Project 1: Shopnstop
Due: 5:00PM, February-22-2019

Contents

1 The Story .......................... 1

2 The Portal (An Overview) .......... 1
   2.1 Types of Listings .................. 2
   2.2 Features of Listings ............... 2

3 Users ................................ 3
   3.1 Realtors .......................... 3
   3.2 Buyers ........................... 3

4 Viewing Listings ................... 3

5 Parsing ............................ 5
   5.1 Parsing Tips ....................... 5

6 Support Code ....................... 6
   6.1 Reading in Input .................. 6
   6.2 Parsing Dates ...................... 6
   6.3 Printing Messages to the Terminal .................. 6

7 Testing ........................... 7
   7.1 Unit Testing ....................... 7
   7.2 System Testing .................... 7
   7.3 Redirection ....................... 8

8 Demo ................................ 9

9 Handin ............................. 9
   9.1 Design Check ...................... 9
   9.2 Final Handin ...................... 9
   9.3 Grading .......................... 10
1 The Story

After Pikachu and the gang moved to the Windy City of Chicago for its amazing transit system, they realized they needed money to actually ride any transportation! After trying to be chefs, artists, and software developers, they finally found success in the realty business. Word of mouth wasn’t getting them enough business to ride all of the transits they wanted, so they decided they would hire a programmer to create a portal to handle business transactions for buyers and realtors!

Your job as the programmer is to create a portal for realtors and buyers, where the respective parties can handle their ends of the realty business. Realtors will be able to add and remove listings, while buyers will be able to book, rent, and purchase the available listings provided by the realtors. With this new program, Pikachu and the gang will have enough money to be able to ride all of the transportation systems they want!

2 The Portal (An Overview)

Before we get started, although all the information you need to know about this project is in this document, we highly recommend you run our demo to fully understand how to interact with the program and what to print/trigger when certain user input is passed in. See Section 8.

The portal will be an interactive command line REPL (Read Evaluate Print Loop), with functionality for both buyers and realtors. Your program should be run from the Portal.java file provided in /course/cs0180/sol/shopnstop/sol/. The portal will have a login page, where the user can:

- Login As A New Buyer: Login to the portal as a new buyer.
- Login As A New Realtor: Login to the portal as a new Realtor.
- Quit: Quit the portal entirely.

This page will look like the following:


Once the user selects whether they login as a Buyer or Realtor, a new set of commands are presented to the user. While both users are presented with the same commands, some options act differently when selected. The common functionality for both users include:
• View Available Listings: View all of the current available listings in the system.
• Logout: Logout of the current session.
• Quit: Quit the portal entirely.

While these act differently depending on which user is logged in:

<table>
<thead>
<tr>
<th>Option</th>
<th>Buyer</th>
<th>Realtor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Listing</td>
<td>Prints an error message informing the Buyer that they do not have permission to perform the requested action.</td>
<td>Prompts the Realtor to add a listing to the system.</td>
</tr>
<tr>
<td>Remove Listing</td>
<td>Prompts the Realtor to remove a listing to the system.</td>
<td></td>
</tr>
<tr>
<td>Select Listing</td>
<td>Prompts Buyer to purchase/rent/book the listing (depending on the type of selected listing)</td>
<td>Prompts the Realtor to edit the selected listing.</td>
</tr>
</tbody>
</table>

2.1 Types of Listings

There are three types of listings: Hotel, Rent, and Purchase. When a buyer chooses a listing, they will have to choose a series of dates to book that listing if it is of type Hotel or Rent (this implies you must handle multiple space separated dates in the format specified in Section 5.1). Another consideration is that Hotels and Rent Listings cannot be double booked. This means that if the Buyer originally booked a Hotel for May 18, 2019 and they try to book the same Hotel for the date May 18, 2019 the result of this attempt will be voided. In sum, the Buyer cannot double book dates for Rent listings or Hotels.

If they wish to Purchase a listing, they do not need to specify this information, as the listing will become theirs instantly. When a Realtor adds a listing, they will need to specify the type of this listing.

For this project you will assume that the Buyer has enough money to purchase, book or rent their desired listing.

Note: You are welcome to generate a default listing each time the portal is run by the user, so that it would be easier for you to test your program out a bit! But please remember to remove this functionality before handing in so that your program will work with our test suite.

2.2 Features of Listings

As we discussed above, there are three different types of listings. Despite the difference in how a Buyer might interact with a listing (purchasing or booking/renting). All listings must hold the following data:

• Name (ex: "CIT")
• Price (ex: 40000000.99)
• Size in square feet (ex: 168800.66)
• Year built (ex: 1988)
• A list of previous owner ratings (ex: (5, 4, 5, 5, 5))

3 Users

You are responsible for incorporating two types of users into your portal: Buyers and Realtors.
Both Users have the same home screen, regardless of their ability to engage with the commands displayed. Namely, both Buyer and Realtor will login and see (without the line break!):


3.1 Realtors

Realtors are able to view, add, and remove listings from the portal. They can sort the listings by price, size, year, and average rating. When a listing is selected, the Realtor will be given the option of editing the listing’s name, price, size and year. Note that the Realtor cannot edit the listing’s past ratings.

Here are the main capabilities that Realtors have:

• Add and remove listings. This includes instantiating all the fields for a specific listing, including ratings.
• Modify the fields of listings.

3.2 Buyers

Buyers are only able to view and select listings in the portal, they cannot add or remove them. They can sort the listings by price, size, and year built but cannot sort by average rating since only Realtors have access to ratings.

Here are the main capabilities that Buyers have:

• Purchase listings. When they are purchased they are removed from the portal and does not become an added field for the Buyer.
• Book dates for Hotels and Rent Listings. When booking Rent Listings or Hotels, Buyers can specify multiple dates they want to reserve the listing for. These dates must be space separated.
• Buyers cannot give ratings.
4 Viewing Listings

When a user requests to view the current listings in the portal, different information will be printed to the terminal depending on whether the user is a Buyer or a Realtor. When a Realtor requests to view the listings in the portal, each listing’s name, type, price, size, year built, and average rating will be printed to the terminal. However, Buyers cannot see the rating of any listings. So, when a Buyer requests to view the listings in the portal, only each listing’s name, type, price, size, and year built are printed to the terminal.

There are a few different ways in which users can sort the current listings in the portal before they are printed to the terminal. When a Realtor requests to view the available listings, they may order the listings by either price, size, year built, or average rating, all in decreasing order. However, since Buyers cannot see the ratings of listings, they may only order the listings by price, size, or year built in decreasing order.

We can use the power of the Comparator interface to sort the listings in the portal. An object that implements Comparator compares objects of type T by containing a method compare( T obj1, T obj2). This method returns either a positive int if obj1 is greater than obj2, 0 if obj1 is equal to obj2, or a negative int if obj1 is less than obj2.

Let’s consider a simple example to illustrate how the Comparator interface works.

Consider the following Squirtle class:

```java
class Squirtle {
    public int sunglassesAmount;
    public double shellRadius;
    public Squirtle(int sunglassesAmount, double shellRadius) {
        this.sunglassesAmount = sunglassesAmount;
        this.shellRadius = shellRadius;
    }
}
```

Now consider a list, squirtleList, with two objects in it: Squirtle babySquirtle = new Squirtle(2, 4.5) and Squirtle teenSquirtle = new Squirtle(32, 7.4). If I were to call Collections.sort(squirtleList) Java wouldn’t know what to use to decide how to order the objects in the list... that’s where Comparators come in!

If I wanted to tell Java to sort the list in increasing order of Squirtle sunglassesAmounts, we could instantiate the following Comparator object.

```java
Comparator<Squirtle> incSunglassesComparator = new Comparator<Squirtle>() {
    public int compare(Squirtle s1, Squirtle s2) {
        return Integer.compare(s1.sunglassesAmount, s2.sunglassesAmount);
    }
};
```

In the case of babySquirtle and teenSquirtle, since teenSquirtle’s sunglassesAmount is greater than that of babySquirtle, Collections.sort(squirtleList, incSunglassesComparator) will place babySquirtle before teenSquirtle. If I wanted to display teenSquirtle before babySquirtle, I would have to flip the return values of the call to compare:
Comparator<Squirtle> decSunglassesComparator = new Comparator<Squirtle>() {
    public int compare(Squirtle s1, Squirtle s2) {
        return Integer.compare(s2.sunglassesAmount, s1.sunglassesAmount);
    }
};

Since similarly, if we wanted to sort Squirtles in decreasing order by their shellRadius, we can do so with the following Comparator:

Comparator<Squirtle> decShellComparator = new Comparator<Squirtle>() {
    public int compare(Squirtle s1, Squirtle s2) {
        return Double.compare(s2.shellRadius, s1.shellRadius);
    }
};

5 Parsing

Your code should gracefully handle user input and should not break regardless of whether the user input was valid. A convenient way to handle this is using try-catch blocks which are covered below. We have provided a class UserIO that handles the parsing of dates and converts a string formatted as “MM-DD-YYYY” to a linked list of Date objects. We have also included a method to convert a list of dates to a string. Be sure to familiarize yourself with the methods in the UserIO class. You will find these methods with descriptions in the src directory.

5.1 Parsing Tips

Java has a few built-in methods that you may find useful in parsing user input. For example, the String class provides a method called split, which is used to divide a String into separate words, which you might find helpful when parsing ratings:

java> String example = new String("I am an example")
java> String[] myArray = example.split(" ")
myArray = {"I", "am", "an", "example"}

The Integer class provides a static method called parseInt for converting a String into an Integer. For example:

java> String number = "18"
java> int myInt = Integer.parseInt(number)
myInt = 18

Be careful with Integer.parseInt; if your String input isn’t a valid integer, it will throw a NumberFormatException. However, this Exception should not reach the user! Instead, you should catch the Exception using a try-catch block. For instance, if in the following piece of code a NumberFormatException is thrown in the try block, the content of the catch block will be executed instead:

```java
try {
    int myInt = Integer.parseInt(number);
} catch (NumberFormatException e) {
    // handle exception
}
```
try {
    int myInt = Integer.parseInt("Not a number!");
} catch (NumberFormatException nfe) {
    System.out.println("You just tried to input something that's not a number!");
}

There are similar constructs for Doubles, Floats, etc.

6 Support Code

To facilitate the Input/Output interaction between your program and its user, we have provided the support code files UserIO.java, LogPrompt.java, LogListing.java, LogConfirmation.java, and LogError.java. These files contain methods that take care of:

- receiving user input by reading from standard input and returning a string.
- parsing and converting Strings into Date objects and vice-versa.
- printing messages to the terminal.

Note: To access these files, please copy over all files in /course/cs0180/src/shopnstop/src into your Shopnstop src directory.

Below is a more detailed description of the support code. You should read through this to understand how to use the support code properly in your own solution. Note that all these methods are static, meaning you don’t need an instance of the class to access them. We can, for example, call getNextInput() from anywhere in our program, as long as we are importing shopnstop.src.UserIO by calling UserIO.getNextInput().

6.1 Reading in Input

The UserIO class contains the static method getNextInput(). You must use this method when reading the user’s input from the terminal. This method takes in no arguments and returns the String that was read from the command line.

6.2 Parsing Dates

Two static methods parseDates and convertDates are also included in the UserIO class.

The parseDates method takes in a String as input, which should be dates in format mm-dd-yyyy, and returns corresponding dates as a list of Date objects.

On the other hand, the convertDates method does the reverse job. That is to say, it takes as input a list of Date objects and converts the dates into formatted strings.
6.3 Printing Messages to the Terminal

Rather than using `println` to print to the terminal for this project, you will use the methods provided in the `LogPrompt.java`, `LogListing.java`, `LogConfirmation.java`, and `LogError.java` files for all of your printing needs. Some details:

- The `LogPrompt` class contains static methods for printing messages to the terminal that prompt the user for input
- The `LogListing` class contains static methods that print details about listings to the terminal
- The `LogConfirmation` class contains static methods that print confirmation messages to the terminal
- The `LogError` class contains static methods that print error messages to the terminal for when the user gives invalid input.

You should never use `print`, `println`, or any other method that prints messages to the terminal, besides the ones provided in the support code, in your code for this project! Otherwise it will become extremely challenging to grade your handin and you will lose points.

7 Testing

Thoroughly testing your project will involve both unit testing and system testing.

7.1 Unit Testing

Unit tests attempt to establish the correctness of small bits of functionality within a program, assuming everything else works as intended. While all code should, in theory, be unit tested, for this project, you need only carry out a few unit tests. Some good pieces of functionality to unit test include:

- Any methods responsible for sorting (for example, right before displaying the listings).
- Any method that is modifying the state of a collection of objects, so when adding and removing Listings, for example.

If you believe your program would benefit from additional unit testing, you are encouraged to use this testing technique!

7.2 System Testing

System testing attempts to establish the correct functionality of an entire system, in which multiple chunks of code interact with one another. In this project, thorough system testing will require you to interact with your portal repeatedly, invoking multiple different behaviors, and making sure that all methods work as intended.
To demonstrate to us that you performed thorough system tests, you should run through multiple portal sessions, and record and submit your inputs and the resulting outputs. Make sure the default commands, such as selecting, removing and adding listings, all work. Then try invalid commands, indices, etc., and record the error messages received. Add this documentation to the end of your README file in a section entitled Testing. Include a header for each of your tests that state what functionality is being tested.

To make this testing process more efficient, you can create a text file of commands that can be run through your programs using a terminal trick known as redirection (see Section 7.3). You can create one (or preferably several) of these files to test the functionality of your program, with each one leading to a different displayed results/prompts. If you would like to split your write up into multiple files, feel free; however, please describe where and what those files are in your README so your grader knows what you did.

Your testing documentation should be sufficient to convince yourselves (and your TAs) that you have thoroughly tested your program.

7.3 Redirection

Redirection is the process of changing where functions like readLine, println, and other I/O operations read or write text. When you run a program, every time it calls readLine, it waits for the user to type some input, and every time it calls println, it outputs some text to the terminal. Not so with redirection. With redirection you can specify where readLine reads lines from, where println outputs text to, and much more! Redirection is accomplished with two meta Heroes, ‘<’ and ‘>’, which redirect input and output, respectively.

Rather than interacting with your portal by entering commands one after another (which could get tedious after hours of debugging), you can create a file listing a series of commands to be executed (in order), say commands.txt, and then run your program on that file. To do this, you would redirect that file to your program, so that functions like readLine would treat the file commands.txt as standard input, instead of waiting for input from the usual standard input (the keyboard). Here is how to specify that your Shopnstop program should redirect its standard input to commands.txt:

```
java shopnstop.sol.Portal < commands.txt
```

Redirecting output is very similar; simply add ‘> output.txt’ to the end of the previous command. This will write everything that would have been printed to standard output to the specified file (creating it if it does not exist, and overwriting it if it does).

As a reminder, to compile your code navigate to ‘/course/cs0180/workspace/javaproject’ and enter:

```
javac -d bin s*/shopnstop/*//*.java
```

To compile your code then navigate into the bin directory (by entering cd bin) and run the program:

```
java shopnstop.sol.Portal < inputs.txt > outputs.txt
```
Which would enter the commands specified in ‘inputs.txt’ and redirect the program’s output to ‘outputs.txt’

Check out the Command Line Guide for more information on working with Java on the command line.

8 Demo

We’ve created a demo for you to interact with. If you have any questions about what your program should be doing in specific cases, the demo should be your first reference point. Your final project should behave identically to the demo.

To access the demo, run

```
shopnstop_demo
```

from anywhere in your terminal.

9 Handin

9.1 Design Check

Design checks will be held on February 12 and 13. We will send out an email detailing how to sign up for design checks; it is in your best interest to sign up as soon as possible (before all the prime time slots have been filled).

**Reminder:** You are required to pair program at least the design check portion of all CS 18 projects. We recommend finding a partner as soon as possible, as you will not be able to sign up for a design check without one.

Before the design check, you and your partner must make sure you have:

- Discussed a detailed class hierarchy including any interfaces, abstract classes and classes you plan on using. Having a good idea of where functionality will live is a good idea, but you don’t have to have all the method names and input/outputs, a general idea is fine. If you want to read more about class diagrams, please refer to [this article](#).

- A clear class diagram to show your TA.

- A clear understanding of how Java’s Comparators work and how you might use them in your implementation of the project.

- A walkthrough of what your program will do based on some sample user input.

9.2 Final Handin

The final handin of your project is due February 22 at 5:00pm. Your `javaproject` should contain the packages `shopnstop.src` (in the `src` directory) and `shopnstop.sol` (in the `sol` directory). Your
code should be part of the shopnstop.sol package. This package must contain Portal.java which, as mentioned earlier, is where your program should be run from. That package should also contain a README.txt file.

Your README file should include:

- instructions for use, describing how a user would interact with your program
- a brief overview of your design, including how all of the pieces of your program fit together
- a description of any features you failed to implement, as well as any extra features you chose to implement
- a description of any known bugs in your program
- a description of how you tested your program
- a list of the people with whom you collaborated

To hand in your files, navigate to the ~/.course/cs0180/workspace/javaproject directory, and run the command `cs018_handin shopnstop`. This will automatically hand in the contents of your entire javaproject directory. Once you have handed in your project, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

**Note:** Only one of you or your partner must hand in the project.

### 9.3 Grading

As with all CS 17/18 projects, a good design will make coding this project significantly easier; so you should spend a fair amount of time working on your program’s design before you begin writing any code.

The design check counts for 15% of your grade, including:

- An overview of your class hierarchy.
- A walkthrough of what your program will do when receiving input from the user.
- An explanation of Comparators and how you plan on using them.

The goal of this project is for you to familiarize yourself with, and then apply, basic principles of object-oriented design. As such, 15% of your grade will assess your design only.

Functionality counts for 50% of your grade, including:

- Correct implementation of basic portal mechanics.
- Correct implementation of all required listing types.
- Correct application of user inputs.
As always, partial functionality merits partial credit.

The final 20% of your grade will be reserved for comments, testing, and style. You should include documentation comments and test cases for all non-trivial methods. You should also perform system testing, to test interactions among methods. Additionally, comment any code which would otherwise be unclear.

A Double Dispatch

Searching desperately for ways to practice your new found object-oriented programming skills, you decide to model the behaviors of your four favorite head TAs—Alex, Mae, Eleanor, Liam,—in Java. Knowing how special each individual one of them is, you decide to define an abstract class Person, and then subclass that class for each head TA. Here’s what your Person class looks like:

```java
package doubleDispatch;

/**
 * An abstract class that represents persons.
 */

public abstract class Person {
    String name;
    String greeting;

    /**
     * A constructor for Person
     */
    public Person() {
        this.name = "Person";
        this.greeting = "Hey There! I'm a Person!";
    }

    /**
     * Talks to another person by printing a greeting.
     * @param aPerson - the person to talk to
     */
    public void talkTo(Person aPerson) {
        System.out.print(this.greeting + " ");
    }

    /**
     * Specific behavior for talking to Alex
     */
    public abstract void talkTo(Alex aAlex);

    /**
     * Specific behavior for talking to Eleanor
     */
    public abstract void talkTo(Eleanor aEleanor);
}
```
Your `Person` class includes a method for one person to talk, in a completely generically manner, to another. But then your each of your four subclasses (one per head TA) includes four additional `talkTo` methods specifying how each head TA talks to all the others (including themselves!). For example, here’s what Alex’s class (the labs head TA) looks like:

```java
package doubleDispatch;

/*
 * A class that represents Alex!
 */
public class Alex extends Person {

    /*
     * A constructor for Alex
     */
    public Alex() {
        this.name = "Alex";
        this.greeting = "Hey there! I'm Alex";
    }

    @Override
    public void talkTo(Alex aAlex) {
        super.talkTo((Person) this);
        System.out.println("Alex, I just love talking to myself");
    }

    @Override
    public void talkTo(Eleanor eEleanor) {
        super.talkTo((Person) this);
        System.out.println("Eleanor! How are homeworks? Labs are great!");
    }

    @Override
    public void talkTo(Liam aLiam) {
        super.talkTo((Person) this);
        System.out.println("Liam! How are homeworks as well? Labs are great!");
    }

    @Override
    public void talkTo(Mae aMae) {

```
super.talkTo((Person) this);
System.out.println("Mae, how are projects? Labs are great!");
}
}

Now Alex can greet Eleanor in a specific manner. Likewise, she can also greet all the others in a specific manner. And they can greet her. So, are we all set then? Well, not quite. The trouble is, someone went along and implemented the following method:

```java
/**
 * Determine a student's favorite HTA.
 * @param studentName - the name of a student
 * @return that student's favorite HTA
 */
public static Person getFavoriteHTA(String studentName) {
    if (studentName.equals("Black Panther")) {
        return new Alex();
    } else if (studentName.equals("Wonder Woman")) {
        return new Eleanor();
    } else if (studentName.equals("Wolverine")) {
        return new Liam();
    } else {
        return new Mae();
    }
}
```

The return type of this new method is, aptly, `Person`. That way, it can return any of a `Alex`, or a `Eleanor` or a `Liam`, or a `Mae`. But a difficulty arises if Alex wants to talk to someone’s favorite head TA, say `getFavoriteHTA("Wonder Woman")`. Although Wonder Woman’s favorite head TA appears to be `Eleanor`, the `getFavoriteHTA` method returns a `Person`. So, when Alex attempts to talk to Wonder Woman’s favorite head TA, the method that she will invoke is the generic `talkTo` method, which she uses to talk (completely generically) to a `Person`. And this is not what she wants to do!

So how do we fix this problem? That is, how can Alex talk specifically to Eleanor, while, at the same time, talking to `getFavoriteHTA("Wonder Woman")`, who she doesn’t know is Eleanor, but knows only is a `Person`? One pattern that is used to implement this functionality is called *double dispatch*. The way it works is: we flip things around. Since in the usual setup, Alex doesn’t know that she is talking to Eleanor, and so she can’t address her in the specific way that she wants to, we instead have Eleanor, who knows she is Eleanor, “be talked to by” Alex.

In code, what this amounts to is the following: in the subclass `Alex`, we override the inherited `talkTo` method—the one that talks to a `Person p`—so that `p`, which knows its own type (`Eleanor`), invokes its own method, which specifies that it should be talked to by Alex: e.g., `p.beGreetedByAlex()`. Then, in the subclass `Eleanor`, we define a method `beGreetedByAlex()` which includes the specific message Alex intends for Eleanor. Here’s what this looks like:

```java
// in Alex.java
@@Override
public void talkTo(Person p) {
```
Once again, since the head TAs do not know who each student’s favorite head TA is, they do not know specifically who they are talking to. But the person being talked to knows who they are. So the trick is to let the person being talked to invoke the appropriate behavior.

The double dispatch model enables a caller to defer the behavior to a callee. Since the callee knows its own type, we need only tell the callee who the caller is, which we do by calling the corresponding method. And voila! Our head TAs can interact appropriately, and specifically!

This code is in the /course/cs0180/src/shopnstop, including a well-implemented version as well as version that makes poor use of double dispatch. If you still feel shaky on double dispatch, feel free to play around with the example code to get a hang of it.

B Loggers

Following are a few tables with the methods that live in the different logger files and when each method should be used. Methods with asterisks (*) take in arguments, as specified in the respective Java files.

B.1 Log Error - LogError.java

Contains all the methods that print error messages when user enters an invalid command or value.

<table>
<thead>
<tr>
<th>Method name</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>indexNotValid</td>
<td>User enters an invalid index.</td>
</tr>
<tr>
<td>onlyRealtorsAddListings</td>
<td>Buyer tries to add listing to portal.</td>
</tr>
<tr>
<td>onlyRealtorsRemoveListings</td>
<td>Buyer tries to remove listing from portal.</td>
</tr>
<tr>
<td>invalidListingType</td>
<td>Realtor tries to create listing with invalid listing type, size, year, ratings, index.</td>
</tr>
<tr>
<td>invalidListingSize</td>
<td>Realtor tries to create listing with invalid listing type, size, year, ratings, index.</td>
</tr>
<tr>
<td>invalidListingYearBuilt</td>
<td>Realtor indicates invalid field to change from listing.</td>
</tr>
<tr>
<td>invalidListingRatings</td>
<td>Buyer tries to book dates that have already been booked.</td>
</tr>
<tr>
<td>invalidFieldChange</td>
<td>Buyer tries to book dates that have already been booked.</td>
</tr>
</tbody>
</table>
B.2 Log Prompt - LogPrompt.java

Contains all the methods that prompt both Buyers and Realtors (will expect user input).

<table>
<thead>
<tr>
<th>Method name</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>login</td>
<td>User accesses login screen.</td>
</tr>
<tr>
<td>home</td>
<td>User accesses home screen.</td>
</tr>
<tr>
<td>buyerSortingOptions</td>
<td>Buyer wishes to display listings.</td>
</tr>
<tr>
<td>realtorSortingOptions</td>
<td>Realtor wishes to display listings.</td>
</tr>
<tr>
<td>newListingType</td>
<td></td>
</tr>
<tr>
<td>newListingName</td>
<td>Realtor fills in new listing’s type, name, price, size, year built, ratings</td>
</tr>
<tr>
<td>newListingPrice</td>
<td></td>
</tr>
<tr>
<td>newListingSize</td>
<td></td>
</tr>
<tr>
<td>newListingYearBuilt</td>
<td></td>
</tr>
<tr>
<td>newListingRatings</td>
<td></td>
</tr>
<tr>
<td>removeListingIndex</td>
<td>Realtor removes listing based on index.</td>
</tr>
<tr>
<td>listingFieldChange</td>
<td>Realtor inputs which field from a listing to change.</td>
</tr>
<tr>
<td>enterDates</td>
<td>Buyer books dates for a Hotel or Rental listing.</td>
</tr>
<tr>
<td>listingIndex</td>
<td>User selects listing from index.</td>
</tr>
</tbody>
</table>

B.3 Log Listing - LogListing.java

Contains all the methods related to printing information related to displaying sorted listings.

<table>
<thead>
<tr>
<th>Method name</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>forHotelRealtor*</td>
<td>Displays following listings for Realtor: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>forRentRealtor*</td>
<td>Displays following listings for Realtor: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>forPurchaseRealtor*</td>
<td>Displays following listings for Realtor: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>forHotelBuyer*</td>
<td>Displays following listings for Buyer: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>forRentBuyer*</td>
<td>Displays following listings for Buyer: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>forPurchaseBuyer*</td>
<td>Displays following listings for Buyer: Hotel, Rent, Purchase</td>
</tr>
<tr>
<td>displayByPrice</td>
<td>Informs user that listings are being displayed in decreasing order by: price, size, year built, rating.</td>
</tr>
<tr>
<td>displayBySize</td>
<td>Informs user that listings are being displayed in decreasing order by: price, size, year built, rating.</td>
</tr>
<tr>
<td>displayByYearBuilt</td>
<td>Informs user that listings are being displayed in decreasing order by: price, size, year built, rating.</td>
</tr>
<tr>
<td>displayByRating</td>
<td>Informs user that listings are being displayed in decreasing order by: price, size, year built, rating.</td>
</tr>
<tr>
<td>displayByDefault</td>
<td>Informs user that listings are being displayed in decreasing order by default criteria, price.</td>
</tr>
</tbody>
</table>
B.4 Log Confirmation - LogConfirmation.java

Contains all the methods related to printing confirmations of purchases, rental transactions, etc.

<table>
<thead>
<tr>
<th>Method name</th>
<th>When to use</th>
</tr>
</thead>
<tbody>
<tr>
<td>purchaseComplete</td>
<td>Buyer successfully purchases a Purchase listing.</td>
</tr>
<tr>
<td>listingRemoved</td>
<td>Realtor successfully removes a listing from portal.</td>
</tr>
<tr>
<td>dateReservation*</td>
<td>Buyer successfully reserves a Hotel or Rent listing for specified dates</td>
</tr>
</tbody>
</table>

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)