Project 1: Stanley’s Storage, Part 1

1 Overview

1.1 Introduction

Stanley’s Storage offers storage units for customers to rent. Locations are dotted around the Northwest. Write supplier code to help Stanley manage his locations, customers, and storage units. This will not include UI; that is being created by another team.

1.2 Storage Locations

Stanley’s Storage locations are identified by a unique name that has two upper-case letters indicating the state, followed by two digits representing the location number within the state, followed by the city name. Examples are “WA23Issaquah” and “OR02Ashland”. Each location is identical in layout, having 12 rows, each with 20 storage units.

1.3 Storage Units

Units are one of three types: standard, humidity controlled, and temperature controlled. Attributes of interest include the unit’s width, length, and height, and the unit’s standard price. Rented units have a customer associated with them[1] as well as a price at which they were rented (which may be different from the standard price).

1.4 Customers

Each location manages its own list of customers. Assume for now that no more than 100 customers will be necessary. For each customer Stanley’s maintains a name, phone number, and account balance.

2 Client Code Requirements

Client code must be able to…

2.1 Storage Locations

- Retrieve the storage unit’s designation
- Retrieve a storage unit[2] by index
- Retrieve a customer by index
- Retrieve a count of current customers
- Add a customer to the list
- Retrieve an array of storage units rented to a specified customer[3]
- Retrieve an array of all empty storage units
- Retrieve an array of all empty storage units of a specified unit type
- Charge monthly rent (charge each customer for each of the units they are renting)

[1] Note the specific language here; units know about customers, but not the other way around, in this model.
[2] We’re in an object-oriented world; you should converse in objects, rather than strings or other data types, wherever it makes sense.
[3] Do not create more than a single array in this process; use no other data structures, either, for this method or any of the similar ones listed.

2.2 Storage Units

- Retrieve each attribute. Price should be rented price or standard price, whichever is applicable
- Retrieve the customer associated with the unit[4]
- Retrieve the rental start date[5]
- Rent the unit to a specified customer, on a specified date
- Release the unit (make it "unrented")

2.3 Customers

- Retrieve each attribute
- Update the customer's name or phone number
- Charge the customer a specified amount, or credit them a specified amount

3 Design Documentation

Before you code, create appropriate design documentation and obtain feedback. Update designs per feedback, then use them during the rest of the development process, and submit them as part of your project. For OOP projects, this will always include UML Class Diagram(s) and UML Object Diagrams; on some projects, additional documentation will be required. Recommended tool: Violet, which creates both diagram types (and others).

4 Code Implementation

4.1 Building Blocks You'll Need

- Classes and objects
- "Has a" relationships
- Arrays (2D)[6]
- Enumerated types

4.2 Preconditions

Establish preconditions for the following cases, throwing appropriate exceptions (with *helpful and informative* messages) when the preconditions are violated:

- Storage location designations must fit the specified pattern discussed above.
- Storage unit lengths and widths are always multiples of four; heights, multiples of two. Dimensions must be positive numbers greater than zero.
- Customer charges and credits must be non-negative.
- Add other preconditions that make sense and are critical to the proper operation of the objects. Consider, for example, important strings that shouldn't be null or empty, and object references that shouldn't be null.
- You should *not* throw exceptions in cases where parameter values would cause runtime errors that would be clearly understandable to client coders, e.g., array out-of-bounds errors.

[4] When the unit is not rented, we'd expect the customer and the rental date to be null.
[5] Let's be smart and use objects, not strings or integers; we're better than that, or should be, now! Use standard American short-date format, e.g., 11/07/2017 for November 7th, 2017.
[6] Only regular (rectangular) 2D arrays are to be used in this project; we'll get to jagged arrays later.

4.3 Other Requirements

- Store all storage units a single, regular (rectangular) 2D array.
- Create all storage units, and fill in their attributes, when the storage location is created; don't postpone unit creation. Make the sizes and prices whatever you want, for now.
- Create no static methods or variables. In a few cases, static *constants* might be reasonable.
- Use no class-level public variables. Class-level public *constants* are okay.
- Use method exposure (public and private) wisely; expose what Client code needs, hide internal-use methods.
- This is supplier code; no main method is part of the model. It's a good idea to create one to show off the project's capabilities, however (e.g., toString results).
- Create an enumerated type for storing the unit type.
- Write a toString method for each class. Include all data helpful in describing the object's state. For containers, include the contents, as well, and ensure the resulting strings look good when displayed.
- Within the created classes, *do not converse with the user*; there should be no input or output. It is okay for methods to return strings that will be used from client code, however, and to include exception messages which will be seen if client code breaks the rules and misuses your class. Main can converse with the user, of course.

4.4 Style

Follow the Course Style Guide, which is linked in the Reference section of the Modules list in Canvas.

5 Testing

Create a JUnit test class for each production class. Ensure that each method and state is fully tested. This must include constructors, accessors, mutators, and preconditions. You do not need to test UI-heavy methods like toString and similar.

Don't overload test methods. Test a few related methods together, perhaps (like the full constructor and related accessors); this way, failure reports will often pinpoint what failed. Test one precondition (for one method) at a time; the first triggered exception bails out of the method, so subsequent tests are skipped.

6 Hints

- Build and test each class, one at a time, starting with the most basic building block.
- Clever use of toString methods will it easier to create toString methods in container classes.
- A complete object diagram will be of great help when navigating project drill-downs.
- If you know a little about regular expressions, you'll be able to make some precondition checking code very compact and concise; check out the *matches* method on String objects.
- When testing, either test with stubbed code (function headers with no bodies and hard-coded return values), or test as if you haven't seen the code; don't test to the code you've already written and convinced yourself is right. Think about what might go wrong in the scenario. Remember that test code can contain more lines than production code. If you write your tests early, they'll benefit you as you flesh out the project; one touch of a button will run all the tests you've written up to this point.

7 Extra Credit

Within the storage location's constructor, read sample customer data from a plain-text file[7] called Customers.txt, stored in your project folder. This file (or its name) shouldn't be passed in from client code; it's the data file used for this purpose, so just read it. Put about ten customers in the file as sample data, and to use for testing and demonstration purposes. Do not hard-code any paths; if you place the file in your project folder, it should work fine with its name alone.

8 Submitting Your Work

Place copies of your final design documents in your project folder. Submit a compressed file (**.zip** or **.jar**) containing the full contents of your project folder. If creating a .jar file with BlueJ (which I highly recommend), make sure and choose "Include source."

9 Grading Matrix and Points Values

Area	**Value**	**Evaluation**
Initial design docs	5%	Did you submit initial design documents, and were they in reasonable shape to begin coding?
Final design docs	5%	Did you submit final design documents, and were they a good representation of the final version of the project?
Class implementation	25%	Were the three classes (four, with main) coded successfully? Were good coding skills used?
Enumerated type implementation	10%	Did you create an enumerated type and use it well in the project? Was it declared at the proper scope?
Array usage	20%	Did you create a 2D array and use it correctly in the project?
Preconditions implemented	10%	Did all pertinent methods have proper preconditions? Did you code preconditions that should *not* have been there?
Style/internal documentation	10%	Did you use JavaDoc notation, and use it properly? Were other elements of style (including the Style Guide) followed?
JUnit tests	15%	Were test classes created for each production class? Were all possible methods tested? Was all state tested?
Extra Credit: read customer file	2%	Did you create a file with ten customers, and read it as requested?
Total	**102%**	

Code that does not compile or run won't be graded

[7] Be sure the text editor you choose doesn't write out any adornments, e.g., RTF is not what we want. Use Notepad on Windows, or TextEdit set to Plain Text mode on the Mac.