CST8132: Object-Oriented Programming (in Java)

Lab Assignment 1: Creating Actor Objects

Introduction

You’ll work towards the construction of The Hobbit battlefield simulator over a number of labs. In this first lab, you’ll begin with a simple class:

- **Actor** class to hold the attributes (strength, speed, etc.) of a single **Actor** object.

There will be several stages before you can unleash the full battlefield simulator:

- **Second Assignment**: Expand your battlefield simulator with the addition of several different subclasses of **Actor** (Hobbit, Wizard etc.).
- **Third Assignment**: Implement an **Army** class (from which you’ll build the Forces of Light and the Forces of Darkness). The **Army** class will be a container class that holds a collection of **Actor** objects (initially the collection will be implemented as an array of references, later as an ArrayList).
- **Fourth Assignment**: Display your **Actor** objects on the screen using JavaFX.
- **Fifth Assignment**: Move **Actor** objects and have them battle amongst themselves as autonomous entities.

Later, you’ll also use file handling to save and restore the attributes of **Actor** objects.

The idea for these labs grew after reading about some very interesting development efforts by WETA, the software-animation development firm associated with the production of The Hobbit and its predecessor, the Lord of the Rings film trilogy. If you want more background information, you can check this site:


Their principle challenge in crafting the initial Lord of the Rings battlefield involved creating the battlefield animation sequences, where there might be up to 100,000 actors (who, as humans, would have been far too expensive to pay, feed, house and costume . . . ). The software developers had to imbue enough intelligence in their computer-generated actors to play out dramatic battlefield sequences. But it didn’t always work out that way.

Once the scene was set and the actors imparted with appropriate attributes and behaviors, the simulator was unleashed; sometimes with surprising results (which are explored in the article referenced above).

Clearly, we don’t have a hundred million development dollars, nor do we have tens of thousands of hours of time to program, but we will explore some interesting object-oriented issues through the creation of a **Battlefield Simulator**.

The Actor Class

The **Actor** class will be used to create individual **Actor** objects, and we’ll start with a modest set of attributes. In subsequent labs, as you continue with the development process, you will probably add more attributes and behaviors, but leave that for now.

Actor Instance Variables

- **name**: This will be a reference to a **String** object, built using the **String** class.
- **strength**: This **double** value will be used to determine the outcome of skirmishes that might take place. As part of your planning, you must determine the range of legitimate values for **strength**. (Defining a legitimate range is an important step in structuring your testing.
- **speed**: This **double** value will affect an actor’s speed of movement during the simulation. Again, determine the range of legitimate values for **speed**.
- **health**: This **double** value equates to the actor’s remaining life-force. And again, determine the range of legitimate values for **health**. It can also have an effect on the speed of the **Actor** object.

Actor static Class Variable

- **actorSerialNumber**: Because this **int** variable is **static**, there will be one-and-only-one instance of the variable that is shared by all **Actor** objects. It will be used to add a sequence number to each new **Actor** object that is created automatically.

Actor Methods

- **Constructor**: Establishes initial values for all instance variables automatically. The **String** variable containing the name will start with “**Actor**” plus the sequence number described above. The constructor also establishes random initial values for all numeric instance variables. Each random value must be within your specified range of legitimate values. Random numbers have different distribution patterns:
  - **flat or even distribution**: With this distribution, numbers will have an equal probability of falling on any point between the upper and lower boundaries.
  - **Gaussian distribution**: This is a fancy name for a “normal” or a “bell curve” distribution. The generated numbers will tend to cluster around the mean and the occur with less frequency on either side of the mean. This distribution reflects real-world situations better . . . so we’ll use a Gaussian distribution.

I have implemented the class **SingletonRandom**, an example of the Singleton Design Pattern.

- **inputAllFields()**: Sets all the object’s fields through keyboard entry. This method can over-write existing values in objects. It will be used to establish values in newly created **Actor** objects, or make edit changes to existing **Actor** objects. For example, you might want to create 1,000 **Actor** objects that have randomly generated values. Then, you might want to edit 1 of the 1,000 to give it a specific name, such as “Gandalf the Grey” and specific **strength**, **health** and **speed** attributes.

You will use input routines provided in my **Input** and **InputGUI** classes. **Input** is implemented using a Singleton Design Pattern. **InputGUI** is implemented with all static methods (not

1 Other terms are often used in place of the term **instance variable**. These include: **field**, **data member**, and **attribute**.

2 Other terms are often used in place of the term **method**. These include: **member function** and **object behavior**.
Testing

For this assignment, you'll create paper-based test plans. For subsequent assignments, you'll capture your testing using an important industry tool: jUnit Testing.

I have completed some of the paper-based test plan work for you. If you check the end of this document, you'll see a partially completed Test Plan chart. You can use my chart as a starting point, and add to it.

In performing the tests, you will need to create a separate class to exercise your Actor class. Look to the first example in Chapter 8 of your textbook for a model of organization. That example builds the Time1 class to handle time values, then builds another class to test the behavior of the Time1 class. In particular, you'll want to ensure that the inputAllFields() and toString() methods behave as expected.

Create at least five Actor objects to prove that each object retains its separate identity, and that values are established correctly by the constructor (which establishes values automatically), and when inputAllFields() is used for keyboard editing.

When you test the program, there are different categories of testing:

- **Boundary Condition**: Variables such as: strength, speed, health have maximum and minimum values. Check and handle these boundary conditions in the inputAllFields() method. If an invalid value is encountered, you can replace it with one that fits within the range or prompt the user to re-enter.
- Were values actually captured in newly created Actor objects?
- Can input values be used to change fields in an existing created Actor object (that is, in support of an edit function).

Submission

Documentation

- **Title Page**: It must identify you, the course, section number, lab professor, assignment name and number, date of submission.

**Problem Statement**: What are you attempting to do? What is the purpose of each of the fields? This statement should be in paragraph form.

**Memory Map**: Shows a sample of data as it would be organized in memory during program execution. You must show how a named reference variable relates to the corresponding unnamed Actor object. Show the five stack-allocated, named reference-to variables you created in your test class (inside your main() method). They should contain a numeric value representing the "id" or "reference-to" value. Of course, this value is the location information for the unnamed Actor objects that you have created using new. Draw an arrow from each named variable to the associated unnamed Actor object. Each Actor object will have four instance variables inside. The three numeric values are primitives, so the numbers are stored directly in that location. The name variable is also a "reference-to" value, so it should have an "id" number that points at some external unnamed String object. Drawing the memory map will convince me that you understand the difference between primitive variables and reference-to variables.

- **Test Plan**: The test plan is best organized in a chart format. I have supplied an example showing the best organization. Many of the cells in the chart are already filled in (to clarify my expectations). You are to fill in some of the open cells.
- **Program Code**: Your code must adhere to the Java Standard published by Oracle. It can be accessed at:


This standard includes guidelines for variable and method names; for indentation patterns; for the use of braces etc. Each major section of code should be commented (at the left margin ahead of start of the section). Many individual lines of code will have a line-specific comment: in such cases, the comment should appear on the same line and to the right of the Java code. You can check the Input class supplied by me for an example of comprehensive comments.

Indentation shows program structure. Eclipse will set consistent indentation automatically. If indentation becomes ragged, you can reformat your code according to the current formatting style in Eclipse. The shortcut key is **<Ctrl-Shift-F>**.

**Due Date**

Midnight Sunday February 2\textsuperscript{nd}

**Submission Procedures and Evaluation**

The following procedures have been defined by your lab professors. If you need further clarification, you can check with your lab professor.

Assignments must be submitted electronically via the link in Blackboard on or before the due date. Late submissions will receive a 0. Do not wait until the last minute to submit your assignments. Each assignment should consist of a single .zip file containing all requested documentation, code, memory maps, etc. as outlined in the above document. This zip file must be labeled as follows:

**YourLastName_YourFirstName_Assign#_Your LabProfessorsName.zip**

For example:

**Smith_Bob_Assign1_CarolynMacIsaac.zip**

---

3 The use of *get* and *set* methods are design patterns. Some refer to the pattern as *getter* and *setter*. Some refer to them as the accessor and mutator design pattern.
If your lab is on Friday at 10:00 a.m. then your lab professor is Dave Houtman. Otherwise your lab professor is Carolyn MacIsaac.

Most documents can be submitted in Notepad, Wordpad or Word format. Diagrams may be submitted using Visio or Word (select Insert--> Shapes --> Flowchart. To add text boxes, select Insert-->Textbox.) If you draw a memory map with paper and pencil, you can scan the document and attach to your submission. All submissions must be electronic.

Java files should be clearly and correctly labeled with the .java extension after the filename. All submitted code must be 'human-readable' and should display in a simple text editor (like Notepad) without any additional non-printing characters appearing in the file (which can occur when code is copied directly from Wordpad, Word, or .pdf formats).

All code submitted must work 'out of the box', without modification.
### Test Plan: Creating Objects Using Class Actor (partially complete)

<table>
<thead>
<tr>
<th>Task</th>
<th>Pre-condition</th>
<th>Post-condition</th>
<th>Test Method</th>
<th>Expected Result</th>
<th>Actual Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Actor inputAllFields():</strong> test input of <code>String</code> values.</td>
<td>Actor object exists.</td>
<td>Actor object has correctly populated value in <code>String</code> field.</td>
<td>Enter string of characters with embedded spaces and other special characters. Inspect field value through debugger during program execution. Output field to screen through use of <code>toString()</code>.</td>
<td>Capture entire sequence of characters up to the <code>&lt;Enter&gt;</code>.</td>
<td>Success</td>
</tr>
<tr>
<td><strong>Actor inputAllFields():</strong> test input of <code>double</code> values.</td>
<td>Actor object exists.</td>
<td>Actor object has correctly populated value in <code>int</code> fields: <code>strength</code>, <code>speed</code> and <code>health</code>.</td>
<td>Enter non-numeric input. Trace execution through debugger. Watch for meaningful prompts on screen to alert user to error.</td>
<td>Program rejects input and prompts user for numeric input.</td>
<td>Successfully traps user error.</td>
</tr>
<tr>
<td><strong>Actor toString():</strong> test display of object contents, using a direct call of <code>toString()</code>.</td>
<td></td>
<td></td>
<td>Enter numeric input, but outside acceptable ranges. Trace execution through debugger. Watch for meaningful prompts on screen to alert user to error.</td>
<td>Program rejects input and prompts user for numeric input within specified range.</td>
<td>Successfully traps user error.</td>
</tr>
<tr>
<td><strong>Actor toString():</strong> test display of object contents, using an implicit call of <code>toString()</code> by outputing the object.</td>
<td></td>
<td></td>
<td>Enter numeric input within acceptable range.</td>
<td>Capture double value and populate target fields: <code>strength</code>, <code>speed</code> and <code>health</code>.</td>
<td>Success</td>
</tr>
</tbody>
</table>

Note that these two rows are subcases of the test of `int` values. Thus, the first three columns do not need entries.

These cells are to be completed by you.