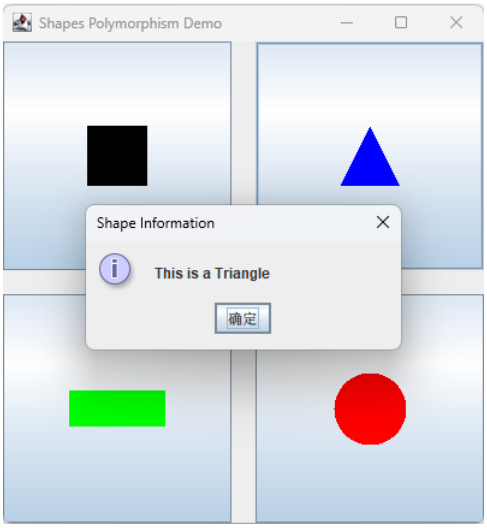
# [Week 13] Polymorphism and Abstract Class

## Tutorial

In this tutorial, we will use a Swing application to demonstrate polymorphism and abstract class. Before we get started, please download the sample code from BB. Execute the code in the `tutorial` package and you'll see the following window; when you click any shape, it pops up a message showing the name of the shape.




### 1. Abstract Class

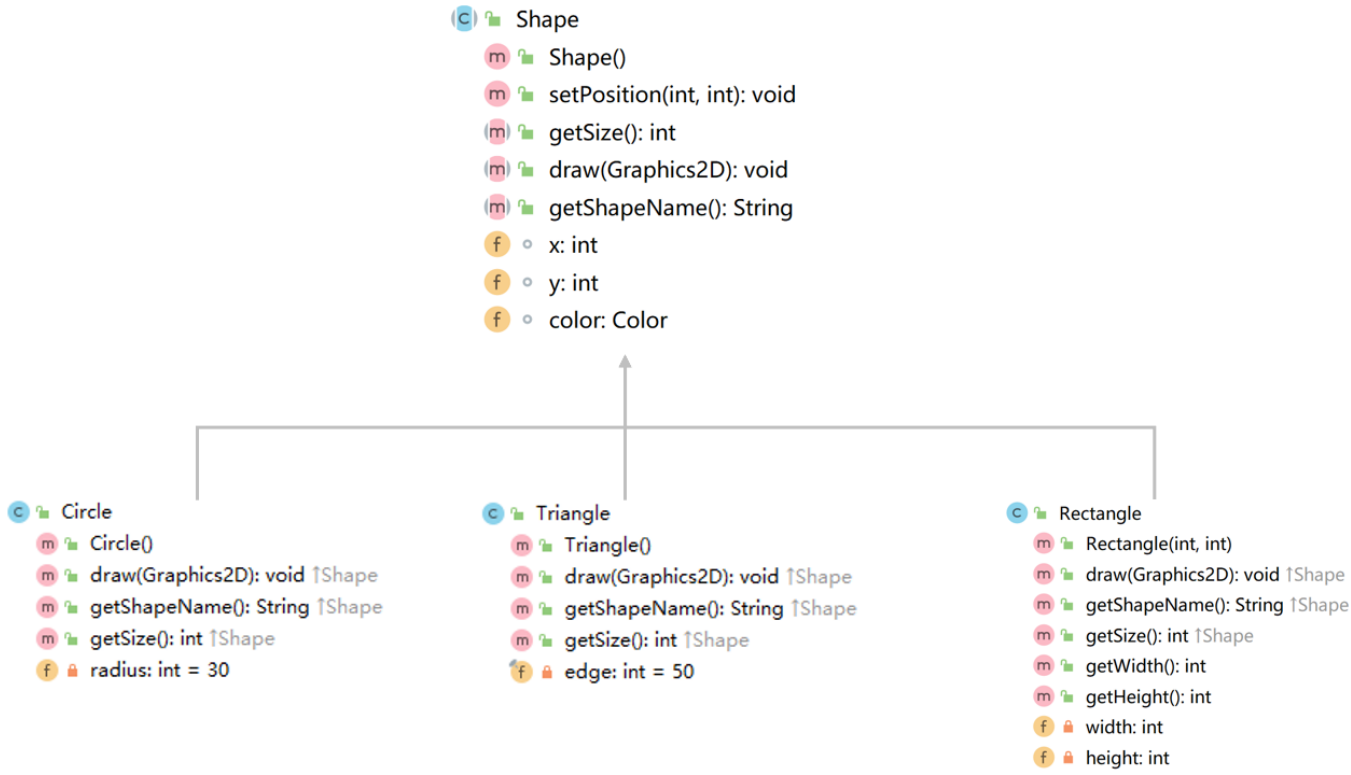
First, we define `Shape` to be an abstract class. It has abstract methods `getSize()`, `getShapeName()`, and `draw()`. Shape doens't have a concrete implementation for these methods, because we want to enforce all subclasses to implement them.

An abstract class can also define fields and concrete methods as normal classes. In this case, we define `x`, `y` and `color` as fields, and `setPosition()` as a concrete method.

### 2. Subclasses

We have three subclasses of `Shape`: `Circle`, `Rectangle`, and `Triangle`, as shown in the diagram below. We use private fields to represent distinct properties of different shapes, e.g., `radius` for circles, `edge` for triangles, `width` and `height` for rectangles. Note that, the `Rectangle` class can be used for squares as well, because a square is a special case of a rectangle (`width == height`).

For these subclasses to be concrete classes, they must implement all abstract methods in the superclass. Check out the code for details.



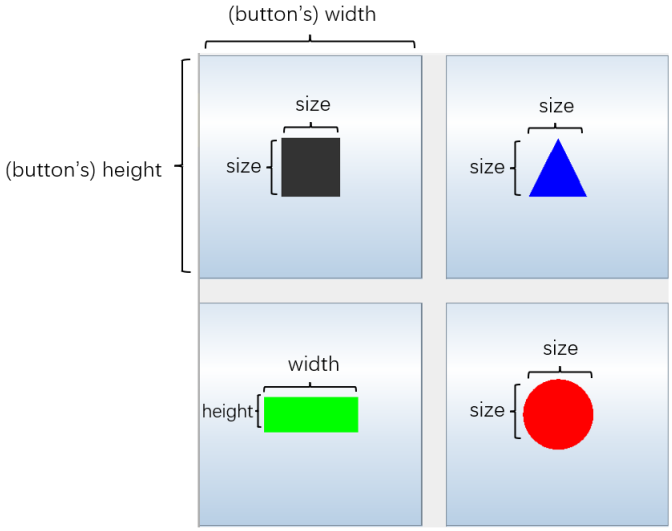

Note that, the `getSize()` method in `Shape` returns a shape's size, which will be used later to calculate the shape's position on the window. `Circle`, `Triangle` and `Rectangle` (as squares) override this `getSize()` method. Rectangles, however, may not be symmetric. Hence, we add `getWidth()` and `getHeight()` for it.

## 3. Polymorphism

Observe `DrawingPanel` for polymorphism usages. First, it creates four shapes using polymorphsim.

```
// Polymorphism array
Shape[] shapes = new Shape[]{
        new Rectangle(50,50), // square
        new Triangle(),
        new Rectangle(80,30),
        new Circle()
};
```

When we iterate the array and paint each `shape` at the center of each button, we use the `getSize()` as a polymorphism call for square, circle, and triangle. Since rectangles are not symmetric, we treat them differently: we check if the shape is a rectangle, then type cast it to a rectangle and use `getWidth()` and `getHeight()` to calculate the position of rectangles.




```
boolean isSquare = false;
if(shape instanceof Rectangle){
    Rectangle rect = (Rectangle) shape;
    isSquare = rect.getWidth() == rect.getHeight();
}

// Center shape in button
if(shape instanceof Rectangle && !isSquare){
    shape.setPosition((getWidth() - ((Rectangle) shape).getWidth())/2,
    (getHeight() - ((Rectangle) shape).getHeight())/2);
}
else{ // Polymorphism call on getSize()
    shape.setPosition(
        (getWidth() - shape.getSize()) / 2,
        (getHeight() - shape.getSize()) / 2
    );
}
```

In the code for pop-up dialog, the `getShapeName()` method is a polymorphism call that works for every shape in the shape array.

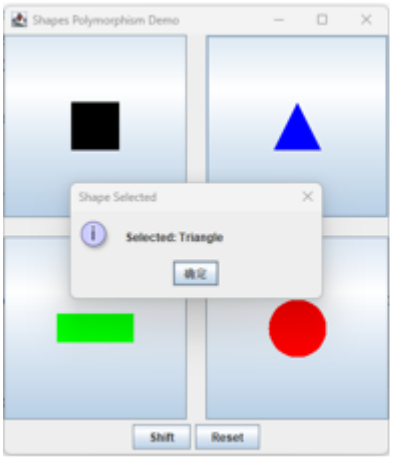
```
// Add click event. Polymorphism call on getShapeName()
buttons[i].addActionListener(e -> JOptionPane.showMessageDialog(
        this,
        "This is a " + shape.getShapeName(),
        "Shape Information",
        JOptionPane.INFORMATION_MESSAGE
));
```

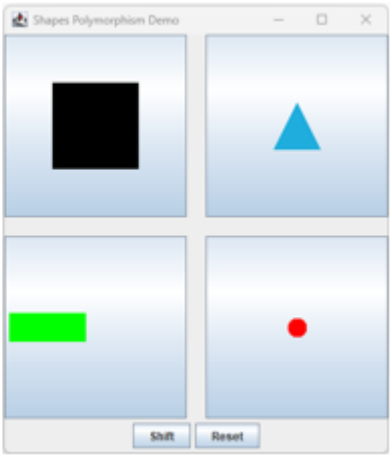
## 4. Layout

In this sample, we are using a 2x2 `GridLayout` to arrange the buttons. Finally, the `DrawingPanel` is added to the center of the frame.

# Exercise

1. Modify the UI to add a `shift` and a `reset` button.
2. Users could click any shape, which pops up a message showing that the shape is selected.
3. Click the `shift` button, which shifts the selected shape (if no shape is selected, should pop up a warning message):
    - Square: enlarge the shape.
    - Circle: change the color (randomly).
    - Rectangle: move to the left.
    - Triangle: shrink the triangle.
4. Click the `reset` button, which resets all shapes to their original positions, sizes or colors.




We have provided the incomplete code in the `practice` package. This time, we use a `BorderLayout` to arrange the buttons. You need to:

1. Add **abstract** methods `shift()` and `reset()` in `Shape`.
2. Override the `shift()` and `reset()` method in `Circle`, `Rectangle`, and `Triangle`, because each shape has different behaviors.
3. Fill in the `TODO` in the `DrawingPanel` class to implement the behaviors of the shift and reset buttons.
4. You may add any additional methods or fields if needed.