Week 3, Lab A – Inheritance

# Lab Intro & Prep

This lab will use **Processing** to illustrate the concepts of inheritance. Some starter code will be provided, which we will refactor and improve as we work through the lab exercises.

## Learning Objectives

* Understand inheritance and how to use it to develop new classes that are based on existing classes
* Download **RandomShapes.zip** from the Week 3 Moodle area and extract it to a suitable location

# Exercise 1 – RandomShapes

Run the supplied program and observe the output (Fig 1). You should see stars and circles that move around the canvas in a pleasing manner.

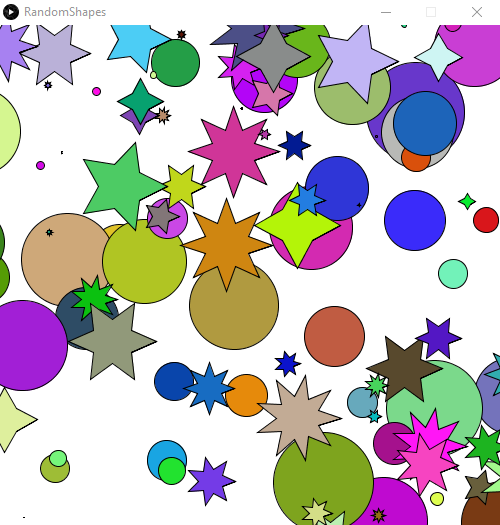


Fig 1 - RandomShapes Output

## Program Overview

This subsection provides an overview of the supplied code before delving into the exercise. If you have read through the code and understand it, you can go straight to Exercise 1.

Currently, the **RandomShapes** program has two classes – Circle and Star – the UML Class diagrams for these classes are depicted in Fig 2.

The **Circle** class contains 6 variables to record the position of each circle, the size, colour, and an xt and yt variable (the xt and yt variables will store the location the object will move to). The class also contains methods to move, render, and update – each of these methods do what you would expect.

The **Star** class contains the same 6 variables and 3 methods, but also contains a variable to record the number of points a star object has.

The main tab contains two arrays – one to store **circles**, the other to store **stars**. Each of these can store 50 of their respective objects (note: the arrays will store references to objects, not the objects themselves). Two loops in the setup procedure deal with creating 50 circle objects and 50 star objects, respectively. The draw procedure renders a white background and then loops through each of the arrays (using a for-each / enhanced for loop) and calls the update method on each object in the array.

Recall from the lecture, we can place common attributes and methods that similar classes have in common and move them into a base class (referred to as a superclass).

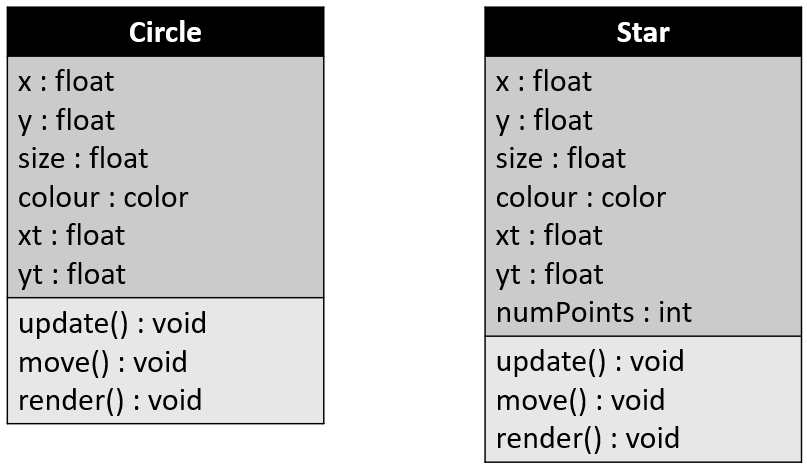


Fig 2 - RandomShapes Program UML Class Diagrams (Circle and Star)

## Part 1 – Introducing a Shape Superclass

1. Create a new tab named **Shape**. In this tab, create a class named **Shape**
2. Remove the variables that the **Circle** and **Star** class have in common and place these into the **Shape** class – this will serve as the superclass for the Circle and Star classes
3. Add the relevant code to the top of the **Circle** and **Star** class so that they both **extend** the **Shape** class. This will cause both of these classes to inherit any variables and methods in the **Shape** class
4. Every **Circle** and **Square** moves in the same way – move the **move** method to the Shape superclass and remove these definitions from the **Circle** and **Star** classes
5. The update method is also the same for the **Circle** and **Star** classes – place this method in the superclass and remove it from the subclasses
6. You will notice that the **Shape** class does not have a render method, meaning render is underlined red in Processing, since it is not defined in the Shape superclass. Add a **render** method to the **Shape** class that does nothing (has an empty body) – we will see how we can improve this later.
7. Add the constructor below to the **Shape** class. This constructor will be responsible for initialising a shape object’s x, y, size, and colour variables – something that every Shape has

Shape( float x, float y, float size, color colour) {

this.x = x;

this.y = y;

this.size = size;

this.colour = colour;

// set a target to move towards

xt = x;

yt = y;

}

1. We can now modify the constructors in the **Circle** and **Star** classes. Recall from the lecture that a class can call the constructor of its immediate superclass via the **super** keyword. In our case, the superclass constructor deals with x, y, size, and colour
   1. **Circle** class – replace the code in the **Circle** constructor with a single line of code that calls the superclass’ (Shape) constructor
   2. **Star** class – replace the code in the **Star** constructor with a line of code that calls the superclass’ constructor – your **Star** constructor should take care of initialising numPoints still – since not every Shape has that

At this point, your program should resemble the UML diagram depicted in Fig 3.

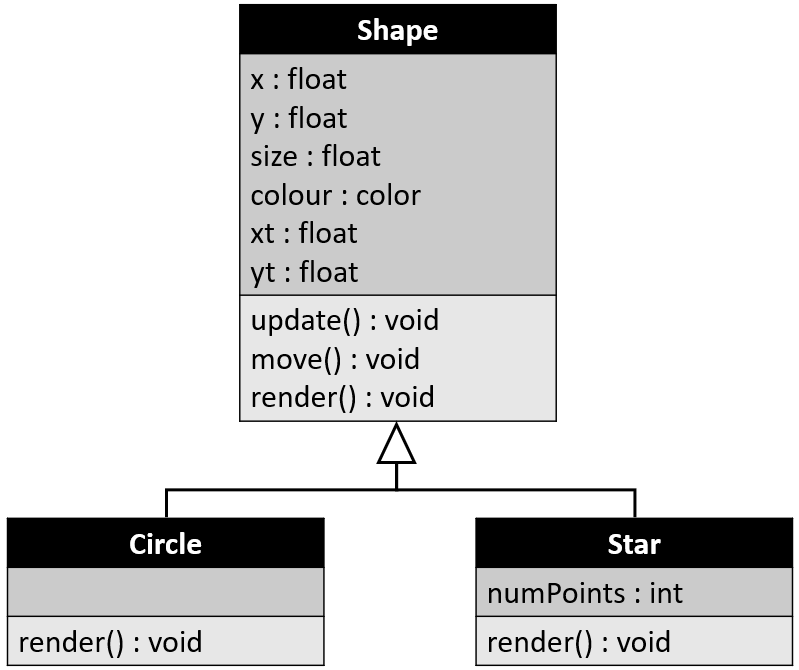


Fig - RandomShapes Inheritance Design

# Exercise 2 – Square Class

We will now create a new class which will inherit from the **Shape** class.

1. Create a new tab named **Square** and class named **Square** that *extends* the **Shape** class.
2. Define a constructor that accepts floats for x, y, size, colour (this constructor should call the superclass constructor)
3. Override the inherited **render** method so that it draws a square (you could either use the built-in square command or rect)
4. Override the inherited **move** method so that the movement of a square differs to that of a **Circle** or **Star** (e.g. you can move the x and y by random values)
5. Create an array of 50 squares in the main tab, create the objects using a for loop within setup, and then use a loop within draw to render the squares

# Exercise 3 – Declaring abstractness

Currently, the **render** method of the **Shape** class has an empty body. Every subclass of the **Shape** class overrides this method – which makes sense since every type of Shape *looks* different – but what exactly does a **Shape** *look* like?

Since the render method of the Shape class does not provide a concrete implementation, we can declare it as **abstract.** The moment you declare any method in a class as being abstract, the entire class must be declared abstract – as the class contains some incomplete implementation.

1. Replace the empty **render** method in the **Shape** class with the line of code below:

**abstract void** render();

Since abstract methods do not provide implementations – they do not have bodies, meaning we only provide the method header, followed by a semi-colon. It will be up to the subclasses of the Shape class to provide implementation details for this method (which all of our subclasses do).

1. As mentioned earlier, the moment we declare a method as abstract, we must declare the entire class as abstract - because the class now contains some incomplete implementation. Add the abstract keyword before the class keyword:

**abstract class** Shape

What happens If you try to create an object of the Shape class now?