University of Exeter

College of Engineering, Mathematics and Physical Sciences

ECM1410

Object-Oriented Programming

Continuous Assessment

Date Set: 14th February 2020
Date Due: 13th March 2020

This CA comprises 40% of the overall module assessment.

This is a **pair** exercise, and your attention is drawn to the guidelines on collaboration and plagiarism in the College Handbook (`https://student-harrison.emps.ex.ac.uk/`).

This assessment covers the use and implementation of a range of object-oriented concepts using the Java programming language that you are covering in ECM1410. The assignment is *summative*. Please ensure you read the entire document before you begin the assessment.

Note, as indicated in the earlier *"ECM1410 Object-Oriented Programming Development paradigm in summative CA"* document, there is also a deadline of midday on 21st February for letting the coordinator know if you are having difficulties contacting your partner, or getting them to contribute.

1 Development paradigm

The summative CA for ECM1410 is a pair submission — with details as circulated previously in the document *"ECM1410 Object-Oriented Programming Development paradigm in summative CA"*. To reiterate the key points from this earlier document:

- The expectation is that the submission will be weighted 50:50 between pair members. In the unusual circumstance that members of the pair do not contribute 50:50, you have the opportunity to indicate a different ratio you have both agreed on the cover page of the EBART submission, up to a maximum divergence of 60:40.
- Pair programming is categorically **not** two developers working separately on two different machines. Side-by-side communication developing on a single machine is a key aspect of the approach.
- **The module lecturing team reserves the right to split pairs where one student is not engaging with the coursework. The coordinator also reserves the right to assign non-contributing students a mark of 0**. In the rare situation that you are paired with a student who is not contributing (e.g. not replying to emails and/or not meeting up for pair-programming sessions) you must inform Prof. Fieldsend of the situation **within one week of release of the CA specification** to facilitate the aforementioned splitting of pairs if necessary. Both parties of a split pair will be assigned an individual variant of the CA (however, if there are multiple pairs in this situation it may be possible to reform pairs consisting of participating students, and of non-participating students). It is not permitted for a student working on the *individual* variant of the CA to collaborate with any other student.

  Given the above process and timelines, please ensure that you arrange to meet up and start the work as soon as the CA specification is released to reassure yourself that you are partnered with a student who wants to actively contribute to the coursework. It is an expectation that pairs have *at a minimum* four pair-programming sessions in the first week of release of the CA.
- It is not permitted for students working on the pair programming assignment to collaborate on the assignment with any other student *apart* from their named partner. Those doing so will be subject to academic misconduct regulations and penalties. Please refer to the undergraduate handbook for details on collusion, plagiarism, etc (see web link on coversheet of this document).

2 Assignment

This assignment is based around the development of a back-end Java package for a client who has already determined the functionality they require of the system, and have provided an *interface* to be used between the back-end you develop, and the front-end they have developed. They should be able to simply compile in the jar file of the package you develop, with the rest of their system, to result in a fully functioning solution.

2.1 The problem

Erica and Thea run an independent bean bag store, specialising exclusively in artisanal bean bags. Within their shop they stock the bean bags they have for sale, which are periodically replenished as bean bags are sold and new bean bags are acquired. They want an electronic implementation of a stock system so that they can manage the bean bags they have in stock more effectively (and keep track of popularity, how soon

to restock a particular line of bean bags, etc). Erica and Thea already have a front-end developed — however they need the back-end of the system to be completed in a Java package. They do have an number of initial members of this package, which they have provided.

In their current (paper based) stock system they keep track of a number of attributes for a bean bag: the manufacturer, the bean bag name, the price, an ID number (an eight character string holding a positive hexadecimal number)[1], the year of manufacture, the month of manufacture, and an optional free text component containing any additional information they think may be useful to their customers (e.g. on the free trade origin of the beans, or some quirky information regarding famous owners of that particular brand).

Sometimes customers come in and browse the stock, find a bean bag they wish to purchase, but don't have the money to buy it at that point. To help these customers, the back-end system should also permit reservations to be made. Reservations cannot be made on bean bags that are currently not in stock, as Erica and Thea cannot guarantee to find future stock of some items, and don't like to disappoint their customers by promising something they may not be able to deliver. Once a stock item is reserved, it can only be sold to the customer with the matching reservation number (unless the reservation is cancelled).

The shop uses the manufacturer provided ID to distinguish any stock in the system. Two bean bags with the same ID must match on name, manufacturer and free text component — and this needs be checked by the system against existing stock to flag up any data entry errors (which can lead to disappointed customers later). Bean bags with the same ID are therefore interchangeable from a stock viewpoint. Note however, it is possible for two bean bags with the same manufacturer, brand, and free text, to have *different* manufacturer IDs (as they may vary in e.g. colour, hence the assignment of a different ID by the manufacturer).

Erica and Thea want the back-end of the new electronic system developed to be based on their existing paper system, as such they have already designed a Java interface for the new system, which their front-end application will use. You are to develop a class that implements this interface, and also develop with the necessary addistional supporting classes in the Java package called `beanbags`. The operational correctness of the back-end system will be tested through this provided interface on submission.

Your task is to design and write the additional package members to complete the `beanbags` package. You will need to to design and write a class that implements the `BeanBagStore` interface (available on the ECM1410 study resources page, as well as at the end of this document). This implementor class **must** be a public class called `Store`. If it is not, then the front-end system will be unable to compile with your back-end solution, and the operational component of your mark will be **0**. You will need to also write any other package members you deem appropriate to support this class and its functionality. All classes developed must reside in the `beanbags` package.

Alongside the interface, I have provided in the package a set of exception classes which the interface requires, and also an `ObjectArrayList` class that allows you to maintain a list of items (including adding to a list, removing from a list and querying a list contents). You should find the `ObjectArrayList` class useful when maintaining lists of objects in your solution, especially given the package import restrictions of the assignment detailed later. Note that as the `ObjectArrayList` stores object references declared as the `Object` type, you will need to employ *casting* when using its contents.

2.2 Development considerations

The following points should be noted:

[1] The various integer type wrapper classes offer overloaded 'parse' methods that convert `String`s to the wrapped type. One of these has two arguments, with the second argument being the radix (base) used. You may find these methods useful.

- Your source code should include appropriate comments and assertions.
- The stock of the store includes all products currently on the premises (those available for immediate sale and those reserved).
- Erica and Thea live in a utopian city with no thieves, so stock reduction due to theft does not need to be modelled by your system.
- If a bean bag is reserved, and prior to it being sold the price is changed, the lowest price the bean bag reaches should be used on selling (i.e. the buyer should be advantaged, not disadvantaged, by any price changes when stock is reserved).

You will not need to submit an executable application (i.e. you do not need to submit a class with a `public static void main` method which uses the `beanbags` package). This notwithstanding, it is strongly advised that you do write an application to test that your package conforms to the requirements prior to submission.

Apart from the classes you develop yourself, or that you have been given as part of the CA, you must only use those available in the `java.lang` and `java.io` packages. The use of any other packages will result in a **penalty of 10 marks**.

Your should consult the `BeanBagStore` interface for a more detailed description of expected behaviour of a class which implements that interface (provided in the JavaDoc).

3 Submission

Both pair members must submit a pdf document using EBART. This submission must include:

- A cover page which details *both* of the student numbers of the corresponding pair. If, unusually, you have agreed a split which is not 50:50, this page should also detail how you would like the final mark to be allocated to the pair, based upon your agreed input. This cannot exceed 60:40. As the submission is anonymous, again please use your student numbers. Failure to submit any coversheet detailing both members student numbers will incur a **penalty of 5 marks**.

The EBART pdf submission of the first pair member listed on the cover page should also include the following:

- A development log, which includes date, time and duration of pair programming sessions, including which role(s) each developer took in these sessions, with each log entry identified by both members using your student numbers to ensure anonymous marking. If you are working individually for parts of the project you are not conducting pair programming — therefore all log entries must include both students numbers. Failure to submit a log will incur a **penalty of 5 marks**.
- A printout of the source files **the pair** have written for the `beanbag` package (i.e., **not including** the classes and interface that have been provided to you by the ECM1410 team as part of the coursework). Source printouts **must** include line numbering.

Additionally the first listed pair member should also submit a copy of your **full** finished package, using electronic submission at `empslocal.ex.ac.uk/submit`, to the folder 'ECM1410 - beanbags', **in a jar file** named `beanbags.jar`. The jar file must include both the bytecode (*.class*) **and** source files (*.java*) of your submitted package, including the `BeanBagStore` interface, the `ObjectArrayList` and all the exception classes provided to you as part of the CA. I.e. it should be a complete self-contained package, that my test program can interact with, via your `beanbags.Store` class.

Succinctly:

Pair member 1: Submits the full pdf document (comprising coversheet, development log and code printout) to EBART, and the jar file containing the bytecode **and** source files to `empslocal.ex.ac.uk/submit`, to the folder 'ECM1410 - beanbags'.

Pair member 2: Submits just the single-page pdf coversheet via EBART.

4 Advice

1. Do not jump straight into the coding: take time to consider the design of your solution first. Think about the objects that you will use, the data they will contain, what the methods they should provide are (in addition to those mandated via the interface), how they relate to one another, etc. Once you are happy with your design, then start programming. Don't be afraid to reassess your design as you go through, but check on the implications of making a changes on all the other objects in your system that use the changed part (this is where one of the strengths of pair programming will come in, in being able to discuss the design implications).

2. Check your objects behave as you intend — use a testing application and use assertions.

3. Slowly fill out functionality — it is far better to submit a solution that supplies most but not all of the required operations correctly, rather than one that doesn't provide any/doesn't compile, as a submission which does not provide any correct functionality at all will get a 0 for the operation criteria. Start off with a `Store` class that compiles and slowly (incrementally) add functionality. I have provided a class that implements `BeanBagStore` on ELE (called `BadStore`) that does just this — it compiles, but provides none of the correct functionality.

4. Keep copies of your working code. If the worst happens and you had a version that worked on 50% of the operations and you've made changes that seem to have broken everything, it is useful to be able to 'roll-back' to the earlier version and try again.

5. **Do not** change the interface and classes that I have provided for you. If you change them, the markers will not be able to compile my codebase with your submission, and you will receive an 'Operation' component mark of **0**, as the interface will not be able to connect to the front-end of the system.

I would suggest adhering to the following (rough) development plan to keep on track and avoid a crunch situation near the hand-in deadline:

Week 6 Analysis and initial work: break the problem down with your partner, consider the different aspects of the problem and the objects you will need, what their attributes and methods should be. Agree on the development environment to use, what your regular pair-programming slots will and block out these times in your calendars going forward. Start work on writing the supporting classes you'll need, which will be used by the `Store` class.

Week 7 Initial `Store` class implementation: interfaces, packages, jar files, design by contract and assertions are all covered this week in lectures, so you should start fleshing out the functionality of the `Store` class that you've discussed conceptually in week 6. You should also check you can construct a jar file, and use its contents.

Week 8 More implementation and testing: Serialisation and I/O are covered this week in lectures, so you should aim to get the rest of the functionality and testing of the system completed in time to work on the saving and loading functionality of the the store at the end of week 8.

Week 9 Final testing and fixes (of code and pdf documents) ensuring submission in good time for the deadline (see coversheet of this document).

5 Marking Criteria

This assessment will be marked using the following criteria.

Criterion	**Description**	**Marks Available**
Comments & annotations.	The degree of quality and appropriateness of documentation comments, code comments and annotations.	/10
Java conventions.	The degree of adherence to Java naming conventions and formatting. See lecture notes and e.g. `https://google.github.io/styleguide/javaguide.html`	/10
Efficiency & flexibility.	The degree to which the solution is efficient and objects are flexible for reuse.	/10
Operation.	The degree to which the provided `Store` class operates as required, as supported by the package members. Submission of a jar file that cannot be compiled in with the test code (due to e.g. the interface definition being changed, require package members missing, etc.) will receive an operation mark of 0.	/40
OO design.	The degree to which the code object-oriented, well structured and presented, with a coherent design and clear and appropriate management of object states, with well encapsulated objects, appropriate distribution of computational load across objects and appropriate use of types and assertions.	/30
Pair Penalty.	Use of non-permitted packages.	−10
Pair Penalty.	Non-submission of coversheet with pair membership details.	−5
Pair Penalty.	Non-submission of development log.	−5

6 BeanBagStore.java

```
1  package beanbags;
2  import java.io.IOException;
3
4  /**
5   * BeanBagStore interface. The no-argument constructor of a class
6   * implementing this interface should initialise the BeanBagStore
7   * as an empty store with no initial bean bags contained within it.
8   *
9   * @author Jonathan Fieldsend
10  * @version 1.3
11  */
12
13 public interface BeanBagStore
14 {
15
16     /**
17      * Method adds bean bags to the store with the arguments as bean bag details.
18      * <p>
19      * The state of this BeanBagStore must be be unchanged if any exceptions are
20      * thrown.
21      *
22      * @param num               number of bean bags added
23      * @param manufacturer      bean bag manufacturer
24      * @param name              bean bag name
25      * @param id                ID of bean bag
26      * @param year              year of manufacture
27      * @param month             month of manufacture
28      * @throws IllegalNumberOfBeanBagsAddedException   if the number to be added
29      *                           is less than 1
30      * @throws BeanBagMismatchException if the id already exists (as a current in
31      *                           stock bean bag, or one that has been previously
32      *                           stocked in the store, but the other stored
33      *                           elements (manufacturer, name and free text) do
34      *                           not match the pre-existing version
35      * @throws IllegalIDException   if the ID is not a positive eight character
36      *                           hexadecimal number
37      * @throws InvalidMonthException    if the month is not in the range 1 to 12
38      */
39     void addBeanBags(int num, String manufacturer, String name,
40     String id, short year, byte month)
41     throws IllegalNumberOfBeanBagsAddedException, BeanBagMismatchException,
42     IllegalIDException, InvalidMonthException;
43
44     /**
45      * Method adds bean bags to the store with the arguments as bean bag details.
46      * <p>
47      * The state of this BeanBagStore must be be unchanged if any exceptions are
48      * thrown.
49      *
50      * @param num               number of bean bags added
51      * @param manufacturer      bean bag manufacturer
52      * @param name              bean bag name
53      * @param id                ID of bean bag
54      * @param year              year of manufacture
55      * @param month             month of manufacture
56      * @param information       free text detailing bean bag information
57      * @throws IllegalNumberOfBeanBagsAddedException   if the number to be added
58      *                           is less than 1
59      * @throws BeanBagMismatchException if the id already exists (as a current in
```

```
60          *                           stock bean bag, or one that has been previously
61          *                           stocked in the store, but the other stored
62          *                           elements (manufacturer, name and free text) do
63          *                           not match the pre-existing version
64          * @throws IllegalIDException   if the ID is not a positive eight character
65          *                           hexadecimal number
66          * @throws InvalidMonthException    if the month is not in the range 1 to 12
67          */
68         void addBeanBags(int num, String manufacturer, String name,
69         String id, short year, byte month, String information)
70         throws IllegalNumberOfBeanBagsAddedException, BeanBagMismatchException,
71         IllegalIDException, InvalidMonthException;
72
73         /**
74          * Method to set the price of bean bags with matching ID in stock.
75          * <p>
76          * The state of this BeanBagStore must be be unchanged if any exceptions are
77          * thrown.
78          *
79          * @param id                ID of bean bags
80          * @param priceInPence      bean bag price in pence
81          * @throws InvalidPriceException if the priceInPence < 1
82          * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
83          *                          match any bag in (or previously in) stock
84          * @throws IllegalIDException if the ID is not a positive eight character
85          *                           hexadecimal number
86          */
87         void setBeanBagPrice(String id, int priceInPence) throws
88         InvalidPriceException, BeanBagIDNotRecognisedException, IllegalIDException;
89
90         /**
91          * Method sells bean bags with the corresponding ID from the store and removes
92          * the sold bean bags from the stock.
93          * <p>
94          * The state of this BeanBagStore must be be unchanged if any exceptions are
95          * thrown.
96          *
97          * @param num           number of bean bags to be sold
98          * @param id            ID of bean bags to be sold
99          * @throws BeanBagNotInStockException   if the bean bag has previously been in
100         *                      stock, but is now out of stock
101         * @throws InsufficientStockException   if the bean bag is in stock, but not
102         *                      enough are available (i.e. in stock and not reserved)
103         *                      to meet sale demand
104         * @throws IllegalNumberOfBeanBagsSoldException if an attempt is being made to
105         *                      sell fewer than 1 bean bag
106         * @throws PriceNotSetException if the bag is in stock, and there is sufficient
107         *                      stock to meet demand, but the price has yet to be set
108         * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
109         *                          match any bag in (or previously in) stock
110         * @throws IllegalIDException   if the ID is not a positive eight character
111         *                           hexadecimal number
112         */
113        void sellBeanBags(int num, String id) throws BeanBagNotInStockException,
114        InsufficientStockException, IllegalNumberOfBeanBagsSoldException,
115        PriceNotSetException, BeanBagIDNotRecognisedException, IllegalIDException;
116
117        /**
118         * Method reserves bean bags with the corresponding ID in the store and returns
119         * the reservation number needed to later access the reservation
120         * <p>
```

```
121         * The state of this BeanBagStore must be be unchanged if any exceptions are
122         * thrown.
123         *
124         * @param num           number of bean bags to be reserved
125         * @param id            ID of bean bags to be reserved
126         * @return              unique reservation number, i.e. one not currently live
127         *                      in the system
128         * @throws BeanBagNotInStockException   if the bean bag has previously been in
129         *                      stock, but is now out of stock
130         * @throws InsufficientStockException   if the bean bag is in stock, but not
131         *                      enough are available to meet the reservation demand
132         * @throws IllegalNumberOfBeanBagsReservedException if the number of bean bags
133         *                      requested to reserve is fewer than 1
134         * @throws PriceNotSetException if the bag is in stock, and there is sufficient
135         *                      stock to meet demand, but the price has yet to be set
136         * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
137         *                          match any bag in (or previously in) stock
138         * @throws IllegalIDException   if the ID is not a positive eight character
139         *                           hexadecimal number
140         */
141        int reserveBeanBags(int num, String id) throws BeanBagNotInStockException,
142        InsufficientStockException, IllegalNumberOfBeanBagsReservedException,
143        PriceNotSetException, BeanBagIDNotRecognisedException, IllegalIDException;
144
145        /**
146         * Method removes an existing reservation from the system due to a reservation
147         * cancellation (rather than sale). The stock should therefore remain unchanged.
148         * <p>
149         * The state of this BeanBagStore must be be unchanged if any exceptions are
150         * thrown.
151         *
152         * @param reservationNumber           reservation number
153         * @throws ReservationNumberNotRecognisedException  if the reservation number
154         *                          does not match a current reservation in the system
155         */
156        void unreserveBeanBags(int reservationNumber)
157        throws ReservationNumberNotRecognisedException;
158
159        /**
160         * Method sells beanbags with the corresponding reservation number from
161         * the store and removes these sold beanbags from the stock.
162         * <p>
163         * The state of this BeanBagStore must be be unchanged if any exceptions are
164         * thrown.
165         *
166         * @param reservationNumber           unique reservation number used to find
167         *                                    beanbag(s) to be sold
168         * @throws ReservationNumberNotRecognisedException  if the reservation number
169         *                          does not match a current reservation in the system
170         */
171        void sellBeanBags(int reservationNumber)
172        throws ReservationNumberNotRecognisedException;
173
174        /**
175         * Access method for the number of BeanBags stocked by this BeanBagStore
176         * (total of reserved and unreserved stock).
177         *
178         * @return                  number of bean bags in this store
179         */
180        int beanBagsInStock();
181
```

```
182     /**
183      * Access method for the number of reserved bean bags stocked by this
184      * BeanBagStore.
185      *
186      * @return                  number of reserved bean bags in this store
187      */
188     int reservedBeanBagsInStock();
189 
190     /**
191      * Method returns number of bean bags with matching ID in stock (total
192      * researved and unreserved).
193      * <p>
194      * The state of this BeanBagStore must be be unchanged if any exceptions are
195      * thrown.
196      *
197      * @param id            ID of bean bags
198      * @return              number of bean bags matching ID in stock
199      * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
200      *                          match any bag in (or previously in) stock
201      * @throws IllegalIDException   if the ID is not a positive eight character
202      *                           hexadecimal number
203      */
204     int beanBagsInStock(String id) throws BeanBagIDNotRecognisedException,
205     IllegalIDException;
206 
207     /**
208      * Method saves this BeanBagStore's contents into a serialised file,
209      * with the filename given in the argument.
210      *
211      * @param filename      location of the file to be saved
212      * @throws IOException  if there is a problem experienced when trying to save
213      *                      the store contents to the file
214      */
215     void saveStoreContents(String filename) throws IOException;
216 
217     /**
218      * Method should load and replace this BeanBagStore's contents with the
219      * serialised contents stored in the file given in the argument.
220      * <p>
221      * The state of this BeanBagStore must be be unchanged if any exceptions are
222      * thrown.
223      *
224      * @param filename      location of the file to be loaded
225      * @throws IOException  if there is a problem experienced when trying to load
226      *                      the store contents from the file
227      * @throws ClassNotFoundException   if required class files cannot be found when
228      *                      loading
229      */
230     void loadStoreContents(String filename) throws IOException,
231     ClassNotFoundException;
232 
233     /**
234      * Access method for the number of different bean bags currently stocked by this
235      * BeanBagStore.
236      *
237      * @return                  number of different specific bean bags currently in
238      *                          this store (i.e. how many different IDs represented
239      *                          by bean bags currently in stock, including reserved)
240      */
241     int getNumberOfDifferentBeanBagsInStock();
242 
```

```
243     /**
244      * Method to return number of bean bags sold by this BeanBagStore.
245      *
246      * @return                  number of bean bags sold by the store
247      */
248     int getNumberOfSoldBeanBags();
249 
250     /**
251      * Method to return number of bean bags sold by this BeanBagStore with
252      * matching ID.
253      * <p>
254      * The state of this BeanBagStore must be be unchanged if any exceptions are
255      * thrown.
256      *
257      * @param id                ID of bean bags
258      * @return                  number bean bags sold by the store with matching ID
259      * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
260      *                          match any bag in (or previously in) stock
261      * @throws IllegalIDException   if the ID is not a positive eight character
262      *                          hexadecimal number
263      */
264     int getNumberOfSoldBeanBags(String id) throws
265     BeanBagIDNotRecognisedException, IllegalIDException;
266 
267     /**
268      * Method to return total price of bean bags sold by this BeanBagStore
269      * (in pence), i.e. income that has been generated by these sales).
270      *
271      * @return                  total cost of bean bags sold (in pence)
272      */
273     int getTotalPriceOfSoldBeanBags();
274 
275     /**
276      * Method to return total price of bean bags sold by this BeanBagStore
277      * (in pence) with  matching ID (i.e. income that has been generated
278      * by these sales).
279      * <p>
280      * The state of this BeanBagStore must be be unchanged if any exceptions are
281      * thrown.
282      *
283      * @param id                ID of bean bags
284      * @return                  total cost of bean bags sold (in pence) with
285      *                          matching ID
286      * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
287      *                          match any bag in (or previously in) stock
288      * @throws IllegalIDException   if the ID is not a positive eight character
289      *                          hexadecimal number
290      */
291     int getTotalPriceOfSoldBeanBags(String id) throws
292     BeanBagIDNotRecognisedException, IllegalIDException;
293 
294     /**
295      * Method to return the total price of reserved bean bags in this BeanBagStore
296      * (i.e. income that would be generated if all the reserved stock is sold
297      * to those holding the reservations).
298      *
299      * @return                  total price of reserved bean bags
300      */
301     int getTotalPriceOfReservedBeanBags();
302 
303     /**
```

```
304         * Method to return the free text details of a bean bag in stock. If there
305         * are no String details for a bean bag, there will be an empty String
306         * instance returned.
307         * <p>
308         * The state of this BeanBagStore must be be unchanged if any exceptions are
309         * thrown.
310         *
311         * @param id                ID of bean bag
312         * @return                  any free text details relating to the bean bag
313         * @throws BeanBagIDNotRecognisedException  if the ID is legal, but does not
314         *                          match any bag in (or previously in) stock
315         * @throws IllegalIDException   if the ID is not a positive eight character
316         *                           hexadecimal number
317         */
318        String getBeanBagDetails(String id) throws
319        BeanBagIDNotRecognisedException, IllegalIDException;
320
321        /**
322         * Method empties this BeanBagStore of its contents and resets
323         * all internal counters.
324         */
325        void empty();
326
327        /**
328         * Method resets the tracking of number and costs of all bean bags sold.
329         * The stock levels of this BeanBagStore and reservations should
330         * be unaffected.
331         */
332        void resetSaleAndCostTracking();
333
334        /**
335         * Method replaces the ID of current stock matching the first argument with the
336         * ID held in the second argument. To be used if there was e.g. a data entry
337         * error on the ID initially entered. After the method has completed all stock
338         * which had the old ID should now have the replacement ID (including
339         * reservations), and all trace of the old ID should be purged from the system
340         * (e.g. tracking of previous sales that had the old ID should reflect the
341         * replacement ID).
342         * <p>
343         * If the replacement ID already exists in the system, this method will return
344         * an {@link IllegalIDException}.
345         *
346         * @param oldId             old ID of bean bags
347         * @param replacementId     replacement ID of bean bags
348         * @throws BeanBagIDNotRecognisedException  if the oldId does not match any
349         *                          bag in (or previously in) stock
350         * @throws IllegalIDException   if either argument is not a positive eight
351         *                          character hexadecimal number, or if the
352         *                          replacementID is already in use in the store as
353         *                          an ID
354         */
355        void replace(String oldId, String replacementId)
356        throws BeanBagIDNotRecognisedException, IllegalIDException;
357    }
```