

# Contents

|    |                                        |    |
|----|----------------------------------------|----|
| 1  | README                                 | 2  |
| 2  | BasicStorage.java                      | 4  |
| 3  | BoopingSite.java                       | 8  |
| 4  | BoopingSiteTest.java                   | 11 |
| 5  | HotelSortingByProximityComparator.java | 17 |
| 6  | HotelSortingByRatingsComparator.java   | 19 |
| 7  | Locker.java                            | 20 |
| 8  | LockerTest.java                        | 22 |
| 9  | LongTermStorage.java                   | 27 |
| 10 | LongTermTest.java                      | 28 |
| 11 | Spaceship.java                         | 32 |
| 12 | SpaceshipDepositoryTest.java           | 34 |
| 13 | SpaceshipTest.java                     | 35 |
| 14 | StorageWithLTS.java                    | 37 |

# 1 README

```
1 bar246802
2
3
4
5
6 =====
7 =      File description      =
8 =====
9 Spaceship.java - This class represents a spaceship
10 Locker.java - This class represents Locker inside the spaceship
11 LongTermStorage.java - This class represents LTS inside the spaceship
12 BasicStorage.java - This class is an abstract class for basic features of a storage unit
13 BoopingSiteTest.java - This class is used test class BoopingSite
14 BoopingSite.java - This class is part of the - Booping.com - a new hotel booking site.
15 HotelSortingByProximityComparator.java - This class is used to sort given list of hotels by proximity.
16 HotelSortingByRatingsComparator.java - This class is used to sort given list of hotels by Ratings.
17 LockerTest.java - This class is used test class Locker
18 LongTermTest.java - This class is used test class LongTermStorage
19 SpaceshipDepositoryTest.java - This class is used to run full tests on Spaceship.class, LockerTest.class and LongTermTest.cl
20 SpaceshipTest.java - this class is used test class Spaceship
21 StorageWithLTS.java - an abstract class for all the methods related to an object
22   who is connected to LTS and as such has regulations to follow
23
24
25
26 =====
27 =      Design      =
28 =====
29 I choose to use abstract classes such as StorageWithLTS and BasicStorage in order to reuse the features shared btw the objec
30 The StorageWithLTS extends the BasicStorage.
31 The locker extends inheritance from StorageWithLTS .
32 The LongTermStorage BasicStorage.
33 If I could I would have used interfaces with default implemntions of method - that way I could have divided the code more to
34
35
36
37
38 =====
39 = Implementation details =
40 =====
41 Regarding the hotels I implemented two Comparator classes: HotelSortingByProximityComparator and HotelSortingByRatingsCompar
42 I used arraylist to store the hotels inside the different method due to the ease we can sort it with the Collections.sort an
43
44 In the tests I used the approach of creating a test method for each respective method in the tested class - in same cases I
45 who were called from the current main method.
46
47 I used formatters to export the messages in all the classes - that way I could add parameters more easily.
48
49
50
51 =====
52 =      Answers to questions      =
53 =====
54
55 1) Locker - Q: How did you choose to store the information? Why did you prefer it to other methods?
56 A: I choose to store the storage information inside a HashMap - that way I could easily get the count of an existing item an
57 If I have chosen a simple array I would have to implement all these method myself.
58 2)LTS - Q: How did you choose to store the information? How is it different from Locker.java?
59 A: regarding the inventory the answer is the same as the locker but here we only extend the BasicStorage abstract class so w
```

60  
61  
62  
63  
64  
65  
66  
67

3) Q: explain how you chose the dataset for each test  
A: After realizing the first dataset is all the cities, the second is only one city and the last one is empty I made the de  
In must of the tests I used all 3 datasets - and with each dataset I could test another aspect because with the first I coul  
with small amount of data and the fact that all the hotels were in the same city made it easy to check the returned length i  
wanted to make sure the tested method still return empty array even though the dataset is empty.  
4) Q: Hotel - Explain your design decisions. What were your options? Why did you prefer one over another?  
A: I covered some of this in Implementation details, the part about the Comparator classes was very much obligatory but I co  
as I said before I choose the arraylist due to the ease we can sort it with the Collections.sort and moving back and forth b

## 2 BasicStorage.java

```
1  import oop.ex3.spaceship.Item;
2  import java.text.MessageFormat;
3  import java.util.HashMap;
4  import java.util.Map;
5
6  /**
7   * This class is an abstract class for basic features of a storage unit
8   * @author Bar Melinarskiy
9   * @version 16/8/20
10  */
11  public abstract class BasicStorage
12  {
13      //Constants
14      protected static final String ERROR_PREFIX =
15          "Error: Your request cannot be completed at this time. Problem: ";
16      protected static final String ERROR_INSERT_CONTRADICTION =
17          "the locker cannot contain items of type {0}, as it contains a contradicting item";
18      protected static final String ERROR_INSERT_OVERFLOW =
19          "no room for {0} items of type {1}";
20      protected static final int INSERT_ERROR_CODE_CONTRADICTION = -2;
21      protected static final int INSERT_ERROR_CODE = -1;
22      protected static final int SUCCESS = 0;
23      protected static final int INITIAL_SIZE = 0;
24      protected static final int ONE_HUNDRED = 100;
25      protected static final String ERROR_REMOVE_NON_EXISTING_AMOUNT =
26          "the locker does not contain {0} items of type {1}";
27      protected static final String ERROR_REMOVE_NEGATIVE_NUM =
28          "cannot remove a negative number of items of type {0}";
29      protected static final int ERROR_CODE = -1;
30
31
32      // instance variables
33      private int lockerCapacity = INITIAL_SIZE;
34      private int occupiedSpace = INITIAL_SIZE;
35      private Map<String, Integer> inventory = new HashMap<String, Integer>();
36
37      // instance methods
38      public abstract int addItem(Item item, int n);
39
40      /** Removing item/s from this locker
41       * @param item the type of item we are trying to remove from this locker
42       * @param n the number of copies of the current item we are trying to remove
43       * @return 0 if we succeed removing these items, -1 otherwise.
44       */
45      public int removeItem(Item item, int n)
46      {
47          String currentType = item.getType();
48          if(n < INITIAL_SIZE)
49          {
50              String msgFormat = ERROR_PREFIX + ERROR_REMOVE_NEGATIVE_NUM;
51              String errorMessage = MessageFormat.format(msgFormat, currentType);
52              System.out.println(errorMessage);
53              return ERROR_CODE;
54          }
55          else if(n > getItemCount(currentType))
56          {
57              String msgFormat = ERROR_PREFIX + ERROR_REMOVE_NON_EXISTING_AMOUNT;
58              String errorMessage = MessageFormat.format(msgFormat, n, currentType);
59              System.out.println(errorMessage);
```

```

60         return ERROR_CODE;
61     }
62     else if(n > INITIAL_SIZE)
63     {
64         removeFromInventory(item, n);
65     }
66     return SUCCESS;
67 }
68 /** reset this storage
69 */
70 public void initializeStorage()
71 {
72     getInventory().clear();
73     setOccupiedSpace(INITIAL_SIZE);
74 }
75
76 /** Get the number of copies stored in this locker of the given type
77 * @param type the type of item we wanna check
78 * @return the number of Items of type type the locker contains.
79 */
80 public int getItemCount(String type)
81 {
82     int itemCount = INITIAL_SIZE;
83     //Check if the given type is actually stored inside this locker
84     if(inventory.containsKey(type))
85     {
86         itemCount = inventory.get(type);
87     }
88     return itemCount;
89 }
90 /** Get the a map of all the item types contained in the locker
91 * @return a map of all the item types contained in the locker,
92 * and their respective quantities.
93 */
94 public Map<String, Integer> getInventory()
95 {
96     return inventory;
97 }
98 /** Get the long-term storage's total capacity
99 * @return the long-term storage's total capacity.
100 */
101 public int getCapacity()
102 {
103     return lockerCapacity;
104 }
105 /** Get the long-term storage's available capacity
106 * @return a the long-term storage's available capacity
107 */
108 public int getAvailableCapacity()
109 {
110     return lockerCapacity - occupiedSpace;
111 }
112
113 /** Set the long-term storage's total capacity
114 * @param capacity the long-term storage's total capacity.
115 */
116 protected void setCapacity(int capacity)
117 {
118     lockerCapacity = capacity;
119 }
120 /** Set the long-term storage's available capacity
121 * @param size the long-term storage's available capacity
122 */
123 protected void setOccupiedSpace(int size)
124 {
125     occupiedSpace = size;
126 }
127

```

```

128     /** Get the long-term storage's occupied capacity
129     * @return the long-term storage's occupied capacity
130     */
131     protected int getOccupiedSpace()
132     {
133         return occupiedSpace;
134     }
135
136     /** Adding a new item/s to this locker's inventory
137     * @param item the type of item we are trying to add to this locker
138     * @param n the number of copies of the current item we are trying to add
139     */
140     protected void addToInventory(Item item, int n)
141     {
142         int newCount = n + getItemCount(item.getType());
143         getInventory().put(item.getType(), newCount);
144         //Update available capacity
145         int volumeAddition = n * item.getVolume();
146         setOccupiedSpace(getOccupiedSpace() + volumeAddition);
147     }
148
149     /** Remove a item/s from this locker's inventory
150     * @param item the type of item we are trying to remove from this locker's inventory
151     * @param n the number of copies of the current item we are trying to remove
152     */
153     protected void removeFromInventory(Item item, int n)
154     {
155         int newCount = n - getItemCount(item.getType());
156         if(newCount > INITIAL_SIZE)
157         {
158             getInventory().put(item.getType(), newCount);
159         }
160         else
161         {
162             getInventory().remove(item.getType());
163         }
164         //Update available capacity
165         int volumeDecreased = n * item.getVolume();
166         setOccupiedSpace(getOccupiedSpace() - volumeDecreased);
167     }
168
169     /** Get given type storage units % of a the given Storage-Unit
170     * @param item the item we wanna check
171     * @param n the number of copies of the current item we are trying to add
172     * @return the given type storage units % of a the given Storage-Unit
173     */
174     protected double getItemOccupiedPercentage(Item item, int n)
175     {
176         double percentage = INITIAL_SIZE;
177         int itemCount = getItemCount(item.getType());
178         if(itemCount > INITIAL_SIZE || n > INITIAL_SIZE) //check if this item is stored inside this locker
179         {
180             int totalVolume = item.getVolume() * itemCount + n * item.getVolume();
181             percentage = (double)totalVolume / getCapacity() * ONE_HUNDRED;
182         }
183
184         return percentage;
185     }
186
187     /** Calc given type storage units % of a the given Storage-Unit
188     * @param item the item we wanna check
189     * @param n the number of copies of the current item
190     * @return the expected given type storage units % of a the given Storage-Unit
191     */
192     protected double calcItemOccupiedPercentage(Item item, int n)
193     {
194         double percentage = INITIAL_SIZE;
195         if(n > INITIAL_SIZE) //check if this item is stored inside this locker
196         {

```

```

196         int totalVolume = n * item.getVolume();
197         percentage = (double)totalVolume / getCapacity() * ONE_HUNDRED;
198     }
199
200     return percentage;
201 }
202 /** Check if this locker has enough room for the given items
203 * @param item the item we wanna check
204 * @param n the number of copies of the current item we are trying to add
205 * @return true if there is enough room, false otherwise.
206 */
207 protected boolean checkIfThereEnoughRoom(Item item, int n)
208 {
209     //Check if there enough room in the locker for all the new items
210     int totalVolume = item.getVolume() * n;
211     if(getAvailableCapacity() < totalVolume || n < INITIAL_SIZE)
212     {
213         String msgFormat = ERROR_PREFIX + ERROR_INSERT_OVERFLOW;
214         String errorMessage = MessageFormat.format(msgFormat, n, item.getType());
215         System.out.println(errorMessage);
216         return false;
217     }
218
219     return true;
220 }
221 }

```

### 3 BoopingSite.java

```
1  import oop.ex3.searchengine.*;
2  import java.util.*;
3  /**
4   * This class is part of the - Booping.com - a new hotel booking site.
5   * @author Bar Melinarskiy
6   * @version 26/8/20
7   */
8  public class BoopingSite
9  {
10     //Constants
11     private static final int EMPTY = 0;
12     // instance variables
13     private String datasetName;
14
15     /** Contractor
16      * @param name the name of the dataset
17      */
18     public BoopingSite(String name)
19     {
20         datasetName = name;
21     }
22     /** Get hotels located in the given city sorted by ratings
23      * @param city the city the check
24      * @return an array of hotels located in the given city, sorted from
25      * the highest star-rating to the lowest. Hotels that have the
26      * same rating will be organized according to the alphabet order of
27      * the hotel's (property) name. In case there are no hotels in the
28      * given city, this method returns an empty array
29      */
30     public Hotel[] getHotelsInCityByRating(String city)
31     {
32         ArrayList<Hotel> hotelsToSort = getHotelsInCity(city);
33         if(!hotelsToSort.isEmpty())
34         {
35             Collections.sort(hotelsToSort, new HotelSortingByRatingsComparator());
36         }
37
38         Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
39         hotelsToSort.toArray(hotelsToReturn);
40         return hotelsToReturn;
41     }
42     /** Get hotels sorted according to their Euclidean distance from the given geographic location
43      * @param latitude geographic coordinate that specifies the north-south position of a point
44      * on the Earth's surface. valid values are btw [-90,90] degrees.
45      * @param longitude a geographic coordinate that specifies the east-west position of a point
46      * on the Earth's surface. valid values are btw [-180,180] degrees.
47      * @return an array of hotels, sorted according to their Euclidean distance from the given geographic
48      * location, in ascending order. Hotels that are at the same distance from the given location are
49      * organized according to the number of points-of-interest for which
50      * they are close to (in a decreasing order).
51      * In case of illegal input, this method returns an empty array.
52      */
53     public Hotel[] getHotelsByProximity(double latitude, double longitude)
54     {
55         ArrayList<Hotel> hotelsToSort = new ArrayList<Hotel>();
56         if(checkCoordinates(latitude, longitude) == true)
57         {
58             hotelsToSort = getAllHotels();
59             Collections.sort(hotelsToSort, new HotelSortingByProximityComparator(latitude, longitude));
```



```

60     }
61
62     Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
63     hotelsToSort.toArray(hotelsToReturn);
64     return hotelsToReturn;
65 }
66 /** Get hotels in given city sorted according to their Euclidean distance
67 * from the given geographic location
68 * @param city the city the check
69 * @param latitude geographic coordinate that specifies the north-south position of a point
70 * on the Earth's surface. valid values are btw [-90,90] degrees.
71 * @param longitude a geographic coordinate that specifies the east-west position of a point
72 * on the Earth's surface. valid values are btw [-180,180] degrees.
73 * @return an array of hotels in the given city, sorted according to their Euclidean distance
74 * from the given geographic location, in ascending order.
75 * Hotels that are at the same distance from the given location
76 * are organized according to the number of points-of-interest for
77 * which they are close to (in a decreasing order).
78 * In case of illegal input, this method returns an empty array.
79 */
80 public Hotel[] getHotelsInCityByProximity(String city, double latitude, double longitude)
81 {
82     ArrayList<Hotel> hotelsToSort = new ArrayList<Hotel>();
83     if(checkCoordinates(latitude, longitude) == true)
84     {
85         hotelsToSort = getHotelsInCity(city);
86         Collections.sort(hotelsToSort, new HotelSortingByProximityComparator(latitude, longitude));
87     }
88
89     Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
90     hotelsToSort.toArray(hotelsToReturn);
91     return hotelsToReturn;
92 }
93
94 /** Get all the hotels from the dataset in the given city
95 * @param city - the city to filter with
96 * @return list of all the hotels from the dataset in the given city
97 */
98 private ArrayList<Hotel> getHotelsInCity(String city)
99 {
100     ArrayList<Hotel> hotels = getAllHotels();
101     ArrayList<Hotel> filteredList = new ArrayList<Hotel>();
102     for (Hotel hotel : hotels)
103     {
104         if (hotel.getCity().equals(city))
105         {
106             filteredList.add(hotel);
107         }
108     }
109
110     return filteredList;
111 }
112
113 /** Get all the hotels from the dataset
114 * @return list of all the hotels from the dataset
115 */
116 private ArrayList<Hotel> getAllHotels()
117 {
118     Hotel[] hotels = HotelDataset.getHotels(datasetName);
119     ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
120     return hotelList;
121 }
122 /** Check if the given coordinates are valid
123 * @param latitude geographic coordinate that specifies the north-south position of a point
124 * on the Earth's surface. valid values are btw [-90,90] degrees.
125 * @param longitude a geographic coordinate that specifies the east-west position of a point
126 * on the Earth's surface. valid values are btw [-180,180] degrees.
127 * @return true if the given coordinates are valid, false otherwise.

```

```

128     */
129     private boolean checkCoordinates(double latitude, double longitude)
130     {
131         final int LATITUDE_LOW = -90;
132         final int LATITUDE_HIGH = 90;
133         final int LONGITUDE_LOW = -180;
134         final int LONGITUDE_HIGH = 180;
135
136         if(latitude < LATITUDE_LOW || latitude > LATITUDE_HIGH || longitude < LONGITUDE_LOW ||
137            longitude > LONGITUDE_HIGH)
138         {
139             return false;
140         }
141
142         return true;
143     }
144 }

```

## 4 BoopingSiteTest.java

```
1  import oop.ex3.searchengine.*;
2  import org.junit.*;
3  import java.text.MessageFormat;
4  import java.util.ArrayList;
5  import java.util.Arrays;
6  import static org.junit.Assert.*;
7
8  /**
9   * This class is used test class BoopingSite.
10   * @author Bar MelinarSKIY
11   * @version 26/8/20
12   */
13  public class BoopingSiteTest
14  {
15      //Constants
16      protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
17      private static final String BIG_DATASET = "hotels_dataset.txt";
18      private static final String SMALL_DATASET_1 = "hotels_tst1.txt"; //all hotels in Manali
19      private static final String SMALL_DATASET_2 = "hotels_tst2.txt"; //empty file
20      private static final String NON_EXISTING_CITY = "###@$$#@$$#.nonExisting";
21      private static final int EMPTY = 0;
22      private static final int ZERO = 0;
23      private static final Hotel[] emptyHotelArray = new Hotel[EMPTY];
24      // instance variables
25      private static ArrayList<Hotel> longHotelList;
26      private static ArrayList<Hotel> shortHotelList1; //all hotels in Manali
27      private static ArrayList<Hotel> shortHotelList2; //empty list
28
29      private static ArrayList<String> citiesLongHotelList;
30      private static ArrayList<String> citiesShortHotelList1; // Only Manali
31      private static ArrayList<String> citiesShortHotelList2; //empty list
32
33      private BoopingSite boopingSiteTest1 = new BoopingSite(BIG_DATASET);
34      private BoopingSite boopingSiteTest2 = new BoopingSite(SMALL_DATASET_1);
35      private BoopingSite boopingSiteTest3 = new BoopingSite(SMALL_DATASET_2);
36
37      private static String firstLongCity;
38      private static String firstShort1City;
39      private static String firstShort2City;
40
41      // tests methods
42      /** build testers objects
43       */
44      @BeforeClass
45      public static void setUp()
46      {
47          longHotelList = getAllHotels(BIG_DATASET);
48          shortHotelList1 = getAllHotels(SMALL_DATASET_1);
49          shortHotelList2 = getAllHotels(SMALL_DATASET_2);
50
51          citiesLongHotelList = getAllCities(longHotelList);
52          citiesShortHotelList1 = getAllCities(shortHotelList1);
53          citiesShortHotelList2 = getAllCities(shortHotelList2);
54
55          firstLongCity = citiesLongHotelList.get(ZERO);
56          firstShort1City = citiesShortHotelList1.get(ZERO);
57          firstShort2City = "";
58      }
59  }
```

```

60  /** check method getHotelsInCityByRating
61  */
62  @Test
63  public void getHotelsInCityByRating()
64  {
65      final String method = "getHotelsInCityByRating";
66      //Check method on the large dataset
67      Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsInCityByRating(firstLongCity);
68      ArrayList<Hotel> firstCityHotelsLong = getAllHotelsInCities(longHotelList, firstLongCity);
69      checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);
70      checkRatingOrder(returnedHotelsLong, method);
71      //Check method on the small dataset
72      Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByRating(firstShort1City);
73      ArrayList<Hotel> firstCityHotelsShort1 = getAllHotelsInCities(shortHotelList1, firstShort1City);
74      checkAllHotelsExist(returnedHotelsShort1, firstCityHotelsShort1, method);
75      checkRatingOrder(returnedHotelsShort1, method);
76      //Check method on the empty dataset
77      Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsInCityByRating(firstShort2City);
78      ArrayList<Hotel> firstCityHotelsShort2 = getAllHotelsInCities(shortHotelList1, firstShort2City);
79      checkAllHotelsExist(returnedHotelsShort2, firstCityHotelsShort2, method);
80  }
81
82  /** check method getHotelsByProximity
83  */
84  @Test
85  public void getHotelsByProximity()
86  {
87      final String method = "getHotelsByProximity";
88      checkCoordinatesByProximity(boopingSiteTest1);
89
90      //Check method on the large dataset
91      Hotel hotel = longHotelList.get(ZERO);
92      Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsByProximity(
93          hotel.getLatitude(), hotel.getLongitude());
94      checkAllHotelsExist(returnedHotelsLong, longHotelList, method);
95      checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
96      //Check method on the large dataset - another hotel
97      hotel = longHotelList.get(8);
98      returnedHotelsLong = boopingSiteTest1.getHotelsByProximity(
99          hotel.getLatitude(), hotel.getLongitude());
100     checkAllHotelsExist(returnedHotelsLong, longHotelList, method);
101     checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
102     //Check method on the small dataset
103     hotel = shortHotelList1.get(ZERO);
104     Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsByProximity(
105         hotel.getLatitude(), hotel.getLongitude());
106     checkAllHotelsExist(returnedHotelsShort1, shortHotelList1, method);
107     checkProximityOrder(returnedHotelsShort1, hotel.getLatitude(), hotel.getLongitude(), method);
108     //Check method on the empty dataset
109     Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsByProximity(
110         hotel.getLatitude(), hotel.getLongitude());
111     checkAllHotelsExist(returnedHotelsShort2, shortHotelList2, method);
112 }
113
114 /** check method getHotelsInCityByProximity
115 */
116 @Test
117 public void getHotelsInCityByProximity()
118 {
119     final String method = "getHotelsInCityByProximity";
120     checkCoordinatesInCityByProximity(boopingSiteTest1);
121
122     //Check method on the large dataset
123     Hotel hotel = longHotelList.get(ZERO);
124     Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsInCityByProximity(firstLongCity,
125         hotel.getLatitude(), hotel.getLongitude());
126     ArrayList<Hotel> firstCityHotelsLong = getAllHotelsInCities(longHotelList, firstLongCity);
127     checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);

```

```

128         checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
129         //Check method on the large dataset - another hotel
130         hotel = longHotellList.get(8);
131         returnedHotelsLong = boopingSiteTest1.getHotelsInCityByProximity(firstLongCity,
132                                     hotel.getLatitude(), hotel.getLongitude());
133         firstCityHotelsLong = getAllHotelsInCities(longHotellList, firstLongCity);
134         checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);
135         checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
136         //Check method on the small dataset
137         hotel = shortHotellList1.get(ZERO);
138         Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByProximity(firstShort1City,
139                                     hotel.getLatitude(), hotel.getLongitude());
140         ArrayList<Hotel> firstCityHotelsShort1 = getAllHotelsInCities(shortHotellList1, firstShort1City);
141         checkAllHotelsExist(returnedHotelsShort1, firstCityHotelsShort1, method);
142         checkProximityOrder(returnedHotelsShort1, hotel.getLatitude(), hotel.getLongitude(), method);
143         //Check method on the empty dataset
144         Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsInCityByProximity(firstShort2City,
145                                     hotel.getLatitude(), hotel.getLongitude());
146         checkAllHotelsExist(returnedHotelsShort2, shortHotellList2, method);
147     }
148
149     /** check method checkNonExistingCity
150     */
151     @Test
152     public void checkNonExistingCity()
153     {
154         final String method = "checkNonExistingCity";
155         Hotel hotel = shortHotellList1.get(ZERO);
156         //Check method on the small dataset - getHotelsInCityByRating
157         Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByRating(NON_EXISTING_CITY);
158         checkAllHotelsExist(returnedHotelsShort1, shortHotellList2, method);
159
160         //Check method on the small dataset - getHotelsInCityByProximity
161
162         Hotel[] returnedHotelsShort2 = boopingSiteTest2.getHotelsInCityByProximity(NON_EXISTING_CITY,
163                                     hotel.getLatitude(), hotel.getLongitude());
164         checkAllHotelsExist(returnedHotelsShort2, shortHotellList2, method);
165     }
166
167     /** check the hotels are order correctly by proximity
168     * @param boopingSiteTest the boopingSite to test on
169     */
170     private void checkCoordinatesByProximity(BoopingSite boopingSiteTest)
171     {
172         final String method = "ByProximity - Checking non-valid coordinates";
173         String errorMessage = getErrorPrefix(method);
174         assertEquals(errorMessage, emptyHotelArray,
175                 boopingSiteTest.getHotelsByProximity(-900,-900));
176         assertEquals(errorMessage, emptyHotelArray,
177                 boopingSiteTest.getHotelsByProximity(900,-900));
178         assertEquals(errorMessage, emptyHotelArray,
179                 boopingSiteTest.getHotelsByProximity(-900,900));
180         assertEquals(errorMessage, emptyHotelArray,
181                 boopingSiteTest.getHotelsByProximity(900,900));
182     }
183     /** check the hotels in the current city are order correctly by proximity
184     * @param boopingSiteTest the boopingSite to test on
185     */
186     private void checkCoordinatesInCityByProximity(BoopingSite boopingSiteTest)
187     {
188         final String method = "InCityByProximity - Checking non-valid coordinates";
189         String errorMessage = getErrorPrefix(method);
190
191         String city = citiesLongHotellList.get(0);
192         assertEquals(errorMessage, emptyHotelArray,
193                 boopingSiteTest.getHotelsInCityByProximity(city,-900,-900));
194         assertEquals(errorMessage, emptyHotelArray,
195                 boopingSiteTest.getHotelsInCityByProximity(city,900,-900));

```

```

196         assertEquals(errorMessage, emptyHotelArray,
197             boopingSiteTest.getHotelsInCityByProximity(city, -900, 900));
198         assertEquals(errorMessage, emptyHotelArray,
199             boopingSiteTest.getHotelsInCityByProximity(city, 900, 900));
200     }
201
202     /** Get all the hotels from the dataset
203      * @param datasetName the dataset to fetch hotels from
204      * @return list of all the hotels from the dataset
205      */
206     private static ArrayList<Hotel> getAllHotels(String datasetName)
207     {
208         Hotel[] hotels = HotelDataset.getHotels(datasetName);
209         ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
210         return hotelList;
211     }
212
213     /** Get all the cities in the given hotels list
214      * @param hotels given hotels list to fetch cities from
215      * @return list of all the cities
216      */
217     private static ArrayList<String> getAllCities(ArrayList<Hotel> hotels)
218     {
219         ArrayList<String> citiesList = new ArrayList<String>();
220         for (Hotel hotel : hotels)
221         {
222             if (citiesList.contains(hotel.getCity()) == false)
223             {
224                 citiesList.add(hotel.getCity());
225             }
226         }
227
228         return citiesList;
229     }
230
231     /** Get all the hotels in a given city
232      * @param hotels given hotels list to fetch from
233      * @param city city to check
234      * @return list of all the hotels in the given city
235      */
236     private ArrayList<Hotel> getAllHotelsInCities(ArrayList<Hotel> hotels, String city)
237     {
238         ArrayList<Hotel> hotelsInCityList = new ArrayList<Hotel>();
239         for (Hotel hotel : hotels)
240         {
241             if (city.equals(hotel.getCity()) && hotelsInCityList.contains(hotel) == false)
242             {
243                 hotelsInCityList.add(hotel);
244             }
245         }
246
247         return hotelsInCityList;
248     }
249
250     /** Check all hotels exist
251      * @param hotels given hotels list to check
252      * @param hotelsSource hotels to check against
253      * @param source calling test name
254      */
255     private void checkAllHotelsExist(Hotel[] hotels, ArrayList<Hotel> hotelsSource, String source)
256     {
257         final String method = source + " - checking all hotels exist";
258         String errorMessage = getErrorPrefix(method);
259
260         assertNotNull(errorMessage, hotels);
261         ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
262         assertEquals(errorMessage, hotels.length, hotelsSource.size());
263         for (Hotel hotel : hotelList)

```

```

264     {
265         boolean exist = false;
266         for (Hotel hotelSource : hotelsSource)
267         {
268             if(hotelSource.getUniqueId().equals(hotel.getUniqueId()))
269             {
270                 exist = true;
271                 break;
272             }
273         }
274         assertTrue(errorMessage, exist);
275     }
276 }
277
278 /** Check all hotels are ordered by ratings
279  * @param hotels given hotels list to check
280  * @param source calling test name
281  */
282 private void checkRatingOrder(Hotel[] hotels, String source)
283 {
284     final String method = source + " - checking hotels order by rating";
285     String errorMessage = getErrorPrefix(method);
286
287     assertNotNull(errorMessage, hotels);
288     Hotel previousHotel = null;
289     for (Hotel hotel : hotels)
290     {
291         if(previousHotel == null)
292         {
293             previousHotel = hotel;
294         }
295         else
296         {
297             boolean ratingsOrder = previousHotel.getStarRating() > hotel.getStarRating();
298             boolean ratingsEq = previousHotel.getStarRating() == hotel.getStarRating();
299             boolean nameOrder = previousHotel.getPropertyName().compareTo(
300                 hotel.getPropertyName()) <= ZERO;
301             assertTrue(errorMessage, ratingsOrder || (ratingsEq && nameOrder));
302         }
303     }
304 }
305
306 /** Check all hotels are ordered by proximity
307  * @param hotels given hotels list to check
308  * @param latitude point latitude
309  * @param longitude point longitude
310  * @param source calling test name
311  */
312 private void checkProximityOrder(Hotel[] hotels, double latitude, double longitude, String source)
313 {
314     final String method = source + " - checking hotels order by proximity";
315     String errorMessage = getErrorPrefix(method);
316
317     assertNotNull(errorMessage, hotels);
318     Hotel previousHotel = null;
319     for (Hotel hotel : hotels)
320     {
321         if(previousHotel == null)
322         {
323             previousHotel = hotel;
324         }
325         else
326         {
327             double distancePrevious = calcDistance(latitude, longitude,
328                 previousHotel.getLatitude(), previousHotel.getLongitude());
329             double distanceCurrent = calcDistance(latitude, longitude,
330                 hotel.getLatitude(), hotel.getLongitude());
331             boolean distanceOrder = distancePrevious < distanceCurrent;
332             boolean distanceEq = distancePrevious == distanceCurrent;

```

```

332         boolean poiOrder = previousHotel.getNumPOI() >= hotel.getNumPOI();
333         assertTrue(errorMessage, distanceOrder || (distanceEq && poiOrder));
334     }
335 }
336 }
337
338 /**
339  * Calc distance btw given point and the geographic location
340  * @param x1 first point longitude
341  * @param y1 first point latitude
342  * @param x2 second point longitude
343  * @param y2 second point latitude
344  * @return distance btw given point and the geographic location
345  */
346 private double calcDistance(double x1, double y1, double x2, double y2)
347 {
348     final int TWO = 2;
349     return Math.sqrt(Math.pow(x1 - x2, TWO) +
350                     Math.pow(y1 - y2, TWO));
351 }
352
353 /** Get error msg
354  * @param method param to add to msg
355  * @return error msg
356  */
357 private String getErrorPrefix(String method)
358 {
359     String msgFormat = ERROR_PREFIX;
360     return MessageFormat.format(msgFormat, method);
361 }
362 }

```



## 5 HotelSortingByProximityComparator.java

```
1  import oop.ex3.searchengine.Hotel;
2  import java.util.Comparator;
3  /**
4   * This class is used to sort given list of hotels by proximity.
5   * @author Bar Melinarskiy
6   * @version 26/8/20
7   */
8  public class HotelSortingByProximityComparator implements Comparator<Hotel>
9  {
10     //Constants
11     protected static final int EQ = 0;
12     protected static final int GT = 1;
13     protected static final int LT = -1;
14
15     double latitudeCoordinate;
16     double longitudeCoordinate;
17     /**
18      * Creates a Comparator instance.
19      * @param latitude longitude of point.
20      * @param longitude longitude of longitude.
21      */
22     HotelSortingByProximityComparator(double latitude, double longitude)
23     {
24         latitudeCoordinate = latitude;
25         longitudeCoordinate = longitude;
26     }
27     /**
28      * Compare btw two hotel by proximity & POI.
29      * @param hotel1 first hotel.
30      * @param hotel2 second hotel.
31      * @return the value {@code 0} if {@code x == y};
32      * a value less than {@code 0} if {@code x < y}; and
33      * a value greater than {@code 0} if {@code x > y}
34      */
35     @Override
36     public int compare(Hotel hotel1, Hotel hotel2)
37     {
38         // for comparison
39         int distanceCompare = compareDistance(hotel1, hotel2);
40         int poiCompare = LT;
41         if(hotel2.getNumPOI() == hotel1.getNumPOI())
42         {
43             poiCompare = EQ;
44         }
45         else if(hotel2.getNumPOI() > hotel1.getNumPOI())
46         {
47             poiCompare = GT;
48         }
49
50         // 2-level comparison using if-else block
51         if (distanceCompare == 0)
52         {
53             return ((poiCompare == 0) ? distanceCompare : poiCompare);
54         } else
55         {
56             return distanceCompare;
57         }
58     }
59     /**
```

```

60      * Compares two hotel by their Euclidean distance from the given geographic location
61      * @param hotel1 the first hotel to compare
62      * @param hotel2 the second hotel to compare
63      * @return the value {@code 0} if {@code x == y};
64      *         a value less than {@code 0} if {@code x < y}; and
65      *         a value greater than {@code 0} if {@code x > y}
66      */
67     private int compareDistance(Hotel hotel1, Hotel hotel2)
68     {
69         double distance1 = calcDistance(hotel1.getLongitude(), hotel1.getLatitude());
70         double distance2 = calcDistance(hotel2.getLongitude(), hotel2.getLatitude());
71         return Double.compare(distance1, distance2);
72     }
73     /**
74      * Calc distance btw given point and the geographic location
75      * @param x longitude
76      * @param y latitude
77      * @return distance btw given point and the geographic location
78      */
79     private double calcDistance(double x, double y)
80     {
81         final int TWO = 2;
82         return Math.sqrt(Math.pow(longitudeCoordinate - x, TWO) +
83                             Math.pow(latitudeCoordinate - y, TWO));
84     }
85
86
87
88 }

```

## 6 HotelSortingByRatingsComparator.java

```
1  import oop.ex3.searchengine.*;
2  import java.util.Comparator;
3  /**
4   * This class is used to sort given list of hotels by Ratings.
5   * @author Bar Melinarskiy
6   * @version 26/8/20
7   */
8  public class HotelSortingByRatingsComparator implements Comparator<Hotel>
9  {
10     //Constants
11     protected static final int EQ = 0;
12     protected static final int GT = 1;
13     protected static final int LT = -1;
14
15     /**
16      * Compare btw two hotel by ratings & name.
17      * @param hotel1 first hotel.
18      * @param hotel2 second hotel.
19      * @return the value {@code 0} if {@code x == y};
20      * a value less than {@code 0} if {@code x < y}; and
21      * a value greater than {@code 0} if {@code x > y}
22      */
23     @Override
24     public int compare(Hotel hotel1, Hotel hotel2)
25     {
26         // for comparison
27         int ratingCompare = LT;
28         if(hotel2.getStarRating() == hotel1.getStarRating())
29         {
30             ratingCompare = EQ;
31         }
32         else if(hotel2.getStarRating() > hotel1.getStarRating())
33         {
34             ratingCompare = GT;
35         }
36         int nameCompare = hotel1.getPropertyName().compareTo(hotel2.getPropertyName());
37
38         // 2-level comparison using if-else block
39         if (ratingCompare == 0)
40         {
41             return ((nameCompare == 0) ? ratingCompare : nameCompare);
42         } else
43         {
44             return ratingCompare;
45         }
46     }
47 }
```

## 7 Locker.java

```
1  import oop.ex3.spaceship.Item;
2
3  import java.text.MessageFormat;
4  /**
5   * This class represents Locker inside the spaceship
6   * @author Bar Melinarskiy
7   * @version 16/8/20
8   */
9  public class Locker extends StorageWithLTS
10 {
11     //Constants
12     private static final String WARNING_INSERT =
13         "Warning: Action successful, but has caused items to be moved to storage";
14     private static final int MOVING_TO_LTS_THRESHOLD = 50;
15     private static final int KEEPING_IN_LOCKER_THRESHOLD = 20;
16
17     /** Contractor
18      * @param lts the current ship's Long-Term storage unit.
19      * @param capacity the limit capacity of this locker.
20      * @param constraints an array of pairs of two items that are NOT allowed to reside
21      * together in a locker.
22      */
23     public Locker(LongTermStorage lts, int capacity, Item[][] constraints)
24     {
25         setLongTermStorage(lts);
26         setCapacity(capacity);
27         setStorageRegulations(constraints);
28     }
29     /** Adding a new item/s to this locker
30      * @param item the type of item we are trying to add to this locker
31      * @param n the number of copies of the current item we are trying to add
32      * @return 0 if we succeed adding these items, 1 if we had to move items to lts,
33      * -2 if the insert wasn't successful because of constraints and -1 otherwise.
34      */
35     public int addItem(Item item, int n)
36     {
37         String currentType = item.getType();
38         //Check if this item is not breaking any constraints if let it in the locker
39         if(checkStorageRegulations(currentType))
40         {
41             String msgFormat = ERROR_PREFIX + ERROR_INSERT_CONTRADICTION;
42             String errorMessage = MessageFormat.format(msgFormat, currentType);
43             System.out.println(errorMessage);
44             return INSERT_ERROR_CODE_CONTRADICTION;
45         }
46         //Check if there enough room in the locker for all the new items
47         if(checkIfThereEnoughRoom(item, n) == false)
48         {
49             return INSERT_ERROR_CODE;
50         }
51         //Check % threshold
52         if(getItemOccupiedPercentage(item, n) > MOVING_TO_LTS_THRESHOLD)
53         {
54             return moveToLTS(item, n);
55         }
56         else
57         {
58             //Insert Items to inventory
59             addToInventory(item, n);
```

```

60     }
61
62     return SUCCESS;
63 }
64
65 /** Moving items to the LTS
66  * @param item the type of item we are trying to add to LTS
67  * @param n the total number of copies we currently have in the locker
68  * @return 0 if we succeed adding these items, 1 if we had to move items to lts,
69  * -2 if the insert wasn't successful because of constraints and -1 otherwise.
70  */
71 public int moveToLTS(Item item, int n)
72 {
73     int currentCount = getItemCount(item.getType());
74     int amountToMove = getAmountTOMoveToLTS(item, n + currentCount);
75     int returnCode = getLongTermStorage().addItem(item, amountToMove);
76     //Check the insert to LTS was successful
77     if(returnCode == SUCCESS)
78     {
79         if(amountToMove > n)
80         {
81             //remove the items from the locker after move
82             returnCode = removeItem(item, amountToMove - n);
83         }
84         else if(amountToMove < n)
85         {
86             //Insert Items to inventory
87             addToInventory(item, n - amountToMove);
88         }
89     }
90     //If the success flag is still on then export the warning
91     if(returnCode == SUCCESS)
92     {
93         System.out.println(WARNING_INSERT);
94         return INSERT_WARNING_CODE;
95     }
96
97     return returnCode;
98 }
99
100 /** get the amount we wish to move to the LTS
101  * @param item the type of item we are trying to move to LTS
102  * @param n the total number of copies we currently have in the locker
103  * @return the amount we wish to move to the LTS.
104  */
105 public int getAmountTOMoveToLTS(Item item, int n)
106 {
107     int amount = 1 , returnAmount = INITIAL_SIZE;
108     while(calcItemOccupiedPercentage(item, amount) <= KEEPING_IN_LOCKER_THRESHOLD
109         && returnAmount < n)
110     {
111         amount++;
112         returnAmount++;
113     }
114
115     return n - returnAmount;
116 }
117 }

```

## 8 LockerTest.java

```
1  import oop.ex3.spaceship.Item;
2  import org.junit.*;
3  import static org.junit.Assert.*;
4  import oop.ex3.spaceship.ItemFactory;
5  import java.text.MessageFormat;
6  import java.util.HashMap;
7  import java.util.Map;
8
9  /**
10   * This class is used test class Locker.
11   * @author Bar Melinarskiy
12   * @version 26/8/20
13   */
14  public class LockerTest
15  {
16      //Constants
17      protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
18      protected static final int INSERT_ERROR_CODE_CONTRADICTION = -2;
19      protected static final int ERROR_CODE = -1;
20      protected static final int INSERT_WARNING_CODE = 1;
21      protected static final int SUCCESS_CODE = 0;
22      protected static final int BAT_INDEX = 0;
23      protected static final int SMALL_HELMET_INDEX = 1;
24      protected static final int BIG_HELMET_INDEX = 2;
25      protected static final int ENGINE_INDEX = 3;
26      protected static final int FOOTBALL_INDEX = 4;
27      protected static final int INITIAL_SIZE = 0;
28
29      // instance variables
30      private static LongTermStorage longTermUnit = new LongTermStorage();
31      static private final int capacity = 100;
32      static private Locker testLocker;
33      static private final Item[] constraints = ItemFactory.getConstraintPairs();
34      static private final Item[] allItems = ItemFactory.createAllLegalItems();
35      //test methods
36      /** build testers objects
37       */
38      @BeforeClass
39      public static void setUp()
40      {
41          testLocker = new Locker(longTermUnit, capacity, constraints);
42      }
43      /** test method getLongTermStorage
44       */
45      @Test
46      public void getLongTermStorage()
47      {
48          assertNotNull(testLocker.getLongTermStorage());
49      }
50      /** test method removeItem
51       */
52      @Test
53      public void removeItem()
54      {
55          removeNegative();
56          regularRemove();
57          nonExistingCountRemove();
58      }
59      /** test method getInventory
```

```

60     */
61     @Test
62     public void getInventory()
63     {
64         final String method = "getInventory";
65         String errorMessage = getErrorPrefix(method);
66         assertNotNull(errorMessage, testLocker.getInventory());
67     }
68     /** test method checkCapacity
69     */
70     @Test
71     public void checkCapacity()
72     {
73         final String method = "checkCapacity";
74         String errorMessage = getErrorPrefix(method);
75         assertEquals(errorMessage, capacity, testLocker.getCapacity());
76     }
77     /** test method addItem
78     */
79     @Test
80     public void addItem()
81     {
82         addItemOverflow();
83         addMoreThanHalfCapacityAtOnce();
84         afterInsertMoreThanHalfCapacity();
85         checkFillingUpLTS();
86         regularInsert();
87         insertNegative();
88         checkConstraints();
89         initLocker();
90     }
91     /** Check count of an item in locker
92     * @param typeIndex index of item to check
93     * @param count count to check
94     * @param source calling test name
95     */
96     private void checkCount(int typeIndex, int count, String source)
97     {
98         final String method = source + " - checking count of item";
99         String errorMessage = getErrorPrefix(method);
100
101         assertEquals(errorMessage, count,
102             testLocker.getItemCount(allItems[typeIndex].getType()));
103         if(count > INITIAL_SIZE)
104         {
105             assertTrue(errorMessage,
106                 testLocker.getInventory().get(allItems[typeIndex].getType()) == count);
107         }
108         else
109         {
110             assertNull(errorMessage,
111                 testLocker.getInventory().get(allItems[typeIndex].getType()));
112         }
113     }
114
115     /** Try add at once more than the capacity
116     */
117     private void addItemOverflow()
118     {
119         final String method = "addItemOverflow";
120         String errorMessage = getErrorPrefix(method);
121         assertEquals(errorMessage,
122             ERROR_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 20));
123     }
124     /** insert more then 50% at once - at least 5 engines are supposed to move to the LTS
125     */
126     private void addMoreThanHalfCapacityAtOnce()
127     {

```

```

128     final String method = "addMoreThanHalfCapacityAtOnce";
129     String errorMessage = getErrorPrefix(method);
130     assertEquals(errorMessage,
131         INSERT_WARNING_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 7));
132     checkRightAmountAfterMoveToLTS();
133 }
134
135 /** after insert more than 50% - at least 5 engines are supposed to move to the LTS
136  */
137 private void afterInsertMoreThanHalfCapacity()
138 {
139     final String method = "afterInsertMoreThanHalfCapacity";
140     String errorMessage = getErrorPrefix(method);
141     assertEquals(errorMessage,
142         INSERT_WARNING_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 4));
143     checkRightAmountAfterMoveToLTS();
144 }
145 /** check locker inventory after moving items to LTS
146  */
147 private void checkRightAmountAfterMoveToLTS()
148 {
149     final String method = "afterInsertMoreThanHalfCapacity";
150     String errorMessage = getErrorPrefix(method);
151
152     assertTrue(errorMessage, testLocker.getAvailableCapacity() >= 80);
153     assertTrue(errorMessage, testLocker.getItemCount(allItems[ENGINE_INDEX].getType()) <= 2);
154     assertTrue(errorMessage,
155         testLocker.getInventory().get(allItems[ENGINE_INDEX].getType()) <= 2);
156     assertTrue(errorMessage,
157         testLocker.getInventory().containsKey(allItems[ENGINE_INDEX].getType()));
158     checkCapacity();
159 }
160 /** check successful insert
161  */
162 private void regularInsert()
163 {
164     final String method = "regularInsert";
165     String errorMessage = getErrorPrefix(method);
166
167     assertEquals(errorMessage,
168         SUCCESS_CODE, testLocker.addItem(allItems[SMALL_HELMET_INDEX], 1));
169     checkCount(SMALL_HELMET_INDEX, 1, method);
170 }
171 /** check locker add item follows constraints
172  */
173 private void checkConstraints()
174 {
175     final String method = "checkConstraints";
176     String errorMessage = getErrorPrefix(method);
177     //Check first element in pair
178     assertEquals(errorMessage,
179         SUCCESS_CODE, testLocker.addItem(allItems[FOOTBALL_INDEX], 1));
180     checkCount(FOOTBALL_INDEX, 1, method);
181     assertEquals(errorMessage,
182         INSERT_ERROR_CODE_CONTRADICTION, testLocker.addItem(allItems[BAT_INDEX], 1));
183     checkCount(BAT_INDEX, 0, method);
184     assertEquals(errorMessage,
185         SUCCESS_CODE, testLocker.removeItem(allItems[FOOTBALL_INDEX], 1));
186     checkCount(FOOTBALL_INDEX, 0, method);
187     //Check second element in pair
188     assertEquals(errorMessage,
189         SUCCESS_CODE, testLocker.addItem(allItems[BAT_INDEX], 1));
190     checkCount(BAT_INDEX, 1, method);
191     assertEquals(errorMessage,
192         INSERT_ERROR_CODE_CONTRADICTION, testLocker.addItem(allItems[FOOTBALL_INDEX], 1));
193     checkCount(FOOTBALL_INDEX, 0, method);
194 }
195 /** check insert of negative amount

```



```

196     */
197 private void insertNegative()
198 {
199     final String method = "insertNegative";
200     String errorMessage = getErrorPrefix(method);
201     //insert non-existing item with negative amount
202     assertEquals(errorMessage,
203         ERROR_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], -2));
204     checkCount(BIG_HELMET_INDEX, 0, method);
205     //insert existing item with negative amount
206     assertEquals(errorMessage,
207         SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 2));
208     assertEquals(errorMessage,
209         ERROR_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], -2));
210     checkCount(BIG_HELMET_INDEX, 2, method);
211 }
212 /** check insert fails when LTS is full
213     */
214 private void checkFillingUpLTS()
215 {
216     final String method = "checkFillingUpLTS";
217     String errorMessage = getErrorPrefix(method);
218
219     int i = 0;
220     removeAllCopiesOfItem(allItems[ENGINE_INDEX].getType());
221     while(testLocker.getLongTermStorage().getAvailableCapacity() > 50 && i < 20)
222     {
223         addMoreThanHalfCapacityAtOnce();
224         i++;
225     }
226
227     assertTrue(errorMessage,
228         testLocker.getLongTermStorage().getAvailableCapacity() < 50);
229     assertEquals(errorMessage, ERROR_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 5));
230     removeAllCopiesOfItem(allItems[ENGINE_INDEX].getType());
231 }
232 /** init the locker btw tests
233     */
234 private void initLocker()
235 {
236     //Make copy of inventory and delete it all from the locker
237     Map<String, Integer> currentInventory = new HashMap<String, Integer>(testLocker.getInventory());
238     for(Map.Entry<String, Integer> entry : currentInventory.entrySet())
239     {
240         removeAllCopiesOfItem(entry.getKey());
241     }
242     //Check that now the inventory id indeed empty
243     assertEquals(0, testLocker.getInventory().size());
244 }
245 /** delete item from locker
246     */
247 private void removeAllCopiesOfItem(String type)
248 {
249     final String method = "checkFillingUpLTS";
250     String errorMessage = getErrorPrefix(method);
251
252     int count = testLocker.getItemCount(type);
253     Item itemToDelete = ItemFactory.createSingleItem(type);
254     assertEquals(errorMessage, SUCCESS_CODE, testLocker.removeItem(itemToDelete, count));
255 }
256 /** try removing negative amount
257     */
258 private void removeNegative()
259 {
260     final String method = "removeNegative";
261     String errorMessage = getErrorPrefix(method);
262
263     //Remove existing item with negative amount

```

```

264         assertEquals(errorMessage,
265             ERROR_CODE, testLocker.removeItem(allItems[FOOTBALL_INDEX], -2));
266         //Remove non existing item with negative amount
267         assertEquals(errorMessage,
268             ERROR_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], -2));
269     }
270     /** try removing successfully
271     */
272     private void regularRemove()
273     {
274         final String method = "regularRemove";
275         String errorMessage = getErrorPrefix(method);
276         assertEquals(errorMessage,
277             SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 1));
278         assertEquals(errorMessage,
279             SUCCESS_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], 1));
280         checkCount(BIG_HELMET_INDEX, 0, method);
281     }
282     /** try removing over the real amount
283     */
284     private void nonExistingCountRemove()
285     {
286         final String method = "regularRemove";
287         String errorMessage = getErrorPrefix(method);
288         assertEquals(errorMessage,
289             SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 1));
290         assertEquals(errorMessage,
291             ERROR_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], 10));
292     }
293
294     /** Get error msg
295     * @param method param to add to msg
296     * @return error msg
297     */
298     private String getErrorPrefix(String method)
299     {
300         String msgFormat = ERROR_PREFIX;
301         return MessageFormat.format(msgFormat, method);
302     }
303
304 }

```

## 9 LongTermStorage.java

```
1  import oop.ex3.spaceship.Item;
2  /**
3   * This class represents LTS inside the spaceship
4   * @author Bar Melinarskiy
5   * @version 16/8/20
6   */
7  public class LongTermStorage extends BasicStorage
8  {
9      //Constants
10     int LTS_CAPACITY = 1000;
11
12     /** Contractor
13      */
14     public LongTermStorage()
15     {
16         setCapacity(LTS_CAPACITY);
17     }
18     /** Adding a new item/s to this lts
19      * @param item the type of item we are trying to add to this locker
20      * @param n the number of copies of the current item we are trying to add
21      * @return 0 if we succeed adding these items, -1 otherwise.
22      */
23     public int addItem(Item item, int n)
24     {
25         //Check if there enough room in the locker for all the new items
26         if(checkIfThereEnoughRoom(item, n) == false)
27         {
28             return INSERT_ERROR_CODE;
29         }
30         //Insert Items to inventory
31         addToInventory(item, n);
32         return SUCCESS;
33     }
34     /** reset the lts storage
35      */
36     public void resetInventory()
37     {
38         initializeStorage();
39     }
40 }
```

## 10 LongTermTest.java

```
1  import oop.ex3.spaceship.Item;
2  import org.junit.*;
3  import static org.junit.Assert.*;
4  import oop.ex3.spaceship.ItemFactory;
5
6  import java.text.MessageFormat;
7
8  /**
9   * This class is used test class LongTermStorage.
10   * @author Bar MelinarSKIY
11   * @version 26/8/20
12   */
13  public class LongTermTest
14  {
15      //Constants
16      protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
17      protected static final int INITIAL_SIZE = 0;
18      protected static final int SUCCESS_CODE = 0;
19      protected static final int ERROR_CODE = -1;
20      protected static final int BAT_INDEX = 0;
21      protected static final int SMALL_HELMET_INDEX = 1;
22      protected static final int BIG_HELMET_INDEX = 2;
23      protected static final int ENGINE_INDEX = 3;
24      protected static final int FOOTBALL_INDEX = 4;
25
26      // instance variables
27      static private final int capacity = 1000;
28      private static LongTermStorage ltsTest;
29      static private final Item[] allItems = ItemFactory.createAllLegalItems();
30
31      // tests methods
32      /** build testers objects
33       */
34      @BeforeClass
35      public static void setUp()
36      {
37          ltsTest = new LongTermStorage();
38      }
39
40
41      /** check method removeItem
42       */
43      @Test
44      public void removeItem()
45      {
46          removeNegative();
47          regularRemove();
48          nonExistingCountRemove();
49      }
50      /** check method getInventory
51       */
52      @Test
53      public void getInventory()
54      {
55          final String method = "getInventory";
56          String errorMessage = getErrorPrefix(method);
57          assertNotNull(errorMessage, ltsTest.getInventory());
58      }
59      /** check method getCapacity
```

```

60     */
61     @Test
62     public void getCapacity()
63     {
64         final String method = "getCapacity";
65         String errorMessage = getErrorPrefix(method);
66         assertEquals(errorMessage, capacity, ltsTest.getCapacity());
67     }
68     /** check method addItem
69     */
70     @Test
71     public void addItem()
72     {
73         addItemOverflow();
74         regularInsert();
75         insertNegative();
76         checkConstraints();
77     }
78
79     /** check method resetInventory
80     */
81     @Test
82     public void resetInventory()
83     {
84         final String method = "resetInventory";
85         String errorMessage = getErrorPrefix(method);
86
87         ltsTest.addItem(allItems[SMALL_HELMET_INDEX], 1);
88         assertTrue(errorMessage, ltsTest.getInventory().size() > 0);
89         ltsTest.resetInventory();
90         //Check that now the inventory is indeed empty
91         assertEquals(errorMessage, 0, ltsTest.getInventory().size());
92     }
93
94     /** Try add at once more than the capacity
95     */
96     private void addItemOverflow()
97     {
98         final String method = "addItemOverflow";
99         String errorMessage = getErrorPrefix(method);
100         assertEquals(errorMessage, ERROR_CODE, ltsTest.addItem(allItems[ENGINE_INDEX], 2000));
101     }
102
103     /** Try add successfully
104     */
105     private void regularInsert()
106     {
107         final String method = "regularInsert";
108         String errorMessage = getErrorPrefix(method);
109         assertEquals(errorMessage,
110             SUCCESS_CODE, ltsTest.addItem(allItems[SMALL_HELMET_INDEX], 1));
111         checkCount(SMALL_HELMET_INDEX, 1, method);
112     }
113
114     /** Check count of an item in lts
115     * @param typeIndex index of item to check
116     * @param count count to check
117     * @param source calling test name
118     */
119     private void checkCount(int typeIndex, int count, String source)
120     {
121         final String method = source + " - checking count of item";
122         String errorMessage = getErrorPrefix(method);
123
124         assertEquals(errorMessage, count, ltsTest.getItemCount(allItems[typeIndex].getType()));
125         if(count > INITIAL_SIZE)
126         {
127             assertTrue(errorMessage,

```

```

128         ltsTest.getInventory().get(allItems[typeIndex].getType()) == count);
129     }
130     else
131     {
132         assertNull(errorMessage, ltsTest.getInventory().get(allItems[typeIndex].getType()));
133     }
134 }
135 /** Try add negative amount
136  */
137 private void insertNegative()
138 {
139     final String method = "insertNegative";
140     String errorMessage = getErrorPrefix(method);
141
142     //insert non-existing item with negative amount
143     assertEquals(errorMessage,
144         ERROR_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], -2));
145     checkCount(BIG_HELMET_INDEX, 0, method);
146     //insert existing item with negative amount
147     assertEquals(errorMessage,
148         SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 2));
149     assertEquals(errorMessage,
150         ERROR_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], -2));
151     checkCount(BIG_HELMET_INDEX, 2, method);
152 }
153 /** Check LTS has no constraints
154  */
155 private void checkConstraints()
156 {
157     final String method = "checkConstraints";
158     String errorMessage = getErrorPrefix(method);
159
160     //Check first element in pair
161     assertEquals(errorMessage,
162         SUCCESS_CODE, ltsTest.addItem(allItems[FOOTBALL_INDEX], 1));
163     checkCount(FOOTBALL_INDEX, 1, method);
164     assertEquals(errorMessage,
165         SUCCESS_CODE, ltsTest.addItem(allItems[BAT_INDEX], 1));
166     checkCount(BAT_INDEX, 1, method);
167     ltsTest.resetInventory();
168     //Check second element in pair
169     assertEquals(errorMessage,
170         SUCCESS_CODE, ltsTest.addItem(allItems[BAT_INDEX], 1));
171     checkCount(BAT_INDEX, 1, method);
172     assertEquals(errorMessage,
173         SUCCESS_CODE, ltsTest.addItem(allItems[FOOTBALL_INDEX], 1));
174     checkCount(FOOTBALL_INDEX, 1, method);
175 }
176 /** try remove negative amount
177  */
178 private void removeNegative()
179 {
180     final String method = "removeNegative";
181     String errorMessage = getErrorPrefix(method);
182
183     //Remove existing item with negative amount
184     assertEquals(errorMessage,
185         ERROR_CODE, ltsTest.removeItem(allItems[FOOTBALL_INDEX], -2));
186     //Remove non existing item with negative amount
187     assertEquals(errorMessage,
188         ERROR_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], -2));
189 }
190 /** try remove successfully
191  */
192 private void regularRemove()
193 {
194     final String method = "regularRemove";
195     String errorMessage = getErrorPrefix(method);

```

```

196
197         assertEquals(errorMessage,
198             SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 1));
199         assertEquals(errorMessage,
200             SUCCESS_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], 1));
201         checkCount(BIG_HELMET_INDEX, 0, method);
202     }
203     /** try remove over the right amount
204     */
205     private void nonExistingCountRemove()
206     {
207         final String method = "regularRemove";
208         String errorMessage = getErrorPrefix(method);
209         assertEquals(errorMessage,
210             SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 1));
211         assertEquals(errorMessage,
212             ERROR_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], 10));
213     }
214
215     /** Get error msg
216     * @param method param to add to
217     * @return error msg
218     */
219     private String getErrorPrefix(String method)
220     {
221         String msgFormat = ERROR_PREFIX;
222         return MessageFormat.format(msgFormat, method);
223     }
224 }

```

# 11 Spaceship.java

```
1  import oop.ex3.spaceship.Item;
2  /**
3   * This class represents a spaceship
4   * @author Bar Melinarskiy
5   * @version 16/8/20
6   */
7  public class Spaceship
8  {
9      //Constants
10     private static final int ZERO = 0;
11     protected static final int ERROR_NOT_VALID_ID = -1;
12     protected static final int ERROR_NOT_VALID_CAPACITY = -2;
13     protected static final int ERROR_LOCKERS_OVERFLOW = -3;
14     protected static final int SUCCESS_CODE = 0;
15     // instance variables
16     private String shipName;
17     private LongTermStorage longTermUnit = new LongTermStorage();
18     private Locker[] lockers;
19     private int lockerCounter = ZERO;
20     private int maxNumOfLockers;
21     private int[] shipsCrew;
22     private Item[] [] storageRegulations;
23
24     /** Contractor
25      * @param name the current ship's name.
26      * @param crewIDs array of crew's ID numbers.
27      * @param numOfLockers the capacity of lockers this ship can have.
28      * @param constraints an array of pairs of two items that are NOT allowed to reside
29      * together in a locker.
30      */
31     public Spaceship(String name, int[] crewIDs, int numOfLockers, Item[] [] constraints)
32     {
33         shipName = name;
34         shipsCrew = crewIDs;
35         maxNumOfLockers = numOfLockers;
36         lockers = new Locker[numOfLockers];
37         storageRegulations = constraints;
38     }
39     /** Get the long term storage unit object
40      * @return the long-term storage object associated with that Spaceship.
41      */
42     public LongTermStorage getLongTermStorage()
43     {
44         return longTermUnit;
45     }
46     /** creates a new Locker object inside the spaceship
47      * @param crewID the locker's owner's ID.
48      * @param capacity the capacity the new locker could hold.
49      */
50     public int createLocker(int crewID, int capacity)
51     {
52         //Check this is a real crew member
53         if(checkID(crewID) == false)
54         {
55             return ERROR_NOT_VALID_ID;
56         }
57
58         if(capacity < ZERO)
59         {
```



```

60         return ERROR_NOT_VALID_CAPACITY;
61     }
62
63     if(lockerCounter == maxNumOfLockers)
64     {
65         return ERROR_LOCKERS_OVERFLOW;
66     }
67
68     lockers[lockerCounter] = new Locker(getLongTermStorage(), capacity, storageRegulations);
69     lockerCounter++;
70     return SUCCESS_CODE;
71 }
72 /** Get the crew's ids.
73 * @return an array of the crew's IDs numbers
74 */
75 public int[] getCrewIDs()
76 {
77     return shipsCrew;
78 }
79 /** Get ship's lockers.
80 * @return an array of the ship's lockers
81 */
82 public Locker[] getLockers()
83 {
84     return lockers;
85 }
86
87 /** Check given id
88 * @param id the id to check
89 * @return true if the id is valid, false otherwise.
90 */
91 public boolean checkID(int id)
92 {
93     for(int i = 0; i < shipsCrew.length; i++)
94     {
95         if(shipsCrew[i] == id)
96         {
97             return true;
98         }
99     }
100     return false;
101 }
102 }

```

## 12 SpaceshipDepositoryTest.java

```
1  import org.junit.runner.RunWith;
2  import org.junit.runners.Suite;
3
4  @RunWith(Suite.class)
5
6  @Suite.SuiteClasses({
7      LockerTest.class,
8      LongTermTest.class,
9      SpaceshipTest.class
10 })
11
12 /**
13  * This class is used to run full tests on Spaceship.class, LockerTest.class
14  * and LongTermTest.class
15  * @author Bar Melinarskiy
16  * @version 16/8/20
17  */
18 public class SpaceshipDepositoryTest
19 {
20 }
```

## 13 SpaceshipTest.java

```
1  import oop.ex3.spaceship.Item;
2  import org.junit.*;
3  import static org.junit.Assert.*;
4  import oop.ex3.spaceship.ItemFactory;
5  import java.text.MessageFormat;
6  /**
7   * This class is used test class Spaceship.
8   * @author Bar Melinarskiy
9   * @version 26/8/20
10  */
11  public class SpaceshipTest {
12      //Constants
13      protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
14      protected static final int ERROR_NOT_VALID_ID = -1;
15      protected static final int ERROR_NOT_VALID_CAPACITY = -2;
16      protected static final int ERROR_LOCKERS_OVERFLOW = -3;
17      protected static final int SUCCESS_CODE = 0;
18
19      // instance variables
20      static private Spaceship testSpaceShip;
21      static final private Item[] constraints = ItemFactory.getConstraintPairs();
22      static final private int[] crewIDs = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
23      static final private int numOfLockers = 3;
24      //test methods
25
26      /** build testers objects
27       */
28      @BeforeClass
29      public static void createTestObjects()
30      {
31          testSpaceShip = new Spaceship("test", crewIDs, numOfLockers, constraints);
32      }
33      /** test method getLongTermStorage
34       */
35      @Test
36      public void getLongTermStorage()
37      {
38          final String method = "getLongTermStorage";
39          String errorMessage = getErrorPrefix(method);
40          assertNotNull(errorMessage, testSpaceShip.getLongTermStorage());
41      }
42      /** test method createLocker
43       */
44      @Test
45      public void createLocker()
46      {
47          final String method = "createLocker";
48          String errorMessage = getErrorPrefix(method);
49          //wrong id
50          assertEquals(errorMessage,
51              ERROR_NOT_VALID_ID, testSpaceShip.createLocker(20, 10));
52          //Non valid capacity
53          assertEquals(errorMessage,
54              ERROR_NOT_VALID_CAPACITY, testSpaceShip.createLocker(1, -4));
55          for(int i = 0 ; i <= numOfLockers; i++)
56          {
57              if(i < numOfLockers)
58              {
59                  //Valid new locker
```

```

60         assertEquals(errorMessage,
61             SUCCESS_CODE, testSpaceShip.createLocker(i + 1, 100));
62         //Check the creation was really successful
63         assertNotNull(errorMessage, testSpaceShip.getLockers()[i]);
64     }
65     else
66     {
67         //One too many locker, check if we get error code for overflow
68         assertEquals(errorMessage,
69             ERROR_LOCKERS_OVERFLOW, testSpaceShip.createLocker(i + 1, 100));
70     }
71 }
72
73 }
74 /** test method getCrewIDs
75  */
76 @Test
77 public void getCrewIDs()
78 {
79     final String method = "getCrewIDs";
80     String errorMessage = getErrorPrefix(method);
81
82     int[] crewArray = testSpaceShip.getCrewIDs();
83     assertNotNull(errorMessage, crewArray);
84     //Check valid length of array
85     assertEquals(errorMessage, crewIDs.length, crewArray.length);
86     //Check all the IDs are there
87     assertEquals(errorMessage, crewArray, crewIDs);
88 }
89 /** test method getLockers
90  */
91 @Test
92 public void getLockers()
93 {
94     final String method = "getCrewIDs";
95     String errorMessage = getErrorPrefix(method);
96
97     Locker[] lockersArray = testSpaceShip.getLockers();
98     assertNotNull(errorMessage, lockersArray);
99     //Check valid length of array
100    assertEquals(errorMessage, numOfLockers, lockersArray.length);
101 }
102
103 /** Get error msg
104  * @param method param to add to msg
105  * @return error msg
106  */
107 private String getErrorPrefix(String method)
108 {
109     String msgFormat = ERROR_PREFIX;
110     return MessageFormat.format(msgFormat, method);
111 }
112 }

```

## 14 StorageWithLTS.java

```
1  import oop.ex3.spaceship.Item;
2  /**
3   * This class is an abstract class for all the methods related to an object
4   * who is connected to LTS and as such has regulations to follow
5   * @author Bar Melinarskiy
6   * @version 16/8/20
7   */
8  public abstract class StorageWithLTS extends BasicStorage
9  {
10     //Constants
11     protected static final int PAIR_FIRST_ELEMENT = 0;
12     protected static final int PAIR_SECOND_ELEMENT = 1;
13     protected static final int INSERT_WARNING_CODE = 1;
14
15     // instance variables
16     private static LongTermStorage longTermUnit;
17     private static Item[] [] storageRegulations;
18
19     // instance methods
20     /** Get the long term storage unit object
21      * @return the long-term storage object associated with that Spaceship.
22      */
23     public LongTermStorage getLongTermStorage()
24     {
25         return longTermUnit;
26     }
27
28     /** Set the long term storage unit object
29      * @param lts the current ship's Long-Term storage unit.
30      */
31     public void setLongTermStorage(LongTermStorage lts)
32     {
33         longTermUnit = lts;
34     }
35
36     /** Set locker's storage regulations
37      * @param constraints an array of pairs of two items that are NOT allowed to reside
38      * together in a locker.
39      */
40     protected void setStorageRegulations(Item[] [] constraints)
41     {
42         storageRegulations = constraints;
43     }
44
45     /** Check storage regulations
46      * @param type the type of item we wanna check
47      * @return true if this item can not be stores inside this locker because of constraints,
48      * false otherwise.
49      */
50     protected boolean checkStorageRegulations(String type)
51     {
52         Item firstItem, secondItem;
53         String firstType, secondType;
54         //Loop through the given pairs
55         for(int row = 0; row < storageRegulations.length; row++)
56         {
57             //Get the current pairs' info
58             firstItem = storageRegulations[row][PAIR_FIRST_ELEMENT];
59             firstType = firstItem.getType();
```

```

60     secondItem = storageRegulations[row][PAIR_SECOND_ELEMENT];
61     secondType = secondItem.getType();
62     //Check if the current first element is eq to the given type
63     if(firstType.equals(type) && getItemCount(secondType) > INITIAL_SIZE)
64     {
65         return true;
66     }
67     //Check if the current second element is eq to the given type
68     else if(secondType.equals(type)&& getItemCount(firstType) > INITIAL_SIZE)
69     {
70         return true;
71     }
72 }
73
74 return false; //if we reached this point then there is no problem adding this item
75 }
76
77 }

```