Introduction - the SeaPort Project series

For this set of project, 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 the 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 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.

General Objectives

Here are some notes about the projects, the particular features of object-oriented design and object-oriented programming (OOD/OOP) the we want to cover in this class and some of the features of Java to help support that style of programming. We also want to explore the Java GUI system a little, with particular emphasis on viewing the data structures and effective ways to display the running of multiple threads competing for resources.

The particular scenarios selected for each semester try ask you to implement as many of these objectives as possible in some compelling way. We are always open to additions and suggestions.

General objects for each project:

- Project 1 - classes, text data file, GUI, searching
  - Define and implement appropriate classes, including:
    - instance and class variables,
    - constructors,
    - toString methods, and
    - other appropriate methods.
  - Read data from a text file:
    - specified at run time,
      - `JFileChooser jfc = new JFileChooser (".");`
        // start at dot, the current directory
    - using that data to create instances of the classes,
    - creating a multi-tree (class instances related in hierarchical, has-some, relationships), and
    - organizing those instances in existing JDK structures which can be sorted, such as ArrayList's.
  - Create a simple GUI:
    - presenting the data in the structures with with some buttons and
    - text fields supporting SEARCHING on the various fields of each class.
- 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.
- Project 3 - More JDK classes - GUI's and threads
    - Explore other GUI classes, such as JTree, JTable, and JProgressBar.
    - Create and run threads
        - Competing for one resource.
- Project 4 - Concurrency
    - Resource pools
        - Threads competing for multiple resources
    - Blocking threads
    - Extending the GUI interface to visualize the resource pools and progress of the various threads.

Documentation

HINT: 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

CAUTION: the file size should be reasonable, currently < 5 MBytes, mostly < 1 MB.

Project 1 - Also see Grading

The goals are:

1. Create a GUI
2. Let the user select a data file, using JFileChooser
3. Read the data file, creating the specified internal data structure (see the Introduction for the classes and variables of the structure).
4. Display the internal data structure in a nice format in the GUI
    1. use JScrollPane and JTextArea
5. Display the results of a Search specified by the user
    1. JTextField to specify the search target
    2. Searching targets: name, index, skill would be a minimum
       you are encouraged to provide other options
    3. Note that a search may return more than one item
    4. DO NOT create new data structures (beyond the specified internal data structure) to search
       you may create a structure of found items as a return value

Data file format:

- Each item in the simulation will appear on a single line of the data file.
- The data file may have comment lines, which will start with a //.
- There may be blank lines in the data file, which your program should accept and ignore.
- The data lines will start with one of the following flag values, indicating which item is being specified, its name, its index, and the index of its parent - which is used to specify the connections used to create the internal data structure (ie, assign an item to it parent or parent ArrayList).
- For most items there will be additional fields appropriate to the class of that item.
- The fields on a line are space delimited (perhaps more than one space)
    - This works well with Scanner methods, such as `next()`, `nextInt()` and `nextDouble()`.

Here are the details of valid lines, with an example of each line.

```
// port   name index parent(null)
//    port   <string> <int> <int>
port Kandahar 10002 0

// dock   name index parent(port)
//    dock   <string> <int> <int>
  dock Pier_5 20005 10001 30005

// ship   name index parent(dock/port) weight length width draft
//    ship   <string> <int> <int> <double> <double> <double>
<double>
    ship               Reason 40003 10000 165.91 447.15 85.83
27.07
```

```
// cship  name index parent(dock/port) weight length width draft
cargoWeight cargoVolume cargoValue
//    cship  <string> <int> <int> <double> <double> <double>
<double> <double> <double> <double>
    cship               Suites 40003 10000 165.91 447.15 85.83
27.07 125.09 176.80 857.43

// pship  name index parent(dock/port) weight length width draft
numPassengers numRooms numOccupied
//    pship  <string> <int> <int> <double> <double> <double>
<double> <int> <int> <int>
    pship      "ZZZ_Hysterics" 30002 20002 103.71 327.92 56.43
30.23 3212 917 917

// person name index parent skill
//    person <string> <int> <int> <string>
    person              Alberto 50013 10001 cleaner

// job    name index parent duration [skill]* (zero or more,
matches skill in person, may repeat)
//    job    <string> <int> <int> <double> [<string>]* (ie, zero
or more)
    job         Job_10_94_27 60020 30007 77.78 carpenter cleaner
clerk
```

You may assume that the data file is correctly formatted and that the parent links exist and are encountered in the data file as item indices before they are referenced as parent links.

There is a Java program (CreateSeaPortDataFile.java) provided with this package that will generate data files with various characteristics with the correct format. You should be using the program to generate your own data files to test various aspects of your project programs.

Sample input file:

```
// File: aSPaa.txt
// Data file for SeaPort projects
// Date: Sat Jul 09 22:51:16 EDT 2016
// parameters: 1 1 5 5 1 5
//   ports, docks, pships, cships, jobs, persons

// port   name index parent(null)
//    port   <string> <int> <int>
port Lanshan 10000 0

// dock   name index parent(port)
//    dock   <string> <int> <int>
  dock Pier_4 20004 10000 30004
  dock Pier_0 20000 10000 30000
  dock Pier_1 20001 10000 30001
  dock Pier_3 20003 10000 30003
  dock Pier_2 20002 10000 30002

// pship  name index parent(dock/port) weight length width draft
numPassengers numRooms numOccupied
//    pship  <string> <int> <int> <double> <double> <double>
<double> <int> <int> <int>
    pship           Gallinules 30000 20000 125.99 234.70 60.67
37.14 746 246 246
    pship               Remora 30001 20001 126.38 358.27 74.12
31.54 3768 979 979
    pship     Absentmindedness 30004 20004 86.74 450.43 33.13
41.67 2143 920 920
    pship        Preanesthetic 30003 20003 149.85 483.92 125.71
31.21 166 409 83
    pship            Shoetrees 30002 20002 134.41 156.96 120.31
35.20 1673 633 633

// cship  name index parent(dock/port) weight length width draft
cargoWeight cargoVolume cargoValue
//    cship  <string> <int> <int> <double> <double> <double>
<double> <double> <double> <double>
    cship            Erosional 40001 10000 200.80 242.33 38.31
23.49 172.73 188.54 235.57
    cship            Kielbasas 40000 10000 120.85 362.55 96.82
19.09 33.08 188.31 261.57
    cship             Generics 40002 10000 79.90 234.26 73.18
15.71 125.27 179.00 729.95
    cship            Barcelona 40003 10000 219.92 443.54 104.44
34.16 86.69 139.89 813.72
    cship              Toluene 40004 10000 189.12 448.99 73.97
37.67 88.90 175.03 1002.63

// person name index parent skill
//    person <string> <int> <int> <string>
    person                 Sara 50000 10000 electrician
    person                Duane 50002 10000 inspector
    person                Betsy 50004 10000 cleaner
    person               Archie 50003 10000 captain
    person               Thomas 50001 10000 clerk
```

Sample output as plain text - which should be displayed in a JTextArea on a JScrollPane in the BorderLayout.CENTER area of a JFrame:

```
>>>>> The world:


SeaPort: Lanshan 10000

  Dock: Pier_4 20004
    Ship: Passenger ship: Absentmindedness 30004

  Dock: Pier_0 20000
    Ship: Passenger ship: Gallinules 30000

  Dock: Pier_1 20001
    Ship: Passenger ship: Remora 30001

  Dock: Pier_3 20003
    Ship: Passenger ship: Preanesthetic 30003

  Dock: Pier_2 20002
    Ship: Passenger ship: Shoetrees 30002

 --- List of all ships in que:
   > Cargo Ship: Erosional 40001
   > Cargo Ship: Kielbasas 40000
   > Cargo Ship: Generics 40002
   > Cargo Ship: Barcelona 40003
   > Cargo Ship: Toluene 40004

 --- List of all ships:
   > Passenger ship: Gallinules 30000
   > Passenger ship: Remora 30001
   > Passenger ship: Absentmindedness 30004
   > Passenger ship: Preanesthetic 30003
   > Passenger ship: Shoetrees 30002
   > Cargo Ship: Erosional 40001
   > Cargo Ship: Kielbasas 40000
   > Cargo Ship: Generics 40002
   > Cargo Ship: Barcelona 40003
   > Cargo Ship: Toluene 40004

 --- List of all persons:
   > Person: Sara 50000 electrician
   > Person: Duane 50002 inspector
   > Person: Betsy 50004 cleaner
   > Person: Archie 50003 captain
   > Person: Thomas 50001 clerk
```

Suggestions:

Methods that should be implemented.

Each class should have an appropriate toString method. Here is an example of two such methods:

- In SeaPort - showing all the data structures:

```
   public String toString () {
      String st = "\n\nSeaPort: " + super.toString();
      for (Dock md: docks) st += "\n" + md;
      st += "\n\n --- List of all ships in que:";
      for (Ship ms: que ) st += "\n   > " + ms;
      st += "\n\n --- List of all ships:";
      for (Ship ms: ships) st += "\n   > " + ms;
      st += "\n\n --- List of all persons:";
      for (Person mp: persons) st += "\n   > " + mp;
      return st;
   } // end method toString
```

- In PassengerShip, using parent toString effectively:

```
   public String toString () {
      String st = "Passenger ship: " + super.toString();
      if (jobs.size() == 0)
         return st;
      for (Job mj: jobs) st += "\n       - " + mj;
      return st;
   } // end method toString
```

Each class should have an appropriate Scanner constructor, allowing the class to take advantage of super constructors, and any particular constructor focusing only on the addition elements of interest to that particular class. As an example, here's one way to implement the PassengerShip constructor:

- PassengerShip Scanner constructor, the earlier fields are handled by Thing (fields: name, index, parent) and Ship (fields: weight, length, width, draft) constructors.

```
   public PassengerShip (Scanner sc) {
      super (sc);
      if (sc.hasNextInt()) numberOfPassengers =
sc.nextInt();
      if (sc.hasNextInt()) numberOfRooms = sc.nextInt();
      if (sc.hasNextInt()) numberOfOccupiedRooms =
sc.nextInt();
   } // end end Scanner constructor
```

In the World class, we want to read the text file line by line. Here are some useful methods types and code **fragments** that you should find helpful:

- Handling a line from the file:

```
   void process (String st) {
   //      System.out.println ("Processing >" + st + "<");
```

```
            Scanner sc = new Scanner (st);
            if (!sc.hasNext())
               return;
            switch (sc.next()) {
               case "port"   : addPort      (sc);
                  break;
```

- Finding a ship by index - finding the parent of a job, for example:

```
         Ship getShipByIndex (int x) {
            for (SeaPort msp: ports)
               for (Ship ms: msp.ships)
                  if (ms.index == x)
                     return ms;
            return null;
         } // end getDockByIndex
```

- Linking a ship to its parent:

```
         void assignShip (Ship ms) {
            Dock md = getDockByIndex (ms.parent);
            if (md == null) {
               getSeaPortByIndex (ms.parent).ships.add (ms);
               getSeaPortByIndex (ms.parent).que.add (ms);
               return;
            }
            md.ship = ms;
            getSeaPortByIndex (md.parent).ships.add (ms);
         } // end method assignShip
```

You will probably find the comments in the following helpful, they are mostly about similar projects and general issues in Java relevant to our programs:

- CMSC 335 Information
- Cave Strategy - getting started

Project 2 - Also see Grading

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

Elaboration:

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 in part 2.
5. Again, the GUI elements should be distinct from the other classes in the program.

Project 3 - Also see Grading

Implement threads and a GUI interface using advanced Java Swing classes.

The project will be graded according the criteria for the final project - see below.

Elaboration:

1. Required data the data structure specified in Project 1:
    1. World has SeaPort's
    2. SeaPort has Dock's, Ship's, and Person's
    3. Dock has a Ship
    4. Ship has Job's
    5. PassengerShip
    6. CargoShip
    7. Person has a skill
    8. Job requires skills- NEW CLASS for this project!
    9. PortTime
2. Extend Project 2 to use the Swing class JTree effectively to display the contents of the data file.
    - (Optional) Implement a JTable to also show the contents of the data file. There are lots of options here for extending your program.
3. Threads:
    - Implement a thread for each job representing a task that ship requires.
    - Use the synchronize directive to avoid race conditions and insure that a dock is performing the jobs for only one ship at a time.
        - the jobs of a ship in the queue should not be progressing
        - when all the jobs for a ship are done, the ship should leave the dock, allowing a ship from the que to dock
        - once the ship is docked, the ships jobs should all progress
        - in Project 4, the jobs will also require persons with appropriate skills.
    - The thread for each job should be started as the job is read in from the data file.
    - Use delays to simulate the progress of each job.
    - Use a JProgressBar for each job to display the progress of that job.
    - Use JButton's on the Job panel to allow the job to be suspended or cancelled.
4. As before, the GUI elements should be distinct (as appropriate) from the other classes in the program.
5. See the code at the end of this posting for some suggestions.

Suggestions for Project 3 Job class. Here is a sample of code for a Job class in another context, the Sorcerer's Cave project. The code for this class will need some modifications, but this should give you an idea of the issues involved.
In fact, you should find much of this code redundant.

Also, some of the code at the following sites might give you some ideas about how to proceed with this project:

- Project 3 Example - Sorcerer's Cave, note that even this one isn't complete
- run method - detailed analysis of the run method in the Job class in the Cave project

```
// j:<index>:<name>:<creature index>:<time>[:<required artifact
type>:<number>]*
class Job extends CaveElement implements Runnable {
  static Random rn = new Random ();
  JPanel parent;
  Creature worker = null;
  int jobIndex;
  long jobTime;
  String jobName = "";
  JProgressBar pm = new JProgressBar ();
  boolean goFlag = true, noKillFlag = true;
  JButton jbGo   = new JButton ("Stop");
  JButton jbKill = new JButton ("Cancel");
  Status status = Status.SUSPENDED;

  enum Status {RUNNING, SUSPENDED, WAITING, DONE};

  public Job (HashMap <Integer, CaveElement> hmElements, JPanel
cv, Scanner sc) {
    parent = cv;
    sc.next (); // dump first field, j
    jobIndex = sc.nextInt ();
    jobName = sc.next ();
    int target = sc.nextInt ();
    worker = (Creature) (hmElements.get (target));
    jobTime = sc.nextInt ();
    pm = new JProgressBar ();
    pm.setStringPainted (true);
    parent.add (pm);
    parent.add (new JLabel (worker.name,
SwingConstants.CENTER));
    parent.add (new JLabel (jobName    ,
SwingConstants.CENTER));

    parent.add (jbGo);
    parent.add (jbKill);

    jbGo.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent e) {
        toggleGoFlag ();
      }
    });

    jbKill.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent e) {
        setKillFlag ();
      }
    });

    new Thread (this).start();
  } // end constructor

//      JLabel jln = new JLabel (worker.name);
//        following shows how to align text relative to icon
//      jln.setHorizontalTextPosition (SwingConstants.CENTER);
//      jln.setHorizontalAlignment (SwingConstants.CENTER);
//      parent.jrun.add (jln);

  public void toggleGoFlag () {
    goFlag = !goFlag;
  } // end method toggleRunFlag

  public void setKillFlag () {
    noKillFlag = false;
    jbKill.setBackground (Color.red);
  } // end setKillFlag

  void showStatus (Status st) {
    status = st;
    switch (status) {
      case RUNNING:
        jbGo.setBackground (Color.green);
        jbGo.setText ("Running");
        break;
      case SUSPENDED:
        jbGo.setBackground (Color.yellow);
        jbGo.setText ("Suspended");
        break;
      case WAITING:
        jbGo.setBackground (Color.orange);
        jbGo.setText ("Waiting turn");
        break;
      case DONE:
        jbGo.setBackground (Color.red);
        jbGo.setText ("Done");
        break;
    } // end switch on status
  } // end showStatus

  public void run () {
    long time = System.currentTimeMillis();
    long startTime = time;
    long stopTime = time + 1000 * jobTime;
    double duration = stopTime - time;

    synchronized (worker.party) { // party since looking forward
to P4 requirements
      while (worker.busyFlag) {
        showStatus (Status.WAITING);
        try {
          worker.party.wait();
        }
        catch (InterruptedException e) {
        } // end try/catch block
      } // end while waiting for worker to be free
      worker.busyFlag = true;
    } // end sychronized on worker

    while (time < stopTime && noKillFlag) {
      try {
        Thread.sleep (100);
      } catch (InterruptedException e) {}
      if (goFlag) {
        showStatus (Status.RUNNING);
        time += 100;
        pm.setValue ((int)(((time - startTime) / duration) *
100));
      } else {
        showStatus (Status.SUSPENDED);
      } // end if stepping
    } // end runninig

    pm.setValue (100);
    showStatus (Status.DONE);
    synchronized (worker.party) {
      worker.busyFlag = false;
      worker.party.notifyAll ();
    }

  } // end method run - implements runnable

  public String toString () {
    String sr = String.format ("j:%7d:%15s:%7d:%5d", jobIndex,
jobName, worker.index, jobTime);
    return sr;
  } //end method toString

} // end class Job
```

Project 4 - Also see Grading

Extend project 3 to include making jobs wait until people with the resources required by the job are available at the port.

Elaboration:

1. Reading Job specifications from a data file and adding the required resources to each Job instance.
2. Resource pools - SeaPort.ArrayList <Person> list of persons with particular skills at each port, treated as resource pools, along with supporting assignment to ships and jobs.
3. Job threads - using the resource pools and supporting the concept of blocking until required resources are available before proceeding.
4. The Job threads should be efficient:
    1. If the ship is at a dock and all the people with required skills are available, the job should start.
    2. Otherwise, the Job should not hold any resources if it cannot progress.
    3. Use synchronization to avoid race conditions.
    4. Each Job thread should hold any required synchronization locks for a very short period.
    5. When a job is over, all the resources used by the job (the people) should be released back to the port.
    6. When all the jobs of a ship are done, the ship should depart the dock and if there are any ships in the port que, one of then should should be assigned to the free dock, and that ships jobs can now try to progress.
    7. NOTE: If a job can never progress because the port doesn't have enough skills among all the persons at the port, the program should report this and cancel the job.
5. GUI showing:
    - Resources in pools - how many people with skill are currently available
    - Thread progress, resources acquired, and resources requests still outstanding

Grading

Deliverables - for all projects:

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 doc, docx, OpenOffice odf, pdf, rtf.

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

Documentation format and length. The documentation describing and reflecting on your design and approach should be written using Microsoft Word, 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, IEEE or ACM style.

Coding hints:

- Code format:
  - 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 with errors will be returned ungraded
- 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
  - JProgressBar in Projects 3 and 4
  - JSplitPane - optional, but gives user even more control over display panels

Grading Table

Attributes	Value
Project design	20 points
Project functionality	40 points
Test data	20 points
Approach documentation	15 points
Approach documentation grammar and spelling	5 points

Total	100 points

Package

The files in the zip file package:

- index.html - this file, project descriptions with suggestions and hints
- CreateSeaPortDataFile.java - command line driven java application source code
    - optional parameters - random indicates that the exact number is based on a random number
        - ? will print a usage message
        - [0] - first parameter, a String specifying the output file name
        - [1] - number of ports
        - [2] - number of docks per port (random)
        - [3] - number of passenger ships per port (random)
        - [4] - number of cargo ships per port (random)
        - [5] - number of jobs per ship (random)
        - [6] - number of persons per port (random)
- files used to generate names
    - You can change the contents of these files, or substitute simply the giving a different list of names file one of the following file names.
    - personNames.txt - 1000 names from census lists of popular names
    - portNames.txt - 1741 names of ports around the world
    - shipNames.txt - 109,582 words from a dictionary - SIL International Linguistics Department, 7500 W. Camp Wisdom Road, Dallas, TX 75236, U.S.A.
      I have not taken the time to remove words that might be offensive - there are just too many of them.
    - skillNames.txt - just a random selection. Note that making this longer makes it harder for jobs to find the skills they require at the ports.
- Sample data files - generated by the CreateSeaPortDataFile.java application
    - aSPaa.txt - (1 1 5 5 1 5) - a relatively small data set, 1 port, 5 docks, 11 ships, 5 people
    - aSPab.txt - much larger example with 5 ports, 38 (or so) docks
    - aSPac.txt - (8, 15, 20, 20, 5, 30) - another large data file
    - aSPad.txt - (3, 4, 5, 6, 7, 8) - a middling size data file
    - aSPae.txt - (8, 15, 20, 20, 5, 30) - another large data file, same parameters as aSPac, but different (random) output from the creator program.