OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
  - Project 6: 6月8日晚9点

- 复习
  - 继承
    - 代码重用
    - 向上转换 (upcasting) 和多态
      - 父类的引用可以指向子类的对象
      - 通过父类引用调用子类对象的方法
  - 接口
    - ~~代码重用~~
    - 向上转换 (upcasting) 和多态

问题：
如何向上转换到多个类型？

- 复习
  - 抽象类
    - 抽象类包含抽象方法，只有方法名，参数，返回值，没有方法的实现
    - 抽象类不能实例化
    - 若子类没有重写父类中的抽象方法，子类仍为抽象类

```
abstract class Instrument {
    public abstract void play(int note) ;
}
```

```
public class Wind extends Instrument {
    public void play(int note) {
        System.out.println("Wind.play()" + n);
    }
}
```

```
public class Stringed extends Instrument {
    public void play(int note) {
        System.out.println("Stringed.play()" + n);
    }
}
```

- 复习
  - 接口
    - “ 所有方法都是抽象方法”
    - 一个类可以实现多个接口

```
interface Instrument {
    void play(int note) ;
    String what();
}
```

```
class Stringed implements Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
   public String what() {return “Stringed”;}
}
```

```
interface CanFight {
    void fight();
}
```

```
interface CanSwim {
    void swim();
}
```

```
interface CanFly {
    void fly();
}
```

```
class ActionCharacter {
    public void fight() { }
}
```

```
class Hero extends ActionCharacter
         Implements CanFight, CanSwim, CanFly{
    public void fly() { }
    public void swim() { }
}
```

```
public class Adventure {
    public static void t(CanFight x) { x.fight();}
    public static void u(CanSwim x) { x.swim();}
    public static void v(CanFly x) { x.fly();}
    public static void w(ActionCharacter x) { x.fight();}
    public static void main(String []args) {
        Hero h = new Hero();
        t(h);   u(h);   v(h);   w(h);
    }
}
```

实现多个接口可以 upcast 到不同的类

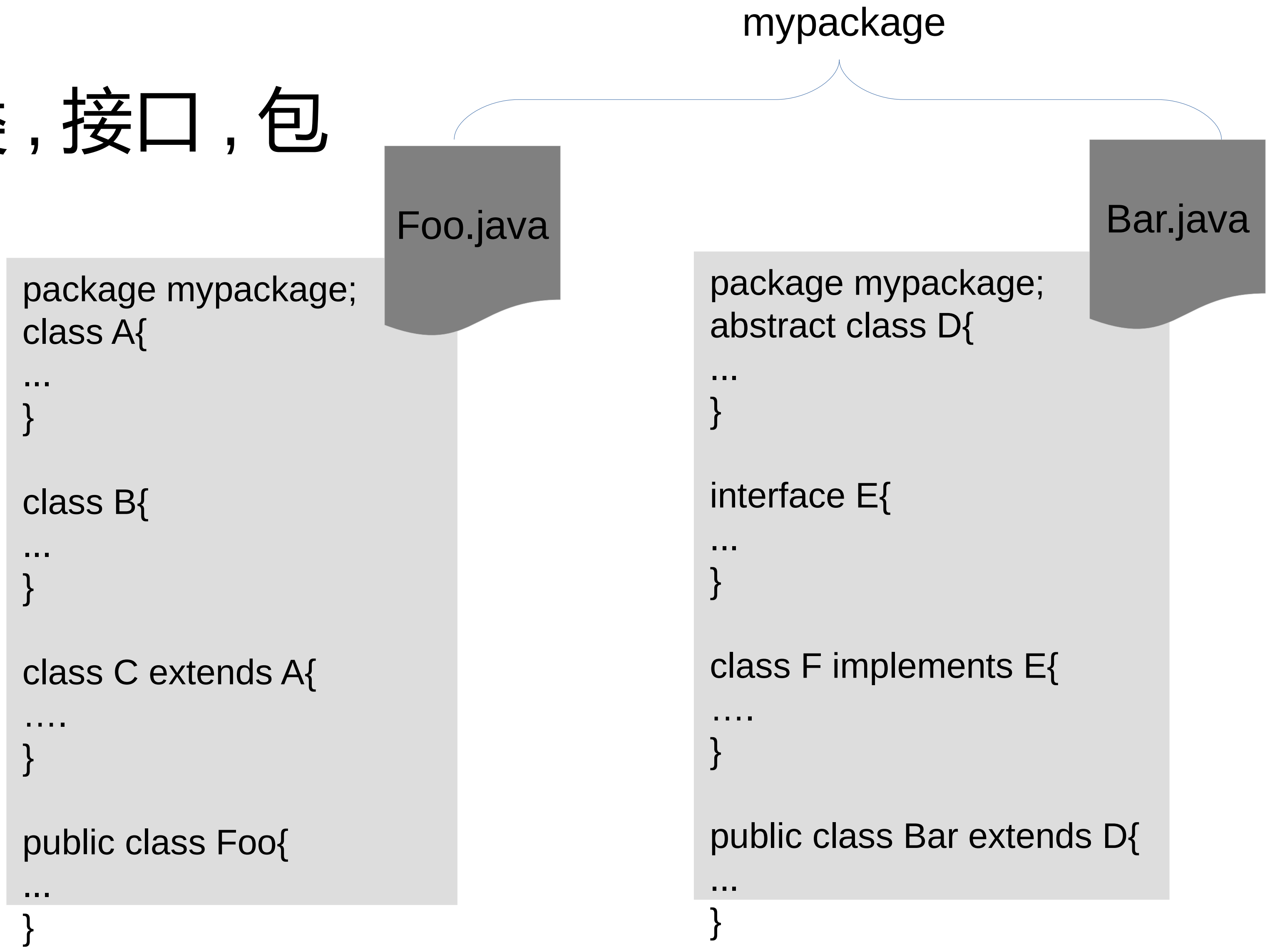
- 关于继承 / 抽象类 / 接口
  - 代码复用
  - Upcasting
  - 多态
  - 隔离方法的定义与实现

OOP with Java

- 内部类
  - 普通内部类
  - 匿名内部类
- 嵌套类
- 内部类的作用

内部类

- 复习：类，接口，包



mypackage

Foo.java

```
package mypackage;
class A{
...
}

class B{
...
}

class C extends A{
….
}

public class Foo{
...
}
```

Bar.java

```
package mypackage;
abstract class D{
...
}

interface E{
...
}

class F implements E{
….
}

public class Bar extends D{
...
}
```

内部类

- 内部类 (Inner class)
  - 定义在一个类的内部
  - 与组合不同

Inner class

```
class Outer{
    …
    class Inner{
        ...
    }
    ...
}
```

Composition

```
class Outer{
    …
    Inner in = new Inner();
    ...
}
class Inner{
    ….
}
```

```
public class Parcel{
    class Contents{
        private int i = 11;
        public int value() {return i;}
    }
    class Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }

    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        p.ship(“Tasmania”);
    }
}
```

- 内部类的作用
  - 帮助隐藏实现细节
  - 代码组织
  - …

```
public class Parcel{
    class Contents{
        private int i = 11;
        public int value() {return i;}
    }
    class Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }
    public Destination to(String s){
        return new Destination(s);
    }
    public Contents contents(){
        return new Contents();
    }
    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Parcel.Destination d = p.to(“Tasmania”);
        Parcel.Contents c = p.contents();
    }
```

- 内部类的引用
  - 在外部类中： InnerClassName
  - 在其他类中： OutClassName.InnerClassName

内部类

- 内部类与外部类的关系
  - 内部类的对象隐含了一个引用，指向包含它的外部类对象
  - 内部类对象能够访问该外部对象的所有成员 / 方法
    - public, private, protected

Inner class

```
class Outer{
    …
    class Inner{
        ...
    }
    ...
}
```

Composition

```
class Outer{
    …
    Inner in = new Inner();
    ...
}
class Inner{
    ….
}
```

内部类

- 内部类和外部类的关系
  - 内部类的对象隐含了一个引用，指向包含它的外部类对象
  - 如何在内部类中访问外部类对象的引用？
    - OuterClassName.this
  - 如何创建内部类的对象
    - 非静态环境中中： 直接创建
    - 静态环境中 : OuterClassObject.new

内部类

- 如何在内部类中访问外部类对象的引用？
  - OuterClassName.this

```
public class Outer{

    void f() { System.out.println(“Outer.f()”);}

    class Inner{
        public Outer g() {return Ourter.this;}
    }

    public Inner inner() { return new Inner(); }

    public static void main(String []args){
        Outer o = new Outer();
        Outer.Inner i = o.inner();
        i.g().f();
    }
}
```

内部类

- 如何创建内部类的对象
  - 在外部类的方法中：直接创建

```
public class Outer{
    void f() { System.out.println(“Outer.f()”);}
    class Inner{
        public Outer g() {return Ourter.this;}
    }
    public Inner inner() { return new Inner(); }

    public static void main(String []args){
        Outer o = new Outer();
        Outer.Inner i = o.inner();
        i.g().f();
    }
}
```

内部类

- 如何创建内部类的对象
  - 其他地方：OuterClassObject.new
    - 内部类的对象隐含了一个引用，指向包含它的外部类对象
    - 创建内部类对象前，需要有包含它的外部类对象

```
public class Outer{
    class Inner{  }

    public static void main(String []args){
        Outer o = new Outer();
        Outer.Inner i = o.new Inner();
    }
}
```

```
public class Parcel{
    class Contents{
        private int i = 11;
        public int value() {return i;}
    }
    class Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }

    public Destination to(String s){
        return new Destination(s);
    }
    public Contents contents(){
        return new Contents();
    }

    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Parcel.Destination d = p.to(“T”);
        Parcel.Contents c = p.contents();
    }
}
```

```
public class Parcel{
    class Contents{
        private int i = 11;
        public int value() {return i;}
    }
    class Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }

    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Parcel.Destination d = p.new Destination(“T”);
        Parcel.Contents c = p.new Contents();
    }
}
```

```
class Parcel{
    private class PContents implements Contents{
        private int i = 11;
        public int value() {return i;}
    }
    private class PDestination implements Destination{
        private String label;
        PDestination(String r) {label = r;}
        String readLabel() { return label;}
    }
    public Destination to(String s){
        return new PDestination(s);
    }
    public Contents contents(){
        return new PContents();
    }
    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
}
public class ParcelTest{
    public static void main(String []args){
        Parcel p = new Parcel();
        // Destination d = p.new PDestination(“T”);
        // Contents c = p.new PContents();
        System.out.println(d.readLabel());
    }
}
```

```
public interface Destination{
    String readLabel();
}

public interface Contents{
    int value();
}
```

1. private 的内部类可以完全隐藏内部类
2. 外界仅知道接口，并不知道内部类的存在
 - 此时为内部类增添新的方法没有意义 .

```
// Destination d = p.new PDestination(“T”);
// Contents c = p.new PContents();
// compile error:
//    - private inner class can not be accessed
```

内部类

- **内部类** - 外部类，　子类 - 父类
  - 一个子类对象包含的父类对象仅绑定到该子类对象 (super())
  - 一个内部类对象绑定的外部类对象可以绑定到多个不同的内部类对象 (p.new)
- 内部类与外部类之间没有 upcasting 关系，但对象间关系更灵活
- 子类与父类之间有类型关系，但对象间绑定关系固定

内部类

- 其他类型的内部类
  - 定义在方法中的内部类
  - 定义在任意作用域中的内部类

内部类

- 定义在方法中的内部类
  - 也称为 local inner class
  - 在方法之外，该类不可见

```
public class Parcel{
    public Destination to(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String r) {label = r;}
            public String readLabel() { return label;}
        }
        return new PDestination(s);
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Destination d = p.to(“T”);
    }
}
```

```
public interface Destination{
    String readLabel();
}
```

内部类

- 定义在任意作用域中的内部类
  - 在该作用域之外不可见

```
public class Parcel{
    public Destination to(String s) {
        if (s != null) {
            class PDestination implements Destination {
                private String label;
                private PDestination(String r) {label = r;}
                public String readLabel() { return label;}
            }
            return new PDestination(s);
        }
        return null;
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Destination d = p.to(“T”);
    }
}
```

```
public interface Destination{
    String readLabel();
}
```

```
public class Sequence{
    private Object[] items;
    private int next = 0;
    public Sequence (int size) {items = new Object[size];}
    public void add(Object x){
        if (next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector{
        private int i = 0;
        public boolean end() {return i == items.length;}
        public Object current () {return items[i];}
        public void next() { if(i < items.length) i++; }
    }
    public Selector selector(){
        return new SequenceSelector(s);
    }
    public static void main(String []args){
        Sequence seq = new Sequence(10);
        for (int i = 0; i < 10; ++i)
            seq.add(Integer.toString(i));
        Selector s = seq.selector();
        while(!s.end()) {
            System.out.println(s.current() + " ");
            s.next();
        }
    }
}
```

```
interface Selector{
    boolean end();
    Object current();
    void next();
}
```

1. Sequence 类包含内部类 SequenceSelector
2. 内部类实现接口 Selector
3. 内部类能访问 Sequence 的 private 成员
4. 内部类为 private
5. 内部类的对象隐藏包含一个外部类对象的引用
   - 由编译器自动完成

6. 复习 : upcasting: Object / selector()
7. 复习 : 还有哪些隐藏引用 ?

内部类

- 总结
  - 定义在类的内部
  - 隐含指向一个指向外部类对象的引用
  - 作用：帮助隐藏细节

匿名类

- 匿名内部类(匿名类)
  - 没有名字的内部类
  - 必须继承某个类，或实现某个接口
  - 更进一步的隐藏：类名

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            // anonymous inner class definition
            private int i = 11;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

```
public interface Contents{
    int value();
}
```

“ 创建一个实现 Contents 的匿名类”

语法解释

1. “;” 为 return 语句的分号
2. 在 return 语句中定义匿名类
   - 实现 Contents 接口
   - 花括号内部
3. 创建一个该匿名类的对象
   - new Content () { }

匿名类

- 匿名类

```
public class Parcel{

  public Contents contents(){
    return new Contents() {
      // anonymous inner class definition
      private int i = 11;
      public int value() {return i;}
    };
  }


  public static void main(String []args){
    Parcel p = new Parcel();
    Contents c = p.contents();
  }
}
```

```
public class Parcel{

  class PContents implements Contents{
     private int i = 11;
     public int value() {return i;}
   }
   public Contents contents(){
     return new PContents() ;
   }


   public static void main(String []args){
     Parcel p = new Parcel();
     Contents c = p.contents();
   }
}
```

匿名类

- 匿名类
  - 没有名字
  - 没有构造函数
  - 同时定义和创建
  - 必须继承另一个类或者实现一个接口

匿名类

- 匿名类**必须**继承另一个类 / 实现一个接口
  - 父类构造函数带有参数？

```
public class Parcel{

    public Wrapping wrapping(int x){
        return new Wrapping(x) {
            public int value() {
                return super.value() * 47;
            }
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Wrapping w = p.wrapping(10);
    }
}
```

```
public class Wrapping{
    private int i;
    public Wrapping(int i) { i=x; }
    public int value() { return i; }
}
```

匿名类

- 匿名类
  - 使用外部变量对匿名类数据成员初始化
    - 外部变量需要 final

```
public class Parcel{

    public Contents contents(final int v){
        return new Contents() {
            private int i = v;
            public int value() {return i;}
        };
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents(13);
    }
}
```

```
public interface Contents{
    int value();
}
```

匿名类

- 匿名类没有构造函数
  - Instance initialization

```
public class Parcel{

    public Contents contents(){
        return new Contents() {
            private int i;
            { // instance initialization
                System.out.println(“Instance Initialization”);
                i = 11;
            }
            public int value() {return i;}
        };
    }
    public static void main(String []args){
        Parcel p = new Parcel();
        Contents c = p.contents();
    }
}
```

```
public interface Contents{
    int value();
}
```

匿名类

- 应用：工厂模式
  - 更灵活的构造对象方式

```
interface Service {
    void method1();
    void method2();
}
```

```
class Impl1 implements Service {
    public void method1() {
       System.out.println(“Imp1.method1”);
    }
    public void method2() {
       System.out.println(“Imp1.method2”);
    }
}
```

```
class Impl2 implements Service {
    public void method1() {
       System.out.println(“Imp2.method1”);
    }
    public void method2() {
       System.out.println(“Imp2.method2”);
    }
}
```

```
public class TestService {
    public static void consume(Service s) {
        s.method1();
        s.method2();
    }
    public static void main(String []args){
        Service s1 = new Impl1();
        Service s2 = new Impl2();
        consume(s1);
        consume(s2);
    }
}
```

当构造对象 / 初始化比较繁琐时，
可以增加一层包装

```
interface Service {
    void method1();
    void method2();
}
```

```
class Impl1 implements Service {
    public void method1() {
       System.out.println(“Imp1.method1”);
    }
    public void method2() {
       System.out.println(“Imp1.method2”);
    }
}
```

```
class Impl2 implements Service {
    public void method1() {
       System.out.println(“Imp2.method1”);
    }
    public void method2() {
       System.out.println(“Imp2.method2”);
    }
}
```

```
interface ServiceFactory {
    Service getService();
}
```

```
class Impl1Factory implements ServiceFactory {
    public Service getService() {
        return new Impl1();
    }
}
```

```
class Impl2Factory implements ServiceFactory {
    public Service getService() {
        return new Impl2();
    }
}
```

```
public class TestService {
    public static void consume(ServiceFactory sf) {
        Service s = sf.getService();
        s.method1(); s.method2();
    }
    public static void main(String []args){
        ServiceFactory sf1 = new Impl1Factory();
        ServiceFactory sf2 = new Impl2Factory();
        consume(sf1);
        consume(sf2);
    }
}
```

```
interface Service {
    void method1();
    void method2();
}
```

```
interface ServiceFactory {
    Service getService();
}
```

```
class Impl1 implements Service {
    public void method1() {
       System.out.println(“Imp1.method1”);
    }
    public void method2() {
       System.out.println(“Imp1.method2”);
    }

    public static ServiceFactory factory =
        new ServiceFactory() {
           public getService() {
              return new Impl1();
        }
    };
```

```
class Impl2 implements Service {
    public void method1() {
       System.out.println(“Imp2.method1”);
    }
    public void method2() {
       System.out.println(“Imp2.method2”);
    }

    public static ServiceFactory factory =
        new ServiceFactory() {
           public getService() {
              return new Impl2();
        }
    };
```

```
public class TestService {
    public static void consume(ServiceFactory sf) {
        Service s = sf.getService();
        s.method1(); s.method2();
    }
    public static void main(String []args){
        consume(Impl1.factory);
        consume(Impl2.factory);
    }
}
```

匿名类

- 总结
  - 没有名字
  - 没有构造函数
  - 同时定义和创建
  - 必须继承另一个类或者实现一个接口

嵌套类

- 内部类
  - 内部类的对象隐含了一个引用，指向包含它的外部类对象
- 静态的内部类
  - 不需要外部类的对象即可创建
  - 也称为嵌套类 (nested class)

嵌套类

- 嵌套类
  - 不包含指向外部类对象的引用
  - 无法访问外部类的非静态成员

```
public class Parcel{
    private static class PContents implements Contents{
        private int i = 11;
        public int value() {return i;}
    }
    private static class PDestination implements Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }
    public static Destination to(String s){
        return new PDestination(s);
    }
    public static Contents contents(){
        return new PContents();
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Destination d = p.to(“T”);
        Contents c = p.contents();

        Destination d1 = to(“T”);    // without an object of Parcel
        Contents c1 = contents();  // without an object of Parcel
    }
}
```

```
public interface Destination{
    String readLabel();
}

public interface Contents{
    int value();
}
```

嵌套类

- 接口中的内部类
  - 接口：
    - 通常只有方法的说明，不含实现
    - 所有成员默认为 public static
    - **接口中的内部类**
      - 默认是静态内部类（即，嵌套类）
  - 接口中的内部类
    - 让接口重拾"重用"的功能

嵌套类

```
public interface ClassInInterface {
    void f();
    class Test implements ClassInInterface{
        public void f() {
            System.out.println(“hello”);
        }
        public static void main(String []args){
            new Test().f();
        }
    }
}
```

嵌套类

- 总结
  - 静态的内部类
  - 不包含指向外部类对象的引用
  - 接口中的内部类是嵌套类

内部类的作用

- 内部类的用途
  - 内部类通常继承一个类或者实现一个接口

```
public class Parcel{
    private class PContents implements Contents{
        private int i = 11;
        public int value() {return i;}
    }
    private class PDestination implements
Destination{
        private String label;
        Destination(String r) {label = r;}
        String readLabel() { return label;}
    }
    public Destination to(String s){
        return new PDestination(s);
    }
    public Contents contents(){
        return new PContents();
    }
    public void ship(String dest){
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }

    public static void main(String []args){
        Parcel p = new Parcel();
        Destination d = p.to(“T”);
        Contents c = p.contents();
    }
}
```

```
public interface Destination{
    String readLabel();
}

public interface Contents{
    int value();
}
```

问题：
为何不在原始类上直接实现该接口？

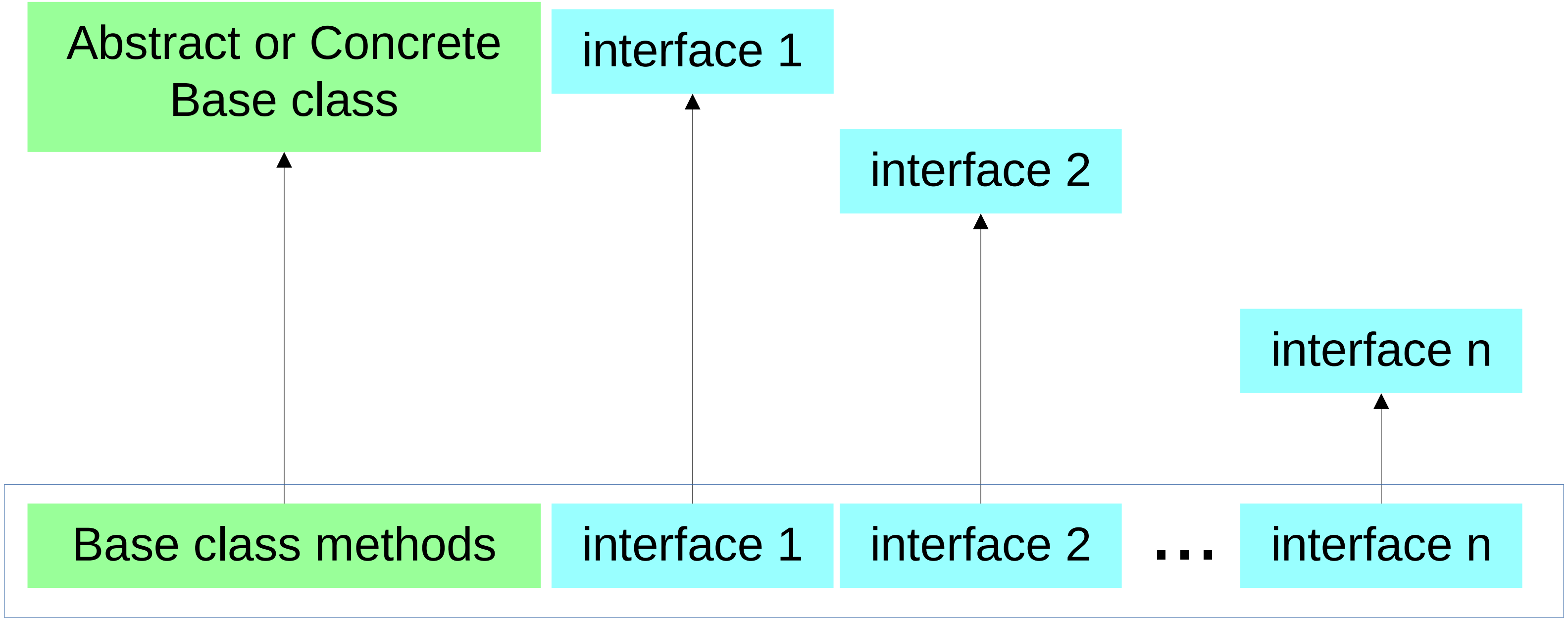
回答：
1. 如果可以，那么就做！
2. 有时不行
  - 外部类已经确定，无法修改
  - 内部类可以灵活的继承 / 实现其他接口

内部类的作用

- 为什么引入内部类
  - 重新考虑多继承

内部类的作用

- 多继承
  - 复习
    - 父类只能有一个普通类 / 抽象类



Methods of the derived class

内部类的作用

- 多继承
  - 可以通过多个内部类继承多个类 / 抽象类 / 接口

```
public interface A { }
public interface B { }
class X implements A, B {}

class Y implements A{
    B makeB() {
        return new B(){};
    }
}
```

```
interface A { }
interface B { }
class X implements A, B {}


class Y implements A{
    B makeB() {
        return new B(){};
    }
}

public class Test{
    static void takeA(A a) {}
    static void takeB(B b) {}
    public static void main(String []args){
        X x = new X();
        Y y = new Y();
        takeA(x); takeB(x);
        takeA(y); takeB(b.makeB());
    }
}
```

```
class A { }
abstract class B { }
// class X implements A, B {}
// won't compile

class Y extends A{
    B makeB() {
        return new B(){};
    }
}

public class Test{
    static void takeA(A a) {}
    static void takeB(B b) {}
    public static void main(String []args){
        Y y = new Y();
        takeA(y); takeB(b.makeB());
    }
}
```

内部类的作用

- 在类中使用内部类
  - 同一个内部类可以有多个实例，每个实例有不同的状态
  - 对同一接口，可以有不同的内部类实现
  - 创建内部类对象可以按需创建
  - 不必遵从 is-a 关系

内部类的作用

- 应用：事件驱动系统 (event-driven system)
  - 控制一组事件
  - 每个事件有准备时间，当准备妥当，状态转为 ready
  - 每个事件有方法 action()，表示事件的内容

```
public abstract class Event {
    private long eventTime;
    protected final long delayTime;
    public Event(long dt) {
        delayTime = dt;
        start();
    }
    public void start(){
        eventTime = System.nanoTime() + delayTime;
    }
    public boolean ready(){
        return System.nanoTime() >= eventTime
    }
    public abstract void action();
}
```

```
public  class Controller {
    private List<Event> eventList = new ArrayList<Event>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while (eventList.size()>0)
            for (Event e: eventList) {
                if (e.ready()){
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
                }
            }
    }
}
```

```
public class GreenhouseControls extends Controller {
    private boolean light = false;
    public LightOn extends Event {
        public LightOn(long dt) { super(dt); }
        public void action() { light = true;}
        public toString() {System.out.println(“Light on”);}
    }

    public LightOff extends Event {
        public LightOff(long dt) { super(dt); }
        public void action() { light = false;}
        public toString() {System.out.println(“Light off”);}
    }

    private boolean water = false
    public WaterOn extends Event {
        public LightOn(long dt) { super(dt); }
        public void action() { water = true;}
        public toString() {System.out.println(“Water on”);}
    }
    public WaterOff extends Event {
        public LightOff(long dt) { super(dt); }
        public void action() { water = false;}
        public toString() {System.out.println(“Wat
    }
}
```

```
public class Greenhouses {
    public static void main(String[] args){
        GreenhouseControls gc = new GreenhouseControls();
        gc.add(gc.new LightOn(200));
        gc.add(gc.new WaterOn(400));
        gc.add(gc.new WaterOff(600));
        gc.add(gc.new LightOff(800));
        gc.run();
    }
}
```

内部类的作用

- 总结
  - 可以通过多个内部类继承多个类 / 抽象类 / 接口