《面向对象设计与构造》课程 Lec3-继承、多态与抽象

2018

OO课程组

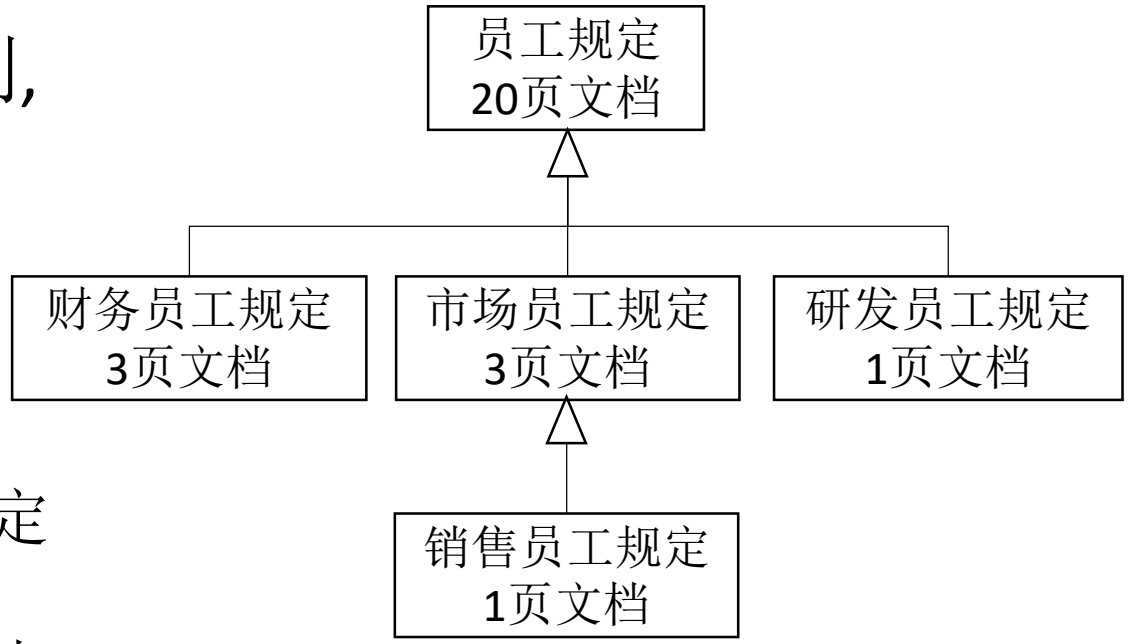
北京航空航天大学

内容提要

- 为什么需要继承
- 与父类的交互
- 对象状态比较
- 多态
- 接口
- 抽象方法与抽象类
- 作业

为什么需要继承机制

- 以公司雇员规定为例
- 基本规定: 工作时间安排, 假期, 福利, 工作守则 ...
  - 所有新招聘员工都要学习基本规定
  - 基本规定手册(比如20页)
- 每个部门都有自己具体的规定
  - 相应部门的员工要学习相应的具体规定(一般1~3页)
  - 这些具体规定会增加一些新的要求，也可能会对基本规定中的条款做出修订



针对不同类的行为分离

- 为什么不针对财务员工直接制定23页的员工规定？24页的市场销售员工规定，21页的研发员工规定？
- 采用多个有侧重点的员工规定的优点：
  - maintenance: 如果公司员工基本规定发生变化，只需修改一次
  - locality: 可以快速找到针对市场销售员工的相关规定
- 领域事实
  - 一般性的员工规定本身就有意义（统一培训等）
  - 部门具体规定也具有重要意义

员工规定

- 考虑下面的员工规定
  - 一周工作40小时
  - 普通员工工资年薪80000, 研发人员增加50000，市场人员增加30000，销售人员增加50000
  - 每年2周假期，销售人员额外增加一周
  - 销售人员使用粉色表格，其他所有员工报销使用黄色表格
- 每个部门的员工都有特定的职责
  - 财务人员审查报销单据
  - 市场人员负责收集产品的市场反馈
  - 研发人员负责按照客户需求研制产品
  - 销售人员负责产品宣传和销售

Employee类

```
public class Employee {
    public int getHours() {
        return 40;              // 一周工作40小时
    }

    public double getSalary() {
        return 80000.0;         // 年薪RMB80000
    }

    public int getVacationDays() {
        return 10;              // 2周带薪假
    }

    public String getReimbursementForm() {
        return "yellow";        // 报销使用黄色表格
    }
}
```

- 如何实现财务人员(FinancialOfficer)类？

冗余的实现方案

```
// A redundant class to represent FinancialOfficer.
public class FinancialOfficer {
    public int getHours() {
        return 40;              // 一周工作40小时
    }

    public double getSalary() {
        return 80000.0;         // 年薪RMB80000
    }

    public int getVacationDays() {
        return 10;              // 2周带薪假
    }

    public String getReimbursementForm() {
        return "yellow";        // 报销使用黄色表格
    }

    public boolean inspect(Bill[] b) {// 审查报销单据
        …
    }
}
```

基于共享的实现方案

- 只有FinancialOfficer才提供inspect，其他的操作与Employee一致

```
// A class to represent FinancialOfficer.
public class FinancialOfficer {
    Employee em; //accessing all the contents defined the Employee class

    public boolean inspect(Bill[] b) {//审查报销单据

        …
    }
}
```

继承

- **inheritance**: 基于已有类来构造新类的机制，可以重用已有的属性和行为
  - 能够把相关的类按照层次组织起来
  - 能够在类之间共享设计与实现

- 继承层次
  - **Superclass**: 被扩展的父类
  - **subclass**: 扩展了superclass的子类
    - 子类自动获得了父类的所有属性和方法
    - **但不见得可以直接访问所有的属性和方法**

```
public class subclass extends superclass {}
```

基于继承的FinancialOfficer

```
// A class to represent FinancialOfficer.
public class FinancialOfficer extends Employee {
    public boolean inspect(Bill[] b) {//审查报销单据
        …
    }
}
```

- 通过使用继承机制，可以只描述专属于子类的属性和方法
  - FinancialOfficer从Employee继承 getHours, getSalary, getVacationDays和getReimbursementForm方法
  - FinancialOfficer增加了一个专有方法inspect

市场人员类

- 市场人员的相关规定:
  - 多获得30000的年薪
  - 有特定职责：收集市场反馈
- 要求：希望从Employee继承已有的方法，并对方法细节进行调整

对父类方法的重写

- **重写(override)**: 针对父类中的给定方法，在子类中提供一个新的实现，以取代从父类通过继承获得的实现

```
public class Marketer extends Employee {
    // 重写getSalary方法
    public double getSalary() {
        return 110000;
    }
    ...
}
```

继承层次

- 如果需要，可以构造多个层次的继承，从而形成一棵继承树
  - 销售人员继承市场人员

```
public class Seller extends Marketer {
      // 重写getReimbursementForm方法
       public String getReimbursementForm() {
          return "pink";
       }

      // 重写getVacationDays方法
      public int getVacationDays() {
          return 15;              // 3周带薪假
      }

      // 重写getSalary方法
      public double getSalary(){
          return 130000;
      }

      public void advertise(Product p) {
          …
      }
 }
```

Seller类中有几个getSalary方法？

与父类的交互

- 有时候需要对superclass的行为进行调整
  - 如给所有员工加薪5000
    - 研发人员薪水：135000
    - 市场人员：115000
    - 销售人员：135000
    - 其他所有：85000
- 哪些类的行为需要调整？
  - Employee, …

修改公共父类

```
public class Employee {
    public int getHours() {
        return 40;              // 一周工作40小时
    }

    public double getSalary() {
        return 85000.0;         // 年薪RMB85000.00
    }

    ...
}
```

- Employee 的多个子类也需要修改
  - 它们重写Employee的getSalary

繁琐的解决方案

```
public class Developer extends Employee {
    public double getSalary() {
        return 135000.0;
    }
    ...
}

public class Marketer extends Employee {
    public double getSalary() {
        return 115000.0;
    }
    ...
}
```

子类重写父类方法时应该建立与父类相应方法的联系，但是目前的解决方案却割裂了！

优化的方案

- 在子类方法中调用父类被重写方法：建立与父类方法的**逻辑连接**

  `super.`**method**`(`**parameters**`)`

  重用父类方法的计算
  修正/调整父类方法的计算

- Example:

```
public class Seller extends Marketer {
    public String getReimbursementForm() {
        return "pink";
    }

    public int getVacationDays() {
        return super.getVacationDays() + 5;
    }

    public double getSalary() {
        return super.getSalary() + 20000.0;
    }
    …
}
```

关于继承的讨论

- 使用继承来解决哪几个问题？
- 继承与类型层次
  - 继承的结果形成了类层次，但不见得是类型层次
  - 实际项目中如何识别和构造类型层次？

继承带来的潜在构造方法问题

- 如果不显式为类定义构造方法，则会有默认的无参数构造方法
  - 按照语言定义的默认规则来初始化成员属性的取值
- 在默认构造方法中会自动调用父类构造方法
  - 通过super()调用
  - 确保所有成员属性都得到初始化
- 一旦定义显式构造方法，则默认规则失效
- 如果给Employee类增加一个Employee(int)方法，会有什么结果？
  - public class Employee{…Employee(int years){…}…}
  - public class Developer extends Employee{…} //不定义Developer构造方法

继承带来的潜在构造方法问题

- 会导致子类编译出现错误

```
Developer.java:2: cannot find symbol
symbol  : constructor Employee()
location: class Employee
public class Developer extends Employee {
       ^
```

  - 原因：我们没有为Developer定义显式构造方法，因此其默认构造方法是Developer()，且同时会调用super()。然而没有Employee()这个方法
- ==》子类不会继承父类的构造器

对父类构造方法的调用

语法：`super(`**parameters**`);`

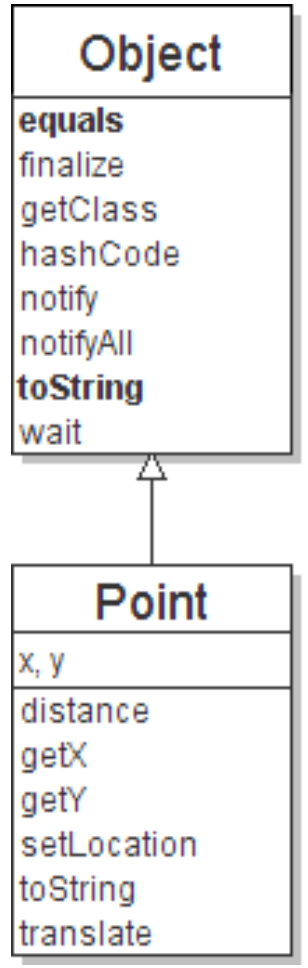
- Example:

```
public class Developer extends Employee {
    public Developer(int years) {
        super(years);  // calls Employee constructor
    }
    ...
}
```

- super调用必须是子类构造方法的第一条语句(为什么？)

Java的Object类

- Object是Java定义的根类
  - 每个类都会默认继承Object
- Object类提供的方法包括
  - `public String toString()`
    返回相应对象的文本表示
  - `public boolean equals(Object other)`
    比较对象的状态是否相同



Object类型对象

- 可以通过Object类型变量来访问任何对象

```
Object o1 = new Point(5, -3);
Object o2 = "hello there";
Object o3 = new Scanner(System.in);
```

- 使用Object类型变量只能访问Object类所定义的方法

```
String s = o1.toString();       // ok
int len = o2.length();          // error
String line = o3.nextLine();    // error
```

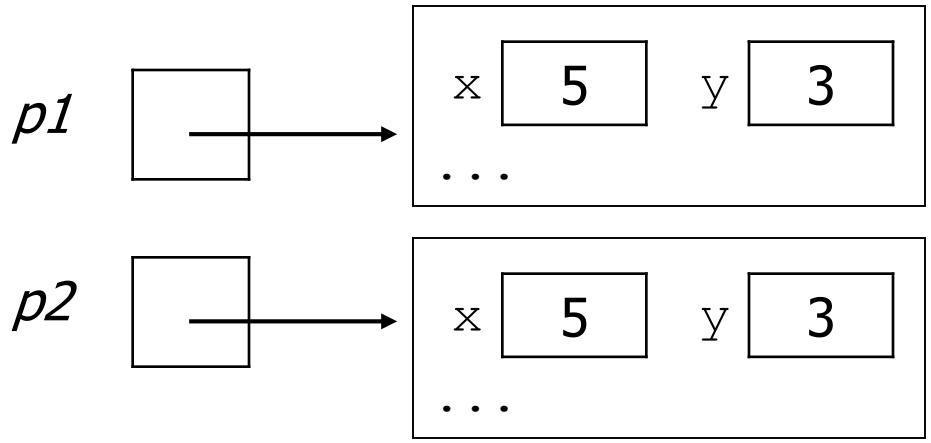
- 为了达到处理的一般性，可以在方法中使用类型为Object的参数

```
public void checkForNull(Object o) {
    if (o == null) {
        throw new IllegalArgumentException();
    }
}
```

对象比较回顾

- ‘==’ 操作符
  - 比较的是两个对象的引用，而不是对象的状态
  - 只有两个变量指向的是同一个对象时，才返回true

```
Point p1 = new Point(5, 3);
Point p2 = new Point(5, 3);
if (p1 == p2) {    // false
    System.out.println(“identical");
}
```



对象比较回顾

- equals方法比较两个对象的状态

```
if (str1.equals(str2)) {
    System.out.println("the strings are equal");
}
```

- 默认情况下，Object提供的equals方法就是比较对象的引用，因为它不拥有任何比较状态的知识

```
if (p1.equals(p2)) {    // false :-(
    System.out.println("equal");
}
```

使用equals的对象状态比较

- 如果是一个不可变对象，则需要重写实现equals方法
  - 比较两个对象的状态是否一致
  - 如对Point对象，当两个Point对象拥有相同的x和y坐标位置，则返回true

- 一种方案：

```
public boolean equals(Point other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

问题：Object类的equals规格是boolean equals(Object)

使用equals的对象状态比较

```
public boolean equals(Object other) {
    if (x == other.x && y == other.y) {
        return true;
    } else {
        return false;
    }
}
```

- 这个实现有什么错误？

类型转换

- 把Object对象转换(type-cast)为关注的对象，如Point.

```
public boolean equals(Object o) {
    Point other = (Point) o;
    return x == other.x && y == other.y;
}
```

```
(other is Point对象) && (x == other.x) &&
(y == other.y)
```

- 如果调用时输入了一个不是Point类型的对象会如何？
- 转换对象的类型只是改变对象引用的规格假设，不改变所指向的对象内存本身

类型转换

```
Point p = new Point(7, 2);
if (p.equals("hello")) {
    …
}
```

- **Point other = (Point) o会抛出异常**

```
Exception in thread "main"
java.lang.ClassCastException: java.lang.String
        at Point.equals(Point.java:xx)
```

使用instanceof来做类型检查

```
if (variable instanceof type)
{
    statement(s);
}
```

- 该关键词用于查询一个变量所引用的对象是否为某个特定类型

```
String s = "hello";
Point p = new Point();
```

- 变量类型和对象类型

表达式	结果
s **instanceof** Point	**false**
s **instanceof** String	true
p **instanceof** Point	true
p **instanceof** String	**false**
p **instanceof** Object	true
s **instanceof** Object	true
null **instanceof** String	**false**
null **instanceof** Object	**false**

最终的equals方法重写结果

```
// Returns whether o refers to a Point object with
// the same (x, y) coordinates as this Point.
public boolean equals(Object o) {
    if (o instanceof Point) {
        // o is a Point; cast and then compare
        Point other = (Point) o;
        return x == other.x && y == other.y;
    } else {
        // o is not a Point; no chance to be equal
        return false;
    }
}
```

对象类型转换的限制

- 可以往上转换
- 往下转换必须非常小心

```
Employee eric = new Marketer();
  ((Marketer) eric).collect (…);      // ok
  ((Seller) eric).advertise(…);  // exception

  // ClassCastException: Marketer object doesn't know how to advertise
```

- 不可以水平转换

```
  Developer linda = new Developer();
  ((FinancialOfficer) linda).inspect(…);     // error
```

多态

- **polymorphism**: 对于给定的方法，获得不同但可归一化的对象行为设计与实现机制
  - 通过overload机制而获得的可归一化方法（重载）
  - 通过override机制而获得的可归一化方法（重写）
- 多态机制为表达复杂设计逻辑提供了简化手段
  - Overload：区分不同场景来重载方法，避免一个方法处理多种场景而导致其逻辑复杂
  - Override：让每个类根据其功能要求来重写所继承的方法，使用者无需区别对待通过统一方式即可获得相应的功能

基于重载的静态多态

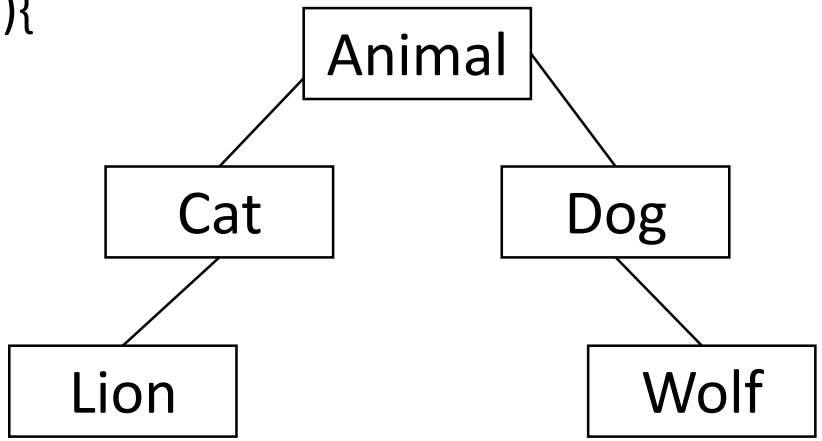
- 从设计角度看，一个方法规范了类的一种能力
- 随着功能的扩展或演化，这种能力也需要演化来处理多种不同的场景，即处理不同的输入情况
- 这时需要对原方法进行overload扩展，从而使得相应能力得到演化
  - 求和(add)是一个类math的方法，int add(int, int)
  - 现在希望扩展该方法的能力，能够支持整数、实数的综合加法： int add(int, double), int add(double, double)
  - 更进一步扩展以支持对不定长集合元素求和： int add(int[]), int add(int[], int[])
- 重载发生在一个类内部，编译时解析方法的多态性（静态）

基于重写的动态多态

重写发生在父类与子类间，子类重新实现从父类继承的方法。
调用一个对象的重写方法时，具体调用哪个则要在运行时通过dispatch机制来确定（动态）

想了解动物如何叫的？每个动物都实现自己的talk方法。

```
void Hear(Animal a){
    a.talk();
}
```




```
public class EmployeeMain {
    public static void main(String[] args) {
        Developer lisa = new Developer();
        Seller steve = new Seller();
        printInfo(lisa);
        printInfo(steve);
    }

    public static void printInfo(Employee empl) {
        System.out.println("salary: " + empl.getSalary());
        System.out.println("v.days: " + empl.getVacationDays());
        System.out.println("r.form: " + empl.getReimbursementForm());
        System.out.println();
    }
}
```

OUTPUT:

```
salary: 130000.0
v.days: 10
r.form: yellow

salary: 130000.0
v.days: 15
r.form: pink
```

通过父类对象来归一化使用重写方法

- 通过一个父类型的数组来存储所有可能的子类型对象，从而组织简洁的代码

```
public class EmployeeMain2 {
    public static void main(String[] args) {
        Employee[] e = { new Developer(),   new Marketer(),
                         new Seller(), new FinancialOfficer() };

        for (int i = 0; i < e.length; i++) {
            System.out.println("salary: " + e[i].getSalary());
            System.out.println("v.days: " + e[i].getVacationDays());
            System.out.println();
        }
    }
}
```

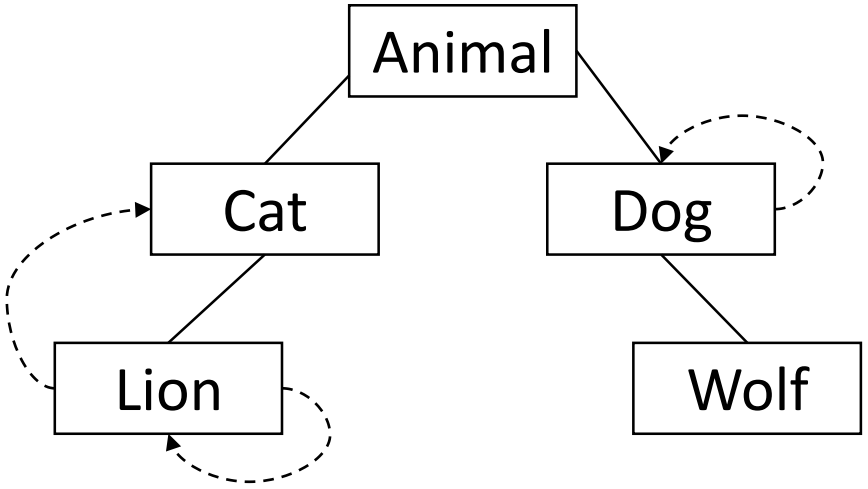
Output:

```
salary: 130000.0
v.days: 10
salary: 110000.0
v.days: 10
salary: 130000.0
v.days: 15
salary: 80000.0
v.days: 10
```

重写多态的调用分析

- 给定类型层次
  - 不同层次的类会选择性重写某些方法
  - 一个类的方法可能会调用该类自身或上面层次类的方法
  - 需要根据具体对象的类型来分析具体调用哪个方法

```
void Hear(Animal a){
    a.talk();
}
```



例子分析

```
public class Foo {
    public void method1() {
        System.out.println("foo 1");
    }

    public void method2() {
        System.out.println("foo 2");
    }

    public String toString() {
        return "foo";
    }
}

public class Bar extends Foo {
    public void method2() {
        System.out.println("bar 2");
    }
}
```

```
public class Baz extends Foo {
    public void method1() {
        System.out.println("baz 1");
    }

    public String toString() {
        return "baz";
    }
}

public class Mumble extends Baz {
    public void method2() {
        System.out.println("mumble 2");
    }
}
```

```
Foo[] pity = {new Baz(), new Bar(), new
  Mumble(), new Foo()};
for (int i = 0; i < pity.length; i++) {
    System.out.println(pity[i]);
    pity[i].method1();
    pity[i].method2();
    System.out.println();
}
```

用5分钟时间给出执行结果，注意

先梳理类的层次关系和方法调用

重写方法执行分析表

method	Foo	Bar	Baz	Mumble
method1	foo 1	*foo 1*	baz 1	*baz 1*
method2	foo 2	bar 2	*foo 2*	mumble 2
toString	foo	*foo*	baz	*baz*

更复杂的例

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    System.out.println();
    food[i].b();
    System.out.println();
}
```

- 一个方法会调用另外一个方法，且混杂着方法的重写

```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
```

```
public class Yam extends Lamb {
    public void a() {
        System.out.print("Yam a   ");
        super.a();
    }
    public String toString() {
        return "Yam";
    }
}

public class Spam extends Yam {
    public void b() {
        System.out.print("Spam b   ");
    }
}
```

更复杂的例子

- Lamb继承了Ham的方法a，a调用 b，但是Lamb重写了方法 b...

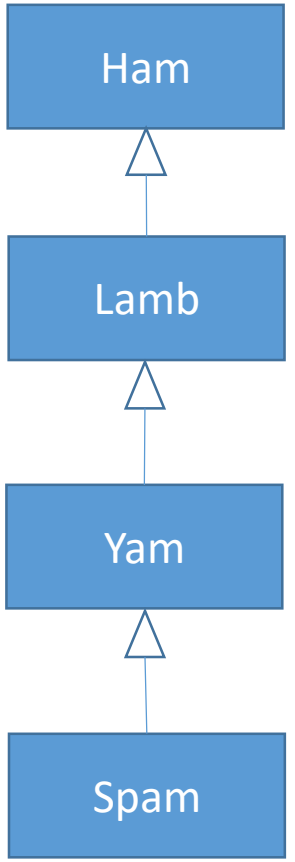
```
public class Ham {
    public void a() {
        System.out.print("Ham a   ");
        b();
    }
    public void b() {
        System.out.print("Ham b   ");
    }
    public String toString() {
        return "Ham";
    }
}

public class Lamb extends Ham {
    public void b() {
        System.out.print("Lamb b   ");
    }
}
```

- Lamb对象的方法a执行结果（为什么不是Ham b？）：

  Ham a **Lamb b**

表格分析法

method	Ham	Lamb	Yam	Spam
a	Ham a **b()**	*Ham a* ***b()***	Yam a Ham a **b()**	*Yam a* *Ham a* ***b()***
b	Ham b	Lamb b	Lamb b	Spam b
toString	Ham	*Ham*	Yam	*Yam*



最终结果

```
Ham[] food = {new Lamb(), new Ham(), new Spam(), new Yam()};
for (int i = 0; i < food.length; i++) {
    System.out.println(food[i]);
    food[i].a();
    food[i].b();
    System.out.println();
}
```

- 输出

```
Ham
Ham a   Lamb b
Lamb b

Ham
Ham a   Ham b
Ham b

Yam
Yam a   Ham a   Spam b
Spam b

Yam
Yam a   Ham a   Lamb b
Lamb b
```

接口

- 有些类虽然没有层次关系，但可以有共性行为
  - `Circle`,`Rectangle`和`Triangle`
  - 它们需要的共性行为包括
    --计算几何形状的外周长(perimeter)
    --计算几何形状的面积(area)

- 对于不同的几何类型，这些共性行为的内涵解释(即计算方式)可能都不同
  - 使用接口，而不是公共父类(因为没什么具体计算行为可复用)

接口

- Rectangle (宽*w*, 高*h*):

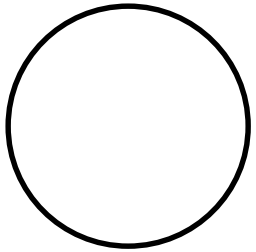
  area = $w\ h$

  perimeter= $2w + 2h$

- Circle (半径*r*):

  area = $\pi\ r^2$

  perimeter= $2\ \pi\ r$

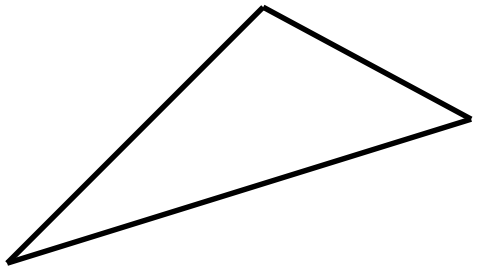


- Triangle (边长*a*, *b*, *c*)

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

  其中$s = ½\ (a + b + c)$

  perimeter= $a + b + c$



接口

- 为不同几何类型实现相应的方法`perimeter`和`area`.
- 目标：客户代码无需区分不同的几何类型
  - 直接获得任意几何形状的面积和周长
  - 能够创建数组来管理各种可能的几何对象
  - 能够在屏幕上画出几何形状

接口

- **interface**: 对一组类共性行为的抽取结果，使得设计规格和实现相分离
  - 继承是一种层次抽象和代码复用机制
  - 接口是一种行为层次抽象，无关代码复用
  - 行为职责类比:
    - “我是授课老师。这表明我知道如何设置课程大纲、如何设置作业、如何出考试卷"
    - “我是一个封闭的几何形状。这表明我知道如何计算我的面积和周长"

接口

```
public interface name {
    public type name(type name, ..., type name);
    public type name(type name, ..., type name);
     …
}
```

Example:

```
public interface Vehicle {
    public double speed();
    public void setDirection(int direction);
}
```

- **接口中的方法是抽象方法(abstract method)**，即不提供实现的方法
  - 允许/要求相应的类来实现相应的接口

接口的实现

```
public class name implements [interface1], [interface2]... {
    ...
}
```

- Example:

```
public class Bicycle implements Vehicle {
    ...
}
```

- 如果一个类实现(*implements*)一个接口
  - 这个类必须包括接口中规定的所有方法，否则无法通过编译

接口及其实现举例

```
// Represents circles.
public class Circle implements ClosedShape {
    private double radius;

    // Constructs a new circle with the given radius.
    public Circle(double radius) {
        this.radius = radius;
    }

    // Returns the area of this circle.
    public double area() {
        return Math.PI * radius * radius;
    }

    // Returns the perimeter of this circle.
    public double perimeter() {
        return 2.0 * Math.PI * radius;
    }
}
```

```
public interface ClosedShape {
    public double area();
    public double perimeter();
}
```

Interface + polymorphism

- 接口可规范一组具有共性特征类的共同行为，客户代码结合多态机制可简化实现

```
public static void printInfo(ClosedShape s) {
    System.out.println("The shape: " + s);
    System.out.println("area : " + s.area());
    System.out.println("perim: " + s.perimeter());
}
```

  - 任何*实现了相应接口的对象*都可以传递进行处理

```
Circle circ = new Circle(12.0);
Rectangle rect = new Rectangle(4, 7);
Triangle tri = new Triangle(5, 12, 13);
printInfo(circ);
printInfo(tri);
printInfo(rect);
```

就该例子而言，比较使用继承与接口的区别！

Interface + polymorphism

- 实现多个接口，但是有同名方法

```
public interface IA{
    public int m();
}
```

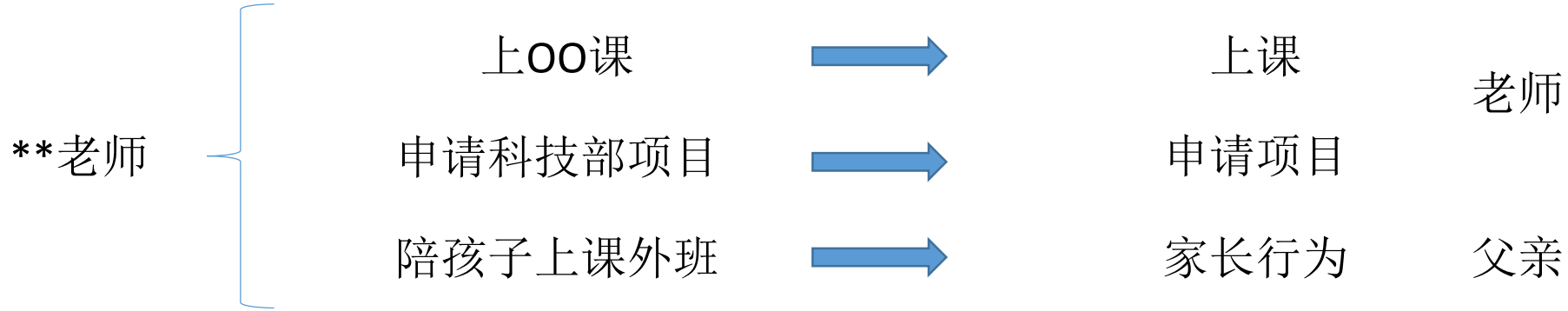
```
public interface IB{
    public void m();
}
```

```
public class Test implements IA, IB{
    public int m(){
        return 0;
    }

    public void m(){
        System.out.println(“hello”);
    }
}
```

继承与接口实现的差异

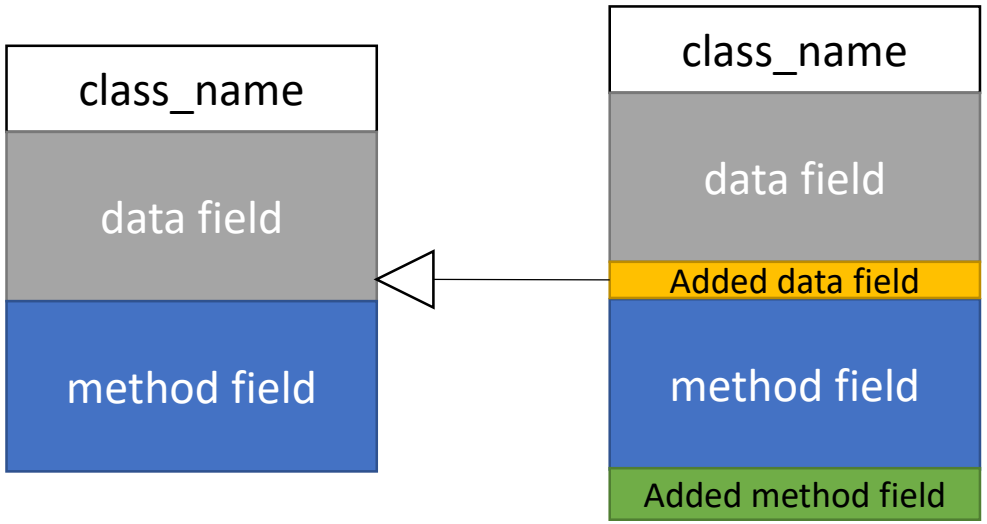
- 从机制的角度：
  - 继承：关注类型抽象，建立对象内在属性和行为的抽象层次

  - 接口实现：针对共性行为抽象（接口），把对象纳入到共性行为层次中

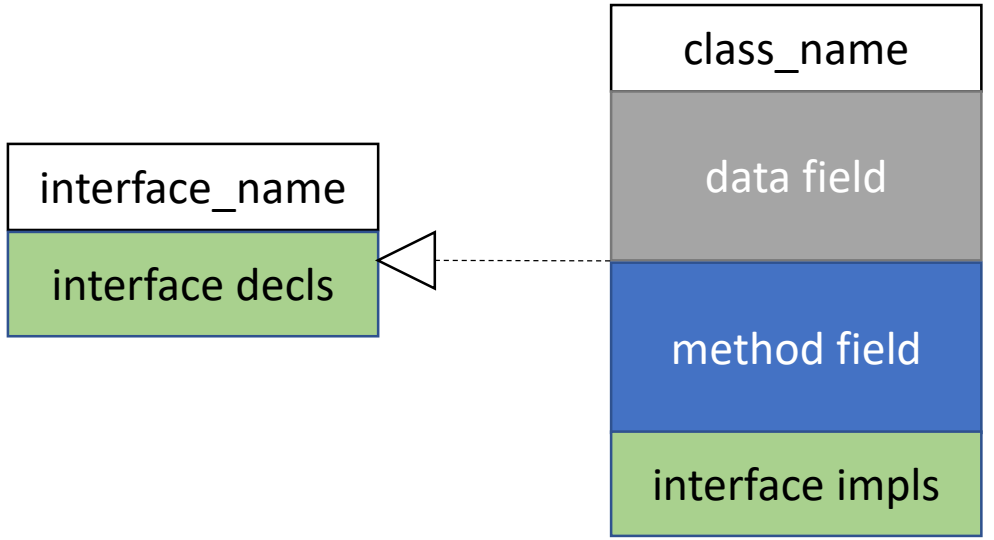


类型抽象与接口抽象

- 类型抽象关注数据以及在数据状态上的操作行为
- 接口抽象关注具有外在统一语义的操作行为



- 给定两个类B和C，都继承自A
  - 你能对B和C做什么概括？



- 给定两个类D和E，都实现接口Y
  - 你能对D和E做什么概括？

继承与接口实现的差异

- 从设计的角度：
  - 继承：“is a”的关系，子类“是一个”父类
  - 接口实现：“can do”的关系，实现类可以做接口规定的行为
  - 从两个方向进行设计：
    - top-down：如果不关心下层和上层在数据上是否有共性，应该使用接口与接口实现机制，否则就应使用继承机制。
    - bottom-up：b-u方向，如果下层没必要从上层获得重用数据（定义），则应使用接口及接口实现机制；否则应使用继承机制

继承与接口实现的差异

- 从使用的角度：
  - 继承：一个类只能继承一个父类
    - 当子类重写父类方法或覆盖父类属性时，Java根据所引用对象的实际类型来确定所要访问的具体方法和属性
  - 接口实现：一个类可以实现多个接口
    - 接口中只有抽象操作
    - 一个类实现多个接口，编译时可确定哪个方法被调用

抽象方法与抽象类

- 接口类中定义的方法都是抽象方法
- 也可以在普通类中声明抽象的方法

  public abstract void draw(int size);
- 任何一个包括抽象方法的类都是抽象类，必须使用abstract关键词

  abstract class MyClass {...}
- 抽象类是一个不完整的类，因而不能实例化

抽象方法与抽象类

- 抽象类可以通过继承机制被扩展
  - 如果子类提供了抽象类中所有抽象方法的实现，则子类可以被实例化
  - 否则，子类仍然是抽象类(使用abstract关键词)
- 即便一个类不拥有抽象方法，仍然可以声明它为抽象类，从而阻止对该类进行实例化

抽象方法与抽象类

- 对照前面介绍的接口类ClosedShape，也可声明为普通类，它的子类包括Oval, Rectangle, Triangle, Hexagon等
- ClosedShape是一种抽象的概念，需要阻止程序员来直接实例化
  - 把ClosedShape声明为抽象类
  - 用户可以实例化Oval和Rectangle等对象，并通过其父类型ClosedShape来引用

```
public abstract class ClosedShape {
    abstract int draw();
    abstract double perimeter();
    abstract double area();
}
```

抽象类与接口

- 共同点：
  - 都是上层的抽象层
  - 都不能被实例化
  - 都能包含抽象的方法，这些抽象的方法用于描述具备的功能，但是不提供具体的实现
- 不同点：抽象类用于继承，接口用于实现
  - 在抽象类中可以写非抽象的方法，从而避免在子类中重复书写他们，这样可以提高代码的复用性
  - 接口中只能有抽象的方法

作业

- 继续第二次作业的单电梯系统，基本功能要求保持不变
- 新增功能
  - 要求同时满足如下两个原则，一旦违反视为wrong
    - 电梯在完成对一个请求的响应之前，不能改变运动方向
    - 电梯在响应一个请求的过程中，如果有在当前运动方向上能够响应的请求，则必须进行响应，这样的请求称为“捎带响应请求”
  - 要求按照电梯实际响应情况，在楼层类记录相关的请求数、电梯经过和停靠的次数，并命令行输出
- 新增设计要求(设计要求也要进行检查，如果违背报incomplete型bug)
  - 使用继承机制，不要覆盖或删除第二次作业的傻瓜调度schedule代码，增加一个子类来重写schedule方法，注意子类方法要与父类方法有交互
  - 使用interface来归纳电梯的运动方法
  - 重写toString方法来获得电梯运行状态和时刻的观察

作业提示

ALS_Schedule (A Little Smart Schedule)
(1)只要队列不为空，每次都取出队列头请求来调度（同傻瓜调度策略）
(2)电梯在运动过程中不能突然改变运动方向
(3)在调度电梯完成一个请求的过程中，可以让电梯完成“顺路”的楼层请求
(4)调度算法要确保电梯在当前运动方向上完成所有能完成的**电梯内请求**

关于“顺路捎带”的请求
设电梯当前状态为e=(e_n,sta,n)，即当前所处楼层为e_n，运动状态为sta，当前运动的目标为楼层n，则
(1) (e.sta = UP → 10>=e.n>e.e_n) || (e.sta = DOWN→1<=e.n<e.e_n)
(2)对于任意一个楼层请求r=(FR,n,dir,t)，如果是电梯当前运动状态下的顺路请求，则一定有:
(r.dir=e.sta) && ((r.dir=UP→(r.n <= e.n)&&(r.n>e.e_n)) || (r.dir=DOWN→(r.n>=e.n)&&(r.n<e.e_n)))
(3)对于任意一个电梯运载请求r=(ER,n,t)，如果是电梯当前运动状态下的顺路请求，则一定有:
(e.sta=UP→(r.n > e.e_n)) || (e.sta=DOWN→ (r.n<e.e_n))

作业要求

- 提交物
  - 源代码、readme
  - 注意代码一定提交到gitlab上，否则无法提交作业
- 测试要求
  - 设计请求序列，重点检查是否违背电梯运动的两个原则
  - 针对新增的三个设计要求进行检查
    - 未实现，作为incomplete类bug报告，需要写清楚哪个要求不满足，相应的类和方法也要报告
- 建议
  - 第二次作业的bug一定要修复，否则引入ALS调度后会导致产生很多新的bug，甚至被报告第二次作业就存在的bug（且不能进行申诉）
  - 开发和测试时一定要抓住电梯状态和请求队列状态


第三次作业公测
非法输入
格式错误
随机字符串
关键字错误
FR/ER
UP/DOWN
括号匹配失败
缺半边括号
有多余括号
超出范围
楼层数超出范围（1-10）
时间超出范围（0-2^31-1）
特殊情况
1楼不能下，10楼不能上
输入未按照时间顺序排列
第一个请求不是(FR,1,UP,0)
时间非整数
正常情况测试
单一捎带
e.sta==UP(1<=e.e_n
楼层请求r.n<=e.e_n：不捎带
楼层请求e.e_n
楼层请求r.n>e.n：不捎带
电梯请求r.n<=e.e_n：不捎带
电梯请求e.e_n
电梯请求r.n>e.n：捎带
e.sta==DOWN(1<=e.n
楼层请求r.n>=e.e_n：不捎带
楼层请求e.n<=r.n
楼层请求r.n
电梯请求r.n>=e.e_n：不捎带
电梯请求e.n<=r.n
电梯请求r.n
e.sta==STILL
完成所有请求停在某一楼层，没有顺路捎带
在捎带过程中停靠在某楼层，捎带情况根据电梯挺靠前状态决定
合法输入
多个捎带
e.sta==UP(1<=e.e_n
请求楼层全部处于(e.e_n,r主.n]，全部捎带
部分请求处于[1,e.e_n]和(r主.n,10]，这类请求在本次主请求处理完成后，选队列位置最前的成为主请求
e.sta==DOWN(1<=e.n
请求楼层全部处于[r主.n,e.e_n)，全部捎带
部分请求处于[1,r主.n)和[e.e_n,10]，这类请求在本次主请求处理完成后，选队列位置最前的成为主请求
e.sta==STILL
完成所有请求停在某一楼层，直接选择队列最前请求为主请求
在捎带过程中停靠在某楼层，处理完当前主请求后重新寻找主请求
边界情况测试
副请求发出时间==主请求发出时间，捎带
副请求时间==主请求开门时间，不捎带
副请求时间==主请求开关门完毕时间，不捎带
其他功能测试
同质：开关门完毕前，该楼层多个同向请求认为是同一个
乱序：请求产生时间小于前一个请求的产生时间，直接删除