Learning Outcomes

Upon completion of this assignment, students should be able to

- Create an object class that contains more complex data fields.
- Handle interactions between several object classes and a client class.
- Write a client class to test the object class..

Introduction

You will create a class called `Collection` to manage information about a collection consisting of `Collectible` items. The collection will include an array of `Collectible` objects, and the `Collection` class will support some simple statistics to be run on the collection. You will also write a client class to test the features of your `Collection` class.

You will use the `Collectible` class that you wrote for the previous assignment. If you did all the required implementation for that assignment, nothing should be changed in the `Collectible` class. In particular, fields must remain `private`.

Deliverables

Submission medium: Submit the following files through Brightspace:

- Source file for the `Collectible` class, named `Collectible.java`
- Source file for the `Collection` class, named `Collection.java`.
- Source file for your test client, `CollectionDriver.java`.
- `README` file.

Method

You will implement the `Collectible` class and write a `CollectionDriver` class.

`Collection` class

The `Collection` class will have the following `private` data fields:

1. `String name`: a name for the collection.
2. `Collectible[] items`: an array of `Collectible` objects. Default size of `items` is 50 (use a class constant [`static final` so that the array capacity is easy to change).
3. `int size`: the number of collectibles in the collection. NOTE: This is NOT the same as the capacity of the `items` array.

The class must have the following public methods.

- Constructors:

  - Default constructor that creates the `items` array with null `Collectible` objects and sets `size` to 0.
  - Parameterized constructor that takes a `String` parameter for the collection `name` and sets the other fields as in the default constructor.

- `setName(String name)`: sets the `name` field to the value of the incoming parameter.
- `addItem(Collectible item)`: adds a `Collectible` object to the `items` array.
- `deleteItem(int index)`: deletes the item from the given index position in the `items` array.
- `getSummary()`: returns a `String` with the following statistics about the collection:
  - Total number of items in a category such as "`Star_Trek: 14; Space: 10; Tolkien: 18`".
  - Total value for the collection, formatted to exactly 2 places to the right of the decimal (such as "`Total value: $2265.74`").
  - The final return value for the above example should look like:

```
Star_Trek: 14; Space: 10; Tolkien: 18
Total value: $4365.74
```

- `printReport()`. This method will print a report containing the following, formatted in human-friendly columns:
  - An item-by-item list of the collection.
  - Summary for each category in the collection.
  - Collection summary, total number of items and total value.
  - Your output must look like this sample (slightly different spacing is fine, but all the information must be included and arranged in columns):

```
Item Type    Name        Category     Value   Quantity
----------   -----       --------     -----   --------
FunkoPop     Spock       Star_Trek    $12.95  1
Model        Enterprise  Star_Trek    $39.95  1
. . .
Book         Hobbit      Tolkien      $23.95  2

SUMMARY BY CATEGORY:
Category   Number Items  Total Value
--------   ------------  -----------
Star_Trek       14         $1526.98
Space           10          $365.98
Tolkien         18         $2472.78

SUMMARY TOTALS:
Number of items: 42
Total value:     $2265.74
```

Notes on the sample output:

    * The ". . ." means that there are more items in the list. I have shown only a few items in the example. You must print ALL the items in the collection.
    * There could be more categories than the 3 shown in the example.
    * The summary statistics given in the example are arbitrary. They do not represent true statistics from input files.

- `toString()`: returns a `String` representation of the collection's name and size, for example: "`Awesome Collection, 27 items`". If there is no name and nothing has been added to the collection, output should look like: "`null, 0 items`".

See "Design Considerations" below for some reminders and tips.

`CollectionDriver` class

You must write your own test client for the program. Keep in mind that I will have my own test client that I will use to test your code.

Your client file must do the following tasks:

- Handle file input. The input file name must be read from command line, not entered interactively.
- Test all functions of the `Collection` class. Testing must include:
    1. Both constructors: create an object with the default constructor, and another one with the parameterized constructor.
    2. `addItem`, with data from the input file used to create `Collectible` objects to add to the collection. Note that your client must correctly handle any exceptions (see Input/Output for more details of exceptions/errors that must be handled).
    3. `deleteItem` with several calls to remove items. Ensure that the size of the collection is properly updated.
    4. `getSummary`.
    5. `printReport`.

    For all tests, how will you determine if your classes are working correctly?

Input/Output

User Interface

The program runs without user input once launched . The name of the input file will be entered by the user at the command line when the program is run, for example:

```
java CollectionDriver itemsInfo1.txt
```

See the previous assignment for more about handling the command line argument.

File Formats

Input files are the same as the input files in the previous assignment. I have repeated the details about the input files here for convenience.

The files read by this program are **`text`** files with the information for one `Collectible` object on each line. The tokens are separated by whitespace; you may assume there is no whitespace inside the `String` tokens. The data appears in this order in the line:

```
<type> <name> <description> <category> <number> <value>
```

Here is a sample of part of an input file:

```
FunkoPop Spock Spock_with_cat Star_Trek 1 12.95
Model Enterprise TOS1701 Star_Trek 1 39.95
ActionFigure DarthVader Star_Wars 1977_12inch 1 130.59
ModelKit SaturnV Revell_Big_SaturnV Space 1 215.00
Book Hobbit Illustrated_JCatlin Tolkien 2 23.95
...
```

The "..." in the above example file stand for some number of additional lines of input. You will not know in advance how many items are in each input file. Sample data files are found in the `git` repository `a05Classes2` in the course Gitea organization. Data files are of differing lengths. Be sure to test your code with all the provided data files. I will test your code with data files that you have not seen that are of different lengths than the provided test files. You do not have to handle a situation where the input file has more items than the value given by the class constant for the default array size of `items`.

Design Considerations

You need to write `private` helper methods in the `Collection` class to compute the statistics. It is not good design to have the calculations performed inside the `printReport` method; keep your code as functionally modular as possible. These helper methods should be given `private` access so that only methods inside the `Collection` object can access them. They should never be called from a client class. Neither `printReport` nor `toString` should have loops to count the number of items in the collection or in a category or to add values.

When calculating statistics, remember that there may be some items with quantity of more than 1. Be sure that the value and number statistics use the correct number for each item.

Exception/Error Handling

Your client code must correctly handle exceptions related to input files. These exceptions are:

- `FileNotFoundException` if the given input file cannot be found.
- Any I/O error while reading the input file.
- Any parse error from the `Scanner` while parsing the input file data.

Exceptions MUST be handled with `try/catch` blocks.

Implementation Details

For this assignment you must use an array of `Collectible` objects in `Collection` (you may not use anything like a `ArrayList` or any ready-made data structure). No variables may be global in the client class.

Documentation

`README`

Must include instructions on how to **compile** and how to **run** the program as submitted. Please refer to the README document in Brightspace.

Code style and commenting

Code style and comments must meet the department standards as described in the code style document available in Brightspace. Be particularly careful to include required method heading comments on ALL methods in the program.

Submission

Submit the 4 deliverables in Brightspace by the assignment deadline.

- Source files:
  - Client class, `CollectionDriver.java`.
  - Object class, `Collectible.java`.
  - Object class, `Collection.java`.
- `README` file.