Project 2

Introduction - the SeaPort Project series

For this set of projects for the course, we wish to simulate some of the aspects of a number of Sea Ports.

Here are the classes and their instance variables we wish to define:

- SeaPortProgram extends JFrame
  - variables used by the GUI interface
  - world: World
- Thing implement Comparable <Thing>
  - index: int
  - name: String
  - parent: int
- World extends Thing
  - ports: ArrayList <SeaPort>
  - time: PortTime
- SeaPort extends Thing
  - docks: ArrayList <Dock>
  - que: ArrayList <Ship> // the list of ships waiting to dock
  - ships: ArrayList <Ship> // a list of all the ships at this port
  - persons: ArrayList <Person> // people with skills at this port
- Dock extends Thing
  - ship: Ship
- Ship extends Thing
  - arrivalTime, dockTime: PortTime
  - draft, length, weight, width: double
  - jobs: ArrayList <Job>
- PassengerShip extends Ship
  - numberOfOccupiedRooms: int
  - numberOfPassengers: int
  - numberOfRooms: int
- CargoShip extends Ship
  - cargoValue: double
  - cargoVolume: double
  - cargoWeight: double
- Person extends Thing
  - skill: String
- Job extends Thing - optional till Projects 3 and 4
  - duration: double
  - requirements: ArrayList <String>
    // should be some of the skills of the persons
- PortTime
  - time: int

Eventually, in Projects 3 and 4, you will be asked to show the progress of the jobs using JProgressBar's.

Here's a very quick overview of all projects:

1. Read a data file, create the internal data structure, create a GUI to display the structure, and let the user search the structure.
2. Sort the structure, use hash maps to create the structure more efficiently.
3. Create a thread for each job, cannot run until a ship has a dock, create a GUI to show the progress of each job.
4. Simulate competing for resources (persons with particular skills) for each job.

Project 2 General Objectives

Project 2 - Map class, Comparator, sorting

- Use the JDK Map class to write more efficient code when constructing the internal data structures from the data file.
- Implement SORTING using the Comparator interface together with the JDK support for sorting data structures, thus sorting on different fields of the classes from Project 1.
- Extend the GUI from Project 1 to let the user sort the data at run-time.

Documentation Requirements:

You should start working on a documentation file before you do anything else with these projects, and fill in items as you go along. Leaving the documentation until the project is finished is not a good idea for any number of reasons.

The documentation should include the following (graded) elements:

- Cover page (including name, date, project, your class information)
- Design
  - including a UML class diagram
  - classes, variables and methods: what they mean and why they are there
  - tied to the requirements of the project
- User's Guide
  - how would a user start and run your project
  - any special features
  - effective screen shots are welcome, but don't overdo this
- Test Plan
  - do this BEFORE you code anything
  - what do you EXPECT the project to do
  - justification for various data files, for example
- Lessons Learned
  - express yourself here
  - a way to keep good memories of successes after hard work

Project 2 Specific Goals:

Extend Project 1 to use advanced data structures and support sorting on various keys.

1. Required data structure - the data structure specified in Project 1:
    a. World has SeaPort's
    b. SeaPort has Dock's, Ship's, and Person's
    c. Dock has a Ship
    d. Ship has Job's
    e. PassengerShip
    f. CargoShip
    g. Person has a skill
    h. Job requires skills - optional until Project 3
    i. PortTime
2. Use the HashMap class to support efficient linking of the classes used in Project 1.
    1. The instances of the hash map class should be local to the readFile (Scanner) method.
    2. These instances should be passed as explicit parameters to other methods used when reading the data file.
        1. For example, the body of the methods like the following should be replaced to effectively use a <Integer, Ship> hash map, the surrounding code needs to support this structure:
           ```
           Ship getShipByIndex (int x, java.util.HashMap <Integer, Ship> hms) {
             return hms.get(x);
           } // end getDockByIndex
           ```
        2. Since the body of this method has become trivial, perhaps the call to this method can be simply replaced by the get method of the HashMap.
        3. Your code should be sure to handle a null return from this call gracefully.
    3. The instances should be released (go out of scope, hence available for garbage collection) when the readFile method returns.
    4. Comments: The idea here, besides getting some experience with an interesting JDK Collections class, is to change the operation of searching for an item with a particular index from an O(N) operation, ie searching through the entire data structure to see if the code can find the parent index parameter, to an O(1) operation, a hash map lookup. Of course, this isn't so very interesting in such a small program, but consider what might happen with hundreds of ports, thousands of ships, and perhaps millions of persons and jobs.
    5. Comments: Also, after the readFile operation, the indices are no longer interesting, and could be completely eliminated from the program. In this program, removing the index references could be accomplished by removing those variables from the parent class, Thing.
3. Implement comparators to support sorting:
    - ships in port que ArrayList's by weight, length, width, draft within their port que
    - all items withing their ArrayList's by name
    - OPTIONALLY: sorting by any other field that can be compared
    - The sorting should be within the parent ArrayList
4. Extend the GUI from Project 1 to allow the user to:
    - sort by the comparators defined above.
5. Again, the GUI elements should be distinct from the other classes in the program.

Deliverables:

1. Java source code files
2. Data files used to test your program
3. Configuration files used
4. A well-written document including the following sections:
    a. Design: including a UML class diagram showing the type of the class relationships
    b. User's Guide: description of how to set up and run your application
    c. Test Plan: sample input and ***expected*** results, and including test data and results, with screen snapshots of some of your test cases
    d. Optionally, Comments: design strengths and limitations, and suggestions for future improvement and alternative approaches
    e. Lessons Learned
    f. Use one of the following formats: MS Word docx or PDF.

Your project is due by midnight, EST, on the day of the date posted in the class schedule. We do not recommend staying up all night working on your project - it is so very easy to really mess up a project at the last minute by working when one was overly tired.

Your instructor's policy on late projects applies to this project.

Submitted projects that show evidence of plagiarism will be handled in accordance with UMUC Policy 150.25 — Academic Dishonesty and Plagiarism.

Format:

The documentation describing and reflecting on your design and approach should be written using Microsoft Word or PDF, and should be of reasonable length. The font size should be 12 point. The page margins should be one inch. The paragraphs should be double spaced. All figures, tables, equations, and references should be properly labeled and formatted using APA style.

Coding Hints:

- Code format: (See Google Java Style guide for specifics (https://google.github.io/styleguide/javaguide.html))
    - header comment block, including the following information in each source code file:
    - file name
    - date
    - author
    - purpose
    - appropriate comments within the code
    - appropriate variable and function names
    - correct indentation
- Errors:
    - code submitted should have no compilation or run-time errors
- Warnings:
    - Your program should have no warnings

- Use the following compiler flag to show all warnings:
  javac -Xlint *.java
- More about setting up IDE's to show warnings
- Generics - your code should use generic declarations appropriately, and to eliminate all warnings

- Elegance:
  - just the right amount of code
  - effective use of existing classes in the JDK
  - effective use of the class hierarchy, including features related to polymorphism.
- GUI notes:
  - GUI should resize nicely
  - DO NOT use the GUI editor/generators in an IDE (integrated development environment, such as Netbeans and Eclipse)
  - Do use JPanel, JFrame, JTextArea, JTextField, JButton, JLabel, JScrollPane
    - panels on panels gives even more control of the display during resizing
    - JTable and/or JTree for Projects 2, 3 and 4
    - Font using the following gives a nicer display for this program, setting for the JTextArea jta:
      ```
      jta.setFont (new java.awt.Font ("Monospaced", 0, 12));
      ```
  - GridLayout and BorderLayout - FlowLayout rarely resizes nicely
    - GridBagLayout for extreme control over the displays
    - you may wish to explore other layout managers
  - ActionListener, ActionEvent - responding to JButton events
    - Starting with JDK 8, lambda expression make defining listeners MUCH simpler. See the example below, with jbr (read), jbd (display) and jbs (search) three different JButtons.
      jcb is a JComboBox <String> and jtf is a JTextField.
      ```
         jbr.addActionListener (e -> readFile());
         jbd.addActionListener (e -> displayCave ());
         jbs.addActionListener (e -> search ((String)(jcb.getSelectedItem()),
      jtf.getText()));
      ```
  - JFileChooser - select data file at run time
  - JSplitPane - optional, but gives user even more control over display panels

Grading Rubric:

Attribute	**Meets**	**Does not meet**
Design	**20 points** Contains just the right amount of code. Uses existing classes in the JDK effectively.	**0 points** Does not contain just the right amount of code. Does not use existing classes in the JDK effectively.

	Effectively uses of the class hierarchy, including features related to polymorphism. The instances of the hash map class should be local to the readFile (Scanner) method. These instances should be passed as explicit parameters to other methods used when reading the data file. GUI elements should be distinct from the other classes in the program	Does not effectively use of the class hierarchy, including features related to polymorphism. The instances of the hash map class are not local to the readFile (Scanner) method. These instances are not passed as explicit parameters to other methods used when reading the data file. GUI elements are not distinct from the other classes in the program.
Functionality	**40 points** Contains no coding errors. Contains no compile warnings. Builds from Project 1. Includes all required data structures specified in Project 1. Uses the HashMap class to support efficient linking of the classes used in Project 1. Implements comparators to support sorting. Sorting should be within the parent ArrayList. Extends the GUI from Project 1 to allow the user to sort by the comparators.	**0 points** Contains coding errors. Contains compile warnings. Does not build from Project 1. Does not include all required data structures specified in Project 1. Does not use the HashMap class to support efficient linking of the classes used in Project 1. Does not implement comparators to support sorting. Sorting is not within the parent ArrayList. Does not extend the GUI from Project 1 to allow the user to sort by the comparators.
Test Data	**20 points** Tests the application using multiple and varied test cases.	**0 points** Does not test the application using multiple and varied test cases.
Documentation and submission	**15 points** Source code files include header comment block, including file name, date, author, purpose,	**0 points** Source code files do not include header comment block, or include file name, date, author, purpose, appropriate

	appropriate comments within the code, appropriate variable and function names, correct indentation. Submission includes Java source code files, Data files used to test your program, Configuration files used. Documentation includes a UML class diagram showing the type of the class relationships. Documentation includes a user's Guide describing of how to set up and run your application. Documentation includes a test plan with sample input and ***expected*** results, test data and results and screen snapshots of some of your test cases. Documentation includes Lessons learned. Documentation is in an acceptable format.	comments within the code, appropriate variable and function names, correct indentation. Submission does not include Java source code files, Data files used to test your program, Configuration files used. Documentation does not include a UML class diagram showing the type of the class relationships. Documentation does not include a user's Guide describing of how to set up and run your application. Documentation does not include a test plan with sample input and ***expected*** results, test data and results and screen snapshots of some of your test cases. Documentation does not include Lessons learned. Documentation is not in an acceptable format.
Documentation form, grammar and spelling	**5 points** Document is well-organized. The font size should be 12 point. The page margins should be one inch. The paragraphs should be double spaced. All figures, tables, equations, and references should be properly labeled and formatted using APA style.	**0 points** Document is not well-organized. The font size is not 12 point. The page margins are not one inch. The paragraphs are not double spaced. All figures, tables, equations, and references are not properly labeled or formatted using APA style. The document should contains many spelling and grammatical errors.

	The document should contain minimal spelling and grammatical errors.	