



第8章 多态

导入

- 封装
 - 隐藏细节, 释放接口
- 继承
 - 导出类可作为基类来处理
- 面向对象三大特征之三: 多态
- 持续的解耦合
 - 接口(做什么)
 - 实现(怎么做)
- 更高的可扩展性, 程序可以持续增长

8.1 再论向上转型



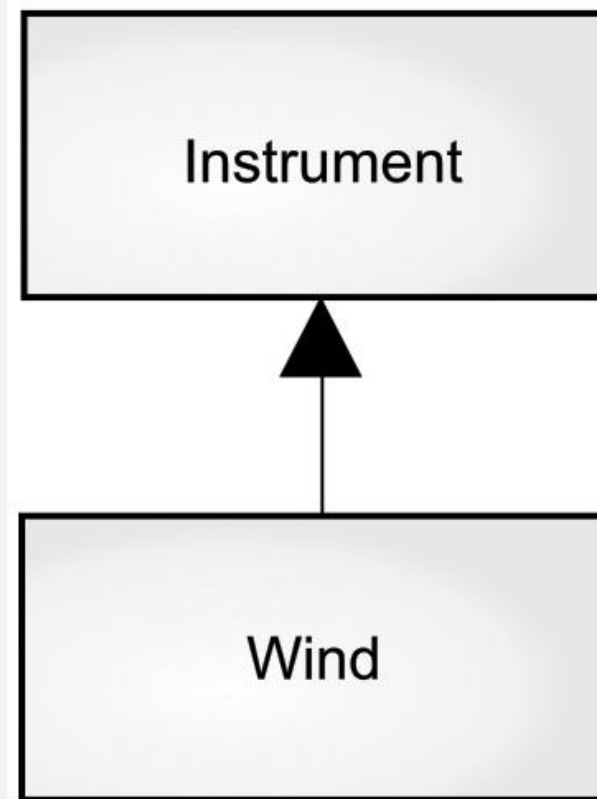
忘记类型



只与基类交互




练习1




```
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}
```


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

```
class Music{  
    public static void tune(Instrument i) {  
        i.play(Note.MIDDLE_C);  
    } 
```

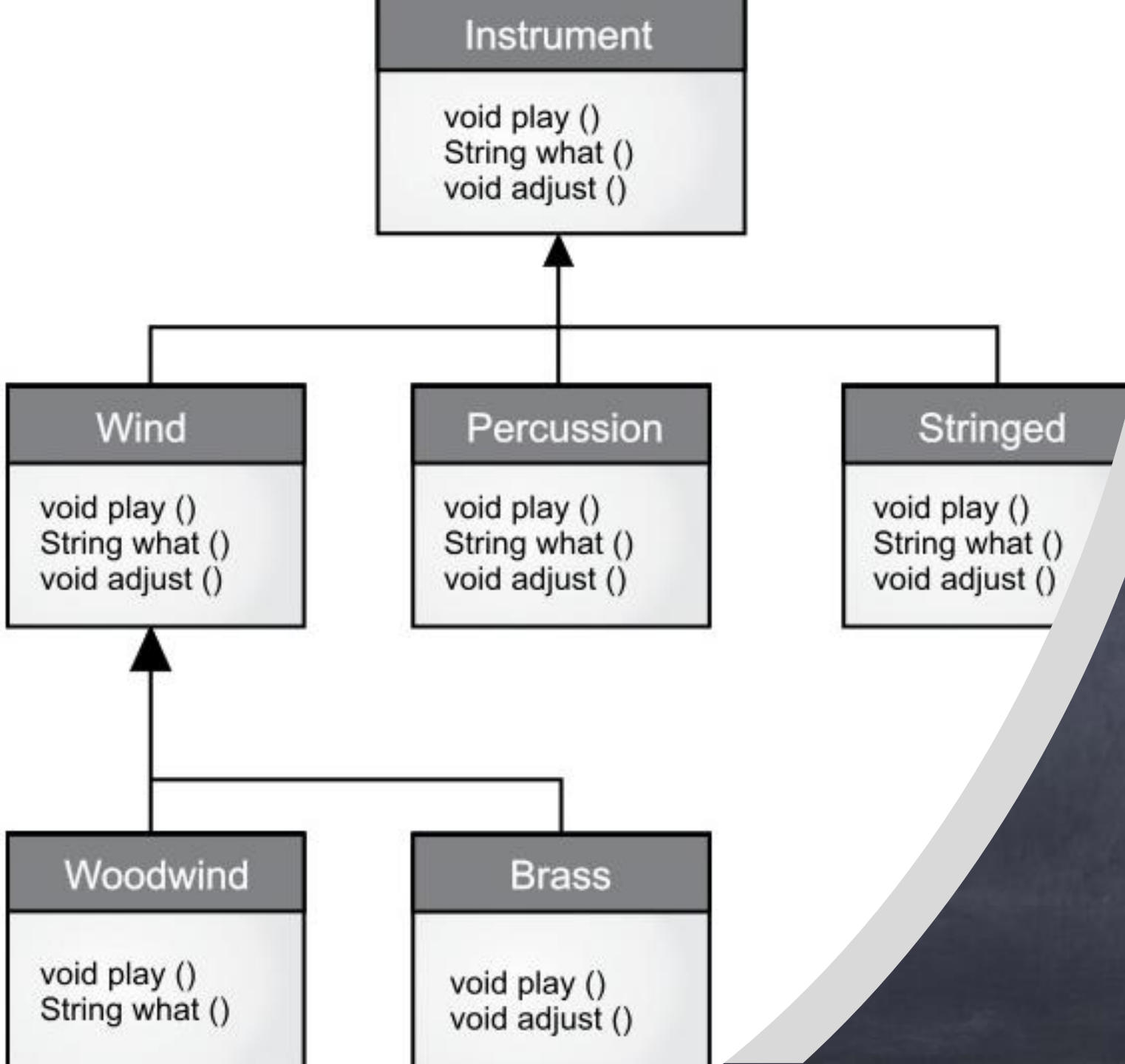
```
    public static void main(String[] args){
```

```
public class Music2 {   
    public static void tune(Wind i) {  
        i.play(Note.MIDDLE_C);  
    }  
}
```

```
public static void tune(Stringed i) {  
    i.play(Note.MIDDLE_C);  
}
```

```
public static void main(String[] args) {  
    tune(wind); // No upcasting   
    tune(violin);  
}
```

tune(wind); //向上转型



后期绑定与可扩展性



- 后期绑定 - 虚函数
- 增加新类不影响原有代码
- 练习 2-10

什么不是多态的?



私有方法

子类不可见



字段

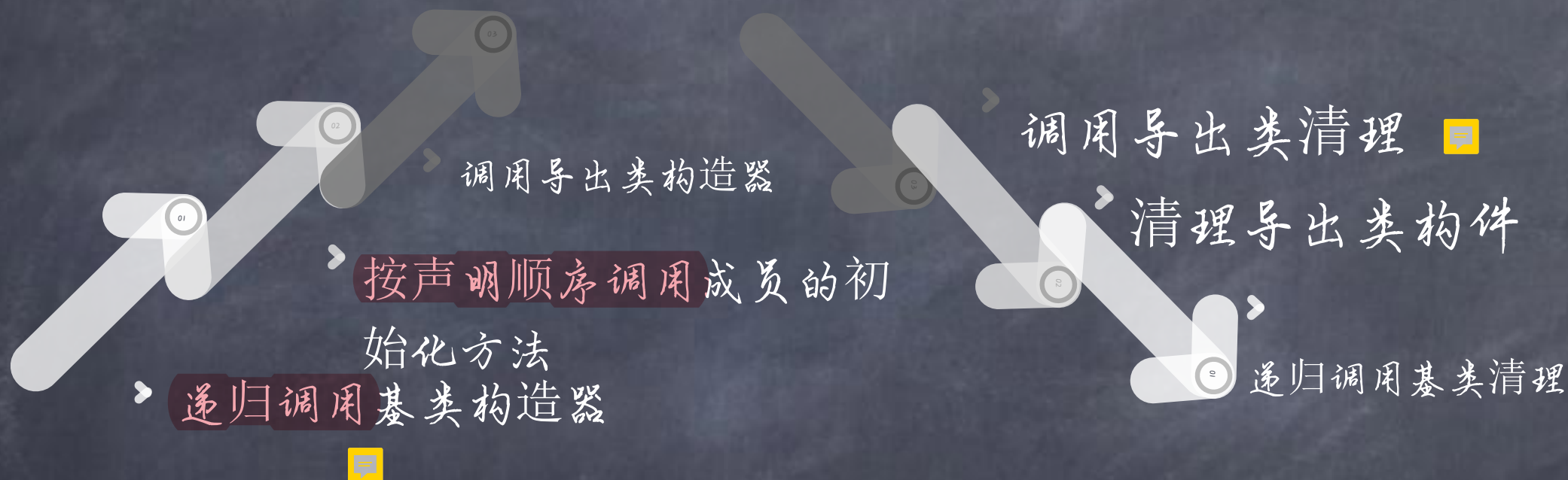
编译器解析,前期绑定



静态方法

与类绑定,不是对象

构造器与多态



- 构造器本身不是多态的
- 内部不要调用多态方法
- 私有方法不是多态的
- 练习₁₂

协变返回类型

```
class Mill {  
    Grain process() {  
        return new Grain();  
    }  
}
```

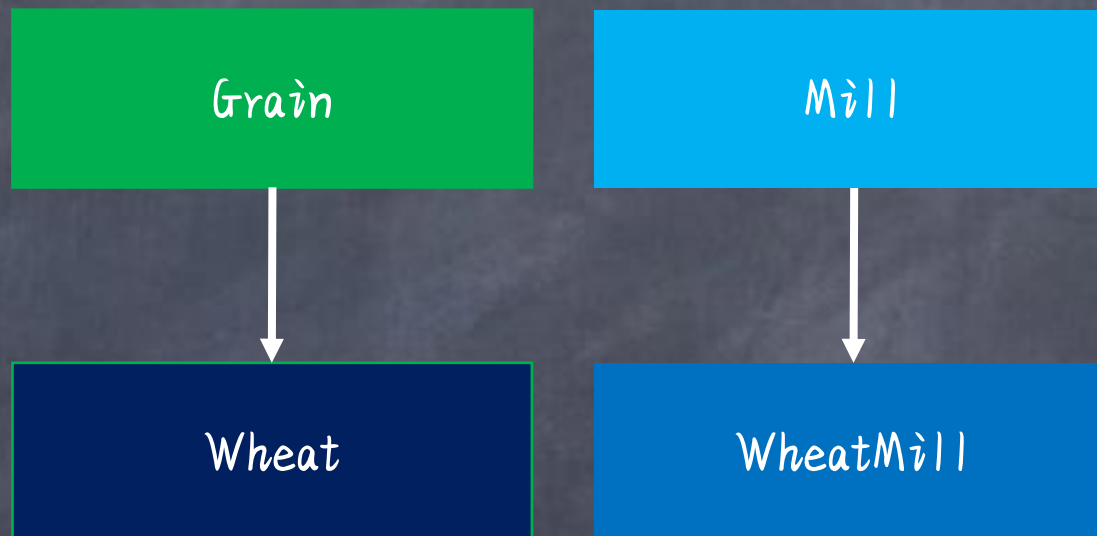
```
class WheatMill extends Mill {
```

```
    @Override
```

```
    Wheat process() { //返回类型是导出类型
```

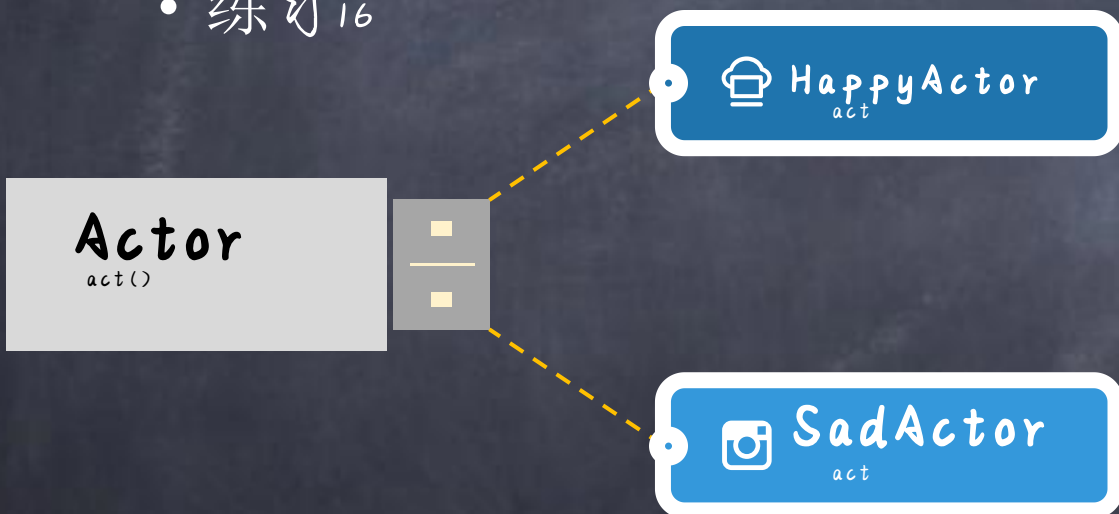
```
        return new Wheat();
```

```
}
```



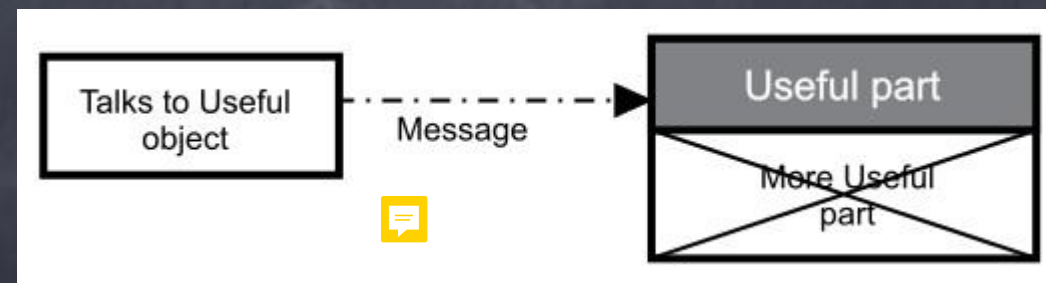
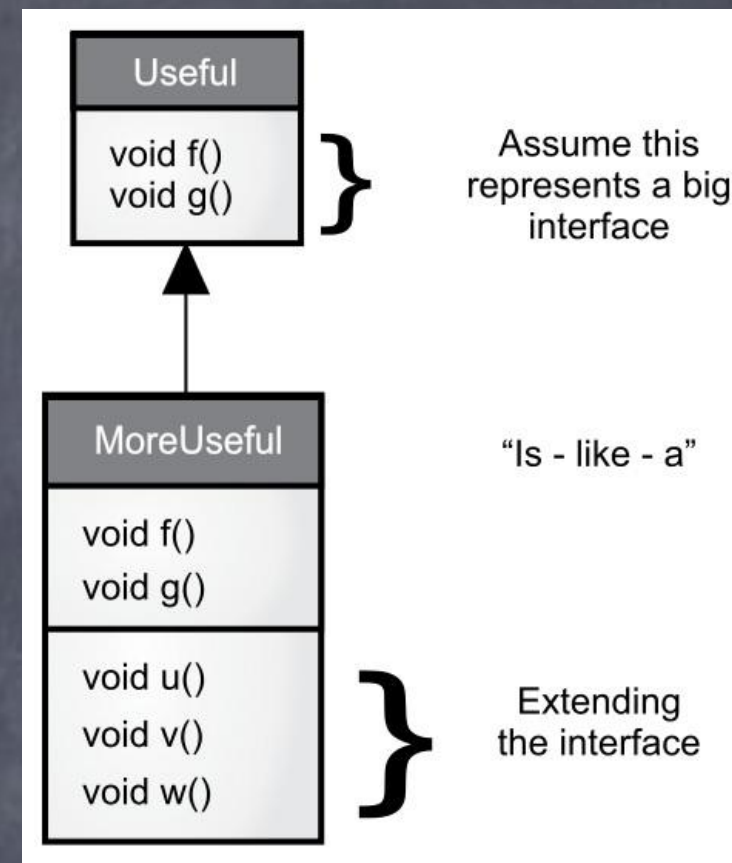
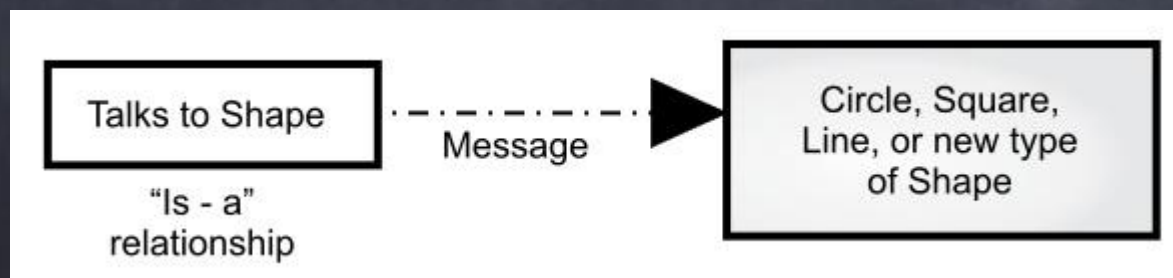
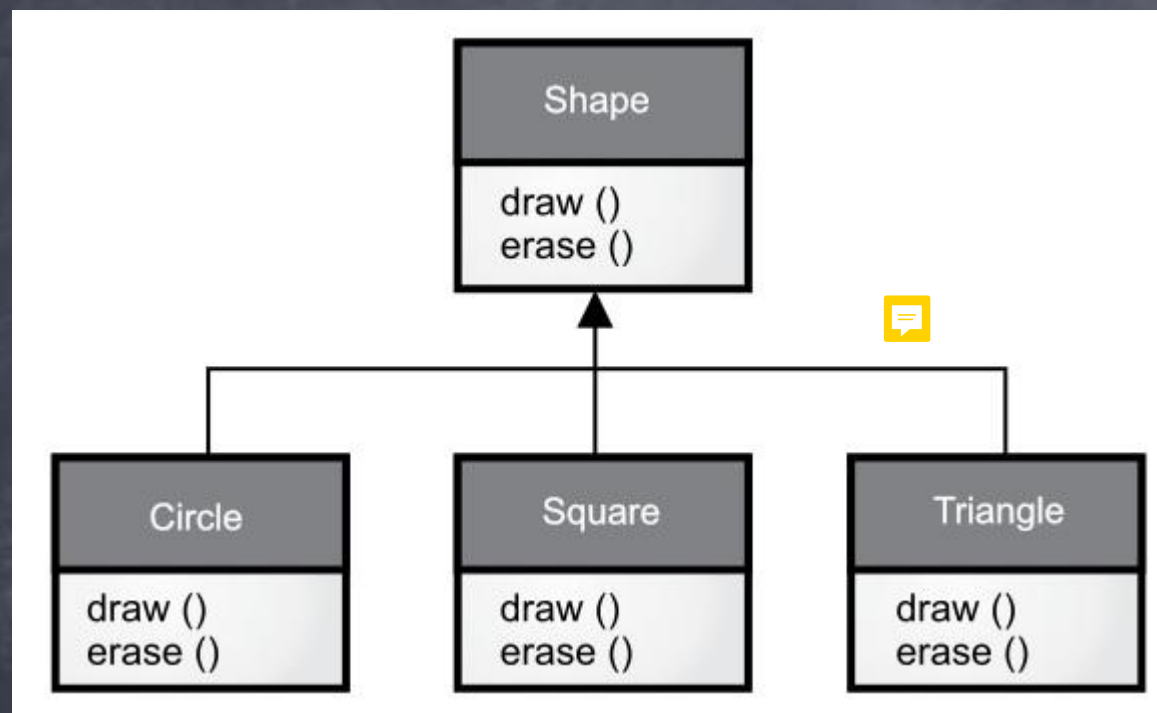
用继承进行设计

- 用继承表示行为差异
- 用组合灵活使用功能
- 练习16



```
class Stage {  
    private Actor actor = new HappyActor();  
    public void change() {  
        actor = new SadActor();  
    }  
    public void performPlay() {  
        actor.act();  
    }  
}
```

纯继承与扩展



向下转型与运行时类型信息

- 向上转型永远是安全的

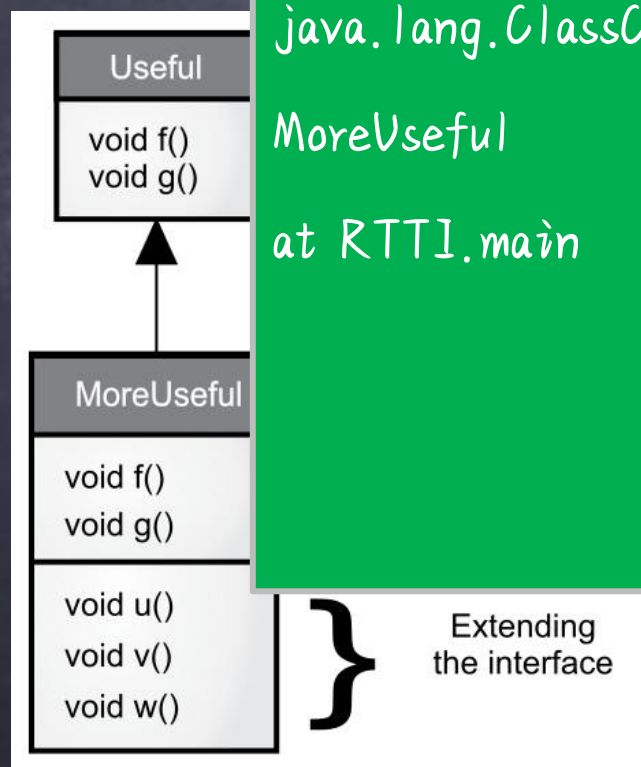
- 向下转

Exception in thread "main"

java.lang.ClassCastException: Useful cannot be cast to

MoreUseful

at RTTI.main



```
public class RTTI {
```

```
    public static void main(String[] args) {
```

```
        // ... some code ...
    }
}
```

```
}
```


课堂练习

练习1: (2) 创建一个**Cycle**类, 它具有子类**Unicycle**、**Bicycle**和**Tricycle**。演示每一个类型的实例都可以经由**ride()**方法向上转型为**Cycle**。

练习2: (1) 在几何图形的示例中添加**@Override**注解。

练习3: (1) 在基类**Shapes.java**中添加一个新方法, 用于打印一条消息, 但导出类中不要覆盖这个方法。请解释发生了什么。现在, 在其中一个导出类中覆盖该方法, 而在其他的导出类中不予覆盖, 观察又有什么发生。最后, 在所有的导出类中覆盖这个方法。

练习4: (2) 向**Shapes.java**中添加一个新的**Shape**类型, 并在**main()**方法中验证: 多态对新类型的作用是否与在旧类型中的一样。

练习5: (1) 以练习1为基础, 在**Cycle**中添加**wheels()**方法, 它将返回轮子的数量。修改**ride()**方法, 让它调用**wheels()**方法, 并验证多态起作用了。

练习6: (1) 修改**Music3.java**, 使**what()**方法成为根**Object**的**toString()**方法。试用**System.out.println()**方法打印**Instrument**对象 (不用向上转型)。

练习7: (2) 向**Music3.java**添加一个新的类型**Instrument**, 并验证多态性是否作用于所添加的新类型。

练习8: (2) 修改**Music3.java**, 使其可以像**Shapes.java**中的方式那样随机创建**Instrument**对象。

练习9: (3) 创建**Rodent** (啮齿动物): **Mouse** (老鼠), **Gerbil** (鼯鼠), **Hamster** (大颊鼠), 等等这样一个的继承层次结构。在基类中, 提供对所有的**Rodent**都通用的方法, 在导出类中, 根据特定的**Rodent**类型覆盖这些方法, 以便它们执行不同的行为。创建一个**Rodent**数组, 填充不同的**Rodent**类型, 然后调用基类方法, 观察发生什么情况。

练习10: (3) 创建一个包含两个方法的基类。在第一个方法中可以调用第二个方法。然后产生一个继承自该基类的导出类, 且覆盖基类中的第二个方法。为该导出类创建一个对象, 将它向上转型到基类型并调用第一个方法, 解释发生的情况。

课堂练习

练习16: (3) 遵循Transmogrify.java这个例子，创建一个Starship类，包含一个AlertStatus引用，此引用可以指示三种不同的状态。纳入一些可以改变这些状态的方法。

练习17: (2) 使用练习1中的Cycle的层次结构，在Unicycle和Bicycle中添加balance()方法，而Tricycle中不添加。创建所有这三种类型的实例，并将它们向上转型为Cycle数组。在该数组的每一个元素上都尝试调用balance()，并观察结果。然后将它们向下转型，再次调用balance()，并观察将所产生什么。

练习12: (3) 修改练习9，使其能够演示基类和导出类的初始化顺序。然后向基类和导出类中添加成员对象，并说明构建期间初始化发生的顺序。

作业

- 提交练习₁₀



提问