CS-110 2017/18 - Coursework 2: The Solar System Model

Neal Harman, Nov 2017

As before the first part of the marking process is automated - your code will be automatically compiled and run. This time however:

- The 'main' automatic test will use a class that I will supply which will tell you if your code passes or fails. (This wasn't possible last time because you didn't know about classes.)
- There will be a second automatic test but there is no defined output - your code just has to read a series of strings and act correctly

The Solar System Model

You are going to write a class (or ideally *classes* - hint!) that models solar systems in a very minimal way. (And please don't be one of those picky people who point out "there is only one solar system; others are planetary systems" - doesn't help).

Our solar systems will have:

- A *name*
- A series of zero or more *planets* each of which has
    - A *name*
    - A *mass* (expressed in terms of *Earth-masses* - so the mass of the Earth is 1.0)
    - A *distance* from it's star (expressed in *Astronomical Units,* or AU- where the Earth is 1.0 AU from the Sun)
    - An *orbital period* expressed in *years*.

However, since the orbital period is a function of the distance, you will *not* specify the orbital period when you create a planet - instead it will be *computed* (formula is below).

What You Must Write

1. A class (or classes!) to define a solar system. You **must** stick to the names (*including exact case*) and methods below or your code will not work

- **Name of Class** - the primary class must be called `SolarSystem`: you may write sub classes to help you implement it in a better way if you wish (and there can be as many as you like - *don't over do it* - and they can be called what you like). This is a hint! There is *at least* one 'sensible' additional class here (but don't overdo it).
- **Constructor** - there should be one constructor `SolarSystem(String name)` where `name` is the name you wish to give a solar system.
- **addPlanet** - there should be a method `void addPlanet(String name, double mass, double distance)` that creates a new planet with the specified `name`, `mass` and `distance` from its star.
- **toString** - there should be a method `String toString()` that returns a string containing:
    - The name of the solar system followed by a newline;
    - data for each planet in the following **exact** format with each line ending in a newline.

```
Planet X has a mass of A Earths, is BAU from its star and orbits in C years
```

where X is the name, A is the mass, B is the distance from the star (both as entered above when a planet is created by a call to addPlanet; and C is the orbital period rounded to three decimal places. *In all cases, A, B and C need to be returned rounded to three decimal places.* It is up to you if you store them this way, or round them when they are output.

- **Note** that as before the *exact* format of the text string is *critical* to the correct operation of the program - you can find examples of the format in the test file (included with the coursework) called `AutoTest.java`.
- **Other methods** - you will need to provide other methods (and maybe constructors) too: these will be up to you and depend on how you approach dealing with the second part of testing (see below).

2. A class that reads in strings representing the name of a solar system and the names of its planets (one per line) followed by the word "done". This class should have the following properties:

- **Name of Class** - the class should be called `FantasySolarSystem`. Again you can write additional sub-classes if you want but it's hard to see a good reason why you should need to in this case (unlike in part 1 above). Y*our* `FantasySolarSystem` *class should use your* `SolarSystem` *class (1. above) to function - you will lose marks if you write two completely separate programs.*
- **Input Solar System Name** - the class should first read in the name of the solar system being created as a line of text.
- **Input Planet Names** - the class should then read in planet names until the user types "done". (It makes sense that you read in the solar system and planet names, and "done" using a `Scanner` and `nextLine();`)
- **Create Planets** - using the names typed in, the class should generate *random* values for the mass and distance and use that information to create and add planets to the solar system.
- **Print Output in Table** - once the user has entered "done" the class should print out information about the solar system in a formatted table. For example, suppose we entered the following information:

```
FruitSystem
Lemon
Orange
Banana
done
```

where FruitSystem is the name of the solar system and the others are names of planets. The output might be:

```
Enter the name of the solar system: FruitSystem
Now enter planet names - type 'done' to finish
Enter name: Lemon
Planet Lemon has a mass of 14.716 Earths, is 1.216AU from its star, and orbits in 1.341 years
Enter name: Orange
Planet Orange has a mass of 6.964 Earths, is 2.583AU from its star, and orbits in 4.151 years
Enter name: Banana
Planet Banana has a mass of 15.862 Earths, is 0.254AU from its star, and orbits in 0.128 years
Enter name: done
Name            Mass            Distance        Period (years)
Lemon           14.716          1.216           1.341
Orange          6.964           2.583           4.151
Banana          15.862          0.254           0.128
```

Text in **bold** is printed by the computer; normal text is entered by the user. Note that in this example after every planet name is entered a one-line summary is printed - this is *not* necessary (but is a nice touch:-). The table also does **not** have to look exactly like this - a*ny reasonably well formatted table is fine*. Notice that as before the mass, distance and period are *rounded* to three decimal places.

NOTE

- **The mass and distance values are random** - even if you enter the same names you will get different results for these.
- *Unlike the first coursework and the first part of this one,* the output does not have to **exactly** match the format above - provided your `FantasySolarSystem` class can read in the name of

the system followed by the names of the planets (followed by “done”) and can output something that looks similar to the final table then it will be considered correct.

Testing Part 1

The first part of testing your submission will be done by running the `AutoTest.java` class that I have supplied to you. To do this, simply put all your java files in the same folder as `AutoTest.java` and type:

```
javac *.java
java AutoTest
```

The first line compiles all the Java code (yours and `AutoTest.java`) and the second runs the `AutoTest` class. The output should be:

```
Pass!
```

Any other output means your code is not correct, and you need to work out what’s wrong. The error is likely to be in the exact format of the output - if you open and read `AutoTest.java` you can see exactly what it expecting.

Testing Part 2

The second part of testing uses a pre-defined input file - however, unlike the first coursework, the exact format of the output is *not* defined. An example file is supplied as part of the coursework - however it is very simple to write your own if you wish: just create a text file with the name of the solar system on the first line; the name(s) of the planets on the following lines; with “done” on the last line (like the example above). To run your test type:

```
javac *.java
java FantasySolarSystem < testfile.txt
```

where `testfile.txt` is the name of your test file containing test data (the ‘<‘ character directs your program to read from the file instead of the keyboard). Note this works on a Mac or Windows (or Linux). This should generate data similar to that above which will be *manually* checked and does *not* have to correspond to a particular format or patten (other than the final output being in a tabular form with all the orbital periods reported in years).

Learning from Feedback

For this coursework 5% of the marks are available for responding to feedback or showing how you have improved from the previous coursework.

- **If your mark for coursework 1 was less than 100%** then you should write, in a block comment at the top of your code, how you have responded to feedback. That is, what issues in your first coursework - recorded in the marking rubric - you have improved.
- **If your mark for coursework 1 was 100%**, then you should write, in a block comment at the top of your code, in what way you feel you have improved your work (obviously you can’t meaningfully respond to actual feedback in this case).
- **In BOTH cases** you should also consider and respond to any of the general feedback (supplied as part of the coursework 1 feedback pack on Blackboard).

Useful Information

You will find the following information useful.

- To calculate the orbital period from the distance where the distance is in AU and the orbital period is in years, the formula is

```
period = Math.sqrt(distance*distance*distance);
```

- To generate random numbers in the range 0 to n, where n is a double, in Java you should use code like this:

```
import java.util.Random;
```

```
…
Random rnd = new Random();

double rand = rnd.nextDouble()*n;
```

- Alone, `nextDouble()` generates a random `double` in the range 0 to n. You will need to produce random numbers for the distance and mass - the values of n (it should be different in each case) are up to you - however, for distance it should certainly be > 1.
- To round a number in Java to three decimal places:
  ```
  double rounded = Math.round(number*1000)/1000.0;
  ```
  Note that you have to write 1000.0 for one or both of the numbers in the formula to force it to be a `double`.
- There are other ways to round and you're welcome to use them - but *note that you need to be careful to make sure the output you get matches the required output in* `AutoTest.java`
- There are several ways of using formatting strings to generate output tables in Java and you might want to look up `System.out.printf()`. *However*, it's also perfectly possible to do it just using `System.out.println()` and *tab* characters - to embed a tab character in a string in Java use "`\t`". That is, to print lemon and orange separated by a tab: `System.out.println("lemon\tOrange")`.
- When you first get output from `AutoTest.java` it is unlikely to be correct and you will have to work out what is different - you can do this manually but it will take time. Better to copy and paste the actual and expected output into two files in an editor and use that to compare them. For Notepad++ you can download a compare plugin; Textwranger (Mac) does it without any extra plugins. Just google *Notepad++ compare files* (or *Textwranger compare files*). This will be a lot easier than manually doing it - you have a computer which is good at things like that, so use it. *Alternatively, and quicker/easier/more sophisticated,* you can use the currently commented out code in `AutoTest.java` that calls a library method to highlight the differences - see last point in *Resources* below.
- One of the most fiddly bits of this will be to get the numbers output by your program and `AutoTest.java` to match exactly. Remember, `mass, distance` and, orbital period needs to be rounded as the *last step* in the *calculation* process - you may either *store* them in rounded form, or round them before you *output* them.

Resources

In order to help you, there is a zip file accompanying this coursework. This contains:

- `AutoTest.java` - this is the test program I will use for Part 1 Testing. You should run this program using the commands shown above.
- `FruitSystem.txt` - an example input file for Part 2 Testing. This is a the actual input file I will run to test your code. Again you can run this test using commands listed above.
- `PreTest_nix.sh` and `PreTest_win.bat` – these are the scripts you should run to check your program passes both parts of the tests. They will compile your code, run `AutoTest` (the output of which should be 'Pass!') and then run your `FantasySolarSystem` class (which will read in TestSystem.txt and print out a table containing the data). *If you run the test script for your operating system and get these results, your program works correctly.*
- `SolarTest_nix.sh` and `SolarTest_win.bat` - these are the scripts I will run to check your program passes both parts of the tests. They do exactly the same as `PreTest_nix.sh` and `PreTest_win.bat` except they also automatically unzip your final submission (see below about creating a zip file for submission). *If you run the test script for your operating system once you've created the zip file you intend to submit you will be able to check both that your code works and you've created the zip file correctly.*
- The *Apache Project* library file `common-lang3.jar`. This is a Java Archive file which you can, *optionally*, use to help you find out where your output differs from that expected in your program. There is guidance on Blackboard on how to use this from the command line, Netbeans and Eclipse. Using library files is very common in practice in programming, so it's worth starting to learn how to do it. In this case, it uses the (static) `StringUtils.difference()` method to highlight where two strings differ.

Guidance Notes

These are *nearly* the same as the last assignment.

- *The biggest mistake you can make is failing to start the work early enough* - you need to not only get the code written and compiling, but also working and matching the output of the automatic tests. You need to give yourself time to deal with any hitches, and see me if you need to.
- *A **related** mistake you can make with this coursework is not **finishing** early enough* - this is because you need time to create *and check* the zip file you have to submit.
- *The second biggest mistake you can make is just to start writing code without thinking about what you are doing.* Don't just jump in and start writing - make a plan!
- *Break the program into parts* - you can write a successful solution to this coursework using either two or (better) three classes in total, and you need to break the problem into the parts corresponding to those classes..
- *Remember in object oriented programming: classes are 'kinds of thing'; objects are actual 'things'; methods are 'actions' that 'things' can do (or have done to them).* Also remember 'things' can be hierarchical - in the Bank Account example in the notes the Bank is one 'kind of thing'; which is made up of accounts which are another 'kind of thing'. In this case I've already told you what one of the 'kinds of thing' is; you need to work out if there is one or more other 'kinds of thing'.
- *Write and test in parts* - there is no need to try to do everything in one go: write the program in parts so you can test and check each part on it's own without worrying about the rest of it.
- *Don't be afraid to write test code* - if you're going to write your program in parts, you need write some extra test code. Sometimes people shy away from writing this code when they start - thinking the work is 'wasted' if it's not code they are going to submit. But it makes it easier and quicker to solve the problems you face, so it's **not** wasted effort - *it's less effort*
- *Don't be afraid to ask for help* - it's my job to help and guide you (not do it for you though) so please ask if you have problems. Also remember that discussion on the forum is fine and encouraged, provided your not asking exactly how to do it (or telling people exactly how to do it).
- *Don't be tempted by unfair practice* - it's sometimes tempting to submit someone else's solution (maybe changed a bit…), or get something off the internet, or ask someone else to do it. If you do this you risk serious penalties for breaching academic integrity. Remember we run submissions though software systems that detect these practices.
- *Read the marking criteria* - you have access to the marking rubric on Blackboard that tells you what will be assessed, how many marks are for each part, and what you need to do to get a mark in each category. Please pay attention to this.
- *Don't get carried away* - every year I get advanced solutions to coursework that gets a lower mark than some simpler solutions. This is because the advanced solutions focus too much on the 'advanced' things and forget the simpler basics (which are in the marking scheme).

Preparing for Submission

The submission for this project will be a zip file called `SolarSystem.zip` (case must be exact) containing:

- `FantasySolarSystem.java` (note the case must be exactly correct) - this is the class that runs the part 2 tests above and which you must implement.
- `SolarSystem.java` - your implementation of the class `SolarSystem.java` (again case must be exact).
- Any other Java classes you implement - if your implementation uses any other classes, you must include them too (the .java files only).
- You can include `AutoTest.java` but (a) you don't have to and (b) it does not matter if you do.
- It's important that when you create your zip file you don't accidentally include the folder your files are in. There are videos uploaded to Blackboard showing you how to do this.
- *Do not just submit a zipped Netbeans of Eclipse project - that won't work.*

Before You Submit

After you have created your zip file, make a copy somewhere else on your computer and run either `SolarTest_nix.sh` *or* `SolarTest_win.bat` *(depending if you are using a Mac or Windows). This will run the complete test suite and show you if your code is both correctly zips and works properly.*

Submission

Please note the submission details very carefully.

- You must **submit** your work via the coursework link on Blackboard by **11.00 on Wednesday 13th December 2017**. Do not email your work to me.
- You must upload a file called `SolarSystem.zip` containing your Java code. Also do not submit the .class files - just the .java files.
- ***Late submissions will receive a mark of zero in accordance with College policy*** - although you will get feedback. If you have **extenuating circumstances** which prevents you from submitting on time, *you must inform me as soon as possible* in order to make a case to remove the submission penalty. If you don't you will almost certainly not get the penalty lifted. Note that any case for extenuating circumstances almost certainly needs to be supported by *evidence* if its' going to be successful.
- I *strongly suggest* you make sure you are consistently using *either* sequences of four spaces *or* tabs for indenting. Do not mix both as this can often look like inconsistent indenting after it's been through Blackboard. You can use Netbeans, Eclipse, Notepad++ (or whatever other editor you are using) to automatically do this for you.
- You will receive **feedback** and a **mark** for your work on **5th January 2018**.
- On the same date I will release a range of **sample solutions** and **general feedback** - after that date I cannot accept any late submissions *regardless* of the reason.
- The **assessment criteria** for your work are defined by the **marking rubric on Blackboard**. This defines: the categories you will be assessed against; how many marks are available for each category; and what you need to do to get those marks. *It's important you study this carefully before you attempt the coursework - no matter how good your code is, if it does not meet the assessment criteria you will not get a high mark.*
- The **individual feedback** you will receive will consist of the marking rubric, identifying how well you have done in each category together with additional comments if appropriate.
- If you have **any problems with submission**, send me an email with **[submission]** (that *exact* text) in the title, with details of your problem and I'll advise you what to do - but do not include your actual coursework submission unless you are asked to.