

Abstract Classes and Interfaces



EECS2030: Advanced
Object Oriented Programming
Fall 2017

CHEN-WEI WANG

Abstract Class (1)

Problem: A polygon may be either a triangle or a rectangle. Given a polygon, we may either

- Grow its shape by incrementing the size of each of its sides;
 - Compute and return its perimeter; or
 - Compute and return its area.
- For a rectangle with *length* and *width*, its area is $length \times width$.
 - For a triangle with sides a , b , and c , its area, according to Heron's formula, is

$$\sqrt{s(s-a)(s-b)(s-c)}$$

where

$$s = \frac{a + b + c}{2}$$

- How would you solve this problem in Java, while minimizing code duplicates?

Abstract Class (2)

```
public abstract class Polygon {
    double[] sides;
    Polygon(double[] sides) { this.sides = sides; }
    void grow() {
        for(int i = 0; i < sides.length; i++) {
            sides[i]++;
        }
    }
    double getPerimeter() {
        double perimeter = 0;
        for(int i = 0; i < sides.length; i++) { perimeter += sides[i]; }
        return perimeter;
    }
    abstract double getArea();
}
```

- Method `getArea` not implemented and shown **signature** only.
- \therefore Polygon cannot be used as a **dynamic type**
- Writing `new Polygon(...)` is forbidden!

Abstract Class (3)

```
public class Rectangle extends Polygon {  
    Rectangle(double length, double width) {  
        super(new double[4]);  
        sides[0] = length; sides[1] = width;  
        sides[2] = length; sides[3] = width;  
    }  
    double getArea() { return sides[0] * sides[1]; }  
}
```

- Method `getPerimeter` is inherited from the super-class `Polygon`.
- Method `getArea` is implemented in the sub-class `Rectangle`.
- \therefore `Rectangle` can be used as a **dynamic type**
- Writing `Polygon p = new Rectangle(3, 4)` allowed!

Abstract Class (4)

```
public class Triangle extends Polygon {
    Triangle(double side1, double side2, double side3) {
        super(new double[3]);
        sides[0] = side1; sides[1] = side2; sides[2] = side3;
    }
    double getArea() {
        /* Heron's formula */
        double s = getPerimeter() * 0.5;
        double area = Math.sqrt(
            s * (s - sides[0]) * (s - sides[1]) * (s - sides[2]));
        return area;
    }
}
```

- Method `getPerimeter` is inherited from `Polygon`.
- Method `getArea` is implemented in the sub-class `Triangle`.
- ∴ `Triangle` can be used as a **dynamic type**
- Writing `Polygon p = new Triangle(3, 4, 5)` allowed!

Abstract Class (5)

```
1 public class PolygonCollector {
2     Polygon[] polygons;
3     int numberOfPolygons;
4     PolygonCollector() { polygons = new Polygon[10]; }
5     void addPolygon(Polygon p) {
6         polygons[numberOfPolygons] = p; numberOfPolygons ++;
7     }
8     void growAll() {
9         for(int i = 0; i < numberOfPolygons; i ++ ) {
10            polygons[i].grow();
11        }
12    }
13 }
```

- **Polymorphism:** Line 5 may accept as argument any object whose **static type** is Polygon or any of its sub-classes.
- **Dynamic Binding:** Line 10 calls the version of `grow` inherited to the **dynamic type** of `polygons[i]`.

Abstract Class (6)

```
1 public class PolygonConstructor {
2     Polygon getPolygon(double[] sides) {
3         Polygon p = null;
4         if(sides.length == 3) {
5             p = new Triangle(sides[0], sides[1], sides[2]);
6         }
7         else if(sides.length == 4) {
8             p = new Rectangle(sides[0], sides[1]);
9         }
10        return p;
11    }
12    void grow(Polygon p) { p.grow(); }
13 }
```

- **Polymorphism:**

- **Line 2** may accept as return value any object whose **static type** is Polygon or any of its sub-classes.
- **Line 5** returns an object whose **dynamic type** is Triangle; **Line 8** returns an object whose **dynamic type** is Rectangle.

Abstract Class (7.1)

```
1 public class PolygonTester {
2     public static void main(String[] args) {
3         Polygon p;
4         p = new Rectangle(3, 4); /* polymorphism */
5         System.out.println(p.getPerimeter()); /* 14.0 */
6         System.out.println(p.getArea()); /* 12.0 */
7         p = new Triangle(3, 4, 5); /* polymorphism */
8         System.out.println(p.getPerimeter()); /* 12.0 */
9         System.out.println(p.getArea()); /* 6.0 */
10
11        PolygonCollector col = new PolygonCollector();
12        col.addPolygon(new Rectangle(3, 4)); /* polymorphism */
13        col.addPolygon(new Triangle(3, 4, 5)); /* polymorphism */
14        System.out.println(col.polygons[0].getPerimeter()); /* 14.0 */
15        System.out.println(col.polygons[1].getPerimeter()); /* 12.0 */
16        col.growAll();
17        System.out.println(col.polygons[0].getPerimeter()); /* 18.0 */
18        System.out.println(col.polygons[1].getPerimeter()); /* 15.0 */
```


Abstract Class (7.2)

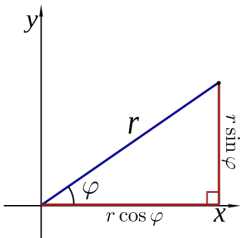
```
1 PolygonConstructor con = new PolygonConstructor();
2 double[] recSides = {3, 4, 3, 4}; p = con.getPolygon(recSides);
3 System.out.println(p instanceof Polygon); ✓
4 System.out.println(p instanceof Rectangle); ✓
5 System.out.println(p instanceof Triangle); ✗
6 System.out.println(p.getPerimeter()); /* 14.0 */
7 System.out.println(p.getArea()); /* 12.0 */
8 con.grow(p);
9 System.out.println(p.getPerimeter()); /* 18.0 */
10 System.out.println(p.getArea()); /* 20.0 */
11 double[] triSides = {3, 4, 5}; p = con.getPolygon(triSides);
12 System.out.println(p instanceof Polygon); ✓
13 System.out.println(p instanceof Rectangle); ✗
14 System.out.println(p instanceof Triangle); ✓
15 System.out.println(p.getPerimeter()); /* 12.0 */
16 System.out.println(p.getArea()); /* 6.0 */
17 con.grow(p);
18 System.out.println(p.getPerimeter()); /* 15.0 */
19 System.out.println(p.getArea()); /* 9.921 */
20 } }
```

Abstract Class (8)

- An **abstract class**:
 - Typically has **at least one** method with no implementation body
 - May define common implementations inherited to **sub-classes**.
- Recommended to use an **abstract class** as the **static type** of:
 - A **variable**
e.g., `Polygon p`
 - A **method parameter**
e.g., `void grow(Polygon p)`
 - A **method return value**
e.g., `Polygon getPolygon(double[] sides)`
- It is forbidden to use an **abstract class** as a **dynamic type**
e.g., `Polygon p = new Polygon(...)` is not allowed!
- Instead, create objects whose **dynamic types** are descendant classes of the **abstract class** ⇒ Exploit **dynamic binding** !
e.g., `Polygon p = con.getPolygon(recSides)`
This is as if we did `Polygon p = new Rectangle(...)`

Interface (1.1)

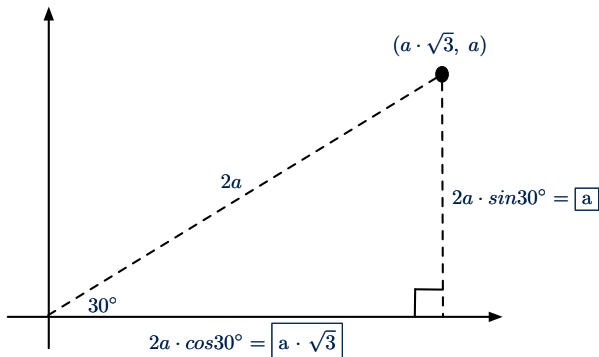
- We may implement `Point` using two representation systems:



- The *Cartesian system* stores the *absolute* positions of x and y .
 - The *Polar system* stores the *relative* position: the angle (in radian) ϕ and distance r from the origin $(0,0)$.
- As far as users of a `Point` object p is concerned, being able to call `p.getX()` and `getY()` is what matters.
- How `p.getX()` and `p.getY()` are internally computed, depending on the **dynamic type** of p , do not matter to users.

Interface (1.2)

Recall: $\sin 30^\circ = \frac{1}{2}$ and $\cos 30^\circ = \frac{1}{2} \cdot \sqrt{3}$



We consider the same point represented differently as:

- $r = 2a, \psi = 30^\circ$ [polar system]
- $x = 2a \cdot \cos 30^\circ = a \cdot \sqrt{3}, y = 2a \cdot \sin 30^\circ = a$ [cartesian system]

Interface (2)

```
interface Point {  
    double getX();  
    double getY();  
}
```

- An interface `Point` defines how users may access a point: either get its `x` coordinate or its `y` coordinate.
- Methods `getX` and `getY` similar to `getArea` in `Polygon`, have no implementations, but *signatures* only.
- \therefore `Point` cannot be used as a *dynamic type*
- Writing `new Point(...)` is forbidden!

Interface (3)

```
public class CartesianPoint implements Point {  
    double x;  
    double y;  
    CartesianPoint(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
}
```

- CartesianPoint is a possible implementation of Point.
- Attributes `x` and `y` declared according to the *Cartesian system*
- All method from the interface `Point` are implemented in the sub-class `CartesianPoint`.
- ∴ CartesianPoint can be used as a **dynamic type**
- `Point p = new CartesianPoint(3, 4)` allowed!

Interface (4)

```
public class PolarPoint implements Point {  
    double phi;  
    double r;  
    public PolarPoint(double r, double phi) {  
        this.r = r;  
        this.phi = phi;  
    }  
    public double getX() { return Math.cos(phi) * r; }  
    public double getY() { return Math.sin(phi) * r; }  
}
```

- PolarPoint is a possible implementation of Point.
- Attributes phi and r declared according to the *Polar system*
- All method from the interface Point are implemented in the sub-class PolarPoint.
- ∴ PolarPoint can be used as a *dynamic type*
- Point p = new PolarPoint(3, $\frac{\pi}{6}$) allowed! [360° = 2π]

Interface (5)

```
1 public class PointTester {
2     public static void main(String[] args) {
3         double A = 5;
4         double X = A * Math.sqrt(3);
5         double Y = A;
6         Point p;
7         p = new CartesianPoint(X, Y); /* polymorphism */
8         print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
9         p = new PolarPoint(2 * A, Math.toRadians(30)); /* polymorphism */
10        print("(" + p.getX() + ", " + p.getY() + ")"); /* dyn. bin. */
11    }
12 }
```

- Lines 7 and 9 illustrate *polymorphism*, how?
- Lines 8 and 10 illustrate *dynamic binding*, how?

Interface (6)

- An **interface**:
 - Has **all** its methods with no implementation bodies.
 - Leaves complete freedom to its **implementors**.
- Recommended to use an **interface** as the **static type** of:
 - A **variable**
e.g., `Point p`
 - A **method parameter**
e.g., `void moveUp(Point p)`
 - A **method return value**
e.g., `Point getPoint(double v1, double v2, boolean isCartesian)`
- It is forbidden to use an **interface** as a **dynamic type**
e.g., `Point p = new Point(...)` is not allowed!
- Instead, create objects whose **dynamic types** are descendant classes of the **interface** ⇒ Exploit **dynamic binding** !

Abstract Classes vs. Interfaces: When to Use Which?

- Use **interfaces** when:
 - There is a *common set of functionalities* that can be implemented via *a variety of strategies*.
e.g., Interface `Point` declares signatures of `getX()` and `getY()`.
 - Each descendant class represents a different implementation strategy for the same set of functionalities.
 - `CartesianPoint` and `PolarPoint` represent different strategies for supporting `getX()` and `getY()`.
- Use **abstract classes** when:
 - *Some (not all) implementations can be shared* by descendants, and *some (not all) implementations cannot be shared*.
e.g., Abstract class `Polygon`:
 - Defines implementation of `getPerimeter`, to be shared by `Rectangle` and `Triangle`.
 - Declares signature of `getArea`, to be implemented by `Rectangle` and `Triangle`.

Index (1)

Abstract Class (1)

Abstract Class (2)

Abstract Class (3)

Abstract Class (4)

Abstract Class (5)

Abstract Class (6)

Abstract Class (7.1)

Abstract Class (7.2)

Abstract Class (8)

Interface (1.1)

Interface (1.2)

Interface (2)

Interface (3)

Interface (4)

Index (2)

Interface (5)

Interface (6)

Abstract Classes vs. Interfaces: When to Use Which?