In this assignment, you will create several kinds of graphical objects, some of which will be compositions of other objects. This work is currently independent of the scanning projects you have done so far. Later, the scanning and graphics objects will unite when scanned commands will be used to manipulate the graphics objects. As always, for each assignment, make a new project, copy the packages and classes from the previous project into this project (using the copy and paste commands provided by Eclipse), and then add the new classes to this project.

As we saw in the first class, the goal of your homework assignment is to create a simulation of the bridge scene in the movie, Monty Python and the Holy Grail. The following figure shows the one I created.

![Figure 1 My Implementation of the Simulation](image-url)
You know enough to create elements of the simulation except the lists of knights and the list of utterances of the knights and the guard. Several of the parts of the project will be extra credit.

In the previous assignments, the “what” was spelled out for you, and the challenge was the “how” – in other words, the interfaces and behaviors of the objects were given to you, and the challenge was to implement these interfaces. In this assignment, aspects of the interfaces you have to implement are not completely specified so that you can use your creativity to make design decisions. Once you have made these decisions, the implementation should be relatively straightforward, involving a simple application of the object composition and graphics concepts you have learned in class.

Thus, the figures shown here are not to be taken literally and are meant to help interpret the instructions, which you should read carefully to determine what is required and what is optional.

Each graphical class should follow ObjectEditor conventions so it can be displayed using ObjectEditor, as shown in Figure 1.

**Rotating Line (Extra Credit)**
Create a variation of one of the line classes and interfaces you saw in the lecture and recitation that has one or more methods for rotating the line around a fixed point (such as the upper left corner or the lower right corner). For example, you can define a method that takes as a parameter the angle by which it should rotate (clockwise or anticlockwise) or one that takes the angle to which it should rotate.

One way to implement such a line is to keep an internal instance of APolarPoint, which is not exported as a property, and rotates around the fixed point. This, in turn, implies that you have to work with two coordinate systems – one defined by the fixed point and another defined by ObjectEditor. This feature is not easy – hence it is extra credit.

**Movable Named Knight Avatar**
Create a knight avatar (by defining appropriate classes and interfaces) composed of at least four atomic shapes recognized by ObjectEditor. One of them is a label displaying its name. The other three represent its body parts (such as head, arms, legs) and objects it is carrying (such as a sword). The body parts and carried objects should be connected. The name can be displayed at any offset relative to other components.

The avatar should define one or more methods to move it in the X and Y direction. When an avatar moves, the relative distances between its components remain fixed. Thus, each component should move by the same amount. You can define different methods for moving in the two directions or a single method, and these methods can specify how many units (positive or negative) the avatar should move or the absolute coordinates of the new position. To
implement these methods, you may want to define a particular point in the avatar (such as the upper left corner of its head or the point at which the next meets the head) as its location.

Figure 2 A Movable Knight Avatar

**Movable Named Guard Avatar**
This avatar is much like a knight avatar except that it has at least one component that is different from a knight avatar. As you see in Figure 1, my guard avatar has a round head rather than a square one.

There will be a lot of overlap in the code for the two avatars. If your avatar is a composition of many objects, much of this code can be shared. Later you will use inheritance to increase the sharing.

I recommend you do this part after your knight avatar has been completed as otherwise you might end up duplicating changes in the two avatars.

**Composed Avatar (Extra Credit)**
You can define each kind of avatar using a single class or multiple classes. For instance, you can define a class for the arms for an avatar, which you can use in an unnamed avatar, which you can use in a named avatar, which you can use in a class that defines a named, sword carrying avatar. The more meaningful classes you compose in the implementation of an avatar, the more reusable your code will be. You will get extra credit based on the nature and number of classes you compose and reuse in the two avatars. Composition will be labour intensive, so do it only if you have time.

**Rotatable Named Avatar (Extra Credit)**
Define methods that can be used to rotate an avatar around some fixed point. If you want to implement this feature, it will be useful to compose it from rotatable lines and circles. Images in the avatar will not rotate.
Pose Changing Avatar (Extra Credit)

Define methods that can be used to change the pose of an avatar to simulate walking. For instance, I defined methods to change the angles between arms and legs, as shown below.

Gorge with Bridge

The avatars give you enough experience with object composition, so it is up to you how you create a gorge with a bridge. The simplest approach is to create two parallel lines to simulate a gorge, and a rectangle connected to the two lines for a bridge. You are free to give 3-D effects as I have tried to do, or create images for the gorge and/or bridge.

Standing Area

Create classes and interfaces that define the areas in which the guard and a knight stand. The simplest approach is to define a circle type, as I have done, for both areas. You are free to
create more sophisticated objects. There is one standing area for all knights, in which the front most knight can stand.

**Scene without Knights (Extra Credit)**
Create an object that simulates a scene without knights. The scene should have two standing areas, the gorge with bridge, and a guard. The guard should be in the guard standing area. You will have to build a scene with knights in the next assignment, so doing this work now will save you work later.

You can decorate the scene with other objects such as the two pictures shown in the Figure above and the sign post.

**Scrollable Scene without Knights (Extra Credit)**
Allow the scene to scroll in the X and/or Y directions by defining public methods that move all objects in it in the X and or Y directions.

**Tester**
Write a tester for the class implementing the knight avatar that checks that after the avatar moves, the relative positions of its components do not change.

**Logical and Physical Structure**
Draw the logical and physical structure of an instance of the knight avatar class using the conventions given in class. The best approach would be to draw the structures on your own first and then correct them if necessary by seeing how the debugger and ObjectEditor display them.

**Something Remarkable (Extra Credit)**
In all assignments, you will get extra credit for any work you do that goes beyond what is expected from a student – you are free to use your creativity here. Such work could include remarkable features in your simulation, and extraordinary ways to organize and test your code. Be sure to point out anything you think is remarkable – otherwise the TAs will try and identify it.

**Constraints**
As always, try and follow all style principles you have learned so far in class. Your ability to share code among the various classes you create will, however, be limited by the fact that we have not discussed inheritance and other reusability concepts. You should use only the concepts seen in class so far, and specified in earlier assignments. You can always use (parts of the) code presented in class.

In this, and all other assignments, your getters should not create objects. This means that you should not move an object by replacing it with a new one. For instance, do not
move the name of an avatar by creating a new label. Instead, change the coordinates of the object.

An additional constraint for this and future assignments has to do with packages. We earlier specified the name of the package of the main class: main. As its name says, this package should only have main classes. Group other classes into appropriate packages. In Eclipse, you can drag a class from one package into another to change its package. You should have at least three packages in this project (which, as mentioned before, should include classes from the previous assignment), and these packages should have meaningful names. Include all the scanner code you have written so far in the submission.

**Constraints on Logical Structure**

You may see error messages of the form:

```
E*** Object displayed multiple times: primitiveShapes.ACartesianPoint@29949f78. This time as component: location at position 0 of parent object: collections.AChatHistory@4a009ab0 whose path is:.chatHistory.
```

You get such an error message if in the logical structure of the top object submitted, there is an arrow (reference) to the top object, or at least two references to some other object in the structure, making ObjectEditor visit the object more than once. The second time it visits the object, it points out which index in a collection or property in a bean references that object. I will try and give the original reference also.

It does not matter whether the object is visible or not because in ObjectEditor the object can be made to dynamically appear and go away.

The easy way to fix the problem is to change the logical structure without changing the physical structure. Thus, in this example, there is a method called getLocation() in the 0th element of the chatHistory property of the top level object that returns the previously visited object: primitiveShapes.ACartesianPoint@29949f78. The fix is to get rid of the location property by renaming its getter. Instead of calling it getLocation(), call it something else, like location() or obtainLocation().

It may seem to you that the error message does not actually cause any problems. But they could occur later and sometimes an object could be displayed and other times it may not.

**Optional ObjectEditor Features**

Here are some ObjectEditor features you could use to spruce up your scene. If there is anything else you would like, send mail to comp401-help, and we may be able to help you.

Graphics objects such as lines, ovals, and rectangles can have the following properties given special meaning by ObjectEditor:
Filled: This is a boolean property. If its value is true, then an oval or a rectangle graphics object is filled.

Color: This property is of type java.awt.Color. It can be set to any instance of this class. Some standard values are provided as constants, such as Color.BLUE, Color.RED, and so on. It specifies both the color of the outline of the shape and its filling, if the shape is filled. Thus, if a rectangle has a Color property that is Color.BLUE and a Filled property that is true, then both its outline and its interior will be blue.

Stroke: This is a property of type java.awt.Stroke. It can be assigned any value of this type. In particular, you can assign it values of type java.awt.BasicStroke. For example, if this property has the value new BasicStroke(8.0f) you will get a very thick line.

Font: This property is of type java.awt.Font. It determines the font used in a label or textbox.

Rounded: This property is of type boolean and determines if the edges of a rectangle are rounded.

You may want control over how properties of an object are ordered in both the main window and graphics window. In the latter the ordering is important when the positions of two objects overlap and you want one displayed over another.

Given an object you can order its properties by putting before the getter of a property the annotation:

@Position(n)

where n is the order of the property.

You need the import:

import util.annotations.Position

A property (and its children) that has a smaller position number is shown above the one with a larger one.