SCC212: Advanced Programming

Week 1 Spinning up to Speed!

Welcome back!

It's been a little while since SCC110… and even longer since we've seen you all face to face, so let's take some time to refresh what we know and have a nice gentle start… before I start to work you into shape after eating all those lockdown biscuits!

Task 1: Java Refresher

We'll be building heavily on everything we learned in SCC110 over the next 5 weeks, so it's really important you answer any niggling doubts you may have from last year's materials.

- Study the Java Refresher slides supplied at the end of the Introduction lecture slides, available on the SCC212 moodle page. Make sure you take time to really review the content, and ensure you understand it. I'm considering all the content in there to be a pre-requisite for SCC212. Some of it may be new to you, as there are a few concepts we didn't have time to talk about in SCC110 last year.

- Discuss anything you don't fully understand with one of the teaching assistants in your lab, and/or one of your classmates. If anything comes up that the TAs can't answer (go on… try to think of a question they can't answer!!!), ask me in the lecture on Monday. We'll have timeslots for interactive question and answer.

- If some of the concepts in the slide deck are new to you (e.g. enumerations, foreach loops or inline conditionals), take time to write, compile and execute one or two very simple programs of your own invention that use these techniques.

- When you're comfortable, move on to Task 2 overleaf.

Task 2: Modelling the Solar System

Over the next few weeks, we'll be putting some of the theory about object orientated programming you learned last year into practice, and then developing it further as we discuss new concepts. You will also be integrating some classes we've written for you with classes of your own design. **The aim is for you to gain experience of developing a well-designed OO program**.

The Task

Your task this week is to create a simple moving model of the solar system, as illustrated below.

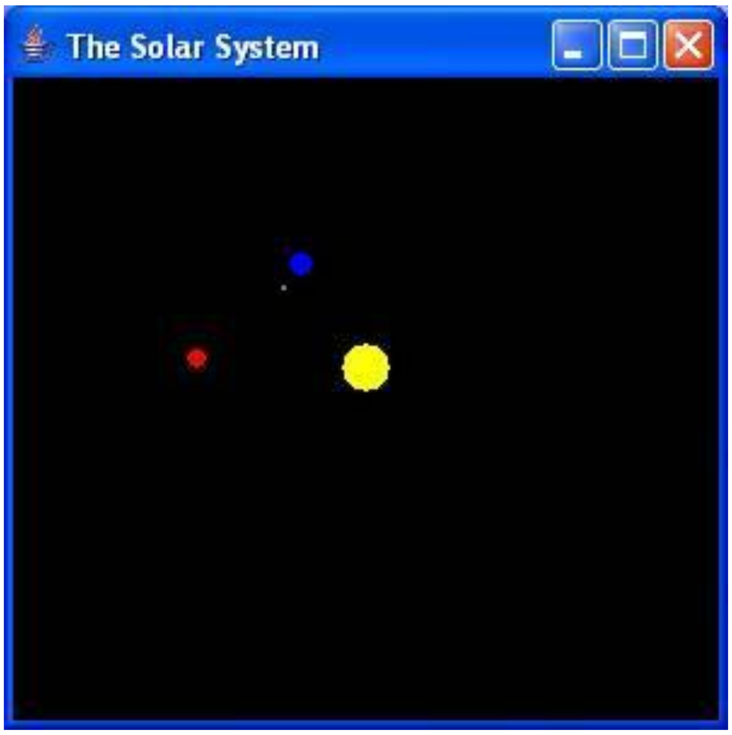

Figure 1 - An example model of the inner solar system

So we can focus our time on OO design rather than graphics, I've written a class for you that handles the graphical components called **SolarSystem**. You can download the java source file and associated **JavaDoc** for this class from the associated resources file on Moodle. I've provided the source code for the SolarSystem class, but you are not expected to change it**. In fact, for this exercise, you are not permitted to modify the SolarSystem.java file… just create instances of it**.

Exploring the SolarSystem

Using the SolarSystem class provided write a simple Java program that will:

- Create a window using the SolarSystem class.
- Draw a yellow sun in the centre of the window.
- Draw a blue earth revolving around the Sun.

Analyse your work

Consider the limitations of your solution. Pay particular attention to ***extensibility*** and ***scalability***. Discuss these limitations with a TA.

Ask yourself if you have used all the tools and techniques from your courses so far. E.g. I'm sure you did all the following right?

- Split your work into coherent classes
- Commented your code appropriately
- Diagnosed any bugs you had with a runtime debugger
- Created a private git repository for your work
- Created clean git commits messages regularly pushed back to github.

Remember – all the above are expectations in professional software development… You should now be practicing doing this for all your work… regardless of whether or not it is explicitly assessed. ☺

Task 3: Building an OO Model of the Solar System (Doing it properly!)

In the previous task you experimented with the SolarSystem class to create a simple graphical model. However, what you created was very likely a point solution. In all likelihood, it was not reusable, extensible, or elegant.

When you write OO code, you should **always** consider how it will be used in the future – quite possibly by other programmers as part of a solution to a different problem… so it's your job to make that as easy as possible. Software Engineers "pay it forward" by taking pride in their work – such that other software engineers can integrate it with their own code easily…

Focus your efforts this week on creating the **most elegant** (not the fastest to develop) solution to the SolarSystem model that you can. Your code should have Sprezzatura!

There are three main types of object in the solar system. There are suns, planets and moons. (There are others too, like comets and asteroids and aliens but let's keep it simple for now!). Note that these are similar in many ways - they will all have some attribute in common, but have different behaviour: Suns don't move, planets orbit the sun and moons orbit planets…

Making use of the concepts we've been discussing in lectures, **refactor** the program you wrote in Task 2 to create the most **elegant**, **reusable** and **extensible** solution you can. More specifically:

- Create class(es) to represent at least the three types of solar object listed above.
- Your classes should model the state of each object in your solar system.
- Write suitable constructors for each of your classes.
- Write method(s) to update the position of an object according to its velocity.
- Take care to structure your classes such that they contain little or no repeated variables or code.
- Write a suitable main method that uses your new classes and methods to render an animated model of our own solar system (the sun, Mercury, Venus, Earth, Mars, Jupiter, Saturn , Uranus, Neptune and associated moons (n.b. the Galilean moons will do for Jupiter – no need to do all the little ones!).
- Think carefully about how you will store instances of your classes and show how polymorphism can help to make code simpler and more elegant.

If you have completed this basic specification, develop your software further:

- Enhance your program to model the asteroid belt and Saturn's rings to demonstrate the scalability of your program. The asteroid belt is a set of asteroids orbiting the Sun between Mars and Jupiter. It contains over 700,000 asteroids measuring 1km in diameter or larger (although you don't have to model quite that many!!). Your code should pay particular attention to achieving this through reuse of your existing OO classes, and maintaining elegant, scalable code.
- Support objects that do not have perfectly circular orbits, such as comets.
- Ensure your classes are well documented using Javadoc.
- Consider other user experience features such as accelerated time, zoom, etc.

Remember to apply everything you have learned about elegance and best programming practice since arriving at Lancaster. Enrolling on a new course doesn't mean you should forget the lessons from previous courses!

Portfolio Contribution

As discussed in the introductory lecture, all practical work this term will contribute to your portfolio assessment.

This particular piece of work will carry few marks for core functionality. I expect your program to work (programs failing to meet the basic specification will not receive a passing grade). Marks will be allocated based on the elegance of the work you submit, taking into account your use of OO design, encapsulation, inheritance, polymorphism, code style and the general professionalism of the work. Essentially, marks will be allocated based on the non-functional characteristics of your work.

Hacker Edition

There are many graphics libraries out there that are far more capable than the simple SolarSystem class I have provided.

As part of a **new** project (don't delete your solution using SolarSystem – you'll need to provide that for assessment!), why not review some of those, see what object model they have (the classes and inheritance hierarchy they use), and try to port your solution to work on one of those. It might be especially fun to try a 3D graphics library, that would provide you with the ability for enhanced UI features like 3D zoom, 3D rotation, texture mapping etc.

For example, Java contains a package called JavaFX that supports advanced 2D and 3D graphics. You can find a simple standalone example that shows a 3D cube on one of my git repositories. This works fine on the lab machines/windows without the need to install any further software.

Resources

- Joe's example repo, including JavaFX libraries: https://github.com/finneyj/Java3DSamples
- Open JavaFX Homepage https://openjfx.io
- Planetary Textures:
    - https://www.solarsystemscope.com/textures/
    - http://planetpixelemporium.com/planets.html