**title1.png**

**SCC212: Advanced Programming**

**Student Lab Book**

**Name: Adam Cornforth**

**Table of Contents**

Keep an up to date Table of Contents here.

**Unit 1: OO Programming**

**Exercise 1: Modeling the Solar System**

**Expected Time to Complete: 2 weeks.**

**Aim of the Exercise**

This week we'll be putting some of the theory about object orientation you learned last term into practice. You’ll be writing some of your own classes build them into your own **inheritance hierarchy**. You’ll also be taking some classes we’ve written for you, and integrating them with creating your own classes. **The aim is for you to gain experience of developing a well-designed OO program.**

**The Task**

Your task this week is to use OO concepts to create a simple moving model of the solar system, as illustrated below. This isn’t a course on GUI programming though, so 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. You are free to use any IDE (or command line) as you see fit. I’ve provided the source code for the **SolarSystem** class here for you to use, but you are **not expected to change (or understand) the internals of this code.** In fact, for this exercise, **you are not permitted to modify the SolarSystem.java file… just create instances of it.**



**Step 1: Exploring the SolarSystem Class.**

Based on the provided JavaDoc, write a simple *rendering* program wholly contained within a main method 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.

**Hint**: You’ll need to read and experiment with the **drawSolarObject** and **finishedDrawing** methods.

**Solution:**

**class** **Driver** {

**public** **static** void main(String[] args) **throws** InterruptedException {

SolarSystem solarSystem = **new** SolarSystem(900, 700);

**for** (int i=0; **true**; i++) {

*// Draw the sun*

solarSystem.drawSolarObject(0, 0, 40, "YELLOW");

*// Draw the earth*

solarSystem.drawSolarObjectAbout(180, i, 20, "BLUE", 0, 0);

*// Draw the moon*

solarSystem.drawSolarObjectAbout(180, i, 5, "GRAY", 30, i\*5);

*// Refresh the rocks and wait*

solarSystem.finishedDrawing();

Thread.sleep(10);

}

}

}

**Analysis: limitations – extensibility & scalability**

The issue with this simple solution is that if the model of the solar system included more planets, this would mean keeping track of many distance, angle and size variables just to move planets around if you want to introduce more planets to the model.

Further, it would be best to encapsulate suns, planets and moons into their own classes. This would help if for example you wanted to change the colour of all the moons in the program, or if you wanted to make a moon orbit it’s planet at all times even if the planet changes position (you would pass an instance of the planet class to the moon’s object instance which contains in it the planet’s distance and angle data for accessing by the moon on every angle refresh).

**Step 2: Building an OO Model of the Solar System**

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 Klingons but let’s keep it simple for now!). Note that these are similar in many ways - they all have a name, colour, diameter and location, but have different behaviour. Suns don't move, planets orbit the sun and moons orbit planets.

* Create a class to represent a **point in space**.
* Create a class to represent each of the three types of solar object listed above, including colour, position and size information.
* Include a ‘speed’ variable in each relevant class to model orbital velocity.
* Write method called ‘move’ in each relevant class that updates the modelled position of the object according to its speed. You can assume the move method will be called periodically…
* Write suitable constructors for each of your classes.
* Write a suitable main method that uses your new classes and methods to render a moving model of the inner solar system (the sun, Mercury, Venus, Earth, Mars and associated moons).

**Solution (Please see next page) :**

**Driver.java**

**class** **Driver** {

**public** **static** void main(String[] args) **throws** InterruptedException {

SolarSystem solarSystem = **new** SolarSystem(1000, 800);

*// Initialise the sun*

Sun sun = **new** Sun(solarSystem, 0, 5, 0, "YELLOW");

*// Initialise planets (solarSystem, distance, diameter, velocity, colour)*

*// And moons (solarSystem, planetAbout, distanceFromPlanet, diameter, sidereal period, colour)*

Planet mercury = **new** Planet(solarSystem, 0.1, 0.38, 0.2, "PINK");

Planet venus = **new** Planet(solarSystem, 0.5, 0.95, 0.6, "WHITE");

*// Init earth + moon*

Planet earth = **new** Planet(solarSystem, 1, 1, 1, "BLUE");

Moon moon = **new** Moon(solarSystem, earth, 0.3, 0.2, 27, "LIGHT\_GRAY");

*// Init mars + moons*

Planet mars = **new** Planet(solarSystem, 2, 0.53, 1.9, "RED");

Moon phobos = **new** Moon(solarSystem, mars, 0.2, 0.1, 0.3, "LIGHT\_GRAY");

Moon deimos = **new** Moon(solarSystem, mars, 0.1, 0.1, 1.319, "LIGHT\_GRAY");

*/\*\**

*\* Redraw the sun, planets and moons using a main angle which corresponds to earth's current angular location*

*\* Every other planet and moon will have different angles depending on their velocity (angle/velocity)*

*\*\*/*

**for** (double angle = 0; **true**; angle += 0.05) {

sun.move(0);

mercury.move(angle);

venus.move(angle);

earth.move(angle);

moon.move(angle);

mars.move(angle);

phobos.move(angle);

deimos.move(angle);

solarSystem.finishedDrawing();

}

}

}

**SpaceObject.java:**

**abstract class SpaceObject {**

int sunSize = 30;

double distance, angle, diameter, velocity;

String colour;

SolarSystem solarSystem;

*/\*\**

*\* Create a SpaceObject that has upscaled diameters and*

*\* distances within. Distances, diameters and velocities*

*\* are inputted in relation to the Earth's distance,*

*\* diameter and velocity (which are all 1).*

*\*\*/*

**public** SpaceObject(SolarSystem solarSystem, double distance, double diameter, double velocity, String colour) {

**this**.solarSystem = solarSystem;

**this**.angle = 0;

**this**.velocity = velocity;

**this**.colour = colour;

*/\*\**

*\* Scale the distances and diameters up to pixel sizes.*

*\* Notice that the distance is increased to accommodate*

*\* for the sun*

*\*\*/*

**this**.diameter = (double)Math.round(((diameter \* 10) \* 1) / 1);

**this**.distance = (double)Math.round(((distance \* 45) \* 1) / 1) + ((distance > 0) ? **this**.sunSize : 0);

}

*/\*\**

*\* To be implemented by subclasses. Implemented differently*

*\* in each as the sun is drawn in the same place at all times,*

*\* planet just updates its angle, and the moons update their*

*\* angle relative to their planet*

*\*\*/*

**abstract** **public** void move(double angle);

}

**Sun.java:**

**class Sun extends SpaceObject {**

*/\*\**

*\* Create the sun using drawSolarObject with the passed*

*\* parameters*

*\*/*

**public** Sun(SolarSystem solarSystem, double distance, double diameter, double velocity, String colour) {

**super**(solarSystem, distance, diameter, velocity, colour);

**this**.solarSystem.drawSolarObject(**this**.distance, **this**.angle, **this**.diameter, **this**.colour);

}

*/\*\**

*\* Sun does not need to move in this solar system model,*

*\* so distance and angles are just hard coded in*

*\*/*

**public** void move(double angle) {

**this**.solarSystem.drawSolarObject(0, 0, **this**.diameter, **this**.colour);

}

}

**Planet.java:**

**class Planet extends SpaceObject {**

*/\*\**

*\* Create a planet using drawSolarObject with the passed*

*\* parameters*

*\*/*

**public** Planet(SolarSystem solarSystem, double distance, double diameter, double velocity, String colour) {

**super**(solarSystem, distance, diameter, velocity, colour);

**this**.solarSystem.drawSolarObject(**this**.distance, **this**.angle, **this**.diameter, **this**.colour);

}

*/\*\**

*\* Draw the planet again, updating based on the angle passed.*

*\**

*\* The angle is updated using the planet's set velocity. The*

*\* planet's angle is updated relative to earth's. If the*

*\* planet has no velocity, the angle is set to 0.*

*\*/*

**public** void move(double angle) {

**this**.solarSystem.drawSolarObject(**this**.distance, **this**.angle = (**this**.velocity != 0) ? angle/**this**.velocity : 0, **this**.diameter, **this**.colour);

}

}

**Moon.java:**

**class Moon extends SpaceObject {**

Planet planetAbout;

*/\*\**

*\* Create a moon using drawSolarObjectAbout with the passed*

*\* parameters. drawSolarObjectAbout is populated with the*

*\* passed planet's distance and angle (planetAbout)*

*\**

*\* Instead of passing a velocity relative to earth into*

*\* the moon, a sidereal period is passed instead (easier*

*\* to find on the web). This value is then converted to a*

*\* velocity that is relative to Earth's. This makes for*

*\* easier initialisation of moons*

*\*/*

**public** Moon(SolarSystem solarSystem, Planet planetAbout, double distance, double diameter, double sidereal, String colour) {

**super**(solarSystem, distance, diameter, sidereal/(365/planetAbout.velocity), colour);

**this**.planetAbout = planetAbout;

**this**.solarSystem.drawSolarObjectAbout(planetAbout.distance, planetAbout.angle, diameter, **this**.colour, **this**.distance, 0);

}

*/\*\**

*\* Draw the moon again, updating based on the angle passed.*

*\**

*\* The angle is updated using the moon's set velocity. The*

*\* moon's angle is updated relative to its planet. If the*

*\* moon has no velocity, the angle is set to 0.*

*\**

*\* The moon's distance has the size of the sun deducted from*

*\* it because it orbits a planet rather than the sun like*

*\* the planets do.*

*\*/*

**public** void move(double angle) {

System.out.println("Planet about angle: " + **this**.planetAbout.angle);

**this**.solarSystem.drawSolarObjectAbout(**this**.planetAbout.distance, **this**.planetAbout.angle, **this**.diameter, **this**.colour,

**this**.distance - **this**.sunSize,

**this**.angle = (**this**.velocity != 0) ? angle/**this**.velocity : 0);

}

}

**Analysis:**

Describe the benefits you gained by creating these new classes. Describe the OO principle that brought these benefits.

Identify any parts of your code you feel are inelegant (particularly in terms of repetition and bloat), and explain why.

What examples are there of bad programming practice in the SolarSystem API? Explain why, and how these could be corrected…

**Step 3: Refactoring for Scalability and Elegance**

Redesign your code using inheritance to add scalability and elegance to your classes… More specifically:

* Introduce an inheritance hierarchy into your classes, in order to remove any repeated variables and code in your classes.
* Show how you can use a single array in your main method to hold references to instances of all your classes, and how this can be used to minimize the complexity of your rendering program.
* Extend your program to include a model of Jupiter (and its moons) and the asteroid belt 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!). Model the asteroid belt by placing a large number of asteroids randomly between Mars and Jupiter. Your code should pay particular attention to achieving this through reuse of your existing OO classes…

**Solution:**

**<<Paste your solution in here. Use any additional media you feel useful>>**

**Analysis:**

Sketch a diagram of your inheritance hierarchy. It should show the key methods and instance variables in each class.

Describe the OO principles that are introduced in this revision, giving examples of where and how your code uses them.

**Reflection:**

Record here any feedback you have gained on your work, along with your own analysis on what you believe you have learned. List any weaknesses you are aware of in your work, and explain how you plan to address them.

**Unit 1: OO Programming**

**Exercise 2: Bomb Sweeper**

**Expected Time to Complete: 1 week.**

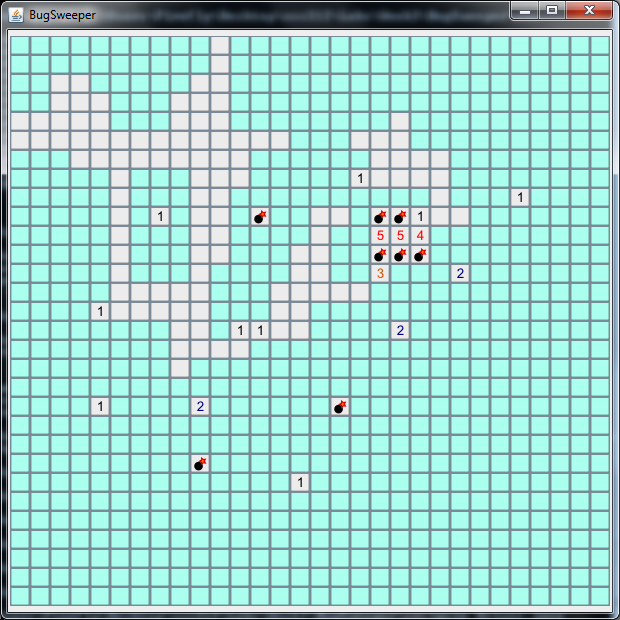
**Aim of the Exercise**

This exercise focusses upon testing and strengthening your understanding of object references, inheritance and implementing recursive algorithms. **The aim is for you to gain experience of writing more algorithmically complex code in Java while maintaining code elegance through iteration and recursion.**

**The Task**

Your task this week is to implement the logic behind the well-known game of minesweeper. The graphical functionality has been provided for you, as has a template for a (reasonably!) elegant object model. For this task, you should only need to modify the **SmartSquare** class, which provides a basic template for your coursework.

**You are not permitted to modify the GameBoard or GameSquare classes. Restrict your work to the SmartSquare class, and/or any other new classes you may wish to create.**



**Step 1: Detecting Bombs**

Review the Javadoc documentation for the provided classes, and study the **SmartSquare** source code. Note that the **SmartSquare** class is a specialisation of a **GameSquare** that also holds application specific information. In this case, note that in addition to its inherited instance variables, the **SmartSquare** class holds information on whether or not a given square contains a bomb. This is generated by a random process in the constructor, such that 1:10 squares contain a bomb.

Based on the provided JavaDoc and library code, develop the **SmartSquare** class such that when a square is clicked it, it obeys the following rules. **Try to implement this code in as elegant a fashion as you can.**

* Displays an image of a bomb if that square has a bomb on it.
* Displays the number of bombs in surrounding squares otherwise. Note that surrounding squares a classified as the eight squares that are horizontally, vertically or diagonally adjacent to the square.
* Your program must work for a board of any size or shape (you can modify this in the main method in the Driver class).

**Hint**: You’ll need to study the methods and instance variables you have inherited from the **GameSquare** class.

**Solution:**

**import** **javax.swing.\***;

**import** **java.awt.\***;

**import** **java.util.\***;

**public** **class** **SmartSquare** **extends** GameSquare

{

**private** boolean clicked = **false**;

**private** boolean thisSquareHasBomb = **false**;

**public** **static** **final** int MINE\_PROBABILITY = 10;

**public** SmartSquare(int x, int y, GameBoard board)

{

**super**(x, y, "images/blank.png", board);

Random r = **new** Random();

thisSquareHasBomb = (r.nextInt(MINE\_PROBABILITY) == 0);

}

**public** void clicked()

{

**this**.clicked = **true**;

**if**(hasBomb()) {

setImage("images/bomb.png");

} **else** {

System.out.println("From square x:" + **this**.xLocation + " y:" + **this**.yLocation + " Max = x: " + (**this**.xLocation + 1) + " y: " + (**this**.yLocation + 1));

setImage( "images/" +

bombSweep(0, **this**.xLocation + 1, **this**.yLocation + 1, **this**.xLocation - 1, **this**.yLocation - 1) +

".png");

}

}

**public** int bombSweep(int bombs, int maxX, int maxY, int currentX, int currentY)

{

SmartSquare square = (SmartSquare)board.getSquareAt(currentX, currentY);

*// Only want to check for bombs if the square we're on is actually a square*

**if**(square **instanceof** SmartSquare) {

*// Increment bombs if the square has one*

**if**(square.hasBomb())

bombs += 1;

}

*// If we're on the last square, just return the bomb count*

**if**(currentX == maxX && currentY == maxY)

**return** bombs;

*// If we're at the end of an X column, we need to increment the X position and reset the Y position*

**if**((currentX == maxX - 2 || currentX == maxX - 1) && currentY == maxY)

**return** bombSweep(bombs, maxX, maxY, ++currentX, currentY - 2);

*// Traverse an X column*

**if**(currentX <= maxX && currentY <= maxY)

**return** bombSweep(bombs, maxX, maxY, currentX, ++currentY);

**return** bombs;

}

**public** boolean hasBomb()

{

**return** thisSquareHasBomb;

}

}

**Analysis:**

Did you find it was challenging to elegantly implement these rules? Explain why or why not, emphasizing areas you feel are elegant or inelegant.

The GameBoard class is responsible for creating the instances of GameSquare (or its subclasses). In the current design, GameBoard has an explicit reference to SmartSquare in its constructor to build instances of your class. Describe a way you could modify the code such that GameBoard can be made independent of the exact subclass of GameSquare being used, and hence be future proofed … *be specific!*

**Step 2: Expanding into Empty Space**

Traditionally, minesweeper implementations enhance the playing experience by reducing any unnecessary work of the player.

Using recursion, update your program such that when a square is clicked that has zero bombs in its surrounding squares, it not only reveals that square, but also any horizontally or vertically adjacent squares that also have zero bombs in its surrounding squares.

**Solution (SmartSquare.java):**

**import** **javax.swing.\***;

**import** **java.awt.\***;

**import** **java.util.\***;

**public** **class** **SmartSquare** **extends** GameSquare

{

**private** boolean clicked = **false**;

**private** boolean thisSquareHasBomb = **false**;

**private** SmartSquare[] adjacent = **new** SmartSquare[9];

**public** **static** **final** int MINE\_PROBABILITY = 10;

**public** SmartSquare(int x, int y, GameBoard board)

{

**super**(x, y, "images/blank.png", board);

Random r = **new** Random();

thisSquareHasBomb = (r.nextInt(MINE\_PROBABILITY) == 0);

}

**public** void clicked()

{

**this**.clicked = **true**;

**if**(hasBomb())

setImage("images/bomb.png");

**else**

setImage("images/" + bombSweep(0, **this**.xLocation + 1, **this**.yLocation + 1, **this**.xLocation - 1, **this**.yLocation - 1) + ".png");

}

**public** int bombSweep(int bombs, int maxX, int maxY, int currentX, int currentY)

{

SmartSquare square = (SmartSquare)board.getSquareAt(currentX, currentY);

*/\*\**

*\* 1. Count bombs, save adjacent squares*

*\* Only want to check for bombs if the square we're on is actually a square*

*\* Also need to add selected square to an array of adjacent squares for traversal*

*\* later.*

*\*\*/*

**if**(square **instanceof** SmartSquare) {

**if**(square.hasBomb())

bombs += 1;

*// Add square to adjacent array (each x/y coord maps to a unique 0-8 array index)*

*// This is so that if we have to we can traverse their adjacent squares (and recursively so)*

adjacent[((currentX - (maxX - 2)) \* 3) + (currentY - (maxY - 2))] = square;

}

*/\*\**

*\* 2. Traverse more squares if needed, return final bomb count*

*\* If we're on the last square, just return the bomb count and check adjacent*

*\* squares if there's no bombs surrounding the selection*

*\*/*

**if**(currentX == maxX && currentY == maxY) {

SmartSquare[] tempAdjacent = **new** SmartSquare[9];

*// Copy adjacent squares to temp variable and then*

*// clear main adjacent squares array*

**for** (int i=0; i < 9; i++) {

tempAdjacent[i] = adjacent[i];

adjacent[i] = **null**;

}

*// No bombs surrounding our square? Traverse and reveal adjacent squares & their bomb adjacency counts*

**if**(bombs == 0) {

*// Need to check if each adjacent square has bombs*

**for** (int i=0; i < 9; i++) {

*// Check if adjacent square is actually a square and not a boundary, and check if it has been clicked already*

**if**(tempAdjacent[i] **instanceof** SmartSquare && !tempAdjacent[i].clicked) {

**if**(!tempAdjacent[i].hasBomb()) {

tempAdjacent[i].clicked = **true**;

tempAdjacent[i].setImage("images/" + bombSweep(0, tempAdjacent[i].xLocation + 1, tempAdjacent[i].yLocation + 1, tempAdjacent[i].xLocation - 1, tempAdjacent[i].yLocation - 1) + ".png");

}

}

}

}

*// Return final total number of bombs surrounding selected square*

**return** bombs;

}

*/\*\**

*\* 3. Traverse around our clicked square and count bombs*

*\* Traversing around the current square, done column by column:*

*\* If at the end of an X column, we need to increment the X position*

*\* and reset the Y position.*

*\*\*/*

**if**((currentX == maxX - 2 || currentX == maxX - 1) && currentY == maxY)

**return** bombSweep(bombs, maxX, maxY, ++currentX, currentY - 2);

*/\*\**

*\* Need to then traverse an X column*

*\*\*/*

**if**(currentX <= maxX && currentY <= maxY)

**return** bombSweep(bombs, maxX, maxY, currentX, ++currentY);

*// Return current total number of bombs surrounding selected square*

**return** bombs;

}

**public** boolean hasBomb()

{

**return** thisSquareHasBomb;

}

}

**Analysis:**

Explain the operation of your recursive algorithm.

Describe the challenges you would face in implementing this algorithm iteratively.

**Reflection:**

Record here any feedback you have gained on your work, along with your own analysis on what you believe you have learned. List any weaknesses you are aware of in your work, and explain how you plan to address them.

**Unit 1: OO Programming**

**Exercise 3: Performance Analysis of Java Collections API**

**Expected Time to Complete: 1 week.**

**Aim of the Exercise**

This exercise requires you to undertake little programming. Instead, it focusses upon evaluating the overall performance characteristics of several of the Java Collections data structure implementations and identifying their relative strengths and weaknesses. **The aim is for you to appreciate the breadth of support the collections API offers, and to gain an understanding of its limitations, particularly as the size of your data structures grow.**

**The Task**

Your task this week is to write a simple test harness in Java that can measure the performance characteristics of the **ArrayList**, **LinkedList**, **HashSet** and **HashMap** data structures.

**Step 1: Temporal complexity of insertion, deletion, contains and find operations**

Temporal complexity concerns itself with the amount of time a computing operation takes to complete. This is often calculated theoretically, but we’re going to do it experimentally…

For this task:

* Download the simple Person class provided with this assignment. Study the class and note it contains a String representation of a name and an integer representation of age. Note the use of method overriding of the **toString()**, **equals()** and **hashcode()** methods.
* Write a simple main method that creates lots of instances of the provided Person class and adds them to a generic instantiation of a Collection (e.g. an **ArrayList**). Make it so you as a programmer can define how many instances to create.
* Add some code to your main method to measure how long your insertion code takes to execute. You might find the **System.currentTimeMillis()** and/or **System.nanoTime()** methods useful… note: operating system clocks aren’t that accurate (an can be as accurate to only 10ms). For small operations, you may therefore need to execute them ‘N’ times, measure the total time, then divide by ‘N’ to determine an accurate execution time…

**Solution:**

**<<Paste your solution in here. Use any additional media you feel useful>>**

**Analysis:**

For the **ArrayList**, **LinkedList**, **HashSet** and **HashMap** data structures, undertake the following performance analysis. In all cases, use any appropriate methods provided by these data structure classes to provide best performance.

* Measure and record the time taken to perform add operations. You should record the results in a spreadsheet. Plot a graph of how long the various data structures take to insert items in the list. You are free to use as many sample points as you feel is necessary to generate useful results (but I expect you’ll be generating results for collections containing a million instances or more!).
* Measure, record and graph the time taken to determine if a given instance is in the collection. You should choose an instance to find that is midway through the data structure.
* Measure, record and graph the time taken to retrieve the Nth item in the collection. You should find an item that is half way through the collection. e.g. if you have 1000 items, lookup item 500. You should only refer to an item by its location for this measurement, and this test only applies to the List data structures.
* Measure, record and graph the time taken to find an instance with a given name.
* Measure, record and graph the time taken to remove an instance from a collection.

Based on your results, provide a short analysis of what you have found. Give an example of an application each data structure would be well suited to.

**Reflection:**

Record here any feedback you have gained on your work, along with your own analysis on what you believe you have learned. List any weaknesses you are aware of in your work, and explain how you plan to address them.