Bean Bag Management

Pair Member 1: 680033128

Pair Member 2: 690065435

14/02/2020 (Friday - 2h 30m: 2:45pm to 5:15pm)

- We discussed timings for future meetings and began evaluating the required functionality of the program as well as consideration for the best ways to implement said functionality.
- We created a shared work file so we can be viewing the log and code on our computers whenever we meet up and easily make changes to the code whilst writing our report.
- We planned what kinds of data structures would be used for different parts of the program.

15/02/2020 (Saturday - 1h 45m: 12:30pm to 2:15pm)

- We began by evaluating the exception classes as well as the basic classes sent with the project briefing.
- We considered storing data within a dictionary type structure, where the key is the 8-character hex identifier and the value contains a list of information pertaining to different configurations of beanbags.
    - For stock indexing to work with this data structure we would need to include a field for quantity such that any bean bags which are interchangeable with identical details are counted correctly as these could have identical hex identifiers or different identifiers if the differed in colour.
    - We needed to assess the ObjectArrayList to see how best to implement the desired data structure without the need for additional modules.

16/02/2020 (Sunday - 1h 10m: 1:30pm to 2:40pm)

- We drafted an example data structure listing all the information required for an individual bean bag.
    - This was done using the examples from the Class, Strings, Array Workshop.
- Person A suggested testing manipulation of data by starting off with a simple function to edit the quantity of an item using its Hex identifier.
- Person B attempted to implement this class and create a driver class to run all the methods contained within the package.
- The Week 5 Quiz included an example of reading from text files.
    - We considered later implementing this feature in our program so that the state of the stock manager is saved upon closing, but first needed to ensure all our functionality worked.

17/02/2020 (Monday - 50m: 9:35am to 10:25am)

- We considered the different data structures that could be implemented.
- We settled on an ObjectArrayList which operates with a dictionary structure using keys and values.

    - Instances of an object can have identical names as they are stored in different memory locations as such it would be possible to count the quantity of beanbags in stock by counting the number of objects created.
    - When an item with an identical hex ID is sold you can simply delete one of the instances of the matching objects.

18/02/2020 (Tuesday - 4h: 10am to 2pm)

- Person A completed a rough draft of the UML diagram and set up a GitHub repository which would allow us to keep track of code revisions and later add the package upon completion as one our projects on LinkedIn.
- Person B wanted to begin the project by implementing a simple class responsible for sending data to the ObjectArrayList as mentioned previously, where the object names would be the HexID of the beanbags.
    - Contained within each object would be all the information relating to a beanbag.
    - We started with a single bean bag and wanted to attempt to change the data held by the object. The simplest test case we devised would be to change.
- Person A suggested a more detailed revision of the availability of both partners, so we decided to add our availability to each other's calendars.
- Person B reconfigured the directories to match those suggested in the 'Jar File Creation Walkthrough' which was recently released.
- Person A found a nice website called draw.io which allowed for seamless completion of our UML file which had previously been drawn out on paper.
    - This was based on the diagram given in the Class, String, Array workshop.

20/02/2020 (Thursday - 1h 40m: 12:40pm to 2:20pm)

- In our OOP lecture today, we learnt about inheritance and the keywords 'super' and 'extends' to refer to parent and child classes.
- Person A considered that we could have the Reservations class inherit the constructor method from Stock classes.
    - To do this this subclass needs to extend the superclass as such we could include additional parameters for 'reserved (Boolean)', 'reference (String)', 'customer (String)'.
    - The access modifier for an overriding method can allow for more but not less access than the overridden method so this approach should be suitable for our application.
- We started to break down the briefing that we had been given to shortlist the key information provided and bullet point the key features that we needed to implement in our system. The package is required to:
    - Keep track of several attributes:
        - An 8-digit hex id (String)
        - Manufacturer (String),
        - Bean bag name (String),
        - priceInPence (int),
        - Year of manufacture (short),

        - Month of manufacture (byte),
        - Number of a type of bean bag (int) both remaining and later sold,
        - Optional free text component containing any additional information (String).
    - Ability to reserve items currently in stock
    - Reserved items must only be sold to customers with a matching reservation number.
    - Customers can cancel reservations and add items back to general item stock.
    - We would need to keep track of an attribute relating to reservationNumber (int)
    - Ability to change price of items
    - Must ensure that and items reserved are sold at the lower price.
- Hence, both agreed it would be appropriate to split the problem up with separate classes for dealing with the reservation system, tracking sales totals and managing the store's stock.
- Person A said that from the information provided it looks like it would be possible to create multiple different types objects containing varying attributes for example one type of objects for stock management and separate type of object for reservations, whilst still being able to store all these objects within the same ObjectArrayList class provided.
- Person B liked the sound of this idea but said it would be wise to start by declaring the required attributes as null variable within their appropriate classes.
    - This would allow for compartmentalisation of different kinds of functions within the overall package where similar functions relating to the same feature would be grouped together.

23/02/2020 (Sunday - 5h 30m: 12pm to 5:30pm)

- Person B said we would need two constructor methods one which included a reference to the optional description variable containing the free text and a second identically named method without this variable.
    - When running the package, the class selects the appropriate method based on the number of variables included when calling the method.
- Person A remembered that no variable for quantity need be included in the constructor as when multiple identically named beanbags are in stock, they would each be referenced as separate objects in memory within the java virtual machine.
- Person B labelled methods getter and setter declared private variables alongside the public methods to call said variables.
- Person A created a new class called Stock responsible for managing any object containing information relating to the details of beanbags in stock.
    - As discussed above we intend to segregate functionality within the program such that we are implementing several smaller classes rather than one large class containing all functionality.
- We discussed at length about the benefits of separating different parts of the package some of the benefits include upgradeability of the system, readability, error checking/bug fixes, reliability.

    - This may however come at the cost of a less efficient system in terms of the amount of storage required to store a greater number of objects in the ObjectArray.
    - This is as a result of splitting up reservations and beanbag details into two separate objects.
- Upon later review of our code, person B realised that within the reservations object we would need to store the price of the item at the time it was reserved.
    - This would allow us to later compare this price with the current price of any identical items in general stock and change the customer the lower of the two prices.
    - As such Person A changed the code to include a field for the original price.
    - Included an additional getter method in order to retrieve this new field from the object.

- The tables below detail the two different methods that could be implemented in our system. The latter is the format that we have decided to go with however as we continue to add functionality to our program, we will continually evaluate if this data structure is the most appropriate.

- For a stock object:

8-Bit Identifier (String)	A1B2C3D4
Manufacturer (String)	Argos
Name (String)	Recliner
Price (double)	29.99
Year of Manufacture (int)	2019
Month of Manufacture (int)	11
Optional Description (String)	Teal Blue

- For reservations:

8-Bit Identifier (String)	A1B2C3D4
Reserved (Boolean)	True
Reservation Reference (String)	Order-1234
Customer Name (String)	Jane Doe

- The first possibility for data handling would require two separate objects for any reservation. This would lead to better readability within the program and all the other benefits listed above.

- The second method below incorporates all data into a single object so takes up less storage, but any changes made to the code would be more difficult to make due to reduced readability.
    - Hence, bug fixes and error checking would be far more difficult, hindering development of the package classes.
- For an object combining both stock and reservations:

8-Bit Identifier (String)	A1B2C3D4
Manufacturer (String)	Argos
Name (String)	Recliner
Price (double)	29.99
Year of Manufacture (int)	2019
Month of Manufacture (int)	11
Optional Description (String)	Teal Blue
Reserved (Boolean)	True
Reservation Reference (String)	Order-1234
Customer Name (String)	Jane Doe

- Following on from this, Person B suggested that for the reservations class we could follow the same approach as the stock class, such that any reservation objects are named using the reservation number.
    - The difference would be that for reservation, we would never have any identical objects.
- Person B found an error with Person A's code; even when calling the reservation object by its reservation reference, we would still need access to the hex identifier to change the availability of reserved items.
    - We updated our Reserved class to include a field for the identifier within the object.
- Currently, the reserved class only includes a setter method for changing the Boolean value referring to whether the item is reserved.
    - If this is set to false, then the Reservation object is removed from the ObjectArrayList and available for purchase within the general store.
    - Later, we may choose to include setter methods for changing the beanbag that a customer has decided to reserve or the customer name relating to that reservation.
    - However, to keep the initial program simple currently, these fields are only accessible through getter method.

        - Therefore, in order to change any details relating to a reservation, the store would first need to cancel the original reservation before creating a new reservation entirely.
        - This currently isn't a particularly major issue but is something we could revisit at a later stage depending on the functionality required.
        - In some instances this implementation may be better than the alternative, as it reduces the possibility of accidental deletion or editing of details relating to items that a customer has already decided to reserve.

24/02/2020 (Monday - 1h 40m: 12:40pm to 2:20pm)

- Started coding for our project.
- Person A took on the role of the driver whilst Person B was the observer in this instance.
- Began by creating a new Store class and copying over all the constructor methods listed in the BeanBagStore interface class.
    - Person B suggested that we work by first creating a template of all the getter and setter methods described in the briefing and highlighting any key areas where we could potentially reuse parts of the code by overriding certain methods.
    - After we had completed the outline of our project and what it was going to look like, we began filling in the simpler methods responsible for creating bean bag objects.
    - We also coded getters and setters for accessing and changing the quantity of a bean bag stored in the object.
- Person B had realised that the data types of each parameter that we needed to state were given in the BeanBagStore interface.
    - Hence, Person A changed some of the variable types such as month from int to byte and year from int to short.
    - We also took this opportunity to create a separate class containing our main method used to control the rest of the package.

26/02/2020 (Wednesday - 1h 35m: 12:40pm to 2:15pm)

- This session Person B was the driver and Person A the observer.
- Person A realised that later on in development, when we get to creating the getter method for year of manufacture, it would be possible to use a switch statement to convert the byte data type to a String type, containing the full name of the month in which the beanbag was manufactured.
    - This got us thinking about other useful Java features that we had learned in lectures that could be used in our program.
    - Person A determined that we would need some way to enforce the 8-digit hexadecimal naming scheme described in the brief such that only String values conforming to the standard were allowed.

    - Person B suggested it may be possible to use the regex search pattern for input validation. When giving each beanbag a unique identifier, we would revisit these discussions in later stages of development.

27/02/2020 (Thursday - 1h 20m: 12:40pm to 2pm)

- Further to the discussion above regarding the reuse of code, Person A noted that it may be possible to override the setter method for addBeanBags for reuse when selling bean bags.
- When we get to this part of the project, we will evaluate whether this is the case or if there is a better solution.
- Person B realised that we had missed declaration of two parameters (namely reservationNumber and filename) used to identify reserved sales and store the contents of BeanBagStore respectively.

29/02/2020 (Saturday - 2h: 1pm to 3pm)

- With the separate objects for stock and reservations, Person A remembered that you would need a way to store the details relating to reserved items.
- This could be done in one of two ways.
    - The first would be to include an optional reserved field in the stock object which would require yet another constructor method with space for this field.
        - This particular method would only be invoked whenever any reservations are made.
        - This reserved variable would only need to be set when the Boolean value is true.
        - By including this addition method and relying on the same stock objects as before we could use the same setter methods from the Stock class to make changes to reservations instead of having to include duplicate methods within the Reservations class.
    - The alternative is to copy over all the details relating to the item for use in the reserved object.
        - This would reduce the need for an additional constructor method and would not affect storage required, as you remove the stock object from the ObjectArrayList and replace it with a reserved object with identical details.
- Person B noted that we had not even begun to incorporate any functionality into our program and we could possibly use four different implementations for our data structure.
    - As such we decided to keep things as simple as possible for the time being whilst still allowing ourselves to make any necessary changes to our design in the future, should we decide that our chosen structure was no longer suitable.
    - Our UML diagram was significantly useful to us in allowing us to plan ahead, making sure we would not later have to compromise on functionality, whilst

ensuring we did not encounter the headache of needing to rewrite large portions of code at later stages of development.

07/03/2020 (Saturday - 4h: 6:30pm to 12:30am)

- Both agreed to move the log document from Microsoft Word on OneDrive to Google Docs on Drive to make collaboration easier.
- Person B reformatted the existing log for easier readability, making it easier to track previous progress. Changes included bullet-pointing of progress made, and simpler phrasing.

09/03/2020 (Monday - 5h: 11:30am to 4:30pm)

- Went over existing code to confirm that both people had a good understanding about what was going on.
- We created a driver file, StoreDriverApp, to test the front end of the application.
- Person A worked on implementing code for the mismatch exception in the Mismatch file, with person B acting as the observer to check that the code made sense syntactically and logically.
    - We swapped after the working code for Mismatch was completed and tested to work; person B commented on the code, whilst person A observed to ensure that both had a good understanding of the code.
- Person B started adding comments to CheckID, with person A observing to ensure that the comments made sense and were correct.
    - We researched into Java commenting conventions so that we knew we were using commonly accepted standards, which would make it easier for developers to continue working on the code in the future.
- Person A started adding code for the addBeanBags and setBeanBagPrice methods in Store, with person B acting as the observer.
    - We swapped after this was finished; person B added comments to Store, with person A acting the observer.
- Person B added comments to BeanBag, with person A observing to clarify any confusion about parts of the code.
- Person A completed the code for the sellBeanBags method in Store, with person B observing.
    - Once this was completed, we swapped roles to test it. Person B added code to the StoreDriverApp, whilst person A observed to ensure his testing methodology was correct.

10/03/2020 (Tuesday - 4h: 12:30pm to 4:30pm)

- Person A implemented the reserveBeanBags and unreserveBeanBags methods in Store, with person B being the observer.
    - We decided to create a new class, Reservation, to handle the reservations of items using a new array.
    - We swapped roles during the commenting phase; Person B did the commenting whilst person A observed.

- Person B moved all of the exception handling to the Checks class (previously named CheckID), whereas it was previously handled mostly in CheckID, but not entirely. Person A observed to ensure this was done correctly.
    - Mismatch was removed, as all of the code was moved to Checks.
- Person B started implementing the empty method in Store, with person A acting as the observer.
    - We were initially unsuccessful after starting off well, so we swapped roles and successfully implemented it. For this, person A was the driver, and person B was the observer.
- Whilst we were optimising some code, person A suggested changing the bean bag ID checks to be non-case-sensitive, as this would improve the user experience. Person B implemented this after researching how this could be done.
- Person A implemented the replace method from Store, with person B acting as the observer.
    - We swapped roles for the commenting of this. Person B acted as the driver, and person A acted as the observer.
- Person B realised that the Stock class had not been entirely commented on, so they worked on that, with person A observing to ensure comments made sense.
- Person B implemented sellBeanBags for a reservation (selling bean bags which had been previously reserved by the customer), with person A acting as the observer.
    - Person A was the driver for commenting, and person B was the observer
- Person B implemented the getNumberOfSoldBeanBags and getNumberOfSoldBeanBags(id) methods for Store, with person A observing.
    - We were able to reuse most of the code from beanBagsInStock(id), changing some of the parts to be relevant for this use case.
- Person A implemented getTotalPriceOfReservedBeanBags in Store, with person B observing.
    - We swapped roles for commenting this method; person B was the driver, whilst person A was the observer.
- Person B commented on the existing methods and some additional parts within methods in Store, as well as reordering some code for readability, with person A observing to ensure that a mutual understanding was retained.

11/03/2020 (Wednesday - 4h 30m: 1:00pm to 5:30pm)

- Person A updated the reserveBeanBags method and replace method in Store to be fully functional.
    - Person B swapped with person A for the commenting of the new code added to the reserveBeanBags method and replace method.
- Person A implemented the getNumberofDifferentBeanBagsInStock method in Store, with person B observing.
    - Person B swapped with person A for the commenting of this method to ensure a mutual understanding.
- Person B added the getBeanBagDetails method in Store, with person A observing.
    - We swapped roles for the commenting of this code.

- We improved the addBeanBags method by calling a new method, getExistingPrice, within it. Person B acted as the driver, whilst person A was the observer.
    - This ensures consistency in the stock list, as bean bags with the same ID will have the same price when they're added.
    - It complements the method setBeanBagPrice, as it means bean bags added in the future will also have this price set.
- Person A added additional testing methods, toString, array, and printArray, in Store. Person B acted as the observer.
    - These were made to help us test the application.
    - We swapped roles for commenting; person B acted as the driver, whilst person A was the observer.

12/03/2020 (Thursday - 4h: 10:30pm to 2:30pm)

- Person A implemented the saveStoreContents and loadStoreContents methods in Store, with person B observing.
    - We swapped roles for commenting this method to ensure both parties understood the code fully.
- We removed duplicated code by moving it to a new method, countBeanBags. This counts the number of matching IDs for any objects in the given list(s).
    - This new method was called in beanBagsInStock and getNumberOfSoldBeanBags, where an ID is passed as an argument.
        - It replaced the blocks of code which previously made up these methods.
- We decided to remove the toString method, as it wasn't being used in any parts of our program.
- Person B updated the setBeanBagPrice method to sell reserved bean bags to the customer for the lowest price since they reserved it (in the case that they reserved it, and it changed prices since then to be lower). Person A was the observer for this.
    - We swapped after the code was finished, so person A was the driver for commenting, whilst person B observed.
- We tested our app by creating a test driver file, Driver.
    - To ensure that our program was appropriately tested, we isolated the cases one by one (made easier with our alphabetical ordering of methods), and tested different combinations.

13/03/2020 (Friday - 4h 30m: 12am to 4:30am)

- Person B created doc comments for the classes and methods within those classes, with person A acting as the observer to ensure that mistakes weren't being made.
    - We used @inheritDoc for the overridden methods, as the doc comments were already provided for those.
    - We created doc comments for new methods which we had created.
- Person A added assertions to help with final testing before submission, with person B observing to ensure that the assertions made sense.
- We created the jar file for submission.

```java
package beanbags;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;

/**
 * Store class which implements BeanBagStore. Represents the back end of a stock
 * management system help a user manage reservations, sales and restocking of
 * items more efficiently.
 *
 * @author 680033128
 * @author 690065435
 * @version 1.4
 *
 *
 */

public class Store implements BeanBagStore {
    // Initialises the three lists for storing available, reserved, and sold bean
    // bags.
    private ObjectArrayList available = new ObjectArrayList();
    private ObjectArrayList reserved = new ObjectArrayList();
    private ObjectArrayList sold = new ObjectArrayList();

    /**
     * {@inheritDoc}
     */
    @Override
    public void addBeanBags(int num, String manufacturer, String name, String id, short year, byte
month)
            throws IllegalNumberOfBeanBagsAddedException, BeanBagMismatchException,
IllegalIDException,
            InvalidMonthException {
        addBeanBags(num, manufacturer, name, id, year, month, "");
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void addBeanBags(int num, String manufacturer, String name, String id, short year, byte
month,
            String information) throws IllegalNumberOfBeanBagsAddedException,
BeanBagMismatchException,
            IllegalIDException, InvalidMonthException {
        // Throws an exception if the user tries to add a negative number of bean bags.
        if (num <= 0) {
            throw new IllegalNumberOfBeanBagsAddedException(
                    "The number of bean bags '" + num + "' added must be positive.");
        }

        // Throws an exception if the user tries to add a month of manufacture which
        // doesn't exist.
        if (month < 0 | month > 12) {
```

```
            throw new InvalidMonthException("The month of manufacturer '" + month + "' must be between 1 and 12.");
        }

        // Checks if IDs and attributes both match for added bean bags.
        Checks.validId(id);

        // Checks the available stock for any matching bean bags and gets the price of
        // item to use for the new one.
        int existingPrice = getExistingPrice(id);
        for (int i = 0; i < num; i++) {
            BeanBag newBeanBag = new BeanBag(manufacturer, name, id, year, month, information, existingPrice);
            Checks.existingMismatch(newBeanBag, available, reserved, sold);
            available.add(newBeanBag);
        }

        // Throws an AssertionError if the information value is null.
        assert (information != null) : "The bean bag incorrectly has information set to null.";
    }

    /**
     * Method for testing which prints the bean bags in a given list.
     *
     * @param type String which categorises the object as available, reserved, or
     *             sold.
     */
    public void array(String type) {
        // Prints a list of bean bags depending on the category provided.
        switch (type.toLowerCase()) {
            case "available":
            case "a":
                arrayPrint(available, "available");
                break;
            case "reserved":
            case "r":
                arrayPrint(reserved, "reserved");
                break;
            case "sold":
            case "s":
                arrayPrint(sold, "sold");
                break;
        }
    }

    /**
     * Method for testing which iterates through the given list
     * (available/reserved/sold) and prints the bean bags in that list.
     *
     * @param obj  List of bean bags from the category specified.
     * @param type Category of object (available, reserved, or sold).
     */
    public void arrayPrint(ObjectArrayList obj, String type) {
        BeanBag item;
        Reservation held;

        // If the object size is empty, let the user know.
        if (obj.size() == 0) {
```

```
            System.out.println("The '" + type + "' object is empty.");
        } else {
            System.out.println("The '" + type + "' object contains '" + obj.size() + "' items:");
            // Prints a list of the beans bags which are available or sold.
            if (type.equals("available") || type.equals("sold")) {
                for (int j = 0; j < obj.size(); j++) {
                    item = (BeanBag) obj.get(j);
                    System.out.println("[id=" + item.getIdentifier() + ",name=" + item.getName() + ",manufacturer="
                            + item.getManufacturer() + ",year=" + item.getYear() + ",month=" + item.getMonth()
                            + ",information=" + item.getInformation() + ",priceInPence=" + item.getPriceInPence()
                            + "]");
                }
                // Prints a list of the bean bags which are reserved.
            } else if (type.equals("reserved")) {
                for (int j = 0; j < obj.size(); j++) {
                    held = (Reservation) obj.get(j);
                    item = held.getAttributes();
                    System.out.println("[id=" + item.getIdentifier() + ",name=" + item.getName() + ",manufacturer="
                            + item.getManufacturer() + ",year=" + item.getYear() + ",month=" + item.getMonth()
                            + ",information=" + item.getInformation() + ",priceInPence=" + item.getPriceInPence()
                            + ",reservationNumber=" + held.getReservation() + "]");
                }
            }
        }
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int beanBagsInStock() {
        return available.size() + reserved.size();
    }

    // Returns the number of a type of bean bag in stock.
    public int beanBagsInStock(String id) throws BeanBagIDNotRecognisedException, IllegalIDException {
        // Returns the number of bean bags with a matching ID which are either reserved
        // or available.
        return countBeanBags(new ObjectArrayList[] { available, reserved }, id);
    }

    /**
     * Method counts the number of matching IDs for any objects which are in the
     * given list(s).
     *
     * @param objects Array containing the individual objs (the lists of categorised
     *                bean bags).
     * @param id      ID of bean bags.
     * @return Number of bean bags with the matching ID.
     * @throws BeanBagIDNotRecognisedException If the ID is legal, but does not
     *                                         match any bag in (or previously in)
```

```
     *                                      stock.
     * @throws IllegalIDException          If the ID is not a positive eight
     *                                      character hexadecimal number.
     */
    public int countBeanBags(ObjectArrayList[] objects, String id)
            throws BeanBagIDNotRecognisedException, IllegalIDException {
        // Starts the count at 0.
        int count = 0;
        boolean recognised = false;

        Checks.validId(id);

        // Accesses each element of array.
        for (ObjectArrayList object : objects) {
            // Iterates over the stock object and increments the count for each bean
            // bag with a matching ID.
            for (int j = 0; j < object.size(); j++) {
                BeanBag item;
                if (object == reserved) {
                    Reservation held = (Reservation) object.get(j);
                    // Throws an AssertionError if the reserved bean bag fails to return any
                    // attributes.
                    assert (held != null) : "The reserved bean bag incorrectly has information set to
null.";
                    item = held.getAttributes();
                } else {
                    item = (BeanBag) object.get(j);
                }
                if (item.getIdentifier().equalsIgnoreCase(id)) {
                    count += 1;
                    recognised = true;
                }
            }
        }
        // Throws an exception if no bean bag with the matching ID was found.
        if (!recognised) {
            throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
found.");
        }

        return count;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public void empty() {
        // Sets lists to be null for garbage collection before creating a new empty
        // list, which enables more efficient use of memory.
        available = null;
        available = new ObjectArrayList();
        reserved = null;
        reserved = new ObjectArrayList();
        resetSaleAndCostTracking();

        // Throws an AssertionError if the bean bag list wasn't emptied correctly.
        assert (available.size() == 0) : "Available list was not emptied correctly.";
```

```
217.         // Throws an AssertionError if the reserved list wasn't emptied correctly.
218.         assert (reserved.size() == 0) : "Reservation list was not emptied correctly.";
219.     }
220.
221.     /**
222.      * {@inheritDoc}
223.      */
224.     @Override
225.     public String getBeanBagDetails(String id) throws BeanBagIDNotRecognisedException,
IllegalIDException {
226.         BeanBag item = null;
227.         boolean recognised = false;
228.
229.         Checks.validId(id);
230.
231.         // Iterates over the available stock list and increments the count for each bean
232.         // bag with a matching ID.
233.         for (int j = 0; j < available.size(); j++) {
234.             item = (BeanBag) available.get(j);
235.             // Breaks out of the loop if a matching bean bag was found (other matching bean
236.             // bags should have the same details).
237.             if (item.getIdentifier().equalsIgnoreCase(id)) {
238.                 recognised = true;
239.                 break;
240.             }
241.         }
242.
243.         // Prevents iteration through the next for loop if bean bag details were found
244.         // from stock list.
245.         if (recognised) {
246.             return item.getInformation();
247.
248.         }
249.         // Iterates over the reserved stock list and increments the count for each bean
250.         // bag with a matching ID.
251.         for (int j = 0; j < reserved.size(); j++) {
252.             item = (BeanBag) reserved.get(j);
253.             // Breaks out of the loop if a matching bean bag was found (other matching bean
254.             // bags should have the same details).
255.             if (item.getIdentifier().equalsIgnoreCase(id)) {
256.                 recognised = true;
257.                 break;
258.             }
259.         }
260.         // Returns the free text component of the first matching bean bag in the list.
261.         if (!recognised) {
262.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
found.");
263.         }
264.         return item.getInformation();
265.     }
266.
267.     /**
268.      * Method returns the price of a given item based on its ID.
269.      *
270.      * @param id ID of bean bags.
271.      * @return Price of the given bean bag.
272.      */
```

```
273.     public int getExistingPrice(String id) {
274.
275.         // Iterates over the available stock list and get the price from the first
276.         // instance of a matching bean bag ID.
277.         for (int j = 0; j < available.size(); j++) {
278.             BeanBag item = (BeanBag) available.get(j);
279.             if (item.getIdentifier().equalsIgnoreCase(id)) {
280.                 return item.getPriceInPence();
281.             }
282.         }
283.
284.         // Returns 0 as a default price to represent the price not being set.
285.         return 0;
286.     }
287.
288.     /**
289.      * {@inheritDoc}
290.      */
291.     @Override
292.     public int getNumberOfDifferentBeanBagsInStock() {
293.         ObjectArrayList uniqueId = new ObjectArrayList();
294.
295.         // Replaces the object IDs in each of these lists.
296.         ObjectArrayList[] objects = { available, reserved };
297.         BeanBag item;
298.         ObjectArrayList obj;
299.
300.         // Iterates over the lists one by one.
301.         for (ObjectArrayList object : objects) {
302.             // Accesses each element of array.
303.             obj = object;
304.             // Updates the IDs in the available stock, reserved and sold list.
305.             for (int j = 0; j < obj.size(); j++) {
306.                 if (obj == reserved) {
307.                     Reservation held = (Reservation) obj.get(j);
308.                     // Throws an AssertionError if the reserved bean bag fails to return any
309.                     // attributes.
310.                     assert (held != null) : "The reserved bean bag incorrectly has information set to
    null.";
311.                     item = held.getAttributes();
312.                 } else {
313.                     item = (BeanBag) obj.get(j);
314.                 }
315.
316.                 String beanBagID = item.getIdentifier();
317.                 boolean inList = false;
318.
319.                 // Iterates over the list of unique IDs and checks against duplicates.
320.                 for (int i = 0; i < uniqueId.size(); i++) {
321.                     String id = (String) uniqueId.get(i);
322.                     if (id.equalsIgnoreCase(beanBagID)) {
323.                         inList = true;
324.                         break;
325.                     }
326.                 }
327.
328.                 // Adds bean bag ID to list of unique IDs if no duplicates found.
329.                 if (!inList) {
```

```
                    uniqueId.add(beanBagID);
                }
            }
        }

        return uniqueId.size();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int getNumberOfSoldBeanBags() {
        return sold.size();
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int getNumberOfSoldBeanBags(String id) throws BeanBagIDNotRecognisedException,
IllegalIDException {
        // Returns the number of bean bags with a matching ID which have been sold.
        return countBeanBags(new ObjectArrayList[] { sold }, id);
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int getTotalPriceOfReservedBeanBags() {
        int count = 0;

        // Iterates through the reserved list and adds each type of
        for (int j = 0; j < reserved.size(); j++) {
            Reservation held = (Reservation) reserved.get(j);
            // Throws an AssertionError if the reserved bean bag fails to return any
            // attributes.
            assert (held != null) : "The reserved bean bag incorrectly has information set to null.";
            count += held.getAttributes().getPriceInPence();
        }

        return count;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public int getTotalPriceOfSoldBeanBags() {
        int count = 0;

        // Iterates over the sold bean bags list to sum the prices.
        for (int j = 0; j < sold.size(); j++) {
            BeanBag item = (BeanBag) sold.get(j);
            count += item.getPriceInPence();
        }

```

```
387.         return count;
388.     }
389.
390.     /**
391.      * {@inheritDoc}
392.      */
393.     @Override
394.     public int getTotalPriceOfSoldBeanBags(String id) throws BeanBagIDNotRecognisedException,
IllegalIDException {
395.         boolean recognised = false;
396.         int count = 0;
397.
398.         Checks.validId(id);
399.
400.         // Iterates over the sold list and counts bean bags which have a matching ID.
401.         for (int j = 0; j < sold.size(); j++) {
402.             BeanBag item = (BeanBag) sold.get(j);
403.             if (item.getIdentifier().equalsIgnoreCase(id)) {
404.                 count += item.getPriceInPence();
405.                 recognised = true;
406.             }
407.         }
408.
409.         // Throws an exception for unrecognised bean bags.
410.         if (!recognised) {
411.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
found.");
412.         }
413.
414.         return count;
415.     }
416.
417.     /**
418.      * {@inheritDoc}
419.      */
420.     @Override
421.     public void loadStoreContents(String filename) throws IOException, ClassNotFoundException {
422.         try (ObjectInputStream in = new ObjectInputStream(new FileInputStream(filename))) {
423.             // Accesses each element of array.
424.             available = (ObjectArrayList) in.readObject();
425.             reserved = (ObjectArrayList) in.readObject();
426.             sold = (ObjectArrayList) in.readObject();
427.         }
428.     }
429.
430.     /**
431.      * Method manages the selling and cancellation of reservations in the reserved
432.      * bean bag stock.
433.      *
434.      * @param reservationNumber Identifier for each reservation in the list of
435.      *                          reserved bean bags.
436.      * @return Attributes of a given reserved bean bag.
437.      * @throws ReservationNumberNotRecognisedException If the reservation number
438.      *                                                 does not match a current
439.      *                                                 reservation in the system.
440.      */
441.     public Reservation manageReservations(int reservationNumber) throws
ReservationNumberNotRecognisedException {
```

```
442.         boolean recognised = false;
443.         Reservation held = null;
444.
445.         // Iterates over reservations stock to find matching reservation number.
446.         for (int j = 0; j < reserved.size(); j++) {
447.             held = (Reservation) reserved.get(j);
448.             // Throws an AssertionError if the reserved bean bag fails to return any
449.             // attributes.
450.             assert (held != null) : "The reserved bean bag incorrectly has information set to null.";
451.             if (held.getReservation() == reservationNumber) {
452.                 recognised = true;
453.                 break;
454.             }
455.         }
456.
457.         // Throws an exception for unrecognised bean bags.
458.         if (!recognised) {
459.             throw new ReservationNumberNotRecognisedException(
460.                     "This reservation number '" + reservationNumber + "' is not recognised.");
461.         }
462.
463.         return held;
464.     }
465.
466.     /**
467.      * {@inheritDoc}
468.      */
469.     @Override
470.     public void replace(String oldId, String replacementId) throws BeanBagIDNotRecognisedException,
    IllegalIDException {
471.         boolean recognised = false;
472.
473.         // Checks that the old ID exists.
474.         Checks.validId(oldId);
475.         // Checks that the new ID is of a correct format.
476.         Checks.validId(replacementId);
477.
478.         // Replaces the object IDs in each of these lists.
479.         ObjectArrayList[] objects = { available, reserved, sold };
480.         BeanBag item;
481.         int i;
482.         ObjectArrayList obj;
483.
484.         // Iterates over the lists one by one.
485.         for (i = 0; i < objects.length; i++) {
486.             // Accesses each element of array.
487.             obj = objects[i];
488.
489.             // Updates the IDs in the available stock, reserved and sold list.
490.             for (int j = 0; j < obj.size(); j++) {
491.
492.                 if (obj == reserved) {
493.                     Reservation held = (Reservation) obj.get(j);
494.                     // Throws an AssertionError if the reserved bean bag fails to return any
495.                     // attributes.
496.                     assert (held != null) : "The reserved bean bag incorrectly has information set to
    null.";
497.                     item = held.getAttributes();
```

```
498.                 } else {
499.                     item = (BeanBag) obj.get(j);
500.                 }
501.                 if (item.getIdentifier().equalsIgnoreCase(oldId)) {
502.                     item.setIdentifier(replacementId);
503.                     recognised = true;
504.                 }
505.             }
506.         }
507.         // Throws an exception if the old ID doesn't match any ID in stock or previously
508.         // in stock.
509.         if (!recognised) {
510.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + oldId + "' could not be
found.");
511.         }
512.     }
513.
514.     /**
515.      * {@inheritDoc}
516.      */
517.     @Override
518.     public int reserveBeanBags(int num, String id)
519.             throws BeanBagNotInStockException, InsufficientStockException,
IllegalNumberOfBeanBagsReservedException,
520.             PriceNotSetException, BeanBagIDNotRecognisedException, IllegalIDException {
521.         boolean recognised = false;
522.         int reservationNumber = 0;
523.
524.         // Checks if there are enough bean bags with the given ID in stock to reserve.
525.         Checks.validReservation(num, id, countBeanBags(new ObjectArrayList[] { available }, id));
526.
527.         // For the number of bean bean bags the customer wishes to reserve, iterates
528.         // over the list of available bean bags and checks for matching bean
529.         // bags.
530.         for (int i = 0; i < num; i++) {
531.             for (int j = 0; j < available.size(); j++) {
532.                 BeanBag item = (BeanBag) available.get(j);
533.                 if (item.getIdentifier().equalsIgnoreCase(id)) {
534.                     recognised = true;
535.                     // Throws an exception if no price set for this item.
536.                     if (item.getPriceInPence() == 0) {
537.                         throw new PriceNotSetException("No price has been set for this item '" + id +
"'");
538.                     }
539.
540.                     // Assigns a unique reservation number by iterating over the reserved list and
541.                     // checking there isn't a match.
542.                     for (int k = 0; k < reserved.size(); k++) {
543.                         Reservation held = (Reservation) reserved.get(k);
544.                         // Throws an AssertionError if the reserved bean bag fails to return any
545.                         // attributes.
546.                         assert (held != null) : "The reserved bean bag incorrectly has information
set to null.";
547.                         if (reservationNumber != held.getReservation()) {
548.                             break;
549.                         }
550.                         if (reservationNumber == held.getReservation()) {
551.                             reservationNumber += 1;
```

```
552.                         }
553.
554.                     }
555.
556.                     Reservation r = new Reservation(item, reservationNumber);
557.                     reserved.add(r);
558.                     available.remove(item);
559.                     // Prevents too many items from being reserved.
560.                     break;
561.                 }
562.             }
563.         }
564.
565.         // Throws an exception for unrecognised bean bags.
566.         if (!recognised) {
567.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
    found.");
568.         }
569.
570.         return reservationNumber;
571.     }
572.
573.     /**
574.      * {@inheritDoc}
575.      */
576.     @Override
577.     public int reservedBeanBagsInStock() {
578.         return reserved.size();
579.     }
580.
581.     /**
582.      * {@inheritDoc}
583.      */
584.     @Override
585.     public void resetSaleAndCostTracking() {
586.         // Sets sold list to be null for garbage collection before creating a new empty
587.         // list, which enables more efficient use of memory.
588.         sold = null;
589.         sold = new ObjectArrayList();
590.
591.         // Throws an AssertionError if the sales haven't been reset correctly.
592.         assert (this.getNumberOfSoldBeanBags() == 0) : "Sales have not been reset correctly.";
593.     }
594.
595.     /**
596.      * {@inheritDoc}
597.      */
598.     @Override
599.     public void saveStoreContents(String filename) throws IOException {
600.         try (ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(filename))) {
601.             out.writeObject(available);
602.             out.writeObject(reserved);
603.             out.writeObject(sold);
604.             System.out.printf("Saved in %s%n", filename);
605.         }
606.     }
607.
608.     /**
```

```
609.      * {@inheritDoc}}
610.      */
611.     @Override
612.     public void sellBeanBags(int num, String id)
613.             throws BeanBagNotInStockException, InsufficientStockException,
IllegalNumberOfBeanBagsSoldException,
614.             PriceNotSetException, BeanBagIDNotRecognisedException, IllegalIDException {
615.         boolean recognised = false;
616.
617.         // Checks if there is enough available stock of that item to sell.
618.         Checks.validSale(num, id, countBeanBags(new ObjectArrayList[] { available }, id));
619.
620.         // For the number of bean bags the customer wishes to sell, iterates over the
621.         // list of available stock and checks for matching bean bags.
622.         for (int i = 0; i < num; i++) {
623.             for (int j = 0; j < available.size(); j++) {
624.                 BeanBag item = (BeanBag) available.get(j);
625.                 // Searches for matching IDs to find which bean bags to remove from stock list.
626.                 if (item.getIdentifier().equalsIgnoreCase(id)) {
627.                     // Throws an exception if no price has been set for the bean bag.
628.                     if (item.getPriceInPence() == 0) {
629.                         throw new PriceNotSetException("No price has been set for this item '" + id +
"'");
630.                     }
631.
632.                     sold.add(item);
633.                     available.remove(item);
634.                     recognised = true;
635.                     // Prevents too many items from being sold.
636.                     break;
637.                 }
638.             }
639.         }
640.
641.         // Throws an exception for unrecognised bean bags.
642.         if (!recognised) {
643.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
found.");
644.         }
645.     }
646.
647.     /**
648.      * {@inheritDoc}}
649.      */
650.     @Override
651.     public void sellBeanBags(int reservationNumber) throws ReservationNumberNotRecognisedException {
652.         // Iterates over the reserved items for a bean bag with a matching reservation
653.         // number and adds it back to the stock list, removing it from the reserved
654.         // list.
655.         sold.add(manageReservations(reservationNumber).getAttributes());
656.         reserved.remove(manageReservations(reservationNumber));
657.     }
658.
659.     /**
660.      * {@inheritDoc}}
661.      */
662.     @Override
663.     public void setBeanBagPrice(String id, int priceInPence)
```

```
664.             throws InvalidPriceException, BeanBagIDNotRecognisedException, IllegalIDException {
665.         // Assumes bean bag is unrecognised until it finds the bean bag with the
666.         // mentioned ID.
667.         boolean recognised = false;
668.         ObjectArrayList[] objects = { available, reserved };
669.
670.         // Checks the given ID, and sets the price to that ID if valid.
671.         Checks.validId(id);
672.
673.         // Throws an exception if the user attempts to set a price of a negative value.
674.         if (priceInPence < 0) {
675.             throw new InvalidPriceException("The price '" + priceInPence + "' cannot be below zero
    pence.");
676.         }
677.
678.         // Accesses each element of array.
679.         for (ObjectArrayList object : objects) {
680.             // Iterates over the stock object and increments the count for each bean
681.             // bag with a matching ID.
682.             for (int j = 0; j < object.size(); j++) {
683.                 if (object == reserved) {
684.                     // Checks whether the price of a reserved bean bag was reduced whilst
685.                     // waiting for final sale and offer them the lower of the two prices.
686.                     Reservation held = (Reservation) object.get(j);
687.                     // Throws an AssertionError if the reserved bean bag fails to return any
688.                     // attributes.
689.                     assert (held != null) : "The reserved bean bag incorrectly has information set to
    null.";
690.                     BeanBag item = held.getAttributes();
691.                     if ((item.getIdentifier().equalsIgnoreCase(id)) && (priceInPence <
    item.getPriceInPence())) {
692.                         item.setPriceInPence(priceInPence);
693.                         recognised = true;
694.                     }
695.                 } else {
696.                     // Iterates over the list of available bean bags and sets the given price to
697.                     // bean bags with matching IDs.
698.                     BeanBag item = (BeanBag) object.get(j);
699.                     if (item.getIdentifier().equalsIgnoreCase(id)) {
700.                         item.setPriceInPence(priceInPence);
701.                         recognised = true;
702.                     }
703.                 }
704.             }
705.         }
706.
707.         // Throws an exception for unrecognised bean bags.
708.         if (!recognised) {
709.             throw new BeanBagIDNotRecognisedException("This bean bag ID '" + id + "' could not be
    found.");
710.         }
711.     }
712.
713.     /**
714.      * {@inheritDoc}
715.      */
716.     @Override
```

```
717.     public void unreserveBeanBags(int reservationNumber) throws
ReservationNumberNotRecognisedException {
718.         // Iterates over the reserved items for a bean bag with a matching reservation
719.         // number and adds it back to the stock list, removing it from the reserved
720.         // list.
721.         int existingPrice =
getExistingPrice(manageReservations(reservationNumber).getAttributes().getIdentifier());
722.
723.         // Updates bean bags to have the same price as the current price when moving
724.         // them back to the available bean bags list.
725.         if (existingPrice != 0) {
726.             manageReservations(reservationNumber).getAttributes().setPriceInPence(existingPrice);
727.         }
728.         available.add(manageReservations(reservationNumber).getAttributes());
729.         reserved.remove(manageReservations(reservationNumber));
730.     }
731. }
```

```java
package beanbags;

/**
 * Checks class which is called by some Store class methods. Contains exception
 * handler methods to disrupt the program flow when an unplanned event occurs,
 * such as invalid user inputs.
 *
 * @author 680033128
 * @author 690065435
 * @version 1.1
 *
 *
 */

public class Checks {
    /**
     * Exception handler method to check if bean bags have the same ID but different
     * name/manufacturer/information.
     *
     * @param newBeanBag The new bean bag object which the user wants to add to the
     *                   list of available bean bags.
     * @param available  List of available bean bags in the store.
     * @param reserved   List of reserved bean bags in the store.
     * @param sold       List of sold bean bags from the store.
     * @throws BeanBagMismatchException If the id already exists (as a current in
     *                                  stock bean bag, or one that has been
     *                                  previously stocked in the store, but the
     *                                  other stored elements (manufacturer, name
     *                                  and free text) do not match the pre-existing
     *                                  version.
     */
    public static void existingMismatch(BeanBag newBeanBag, ObjectArrayList available,
ObjectArrayList reserved,
            ObjectArrayList sold) throws BeanBagMismatchException {
        // Checks the object IDs in each of these lists to ensure no mismatch occurs.
        ObjectArrayList[] objects = { available, reserved, sold };
        BeanBag item;
        Reservation held;
        int i;
        ObjectArrayList obj;

        // Iterates over the lists one by one.
        for (i = 0; i < objects.length; i++) {
            // Accesses each element of array.
            obj = objects[i];

            // Updates the IDs in the available stock, reserved and sold list.
            for (int j = 0; j < obj.size(); j++) {
                // Casts reserved bean bags separately to available/sold bean bags, as they have
                // different attributes.
                if (obj == reserved) {
                    held = (Reservation) obj.get(j);
                    item = held.getAttributes();
                } else {
                    item = (BeanBag) obj.get(j);
                }
                // Checks name, manufacturer, and additional information for bean bags with the
                // same ID.
```

```
58.                 if (newBeanBag.getIdentifier().equalsIgnoreCase(item.getIdentifier())) {
59.                     // Checks whether names of the bean bags match.
60.                     if (!newBeanBag.getName().equalsIgnoreCase(item.getName())) {
61.                         throw new BeanBagMismatchException("The bean bags do not have the same name
'" + item.getName()
62.                                 + "' does not match '" + newBeanBag.getName() + "' for '" +
item.getIdentifier() + "'");
63.                     }
64.                     // Checks if the manufacturers of the bean bags match.
65.                     if (!newBeanBag.getManufacturer().equalsIgnoreCase(item.getManufacturer())) {
66.                         throw new BeanBagMismatchException("The bean bags do not come from the same
manufacturer '"
67.                                 + item.getManufacturer() + "' does not match '" +
newBeanBag.getManufacturer()
68.                                 + "' for '" + item.getIdentifier() + "'");
69.                     }
70.                     // Checks if the additional information on the bean bags match.
71.                     if (!newBeanBag.getInformation().equalsIgnoreCase(item.getInformation())) {
72.                         throw new BeanBagMismatchException("The bean bags do not have the same
information "
73.                                 + "associated with them '" + item.getInformation() + "' does not
match '"
74.                                 + newBeanBag.getInformation() + "' for '" + item.getIdentifier() +
"'");
75.                     }
76.                 }
77.             }
78.         }
79.     }
80.
81.     /**
82.      * Exception handler method to check if an ID of a bean bag has a valid format.
83.      *
84.      * @param id ID of bean bag.
85.      * @throws IllegalIDException If the ID is not a positive eight character
86.      *                            hexadecimal number.
87.      */
88.     public static void validId(String id) throws IllegalIDException {
89.         // Checks whether the ID consists of eight characters.
90.         if (id.length() == 8) {
91.             try {
92.                 // Throws an exception for bean bag IDs which aren't positive hexadecimals.
93.                 if ((int) Long.parseLong(id, 16) < 0) {
94.                     throw new IllegalIDException(
95.                             "Invalid hexadecimal identifier, '" + id + "'  is not a positive
number");
96.                 }
97.                 // Throws an exception for bean bag IDs which aren't hexadecimal numbers.
98.             } catch (NumberFormatException e) {
99.                 throw new IllegalIDException(
100.                         "Invalid hexadecimal identifier, '" + id + "'  is not a hexadecimal number");
101.             }
102.             // Throws an exception for IDs which don't have a length of 8.
103.         } else {
104.             throw new IllegalIDException(
105.                     "Invalid hexadecimal identifier, '" + id + "'  is not eight characters in
length");
106.         }
```

```
107.     }
108.
109.     /**
110.      * Exception handler method to check if reservations are valid.
111.      *
112.      * @param num       Number of bean bags the user wants to reserve.
113.      * @param id        ID of bean bag the user wants to reserve.
114.      * @param available List of available bean bags of the given ID.
115.      * @throws BeanBagNotInStockException               If the bean bag has
116.      *                                                  previously been in stock,
117.      *                                                  but is now out of stock.
118.      * @throws InsufficientStockException               If the bean bag is in stock,
119.      *                                                  but not enough are available
120.      *                                                  (i.e. in stock and not
121.      *                                                  reserved) to meet sale
122.      *                                                  demand.
123.      * @throws IllegalNumberOfBeanBagsReservedException If the number of bean bags
124.      *                                                  requested to reserve is
125.      *                                                  fewer than 1.
126.      * @throws IllegalIDException                       If the ID is not a positive
127.      *                                                  eight character hexadecimal
128.      *                                                  number.
129.      */
130.     public static void validReservation(int num, String id, int available) throws
BeanBagNotInStockException,
131.             InsufficientStockException, IllegalNumberOfBeanBagsReservedException, IllegalIDException
{
132.         Checks.validId(id);
133.
134.         // Throws an exception if the user attempts to reserve a non-positive number of
135.         // bean bags.
136.         if (num <= 0)
137.             throw new IllegalNumberOfBeanBagsReservedException(
138.                     "The number of bean bags '" + num + "' reserved cannot be less than zero");
139.         Checks.validId(id);
140.         // Throws an exception if there are no bean bags available for reservation.
141.         if (available == 0)
142.             throw new BeanBagNotInStockException("None of these bean bags '" + id + "' are available
for reservation");
143.         // Throws an exception if there aren't enough bean bags available for
144.         // reservation.
145.         if (num > available)
146.             throw new InsufficientStockException("Insufficient stock available for reservation; only
'" + available
147.                     + "' of '" + id + "' are in stock.");
148.     }
149.
150.     /**
151.      * Exception handler method to check if sales are valid.
152.      *
153.      * @param num       Number of bean bags the user wants to reserve.
154.      * @param id        ID of bean bag the user wants to reserve.
155.      * @param available List of available bean bags of the given ID.
156.      * @throws BeanBagNotInStockException           If the bean bag has previously
157.      *                                              been in stock, but is now out of
158.      *                                              stock.
159.      * @throws InsufficientStockException           If the bean bag is in stock, but
160.      *                                              not enough are available (i.e.
```

```
161.      *                                              in stock and not reserved) to
162.      *                                              meet sale demand.
163.      * @throws IllegalNumberOfBeanBagsSoldException If an attempt is being made to
164.      *                                              sell fewer than 1 bean bag.
165.      * @throws IllegalIDException                   If the ID is not a positive
166.      *                                              eight character hexadecimal
167.      *                                              number.
168.      */
169.     public static void validSale(int num, String id, int available) throws
BeanBagNotInStockException,
170.             InsufficientStockException, IllegalNumberOfBeanBagsSoldException, IllegalIDException {
171.         Checks.validId(id);
172.
173.         // Throws an exception if the user attempts to sell a non-positive number of
174.         // bean bags.
175.         if (num <= 0)
176.             throw new IllegalNumberOfBeanBagsSoldException(
177.                     "The number of bean bags '" + num + "' sold cannot be less than zero");
178.         // Throws an exception if there are no bean bags available for sale.
179.         if (available == 0)
180.             throw new BeanBagNotInStockException("None of these bean bags '" + id + "' are available
for sale");
181.         // Throws an exception if there aren't enough bean bags available for sale.
182.         if (num > available)
183.             throw new InsufficientStockException(
184.                     "Insufficient stock available for sale; only '" + available + "' of '" + id + "'
are in stock.");
185.     }
186. }
```

```java
package beanbags;

import java.io.Serializable;

/**
 * BeanBag class which implements Serializable. Creates a new instance of a bean
 * bag object and changes/returns their respective attributes using
 * setter/getter methods.
 *
 * @author 680033128
 * @author 690065435
 * @version 1.1
 *
 *
 */

public class BeanBag implements Serializable {
    // Private instance variables.
    private String manufacturer;
    private String name;
    private String id;
    private short year;
    private byte month;
    private String information;
    private int priceInPence;

    /**
     * Constructor for initialising bean bag objects. Bean bag objects contain the
     * following attributes.
     *
     * @param manufacturer Maker of the beanbag.
     * @param name         Given name of the beanbag.
     * @param id           Unique identifier for identical beanbags.
     * @param year         Year of manufacturer.
     * @param month        Month of manufacturer.
     * @param information  Optional free text component containing further details.
     * @param priceInPence Price in pence.
     *
     */
    public BeanBag(String manufacturer, String name, String id, short year, byte month, String
information,
            int priceInPence) {
        this.manufacturer = manufacturer;
        this.name = name;
        this.id = id;
        this.year = year;
        this.month = month;
        this.information = information;
        this.priceInPence = priceInPence;
    }

    /**
     * Public getter method for unique identifiers.
     *
     * @return Returns the identifier associated with a particular beanbag.
     *
     */
    public String getIdentifier() {
```

```
58.         return id;
59.     }
60.
61.     /**
62.      * Public getter method for bean bag manufacturer.
63.      *
64.      * @return Returns the manufacturer of a bean bag.
65.      *
66.      */
67.     public String getManufacturer() {
68.         return manufacturer;
69.     }
70.
71.     /**
72.      * Public getter method for optional information.
73.      *
74.      * @return Returns the free text component of a bean bag.
75.      *
76.      */
77.     public String getInformation() {
78.         return information;
79.     }
80.
81.     /**
82.      * Public getter method for month of production.
83.      *
84.      * @return Returns the production month of a bean bag.
85.      *
86.      */
87.     public byte getMonth() {
88.         return month;
89.     }
90.
91.     /**
92.      * Public getter method for retrieving the name.
93.      *
94.      * @return Returns the name of a bean bag.
95.      *
96.      */
97.     public String getName() {
98.         return name;
99.     }
100.
101.    /**
102.     * Public getter method for the cost of a bean bag in pence.
103.     *
104.     * @return Returns the price in pence of a bean bag.
105.     *
106.     */
107.    public int getPriceInPence() {
108.        return priceInPence;
109.    }
110.
111.    /**
112.     * Public getter method for year of production.
113.     *
114.     * @return Returns the production year of a bean bag.
115.     *
```

```
116.      */
117.     public short getYear() {
118.         return year;
119.     }
120. 
121.     /**
122.      * Public setter method for unique identifiers.
123.      *
124.      * @param id updates the identifier associated with a particular beanbag.
125.      *
126.      */
127.     public void setIdentifier(String id) {
128.         this.id = id;
129.     }
130. 
131.     /**
132.      * Public setter method for optional information. Method included for future
133.      * development.
134.      *
135.      * @param information Updates the identifier associated with a particular
136.      *                    beanbag.
137.      *
138.      */
139.     public void setInformation(String information) {
140.         this.information = information;
141.     }
142. 
143.     /**
144.      * Public setter method for maker of the beanbag. Method included for future
145.      * development.
146.      *
147.      * @param manufacturer Updates the maker of the bean bag.
148.      *
149.      */
150.     public void setManufacturer(String manufacturer) {
151.         this.manufacturer = manufacturer;
152.     }
153. 
154.     /**
155.      * Public setter method for month of production. Method included for future
156.      * development.
157.      *
158.      * @param month Updates the month of production.
159.      *
160.      */
161.     public void setMonth(byte month) {
162.         this.month = month;
163.     }
164. 
165.     /**
166.      * Public setter method for name of bean bag. Method included for future
167.      * development.
168.      *
169.      * @param name Updates the name.
170.      *
171.      */
172.     public void setName(String name) {
173.         this.name = name;
```

```
174.     }
175.
176.     /**
177.      * Public setter method for editing the price of a bean bag.
178.      *
179.      * @param priceInPence Changes the price of an item.
180.      *
181.      */
182.     public void setPriceInPence(int priceInPence) {
183.         this.priceInPence = priceInPence;
184.     }
185.
186.     /**
187.      * Public setter method for year of production. Method included for future
188.      * development.
189.      *
190.      * @param year Updates the month of production.
191.      *
192.      */
193.     public void setYear(short year) {
194.         this.year = year;
195.     }
196. }
```

```java
package beanbags;

import java.io.Serializable;

/**
 * Reservation class which implements Serializable. Creates a new instance of a
 * reserved bean bag object and changes/returns the bean bag or reservation
 * number.
 *
 * @author 680033128
 * @author 690065435
 * @version 1.1
 *
 *
 */

public class Reservation implements Serializable {
    private BeanBag item;
    // Initialising private variable for reservation number.
    private int reference;

    /**
     * Constructor for initialising reserved bean bag objects. Reserved objects
     * contain the following attributes.
     *
     * @param item      This is equivalent to an instance of a bean bag object.
     * @param reference This contains the reservation number associated with a given
     *                  bean bag.
     *
     */
    public Reservation(BeanBag item, int reference) {
        this.item = item;
        this.reference = reference;
    }

    /**
     * Public getter method for finding all attributes of a particular bean bag.
     *
     * @return Returns a BeanBag object containing all attributes.
     *
     */
    public BeanBag getAttributes() {
        return item;
    }

    /**
     * Public getter method for finding the reservation number of a reserved item.
     *
     * @return Returns only the reservation number of a bean bag without any of its
     *         attributes.
     *
     */
    public int getReservation() {
        return reference;
    }

    /**
     * Public setter method for editing the attributes of a bean bag. Method
```

```
     * included for future development.
     *
     * @param item This replaces the existing BeanBag object related to a
     *             reservation number.
     *
     */
    public void setAttributes(BeanBag item) {
        this.item = item;
    }

    /**
     * Public setter method for editing the reservation number of a bean bag. Method
     * included for future development.
     *
     * @param reference This replaces the reservation number without making changes
     *                  to a bean bags attributes.
     *
     */
    public void setReservation(int reference) {
        this.reference = reference;
    }
}
```