Project: Bunnies – Part 2

Deliverables

Your project files should be submitted for grading by the due date and time specified. Note that there is also an optional Skeleton Code assignment (ungraded) which will indicate level of coverage your tests have achieved. The files you submit to skeleton code assignment may be incomplete in the sense that method bodies have at least a return statement if applicable or they may be essentially completed files. In order to avoid a late penalty for the project, you must submit your files to the Completed Code assignment no later than 11:59 PM on the due date. If you are unable to submit to the grading system, you should e-mail your project Java files in a zip file to your TA before the deadline. Your grade will be determined, in part, by the tests that you pass or fail in your test files and by the level of coverage attained in your source files, as well as our usual correctness tests.

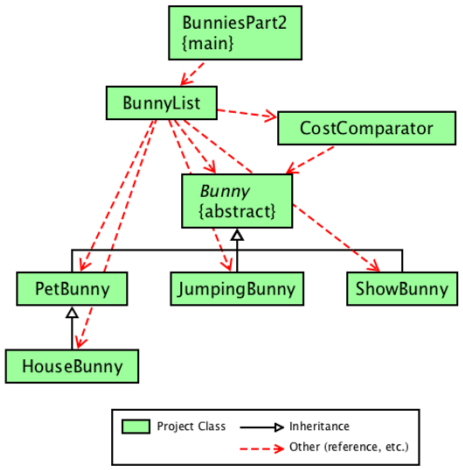
Files to submit for grading:

From Bunnies – Part 1

- Bunny.java
- PetBunny.java, PetBunnyTest.java
- HouseBunny.java, HouseBunnyTest.java
- JumpingBunny.java, JumpingBunnyTest.java
- ShowBunny.java, ShowBunnyTest.java

New in Bunnies – Part 2

- BunnyList.java, BunnyListTest.java
- CostComparator.java, CostComparatorTest.java
- BunniesPart2.java, BunniesPart2Test.java



Recommendations

You should create new folder for Part 2 and copy your relevant Part 1 source and test files (listed above) to it (i.e., do not include BunniesPart1.java, BunniesPart1Test.java). You should create a new jGRASP project and add these source and test files as well as new ones as they are created. You may find it helpful to use the "viewer canvas" feature as you develop and debug your program.

Specifications

Overview: This project is Part 2 of three that that will involve calculating the estimated monthly cost of owning a bunny where the amount is based on the type of bunny, its weight, and various additional costs. In Part 1, you developed Java classes that represent categories of bunny: pet bunny, house bunny (a subclass of pet bunny), jumping bunny, and show bunny. In Part 2, you will implement three additional classes: (1) CostComparator that implements the Comparator interface, (2) BunnyList that represents a list of bunnies and includes several specialized methods, and (3) BunniesPart2 which contains the main method for the program. Note that the main method in BunniesPart2 should create a BunnyList object, read the data file using the readBunnyFile method. BunniesPart2 then prints the summary, the bunnies listed by name and the bunnies listed by estimated monthly cost, and the list of excluded records. You can use BunniesPart2 in conjunction with interactions by running the program in the canvas (or debugger with breakpoints) until the BunnyList object has been created and the data has been read in. You can then enter interactions in the usual way. You can also step into the methods of interest when you run "Debug". In addition to the source files, you will create a JUnit test

file for each class and write one or more test methods to ensure the classes and methods meet the specifications.

- **Bunny, PetBunny, HouseBunny, JumpingBunny, and ShowBunny**

  Requirements and Design: No changes from the specifications in Bunnies – Part 1.

- **BunnyList.java**

  Requirements: The BunnyList class provides methods for reading in data and generating reports (summary and list), adding a bunny, and sorting the bunnies by name and by estimated monthly cost).

  Design: The BunnyList class has fields, a constructor, and methods as outlined below.

(1) **Fields:** All fields below should be private and initialized in the constructor.
(a) *listName* is type String.
(b) *bunnyList* is type Bunny array.
(c) *excludedRecords* is type String array.
Note that the bunny array and excluded records array should grow as items are added. Hence, the length of these arrays should be the same as the number of objects in the arrays. This eliminates the need for separate variables to track the number of objects contained in the arrays, and it also allows the use of for-each loops with the arrays.

(2) **Constructor:** The constructor has no parameters and initializes the fields as follows: listName is initialized to "not yet assigned", bunnyList is initialized to a Bunny array with length zero, and excludedRecords is initialized to a String array with length zero.

(3) **Methods:** Usually a class provides methods to access and modify each of its instance variables (i.e., getters and setters) along with any other required methods. The methods for BunnyList are described below.

- `readBunnyFile` has no return value, accepts the data file name as a String, and has a throws clause for FileNotFoundException. This method creates a Scanner object to read in the file and then reads it in line by line. The first line of the file contains the name of the list, and each of the remaining lines contains the data for a bunny. After reading in the list name, the "bunny" lines should be processed as follows. A bunny line (or record) is read in, a second Scanner is created on the line, and the individual values for the bunny are read in. Be sure to "trim" each value read in. All values should be read as strings. Non-String values should be "parsed" into their respective values using the appropriate wrapper class (e.g., Double.parseDouble(..)). After the values on the line have been read in, an "appropriate" bunny object is created and added to the bunny array using the addBunny method. If the bunny type is not recognized, the record/line should be added to the excluded records array using the addExcludedRecord method. The data file is a "comma separated values" file; i.e., if a line contains multiple values, the values are delimited by commas. So after you set up the Scanner for the bunny lines, you need to change the delimiter to a "," by invoking `useDelimiter(",")` on the Scanner object. Each bunny line in the file begins with a category for the bunny. Your switch statement

should determine which type of Bunny to create based on the first character of the category (i.e., P, H, J, and S for PetBunny, HouseBunny, JumpingBunny, and ShowBunny respectively, ignoring case). The second field in the record is the name, followed by breed, and weight, as well the values appropriate for the category of bunny represented by the line of data. That is, the items that follow weight correspond to the data needed for the particular category (or subclass) of Bunny. An example file, *bunnies1.txt*, is available for download from the course web site. Below are example data records (the first line/record containing the bunny list name is followed by bunny lines/records). Note that two of the records below have invalid categories.

```
Bunny Collection
Pet bunny, Floppy, Holland Lop, 3.5
house Bunny, Spot, Really Mixed, 5.8, 0.15
mouse Bunny, Spots, Mixed, 0.8, 0.15
Jumper Bunny, Speedy, English, 6.3, 25.0
fighting bunny, Slugger, Big Mixed, 16.5, 21.0
Show bunny, Bigun, Flemish Giant, 14.6, 22.0
```

- `getListName` returns the String representing the list name field.
- `setListName` returns nothing, accepts a String and assigns it to list name field.
- `getBunnyList` returns the array containing the Bunny objects.
- `getExcludedRecords` returns the String array representing the excluded records.
- `addBunny` has no return value, accepts a Bunny object (e.g., bunnyIn), increases the capacity of the bunny array by one, and adds the bunny in the last position of the bunnies array. The following two lines accomplish this (assuming `bunnyList` is the bunny array and that `java.util.Arrays` has been imported).
  ```
  bunnyList = Arrays.copyOf(bunnyList, bunnyList.length + 1);
  bunnyList[bunnyList.length - 1] = bunnyIn;
  ```
- `addExcludedRecord` has no return value, accepts a String, increases the capacity of the excludedRecords array by one, and adds the String in the last position of the excludedRecords array. (hint: see code in `addBunny` above)
- `toString` returns a String representing a list of bunnies in the bunny array (<u>does not include a list title</u>); accepts no parameters. A `\n` should be added before and after each Bunny object.
- `totalEstimatedMonthlyCost` returns a double representing the total estimated monthly cost for all of the bunnies in the bunny array.
- `summary` returns a String representing summary information for the bunny list. It includes the list name, the total number of the bunnies, and total estimated monthly cost for the bunnies. Note that this method should call the totalEstimatedMonthlyCost method described above to get the total estimated monthly cost, and it should end with a `\n` character. See example output below.
- `listByName` returns a String representing the bunny list by name (the natural sorting order). The bunny array should be sorted by name before building the String to be returned. The resulting String should include the title and list of bunnies as shown in the example output below. The title should <u>not</u> be preceded by `\n`. Recall, the `toString` method returns the list of bunnies.
- `listByCost` returns a String representing the bunny list by the estimated monthly cost. The bunny array should be sorted by cost (see the CostComparator class below) before building the String to be returned. The resulting String should include the title and list of bunnies as shown in the example output below. The title should <u>not</u> be preceded by `\n`.

- `excludedRecordsList` returns a String representing the list of bunny records/lines that were read from the file but excluded from the bunny array in BunnyList. The resulting String should include the title and list of excluded records/lines as shown in the example output below.

Code and Test: See examples of file reading and sorting in the lecture notes. The Arrays.sort method in the java.util package sorts the array in place. The natural sorting order for Bunny objects is determined by the compareTo method from the Comparable interface. If *bunnyList* is the variable for the array of Bunny objects, it can be sorted in natural order with the following statement.

```
Arrays.sort(bunnyList);
```

The sorting order based on estimated monthly cost is determined by the CostComparator class which implements the Comparator interface (described below). It can be sorted with the following statement.

```
Arrays.sort(bunnyList, new CostComparator());
```

After the bunny array is sorted, the array returned by the getBunnyList method should be in the order resulting from the most recent sort.

- **CostComparator.java**

Requirements and Design: The CostComparator class implements the Comparator interface for Bunny objects. Hence, it implements the following method.

- `compare(Bunny b1, Bunny b2)` that defines the ordering from lowest to highest based on the estimated monthly cost for b1 and b2.

  Note that the *compare* method is the only method in the CostComparator class. An instance of this class should be used as one of the parameters when the Arrays.sort method is used to sort by "estimated monthly cost" (see above). For an example of a class implementing Comparator, see lecture notes on Comparing Objects.

- **BunniesPart2.java**

Requirements and Design: The BunniesPart2 class has only a main method as described below.

- `main` gets the file name from the command line (i.e., args[0]), creates an instance of BunnyList, and then calls its readBunnyFile method to read in the data file and populate the bunny array in BunnyList object. The main method then prints the summary, the bunny list by name, the bunny list by estimated monthly cost, and the list of excluded records. After the summary is printed, be sure to print `\n` characters as needed before each of the three lists is printed. An example data file, *bunnies1.txt*, can be downloaded from the Lab web page. The output from main for this file is on the following page. Note that main should have a throws clause for FileNotFoundException.

Code and Test: The example data file, bunnies1.txt, has been uploaded into the grading system and is available for your test methods to call as needed. If you want to use additional data files, you will need to use .txt as the extension and then upload the data files along with your source

files. After you have implemented the BunniesPart2 class, you should create the test file BunniesPart2Test.java in the usual way. The only test method you need is one that creates an instance of BunniesPart2 (to cover its default constructor), and then checks the class variable *bunnyCount* that was declared in Bunny and inherited by each subclass. In the test method, you should declare and create an instance of BunniesPart2, reset bunny*Count*, create an args array containing the file name bunnies1.txt, call your main method in BunniesPart2, which should result in bunnies1.txt being read in, then assert that bunny*Count* is four (assuming that four objects from the Bunny hierarchy were created and stored in the BunnyList object created when main is called). The following statements accomplish the test.

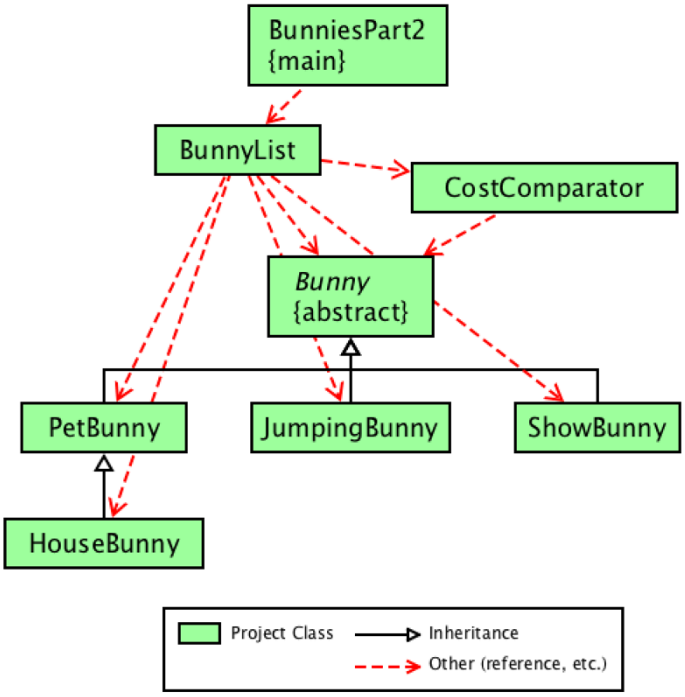
```
BunniesPart2 bPart2Obj = new BunniesPart2(); // test constructor

Bunny.resetBunnyCount();

String[] args = {"bunnies1.txt"};
BunniesPart2.main(args);
Assert.assertEquals(4, Bunny.getBunnyCount());
```

UML Class Diagram

After you have added your classes to the jGRASP project, you should generate the UML class diagram for the project. To layout the UML class diagram, right-click in the UML window and then click Layout > Tree Down. Click in the background to unselect the classes. You can then select the BunniesPart1 class and move it around as appropriate, then do the same for the BunnyList and CostComparator. Note that the dependencies represented by the red dashed arrows indicate that BunniePart1depends on BunnyList which in turn depends on CostComparator, Bunny, and Bunny's subclasses. Note that CostComparator only depends on Bunny.



Example Output

```
 ----jGRASP exec: java BunniesPart2 bunnies1.txt
------------------------------
Summary for Bunny Collection
------------------------------
Number of Bunnies: 4
Total Estimated Monthly Cost: $124.38


------------------------------
Bunnies by Name
------------------------------

Bigun (ShowBunny)   Breed: Flemish Giant   Weight: 14.6
Estimated Monthly Cost: $62.15 (includes $22.00 for grooming)

Floppy (PetBunny)   Breed: Holland Lop   Weight: 3.5
Estimated Monthly Cost: $6.48

Speedy (JumpingBunny)   Breed: English   Weight: 6.3
Estimated Monthly Cost: $40.75 (includes $25.00 for training)

Spot (HouseBunny)   Breed: Really Mixed   Weight: 5.8
Estimated Monthly Cost: $15.01 (includes 15.0% for wear and tear)


------------------------------
Bunnies by Cost
------------------------------

Floppy (PetBunny)   Breed: Holland Lop   Weight: 3.5
Estimated Monthly Cost: $6.48

Spot (HouseBunny)   Breed: Really Mixed   Weight: 5.8
Estimated Monthly Cost: $15.01 (includes 15.0% for wear and tear)

Speedy (JumpingBunny)   Breed: English   Weight: 6.3
Estimated Monthly Cost: $40.75 (includes $25.00 for training)

Bigun (ShowBunny)   Breed: Flemish Giant   Weight: 14.6
Estimated Monthly Cost: $62.15 (includes $22.00 for grooming)


------------------------------
Excluded Records
------------------------------

mouse Bunny, Spots, Mixed, 0.8, 0.15

fighting bunny, Slugger, Big Mixed, 16.5, 21.0


 ----jGRASP: operation complete.
```