# Inheritance

## Overview

In this lab you'll define an inheritance hierarchy of classes to represent different types of shape, such as rectangles, triangles, and circles. The lab will give you an opportunity to consolidate your understanding of how Kotlin inheritance differs from Java inheritance.

## Roadmap

There are 4 exercises in this lab, of which the last exercise is "if time permits". Here's a brief summary of the tasks you'll perform in each exercise; more detailed instructions follow later in this lab doc:

1. Define a basic shape class
2. Augment the shape class with various abstract members
3. Define concrete subclasses for rectangles and triangles
4. (If Time Permits) Investigate sealed classes

## Exercise 1: Define a basic shape class

Define a Shape class, which will act as the superclass for all kinds of shapes. Here are the requirements for this class:

* Define a primary constructor taking the (x, y) coordinates for the shape's top-left corner, and the (x, y) coordinates for its bottom-right corner.
* Define computed properties that get the width and height of the shape.
* Implement concrete methods named moveX() and moveY(), which move the shape by a specified delta amount horizontally and vertically respectively.
* Implement a concrete method named scale(), to scale the shape by a ratio. The caller should be able to pass in 1 or 2 parameters, something like this:
  + scale(1.5) - scale-up horizontally and vertically by 50%
  + scale(1.5,1.8) - scale-up horizontally by 50%, scale-up vertically by 75%
* Override the toString() method to return a textual representation of a shape.

Write some simple client code as follows:

* Create a Shape instance.
* Print the shape's details, via its toString method.
* Move the shape horizontally and vertically.
* Scale the shape.
* Display its new details.

## Exercise 2: Augment the shape class with various abstract members

Add the following abstract members to the Shape class:

* An abstract property named numPoints, indicating the number of points in the shape. All subclasses will have to override this property (e.g. a rectangle has 4 points, a triangle has 3 points, a circle has 0 points, etc.)
* An abstract property that gets computes the area of the shape.
* An abstract property that computes the perimeter of the shape.

Note the following general points:

* You'll have to declare the Shape class as abstract, so that it can contain abstract values and methods.
* This means you won't be able to create Shape instances in the client code, so just comment-out your code for now. You'll fix this in the next exercise…

## Exercise 3: Define concrete subclasses for rectangles and triangles

Define a Rectangle class that extends Shape. Here are some hints:

* Define a primary constructor that takes the top-left and bottom-right coordinates, and passes them up to the superclass constructor.
* Override the numPoints property. Set it to 4.
* Override the area property. The area of a rectangle (width \* height).
* Override the perimeter property. The perimeter is (2\*width + 2\*height).
* Override the toString method to indicate the shape is a rectangle. Then call the superclass toString method to incorporate the basic details about the shape.

Then define a Triangle class that extends Shape. Here are some hints:

* Define a primary constructor that takes the top-left and bottom-right coordinates of the bounding rectangle, and passes them up to the superclass constructor. (For simplicity, assume it's a right-angled triangle sitting nicely inside the bounding rectangle).
* Override the numPoints property. Set it to 3.
* Override the area property. The area of a triangle is (½ width \* height).
* Override the perimeter method. The perimeter is (width + height + hypotenuse). If you need a reminder about how to calculate the hypotenuse for a right-angled triangle, Pythagoras knows…
* Override the toString method to indicate the shape is a triangle. Then call the superclass toString method to incorporate the basic details about the shape.

Also enhance the basic toString method in the Shape class so that it incorporates the shape's number of points, area, and perimeter. When you access these abstract property from the Shape class, it should polymorphically reach down to the appropriate subclass and access the concrete members defined therein.

Now write some client code as follows:

* Create some Rectangle and Triangle instances.
* Print the details of each shape via toString (this method should print out the number of points, area, and perimeter info as appropriate for the given type of shape).

## Exercise 4 (If Time Permits): Investigate sealed classes

Tweak the Shape class definition so that it's a sealed class. Remind yourself what this means in Kotlin.

Define some specific behaviour for each type of shape. For example:

* In the Rectangle class, add a computed property that determines whether the rectangle is in fact a square (i.e. its width and height are the same). Return a suitable boolean result.
* In the Triangle class, add a computed property that encapsulates the calculation of the hypotenuse for the triangle.

Now define a method named processShape in your client code. The method should take any type of Shape as a parameter, and use when pattern matching to deduce its actual runtime type:

* If it's a Rectangle, return a string indicating its "is square" status.
* If it's a Triangle, return a string indicating its hypotenuse value.

Call processShape with a Rectangle parameter, then again with a Triangle parameter. Verify the function performs pattern matching successfully.

Then define a class for another type of shape, e.g. Circle. Add some type-specific functionality, e.g. a method to return the circle's radius. See what happens if you try to run the code now. You should get a compiler warning in the processShape method, indicating the when pattern matching expression doesn't cater for your new type of shape. How does the compiler know this…?

Update your processShape method so that it deals with your new shape type. The code should now compile. So invoke processShape, passing in an instance of your new shape type. Verify it displays the correct results.