# Java Web开发基础

第4讲：面向对象编程思想

—类的关系：组合、继承、多态

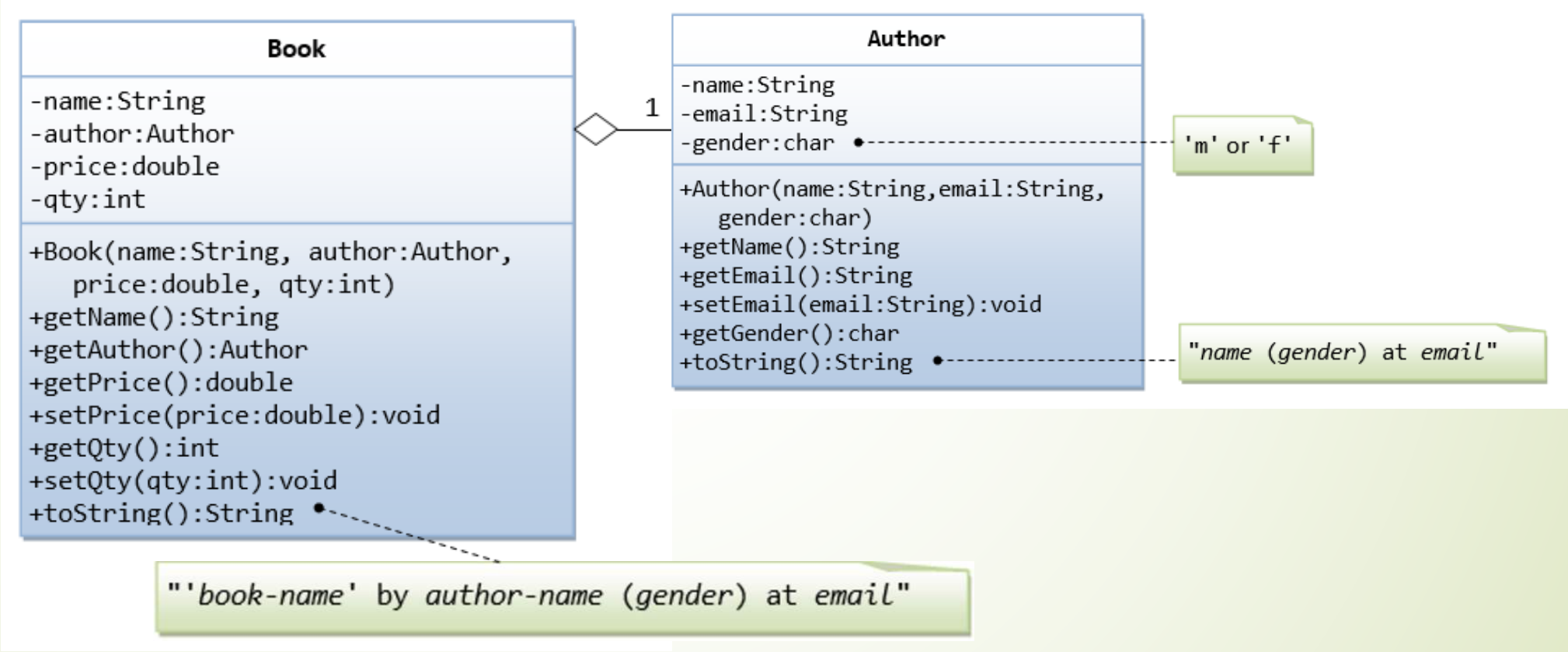
主讲人：康育哲

# 本讲内容

- 组合（Composition）
- 继承（Inheritance）
  - 重写（Override）
  - super关键字
  - protected成员
  - 关于默认构造函数
  - 继承规则
- 多态（Polymorphism）
  - 派生类的可置换性（Substitutability）
  - 向上转换（Upcasting）&向下转换（Downcasting）
  - instanceof运算符
- 抽象（Abstraction）
  - 抽象方法（Abstract Method）&抽象类（Abstract Class）
  - 接口（Interface）&实现（Implementation）
  - 接口vs抽象基类（Abstract Superclass）

# 组合（Composition）

- 概念：类A的某些成员变量是类B的实例，则称类A和B是组合关系



# 组合（Composition）

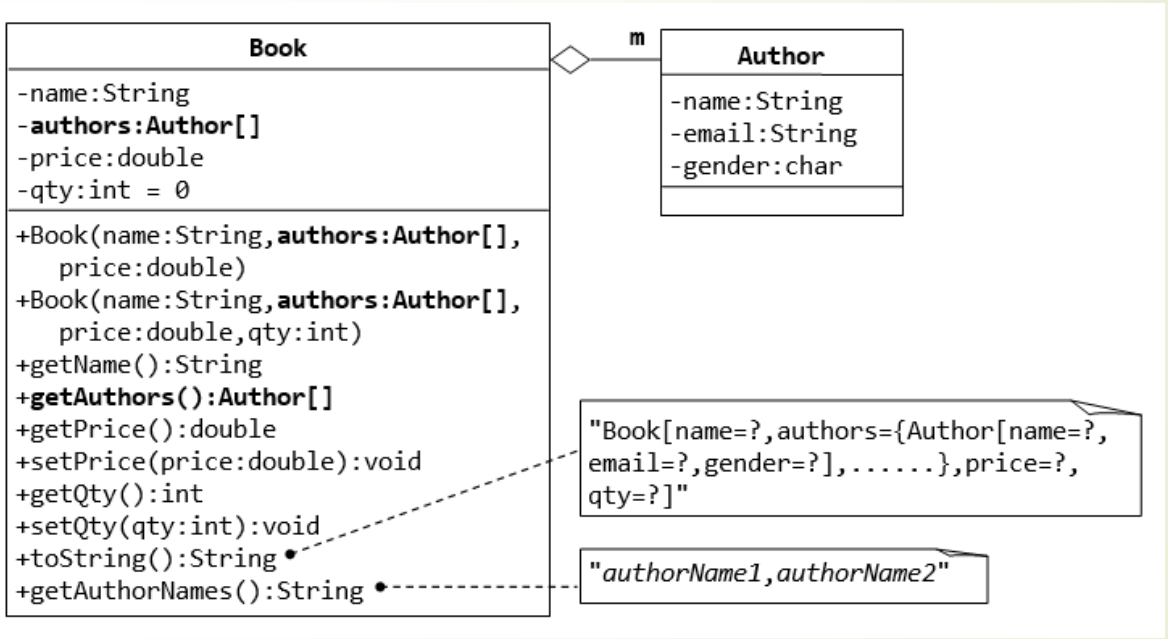
```java
/*
 * The Author class model a book's author.
 */
public class Author {
   // The private instance variables
   private String name;
   private String email;
   private char gender;   // 'm' or 'f'

   // The constructor
   public Author(String name, String email, char gender) {
      this.name = name;
      this.email = email;
      this.gender = gender;
   }

   // The public getters and setters for the private instance variables.
   // No setter for name and gender as they are not designed to be changed.
   public String getName() {
      return name;
   }
   public char getGender() {
      return gender;
   }
   public String getEmail() {
      return email;
   }
   public void setEmail(String email) {
      this.email = email;
   }

   // The toString() describes itself
   public String toString() {
      return name + " (" + gender + ") at " + email;
   }
}
```

```java
/*
 * The Book class models a book with one (and only one) author.
 */
public class Book {
   // The private instance variables
   private String name;
   private Author author;
   private double price;
   private int qty;

   // Constructor
   public Book(String name, Author author, double price, int qty) {
      this.name = name;
      this.author = author;
      this.price = price;
      this.qty = qty;
   }

   // Getters and Setters
   public String getName() {
      return name;
   }
   public Author getAuthor() {
      return author;  // return member author, which is an instance of the class Author
   }
   public double getPrice() {
      return price;
   }
   public void setPrice(double price) {
      this.price = price;
   }
   public int getQty() {
      return qty;
   }
   public void setQty(int qty) {
      this.qty = qty;
   }

   // The toString() describes itself
   public String toString() {
      return "'" + name + "' by " + author;  // author.toString()
   }
}
```

# 组合（Composition）

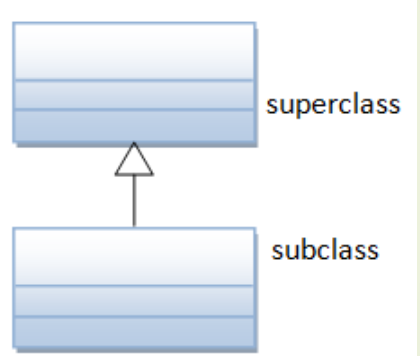
- 练习

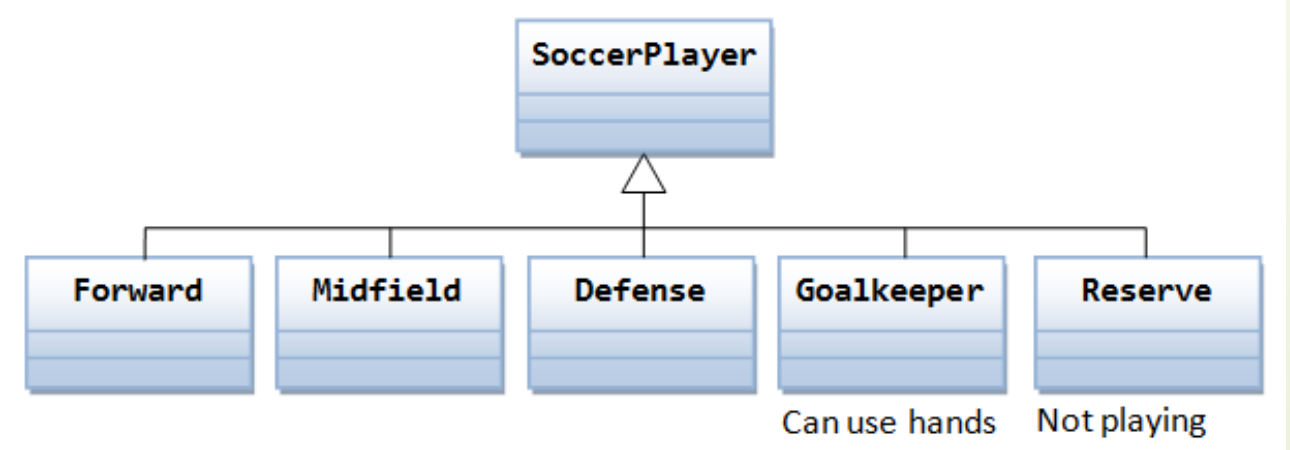
上述示例的扩展实现：一本书可以有多位作者。请根据类图完成对Book类的修改，把原有的author成员变量改为数组，并修改相关方法。Author类的代码可以复用，不用修改。最后写一个含有main方法的测试类TestBook。



# 继承（Inheritance）

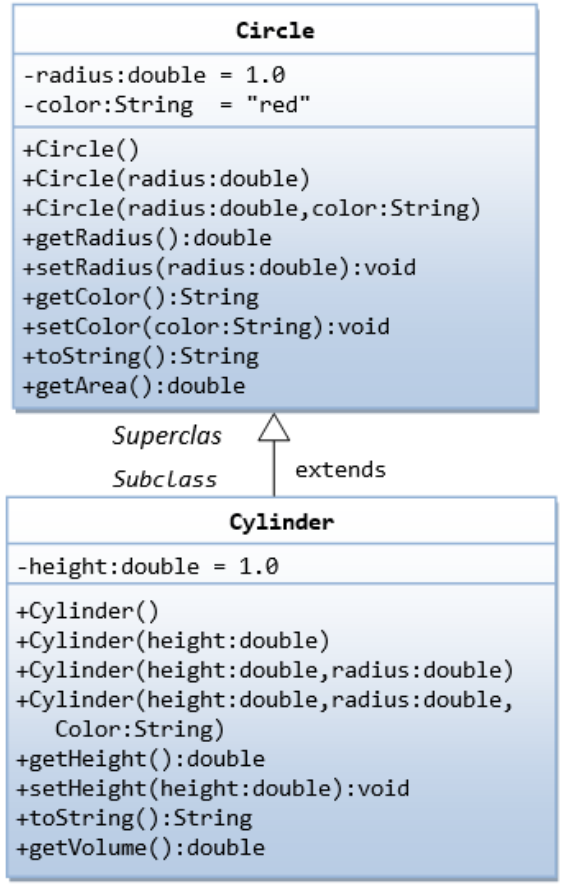
- 概念：如果类A是基于类B的设计和实现做了内部的修改，则称类A继承于类B；类B称为基类（Superclass），类A称为派生类（Subclass）
- 特性：派生类继承基类所有的成员变量和成员方法
- 作用：抽象出公共的属性，减少代码冗余
- 关键字：extends
- UML图示
- 示例



# 继承（Inheritance）

- 重写（Override）
  - 概念：派生类可以直接使用基类方法定义，也可以在不改变积累方法签名的条件下对同名方法定义自己的实现，即为重写
  - 标注（Annotation）：@Override
    - 告诉编译器要检查该方法是否满足重写条件
    - 标注不是必需的
  - 重写vs重载
    - 是否对于在同一个类而言？
      - 重写：否；重载：是
    - 方法签名是否相同？
      - 重写：是；重载：否




```
1  public class Cylinder extends Circle {
2     ......
3     // Override the getArea() method inherited from superclass Circle
4     @Override
5     public double getArea() {
6        return 2*Math.PI*getRadius()*height + 2*super.getArea();
7     }
8     // Need to change the getVolume() as well
9     public double getVolume() {
10       return super.getArea()*height;   // use superclass' getArea()
11    }
12    // Override the inherited toString()
13    @Override
14    public String toString() {
15       return "Cylinder[" + super.toString() + ",height=" + height + "]";
16    }
17 }
```

# 继承（Inheritance）

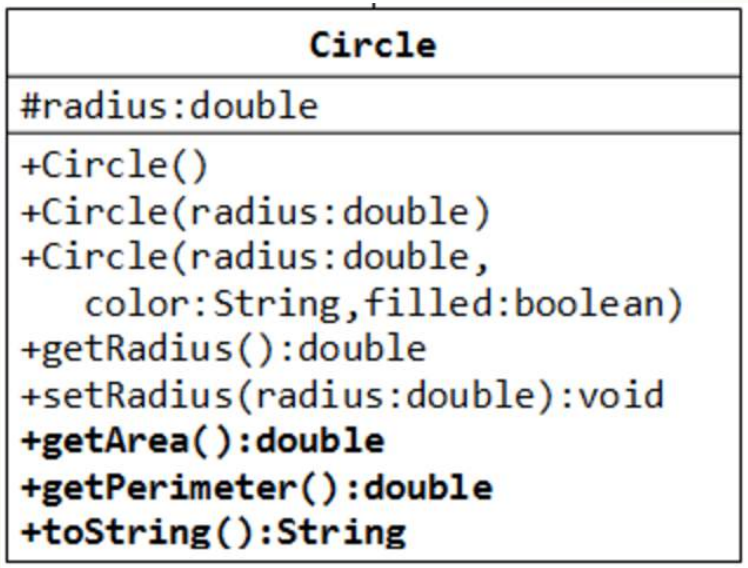
- super关键字
  - 在派生类的重写方法中调用基类的相同方法
  - 在派生类的构造函数中调用基类的构造函数

```
public class Cylinder extends Circle {
   ......
   // Override the getArea() method inherited from superclass Circle
   @Override
   public double getArea() {
      return 2*Math.PI*getRadius()*height + 2*super.getArea();
   }
   // Need to change the getVolume() as well
   public double getVolume() {
      return super.getArea()*height;    // use superclass' getArea()
   }
   // Override the inherited toString()
   @Override
   public String toString() {
      return "Cylinder[" + super.toString() + ",height=" + height + "]";
   }
}
```

```
/*
 * A Cylinder is a Circle plus a height.
 */
public class Cylinder extends Circle {
   // private instance variable
   private double height;

   // Constructors
   public Cylinder() {
      super();  // invoke superclass' constructor Circle()
      this.height = 1.0;
   }
   public Cylinder(double height) {
      super();  // invoke superclass' constructor Circle()
      this.height = height;
   }
   public Cylinder(double height, double radius) {
      super(radius);  // invoke superclass' constructor Circle(radius)
      this.height = height;
   }
   public Cylinder(double height, double radius, String color) {
      super(radius, color);  // invoke superclass' constructor Circle(radius, color)
      this.height = height;
   }
```

# 继承（Inheritance）

- protected成员
  - 类的成员变量和成员方法可以被protected修饰
  - protected成员可以被派生类访问，但不能被外部没有继承关系的类访问
    - private成员不可以被任何外部类访问
  - UML标识：#



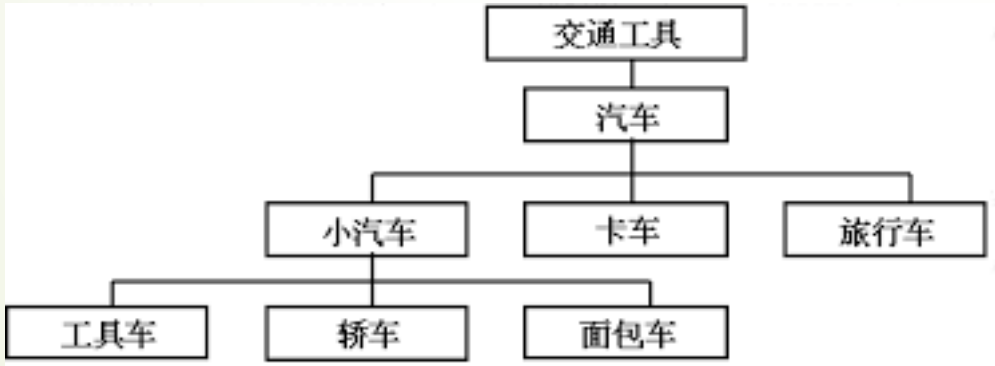
# 继承（Inheritance）

- 关于默认构造方法（default constructor）
  - 如果一个类没有显式定义任何构造方法，Java编译器会为之自动生成一个不带参数的默认构造方法，默认调用基类的默认构造方法
  - 如果基类没有定义默认构造方法，而派生类也没有显式定义构造方法，则编译器报错
  - 只要定义了一个构造方法，则编译器不会自动生成默认构造方法

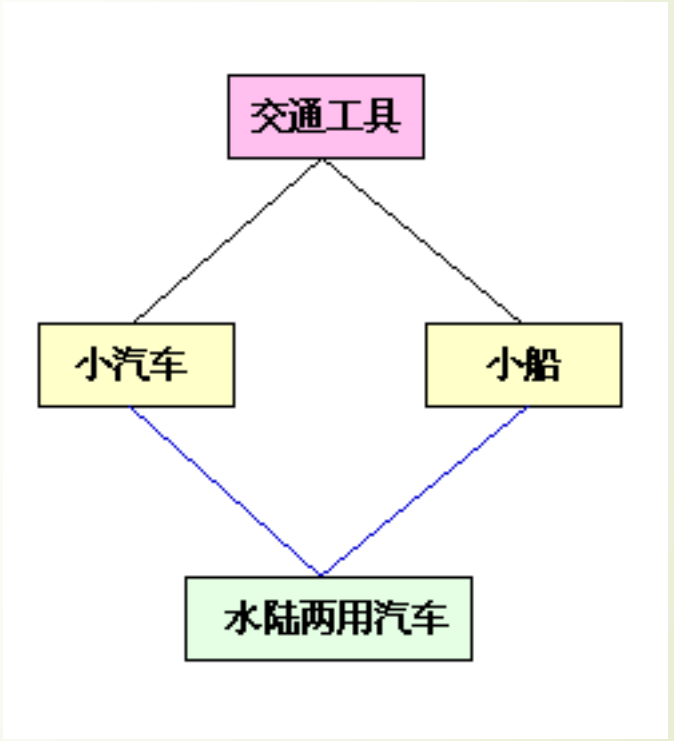
```
// If no constructor is defined in a class, compiler inserts this no-arg constructor
public ClassName () {
   super();   // call the superclass' no-arg constructor
}
```

# 继承（Inheritance）

- 继承规则
  - Java允许多层继承，不允许多重继承
    - 需要多重继承的场合下，一般用组合方式实现
  - 所有Java类都有一个共同基类：java.lang.Object
    - toString()正是Object定义的方法



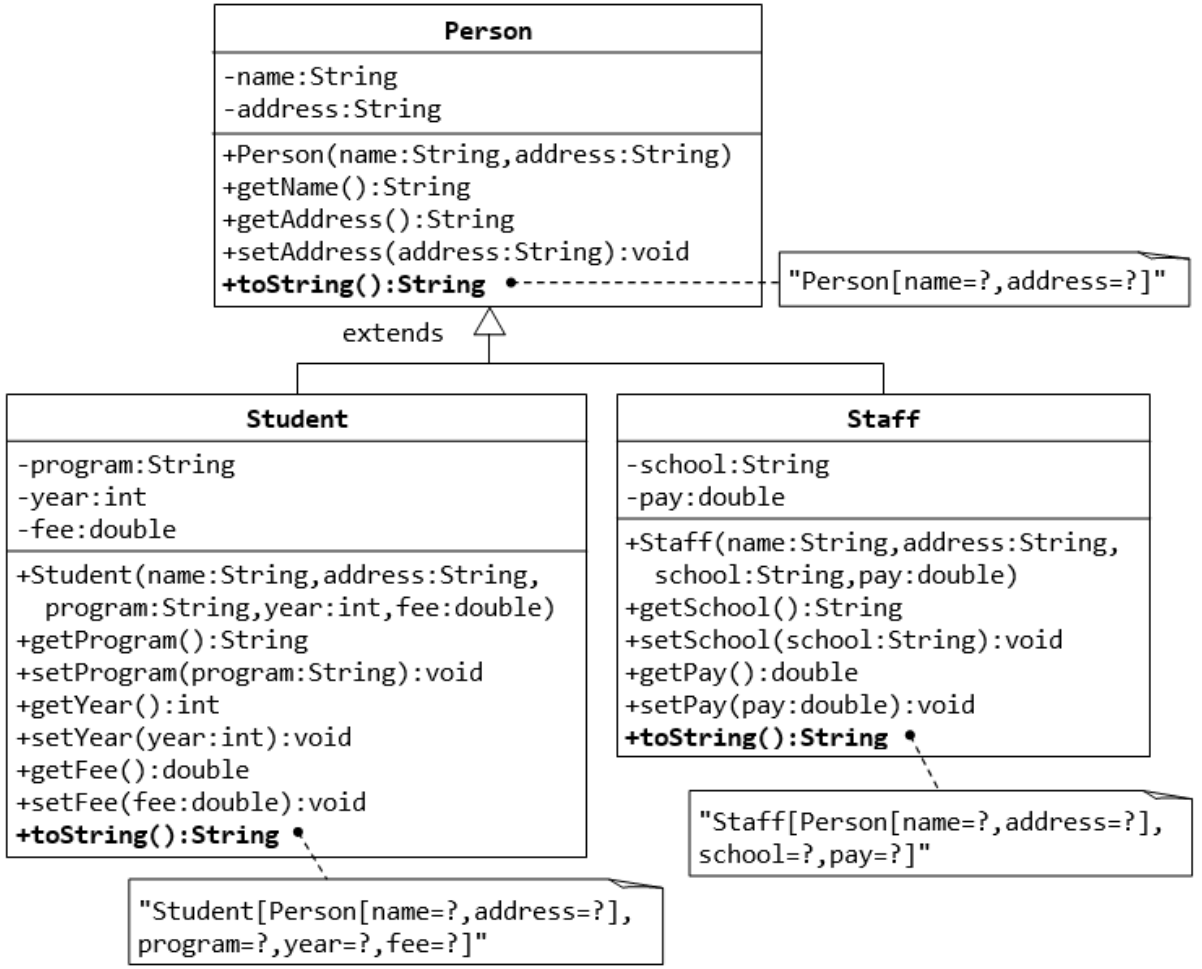

多层继承




多重继承

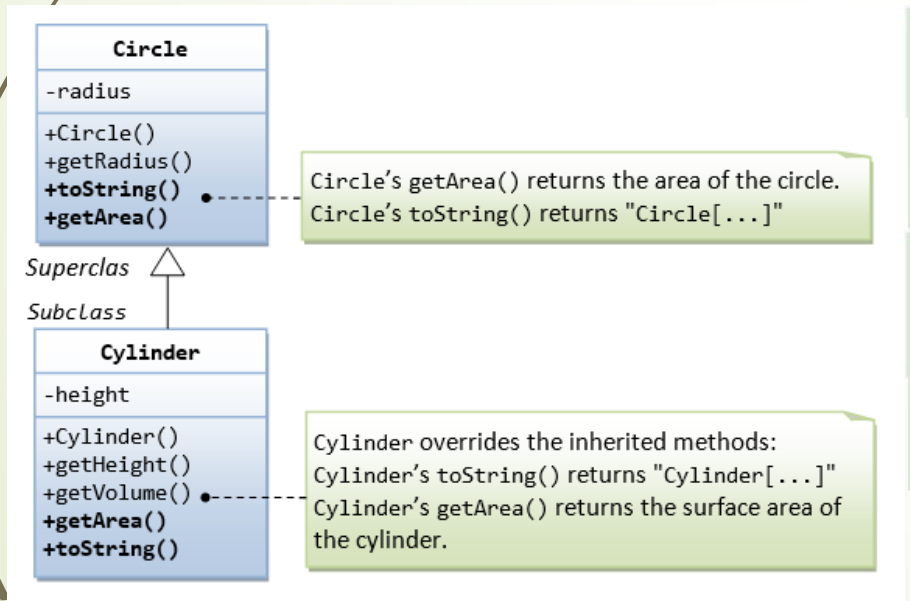
# 继承（Inheritance）

- 练习
  - 根据类图写出代码的实现，并写一个带main方法的测试类TestPerson。



# 多态（Polymorphism）

- 概念：基类实例的同一个动作（方法），由于指向不同的派生类对象而具有不同的表现形式。
- 派生类的可置换性（Substitutability）
  - 派生类具备基类的全部public方法
  - 在只需要基类实例的场合下，更换不同派生类的实例不会对基类方法的调用者产生影响



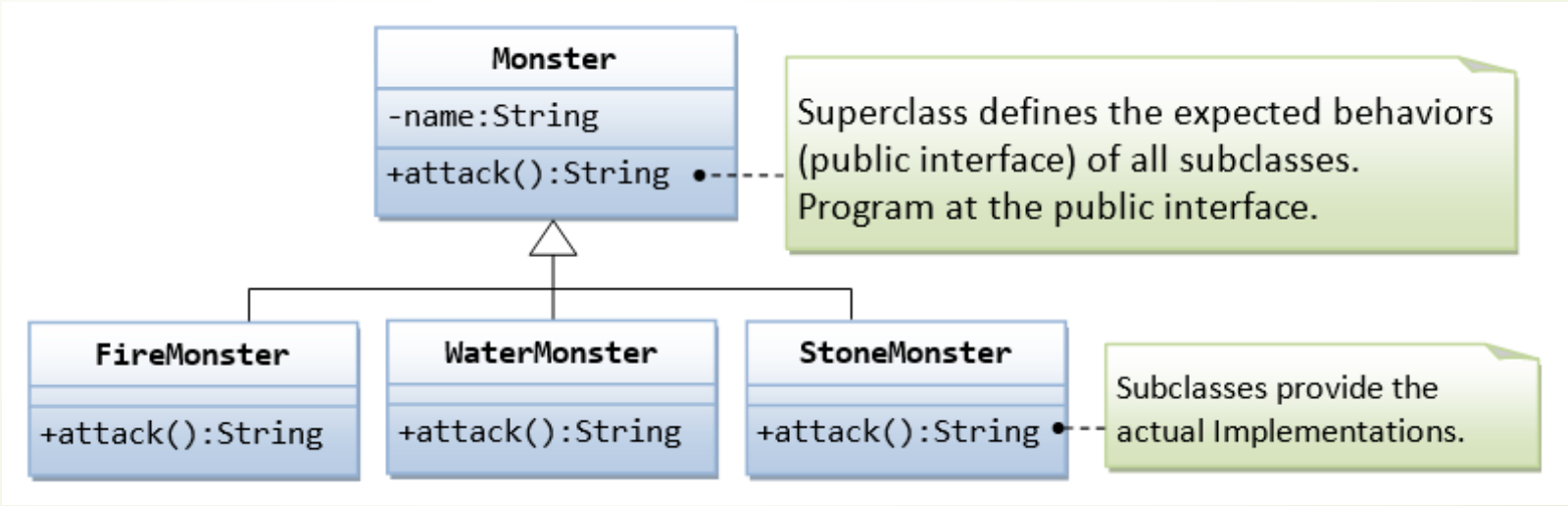

```
// Substitute a subclass instance to a superclass reference
Circle c1 = new Cylinder(1.1, 2.2);

// Invoke superclass Circle's methods
c1.getRadius();

// CANNOT invoke method in Cylinder as it is a Circle reference!
c1.getHeight();  // compilation error
c1.getVolume();  // compilation error

c1.toString();  // Run the overridden version!
c1.getArea();   // Run the overridden version!
```

# 多态（Polymorphism）

- 派生类的可置换性
  - 前提假设
    - class A {
      void m1();
      void m2();
      }
    - class B extends class A {
      void m2();
      void m3();
      }
  - 派生类的实例可以赋给基类的实例变量
    - A a = new B();
  - 如果基类实例变量引用了派生类的实例，就不能对这个变量调用派生类特有的方法
    - a.m1(); a.m2(); // ⭕
    - a.m2(); // ❌
  - 如果基类实例变量引用了派生类的实例，调用的方法若被派生类重写，则调用派生类方法
    - a.m1()实际调用的是B的m2()方法

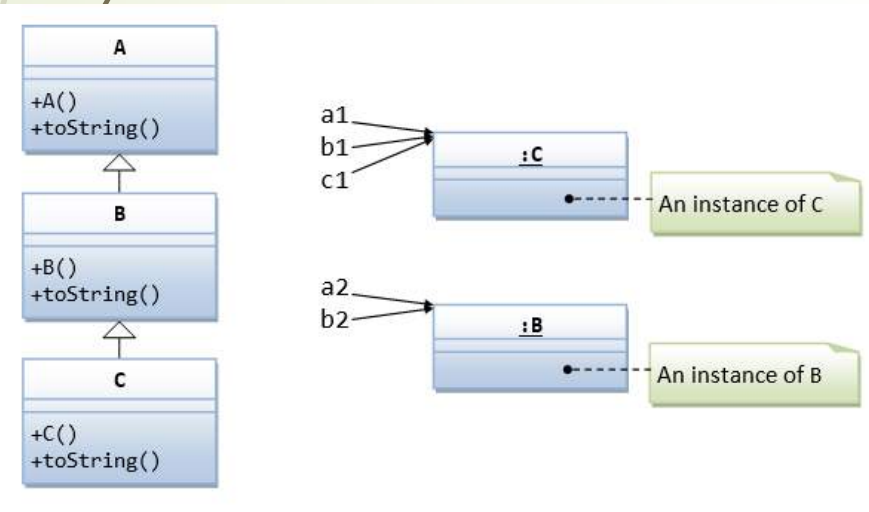
# 多态（Polymorphism）

- 练习
  - 面向对象编程的多态特性可以很好的把接口和实现分离开来。假设你在做一个游戏，可以对玩家生成不同类型的妖怪（目前有火妖、水妖、石妖），每个妖怪都具备发起攻击的动作。请根据类图写出模拟的实现代码，并写一个带main方法的测试类TestMonster。注意测试代码要利用多态特性生成妖怪实例。



# 多态（Polymorphism）

- 向上转换（Upcasting）&向下转换（Downcasting）
  - 向上转换：派生类对象转换为基类的类型
  - 向下转换：基类对象转换为派生类的类型
    - 须使用强制转换运算符
    - 有一定的危险性
  - 没有继承关系的两个类，编译器不允许互相转换




```
public class TestCasting {
   public static void main(String[] args) {
      A a1 = new C();    // upcast
      System.out.println(a1);  // run C's toString()
      B b1 = (B)a1;      // downcast okay
      C c1 = (C)b1;      // downcast okay

      A a2 = new B();  // upcast
      System.out.println(a2);  // run B's toString()
      B b2 = (B)a2;     // downcast okay
      C c2 = (C)a2;     // compilation okay, but runtime error ClassCastException
   }
}
```

# 多态（Polymorphism）

- instanceof运算符
  - 用于判断某个实例是否属于某个类
  - 返回值：boolean类型
  - 用法 *anObject* **instanceof** *aClass*
  - 示例

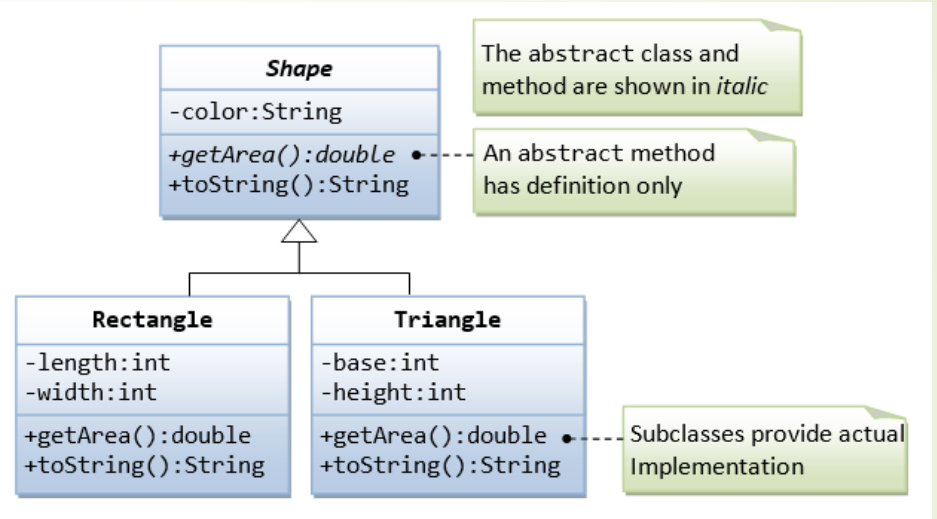
```
Circle c1 = new Circle();
System.out.println(c1 instanceof Circle);  // true

if (c1 instanceof Circle) { ..... }
```

# 抽象（Abstraction）

- 抽象方法&抽象类
  - 抽象方法：只有签名没有实现的方法
    - 使用关键字abstract修饰
    - 必须是public方法
      - 原因：private方法不可以被派生类访问（派生类无法定义实现）
  - 抽象类：包含抽象方法的类
    - 使用关键字abstract修饰
    - 不可直接用于生成实例
      - 原因：定义不完整（有抽象方法未实现）
    - UML图示：*斜体字*

# 抽象（Abstraction）

- 抽象类示例




```
/*
 * This abstract superclass Shape contains an abstract method
 * getArea(), to be implemented by its subclasses.
 */
abstract public class Shape {
   // Private member variable
   private String color;

   // Constructor
   public Shape (String color) {
      this.color = color;
   }

   @Override
   public String toString() {
      return "Shape of color=\"" + color + "\"";
   }

   // All Shape subclasses must implement a method called getArea()
   abstract public double getArea();
}
```

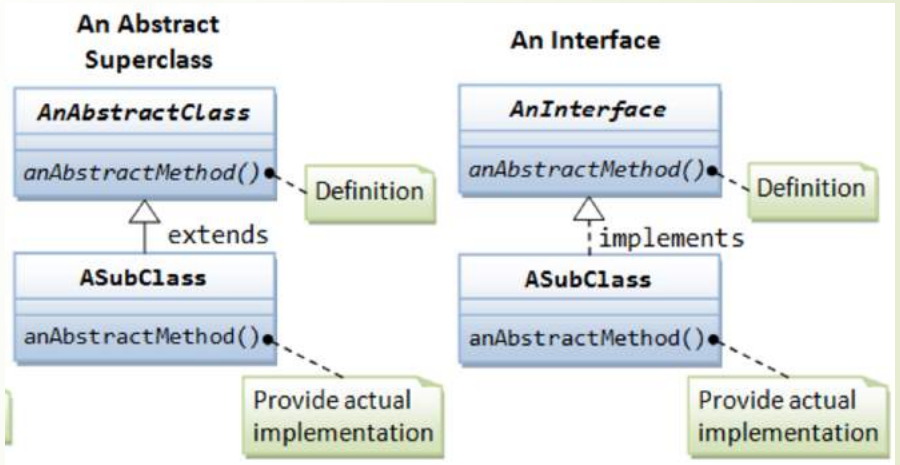
```
public class TestShape {
   public static void main(String[] args) {
      Shape s1 = new Rectangle("red", 4, 5);
      System.out.println(s1);
      System.out.println("Area is " + s1.getArea());

      Shape s2 = new Triangle("blue", 4, 5);
      System.out.println(s2);
      System.out.println("Area is " + s2.getArea());

      // Cannot create instance of an abstract class
      Shape s3 = new Shape("green");   // Compilation Error!!
   }
}
```

# 抽象（Abstraction）

- 练习
  - 前一个打怪游戏的练习中，把基类Monster改为抽象基类，其attack()方法改为抽象方法。

# 抽象（Abstraction）

- 接口（interface）&实现（implementation）
  - 100%的抽象基类，只能含有public的抽象方法
  - 用关键字interface定义
  - 接口的派生类称为实现类，用implements表示继承于接口
    - 接口允许多重继承
    - 实现类必须实现接口的全部方法
  - 命名规范
    - 用形容词（通常为able结尾，表示能力）
    - 首字母大写的驼峰式命名
    - 示例：Serializable、Clonable、Runnable、Movable
  - UML图示
    - 实线箭头表示继承，虚线箭头表示实现



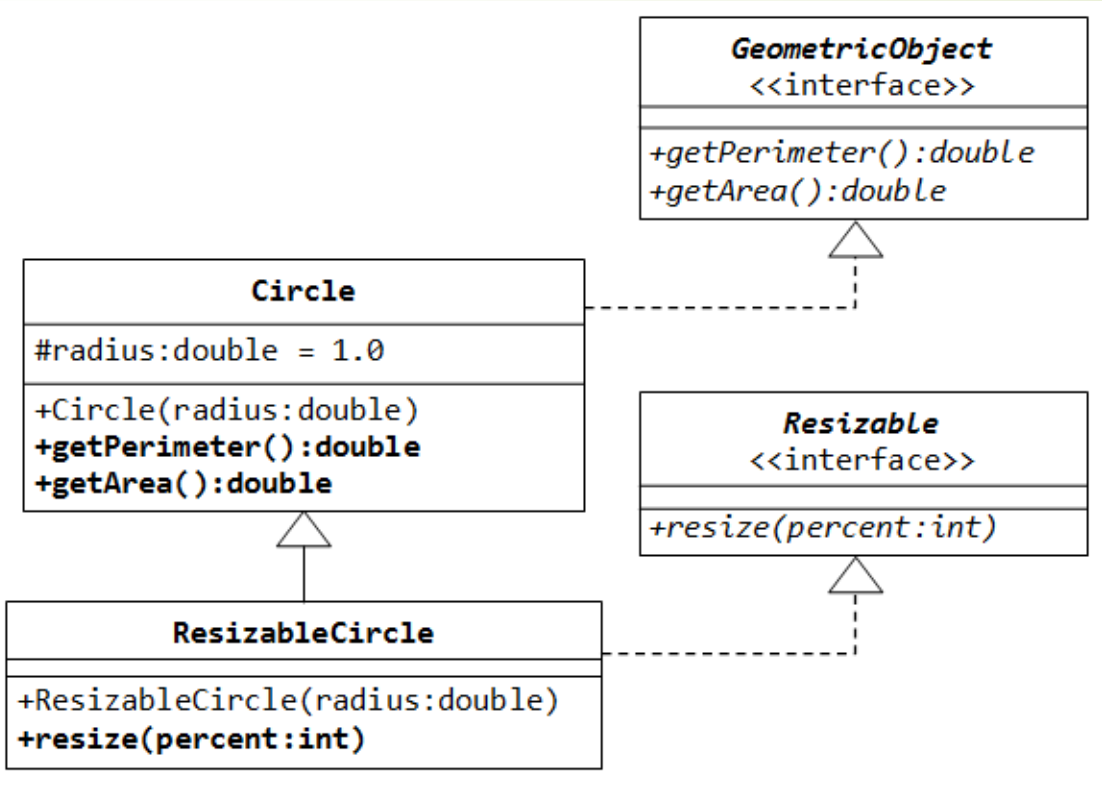

```
/*
 * The interface Shape specifies the behaviors
 * of this implementations subclasses.
 */
public interface Shape {  // Use keyword "interface" instead of "class"
```

```
public class Circle extends Shape implements Movable, Adjustable {
         // extends one superclass but implements multiple interfaces
   .......
}
```

# 抽象（Abstraction）

- 练习
  - 根据类图完成实现代码。

1. 定义表示几何图形的接口GeometricObject
2. 实现表示圆形的类Circle，实现GeometricObject的方法
3. 定义表示可调大小的东西的接口Resizeble
4. 实现表示可调大小的圆形类ResizableCircle，继承Circle，实现Resizable



# 作业

- 根据类图编写实现代码
- Shape是表示图形的抽象类，有两个抽象方法getArea（计算面积）、getPerimeter（计算周长）
- Circle是表示圆的类，继承于Shape，须实现抽象方法；成员变量radius表示半径
- Rectangle是表示矩形的类，继承于Shape，须实现抽象方法；成员变量width表示宽，length表示长
- Square是表示正方形的类，继承于Shape，内部须保证长宽相等
- toString的统一格式（className处输出自身的类名，?输出自身的实际值）：

*className*: color: ?, filled: ?, area = ?, perimeter = ?