EECS 1720 – W2023
LAB 4 :: Inheritance & Polymorphism

Prerequisites – Labs 1-3

Lab Resources:

Java API: https://docs.oracle.com/javase/8/docs/api/
java.awt.Graphics2D (see Java API link)
java.awt package (see Java API link)
java.awt.geom (see Java API link)
Random API https://docs.oracle.com/javase/8/docs/api/java/util/Random.html

Lab Files: http://www.eecs.yorku.ca/course_archive/2022-23/W/1720/labs/lab4/lab4.zip

STEP 1: Importing an Archived Project

In this step, you will import an archived project into your Eclipse workspace, (the files for each lab exercise are embedded). Each question refers to the completion/modification of specific java file(s). The instructions for what is required for each question are included in the document below (STEP 2 onward).

a. You can download the lab project file from the link above (see Resources)

   This file is a *zip file*, also known as an *archive file*. Click on the link below to open the URL in your browser. In the dialog that opens, choose **Save**, not **Open**. This file is typically saved into your *home* or *downloads* folder.

b. Open Eclipse and import this archive
   IMPORTANT: this process is the same as the process you will use in programming tests, so please make sure you follow it, so that you do not have submission difficulties during the lab tests.

   DO NOT DOUBLE CLICK ON THE FILE TO OPEN. By importing this project, you will add it into your *EECS1720* workspace. Do this by:

   i. Open Eclipse (**Applications → Programming → Eclipse**) & set/choose your workspace
   ii. Close "Welcome" tab, and navigate to **File → Import → General → Existing Projects into Workspace.** Hit "next"
   iii. Choose the archive file (zip file you downloaded in (a)). After selecting, hit **Finish**, and the file will open up as a project in your Project Explorer.
   iv. We are now ready to proceed with the Lab Exercises.

STEP 2: Lab Exercises

The exercises in this lab are pitched at building a family of Alien classes that each have a different look and behaviour and can be instantiated and used by a main class BattleField to simulate the "arena" in which a Space Invaders game might take place. You will use inheritance in this lab and will specifically explore polymorphism and polymorphic behaviour.

Exercise 01 (Familiarization of Battlefield and extension of Alien Class):

Goal: Building on the concepts of lab3 (where we built a class representing a Ghost from the classic game "Pacman", the goal of this lab is to create a family of similar (but slightly different) Alien objects for a simplified version of the classic game "Space Invaders".

In the project file for lab 4, a set of three classes are already provided for you: `Alien`, `Hero` & `BattleField`. The Alien class specifies an alien object, that can be instantiated and positioned somewhere on an application window. In this lab we will continue working with the RasterImage class (to define an application window) similar to the windows we have been using in lab3 (however, it is not a huge step to work with the Canvas GUI object placed within a JFrame (instead of a RasterImage object) – *for now, this will be left as a challenge at the end of this lab (as this is still to be discussed more in lectures).*

The Hero class specifies a simple object that represents the player (usually this moves left and right, and can shoot). Finally the BattleField class just represents the game window, which will hold all the characters within it (a bunch of aliens at the top of the window, and the hero at the bottom of the window). In Space Invaders, the aliens normally move in a left-to-right, top-to-bottom pattern and gradually approach the hero, whose job it is to destroy the aliens before they reach the hero (the aliens usually speed up as they get closer too). There are usually other obstacles/objects as well (that act as protection for the hero), but not in our simplified version.

Since there are classes already implemented, you **are asked to work from the project shell provided for this lab** (rather than create your own from scratch).

First, we will run the `BattleField` class (it has a main method implemented). This will generate and display an array of `Alien` objects (that are all instantiated using default color settings (black), arranged in a grid pattern on the `RasterImage` object). Note, the alien class has a drawAlien method that defines and uses polygons to draw the shape of the alien. In practice, this could also be an image on a JLabel or JButton (if we had created an image in another software for example). For now let's keep it simple to focus on inheritance.

In this exercise, familiarize yourself with these classes, then write a few lines of code in your main method in `BattleField.java` so that a random alien from the array "aliens" is selected, and the alien is re-coloured to be `Color.RED`. To choose a random alien, to choose a random column and row from the aliens array (use the ***Random*** class to do this).

Look at the API of `Random` to determine how to get the next random value (there are different methods depending on whether you want the next random number to be an int, double, etc.). This class is much more convenient to use than Math.random().

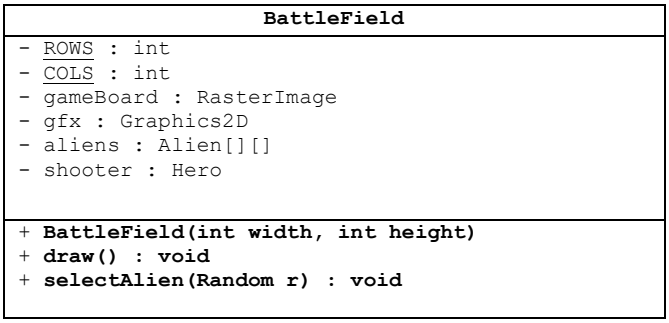
Run your program a couple of times to ensure that it is randomly choosing and painting a different Alien red (only one should be red, the rest black – all with the same geometry).

Keep in mind, after you set the alien's colour property, you will need to re-draw it (see the methods provided in the Alien class for this).

Exercise 02 (modify/reorganize your BattleField class):

In this task, you will modify your `BattleField` class so that it now stores the `RasterImage`, `Graphics2D` objects, `Alien` array, and `Hero` objects as **class variables** (i.e. fields of the class). You should also store the size of the `RasterImage` as class variables.

Complete any setters and getters that would be needed to access and modify the private fields.



Now complete the following:

1. Create/complete a **custom constructor** for the `BattleField` class (as per the UML diagram above) that creates a `RasterImage` by accepting the size of the `RasterImage` (i.e. width and height) as arguments, and creates the same setup of aliens as given in the project file for lab 4 (in main method of BattleField).

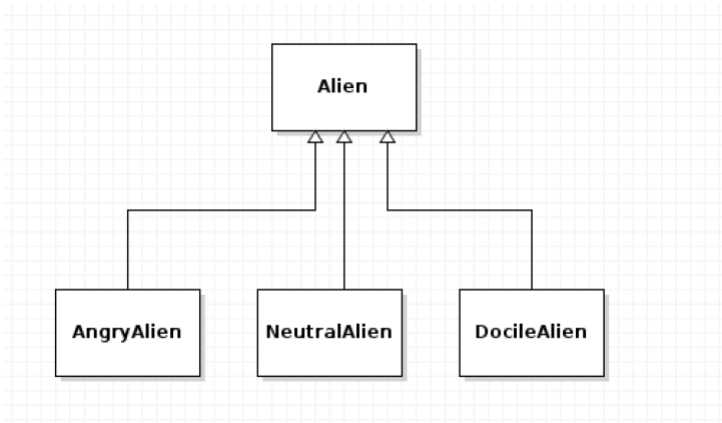
2. Create/complete the method **`draw`**`()` that will draw the objects in the `BattleField`
    a. Note: it is a good idea to:
        i. clear the `RasterImage` or `Graphics2D` object so that the image in the window is blank, then
        ii. make a series of calls to re-draw the `Battlefield` objects (i.e. traverse the array and draw all the aliens, then draw the Hero)

3. Test your constructor and draw method by instantiating a new `BattleField` object in BattleField's main method, using the custom constructor, then invoking the draw method on this BattleField instance.

4. Move your code to select a random Alien to paint red, into the method **`selectAlien(Random r)`.**

Exercise 03 (create a family/hierarchy of Aliens using Inheritance):

In this task, you are required to create three different subclasses of the `Alien` class, as per the following UML diagram:



The idea is that each class will represent a different type of Alien. Now, for the most part, what differs about each Alien will be the way that each is drawn and the way that each advances on the Hero.

A UML diagram for the parent class is given below:

Alien
- position : Point2D.Double - col : Graphics2D - width : double - height : double
+ Alien(Point2D.Double) + Alien(Point2D.Double, double, double) + Alien(Alien) + getColor() : Color + setColor(Color col) : void + getWidth() : double + setWidth(double) : void + getHeight() : double + setHeight(double) : void + getPos() : Point2D.Double + setPos(Point2D.Double) : void + drawAlien(Graphics2D) : void + toString() : String

For this task, create/complete three new classes:

`AngryAlien`, `NeutralAlien`, `DocileAlien`.

Each of these Alien sub-classes should inherit most of the functionality of the `Alien` class, however, you will have to configure two custom constructors that allow the so that, by default:

a. `AngryAlien` is RED, `NeutralAlien` is GREEN, and `DocileAlien` is BLUE
    a. Note, you will need to make calls to superclass constructor (using super), to create the Alien part of each. Note, you need to have constructors that have similar parameters as the parent classes constructors.
    b. You can then use one of Alien's mutator methods, to modify the colour field (note this field is private, and it's access properties should not be modified)

b. Each of these classes **overrides** the *toString* method
    a. So that a suitable text representation of each class object can be outputted as a string to the console.
    b. Hint: you may need to use Color's *toString()* method to get a string representation of the color and then modify/replace parts of the string to get the desired output.
    c. Note: Alien's version of *toString()* does not do everything that is needed (however it can still be useful)
    d. Examples of outputs are shown below:

```
// if a client did the following, this would result in the output shown below:
Alien a = new Alien(new Point2D.Double(100,300));
Alien a1 = new AngryAlien(new Point2D.Double(100,300));
Alien a2 = new NeutralAlien(new Point2D.Double(100,300),65.7,131.56);
Alien a3 = new DocileAlien(new Point2D.Double(100,200),11.5,-56.3);

System.out.println(a.toString());
System.out.println(a1.toString());
System.out.println(a2.toString());
System.out.println(a3.toString());
```

Output:

```
Alien: pos= Point2D.Double[100.0, 300.0], size= 50.0, 40.0
AngryAlien: pos= Point2D.Double[100.0, 300.0], size= 50.0, 40.0, col= [r=255,g=0,b=0]
NeutralAlien: pos= Point2D.Double[100.0, 300.0], size= 65.7, 131.56, col= [r=0,g=255,b=0]
DocileAlien: pos= Point2D.Double[100.0, 200.0], size= 11.5, -56.3, col= [r=0,g=0,b=255]
```

c. Each of these classes **overrides** the *drawAlien* method
    a. So that each alien has a different physical appearance on the screen
    b. Stick to the same dimensions as with the previous lab, but create quite different looking aliens for each subclass (different shapes)
        i. An example of different possible Alien geometries (for subclasses) is provided in the figure at the end of the next step (part d).
    c. Have a look at the way a polygon is defined in the original Alien class, try to use a polygon to define your alien objects (but make them each different).

d. You may test each by creating a client class, or creating and displaying an individual alien in the main method of each of these subclasses

d. Finally, **create a second custom constructor** for the `BattleField` class (one that accepts an integer for the number of different subclasses of Alien to consider). This constructor creates the same type of Alien array, however, randomly creates several different types of Alien.

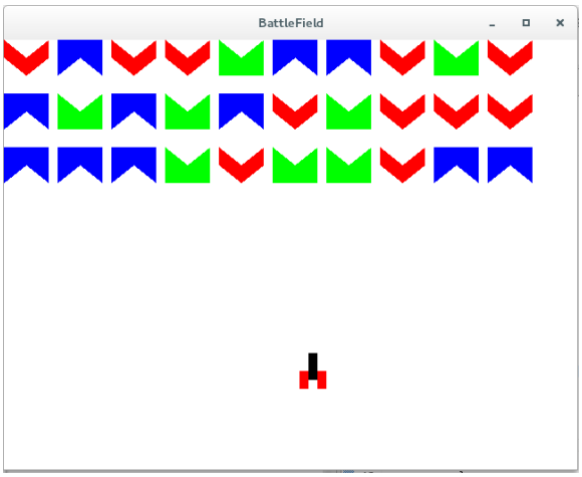
BattleField
- ROWS - COLS - gameBoard : RasterImage - gfx : Graphics2D - aliens : Alien[][] - shooter : Hero
+ BattleField(int width, int height) **+ BattleField(int width, int height, int n)** + draw() : void + selectAlien(Random r) : void

HINT: you may create an array of type Alien, and then instantiate a specific type of Alien to be assigned:

use the integer passed to define a range for a random integer (e.g. from 0-2) that can be used in a switch statement to instantiate and add one of the three possible Alien subclass objects to the aliens array. Remember, an object of a child class (subclass) can be legally substituted in for an object of a parent class (superclass).

** it is not important where you instantiate specific types of aliens in the grid, just that there are some locations that have different aliens.

Note: the draw method of *BattleField* should call the appropriate version of `drawAlien` depending on the object. *[this should happen automatically via dynamic dispatch (the original version of draw should not need to be modified for this Exercise) – this is an example of polymorphism – i.e. the Alien objects all take on different behaviour, depending on the sub-classes actually instantiated]*



e. create a method in the Battlefield class called "**`advanceEnemy()`**", that will move all of the aliens down toward the hero. This method should invoke the "**`advance()`**" method on each Alien object in the BattleField. Depending on the type of Alien instance in aliens[][], the following behaviours should be supported (implying that the advance() method of the Alien parent class is *overridden* by each subclass):

Alien → move down by 20 pixels
AngryAlien → move down (vertically) by 60 pixels
NeutralAlien → move down (vertically) by 20 pixels (as per Alien instances)
DocileAlien → move right or left by 20 pixels

This method should cause the repositioning of all the aliens instantiated when a BattleField object was initially constructed, and it should refresh and redraw the entire scene after the alien objects have been moved. To see this effect of a single call to advanceEnemy(), run the whole program once before moving; then run again showing the scene after a call to advanceEnemy(). **Save the images** of these two windows (e.g. use a screenshot), and submit with your solution.

BattleField
- ROWS - COLS - gameBoard : RasterImage - gfx : Graphics2D - aliens : Alien[][] - shooter : Hero
+ BattleField(int width, int height) + BattleField(int width, int height, int n) + draw() : void + selectAlien(Random r) : void + **advanceEnemy() : void**

Alien
- position : Point2D.Double - col : Graphics2D - width : double - height : double
+ Alien(Point2D.Double) + Alien(Point2D.Double, double, double) + Alien(Alien) + getColor() : Color + setColor(Color col) : void + getWidth() : double + setWidth(double) : void + getHeight() : double + setHeight(double) : void + getPos() : Point2D.Double + setPos(Point2D.Double) : void + drawAlien(Graphics2D) : void + toString() : String **+ advance() : void**

Extra Challenge:

For an extra challenge, you may try to modify your BattleField class so that instead of storing a RasterImage object as an internal class field, you instead make it extend the Canvas class (see Lecture 12 slides, pages 40-43). This alone will not create an application window, but you can either create a JFrame object in main() or create a new client class that extends JFrame, and add the BattleField instance (which IS-A Canvas) into this class. Just like the example in the lecture slides.

Note: To get a `Graphics2D` reference from the `Canvas` object (or `JFrame`) to pass to your existing draw methods, you can query your class instance using the `getGraphics()` method – which returns a `Graphics` reference. This can easily be cast to a `Graphics2D` reference, and used to pass to your draw methods for the Alien family of classes (or the `Hero` class):

```
Graphics2D gfx = (Graphics2D) this.getGraphics();
```

Also, it is best to override Canvas's paint() method (i.e. add it to your `BattleField` class) – see lecture slides – and have that call the original draw method you had defined in `BattleField`.

STEP 3: Submission (Deadline: Tuesday 14th March, 2023, 11:59pm)

You will formally submit the java files associated with all tasks. SUBMIT ALL OF THE *.java FILES (**NO ZIP FILES OR CLASS FILES!!**) to the "lab4" directory on web submit.

Expected files:

```
BattleField.java
Hero.java (unchanged file)
Alien.java
AngryAlien.java
NeutralAlien.java
DocileAlien.java

** plus two image files (screenshots relating to part e)
```

Web-submit can be found at the link: https://webapp.eecs.yorku.ca/submit/

Note, the terminal (in lab or on remotelab) can also be used by navigating to the project directory with the source files to submit, and typing:

```
submit 1720 lab4 *.java *.jpg
```

*.java = all java files in that directory, *.jpg represents any jpg images (or include another extension if saved as some other format, such as *.pdf, *.docx, *.png, etc.)