Contents

1 README

```
bar246802




============================
=      File description     =
============================
Spaceship.java - This class represents a spaceship
Locker.java - This class represents Locker inside the spaceship
LongTermStorage.java - This class represents LTS inside the spaceship
BasicStorage.java - This class is an abstract class for basic features of a storage unit
BoopingSiteTest.java - This class is used test class BoopingSite
BoopingSite.java  - This class is part of the - Booping.com - a new hotel booking site.
HotelSortingByProximityComparator.java -  This class is used to sort given list of hotels by proximity.
HotelSortingByRatingsComparator.java - This class is used to sort given list of hotels by Ratings.
LockerTest.java - This class is used test class Locker
LongTermTest.java - This class is used test class LongTermStorage
SpaceshipDepositoryTest.java - This class is used to run full tests on Spaceship.class, LockerTest.class and LongTermTest.cl
SpaceshipTest.java - this class is used test class Spaceship
StorageWithLTS.java - an abstract class for all the methods related to an object
 who is connected to LTS and as such has regulations to follow



============================
=          Design           =
============================
I choose to use abstract classes such as StorageWithLTS and BasicStorage in order to reuse the features shared btw the objec
The StorageWithLTS extends the BasicStorage.
The locker extends inheritance from StorageWithLTS .
The LongTermStorage BasicStorage.
If I could I would have used interfaces with defualt implemntions of method - that way I could have divided the code more to




============================
=  Implementation details   =
============================
Regarding the hotels I implemented two Comparator classes: HotelSortingByProximityComparator and HotelSortingByRatingsCompar
I used arraylist to store the hotels inside the different method due to the ease we can sort it with the Collections.sort an

In the tests I used the approach of creating a test method for each respective method in the tested class - in same cases I
who were called from the current main method.

I used formatters to export the messages in all the classes - that way I could add parameters more easily.



============================
=    Answers to questions   =
============================

1) Locker - Q: How did you choose to store the information? Why did you prefer it to other methods?
A: I choose to store the storage information inside a HashMap - that way I could easily get the count of an existing item an
If I have chosen a simple array I would have to implement all these method myself.
2)LTS -  Q: How did you choose to store the information? How is it different from Locker.java?
A: regarding the inventory the answer is the same as the locker but here we only extend the BasicStorage abstract class so w
```

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

```java
import oop.ex3.spaceship.Item;
import java.text.MessageFormat;
import java.util.HashMap;
import java.util.Map;

/**
 * This class is an abstract class for basic features of a storage unit
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public abstract class BasicStorage
{
    //Constants
    protected static final String ERROR_PREFIX =
            "Error: Your request cannot be completed at this time. Problem: ";
    protected static final String ERROR_INSERT_CONTRADICTION =
            "the locker cannot contain items of type {0}, as it contains a contradicting item";
    protected static final String ERROR_INSERT_OVERFLOW =
            "no room for {0} items of type {1}";
    protected static final int INSERT_ERROR_CODE_CONTRADICTION = -2;
    protected static final int INSERT_ERROR_CODE = -1;
    protected static final int SUCCESS = 0;
    protected static final int INITIAL_SIZE = 0;
    protected static final int ONE_HUNDRED = 100;
    protected static final String ERROR_REMOVE_NON_EXISTING_AMOUNT =
            "the locker does not contain {0} items of type {1}";
    protected static final String ERROR_REMOVE_NEGATIVE_NUM =
            "cannot remove a negative number of items of type {0}";
    protected static final int ERROR_CODE = -1;


    // instance variables
    private int lockerCapacity = INITIAL_SIZE;
    private int occupiedSpace = INITIAL_SIZE;
    private Map<String, Integer> inventory = new HashMap<String, Integer>();

    // instance methods
    public abstract int addItem(Item item, int n);

    /** Removing item/s from this locker
     * @param item the type of item we are trying to remove from this locker
     * @param n the number of copies of the current item we are trying to remove
     * @return 0 if we succeed removing these items, -1 otherwise.
     */
    public int removeItem(Item item, int n)
    {
        String currentType = item.getType();
        if(n < INITIAL_SIZE)
        {
            String msgFormat = ERROR_PREFIX + ERROR_REMOVE_NEGATIVE_NUM;
            String errorMessage = MessageFormat.format(msgFormat, currentType);
            System.out.println(errorMessage);
            return ERROR_CODE;
        }
        else if(n > getItemCount(currentType))
        {
            String msgFormat = ERROR_PREFIX + ERROR_REMOVE_NON_EXISTING_AMOUNT;
            String errorMessage = MessageFormat.format(msgFormat, n, currentType);
            System.out.println(errorMessage);
            return ERROR_CODE;
        }
        else if(n > INITIAL_SIZE)
        {
            removeFromInventory(item, n);
        }
        return SUCCESS;
    }
    /** reset this storage
     */
    public void initializeStorage()
    {
        getInventory().clear();
        setOccupiedSpace(INITIAL_SIZE);
    }

    /** Get the number of copies stored in this locker of the given type
     * @param type the type of item we wanna check
     * @return the number of Items of type type the locker contains.
     */
    public int getItemCount(String type)
    {
        int itemCount = INITIAL_SIZE;
        //Check if the given type is actually stored inside this locker
        if(inventory.containsKey(type))
        {
            itemCount = inventory.get(type);
        }
        return itemCount;
    }
    /** Get the a map of all the item types contained in the locker
     * @return a map of all the item types contained in the locker,
     * and their respective quantities.
     */
    public Map<String, Integer> getInventory()
    {
        return inventory;
    }
    /** Get the long-term storage's total capacity
     * @return the long-term storage's total capacity.
     */
    public int getCapacity()
    {
        return lockerCapacity;
    }
    /** Get the long-term storage's available capacity
     * @return a the long-term storage's available capacity
     */
    public int getAvailableCapacity()
    {
        return lockerCapacity - occupiedSpace;
    }

    /** Set the long-term storage's total capacity
     * @param capacity the long-term storage's total capacity.
     */
    protected void setCapacity(int capacity)
    {
        lockerCapacity = capacity;
    }
    /** Set the long-term storage's available capacity
     * @param size  the long-term storage's available capacity
     */
    protected void setOccupiedSpace(int size)
    {
        occupiedSpace = size;
    }

```

```java
    /** Get the long-term storage's occupied capacity
     * @return the long-term storage's occupied capacity
     */
    protected int getOccupiedSpace()
    {
        return occupiedSpace;
    }

    /** Adding a new item/s to this locker's inventory
     * @param item the type of item we are trying to add to this locker
     * @param n the number of copies of the current item we are trying to add
     */
    protected void addToInventory(Item item, int n)
    {
        int newCount = n + getItemCount(item.getType());
        getInventory().put(item.getType(), newCount);
        //Update available capacity
        int volumeAddition = n * item.getVolume();
        setOccupiedSpace(getOccupiedSpace() + volumeAddition);
    }

    /** Remove a item/s from this locker's inventory
     * @param item the type of item we are trying to remove from this locker's inventory
     * @param n the number of copies of the current item we are trying to remove
     */
    protected void removeFromInventory(Item item, int n)
    {
        int newCount = n - getItemCount(item.getType());
        if(newCount > INITIAL_SIZE)
        {
            getInventory().put(item.getType(), newCount);
        }
        else
        {
            getInventory().remove(item.getType());
        }
        //Update available capacity
        int volumeDecreased = n * item.getVolume();
        setOccupiedSpace(getOccupiedSpace() - volumeDecreased);
    }

    /** Get given type storage units % of a the given Storage-Unit
     * @param item the item we wanna check
     * @param n the number of copies of the current item we are trying to add
     * @return the given type storage units % of a the given Storage-Unit
     */
    protected double getItemOccupiedPercentage(Item item, int n)
    {
        double percentage = INITIAL_SIZE;
        int itemCount = getItemCount(item.getType());
        if(itemCount > INITIAL_SIZE || n > INITIAL_SIZE) //check if this item is stored inside this locker
        {
            int totalVolume = item.getVolume() * itemCount + n * item.getVolume();
            percentage = (double)totalVolume / getCapacity() * ONE_HUNDRED;
        }

        return percentage;
    }
    /** Calc given type storage units % of a the given Storage-Unit
     * @param item the item we wanna check
     * @param n the number of copies of the current item
     * @return the expected given type storage units % of a the given Storage-Unit
     */
    protected double calcItemOccupiedPercentage(Item item, int n)
    {
        double percentage = INITIAL_SIZE;
        if(n > INITIAL_SIZE) //check if this item is stored inside this locker
        {
            int totalVolume = n * item.getVolume();
            percentage = (double)totalVolume / getCapacity() * ONE_HUNDRED;
        }

        return percentage;
    }
    /** Check if this locker has enough room for the given items
     * @param item the item we wanna check
     * @param n the number of copies of the current item we are trying to add
     * @return true if there is enough room, false otherwise.
     */
    protected boolean checkIfThereEnoughRoom(Item item, int n)
    {
        //Check if there enough room in the locker for all the new items
        int totalVolume = item.getVolume() * n;
        if(getAvailableCapacity() < totalVolume || n < INITIAL_SIZE)
        {
            String msgFormat = ERROR_PREFIX + ERROR_INSERT_OVERFLOW;
            String errorMessage = MessageFormat.format(msgFormat, n, item.getType());
            System.out.println(errorMessage);
            return false;
        }

        return true;
    }
}
```

3 BoopingSite.java

```java
import oop.ex3.searchengine.*;
import java.util.*;
/**
 * This class is part of the - Booping.com - a new hotel booking site.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class BoopingSite
{
    //Constants
    private static final int EMPTY = 0;
    // instance variables
    private String datasetName;

    /** Contractor
     * @param name the name of the dataset
     */
    public BoopingSite(String name)
    {
        datasetName = name;
    }
    /** Get hotels located in the given city sorted by ratings
     * @param city the city the check
     * @return an array of hotels located in the given city, sorted from
     * the highest star-rating to the lowest. Hotels that have the
     * same rating will be organized according to the alphabet order of
     * the hotel's (property) name. In case there are no hotels in the
     * given city, this method returns an empty array
     */
    public Hotel[] getHotelsInCityByRating(String city)
    {
        ArrayList<Hotel> hotelsToSort = getHotelsInCity(city);
        if(!hotelsToSort.isEmpty())
        {
            Collections.sort(hotelsToSort, new HotelSortingByRatingsComparator());
        }

        Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
        hotelsToSort.toArray(hotelsToReturn);
        return hotelsToReturn;
    }
    /** Get hotels sorted according to their Euclidean distance from the given geographic location
     * @param latitude geographic coordinate that specifies the north-south position of a point
     * on the Earth's surface. valid values are btw [-90,90] degrees.
     * @param longitude a geographic coordinate that specifies the east-west position of a point
     * on the Earth's surface. valid values are btw [-180,180] degrees.
     * @return an array of hotels, sorted according to their Euclidean distance from the given geographic
     * location, in ascending order. Hotels that are at the same distance from the given location are
     * organized according to the number of points-of-interest for which
     * they are close to (in a decreasing order).
     * In case of illegal input, this method returns an empty array.
     */
    public Hotel[] getHotelsByProximity(double latitude, double longitude)
    {
        ArrayList<Hotel> hotelsToSort = new ArrayList<Hotel>();
        if(checkCoordinates(latitude, longitude) == true)
        {
            hotelsToSort = getAllHotels();
            Collections.sort(hotelsToSort, new HotelSortingByProximityComparator(latitude, longitude));
        }

        Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
        hotelsToSort.toArray(hotelsToReturn);
        return hotelsToReturn;
    }
    /** Get hotels in given city sorted according to their Euclidean distance
     * from the given geographic location
     * @param city the city the check
     * @param latitude geographic coordinate that specifies the north-south position of a point
     * on the Earth's surface. valid values are btw [-90,90] degrees.
     * @param longitude a geographic coordinate that specifies the east-west position of a point
     * on the Earth's surface. valid values are btw [-180,180] degrees.
     * @return an array of hotels in the given city, sorted according to their Euclidean distance
     * from the given geographic location, in ascending order.
     * Hotels that are at the same distance from the given location
     * are organized according to the number of points-of-interest for
     * which they are close to (in a decreasing order).
     * In case of illegal input, this method returns an empty array.
     */
    public Hotel[] getHotelsInCityByProximity(String city, double latitude, double longitude)
    {
        ArrayList<Hotel> hotelsToSort = new ArrayList<Hotel>();
        if(checkCoordinates(latitude, longitude) == true)
        {
            hotelsToSort = getHotelsInCity(city);
            Collections.sort(hotelsToSort, new HotelSortingByProximityComparator(latitude, longitude));
        }

        Hotel[] hotelsToReturn = new Hotel[hotelsToSort.size()];
        hotelsToSort.toArray(hotelsToReturn);
        return hotelsToReturn;
    }

    /** Get all the hotels from the dataset in the given city
     * @param city - the city to filter with
     * @return list of all the hotels from the dataset in the given city
     */
    private ArrayList<Hotel> getHotelsInCity(String city)
    {
        ArrayList<Hotel> hotels = getAllHotels();
        ArrayList<Hotel> filteredList = new ArrayList<Hotel>();
        for (Hotel hotel : hotels)
        {
            if (hotel.getCity().equals(city))
            {
                filteredList.add(hotel);
            }
        }

        return filteredList;
    }

    /** Get all the hotels from the dataset
     * @return list of all the hotels from the dataset
     */
    private ArrayList<Hotel> getAllHotels()
    {
        Hotel[] hotels = HotelDataset.getHotels(datasetName);
        ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
        return hotelList;
    }
    /** Check if the given coordinates are valid
     * @param latitude geographic coordinate that specifies the north-south position of a point
     * on the Earth's surface. valid values are btw [-90,90] degrees.
     * @param longitude a geographic coordinate that specifies the east-west position of a point
     * on the Earth's surface. valid values are btw [-180,180] degrees.
     * @return true if the given coordinates are valid, false otherwise.
     */
    private boolean checkCoordinates(double latitude, double longitude)
    {
        final int LATITUDE_LOW = -90;
        final int LATITUDE_HIGH = 90;
        final int LONGITUDE_LOW = -180;
        final int LONGITUDE_HIGH = 180;

        if(latitude < LATITUDE_LOW || latitude > LATITUDE_HIGH || longitude < LONGITUDE_LOW ||
           longitude > LONGITUDE_HIGH)
        {
            return false;
        }

        return true;
    }
}
```

4 BoopingSiteTest.java

```java
import oop.ex3.searchengine.*;
import org.junit.*;
import java.text.MessageFormat;
import java.util.ArrayList;
import java.util.Arrays;
import static org.junit.Assert.*;

/**
 * This class is used test class BoopingSite.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class BoopingSiteTest
{
    //Constants
    protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
    private static final String BIG_DATASET = "hotels_dataset.txt";
    private static final String SMALL_DATASET_1 = "hotels_tst1.txt"; //all hotels in Manali
    private static final String SMALL_DATASET_2 = "hotels_tst2.txt"; //empty file
    private static final String NON_EXISTING_CITY = "#$#@$#@$#@$.nonExisting";
    private static final int EMPTY = 0;
    private static final int ZERO = 0;
    private static final Hotel[] emptyHotelArray = new Hotel[EMPTY];
    // instance variables
    private static ArrayList<Hotel> longHotelList;
    private static ArrayList<Hotel> shortHotelList1; //all hotels in Manali
    private static ArrayList<Hotel> shortHotelList2; //empty list

    private static ArrayList<String> citiesLongHotelList;
    private static ArrayList<String> citiesShortHotelList1; // Only Manali
    private static ArrayList<String> citiesShortHotelList2; //empty list

    private BoopingSite boopingSiteTest1 = new BoopingSite(BIG_DATASET);
    private BoopingSite boopingSiteTest2 = new BoopingSite(SMALL_DATASET_1);
    private BoopingSite boopingSiteTest3 = new BoopingSite(SMALL_DATASET_2);

    private static String firstLongCity;
    private static String firstShort1City;
    private static String firstShort2City;

    // tests methods
    /** build testers objects
     */
    @BeforeClass
    public static void setUp()
    {
        longHotelList = getAllHotels(BIG_DATASET);
        shortHotelList1 = getAllHotels(SMALL_DATASET_1);
        shortHotelList2 = getAllHotels(SMALL_DATASET_2);

        citiesLongHotelList = getAllCities(longHotelList);
        citiesShortHotelList1 = getAllCities(shortHotelList1);
        citiesShortHotelList2 = getAllCities(shortHotelList2);

        firstLongCity = citiesLongHotelList.get(ZERO);
        firstShort1City = citiesShortHotelList1.get(ZERO);
        firstShort2City = "";
    }

    /** check method getHotelsInCityByRating
     */
    @Test
    public void getHotelsInCityByRating()
    {
        final String method = "getHotelsInCityByRating";
        //Check method on the large dataset
        Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsInCityByRating(firstLongCity);
        ArrayList<Hotel> firstCityHotelsLong =  getAllHotelsInCities(longHotelList, firstLongCity);
        checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);
        checkRatingOrder(returnedHotelsLong, method);
        //Check method on the small dataset
        Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByRating(firstShort1City);
        ArrayList<Hotel> firstCityHotelsShort1 =  getAllHotelsInCities(shortHotelList1, firstShort1City);
        checkAllHotelsExist(returnedHotelsShort1, firstCityHotelsShort1, method);
        checkRatingOrder(returnedHotelsShort1, method);
        //Check method on the empty dataset
        Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsInCityByRating(firstShort2City);
        ArrayList<Hotel> firstCityHotelsShort2 =  getAllHotelsInCities(shortHotelList1, firstShort2City);
        checkAllHotelsExist(returnedHotelsShort2, firstCityHotelsShort2, method);
    }

    /** check method getHotelsByProximity
     */
    @Test
    public void getHotelsByProximity()
    {
        final String method = "getHotelsByProximity";
        checkCoordinatesByProximity(boopingSiteTest1);

        //Check method on the large dataset
        Hotel hotel = longHotelList.get(ZERO);
        Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsByProximity(
                hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsLong, longHotelList, method);
        checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the large dataset - another hotel
        hotel = longHotelList.get(8);
        returnedHotelsLong = boopingSiteTest1.getHotelsByProximity(
                hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsLong, longHotelList, method);
        checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the small dataset
        hotel = shortHotelList1.get(ZERO);
        Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsByProximity(
                hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsShort1, shortHotelList1, method);
        checkProximityOrder(returnedHotelsShort1, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the empty dataset
        Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsByProximity(
                hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsShort2, shortHotelList2, method);
    }

    /** check method getHotelsInCityByProximity
     */
    @Test
    public void getHotelsInCityByProximity()
    {
        final String method = "getHotelsInCityByProximity";
        checkCoordinatesInCityByProximity(boopingSiteTest1);

        //Check method on the large dataset
        Hotel hotel = longHotelList.get(ZERO);
        Hotel[] returnedHotelsLong = boopingSiteTest1.getHotelsInCityByProximity(firstLongCity,
                hotel.getLatitude(), hotel.getLongitude());
        ArrayList<Hotel> firstCityHotelsLong =  getAllHotelsInCities(longHotelList, firstLongCity);
        checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);
        checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the large dataset - another hotel
        hotel = longHotelList.get(8);
        returnedHotelsLong = boopingSiteTest1.getHotelsInCityByProximity(firstLongCity,
                                                                                 hotel.getLatitude(), hotel.getLongitude());
        firstCityHotelsLong =  getAllHotelsInCities(longHotelList, firstLongCity);
        checkAllHotelsExist(returnedHotelsLong, firstCityHotelsLong, method);
        checkProximityOrder(returnedHotelsLong, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the small dataset
        hotel = shortHotelList1.get(ZERO);
        Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByProximity(firstShort1City,
                hotel.getLatitude(), hotel.getLongitude());
        ArrayList<Hotel> firstCityHotelsShort1 =  getAllHotelsInCities(shortHotelList1, firstShort1City);
        checkAllHotelsExist(returnedHotelsShort1, firstCityHotelsShort1, method);
        checkProximityOrder(returnedHotelsShort1, hotel.getLatitude(), hotel.getLongitude(), method);
        //Check method on the empty dataset
        Hotel[] returnedHotelsShort2 = boopingSiteTest3.getHotelsInCityByProximity(firstShort2City,
                hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsShort2, shortHotelList2, method);
    }

    /** check method checkNonExistingCity
     */
    @Test
    public void checkNonExistingCity()
    {
        final String method = "checkNonExistingCity";
        Hotel hotel = shortHotelList1.get(ZERO);
        //Check method on the small dataset - getHotelsInCityByRating
        Hotel[] returnedHotelsShort1 = boopingSiteTest2.getHotelsInCityByRating(NON_EXISTING_CITY);
        checkAllHotelsExist(returnedHotelsShort1, shortHotelList2, method);

        //Check method on the small dataset - getHotelsInCityByProximity

        Hotel[] returnedHotelsShort2 = boopingSiteTest2.getHotelsInCityByProximity(NON_EXISTING_CITY,
                                                       hotel.getLatitude(), hotel.getLongitude());
        checkAllHotelsExist(returnedHotelsShort2, shortHotelList2, method);
    }

    /** check the hotels are order correctly by proximity
     * @param boopingSiteTest the boopingSite to test on
     */
    private void checkCoordinatesByProximity(BoopingSite boopingSiteTest)
    {
        final String method = "ByProximity - Checking non-valid coordinates";
        String errorMessage = getErrorPrefix(method);
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsByProximity(-900,-900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsByProximity(900,-900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsByProximity(-900,900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsByProximity(900,900));
    }
    /** check the hotels in the current city are order correctly by proximity
     * @param boopingSiteTest the boopingSite to test on
     */
    private void checkCoordinatesInCityByProximity(BoopingSite boopingSiteTest)
    {
        final String method = "InCityByProximity - Checking non-valid coordinates";
        String errorMessage = getErrorPrefix(method);

        String city = citiesLongHotelList.get(0);
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsInCityByProximity(city,-900,-900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsInCityByProximity(city,900,-900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsInCityByProximity(city,-900,900));
        assertArrayEquals(errorMessage, emptyHotelArray,
                          boopingSiteTest.getHotelsInCityByProximity(city,900,900));
    }

    /** Get all the hotels from the dataset
     * @param datasetName the dataset to fetch hotels from
     * @return list of all the hotels from the dataset
     */
    private static ArrayList<Hotel> getAllHotels(String datasetName)
    {
        Hotel[] hotels = HotelDataset.getHotels(datasetName);
        ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
        return hotelList;
    }

    /** Get all the cities in the given hotels list
     * @param hotels given hotels list to fetch cities from
     * @return list of all the cities
     */
    private static ArrayList<String> getAllCities(ArrayList<Hotel> hotels)
    {
        ArrayList<String> citiesList = new ArrayList<String>();
        for (Hotel hotel : hotels)
        {
            if (citiesList.contains(hotel.getCity()) == false)
            {
                citiesList.add(hotel.getCity());
            }
        }

        return citiesList;
    }

    /** Get all the hotels in a given city
     * @param hotels given hotels list to fetch from
     * @param city city to check
     * @return list of all the hotels in the given city
     */
    private ArrayList<Hotel> getAllHotelsInCities(ArrayList<Hotel> hotels, String city)
    {
        ArrayList<Hotel> hotelsInCityList = new ArrayList<Hotel>();
        for (Hotel hotel : hotels)
        {
            if (city.equals(hotel.getCity()) && hotelsInCityList.contains(hotel) == false)
            {
                hotelsInCityList.add(hotel);
            }
        }

        return hotelsInCityList;
    }

    /** Check all hotels exist
     * @param hotels given hotels list to check
     * @param hotelsSource hotels to check against
     * @param source calling test name
     */
    private void checkAllHotelsExist(Hotel[] hotels, ArrayList<Hotel> hotelsSource, String source)
    {
        final String method = source + " - checking all hotels exist";
        String errorMessage = getErrorPrefix(method);

        assertNotNull(errorMessage, hotels);
        ArrayList<Hotel> hotelList = new ArrayList<Hotel>(Arrays.asList(hotels));
        assertEquals(errorMessage, hotels.length, hotelsSource.size());
        for (Hotel hotel : hotelList)
        {
            boolean exist = false;
            for (Hotel hotelSource : hotelsSource)
            {
                if(hotelSource.getUniqueId().equals(hotel.getUniqueId()))
                {
                    exist = true;
                    break;
                }
            }
            assertTrue(errorMessage, exist);
        }
    }

    /** Check all hotels are ordered by ratings
     * @param hotels given hotels list to check
     * @param source calling test name
     */
    private void checkRatingOrder(Hotel[] hotels, String source)
    {
        final String method = source + " - checking hotels order by rating";
        String errorMessage = getErrorPrefix(method);

        assertNotNull(errorMessage, hotels);
        Hotel previousHotel = null;
        for (Hotel hotel : hotels)
        {
            if(previousHotel == null)
            {
                previousHotel = hotel;
            }
            else
            {
                boolean ratingsOrder = previousHotel.getStarRating() > hotel.getStarRating();
                boolean ratingsEq = previousHotel.getStarRating() == hotel.getStarRating();
                boolean nameOrder = previousHotel.getPropertyName().compareTo(
                        hotel.getPropertyName()) <= ZERO;
                assertTrue(errorMessage,  ratingsOrder || (ratingsEq && nameOrder));
            }
        }
    }
    /** Check all hotels are ordered by proximity
     * @param hotels given hotels list to check
     * @param latitude point latitude
     * @param longitude point longitude
     * @param source calling test name
     */
    private void checkProximityOrder(Hotel[] hotels, double latitude, double longitude, String source)
    {
        final String method = source + " - checking hotels order by proximity";
        String errorMessage = getErrorPrefix(method);

        assertNotNull(errorMessage, hotels);
        Hotel previousHotel = null;
        for (Hotel hotel : hotels)
        {
            if(previousHotel == null)
            {
                previousHotel = hotel;
            }
            else
            {
                double distancePrevious = calcDistance(latitude, longitude,
                        previousHotel.getLatitude(), previousHotel.getLongitude());
                double distanceCurrent = calcDistance(latitude, longitude,
                                                  hotel.getLatitude(), hotel.getLongitude());
                boolean distanceOrder = distancePrevious < distanceCurrent;
                boolean distanceEq = distancePrevious == distanceCurrent;
                boolean poiOrder = previousHotel.getNumPOI() >= hotel.getNumPOI();
                assertTrue(errorMessage,  distanceOrder || (distanceEq && poiOrder));
            }
        }
    }

    /**
     * Calc distance btw given point and the geographic location
     * @param  x1 first point longitude
     * @param  y1 first point latitude
     * @param  x2 second point longitude
     * @param  y2 second point latitude
     * @return distance btw given point and the geographic location
     */
    private double calcDistance(double x1, double y1, double x2, double y2)
    {
        final int TWO = 2;
        return Math.sqrt(Math.pow(x1 - x2, TWO) +
                         Math.pow(y1 - y2, TWO));
    }

    /** Get error msg
     * @param method param to add to msg
     * @return error msg
     */
    private String getErrorPrefix(String method)
    {
        String msgFormat = ERROR_PREFIX;
        return MessageFormat.format(msgFormat, method);
    }
}
```

5 HotelSortingByProximityComparator.java

```java
import oop.ex3.searchengine.Hotel;
import java.util.Comparator;
/**
 * This class is used to sort given list of hotels by proximity.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class HotelSortingByProximityComparator implements Comparator<Hotel>
{
    //Constants
    protected static final int EQ = 0;
    protected static final int GT = 1;
    protected static final int LT = -1;

    double latitudeCoordinate;
    double longitudeCoordinate;
    /**
     * Creates a Comparator instance.
     * @param latitude longitude of point.
     * @param longitude longitude of longitude.
     */
    HotelSortingByProximityComparator(double latitude, double longitude)
    {
        latitudeCoordinate = latitude;
        longitudeCoordinate = longitude;
    }
    /**
     * Compare btw two hotel by proximity & POI.
     * @param hotel1 first hotel.
     * @param hotel2 second hotel.
     * @return the value {@code 0} if {@code x == y};
     * a value less than {@code 0} if {@code x < y}; and
     * a value greater than {@code 0} if {@code x > y}
     */
    @Override
    public int compare(Hotel hotel1, Hotel hotel2)
    {
        // for comparison
        int distanceCompare = compareDistance(hotel1, hotel2);
        int poiCompare = LT;
        if(hotel2.getNumPOI() == hotel1.getNumPOI())
        {
            poiCompare = EQ;
        }
        else if(hotel2.getNumPOI() > hotel1.getNumPOI())
        {
            poiCompare = GT;
        }

        // 2-level comparison using if-else block
        if (distanceCompare == 0)
        {
            return ((poiCompare == 0) ? distanceCompare : poiCompare);
        } else
        {
            return distanceCompare;
        }
    }
    /**
```

```java
     * Compares two hotel by their Euclidean distance from the given geographic location
     * @param  hotel1 the first hotel to compare
     * @param  hotel2 the second hotel to compare
     * @return the value {@code 0} if {@code x == y};
     *         a value less than {@code 0} if {@code x < y}; and
     *         a value greater than {@code 0} if {@code x > y}
     */
    private int compareDistance(Hotel hotel1, Hotel hotel2)
    {
        double distance1 = calcDistance(hotel1.getLongitude(), hotel1.getLatitude());
        double distance2 = calcDistance(hotel2.getLongitude(), hotel2.getLatitude());
        return Double.compare(distance1, distance2);
    }
    /**
     * Calc distance btw given point and the geographic location
     * @param  x longitude
     * @param  y latitude
     * @return distance btw given point and the geographic location
     */
    private double calcDistance(double x, double y)
    {
        final int TWO = 2;
        return Math.sqrt(Math.pow(longitudeCoordinate - x, TWO) +
                                 Math.pow(latitudeCoordinate - y, TWO));
    }



}
```

6 HotelSortingByRatingsComparator.java

```java
import oop.ex3.searchengine.*;
import java.util.Comparator;
/**
 * This class is used to sort given list of hotels by Ratings.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class HotelSortingByRatingsComparator implements Comparator<Hotel>
{
    //Constants
    protected static final int EQ = 0;
    protected static final int GT = 1;
    protected static final int LT = -1;

    /**
     * Compare btw two hotel by ratings & name.
     * @param hotel1 first hotel.
     * @param hotel2 second hotel.
     * @return the value {@code 0} if {@code x == y};
     * a value less than {@code 0} if {@code x < y}; and
     * a value greater than {@code 0} if {@code x > y}
     */
    @Override
    public int compare(Hotel hotel1, Hotel hotel2)
    {
        // for comparison
        int ratingCompare = LT;
        if(hotel2.getStarRating() == hotel1.getStarRating())
        {
            ratingCompare = EQ;
        }
        else if(hotel2.getStarRating() > hotel1.getStarRating())
        {
            ratingCompare = GT;
        }
        int nameCompare = hotel1.getPropertyName().compareTo(hotel2.getPropertyName());

        // 2-level comparison using if-else block
        if (ratingCompare == 0)
        {
            return ((nameCompare == 0) ? ratingCompare : nameCompare);
        } else
        {
            return ratingCompare;
        }
    }
}
```

7 Locker.java

```java
import oop.ex3.spaceship.Item;

import java.text.MessageFormat;
/**
 * This class represents Locker inside the spaceship
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public class Locker extends StorageWithLTS
{
    //Constants
    private static final String WARNING_INSERT =
    "Warning: Action successful, but has caused items to be moved to storage";
    private static final int MOVING_TO_LTS_THRESHOLD = 50;
    private static final int KEEPING_IN_LOCKER_THRESHOLD = 20;

    /** Contractor
     * @param lts the current ship's Long-Term storage unit.
     * @param capacity the limit capacity of this locker.
     * @param constraints an array of pairs of two items that are NOT allowed to reside
     * together in a locker.
     */
    public Locker(LongTermStorage lts, int capacity, Item[][] constraints)
    {
        setLongTermStorage(lts);
        setCapacity(capacity);
        setStorageRegulations(constraints);
    }
    /** Adding a new item/s to this locker
     * @param item the type of item we are trying to add to this locker
     * @param n the number of copies of the current item we are trying to add
     * @return 0 if we succeed adding these items, 1 if we had to move items to lts,
     * -2 if the insert wasn't successful because of constraints and -1 otherwise.
     */
    public int addItem(Item item, int n)
    {
        String currentType = item.getType();
        //Check if this item is not breaking any constraints if let it in the locker
        if(checkStorageRegulations(currentType))
        {
            String msgFormat = ERROR_PREFIX + ERROR_INSERT_CONTRADICTION;
            String errorMessage = MessageFormat.format(msgFormat, currentType);
            System.out.println(errorMessage);
            return INSERT_ERROR_CODE_CONTRADICTION;
        }
        //Check if there enough room in the locker for all the new items
        if(checkIfThereEnoughRoom(item, n) == false)
        {
            return INSERT_ERROR_CODE;
        }
        //Check % threshold
        if(getItemOccupiedPercentage(item, n) > MOVING_TO_LTS_THRESHOLD)
        {
            return moveToLTS(item, n);
        }
        else
        {
            //Insert Items to inventory
            addToInventory(item, n);
        }

        return SUCCESS;
    }

    /** Moving items to the LTS
     * @param item the type of item we are trying to add to LTS
     * @param n the total number of copies we currently have in the locker
     * @return 0 if we succeed adding these items, 1 if we had to move items to lts,
     * -2 if the insert wasn't successful because of constraints and -1 otherwise.
     */
    public int moveToLTS(Item item, int n)
    {
        int currentCount = getItemCount(item.getType());
        int amountToMove = getAmountTOMoveToLTS(item, n + currentCount);
        int returnCode = getLongTermStorage().addItem(item, amountToMove);
        //Check the insert to LTS was successful
        if(returnCode == SUCCESS)
        {
            if(amountToMove > n)
            {
                //remove the items from the locker after move
                returnCode = removeItem(item, amountToMove - n);
            }
            else if(amountToMove < n)
            {
                //Insert Items to inventory
                addToInventory(item, n - amountToMove);
            }
        }
        //If the success flag is still on then export the warning
        if(returnCode == SUCCESS)
        {
            System.out.println(WARNING_INSERT);
            return INSERT_WARNING_CODE;
        }

        return returnCode;
    }

    /** get the amount we wish to move to the LTS
     * @param item the type of item we are trying to move to LTS
     * @param n the total number of copies we currently have in the locker
     * @return the amount we wish to move to the LTS.
     */
    public int getAmountTOMoveToLTS(Item item, int n)
    {
        int amount = 1 , returnAmount = INITIAL_SIZE;
        while(calcItemOccupiedPercentage(item, amount) <= KEEPING_IN_LOCKER_THRESHOLD
              && returnAmount < n)
        {
            amount++;
            returnAmount++;
        }

        return n - returnAmount;
    }
}
```

8 LockerTest.java

```java
import oop.ex3.spaceship.Item;
import org.junit.*;
import static org.junit.Assert.*;
import oop.ex3.spaceship.ItemFactory;
import java.text.MessageFormat;
import java.util.HashMap;
import java.util.Map;

/**
 * This class is used test class Locker.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class LockerTest
{
    //Constants
    protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
    protected static final int INSERT_ERROR_CODE_CONTRADICTION = -2;
    protected static final int ERROR_CODE = -1;
    protected static final int INSERT_WARNING_CODE = 1;
    protected static final int SUCCESS_CODE = 0;
    protected static final int BAT_INDEX = 0;
    protected static final int SMALL_HELMET_INDEX = 1;
    protected static final int BIG_HELMET_INDEX = 2;
    protected static final int ENGINE_INDEX = 3;
    protected static final int FOOTBALL_INDEX = 4;
    protected static final int INITIAL_SIZE = 0;

    // instance variables
    private static LongTermStorage longTermUnit = new LongTermStorage();
    static private final int capacity = 100;
    static private Locker testLocker;
    static private final Item[][] constraints = ItemFactory.getConstraintPairs();
    static private final Item[] allItems = ItemFactory.createAllLegalItems();
    //test methods
    /** build testers objects
     */
    @BeforeClass
    public static void setUp()
    {
        testLocker = new Locker(longTermUnit, capacity, constraints);
    }
    /** test method getLongTermStorage
     */
    @Test
    public void getLongTermStorage()
    {
        assertNotNull(testLocker.getLongTermStorage());
    }
    /** test method removeItem
     */
    @Test
    public void removeItem()
    {
        removeNegative();
        regularRemove();
        nonExistingCountRemove();
    }
    /** test method getInventory
```

```java
     */
    @Test
    public void getInventory()
    {
        final String method = "getInventory";
        String errorMessage = getErrorPrefix(method);
        assertNotNull(errorMessage, testLocker.getInventory());
    }
    /** test method checkCapacity
     */
    @Test
    public void checkCapacity()
    {
        final String method = "checkCapacity";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage, capacity, testLocker.getCapacity());
    }
    /** test method addItem
     */
    @Test
    public void addItem()
    {
        addItemOverflow();
        addMoreThanHalfCapacityAtOnce();
        afterInsertMoreThanHalfCapacity();
        checkFillingUpLTS();
        regularInsert();
        insertNegative();
        checkConstraints();
        initLocker();
    }
    /** Check count of an item in locker
     * @param typeIndex index of item to check
     * @param count count to check
     * @param source calling test name
     */
    private void checkCount(int typeIndex, int count, String source)
    {
        final String method = source + " - checking count of item";
        String errorMessage = getErrorPrefix(method);

        assertEquals(errorMessage, count,
                     testLocker.getItemCount(allItems[typeIndex].getType()));
        if(count > INITIAL_SIZE)
        {
            assertTrue(errorMessage,
                    testLocker.getInventory().get(allItems[typeIndex].getType()) == count);
        }
        else
        {
            assertNull(errorMessage,
                       testLocker.getInventory().get(allItems[typeIndex].getType()));
        }
    }

    /** Try add at once more than the capacity
     */
    private void addItemOverflow()
    {
        final String method = "addItemOverflow";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                     ERROR_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 20));
    }
    /** insert more then 50% at once - at least 5 engines are supposed to move to the LTS
     */
    private void addMoreThanHalfCapacityAtOnce()
    {
        final String method = "addMoreThanHalfCapacityAtOnce";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                     INSERT_WARNING_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 7));
        checkRightAmountAfterMoveToLTS();
    }

    /** after insert more than 50% - at least 5 engines are supposed to move to the LTS
     */
    private void afterInsertMoreThanHalfCapacity()
    {
        final String method = "afterInsertMoreThanHalfCapacity";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                INSERT_WARNING_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 4));
        checkRightAmountAfterMoveToLTS();
    }
    /** check locker inventory after moving items to LTS
     */
    private void checkRightAmountAfterMoveToLTS()
    {
        final String method = "afterInsertMoreThanHalfCapacity";
        String errorMessage = getErrorPrefix(method);

        assertTrue(errorMessage,testLocker.getAvailableCapacity() >= 80);
        assertTrue(errorMessage,testLocker.getItemCount(allItems[ENGINE_INDEX].getType()) <= 2);
        assertTrue(errorMessage,
                   testLocker.getInventory().get(allItems[ENGINE_INDEX].getType()) <= 2);
        assertTrue(errorMessage,
                testLocker.getInventory().containsKey(allItems[ENGINE_INDEX].getType()));
        checkCapacity();
    }
    /** check successful insert
     */
    private void regularInsert()
    {
        final String method = "regularInsert";
        String errorMessage = getErrorPrefix(method);

        assertEquals(errorMessage,
                     SUCCESS_CODE, testLocker.addItem(allItems[SMALL_HELMET_INDEX], 1));
        checkCount(SMALL_HELMET_INDEX, 1, method);
    }
    /** check locker add item follows constraints
     */
    private void checkConstraints()
    {
        final String method = "checkConstraints";
        String errorMessage = getErrorPrefix(method);
        //Check first element in pair
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.addItem(allItems[FOOTBALL_INDEX], 1));
        checkCount(FOOTBALL_INDEX, 1, method);
        assertEquals(errorMessage,
                INSERT_ERROR_CODE_CONTRADICTION, testLocker.addItem(allItems[BAT_INDEX], 1));
        checkCount(BAT_INDEX, 0, method);
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.removeItem(allItems[FOOTBALL_INDEX], 1));
        checkCount(FOOTBALL_INDEX, 0, method);
        //Check second element in pair
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.addItem(allItems[BAT_INDEX], 1));
        checkCount(BAT_INDEX, 1, method);
        assertEquals(errorMessage,
                INSERT_ERROR_CODE_CONTRADICTION, testLocker.addItem(allItems[FOOTBALL_INDEX], 1));
        checkCount(FOOTBALL_INDEX, 0, method);
    }
    /** check insert of negative amount
```

```
     */
    private void insertNegative()
    {
        final String method = "insertNegative";
        String errorMessage = getErrorPrefix(method);
        //insert non-existing item with negative amount
        assertEquals(errorMessage,
                ERROR_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], -2));
        checkCount(BIG_HELMET_INDEX, 0, method);
        //insert existing item with negative amount
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 2));
        assertEquals(errorMessage,
                ERROR_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], -2));
        checkCount(BIG_HELMET_INDEX, 2, method);
    }
    /** check insert fails when LTS is full
     */
    private void checkFillingUpLTS()
    {
        final String method = "checkFillingUpLTS";
        String errorMessage = getErrorPrefix(method);

        int i = 0;
        removeAllCopiesOfItem(allItems[ENGINE_INDEX].getType());
        while(testLocker.getLongTermStorage().getAvailableCapacity() > 50 && i < 20)
        {
            addMoreThanHalfCapacityAtOnce();
            i++;
        }

        assertTrue(errorMessage,
                testLocker.getLongTermStorage().getAvailableCapacity() < 50);
        assertEquals(errorMessage, ERROR_CODE, testLocker.addItem(allItems[ENGINE_INDEX], 5));
        removeAllCopiesOfItem(allItems[ENGINE_INDEX].getType());
    }
    /** init the locker btw tests
     */
    private void initLocker()
    {
        //Make copy of inventory and delete it all from the locker
        Map<String, Integer> currentInventory = new HashMap<String, Integer>(testLocker.getInventory());
        for(Map.Entry<String, Integer> entry : currentInventory.entrySet())
        {
            removeAllCopiesOfItem(entry.getKey());
        }
        //Check that now the inventory id indeed empty
        assertEquals(0, testLocker.getInventory().size());
    }
    /** delete item from locker
     */
    private void removeAllCopiesOfItem(String type)
    {
        final String method = "checkFillingUpLTS";
        String errorMessage = getErrorPrefix(method);

        int count = testLocker.getItemCount(type);
        Item itemToDelete = ItemFactory.createSingleItem(type);
        assertEquals(errorMessage, SUCCESS_CODE, testLocker.removeItem(itemToDelete, count));
    }
    /** try removing negative amount
     */
    private void removeNegative()
    {
        final String method = "removeNegative";
        String errorMessage = getErrorPrefix(method);

        //Remove existing item with negative amount
```

```java
        assertEquals(errorMessage,
                    ERROR_CODE, testLocker.removeItem(allItems[FOOTBALL_INDEX], -2));
        //Remove non existing item with negative amount
        assertEquals(errorMessage,
                ERROR_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], -2));
    }
    /** try removing successfully
     */
    private void regularRemove()
    {
        final String method = "regularRemove";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 1));
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], 1));
        checkCount(BIG_HELMET_INDEX, 0, method);
    }
    /** try removing over the real amount
     */
    private void nonExistingCountRemove()
    {
        final String method = "regularRemove";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                SUCCESS_CODE, testLocker.addItem(allItems[BIG_HELMET_INDEX], 1));
        assertEquals(errorMessage,
                    ERROR_CODE, testLocker.removeItem(allItems[BIG_HELMET_INDEX], 10));
    }

    /**  Get error msg
     * @param method param to add to msg
     * @return error msg
     */
    private String getErrorPrefix(String method)
    {
        String msgFormat = ERROR_PREFIX;
        return MessageFormat.format(msgFormat, method);
    }

}
```

9 LongTermStorage.java

```java
import oop.ex3.spaceship.Item;
/**
 * This class represents LTS inside the spaceship
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public class LongTermStorage extends BasicStorage
{
    //Constants
    int LTS_CAPACITY = 1000;

    /** Contractor
     */
    public LongTermStorage()
    {
        setCapacity(LTS_CAPACITY);
    }
    /** Adding a new item/s to this lts
     * @param item the type of item we are trying to add to this locker
     * @param n the number of copies of the current item we are trying to add
     * @return 0 if we succeed adding these items, -1 otherwise.
     */
    public int addItem(Item item, int n)
    {
        //Check if there enough room in the locker for all the new items
        if(checkIfThereEnoughRoom(item, n) == false)
        {
            return INSERT_ERROR_CODE;
        }
        //Insert Items to inventory
        addToInventory(item, n);
        return SUCCESS;
    }
    /** reset the lts storage
     */
    public void resetInventory()
    {
        initializeStorage();
    }
}
```

10 LongTermTest.java

```java
import oop.ex3.spaceship.Item;
import org.junit.*;
import static org.junit.Assert.*;
import oop.ex3.spaceship.ItemFactory;

import java.text.MessageFormat;

/**
 * This class is used test class LongTermStorage.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class LongTermTest
{
    //Constants
    protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
    protected static final int INITIAL_SIZE = 0;
    protected static final int SUCCESS_CODE = 0;
    protected static final int ERROR_CODE = -1;
    protected static final int BAT_INDEX = 0;
    protected static final int SMALL_HELMET_INDEX = 1;
    protected static final int BIG_HELMET_INDEX = 2;
    protected static final int ENGINE_INDEX = 3;
    protected static final int FOOTBALL_INDEX = 4;

    // instance variables
    static private final int capacity = 1000;
    private static LongTermStorage ltsTest;
    static private final Item[] allItems = ItemFactory.createAllLegalItems();

    // tests methods
    /** build testers objects
     */
    @BeforeClass
    public static void setUp()
    {
        ltsTest = new LongTermStorage();
    }


    /** check method removeItem
     */
    @Test
    public void removeItem()
    {
        removeNegative();
        regularRemove();
        nonExistingCountRemove();
    }
    /** check method getInventory
     */
    @Test
    public void getInventory()
    {
        final String method = "getInventory";
        String errorMessage = getErrorPrefix(method);
        assertNotNull(errorMessage, ltsTest.getInventory());
    }
    /** check method getCapacity
```

```java
     */
    @Test
    public void getCapacity()
    {
        final String method = "getCapacity";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage, capacity, ltsTest.getCapacity());
    }
    /** check method addItem
     */
    @Test
    public void addItem()
    {
        addItemOverflow();
        regularInsert();
        insertNegative();
        checkConstraints();
    }

    /** check method resetInventory
     */
    @Test
    public void resetInventory()
    {
        final String method = "resetInventory";
        String errorMessage = getErrorPrefix(method);

        ltsTest.addItem(allItems[SMALL_HELMET_INDEX], 1);
        assertTrue(errorMessage, ltsTest.getInventory().size() > 0);
        ltsTest.resetInventory();
        //Check that now the inventory id indeed empty
        assertEquals(errorMessage, 0, ltsTest.getInventory().size());
    }

    /** Try add at once more than the capacity
     */
    private void addItemOverflow()
    {
        final String method = "addItemOverflow";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage, ERROR_CODE, ltsTest.addItem(allItems[ENGINE_INDEX], 2000));
    }

    /** Try add successfully
     */
    private void regularInsert()
    {
        final String method = "regularInsert";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[SMALL_HELMET_INDEX], 1));
        checkCount(SMALL_HELMET_INDEX, 1, method);
    }

    /** Check count of an item in lts
     * @param typeIndex index of item to check
     * @param count count to check
     * @param source calling test name
     */
    private void checkCount(int typeIndex, int count, String source)
    {
        final String method = source + " - checking count of item";
        String errorMessage = getErrorPrefix(method);

        assertEquals(errorMessage, count, ltsTest.getItemCount(allItems[typeIndex].getType()));
        if(count > INITIAL_SIZE)
        {
            assertTrue(errorMessage,
                    ltsTest.getInventory().get(allItems[typeIndex].getType()) == count);
        }
        else
        {
            assertNull(errorMessage, ltsTest.getInventory().get(allItems[typeIndex].getType()));
        }
    }
    /** Try add negative amount
     */
    private void insertNegative()
    {
        final String method = "insertNegative";
        String errorMessage = getErrorPrefix(method);

        //insert non-existing item with negative amount
        assertEquals(errorMessage,
                ERROR_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], -2));
        checkCount(BIG_HELMET_INDEX, 0, method);
        //insert existing item with negative amount
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 2));
        assertEquals(errorMessage,
                     ERROR_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], -2));
        checkCount(BIG_HELMET_INDEX, 2, method);
    }
    /** Check LTS has no constraints
     */
    private void checkConstraints()
    {
        final String method = "checkConstraints";
        String errorMessage = getErrorPrefix(method);

        //Check first element in pair
        assertEquals(errorMessage,
                     SUCCESS_CODE, ltsTest.addItem(allItems[FOOTBALL_INDEX], 1));
        checkCount(FOOTBALL_INDEX, 1, method);
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[BAT_INDEX], 1));
        checkCount(BAT_INDEX, 1, method);
        ltsTest.resetInventory();
        //Check second element in pair
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[BAT_INDEX], 1));
        checkCount(BAT_INDEX, 1, method);
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[FOOTBALL_INDEX], 1));
        checkCount(FOOTBALL_INDEX, 1, method);
    }
    /** try remove negative amount
     */
    private void removeNegative()
    {
        final String method = "removeNegative";
        String errorMessage = getErrorPrefix(method);

        //Remove existing item with negative amount
        assertEquals(errorMessage,
                     ERROR_CODE, ltsTest.removeItem(allItems[FOOTBALL_INDEX], -2));
        //Remove non existing item with negative amount
        assertEquals(errorMessage,
                     ERROR_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], -2));
    }
    /** try remove successfully
     */
    private void regularRemove()
    {
        final String method = "regularRemove";
        String errorMessage = getErrorPrefix(method);

        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 1));
        assertEquals(errorMessage,
                     SUCCESS_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], 1));
        checkCount(BIG_HELMET_INDEX, 0, method);
    }
    /** try remove over the right amount
     */
    private void nonExistingCountRemove()
    {
        final String method = "regularRemove";
        String errorMessage = getErrorPrefix(method);
        assertEquals(errorMessage,
                SUCCESS_CODE, ltsTest.addItem(allItems[BIG_HELMET_INDEX], 1));
        assertEquals(errorMessage,
                     ERROR_CODE, ltsTest.removeItem(allItems[BIG_HELMET_INDEX], 10));
    }

    /** Get error msg
     * @param method param to add to
     * @return error msg
     */
    private String getErrorPrefix(String method)
    {
        String msgFormat = ERROR_PREFIX;
        return MessageFormat.format(msgFormat, method);
    }
}
```

11 Spaceship.java

```java
import oop.ex3.spaceship.Item;
/**
 * This class represents a spaceship
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public class Spaceship
{
    //Constants
    private static final int ZERO = 0;
    protected static final int ERROR_NOT_VALID_ID = -1;
    protected static final int ERROR_NOT_VALID_CAPACITY = -2;
    protected static final int ERROR_LOCKERS_OVERFLOW = -3;
    protected static final int SUCCESS_CODE = 0;
    // instance variables
    private String shipName;
    private LongTermStorage longTermUnit = new LongTermStorage();
    private Locker[] lockers;
    private int lockerCounter = ZERO;
    private int maxNumOfLockers;
    private int[] shipsCrew;
    private Item[][] storageRegulations;

    /** Contractor
     * @param name the current ship's name.
     * @param crewIDs array of crew's ID numbers.
     * @param numOfLockers the capacity of lockers this ship can have.
     * @param constraints an array of pairs of two items that are NOT allowed to reside
     * together in a locker.
     */
    public Spaceship(String name, int[] crewIDs, int numOfLockers, Item[][] constraints)
    {
        shipName = name;
        shipsCrew = crewIDs;
        maxNumOfLockers = numOfLockers;
        lockers = new Locker[numOfLockers];
        storageRegulations = constraints;
    }
    /** Get the long term storage unit object
     * @return the long-term storage object associated with that Spaceship.
     */
    public LongTermStorage getLongTermStorage()
    {
        return longTermUnit;
    }
    /** creates a new Locker object inside the spaceship
     * @param crewID the locker's owner's ID.
     * @param capacity the capacity the new locker could hold.
     */
    public int createLocker(int crewID, int capacity)
    {
        //Check this is a real crew member
        if(checkID(crewID) == false)
        {
            return ERROR_NOT_VALID_ID;
        }

        if(capacity < ZERO)
        {
            return ERROR_NOT_VALID_CAPACITY;
        }

        if(lockerCounter == maxNumOfLockers)
        {
            return ERROR_LOCKERS_OVERFLOW;
        }

        lockers[lockerCounter] = new Locker(getLongTermStorage(), capacity, storageRegulations);
        lockerCounter++;
        return SUCCESS_CODE;
    }
    /** Get the crew's ids.
     * @return an array of the crew's IDs numbers
     */
    public int[] getCrewIDs()
    {
        return shipsCrew;
    }
    /** Get ship's lockers.
     * @return an array of the ship's lockers
     */
    public Locker[] getLockers()
    {
        return lockers;
    }

    /** Check given id
     * @param id the id to check
     * @return true if the id is valid, false otherwise.
     */
    public boolean checkID(int id)
    {
        for(int i = 0; i < shipsCrew.length; i++)
        {
            if(shipsCrew[i] == id)
            {
                return true;
            }
        }
        return false;
    }
}
```

12 SpaceshipDepositoryTest.java

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)

@Suite.SuiteClasses({
    LockerTest.class,
    LongTermTest.class,
    SpaceshipTest.class
})

/**
 * This class is used to run full tests on Spaceship.class, LockerTest.class
 * and LongTermTest.class
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public class SpaceshipDepositoryTest
{
}
```

13 SpaceshipTest.java

```java
import oop.ex3.spaceship.Item;
import org.junit.*;
import static org.junit.Assert.*;
import oop.ex3.spaceship.ItemFactory;
import java.text.MessageFormat;
/**
 * This class is used test class Spaceship.
 * @author Bar Melinarskiy
 * @version 26/8/20
 */
public class SpaceshipTest {
    //Constants
    protected static final String ERROR_PREFIX = "Error: Test {0} has failed. ";
    protected static final int ERROR_NOT_VALID_ID = -1;
    protected static final int ERROR_NOT_VALID_CAPACITY = -2;
    protected static final int ERROR_LOCKERS_OVERFLOW = -3;
    protected static final int SUCCESS_CODE = 0;

    // instance variables
    static private Spaceship testSpaceShip;
    static final private Item[][] constraints = ItemFactory.getConstraintPairs();
    static final private int[] crewIDs = new int[]{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    static final private int numOfLockers = 3;
    //test methods

    /** build testers objects
     */
    @BeforeClass
    public static void createTestObjects()
    {
        testSpaceShip = new Spaceship("test", crewIDs, numOfLockers, constraints);
    }
    /** test method getLongTermStorage
     */
    @Test
    public void getLongTermStorage()
    {
        final String method = "getLongTermStorage";
        String errorMessage = getErrorPrefix(method);
        assertNotNull(errorMessage, testSpaceShip.getLongTermStorage());
    }
    /** test method createLocker
     */
    @Test
    public void createLocker()
    {
        final String method = "createLocker";
        String errorMessage = getErrorPrefix(method);
        //wrong id
        assertEquals(errorMessage,
                ERROR_NOT_VALID_ID, testSpaceShip.createLocker(20, 10));
        //Non valid capacity
        assertEquals(errorMessage,
                ERROR_NOT_VALID_CAPACITY, testSpaceShip.createLocker(1, -4));
        for(int i = 0 ; i <= numOfLockers; i++)
        {
            if(i < numOfLockers)
            {
                //Valid new locker
                assertEquals(errorMessage,
                        SUCCESS_CODE, testSpaceShip.createLocker(i + 1, 100));
                //Check the creation was really successful
                assertNotNull(errorMessage, testSpaceShip.getLockers()[i]);
            }
            else
            {
                //One too many locker, check if we get error code for overflow
                assertEquals(errorMessage,
                    ERROR_LOCKERS_OVERFLOW, testSpaceShip.createLocker(i + 1, 100));
            }
        }

    }
    /** test method getCrewIDs
     */
    @Test
    public void getCrewIDs()
    {
        final String method = "getCrewIDs";
        String errorMessage = getErrorPrefix(method);

        int[] crewArray = testSpaceShip.getCrewIDs();
        assertNotNull(errorMessage, crewArray);
        //Check valid length of array
        assertEquals(errorMessage, crewIDs.length, crewArray.length);
        //Check all the IDs are there
        assertArrayEquals(errorMessage, crewArray, crewIDs);
    }
    /** test method getLockers
     */
    @Test
    public void getLockers()
    {
        final String method = "getCrewIDs";
        String errorMessage = getErrorPrefix(method);

        Locker[] lockersArray = testSpaceShip.getLockers();
        assertNotNull(errorMessage, lockersArray);
        //Check valid length of array
        assertEquals(errorMessage, numOfLockers, lockersArray.length);
    }

    /** Get error msg
     * @param method param to add to msg
     * @return error msg
     */
    private String getErrorPrefix(String method)
    {
        String msgFormat = ERROR_PREFIX;
        return MessageFormat.format(msgFormat, method);
    }
}
```

14 StorageWithLTS.java

```java
import oop.ex3.spaceship.Item;
/**
 * This class is an abstract class for all the methods related to an object
 * who is connected to LTS and as such has regulations to follow
 * @author Bar Melinarskiy
 * @version 16/8/20
 */
public abstract class StorageWithLTS extends BasicStorage
{
    //Constants
    protected static final int PAIR_FIRST_ELEMENT = 0;
    protected static final int PAIR_SECOND_ELEMENT = 1;
    protected static final int INSERT_WARNING_CODE = 1;

    // instance variables
    private static LongTermStorage longTermUnit;
    private static Item[][] storageRegulations;

    // instance methods
    /** Get the long term storage unit object
     * @return the long-term storage object associated with that Spaceship.
     */
    public LongTermStorage getLongTermStorage()
    {
        return longTermUnit;
    }

    /** Set the long term storage unit object
     * @param lts the current ship's Long-Term storage unit.
     */
    public void setLongTermStorage(LongTermStorage lts)
    {
        longTermUnit = lts;
    }

    /** Set locker's storage regulations
     * @param constraints an array of pairs of two items that are NOT allowed to reside
     * together in a locker.
     */
    protected void setStorageRegulations(Item[][] constraints)
    {
        storageRegulations = constraints;
    }

    /** Check storage regulations
     * @param type the type of item we wanna check
     * @return true if this item can not be stores inside this locker because of constraints,
     * false otherwise.
     */
    protected boolean checkStorageRegulations(String type)
    {
        Item firstItem, secondItem;
        String firstType, secondType;
        //Loop through the given pairs
        for(int row = 0; row < storageRegulations.length; row++)
        {
            //Get the current pairs' info
            firstItem = storageRegulations[row][PAIR_FIRST_ELEMENT];
            firstType = firstItem.getType();
            secondItem = storageRegulations[row][PAIR_SECOND_ELEMENT];
            secondType = secondItem.getType();
            //Check if the current first element is eq to the given type
            if(firstType.equals(type) && getItemCount(secondType) > INITIAL_SIZE)
            {
                return true;
            }
            //Check if the current second element is eq to the given type
            else if(secondType.equals(type)&& getItemCount(firstType) > INITIAL_SIZE)
            {
                return true;
            }
        }

        return  false; //if we reached this point then there is no problem adding this item
    }

}
```