scooter helmets is whether they have a visor.

  The value of a scooter helmet is determined by

  $$value = c_1 + p \cdot c_2,$$

  where c_1 is 80 for helmets without a visor and 160 for helmets with a visor, p is the protection factor of the scooter helmet, and c_2 is 400.
- Bobble hats are protective headgear whose discerning attribute is the diameter of their bobble (in millimetres). The value of a bobble hat is the diameter of its bobble (in millimetres) multiplied by 4 and by its protection factor.

This analysis has led to an object-oriented design with several interfaces and classes. The diagram in Figure 1 shows a graphical sketch of this inheritance hierarchy between the classes and interfaces. A box with `<<interface>>` stands for an interface, a box with `<<abstract>>` stands for an abstract class, and a box with neither stereotype stands for a concrete class. The arrows stand for the "extends" or "implements" relation. For example, the concrete class `BobbleHat` extends the abstract class `ProtectiveHeadgear`, and the concrete class `Crown` implements the interface `HeadGear`.

3.2 Your Mission for the Headgear Part

Your mission for the "headgear part" of the coursework is as follows:

1. Complete the below object-oriented modelling. Use sensible names for your classes, interfaces, methods, and attributes, and distribute common attributes to (possibly abstract) superclasses.
2. Implement your model in Java. Recall and use the principles of encapsulation: make attributes private and provide suitable public accessor and mutator methods.

   Whenever you take a parameter in a public constructor or method where certain parameter values do not make sense (e.g., negative values for a diameter), detect this and throw an `IllegalArgumentException` to inform the user of your class.

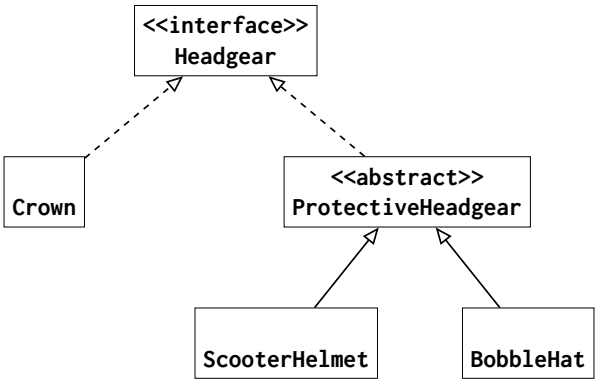

Figure 1: The inheritance hierarchy for our object-oriented data structure representing different kinds of headgear

3. Override the `toString()` method in your classes so that `toString()` will render a string representation of the content of the attributes of this object (including those present in the superclasses that you have written).

4. Implement a `static` method `totalValue` that takes an array of headgear objects and returns the sum of the values of the objects. Your implementation of `totalValue` should *not* use `instanceof` or class casts.

   Here, you should assume that the array reference may *be* `null` (in which case your code should throw an `IllegalArgumentException`), and that the array may also *contain* `null` (also in this case an `IllegalArgumentException` should be thrown).

   The class/interface in which you implement this `static` method is up to you — make a suitable pick.

3.3 Marking Scheme: Part Two

We aim to award marks according to the following scheme.

1. Inheritance hierarchy corresponding to Figure 1 with suitable instance variables and constructors: **15 marks**

2. Instance methods: **10 marks**

3. Class method for computing the total value: **5 marks**

4. Code style and comments: **10 marks**

   This includes (for example) formatting, consistent use of `this`, documentation and Javadoc and `@Override` is used wherever applicable.

5. Regular github commit history with reasonable commit units and messages **10 marks**

4 Submission (both parts)

4.1 Submit using both github classroom and Moodle

The code template for this coursework part is made available to you as a Git repository on GitHub, via an invitation link for GitHub Classroom.

1. First you follow the invitation link for the coursework that is available on the Moodle page of the module.

2. Then *clone* the Git repository from the GitHub server that GitHub will create for you. Initially it will contain the following four files for part 1: `README.md`,`Item.java`, `CargoBox.java`, and `Coursework1Main.java`. It will also contain an empty folder for part two.

3. Enter your name in `README.md` (this makes it easy for us to see whose code we are marking)

4. For Part One, edit `CargoBox.java` according to the requirements of the coursework (i.e., replacing a number of `// TO DO` and dummy implementations of methods with actual code).

5. Add your solution files to the empty folder for Part Two

6. You must also enter the following Academic Declaration into `README.md` for your submission:

   > "I have read and understood the sections of plagiarism in the College Policy on assessment offences and confirm that the work is my own, with the work of others clearly acknowledged. I give my permission to submit my report to the plagiarism testing database that the College is using and test it using plagiarism detection software, search engines or meta-searching software."

   This refers to the document at:

`https://www.bbk.ac.uk/student-services/exams/plagiarism-guidelines`

   A submission without the declaration will get 0 marks.

7. Whenever you have made a change that can "stand on its own", say, "Implemented numberOfItems() method", this is a good opportunity to *commit* the change to your local repository and also to *push* your changed local repository to the GitHub server.

   As a rule of thumb, in collaborative software development it is common to require that the code base should at least still compile after each commit.

Entering your name in `README.md` (using a text editor), then doing a *commit* of your change to the file into the local repository, and finally doing a *push* of your local repository to the GitHub server would be an excellent way to start your coursework activities.
You can benefit from the GitHub server also to synchronise between, e.g., the Birkbeck lab machines and your own computer. You *push* the state of your local repository in the lab to the GitHub server before you go home; later, you can *pull* your changes to the repository on your home computer (and vice versa).

Use meaningful commit messages (e.g., "Implemented numberOfItems() method"), and do not forget to *push* your changes to the GitHub server! For marking, we plan to *clone* your repositories from the GitHub server shortly after the submission deadline. We **additionally** require you to upload to *Moodle* a zip file that contains the folder with your modified `README.md` and a folder containing your working copy and your local Git repository (which should be identical to your repository on GitHub Classroom). One reason is that the time of the upload will tell us if you would like your code to be considered for the regular (uncapped) deadline or for the late (capped) deadline two weeks later. (There is also the (unlikely) case that the GitHub servers have a data loss — then Moodle would provide us with an alternative way of accessing the submission version of your code and your local repository.) Please submit your repository with your commit history.

4.2 Deadlines

- The submission deadline is: 14 November 2024, 14:00 UK time.
- The late cut-off deadline for receiving a capped mark is: 28th November 2024, 14:00 UK time.