

Project 3

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 3 General Objectives

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.

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 3 Specific Goals:

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

1. Required 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.
 - o **(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:
 - o Implement a thread for each **job** representing a task that ship requires.
 - o 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.
 - o The thread for each job should be started as the job is read in from the data file.
 - o Use delays to simulate the progress of each job.
 - o Use a JProgressBar for each job to display the progress of that job.
 - o 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 synchronized 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 running

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

```

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. Effectively uses of the class hierarchy, including features related to polymorphism. GUI elements should be distinct from the other classes in the program.	0 points Does not contain just the right amount of code. Does not use existing classes in the JDK effectively. Does not effectively use of the class hierarchy, including features related to polymorphism. 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.	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.

	<p>Extends Project 2 to use the Swing class JTree effectively to display the contents of the data file.</p> <p>Implements a thread for each job representing a task that ship requires.</p> <p>Uses the synchronize directive to avoid race conditions and insure that a dock is performing the jobs for only one ship at a time.</p> <p>The thread for each job starts as the job is read in from the data file.</p> <p>Uses delays to simulate the progress of each job.</p> <p>Uses a JProgressBar for each job to display the progress of that job.</p> <p>Uses JButton's on the Job panel to allow the job to be suspended or cancelled.</p> <p>Extends the GUI from Project 1 to allow the user to sort by the comparators.</p>	<p>Does not extend Project 2 to use the Swing class JTree effectively to display the contents of the data file.</p> <p>Does not implement a thread for each job representing a task that ship requires.</p> <p>Does not use the synchronize directive to avoid race conditions and insure that a dock is performing the jobs for only one ship at a time.</p> <p>The thread for each job does not start as the job is read in from the data file.</p> <p>Does not use delays to simulate the progress of each job.</p> <p>Does not use a JProgressBar for each job to display the progress of that job.</p> <p>Does not use JButton's on the Job panel to allow the job to be suspended or cancelled.</p> <p>Does not extend the GUI from Project 1 to allow the user to sort by the comparators.</p>
Test Data	<p>20 points</p> <p>Tests the application using multiple and varied test cases.</p>	<p>0 points</p> <p>Does not test the application using multiple and varied test cases.</p>
Documentation and submission	<p>15 points</p> <p>Source code files include header comment block, including file name, date, author, purpose, appropriate comments within the code, appropriate variable and function names, correct indentation.</p> <p>Submission includes Java source code files, Data files used to test your program, Configuration files used.</p>	<p>0 points</p> <p>Source code files do not include header comment block, or include file name, date, author, purpose, appropriate comments within the code, appropriate variable and function names, correct indentation.</p> <p>Submission does not include Java source code files, Data files used to test your program, Configuration files used.</p>

	<p>Documentation includes a UML class diagram showing the type of the class relationships.</p> <p>Documentation includes a user's Guide describing of how to set up and run your application.</p> <p>Documentation includes a test plan with sample input and expected results, test data and results and screen snapshots of some of your test cases.</p> <p>Documentation includes Lessons learned.</p> <p>Documentation is in an acceptable format.</p>	<p>Documentation does not include a UML class diagram showing the type of the class relationships.</p> <p>Documentation does not include a user's Guide describing of how to set up and run your application.</p> <p>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.</p> <p>Documentation does not include Lessons learned.</p> <p>Documentation is not in an acceptable format.</p>
Documentation form, grammar and spelling	<p>5 points</p> <p>Document is well-organized.</p> <p>The font size should be 12 point.</p> <p>The page margins should be one inch.</p> <p>The paragraphs should be double spaced.</p> <p>All figures, tables, equations, and references should be properly labeled and formatted using APA style.</p> <p>The document should contain minimal spelling and grammatical errors.</p>	<p>0 points</p> <p>Document is not well-organized.</p> <p>The font size is not 12 point.</p> <p>The page margins are not one inch.</p> <p>The paragraphs are not double spaced.</p> <p>All figures, tables, equations, and references are not properly labeled or formatted using APA style.</p> <p>The document should contains many spelling and grammatical errors.</p>