System Design

This week

- Complex client/supplier relationships
- Interfaces
- Superclasses

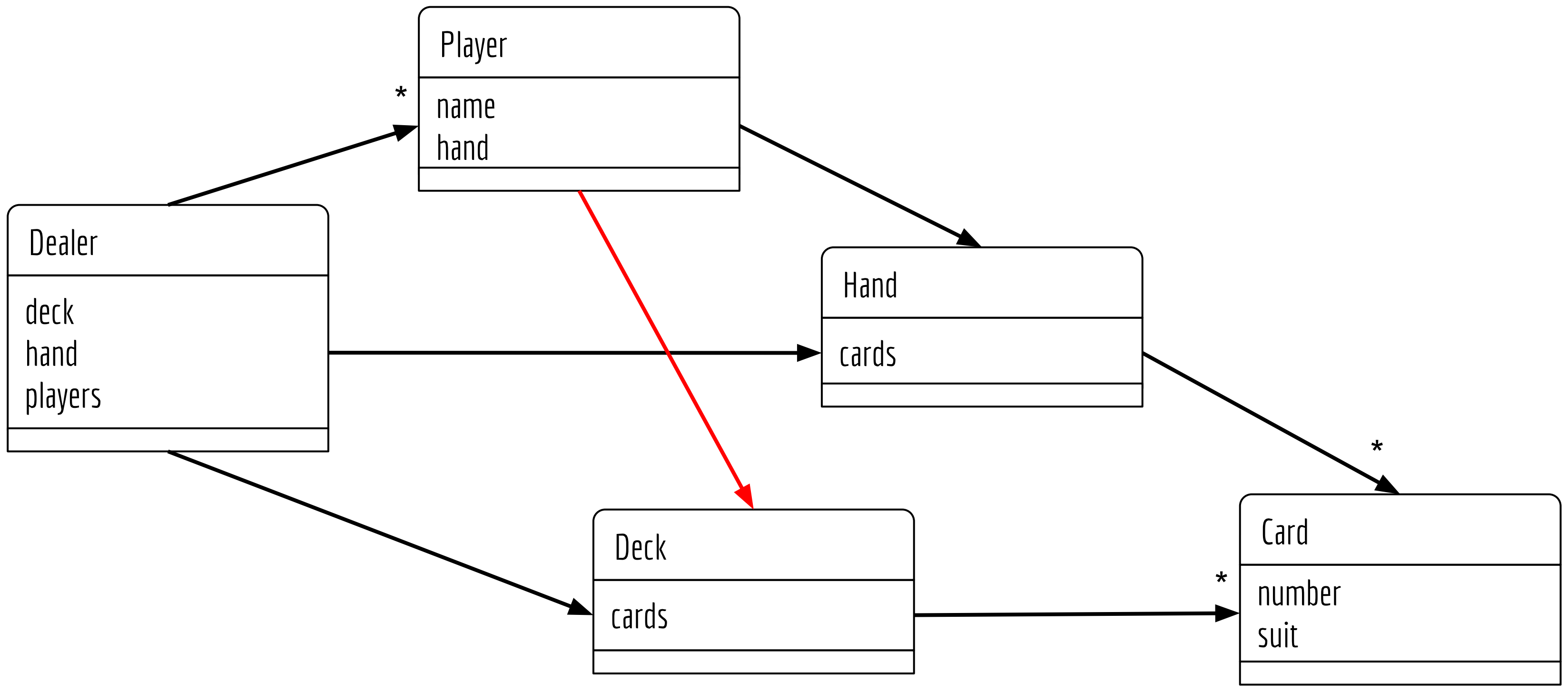
Complex client/supplier relationships

Specification

Blackjack is a game with one dealer and many players. Players don’t play each other, they play against the dealer. You draw cards one at a time aiming to accumulate a higher hand value than the dealer without going over 21 (busting). A blackjack is a special hand with an ace and a 10-valued card (10, Jack, Queen, King). A blackjack beats any hand except for another blackjack. The game proceeds as follows:

1. The deck is shuffled.
2. Each player and the dealer are dealt 2 cards.
3. Anyone with a blackjack stands (accepts no further cards).
4. Each player and the dealer has a turn.
    a. A player’s turn: draw cards until they bust or choose to stand.
    b. A dealer’s turn: draw cards until they bust or the value is greater than 16.
5. The winners are decided.

Class diagram



Relationships

- The dealer “has a” deck. The dealer “deals from” the deck.
- The dealer and the players “draw cards from” the deck.
- The dealer and the players “have a” hand.
- The dealer “manages” many players.
- The deck and the hands “have” many cards.

Problem: A player wants to draw cards from the deck but doesn’t have a deck.
Solution: Pass the deck as a parameter.

```
public class Player {
    public void drawCard(Deck deck)
```

Location table

Classes	Dealer	Deck	Player	Hand	Card
Fields	deck hand players	cards	name hand	cards	number suit
Goals					
shuffle	*	*			
deal	*	*	*	*	
haveTurn	*	*	*	*	*
decide	*		*	*	*

Sample I/O

```
Jack has JC 3S: 13
Choice (d/s): d
Jack has JC 3S 4S: 17
Choice (d/s): d
Jack busts with JC 3S 4S QS: 27!
Jill has 2D 7C: 9
Choice (d/s): d
Jill has 2D 7C 10C: 19
Choice (d/s): s
Dealer has 8H QC: 18
Jack loses with JC 3S 4S QS: 27
Jill wins with 2D 7C 10C: 19
```

DEMO

This demo is of similar complexity to Assignment 1.

Interfaces

Interfaces

```
public interface Polygon {
    double area();
    int numberOfSides();
}
```

- An interface declares a set of methods common to multiple classes.
  E.g. All polygons have area() and numberOfSides() methods.

- Each class provides its own “implementation” of these methods.

Implementing an interface

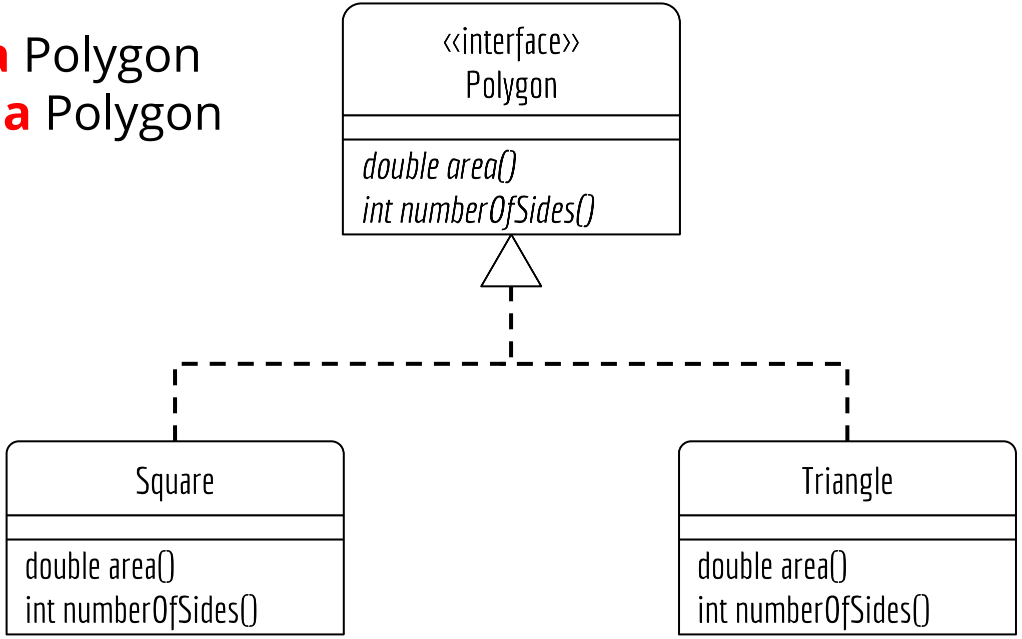
- Implement an interface with the **implements** keyword.
- Override an interface method with the **@Overrides** annotation.
- Methods from an interface must be **public**.

```
public class Square implements Polygon {
     private double size;
     public Square(double size)
     {    this.size = size;    }
     @Override public double area() {
          return size * size;
     }
     @Override public int numberOfSides() {
          return 4;
     }
}
```

```
public class Triangle implements Polygon {
     private double base, height;
     public Square(double base, double height) {
          this.base = base; this.height = height;
     }
     @Override public double area()
     }    return base * height / 2.0;    }
     @Override public int numberOfSides()
     {    return 3;    }
}
```

The “is a” relationship

- A Square **is a** Polygon
- A Triangle **is a** Polygon



The Payoff: Polymorphism

- Polymorphism allows for a single object to have many types.

  `new Square(10)`

- This object has type `Square` and type `Polygon`.

  i.e. It can be used as a Square or a Polygon.

Polymorphism #1

```
public void showArea(Polygon p) {
    System.out.println("Polygon has area " + p.area());
}

showArea(new Square(10));
showArea(new Triangle(8, 4));
```

- The showArea method accepts any `Polygon`.
  i.e. Any object that has `area()` and `numberOfSides()` methods.
  - A `Square` is a `Polygon`. It is accepted.
  - A `Triangle` is a `Polygon`. It is accepted.

Polymorphism #2

```
LinkedList<Polygon> polygons = new LinkedList<Polygon>();
polygons.add(new Square(10));
polygons.add(new Square(7));
polygons.add(new Triangle(3));
```

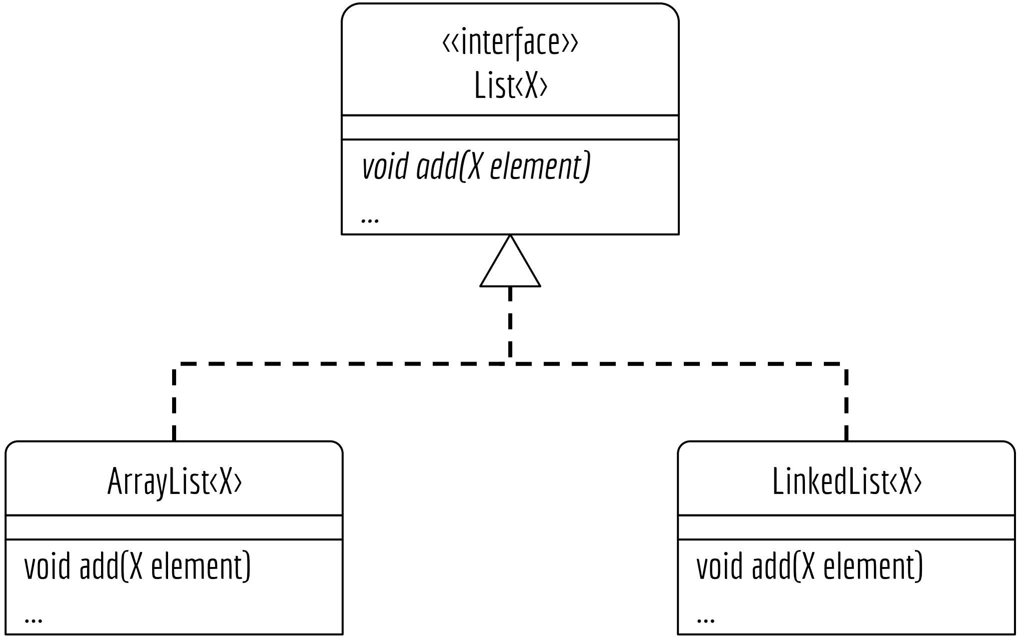
- This list accepts any `Polygon`.

```
for (Polygon p : polygons)
    System.out.println("Polygon has area " + p.area());
```

- Each polygon is known to have an `area()` method.

The List interface

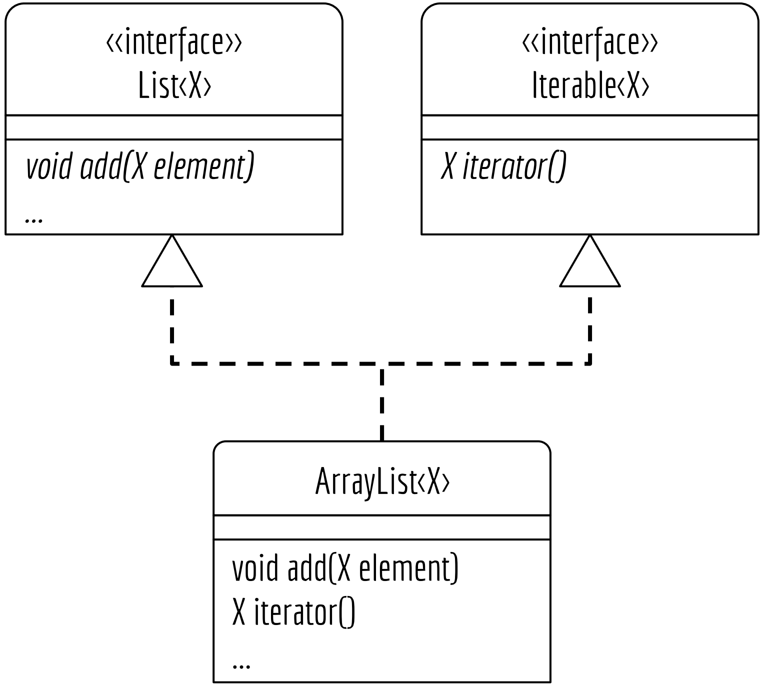
Two implementations of the List "interface".



Implementing multiple interfaces

A class can implement multiple interfaces.

```
public class ArrayList<X>
    implements List<X>, Iterable<X>
```



DEMO

Superclasses

Superclasses

Like interfaces:

- Define methods common to multiple classes.

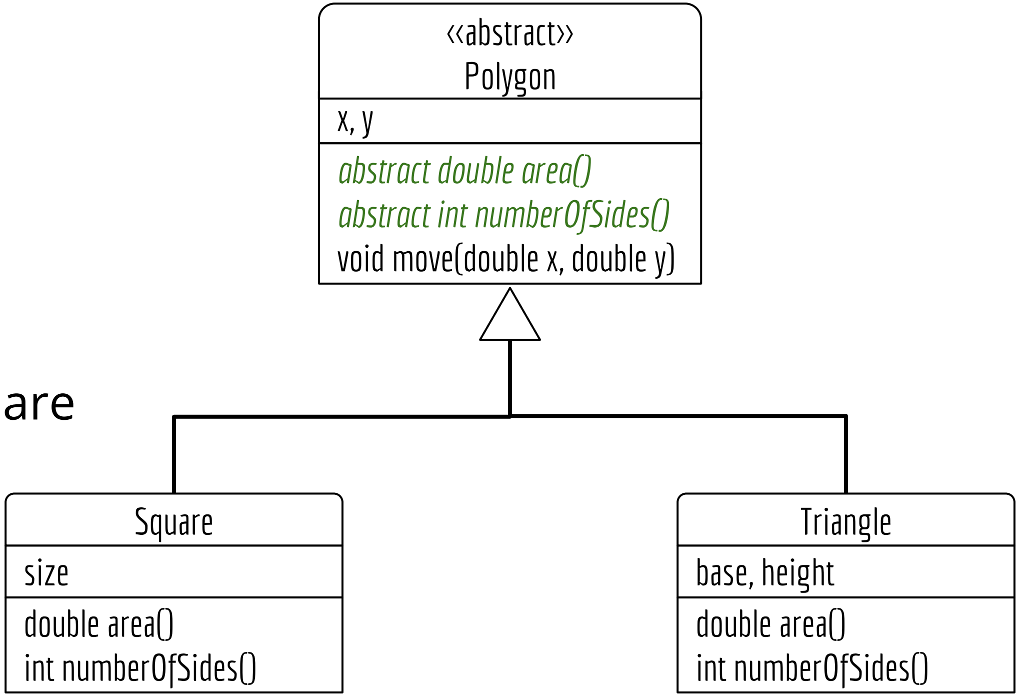
Unlike interfaces:

- Provide implementations for those common methods[1].
- Define common fields.
- Define non-public members.

[1] Since Java 8, interfaces now support this too.

Superclass / Subclass

- A superclass defines common methods and fields.
- Each subclass inherits those common methods and fields.
- Methods which must be implemented in the subclasses are declared “abstract”.
- A class containing abstract methods must also be declared abstract.



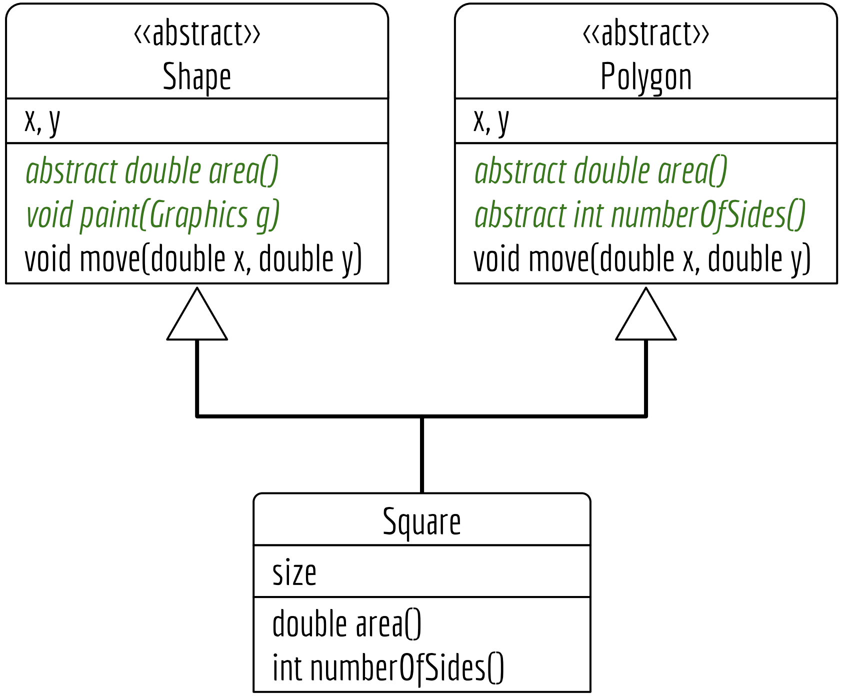
Multiple inheritance not supported

The problem:

- Two superclasses define two different implementations of `move()`.
- Which one gets inherited into `Square`?

Java's solution:

- A subclass cannot extend more than one superclass.



Superclass example

```
public abstract class Polygon {
    protected double x
    protected double y;
    public abstract double area();
    public abstract int numberOfSides();
    public void move(double dx, double dy) {
        x += dx;
        y += dy;
    }
}
```

- “Subclasses” implement the abstract methods, inherit everything else.
- Fields declared `protected` can be accessed by subclasses.

Subclass example

```
public class Square extends Polygon {
    private double size;
    public Square(double size) {
        this.size = size;
    }
    @Override public double area() { return size * size; }
    @Override public int numberOfSides() { return 4; }
}
```

- A subclass extends the superclass.
- Abstract methods must be implemented: area() and numberOfSides()
- Everything else is “inherited”: x, y, move()

Inheritance

- Although Square did not define a `move()` method, Polygon's `move()` method was inherited:

```
Square square = new Square(10);
square.move(2, 3);
```

- Inheritance is a form of **code reuse**.
- Don't repeat code across classes. Put it in a superclass and inherit it.

Method overriding

- Non-abstract methods can also be overridden.
- The superclass's version of the method can be called with `super`.

```
public class Square extends Polygon {
    ...
    @Override
    public void move(double dx, double dy) {
        super.move(dx, dy);
        System.out.println(“I’m a square and I’m moving!”);
    }
}
```

Constructors

- The subclass constructor must call the superclass constructor first.

```
public abstract class Polygon {
    protected double x, y;
    public Polygon(double x, double y)  {
        this.x = x; this.y = y;
    }
}
public class Square extends Polygon {
    private double size;
    public Square(double x, double y, double size) {
        super(x, y);
        this.size = size;
    }
}
```

DEMO