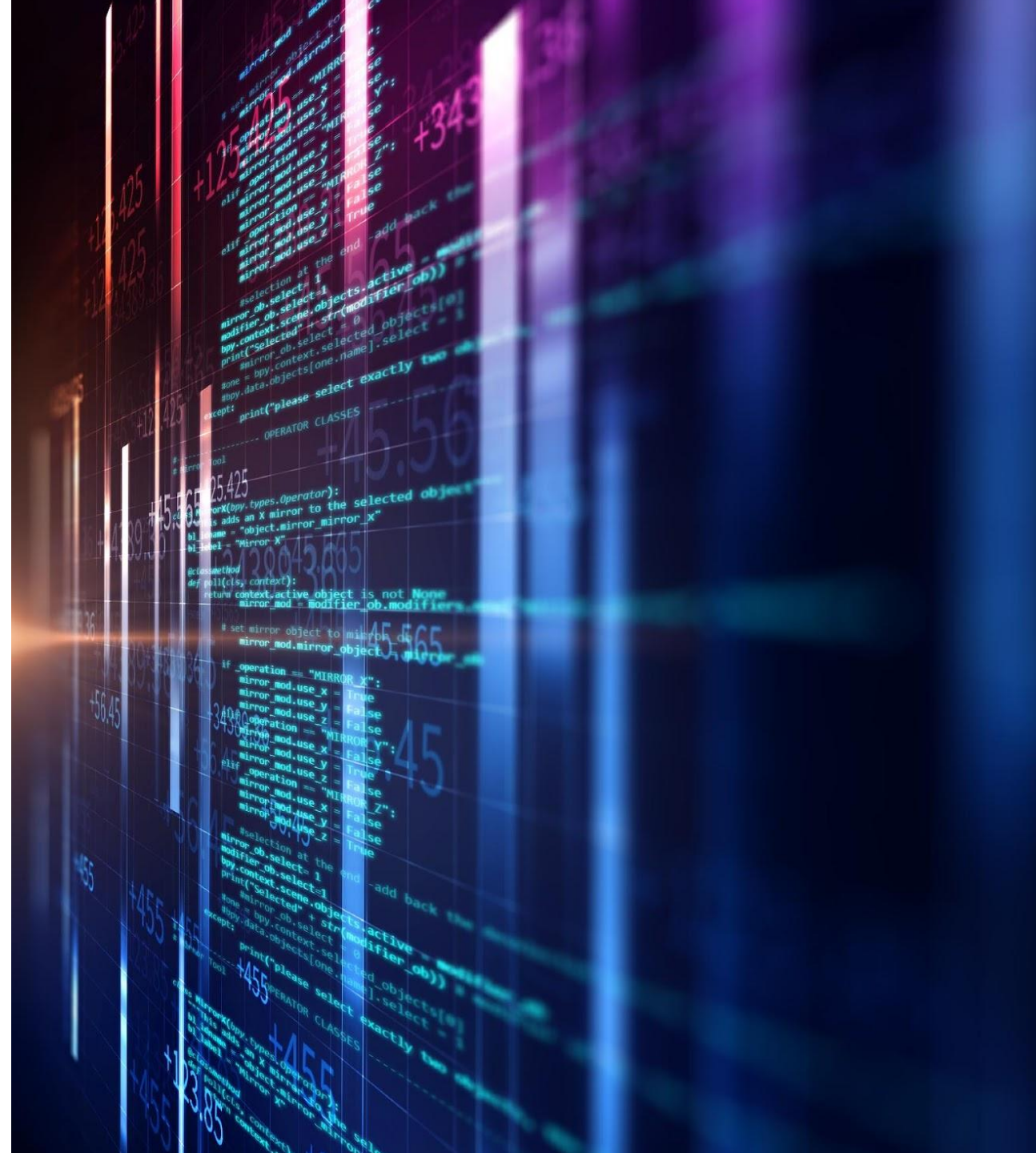


Interfaces

Brandon Krakowsky



Interfaces



Interfaces

- An **interface** declares (describes) methods but does not supply bodies for them
 - An interface is a *pure* abstract class -- *none* of its methods are defined
- To create an **interface**, use the *interface* keyword
- Here's the *KeyListener* interface, for receiving keyboard events (keystrokes)

```
public interface KeyListener {  
    void keyPressed(KeyEvent e); //declares method keyPressed  
    void keyReleased(KeyEvent e); //declares method keyReleased  
    void keyTyped(KeyEvent e); //declares method keyTyped  
}
```

- All the methods are implicitly **public** and **abstract**
 - You can add these qualifiers if you like, but why bother?
- Like an abstract class, you CANNOT instantiate an interface
`KeyListener keyListener = new KeyListener();` //You CANNOT do this



Common Interfaces

- You will sometimes use the supplied Java interfaces
- For example

KeyListener: Interface for receiving keyboard events (keystrokes). Classes interested in processing keyboard events will implement this interface (and all the methods it contains).

Iterable: Implementing this interface allows an object to be the target of a *for loop* statement. (Like an ArrayList)

Comparable: For objects interested in being compared and conforming to order.

Etc. ...



Designing Interfaces

- Sometimes you'll want to design your own interface
- You would write an interface if you want classes of various types to all have a certain set of capabilities
- For example, if you want to be able to create different kinds of vehicles, you might define an interface `Driveable`:

```
public interface Driveable {  
    void turnLeft(double angle);  
    void turnRight(double angle);  
    void reverse();  
    void accelerate(int increment);  
    void decelerate(int decrement);  
}
```

- Now you can create any kind of vehicle with these capabilities by implementing `Driveable`



Implementing an Interface I

- You **extend** a class, but you **implement** an interface
- A class can only extend (subclass) one other class, but it can implement as many interfaces as you like

```
public class MyListener implements KeyListener, ActionListener {  
    ...  
}
```

- A class can **extend** another class **AND** **implement** interfaces

```
public class MyClass extends AnotherClass implements KeyListener,  
ActionListener {  
    ...  
}
```



Implementing an Interface II

- When you say a class **implements** an interface, you are promising to define all the methods that are declared in the interface

```
public class MyKeyListener implements KeyListener {  
    public void keyPressed(KeyEvent e) {...}  
    public void keyReleased(KeyEvent e) {...}  
    public void keyTyped(KeyEvent e) {...}  
}
```

- The “...” indicates actual code (the method bodies) that you must supply
- Now you can create a new **MyKeyListener**
`MyKeyListener myKeyListener = new MyKeyListener();`



Partially Implementing an Interface

- What if you only care about *some* of the methods declared in an interface?
- It's possible for a class to define some, but not all of the methods
- To do this, a class *must* be defined as **abstract**

```
public abstract class MyKeyListener implements KeyListener {  
    public void keyTyped(KeyEvent e) {...};  
}
```

- MyKeyListener is *only* implementing keyTyped
- Thus, MyKeyListener must be defined as **abstract**
- You can even extend an interface (to add additional methods)

```
public interface MyFunkyKeyListener extends KeyListener { ... }
```



What Are Interfaces For?

Reason 1: A class can only extend one other class, but it can implement multiple interfaces

- This allows classes to fill multiple “roles”
- For example, in writing user interfaces, it’s common to have a class be able to handle different kinds of user interactions
 - Button clicks
 - Keyboard events
 - Mouse wheel rotations
 - Etc ...

```
public class MyUserInterface extends Applet implements  
ActionListener, KeyListener {  
    ...  
}
```



What Are Interfaces For?

Reason 2: You can write methods that work for more than one kind of class

- Here's an interface for different kinds of board games

```
public interface RuleSet {  
    boolean isLegal(Move m, Board b);  
    void makeMove(Move m);  
}
```

- Every class that implements RuleSet must have these methods

```
public class CheckersRules implements RuleSet {  
    public boolean isLegal(Move m, Board b) { ... } //implemented  
method  
    public void makeMove(Move m) { ... } //implemented method  
}
```



How to Use an Interface

- Here's another implementation of RuleSet

```
public class ChessRules implements RuleSet {  
    public boolean isLegal(Move m, Board b) { ... } //implemented method  
    public void makeMove(Move m) { ... } //implemented method  
}
```

- And here, an abstract class, is implementing RuleSet

```
public abstract class LinesOfActionRules implements RuleSet {  
    public void makeMove(Move m) { ... } //implemented method  
    public void makeHorizontalMoveOneSpace() { ... } //another concrete method  
}
```

- LinesOfActionRules is implementing one method, *makeMove*, from RuleSet
- It also has another concrete method *makeHorizontalMoveOneSpace*
- Thus, LinesOfActionRules must be defined as abstract



How to Use an Interface

```
RuleSet rulesOfThisGame = new ChessRules();
```

- This assignment IS LEGAL because a rulesOfThisGame object is a RuleSet object

```
if (rulesOfThisGame.isLegal(m, b)) {  
    rulesOfThisGame.makeMove(m);  
}
```

- This statement IS LEGAL because, whatever kind of RuleSet object rulesOfThisGame is, it must have isLegal and makeMove methods

```
RuleSet rulesOfThisGameTwo = new LinesOfActionRules();
```

- This statement IS ILLEGAL because LinesOfActionRules is an abstract class, by default, because it didn't implement all methods of RuleSet



Shape Interface Project



Shape Interface

```
Shape.java
1 /**
2  * Interface for shapes.
3  * @author brandonkrakowsky
4  *
5  */
6 public interface Shape {
7
8     /**
9      * Returns area of shape.
10     * @return area
11     */
12     public double area();
13
14     /**
15      * Returns perimeter of shape.
16      * @return perimeter
17      */
18     public double perimeter();
19
20     /**
21      * Draws shape.
22      */
23     public void draw();
24 }
```

Circle Class

```
Shape.java  Circle.java ✕
1  /**
2   * Represents a circle.
3   * Implements Shape.
4   * @author brandonkrakowsky
5   *
6   */
7  public class Circle implements Shape {
8
9      private static final double PI = Math.PI;
10
11     /**
12      * Radius of circle.
13      */
14     private double radius;
15
16     /**
17      * Creates a circle with given radius.
18      * @param radius of circle
19      */
20     public Circle(double radius){
21         this.radius = radius;
22     }
23 }
```

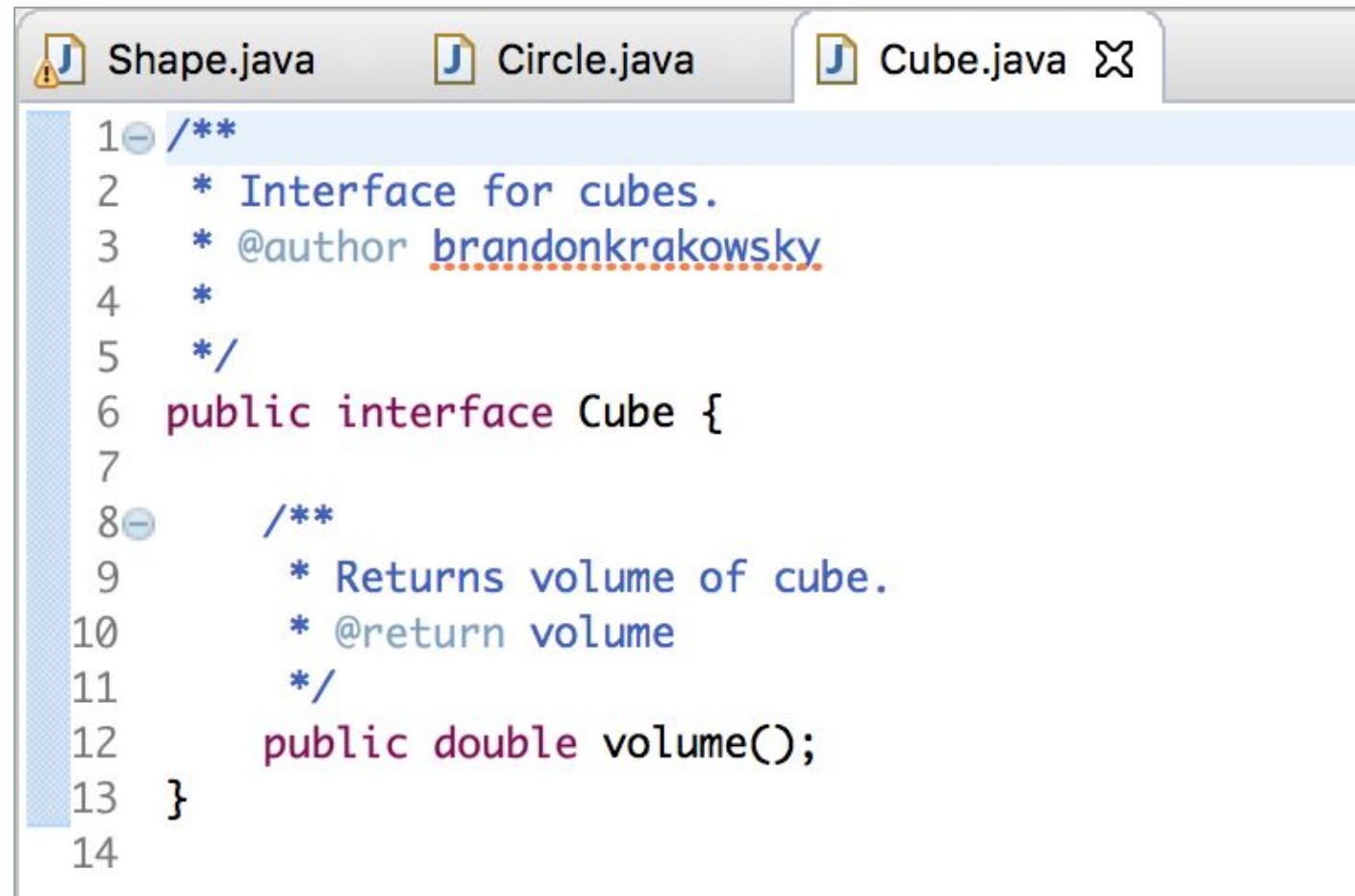


Circle Class

```
23
24- /**
25     * Returns area of circle.
26     */
27- @Override
28- public double area() {
29     return Circle.PI * this.radius * this.radius;
30 }
31
32- /**
33     * Returns perimeter of circle.
34     */
35- @Override
36- public double perimeter() {
37     return 2 * Circle.PI * this.radius;
38 }
39
40- /**
41     * Draws this circle.
42     */
43- @Override
44- public void draw() {
45     System.out.println("Drawing a circle with radius " + this.radius);
46 }
47
```

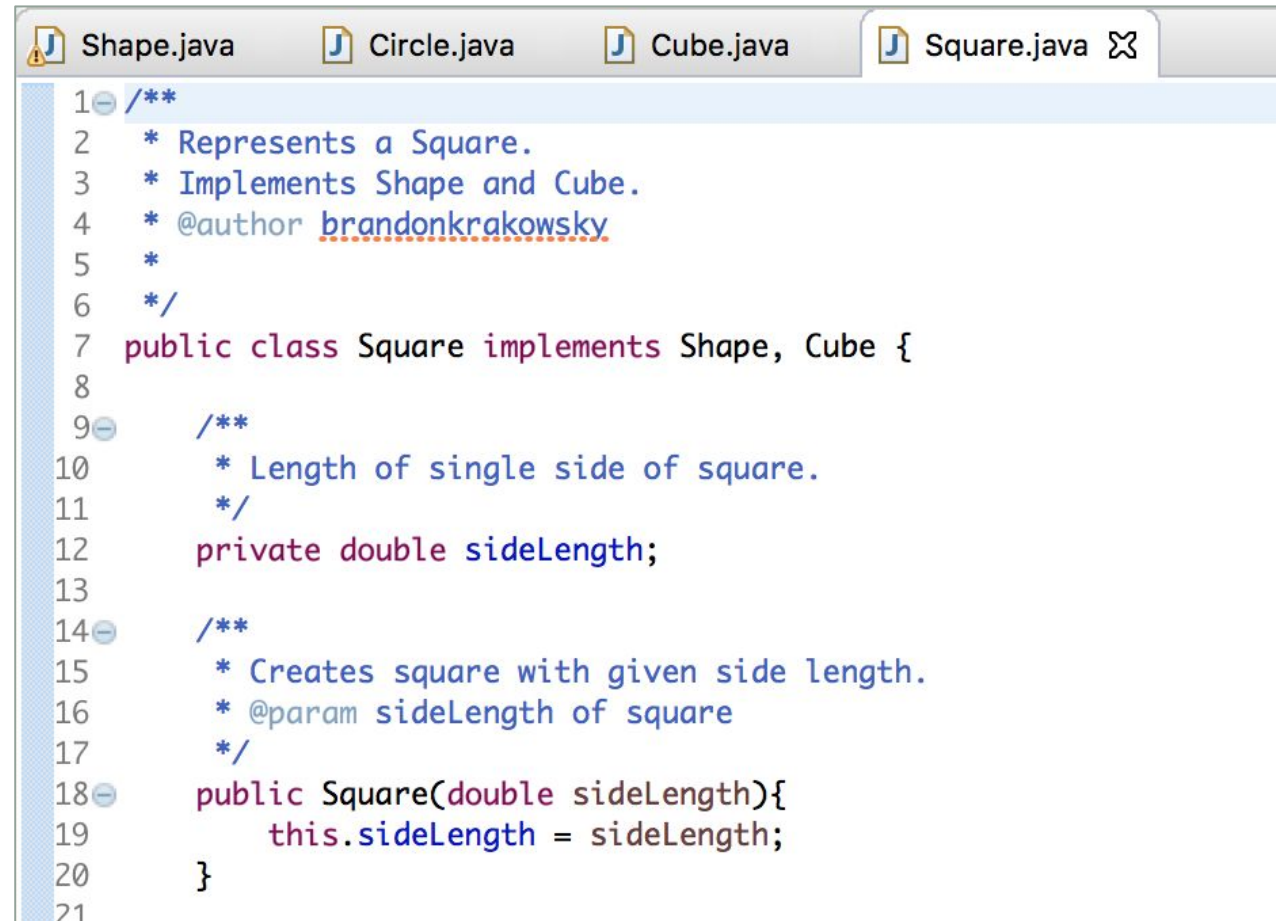


Cube Interface



```
1- /**
2  * Interface for cubes.
3  * @author brandonkrakowsky
4  *
5  */
6  public interface Cube {
7
8-     /**
9      * Returns volume of cube.
10     * @return volume
11     */
12     public double volume();
13 }
14
```

Square Class



```
Shape.java Circle.java Cube.java Square.java ✕
1 /**
2  * Represents a Square.
3  * Implements Shape and Cube.
4  * @author brandonkrakowsky
5  *
6  */
7 public class Square implements Shape, Cube {
8
9     /**
10     * Length of single side of square.
11     */
12     private double sideLength;
13
14     /**
15     * Creates square with given side length.
16     * @param sideLength of square
17     */
18     public Square(double sideLength){
19         this.sideLength = sideLength;
20     }
21 }
```

Square Class

```
21
22- /**
23   * Returns area of square.
24   */
25- @Override
26- public double area(){
27     return this.sideLength * this.sideLength;
28 }
29
30- /**
31   * Returns perimeter of square.
32   */
33- @Override
34- public double perimeter(){
35     return 4 * this.sideLength;
36 }
37
38- /**
39   * Draws this square.
40   */
41- @Override
42- public void draw(){
43     System.out.println("Drawing a square with side length " + this.sideLength);
44 }
45
```



Square Class

```
45
46- /**
47    * Returns volume of cube made up of squares.
48    */
49- @Override
50 public double volume() {
51     return Math.pow(this.sideLength, 3);
52 }
```



Triangle Abstract Class

```
Shape.java  Circle.java  Cube.java  Square.java  Triangle.java ✕

1  /**
2   * Abstract class representing a triangle.
3   * Implements Shape.
4   * @author brandonkrakowsky
5   *
6   */
7  public abstract class Triangle implements Shape {
8
9      /**
10     * Sides of triangle.
11     */
12     protected double sideA, sideB, sideC;
13
14     /**
15     * Creates triangle with given sides.
16     * @param sideA for triangle
17     * @param sideB for triangle
18     * @param sideC for triangle
19     */
20     public Triangle(double sideA, double sideB, double sideC){
21         this.sideA = sideA;
22         this.sideB = sideB;
23         this.sideC = sideC;
24     }
25 }
```

Triangle Abstract Class

```
26-    /**
27     * Returns the perimeter of triangle.
28     */
29-    @Override
30-    public double perimeter() {
31        return this.sideA + this.sideB + this.sideC;
32    }
33
34-    /**
35     * Draws this triangle.
36     */
37-    @Override
38-    public void draw() {
39        System.out.println("Drawing a triangle with length of sides "
40                           + this.sideA + ", "
41                           + this.sideB + ", "
42                           + this.sideC);
43    }
44
```


Right Triangle Class

```
Circle.java Cube.java Square.java Triangle.java RightTriangle.j ✕
1 /**
2  * Represents a right triangle.
3  * Extends Triangle.
4  * @author brandonkrakowsky
5  *
6  */
7 public class RightTriangle extends Triangle {
8
9     /**
10     * Creates Right Triangle with given sides.
11     * @param sideA for triangle
12     * @param sideB for triangle
13     */
14     public RightTriangle(double sideA, double sideB){
15         super(sideA, sideB, RightTriangle.getHypotenuse(sideA, sideB));
16     }
17 }
```

Right Triangle Class

```
18- /**
19     * Returns area of right triangle.
20     */
21- @Override
22- public double area() {
23     return (this.sideA * this.sideB) / 2;
24 }
25
26 //static method: does not require instance of RightTriangle to call
27 //call using class name: RightTriangle.getHypotenuse(a, b);
28- /**
29     * Returns the hypotenuse of right triangle based on given side a and b.
30     * @param side a of triangle
31     * @param side b of triangle
32     * @return hypotenuse
33     */
34- private static double getHypotenuse(double a, double b){
35     return Math.sqrt((a * a) + (b * b));
36 }
37
```

Equilateral Triangle Class

```
Circle.java Cube.java Square.java Triangle.java RightTriangle.j EquilateralTria X
1 /**
2  * Represents an equilateral triangle.
3  * Extends Triangle.
4  * @author brandonkrakowsky
5  *
6  */
7 public class EquilateralTriangle extends Triangle {
8
9     /**
10     * Creates equilateral triangle with given side length.
11     * @param sideLength of equilateral triangle
12     */
13     public EquilateralTriangle(double sideLength){
14         super(sideLength, sideLength, sideLength);
15     }
16 }
```

Equilateral Triangle Class

```
17- /**
18     * Returns area of equilateral triangle.
19     */
20- @Override
21- public double area() {
22     return (Math.sqrt(3) / 4) * (this.sideA * this.sideA);
23 }
24
```


Shape Class

```
Shape.java ✕
25
26 public static void main(String[] args) {
27
28     System.out.println("Circle");
29
30     //create circle with radius
31     Circle circle = new Circle(4);
32
33     //call implemented methods in interface shape
34     System.out.println(circle.area());
35     System.out.println(circle.perimeter());
36     circle.draw();
37
```

Shape Class

```
38
39 System.out.println("");
40 System.out.println("Square");
41
42 //create square with size of side
43 Square square = new Square(4);
44
45 //call implemented methods in interface shape
46 System.out.println(square.area());
47 System.out.println(square.perimeter());
48 square.draw();
49
50 //call implemented methods in interface cube
51 System.out.println(square.volume());
52
```

Shape Class

```
53
54 System.out.println("");
55 System.out.println("Triangles");
56
57 //create right triangle with size of two sides
58 RightTriangle rightTriangle = new RightTriangle(3, 5);
59
60 //call implemented methods in right triangle
61 System.out.println(rightTriangle.area());
62
63 //call implemented methods in triangle
64 System.out.println(rightTriangle.perimeter());
65 rightTriangle.draw();
66
```


Shape Class

```
66
67     System.out.println("");
68
69     //create equilateral triangle with size of one side
70     EquilateralTriangle equalateralTriangle = new EquilateralTriangle(3);
71
72     //call implemented method in equilateral triangle
73     System.out.println(equalateralTriangle.area());
74
75 }
```



Shape Class

```
76
77     System.out.println("");
78     System.out.println("Iterate over arraylist of shapes and call common methods implemented in each.");
79
80     //arraylist of shapes
81     ArrayList<Shape> shapes = new ArrayList<Shape>();
82     shapes.add(circle);
83     shapes.add(square);
84     shapes.add(rightTriangle);
85     shapes.add(equilateralTriangle);
86
87     //iterate over arraylist of shapes and call common methods implemented in each
88     for (Shape s : shapes) {
89         System.out.println(s.area());
90         System.out.println(s.perimeter());
91         s.draw();
92         System.out.println("");
93     }
94 }
```