Abstract Classes and Interfaces

Objectives

The main objective of this practical session is to develop a class hierarchy based on an abstract superclass.

Reference Material

This practical session is based on the *Abstract Classes and Interfaces* chapter.

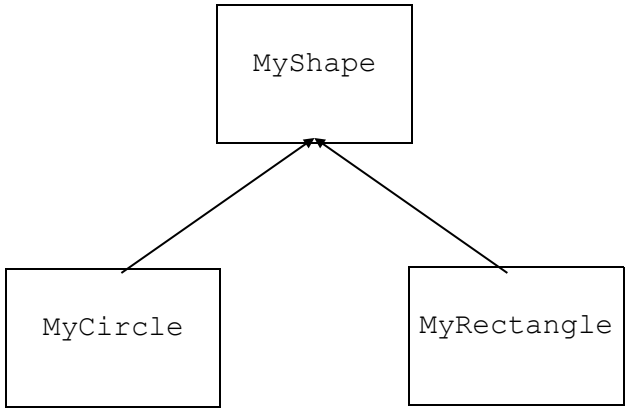
Overview

In this practical, you will use an abstract superclass to share functionality and to impose methods on subclasses. Later, you will implement an interface.

Practical

Extending an Abstract Class

In this exercise, you will define and implement the inheritance hierarchy shown below. `MyShape` is an abstract class, which will define the methods that all the subclasses have in common, such as `draw()`. It will also define the instance variables that are required by all subclasses, such as the top left hand corner co-ordinates, width and height. If necessary, its subclasses `MyCircle` and `MyRectangle` can define their own methods and instance variables.



1. You will notice in your starter folder that it includes four classes: `MyShape`, `MyCircle`, `MyRectangle` and `ShapeTest`.
2. Create a new IntelliJ project and copy these files in. Examine `MyShape`, which contains the skeleton code for the `abstract` class `MyShape`. Note that it defines two `protected` instance variables: `width` and `height`, which define the width

and height of a particular shape. The reason for the `protected` modifier is to make these instance variables visible to subclasses of `MyShape`.

3. Define a constructor that takes two arguments: `width` and `height`, and initialises the corresponding instance-variables.
4. Define an `abstract` method called `calculateArea()`. It does not need any parameters but it will return the calculated area of a shape. As you will discover later, all concrete subclasses of `MyShape` must implement this method.
5. Examine the class `MyRectangle`, which contains the skeleton code for the concrete class `MyRectangle`, a subclass of `MyShape`.
6. Define a constructor that takes two arguments: `width` and `height`. You can pass the values of these arguments on directly to the constructor of the superclass `MyShape`.
7. Implement the `calculateArea()` method that was defined as abstract in `MyShape`. (Hint: the area of a rectangle is defined as width*height)
8. The class `MyCircle` contains the skeleton code for the concrete class `MyCircle`, another subclass of `MyShape`.
9. Define a constructor that takes only one arguments: the `radius`. Pass this value twice into the constructor of the superclass `MyShape`.
10. For each concrete class override the `toString`() method so that we can print out information about the shape – what type of shape it is and what is the size.
11. Implement the `calculateArea()` method that was defined as `abstract` in `MyShape`. Use the `Math.PI` constant – the area of a circle is defined as $\pi * r^2$ where r is the radius.
11. Examine `ShapeTest`, which contains the skeleton code for the test harness `ShapeTest`. Notice that, it creates an array of four `MyShape`-type objects (two `MyRectangles` and two `MyCircles`). The reference to this array is stored in the instance variable, `myShapes`.
12. In the `main()` method, add a `for` loop to call the `calculateArea()` method of each shape in the array of `MyShape` objects.
13. Save the project and run `ShapeTest`.
14. Recognise you have two problems now, consider we want to create a class called `MyTriangle` that will have to have a `public double calculateArea()` method if it wishes to extend `MyShape`, what if it does not wish to offer `calculateArea` functionality but still be a '`MyShape`' for other purposes. Also if you were to now write a `MyTown` class with a lovely `public double calculateArea()` method you will not be able to put it in an array of type `MyShape[]` as `MyTown` will extend a different class and cannot be upcast to `MyShape`. Solution, introduce a `Computable` interface that `MyRectangle`, `MyCircle` and `MyTown` can choose to implement but which `MyTriangle` may choose not to implement.
15. Extra: create another class to represent a trapezoid.