OOP with Java

Yuanbin Wu
cs@ecnu

OOP with Java

- 通知
  - Project 4: 11 月 3 日晚 9 点

- 复习
  - Upcasting
    - 同一基类的不同子类可以被视为同一类型(基类)
    - 放宽类型一致性
    - 简化接口

```
class A{ … }
class B{ … }
A a = new A();
B b = new B();

// A a = new B(); compile error
```

```
class A{ … }
class B extends A{ … }
A a = new A();
B b = new B();

A a = new B(); // upcasting
```

- 复习
  - 多态

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

```
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);
    }
}
```

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

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

多态 (Polymorphism)

参数 Instrument i 可以代表不同的子类，并能正确调用它们的方法 (即，有多种表现形态)

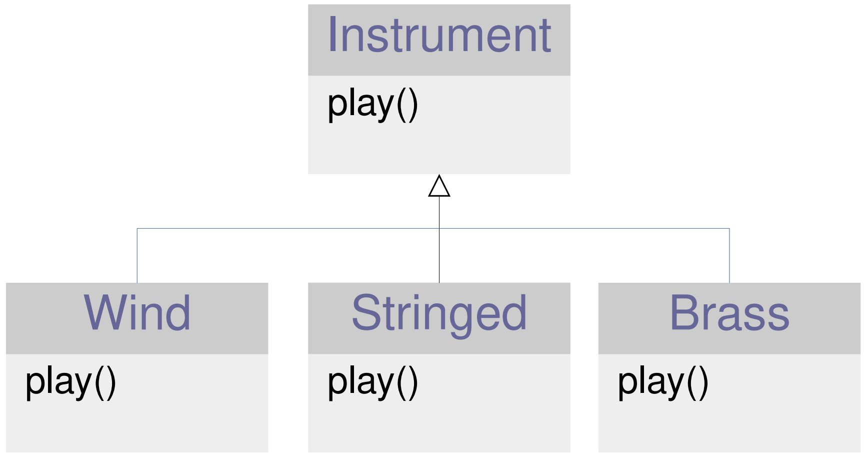
- 复习
  - 静态绑定
    - 函数的调用在编译后便确定，也称为 early binding
    - 优点：快速，易于 debug，缺点：接口繁琐
  - 动态绑定
    - 函数的调用在运行时才能确定 也称 late binding
    - 优点：接口简洁 缺点：函数调用需要额外开销
  - Java 中的所有方法都采用动态绑定，除了
    - final
    - Static
  - 数据成员不动态绑定

OOP with Java

- 抽象类
- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

抽象类

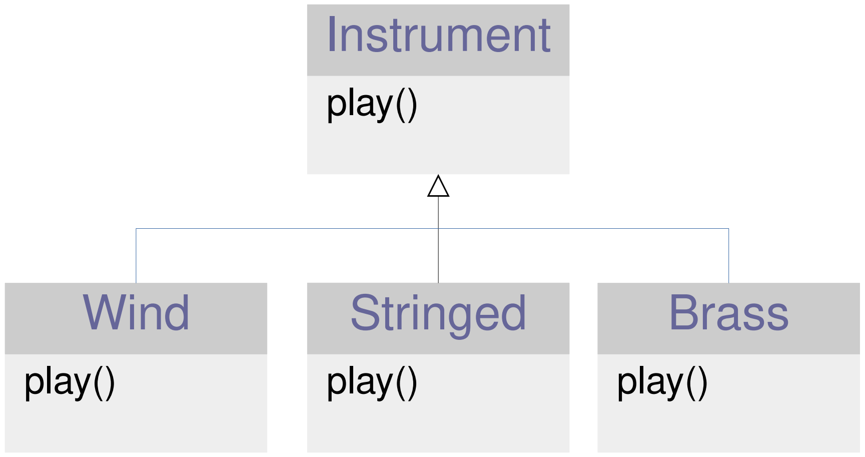
- 父类的方法
  - Instrument: play()
- 子类重写父类的方法
  - Wind: play()
  - Stringed: play()
  - Brass: play()
- 可扩展性
  - 用户程序仅知父类方法
  - 子类修改不会影响用户程序



```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute);
        tune(violin);
        tune(frenchHorn);
    }
}
```

抽象类

- 如果所有子类都将重写该方法
  - Instrument: play() 是否还必要？
- 是否有机制：
  - 在父类中不指定该方法的具体实现
  - 禁止调用父类的该方法



抽象类

- 抽象方法 (abstract method)
  - 仅提供方法的名称，参数和返回值
  - 没有具体实现
  - 使用 abstract 关键字

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

普通方法

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

抽象方法

抽象类

- 抽象类 (abstract class)
  - 包含抽象方法的类称为抽象类

抽象类

- 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);
    }
}
```

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

抽象类

- 是否能直接创建抽象类的对象？
  - 否
  - 抽象类是不完整的类
  - 其中的抽象方法需要在子类补充完整 (重写) 后才有意义

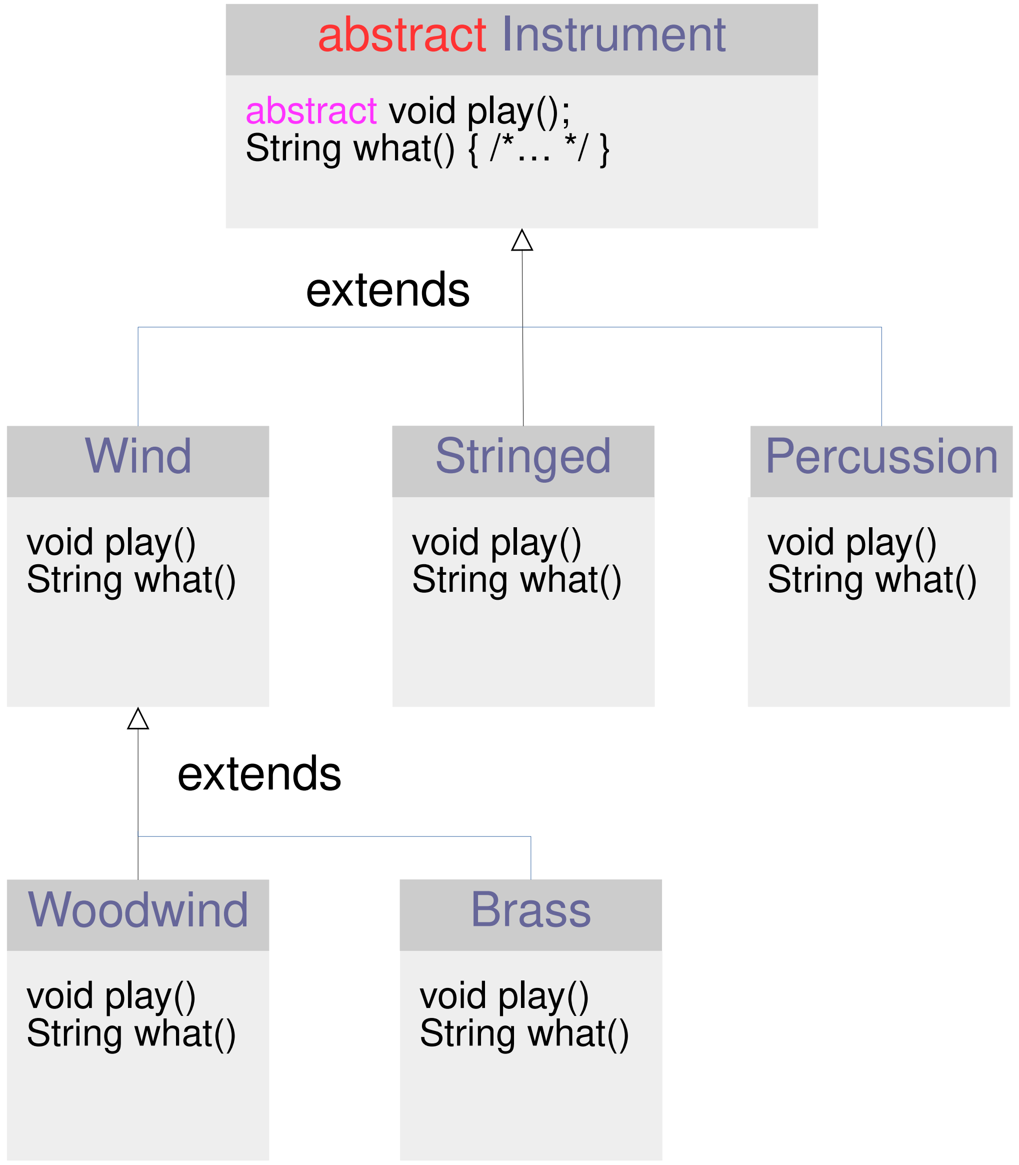
抽象类

- 无法直接创建该类的对象

```
abstract class Instrument {
    public abstract void play(int note) ;
}
class Test {
    public static void main(String []args){
        // Instrument in = new Instrument();
        // compile error: can not create instances of an  abstract class
    }
}
```

super 关键字？

抽象类


abstract Instrument
abstract void play();
String what() { /*… */ }
extends
Wind
void play()
String what()
Stringed
void play()
String what()
Percussion
void play()
String what()
extends
Woodwind
void play()
String what()
Brass
void play()
String what()

```
abstract class Instrument {
    public abstract void play(int note) ;
    public String what() {return “Instrument”;}
}
```

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

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

```
abstract class Wind extends Instrument {
   public abstract void play(int note) ;
   public String what() {return “Wind”;}
}
```

- 若子类没有重写父类中的抽象方法，子类仍为抽象类

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
         Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
         tune(flute);
         tune(violin);
         tune(frenchHorn);
    }
}
```

```
class Woodwind extends Wind {
    public void play(int note) {
        System.out.println(“Woodwind.play()” + n);
    }
   public String what() {return “Woodwind”;}
}
```

```
class Brass extends Wind {
    public void play(int note) {
        System.out.println(“Brass.play()” + n);
    }
   public String what() {return “Brass”;}
}
```

抽象类

- 总结
  - 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**直接**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

接口

- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

接口

- 抽象类
  - 抽象方法
  - 普通方法

```
abstract class Instrument {
    public abstract void play(int note) ;
    public String what() {return “Instrument”;}
}
```

接口

- 接口 (Interface)
  - “ 所有方法都是抽象方法”
  - 只有方法的名称，参数和返回值
  - 没有方法的实现

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

≈

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

接口

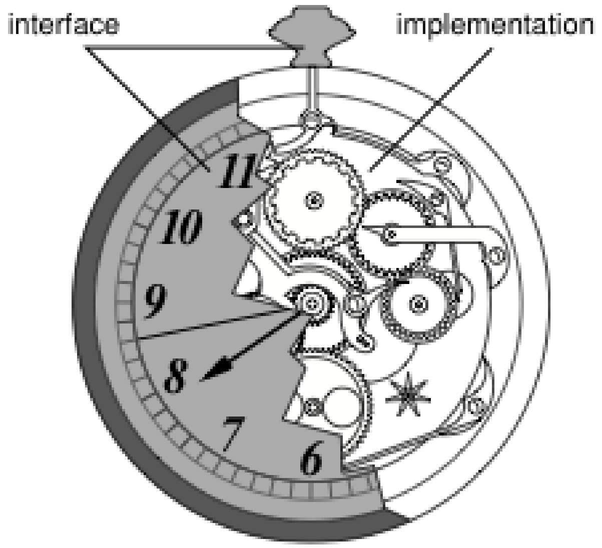
- 继承
  - 重用 (class reusing)
    - 子类重用父类的方法 / 数据
  - upcasting 和多态
    - 父类出现之处可用子类代替
    - 能够调用正确的子类方法 (动态绑定)
- 接口
  - 没有代码重用，仅仅保留 upcasting 和多态

接口

- 接口
  - 所有实现该接口的类都具有接口提供的方法
  - 任何使用该接口类型的方法，都可以使用他的任何一种实现
  - 某种协议 (protocol)

接口

- 接口的实现
  - 接口：方法长什么样？
  - 实现：方法具体怎样工作？



接口

- 接口的实现

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

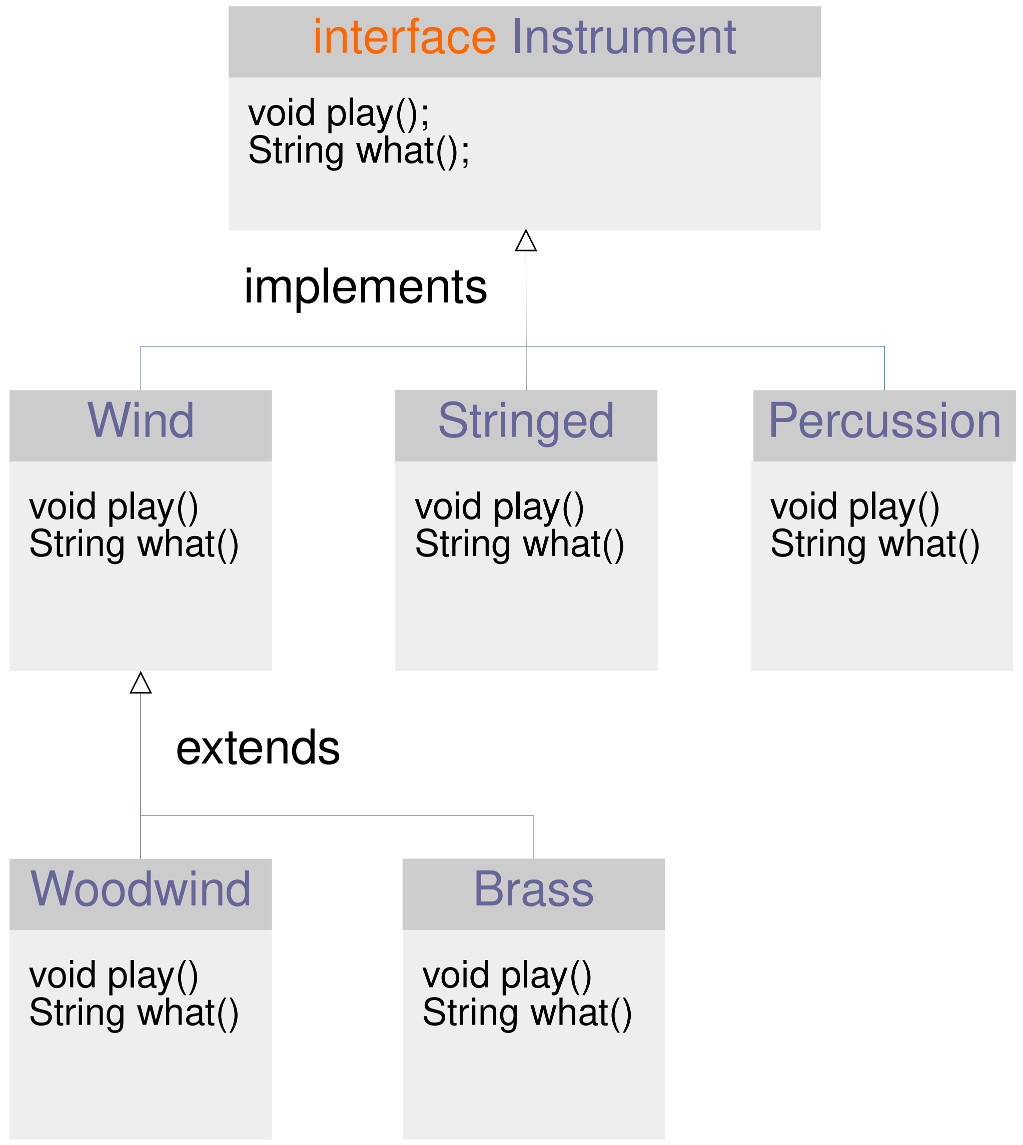
```
class Stringed extends Instrument {
    public void play(int note) {
        System.out.println(“Stringed.play()” + n);
    }
   public String what() {return “Stringed”;}
}
```

```
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”;}
}
```

继承 :
1. extends 关键字
2. 父类 , 子类关系
3. class, extends

接口 :
1. implements 关键字
2. 接口 , 实现关系
3. interface, implements


interface Instrument
void play();
String what();
implements
Wind
void play()
String what()
Stringed
void play()
String what()
Percussion
void play()
String what()
extends
Woodwind
void play()
String what()
Brass
void play()
String what()

```
interface class 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";}
}
```

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

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

```
public class Music {
    public static void tune(Instrument i) {
        i.play();
    }
    public static void main(String []args){
        Wind flute = new Wind();
         Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
         tune(flute);
         tune(violin);
         tune(frenchHorn);
    }
}
```

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

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

- 普通类，抽象类，接口

接口

- 接口
  - 所有方法默认为 public

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

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

接口

- 接口
  - 所有数据默认为 final static
  - 定义常量

```
interface Week {
    int MONDAY = 1;
    int TUESDAY = 2;
    int WEDNESDAY = 3;
    int THURSDAY = 4;
    int FRIDAY = 5;
    int SATURDAY = 6;
    int SUNDAY = 7;
}
```

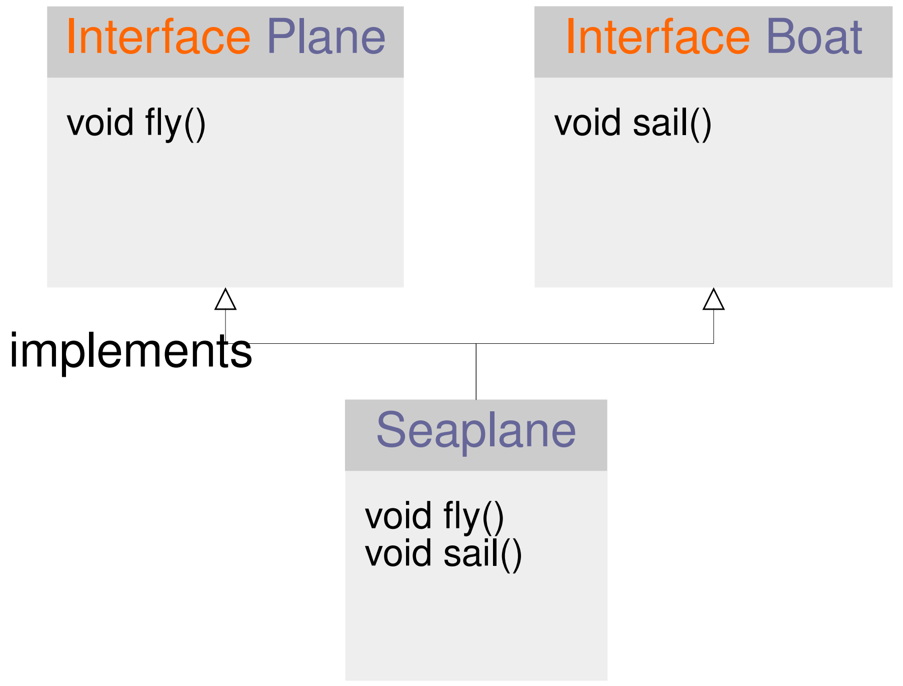
```
class Week {
    public static final int MONDAY = 1;
    public static final int TUESDAY = 2;
    public static final int WEDNESDAY = 3;
    public static final int THURSDAY = 4;
    public static final int FRIDAY = 5;
    public static final int SATURDAY = 6;
    public static final int SUNDAY = 7;
}
```

接口

- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

接口

- 一个类实现多个接口



接口

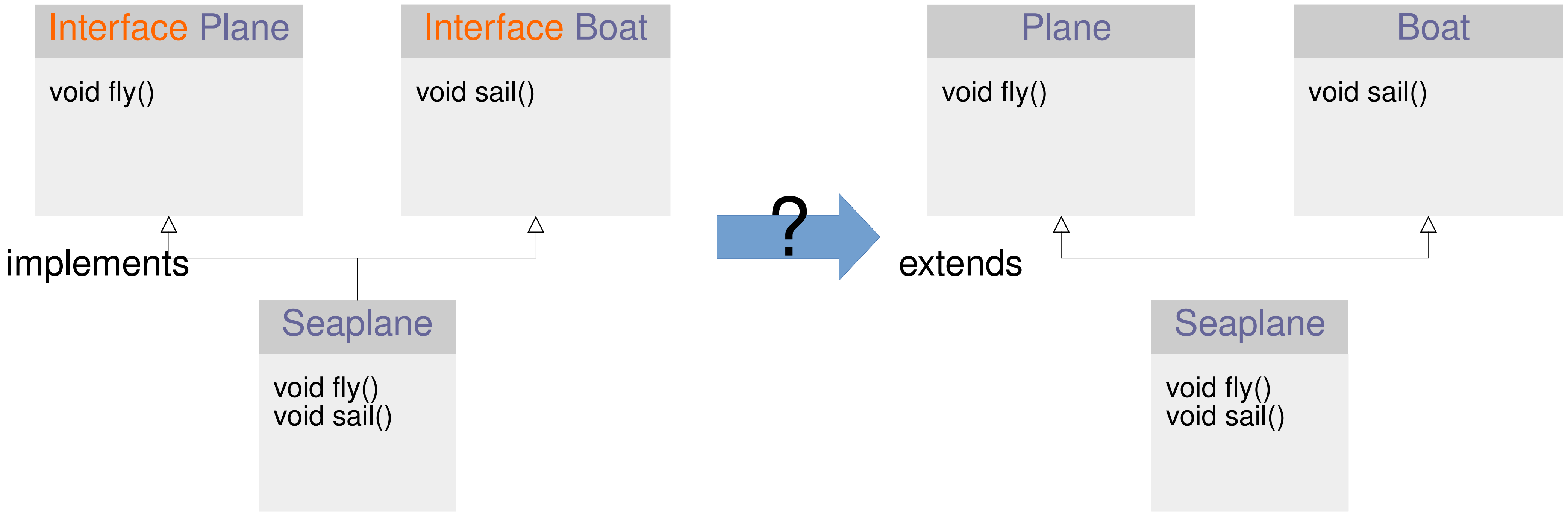
- 一个类实现多接口

```
interface Plane {
    void fly();
}
interface Boat {
    void sail();
}

class Seaplane implements Plane, Boat {
    public void fly(){
        System.out.println(“Fly!”);
    }
    public void sail(){
        System.out.println(“Sail!”);
    }
}
```

接口

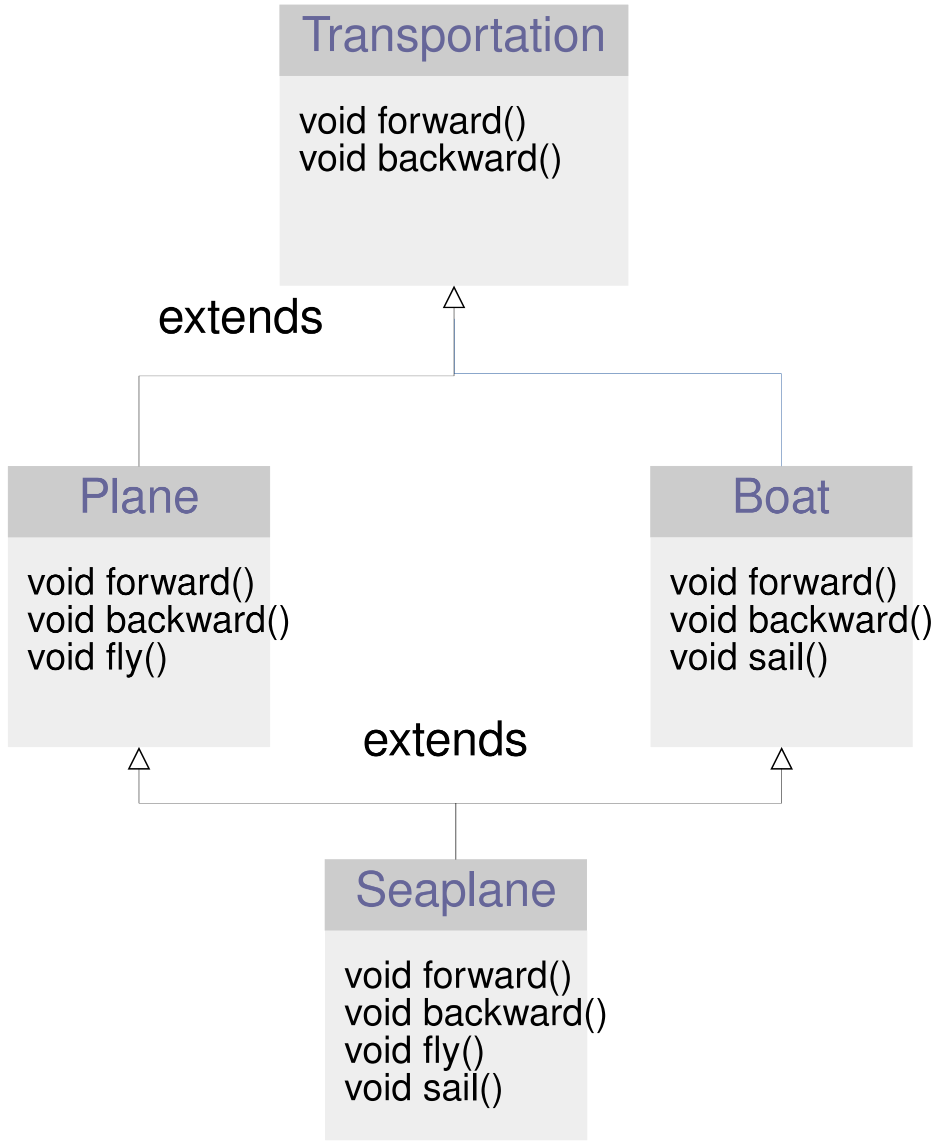
- 问题：
  - 如果将接口替换成普通类会如何？



接口

- 多继承问题
  - Diamond problem

```
Seaplane s = new Seaplane();
// s.forward() which one?
```



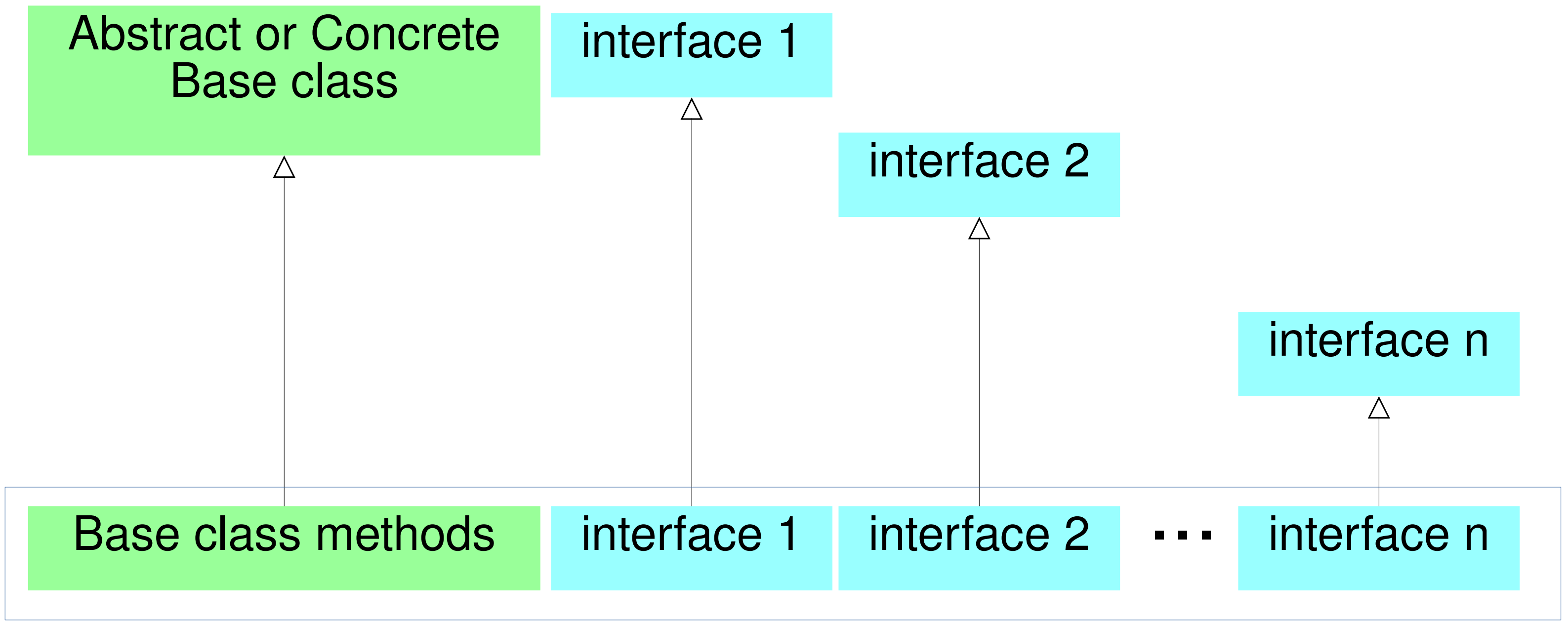
接口

- 多继承问题
  - 父类只能有一个普通类 / 抽象类

```
class A {
    …
}
class B {
    …
}

/* error
class C extends A, B {
    …
}
*/
```

接口



Methods of the derived class

```
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);
    }
}
```

1. 实现多个接口可以 upcast 到不同的类型
   - fight() ?
2. abstract class or interface?

接口

- 实现多个接口
  - 名字冲突

```
interface I1 {
    void f();
}
```

```
interface I2 {
    void f();
}
```

```
interface I3 {
    void f(int i);
}
```

```
interface I4 {
    int f();
}
```

```
class C1 implements I1, I2{
    public void f() {}
}
```

```
class C2 implements I1, I3{
    public void f() {}
    public void f(int i) {}
}
```

```
/* compile error: return type incompatible
class C2 implements I1, I4{
    public void f() {}
}
*/
```

接口

- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

接口

- 扩展接口

```
interface A {
    …
}
interface B extends A{
    …
}

interface D {
    …
}
interface D extends A, C{
    …
}
```

```
interface Monster {
    void menace();
}
```

```
interface DangerousMonster extends Monster{
    void destroy();
}
```

```
class DragonZilla implements DangerousMonster{
    public void menace() { }
    public void destroy() { }
}
```

```
interface Lethal {
    void kill();
}
```

```
interface Vampire extends DangerousMonster, Lethal{
    void drinkblood();
}
```

```
class VeryBadVampire implements Vampire{
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkblood() {}
}
```

```
public class HorrorShow {
    public static void u(Monster x) { x.menace();}
    public static void v(DangerousMonster x) {
        x.menace();
        x.destroy();
    }
    public static void w(Lethal x) { x.kill();}
    public static void main(String []args) {
        DangerousMonster m = DangerousZilla();
        u(m); v(m);
        Vampire a = VeryBadVampire();
        u(a); v(a); w(a);
    }
}
```

接口

- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

接口

- 接口适配器 (Adapter)
  - 方法 f, 参数类型为 Interface1
  - 假设类 A 已存在，它没有实现 Interface1 接口
  - 希望方法 f() 能处理类 A 的对象
    - 复用方法 f() 的代码

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

```
class Bird implements CanFly{
    public void fly() { }
}
```

```
class Insect implements CanFly{
    public void fly() { }
}
```

```
class Person {
    public void walk(){}
    public void buyTicket(){}
    public void takeFlight(){}
}
```

?

```
class Adventure {
    public static void travel(CanFly c) {
        c.fly();
    }
    public static void main(String []args){
        Bird b = new Bird();
        Insect ins = new Insect();
        travel(b); travel(ins);
    }
}
```

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

```
class Person {
    public void walk(){}
    public void buyTicket(){}
    public void takeFlight(){}
}
```

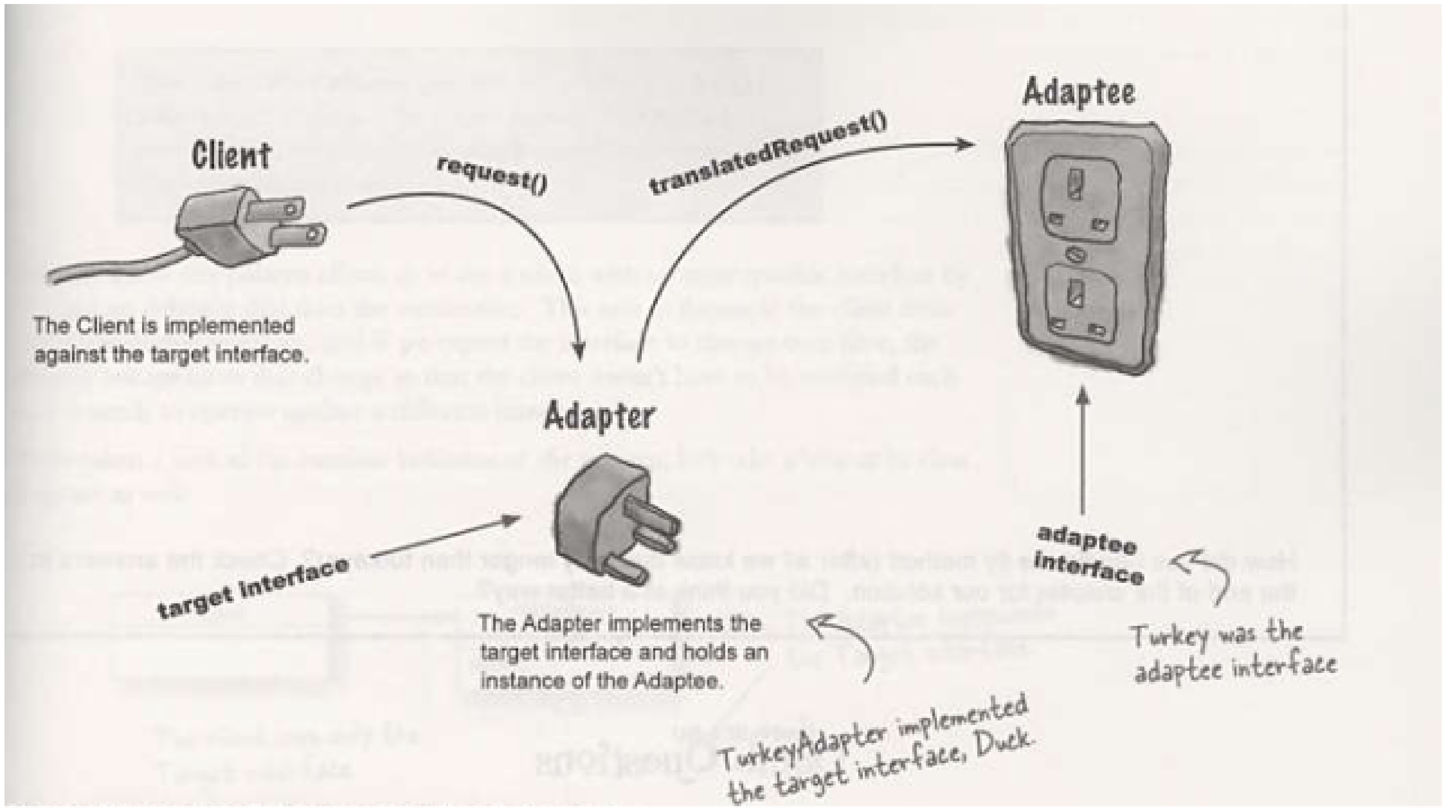
```
class Adventure {
    public static void travel(CanFly c) {
        c.fly();
    }
    public static void main(String []args){
        Bird b = new Bird();
        Insect ins = new Insect();
        travel(b); travel(ins);

        Person p = new Person();
        PersonAdapter pd = new PersonAdapter(p);
        travel(pd);
    }
}
```

```
class PersonAdapter implements CanFly{
    private Person p;
    public PersonAdapter(Person p){
       this.p = p;
    }
    public void fly(){
        p.buyTicket();
        p.takeFlight();
    }
}
```

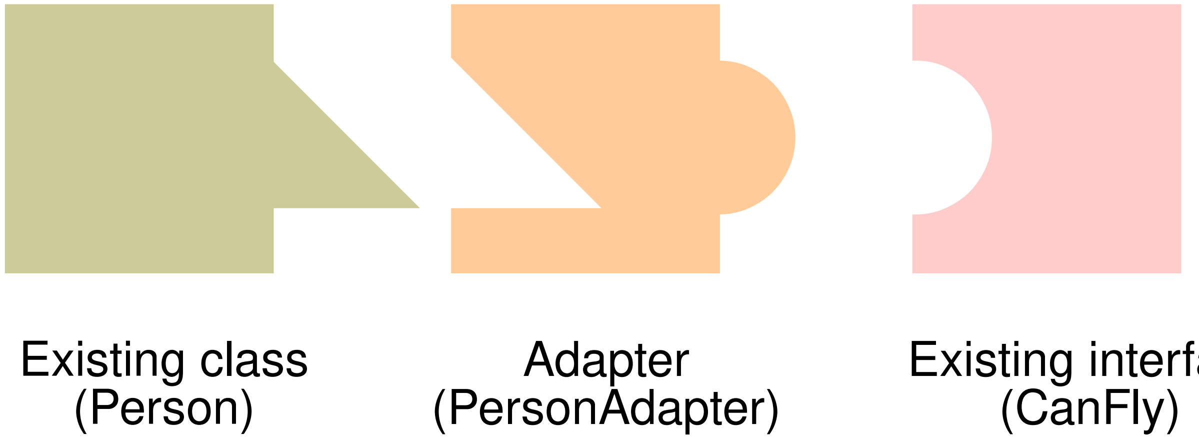
接口

- 接口适配器
  - 通过增加一个接口的实现，使得现有类能够被“适配”到该接口


Client
request()
translatedRequest()
Adaptee
The Client is implemented against the target interface.
Adapter
target interface
adaptee interface
The Adapter implements the target interface and holds an instance of the Adaptee.
Turkey was the adaptee interface
TurkeyAdapter implemented the target interface, Duck.

接口

- 接口适配器


Existing class
(Person)
Adapter
(PersonAdapter)
Existing interface
(CanFly)

接口

- 接口
  - 定义
  - 实现多个接口
  - 扩展接口
  - 接口适配器
  - 应用：工厂模式

接口

- 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);
    }
}
```

总结

- 抽象类
  - 抽象方法：只给出方法的名字，参数，返回值，没有具体实现
  - 抽象类：包含抽象方法的类
  - abstract 关键字
- 接口
  - "所有的方法都是抽象方法"
  - interface, inplements 关键字
  - 接口的扩展 : extends
  - 实现多个接口
  - 接口适配器