**Tutorial 2**

At the end of this tutorial you should be comfortable with the following:-

* Building a drawing app with Gradle and IntelliJ
* Refactoring strategies for more expressive code.
* More advanced Git - resolving Git source conflicts

**This learning in Exs1-4 of this tutorial is part of the Implementation section of Software Engineering for Bioengineers. We are going to write a drawing app. Drawing in Java windows is direct, immediate and fun, and so will give us motivation to get coding and learn more Java.**

**Ex1 Creating a Circle Drawing project with IntelliJ using the Gradle Build Engine**

This exercise will use the Java AWT library. This library will allow us to create a window and draw into it.

An AWT window is created by instantiating an object of class Frame.

<https://docs.oracle.com/javase/7/docs/api/java/awt/Frame.html>

A Frame can then have an object of class Canvas added to it. This is the main drawable area of the frame.

Now we can start to build the drawing program.

1. Create a new Gradle Java project and update build.gradle to include the ‘application’ settings as we did in tutorial 1. Again, add a Main class and a main method entry point, and share the project to a private github repository.

We will now create a class called Drawing. It will inherit from the AWT class Canvas.

<https://docs.oracle.com/javase/7/docs/api/java/awt/Canvas.html>

1. Create a class called Drawing that extends Canvas and overrides its paint method. Here is an example:-

**public void paint(Graphics g) {  
 g.setColor(Color.*black*);  
 g.drawLine(0, 0, 100, 100);**

**}**

1. In the main method, instantiate an object of class Frame and an object of class Drawing.
2. Add the Drawing object to the Frame object with Frame’s ‘add’ method.
3. Call the following methods of the Drawing object:
   1. setSize with the desired width and height of the window
   2. setBackground to the desired Canvas colour – eg **Color.WHITE**
4. Call the following methods of the Frame object:
   1. setLayout with null (we do not need a layout manager when there is only one component)
   2. setSize with the same width and height as Drawing
   3. setVisible with true
   4. addWindowListener as follows:-

addWindowListener(new WindowAdapter() {*// Ends program if close window is clicked*

**public void** windowClosing(WindowEvent e) {

f.dispose();

}

});

1. Build and run the program – you should see the Frame appear, and it should close when you click the Window’s close button.

NOTE: if you want to know how this windows program works, and especially how the ‘paint’ method is called, it is described in Appendix A.

Now we can add some circles to the Drawing.

1. Create a class called Circle.
2. It should have three private fields
   1. one of type int to hold the circle’s radius
   2. one of AWT class Point to hold the circle’s position co-ordinates
   3. one of AWT class Color to hold the circle’s colour
3. Create a constructor that has three arguments of the same types and classes above, and which initializes all three fields with their values.
4. Create a public method in Circle called – eg ‘draw’ – that takes a Graphics object as an argument and draws a circle using that Graphics object’s ‘fillOval’ method: use values taken from the position and colour fields. It should set the drawing colour first with the Graphics object’s setColor method using the colour field.
5. Add a private field of class Circle to the Drawing class.
6. Create a constructor in the Drawing class that instantiates that Circle field, passing into its constructor instantiated objects of class Point and Color, as well as a value for the radius.
7. In Drawing’s paint method, call the Circle’s draw method, passing on the Graphics object that is passed into the paint method.

Note:the code above uses three AWT classes: Point, Color and Graphics

Point is used to specify a 2D location in int co-ordinates

[https://docs.oracle.com/javase/7/docs/api/java/awt/Point.html](https://docs.oracle.com/javase/7/docs/api/java/awt/Point.htmlhttps://docs.oracle.com/javase/7/docs/api/java/awt/Point.html)

Color is used to specify a colour. There are several different constructor overloads for a Color object:

<https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html>

eg instantiating an object col of class Color as follows:

Color col=new Color(45,87,34);

specifies the constructor that takes three ints: [**Color**](https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html#Color(int,%20int,%20int))(int r, int g, int b)

whereas instantiating an object col of class Color as follows:

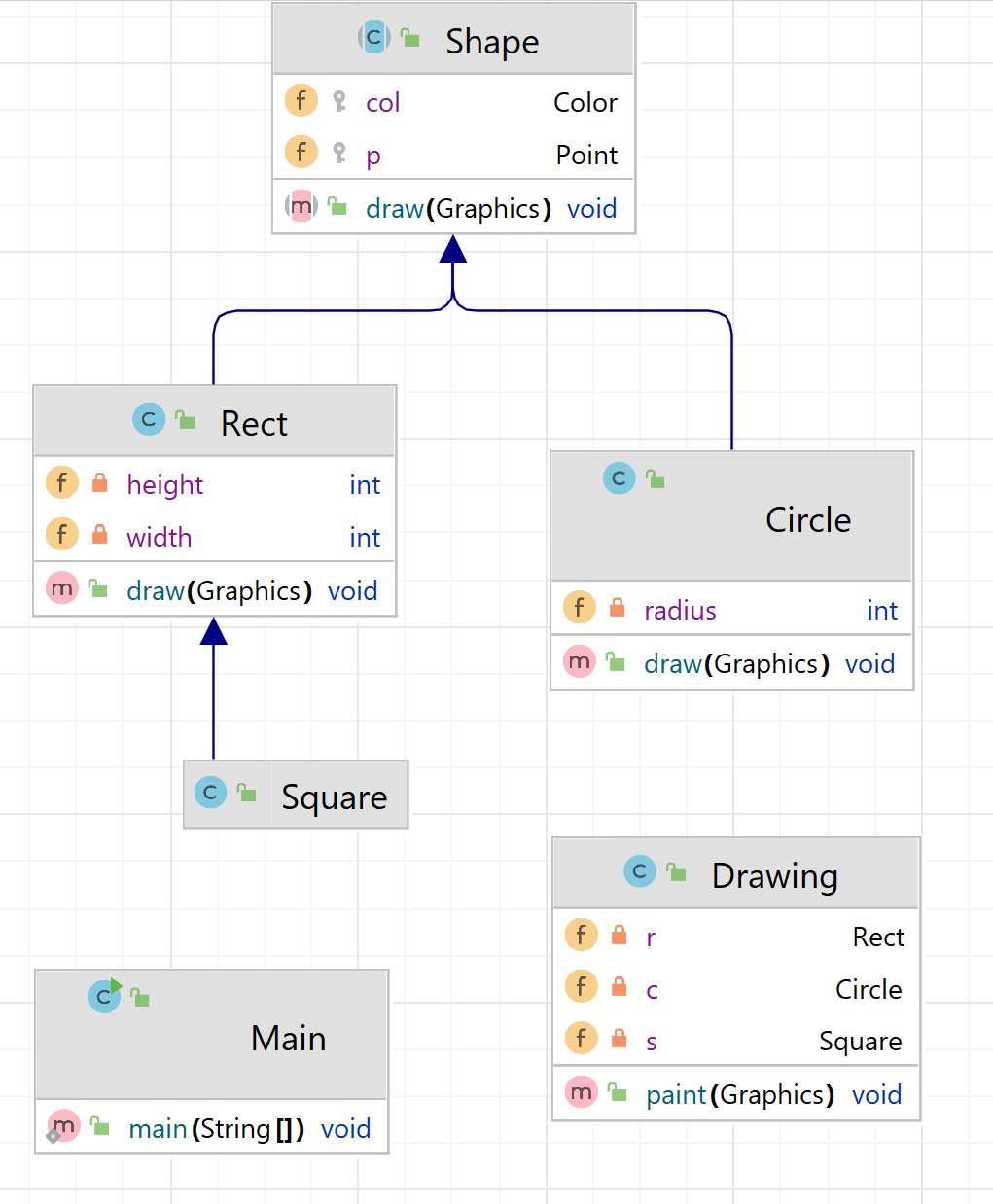
Color col=new Color(0.4,0.3,0.7);

specifies the constructor that takes three floats: [**Color**](https://docs.oracle.com/javase/7/docs/api/java/awt/Color.html#Color(float,%20float,%20float))(float r, float g, float b)

1. Commit, tag it as v1.0 (Git->Tag) and push the current changes to Github (Git->Push (make sure ‘Push tags’ is checked). You should see on Github that there is now a ‘release’ of the project available.

**Ex3 Drawing Rectangles and Refactoring to get a Shape Superclass**

1. Write a class called Rect that does the same as circle, but draws a rectangle of width w and height h. (note the Graphics method to fill a rectangle is fillRect(**x**, **y**,**width**,**height**);)
2. Add a Rect to Drawing so that both the Circle and Rect are drawn. Create a new tag, and commit.
3. There is common code to Circle and Rect that could go in a more abstract superclass called Shape. We could extract this superclass by hand, but IntelliJ has a refactor command to help. Have the Circle class open in IntelliJ: Click Refactor->Extract->Superclass, select the fields and methods common to Circle and Rect (ie draw (check the abstract box), colour and position) and name the superclass ‘Shape’.
4. Change Rect so that it also extends Shape (this has to be done manually), and delete the fields that have now moved to Shape. You will need to call Shape’s constructor using ‘super’, passing in the colour and position from Rect’s constructor.
5. Build and run to make sure that it work correctly.
6. Create a Square class that extends Rect. (hint: it should comprise a constructor containing one line and nothing else!)
7. Right click on the src->main->java folder in the IntellJ Project view on the LHS and click Diagrams->Show Diagram->Java Class Diagrams (assuming you are using the Ultimate version of IntelliJ). You should get something like this (you might have to click m and f to show fields and methods) which shows the inheritance chains:-



1. Git Add and Commit your project – tag it v1.1 and push to the remote repo.

**Ex4 Create a Package and Re-organize Code**

1. Right-click on the src folder in the project view and select New->Package. Call it shapes.
2. Select Shape, Circle, Rect and Square and drag them onto the new package. Agree to the Refactor and click “Do refactor” at the bottom of the screen when prompted.
3. The shapes should all have **package** shapes; at the top, and Drawing should import them.

We will now explore some more of the Collaboration section of Software Engineering for Bioengineers with some more advanced use of Git:-

**Ex 5 Resolving Git Conflicts**

1. Make a change to one of your .java file in the remote repo, and also change the same .java file (but in a different way!) in your local working folder. There are now two versions of the file in existence, your changed version and a version that someone else – perhaps a collaborator – has changed!
2. Add and commit your local changes to your local repo
3. Do a ‘git pull’. It should warn of a conflict, and not perform a merge of local and remote
4. Check the .java file in IntelliJ – it should show both changes with some text to show which change came from where. Change the code to resolve the conflict as you want (you can include one or the other change, or both, or neither or put in a completely different change), making sure that you remove the >>>>>>>, ===========, <<<<<<<<< indicators and any other text added in the comparison.
5. Add and commit your resolved file.
6. Push back to the remote repo. Check that the remote version is the resolved version.

**Appendix A**

**Windows Programs**

How does this program work? A program with a window has a different execution model to programs we have written before. Our programs so far have run from the beginning to the end, executing instructions in the order that we have specified, and then exit. Windows programs, however, are driven by input from the user when they interact with the window. Windows programs respond to ‘events’ such as mouse clicks, mouse moves, menu items, buttons etc when the user selects them. This execution model is as follows:-

A diagram of a process

Description automatically generated

The loop that waits for events and processes them is called a Message loop, or Event loop. It runs continually whilst the window is open. The Message loop is sent events by the computer operating system when they happen.

When writing Windows programs, the primary philosophy is to define how we respond to user interaction, rather than define a top-down sequence of actions.

The program you have written works as follows: Canvas and Frame are classes in the Java ‘Abstract Windows Toolkit’ (AWT). This is a library for creating windows programs. A Frame object creates a window, and a Canvas object is a rectangular area that can be drawn on, and that can be ’added’ into the client area of a Frame.

A diagram of a square with a frame and text

Description automatically generated with medium confidence

Here is a flow-chart of your program – starting in ‘main’. Note that there are two ‘threads’. A thread is a path of execution through the code. We are used to there only being one thread in our programs, but in Java it is possible to start multiple threads of execution that can run in parallel **at the same time**. (These days, different threads might run on different processor cores):-

A diagram of a process

Description automatically generated

When the program runs in ‘main’, it sets up a Frame object, and also a Drawing object (Drawing inherits from Canvas – ie Drawing ‘is a’ Canvas). The Drawing object is added into the Frame object as a drawable area.

When the main thread calls the method ‘setVisible’ in the Frame object, the Frame object starts a new thread of execution that runs in parallel to the main thread of execution. This new thread runs the Message loop.

Whilst the Message loop thread is running, the main thread continues with another method call but then comes to an end. However, the program itself won’t stop until the thread with the Message loop also comes to an end (ie the window is closed by the user).

All of the events received by the Message loop are processed either by the Frame object, or by the Drawing object.

Remember that the Drawing object ‘is a’ Canvas object, ie - it has inherited all of the public methods of the Canvas object.

The only method in Canvas that Drawing overrides and implements itself is the ‘paint’ method – ie the only event that we have written code to handle is the ‘Paint’ event. This event is automatically triggered when the window first appears. The Frame object knows that the Drawing object is responsible for drawing its client area, and so the Frame object tells the Drawing object to draw itself by calling its ‘paint’ method, passing in a ‘Graphics’ object that it has created to implement the drawing tools. Note: this means that you never have to explicitly call ‘paint’ yourself – it is called automatically by the Message loop in Frame.