# day09

## 多态

### 多态的概述

1. 多态 : 面向对象中一大特征, 事物的多种形态
2. 多态发生前提:

a : 需要有子父类继承关系或者是类与接口实现关系

b : 需要子类重写父类方法

c : 多态表达式 : 父类引用指向子类对象

1. 举例说明:

定义出一个Person人类

定义出一个Teacher教师类, 教师属于人类的一种, Teacher extends Person

定义出一个Doctor 医生类, 医生属于人类的一种, Doctor extends Person

多态表达式 :

Person p; // 父类引用, p称为变量引用

new Teacher(); // 子类对象

new Doctor(); // 子类对象

Person p = new Teacher(); // 多态表达式

Person p1 = new Doctor(); // 多态表达式

人类对象p, 可以是教师; 人类对象p1, 可以是医生, 因此人类有多种表现形态

代码

|  |
| --- |
| package com.ujiuye.duotai;  // Person表示人类  public class Person {  // 定义出一个方法功能, eat(), 每一个人都必须吃饭  public void eat() {  System.out.println("每一个人都必须吃饭");  }  } |

|  |
| --- |
| package com.ujiuye.duotai;  public class Teacher extends Person {  @Override  public void eat() {  System.out.println("教师今天想吃凉皮");  }    public void teach() {  System.out.println("教师可以讲课");  }  } |

|  |
| --- |
| package com.ujiuye.duotai;  public class Doctor extends Person {  @Override  public void eat() {  System.out.println("医生吃的很清淡,养生");  }    public void savePerson() {  System.out.println("医生可以治病救人");  }  } |

|  |
| --- |
| package com.ujiuye.duotai;  public class TestDuoTai {  public static void main(String[] args) {  // 多态表达式  Person p = new Teacher();  p.eat();//教师今天想吃凉皮    // 多态表达式  Person p1 = new Doctor();  p1.eat();//医生吃的很清淡,养生  }  } |

### 多态中方法的访问特点

总结 : 编译看左, 运行看右

举例 : 多态表达式, Person p = new Teacher();

1. 编译看左 :

使用多态表达式中的变量引用调用方法功能时, 验证等号左边的类型中是否具有调用方法功能, 如果有可以编译通过; 如果没有代码报错, 编译不通过.

1. 运行看右 :

使用多态表达式的变量引用运行方法功能时, 动态绑定等号右边的类型中实现的方法功能, 因此如果等号右边(子类)中有重写方法, 调用就是子类的重写

代码

|  |
| --- |
| package com.ujiuye.duotai;  public class TestDuoTai {  public static void main(String[] args) {  // 多态表达式  Person p = new Teacher();  p.eat();//教师今天想吃凉皮  //The method teach() is undefined for the type Person  // 1.编译看左, 左边Person类型中没有定义出teach方法, 因此编译环节报错  // p.teach();    // 多态表达式  Person p1 = new Doctor();  // 2.运行看右 , 运行的是等号右边子类中的重写方法  p1.eat();//医生吃的很清淡,养生  }  } |

### 多态的向上向下转型

1. 多态向上转型: 多态表达式本身就是向上转型

父类引用指向子类对象, 向上转型

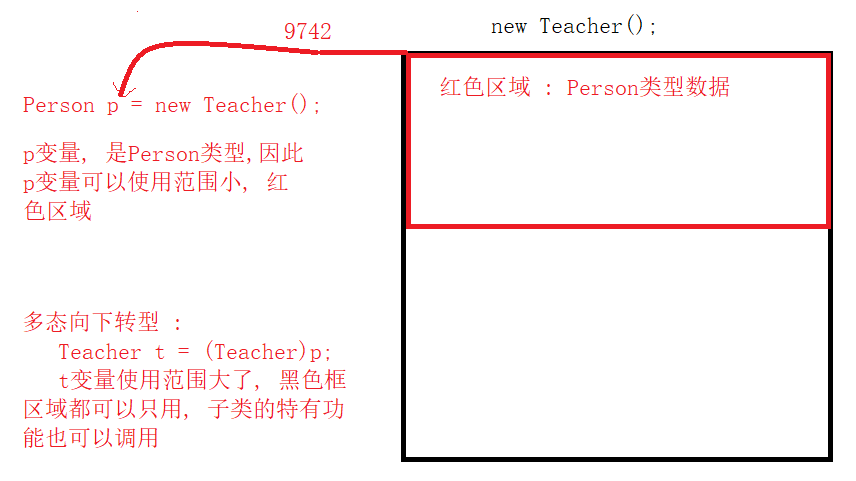
Person p = new Teacher();

本质 : p变量使用范围变小, 只能使用子父类共有功能, 子类特有功能无法调用; (理解: 将年轻人子类转换成一个老年人父类, 人年龄变大, 越老能做的事情就越少)

1. 多态向下转型 : 将指向子类对象的父类引用, 恢复成子类类型本身

Teacher t = (Teacher)p;

本质 : t变量使用范围变大, 变到了整个子类范围, 可以使用子类中特有功能; (理解: 将老年人父类转换成年轻人子类, 人年龄上变小, 越年轻可以做的事情越多)



代码

|  |
| --- |
| package com.ujiuye.duotai;  public class TestDuoTai {  public static void main(String[] args) {  // 多态表达式  Person p = new Teacher();  p.eat();//教师今天想吃凉皮  //The method teach() is undefined for the type Person  // 1.编译看左, 左边Person类型中没有定义出teach方法, 因此编译环节报错  // p.teach();  // 3. 多态向下转型 : 为了将父类引用p,回复成子类Teacher类型本身, 如此可以使用子类中特有方法  Teacher t = (Teacher)p;  t.eat();  t.teach();// 教师可以讲课    // 多态表达式  Person p1 = new Doctor();  // 2.运行看右 , 运行的是等号右边子类中的重写方法  p1.eat();//医生吃的很清淡,养生    Doctor d = (Doctor)p1;  d.savePerson();// 医生可以治病救人  }  } |

### 多态的使用和好处

1. 多态好处 : 提高代码的扩展性, 非常灵活
2. 多态表达式在方法参数列表上的使用场景:

定义出方法功能, 将父类类型作为方法的形式参数, 当方法调用时, 需要传递实际参数, 实际参数可以是当前父类类型本身对象或者可以是这个父类的任意一个子类对象

案例 : 有一个农场(设计成Farmer类), 农场中可以养很多动物, 农场中有一个方法功能(wei), 可以根据提供动物类型不同, 将每一种动物需要吃什么输出

举例 : 给wei方法提供猫类型, 方法输出: 猫吃鱼;

给wei方法提供狗类型, 方法输出 : 狗啃骨头;

给wei方法提供羊类型, 方法输出 : 羊吃青草;

... 有无数种动物, 提供不同的动物类型, wei方法输出不同动物需要喂食的东西

分析 :

1. 定义出Farmer农场类, 在类中定义出一个方法wei

public void wei(动物){// 参数列表需要一个动物, 而这个动物的类型不能确定, 于是就直接将Animal动物的父类作为方法的形式参数, 因为Animal可以表示一切动物

}

1. 动物 : 猫, 狗, 羊...都属于动物, 继承关系,抽取

抽取出一个父类Animal--->实现功能eat, 表示每一个种动物需要吃

猫, 狗, 羊...都作为Animal动物类的子类, 重写eat方法, 为什么重写? 因为每一种动物吃饭的食物不同

代码

|  |
| --- |
| package com.ujiuye.duotaigood;  public class Animal {  public void eat() {  System.out.println("动物喂食");  }  } |

|  |
| --- |
| package com.ujiuye.duotaigood;  public class Cat extends Animal {  @Override  public void eat() {  System.out.println("猫出鱼");  }  } |

|  |
| --- |
| package com.ujiuye.duotaigood;  public class Dog extends Animal {  @Override  public void eat() {  System.out.println("狗啃骨头");  }  } |

|  |
| --- |
| package com.ujiuye.duotaigood;  public class Sheep extends Animal {  @Override  public void eat() {  System.out.println("羊吃青草");  }  } |

|  |
| --- |
| package com.ujiuye.duotaigood;  public class Farmer {  // 将父类类型Animal作为方法形式参数  public void wei(Animal a) {  a.eat();  }  } |

|  |
| --- |
| package com.ujiuye.duotaigood;  public class TestFarmer {  public static void main(String[] args) {  Farmer f = new Farmer();  Cat c = new Cat();  /\*  \* public void wei(Animal a) {// Animal a = new Cat(); 多态  a.eat();  }\*/  // 喂猫  f.wei(c);// new Cat();    Dog d = new Dog();  /\*  \* public void wei(Animal a) {// Animal a = new Dog(); 多态  a.eat();  }\*/  f.wei(d);    /\*  \* public void wei(Animal a) {// Animal a = new Sheep(); 多态  a.eat();  }\*/  Sheep sh = new Sheep();  f.wei(sh);  }  } |

## 抽象类

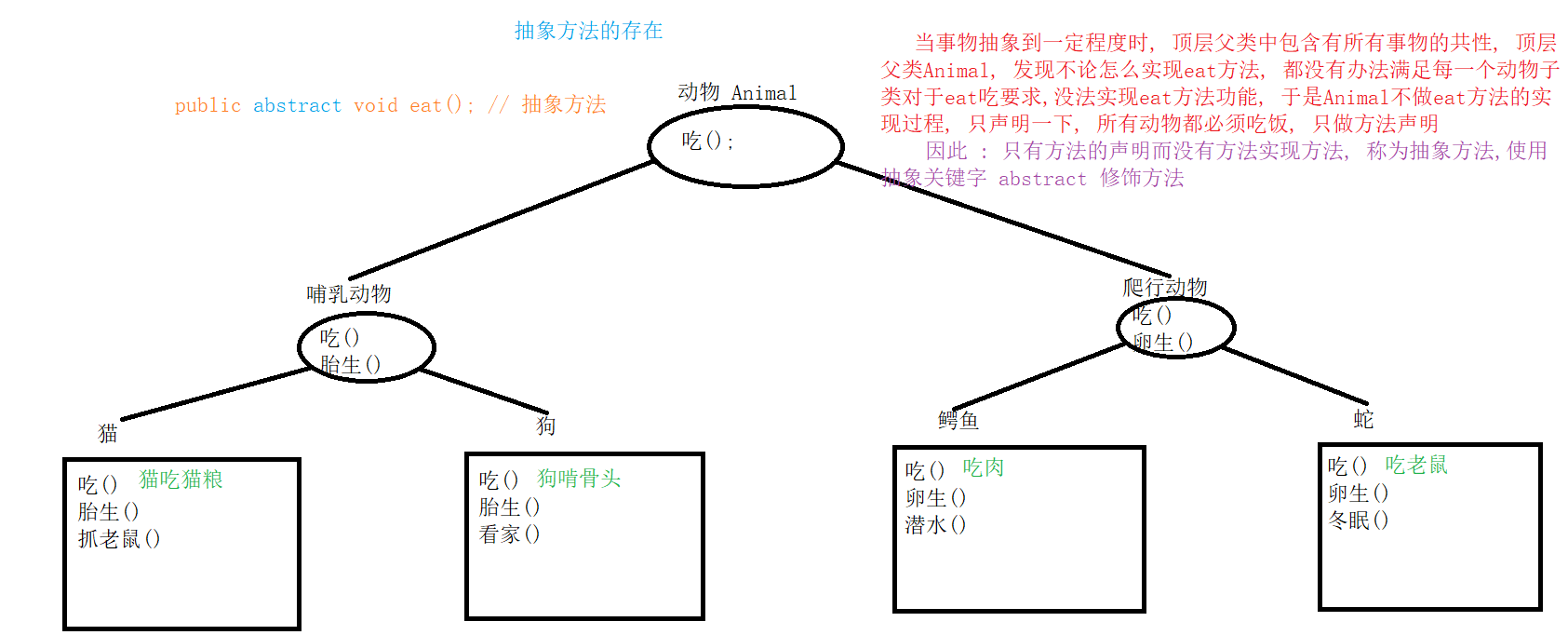
### 抽象方法和抽象类

1. 当事物抽象到一定程度时, 顶层父类中包含有所有事物的共性, 顶层父类Animal, 发现不论怎么实现eat方法, 都没有办法满足每一个动物子类对于eat吃要求,没法实现eat方法功能, 于是Animal不做eat方法的实现过程, 只声明一下, 所有动物都必须吃饭, 只做方法声明
2. 因此 : 只有方法的声明而没有方法实现方法, 称为抽象方法,使用抽象关键字 abstract 修饰方法

修饰符 abstract 返回值类型 方法名(参数列表);

1. 因为抽象方法必须存在于抽象类中, 因此类需要添加abstract关键字, 修饰成一个抽象类

修饰符 abstract class 类名{}

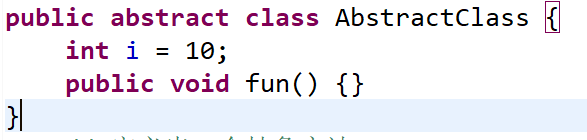


代码

|  |
| --- |
| package com.ujiuye.abstractdemo;  //The type AbstractClass must be an abstract class to define abstract methods  public abstract class AbstractClass {    // 定义出一个抽象方法  // The abstract method eat in type AbstractClass can only be defined by an abstract class  // 抽象方法只能存在于抽象类中  public abstract void eat();  } |

### 抽象使用的特点

1. 抽象方法和抽象类都需要使用abstract关键字修饰的
2. 抽象方法必须存在于抽象类中, 抽象类中可以没有抽象方法



1. 抽象方法因为没有方法体, 因此不能运行; 但是抽象方法存在的意义是, 就是为了给子类限定规则
2. 抽象类不能实例化对象(new对象), 因为抽象类中可能含有抽象方法, 抽象方法不能运行
3. 抽象类等着当父类: 需要子类继承抽象类, 子类中重写从父类中继承到的抽象方法

抽象类子类的前途

1. 子类将从抽象父类中继承到的所有抽象方法全部重写, 子类就是一个普通, 可以正常new对象使用
2. 子类没有将抽象父类中所有抽象方法全部重写, 这个子类就还是一个抽象类, 不能实例化对象, 不能正常使用

代码

|  |
| --- |
| package com.ujiuye.abstractdemo;  //The type AbstractClass must be an abstract class to define abstract methods  public abstract class AbstractClass {  // 定义出一个抽象方法  // The abstract method eat in type AbstractClass can only be defined by an abstract class  // 抽象方法只能存在于抽象类中  public abstract void eat();  } |

|  |
| --- |
| package com.ujiuye.abstractdemo;  //The type AbstractClassZi1 must implement the inherited abstract method AbstractClass.eat()  public class AbstractClassZi1 extends AbstractClass {  /\*  抽象方法重写 :  去掉abstract关键字, 方法添加大括号, 其他与普通方法重写一致  \*/  @Override  public void eat() {  System.out.println("子类重写了抽象方法");  }  } |

|  |
| --- |
| package com.ujiuye.abstractdemo;  public abstract class AbstractClassZi2 extends AbstractClass {    } |

|  |
| --- |
| package com.ujiuye.abstractdemo;  public class TestAbstract {  public static void main(String[] args) {  //Cannot instantiate the type AbstractClass  // 抽象类不能实例化对象(new 对象)  // AbstractClass ac = new AbstractClass();    AbstractClassZi1 ac1 = new AbstractClassZi1();  ac1.eat();  }  } |

### 抽象类的组成

结论 : 抽象类中都可以定义出? 普通类(成员变量, 方法, 构造方法) + 抽象方法

1. 可以定义出成员变量
2. 可以定义非抽象方法
3. 可以定义抽象方法
4. 可以定义出构造方法 :
5. 一个类型中是否可以定义出构造方法与这个类型能否实例化对象(new对象)无关, 与这个类型是否能定义出成员变量有关; 类型中可以出成员变量就可以定义出构造方法
6. 抽象类等着当父类, 子类构造方法中也需要调用抽象父类的构造

注意 : 抽象类也可以使用多态, 抽象父类的引用指向子类对象

代码

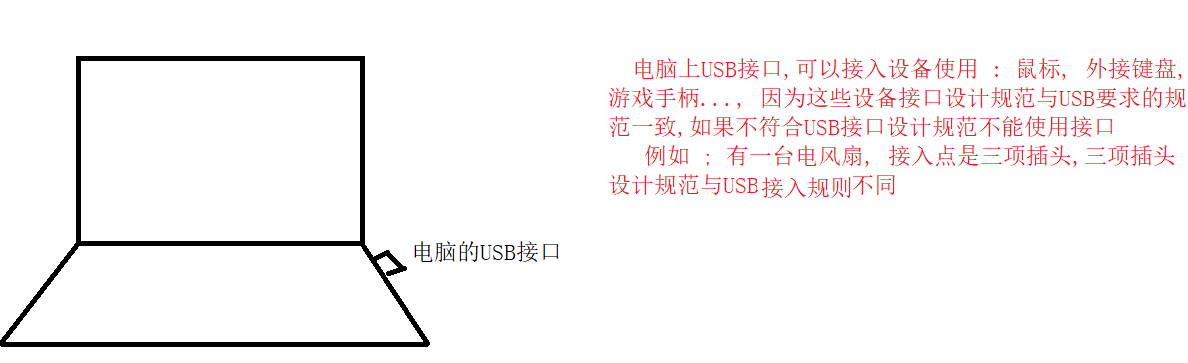
|  |
| --- |
| package com.ujiuye.abstractdemo;  public abstract class Abstract组成 {  // 1. 可以定义出成员变量  int i = 10;  static final int I = 20;  private String name = "张三";    // 2. 可以定义出非抽象方法  public void fun() {  System.out.println("-------");  }    public static void fun2() {  System.out.println("++++++++++++");  }    // 3. 抽象方法  public abstract int getSum(int x, int y);    // 4. 可以定义出构造方法  public Abstract组成() {  System.out.println("我是抽象父类的空参数构造");  }  } |

|  |
| --- |
| package com.ujiuye.abstractdemo;  public class Abstract组成Zi extends Abstract组成 {  @Override  public int getSum(int x, int y) {  return x + y;  }    public static void main(String[] args) {  Abstract组成Zi az = new Abstract组成Zi();  System.out.println(az.getSum(3, 5));  System.out.println(az.i);  az.fun();    // 测试 : 抽象类也可以使用多态  Abstract组成 a = new Abstract组成Zi();  System.out.println(a.getSum(5, 6));// 11  }  } |

## 接口

### 接口的概述

1. 接口 : 一系列规则的组成, 目前学习接口中全是抽象方法(接口中是否全是抽象方法与JDK版本有关, 从JDK8版本往后, 接口中可以添加非抽象方法)



1. 接口优势 : 因为接口中定义全部都是方法声明, 方法的实现由 实现类完成, 如此方法声明与方法的实现分离, 解决类与类之间的高耦合性(耦合 : 类型与类型之间的关联度过高, 修改维护关联度过高)

### 接口的使用和特点

1. 接口的定义: 使用interface关键字, 用于接口类型的定义

修饰符 interface 接口名{// 大驼峰原则定义接口名

// 目前全是抽象方法;

}

注意 : 接口的源文件仍然是一个.java文件, 编译后还是一个.class字节码文件

1. 接口不能实例化对象, 因为接口中全是抽象方法, 抽象方法不能运行
2. 接口等着一个类将接口中的所有抽象方法重写, 类与接口之间是实现关系, 使用implements关键字实现类与接口关系(implements关键字描述的类与接口关系, 与extends关键字描述的类与类之间继承关系, 同等理解)

class Zi implements 父接口{

重写父接口中所有抽象方法;

}

1. 实现类的前途:
2. 实现类将接口中的所有抽象方法全部抽写, 这个实现类就是一个普通类, 可以new对象
3. 实现类没有将接口中的所有抽象方法全部重写, 这个实现类就是一个抽象类, 抽象类不能new对象

代码

|  |
| --- |
| package com.ujiuye.interfacedemo;  public interface MyInterface {  // 在接口中定义出抽象方法  // 修饰符 abstract 返回值类型 方法名(参数列表);  public abstract void eat();    public abstract boolean equal(double x, double y);    //Abstract methods do not specify a body  /\*public void eat2() {    }\*/  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  public class MyInterfaceImpl2 implements MyInterface{  @Override  public void eat() {  System.out.println("接口实现类重写方法eat");  }  @Override  public boolean equal(double x, double y) {  return x == y;  }  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  // 实现类没有将接口中的所有抽象方法全部重写, 这个实现类就是一个抽象类, 抽象类不能new对象  public abstract class MyInterfaceImpl3 implements MyInterface{  @Override  public void eat() {  System.out.println("接口实现类重写方法eat");  }  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  public class TestInterface {  public static void main(String[] args) {  // Cannot instantiate the type MyInterface  // 1. 接口不能实例化  // MyInterface my = new MyInterface();    // 2. 重写了接口中所有抽象方法实现类, 可以正常使用  MyInterfaceImpl2 my2 = new MyInterfaceImpl2();  my2.eat();  System.out.println(my2.equal(3.14, 3.5));  }  } |

### 接口的组成

1. 接口中全部都是成员常量 : 接口中的成员变量默认使用 public static final修饰, 不写或者写一部分修饰符默认补全成public static final
2. 接口中全部都是抽象方法 : 所有抽象方法默认修饰符 public abstract, 不写或者写一部分修饰,默认补全成public abstract
3. 接口中不能定义构造方法 : 因为接口中不能定义成员变量

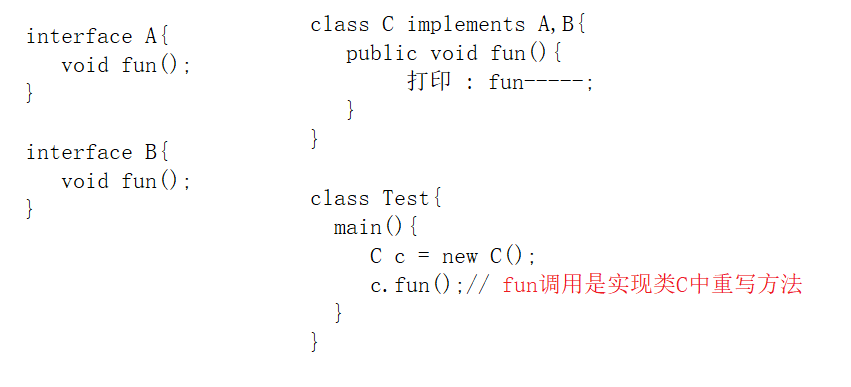
代码

|  |
| --- |
| package com.ujiuye.interfacedemo;  public interface 接口组成 {  // 1. 接口中成员全部默认使用 public static final修饰  int I = 10;  public static int J = 29;  public static final int W = 88;    // 2. 接口中定义抽象方法  public abstract void eat();  // 默认修饰符 : public abstract  public void fun();    // 3. 接口中构造方法  //Interfaces cannot have constructors  // 接口不能有构造方法  /\*public 接口组成() {    }\*/  } |

### 类与类, 类与接口, 接口与接口之间的关系

1. 类与类之间 : 继承关系, extends关键字, 单继承, 不能多继承, 可以多层继承
2. 类与接口之间 : 实现关系, implements关键字, 多实现

多实现 : 一个类可以同时实现多个接口, 多个接口之间使用 , 进行分隔



一个类在实现另外一个类的同时,还可以实现多个接口, 父优先

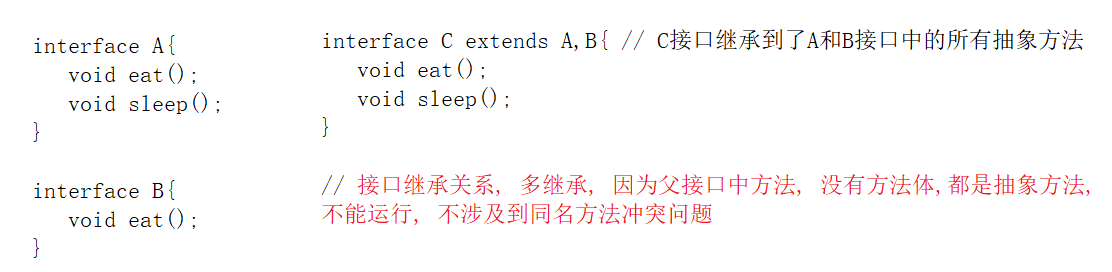
父优先 : 一切以父类为第一继承使用, 重写顺位; 父类中没有的, 再到父接口中寻找(重写)

class Zi extends Fu implements A,B{

// Zi类将所有没有重写的抽象方法重写;

}

1. 接口与接口之间 : 同类型之间都是继承关系, 接口与接口之间继承关系, 可以多继承



代码

|  |
| --- |
| package com.ujiuye.interfacedemo;  public interface A {  void fun();  //void fun1();  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  public interface B {  void fun();  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  // 接口的多实现, 一个类可以同时实现多个接口  public class InterfaceImplAAndB implements A, B {  @Override  public void fun() {    }  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  public class Fu {  public void fun() {  System.out.println("我是父类的fun");  }  } |

|  |
| --- |
| package com.ujiuye.interfacedemo;  // 子类继承一个父类的同时还能实现多个接口  public class Zi extends Fu implements A, B {  } |