# day10

## 抽象类与接口之间使用场景

1. 抽象类与接口之间的区别:

1. 定义类型:

a : 抽象类 : 使用class关键字定义

b : 接口 : 使用interface关键字定义

1. 抽象方法:

a : 抽象类 : 可以包含抽象方法, 也可以没有抽象方法

b : 接口 : 接口中可以定义方法类型, 根据JDK版本决定, 目前接口中全部都是抽象方法

1. 组成方面:

a : 抽象类 : 成员变量, 方法, 构造, 抽象方法

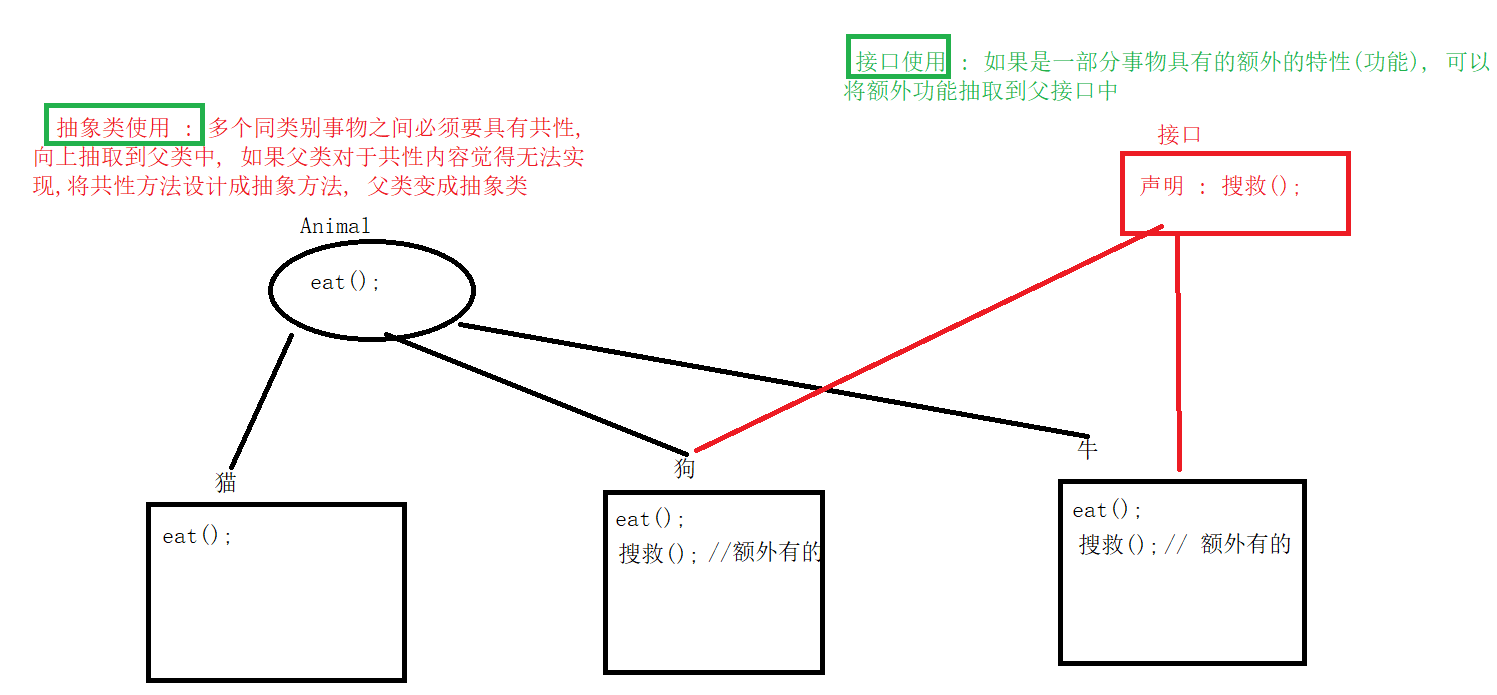
b : 接口 : 成员常量, 抽象方法

1. 与类之间关系:

a : 抽象类 : 类与类之间关系, 继承关系, extends关键字实现, 只能是单继承

b : 接口 : 类与接口之间关系, 实现关系, implements关键字, 可以多实现

1. 抽象类与接口的使用场景:



## 代码块(了解)

需要掌握的就是静态代码块

1. 代码块 : 使用一对大括号将指定代码包裹起来, 根据大括号定义位置不同, 称呼不同, 作用和运行机制也会不同
2. 代码块的分类:

a : 局部代码块(方法中)

b : 构造代码块(成员位置)

c : 静态代码块(成员位置)

d : 同步代码块(非常重要, 多线程才能讲到同步代码块作用)

### 局部代码块

1. 定义格式 : 使用一对大括号包裹代码
2. 位置 : 定义在方法中
3. 作用 :

a : 限定定义在局部代码块中的变量使用范围

b : 减少变量在内存中的存在时间(生命周期) : 在内存中, 如果某一个变量无法再使用, 那么这个变量占有的内存空间就会变成垃圾, 等待JVM虚拟机回收掉, 无用

注意 : 局部代码块只针对于定义在局部代码块中变量生效, 对于局部代码块之外的变量使用没有影响

代码

|  |
| --- |
| package com.ujiuye.block;  public class Demo01\_局部代码块 {  public static void main(String[] args) {  int j = 88;  // 局部代码块  {  // 1. 限定定义在局部代码块中的变量使用范围  int i = 10; // 变量定义  System.out.println(i);  j = 99;// 变量赋值, 局部代码块对于j值, 不会发生影响, 正常使用  }  // i cannot be resolved to a variable  // System.out.println(i);  System.out.println(j);// 99  }  } |

### 构造代码块

1. 定义格式: 使用一对大括号包裹代码
2. 位置 : 成员位置(类中方法外)
3. 作用 :

a : 给对象中成员变量进行赋值

b : 如果同一类中的多个构造方法(重载), 每一个构造中都有一部分相同逻辑, 可以将相同逻辑提升到构造代码块中实现

1. 构造代码块的运行机制:

每次创建对象同时, JVM虚拟机主动调用构造代码块一次, 构造代码块优先于构造方法执行, 构造代码块不能主动调用

代码

|  |
| --- |
| package com.ujiuye.block;  public class Demo02\_构造代码块 {  private String name;  private int age;  // 构造代码块  {  name = "张三";  age = 20;  System.out.println("我是构造代码块");  System.out.println("我是构造方法----lalalalal");  }    public Demo02\_构造代码块() {  // System.out.println("我是构造方法----lalalalal");  }    public Demo02\_构造代码块(String name) {  this.name = name;  // System.out.println("我是构造方法----lalalalal");  }    public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getAge() {  return age;  }  public void setAge(int age) {  this.age = age;  }  public static void main(String[] args) {  // 1. 创建出一个Demo02\_构造代码块对象  Demo02\_构造代码块 demo = new Demo02\_构造代码块();  System.out.println(demo.getName());// 张三  System.out.println(demo.getAge());// 20    Demo02\_构造代码块 demo2 = new Demo02\_构造代码块("小强");  System.out.println(demo2.getName());  }  } |

### 静态代码块(掌握)

1. 定义格式 : 使用一对大括号包裹起来代码, 大括号外使用static静态关键字

static{

逻辑内容;

}

1. 定义位置: 成员位置
2. 作用 :

a : 给静态成员变量进行赋值

b : 静态代码块使用场景 : 如果项目启动, 就有需要执行的代码, 并且只执行一次, 这些代码可以设计在静态代码块中

1. 静态代码块执行机制:

随着类的加载, 只执行一次 : 当一个类对应.class字节码文件进入内存, JVM虚拟机主动调用静态代码块执行一次

注意 : 代码块执行顺序

静态代码块(只执行一次)---->构造代码块(每次创建对象执行一次)--->构造方法(每次创建对象执行一次)

代码

|  |
| --- |
| package com.ujiuye.block;  public class Demo03\_静态代码块 {  int i;  static String schoolName;  // 定义出静态代码块  static {  schoolName = "第一中学";  // i = 88;  System.out.println("静态代码块执行了");// 1.静态代码块执行了  }  // 构造代码块  {  i = 10;  System.out.println("我是构造代码块" + i);// 3. 我是构造代码块10 // 5  }    public Demo03\_静态代码块() {  System.out.println("我是构造方法");// 4. 我是构造方法 // 6  }    public static void main(String[] args) {  System.out.println(Demo03\_静态代码块.schoolName);// 2. 第一中学  Demo03\_静态代码块 demo = new Demo03\_静态代码块();  Demo03\_静态代码块 demo1 = new Demo03\_静态代码块();  }  } |

## Eclipse快捷键

1. alt + / : 代码的自动补全, 需要给出代码的一部分才能补全
2. main , 自动补全main方法
3. syso, 自动补全标准输出语句
4. 关键字补全
5. 自定命名
6. 部分语法结构补全, if, for...
7. 常用快捷键 :
8. ctrl + n : 弹出创建项目工程, 创建类窗口
9. ctrl + shift + o : 自动导包, 将当前代码中所有没有导包类型进行自动包导入
10. 注释:

单行注释 : ctrl + / 取消单行注释 : ctrl + /

多行注释 : 需要选中需要注释代码行数 ctrl + shift + / 取消多行注释 : ctrl + shift + \

1. 删除行 : ctrl + d, 每次删除鼠标所在当前行
2. F2 (Fn + F2): 表示重命名, 可以给类, 包进行重命名
3. ctrl + 变量名/方法名/类名 : 查看源代码
4. ctrl + o : outline 大纲, 表示查看当前类型中主要内容(成员变量, 方法)
5. ctrl + f : find, 查找, 在当前整个类中查找
6. 代码自动生成

alt + shift + s : 比较常见的自动生成代码有 : set和get方法, equals和hashCode方法, toString, 空参数和有参数构造

## 内部类(了解)

1. 内部类 : 表示将一个类定义在另外一个类的内部
2. 举例 : 定义出一个身体Body, 身体中包含很多器官, 心脏, 肺, 肾, 肝..., 例如心脏, 很复杂, 没有办法使用某一个成员变量或者一个方法功能就将心脏可以实现的所以功能描述清楚, 于是可以在Body身体中, 为心脏存在设置一个内部类
3. 内部类根据定义的位置不同, 分成 : 成员内部类 , 局部内部类
4. 成员内部类(理解成一个成员变量)

a : 普通成员内部类

b : 私有成员内部类

c : 静态成员内部类

1. 局部内部类(理解成一个局部变量)

a : 普通局部内部类

b : 匿名内部类(掌握)

### 普通成员内部类

1. 定义格式:

class Body{// 外部类

class Heart{// 内部类, 理解成一个成员变量

属性; 方法功能;

}

}

1. 普通成员内部类使用:
2. 成员内部类可以直接使用外部类中成员
3. 普通成员内部类的使用 : 可以在外部类中, 创建出一个内部类对象, 使用内部类对象名.调用内部类中成员变量和方法功能
4. 普通成员内部类在除了外部类的其他类型中 : 调用方式为

外部类.内部类 内部类变量 = new 外部类().new 内部类对象();

代码

|  |
| --- |
| package com.ujiuye.innerclass;  public class Body {// 外部类  // blood变量表示血压  // private当前类中使用, 除了Body之外的其他外类中不能直接访问  int blood = 120;    // 普通成员内部类  class Heart{// 心脏, 理解成一个成员变量  int jump = 80;    public void show() {  System.out.println("血压为" + blood + ",心跳为:" + jump);  }  }    // 定义出一个方法功能 : 使用普通成员内部类Heart中内容  public void useHeart() {  System.out.println(blood);// 120  Heart h = new Heart();  System.out.println(h.jump);// 80  h.show();  }  } |

|  |
| --- |
| package com.ujiuye.innerclass;  public class TestInnerClass {  public static void main(String[] args) {  // 1. 普通成员内部类测试  Body b = new Body();  b.useHeart();  // 访问成员变量  System.out.println(b.blood);  System.out.println("------------");    // 3)普通成员内部类在除了外部类的其他类型中 : 调用方式为  // 外部类.内部类 内部类变量 = new 外部类().new 内部类对象();  Body.Heart h1 = new Body().new Heart();  System.out.println(h1.jump);  h1.show();  }  } |

### 私有成员内部类

1. 定义格式:

class Body{// 外部类

private class Gan{// 私有成员内部类

属性; 方法功能;

}

}

1. 私有成员内部类的使用:
2. 私有成员内部类, 可以直接使用外部类的成员
3. 因为内部类私有化, 只能在当前外部类中使用, 因此需要在外部类中, 定义出访问私有成员内部类的方法, 通过调用方法间接访问私有成员内部类中数据

代码

|  |
| --- |
| package com.ujiuye.innerclass;  public class Body1 {// 外部类  // drink表示喝酒的斤数  private int drink = 5;    private class Gan{// 私有成员内部类, 理解成是一个私有成员变量  String color = "black";    public void show() {  System.out.println("每天都要喝" + drink + "斤白酒,少喝点,不然以后肝就要变成" + color);  }  }    // 对外提供使用私有成员内部类Gan方法功能  public void useGan() {  // 创建出一个私有成员内部类对象  Gan gan = new Gan();  System.out.println(gan.color);  gan.show();  }  } |

|  |
| --- |
| package com.ujiuye.innerclass;  public class TestInnerClass {  public static void main(String[] args) {  // 1. 私有成员内部类测试  Body1 b1 = new Body1();  b1.useGan();  }  } |

### 静态成员内部类

1. 定义格式:

class Body{// 外部类

static class Shen{// 静态成员内部类, 理解成一个静态成员变量

}

}

1. 静态成员内部类使用:
2. 静态成员内部类中不能直接使用外部类中非静态成员, 需要创建出一个外部类对象, 使用外部类对象名.调用外部类中非静态成员
3. 静态成员内部类中非静态数据, 需要创建出一个静态成员内部类对象才能调用

外部类名.静态内部类名 静态内部类对象名 = new 外部类名.静态内部类();

代码

|  |
| --- |
| package com.ujiuye.innerclass;  public class Body2 {// 外部类  int day = 1;    static class Shen{  static int drink = 2;  int wc = 5;    public void show() {  // 1. 静态成员内部类中, 不能直接使用外部类中非静态成员, 可以先创建出一个外部类对象  // 使用对象名.调用  Body2 b2 = new Body2();  System.out.println(b2.day + "天,喝了" + drink + "升水,需要去卫生间" + wc+"次");  }  }  } |

|  |
| --- |
| package com.ujiuye.innerclass;  public class TestInnerClass {  public static void main(String[] args) {  // 3. 静态成员内部类测试  System.out.println(Body2.Shen.drink);// 2  // 创建出一个静态成员内部类对象  Body2.Shen b2 = new Body2.Shen();  System.out.println(b2.wc);// 5  b2.show();// 1天,喝了2升水,需要去卫生间5次  }  } |

### 普通局部内部类

1. 定义格式:

public void breath(){// 方法

class Fei{// 普通局部内部类

}

}

1. 普通局部内部类使用:

局部内部类相当于是一个局部变量理解方式, 局部变量随着方法的调用才能使用, 无法单独将局部变量获取到, 同理, 局部内部类也不能直接获取到, 只能在当前方法中, 定义出局部内部类, 创建出局部内部类对象使用, 调用带有局部内部类的方法相当于间接使用局部内部类

代码

|  |
| --- |
| package com.ujiuye.innerclass;  public class Demo01\_普通局部内部类 {  public static void main(String[] args) {  breath();  }    // 定义出一个功能  public static void breath() {  // 局部变量无法单独获取到  int day = 2;  class Fei{// 局部内部类, 理解成一个局部变量  int smoke = 50;  String color = "黑色";  public void show() {  System.out.println(day+"天,每天都抽" + smoke + "多根烟,老了以后肺就会变成" + color);  }  }  // 定义一个局部内部类之后, 马上创建出一个局部内部类对象使用  Fei f = new Fei();  System.out.println(f.color);  System.out.println(f.smoke);  f.show();  }  } |

### 匿名内部类(掌握)

1. 匿名内部类 : 本质上就是作为一个类的子类或者是一个接口的实现类存在

匿名内部类 : 语法结构, 简化一个子类或者接口实现类的设计过程

1. 匿名内部类语法结构:

new 父类或者父接口(){// 大括号就表示父类的子类或者父接口的实现类实现过程

// 需要重写父类或者父接口中抽象方法

};

第2点整体语法结构, 表示就是一个匿名内部类对象, 只可以使用一次

1. 匿名内部类对象优化:

因为匿名内部类对象只能使用一次, 有局限性, 因此将匿名内部类对象使用多态表达式进行优化, 设计成有名的内部类, 就可以多次的使用匿名内部类中实现的功能

父类或父接口 变量名 = new 父类或者父接口(){

// 需要重写父类或者父接口中抽象方法

};

代码

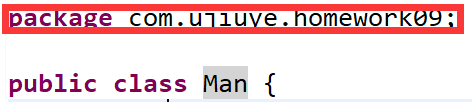
|  |
| --- |
| package com.ujiuye.innerclass;  public class Demo02\_匿名内部类使用 {  public static void main(String[] args) {  // MyInter接口的实现类对象(匿名内部类对象)  new MyInter() {  @Override  public void fun() {  System.out.println("匿名内部类实现fun");  }  @Override  public int getSum(int x, int y) {  return x + y;  }  }.fun();    // 匿名内部类优化 : 利用接口的多态性  // 父接口引用 指向 接口实现类对象  // my对象名调用的就是等号右边实现类中重写方法  MyInter my = new MyInter() {  @Override  public void fun() {  System.out.println("有名内部类实现fun");  }  @Override  public int getSum(int x, int y) {  return x + y;  }  };    my.fun();  System.out.println(my.getSum(3, 5));  }  }  interface MyInter {  public abstract void fun();  public abstract int getSum(int x, int y);  } |

## 包package

1. 包 : package关键字, 包主要作用就是分类管理类文件, 包就是项目工程下子文件夹
2. 包的使用:

a : 包的命名 : 全球唯一, 公司域名的倒叙写作, com.ujiuye.包名

b : 包的创建和声明 : 创建一个包,使用package,每一个类型中,都有package的声明, 表示当前类型所在的包文件夹路径



c : 同一个包下类名不能重复; 不同包下可以存在相同的类名, 因为包路径不同区分不同的类文件

类名本质上带有包路径

com.ujiuye.innerclass.Test;

com.ujiuye.homework09\_3.Test;

d : 导包:

1. 同一个包下的类,可以直接使用,不需要导包
2. 不同包下的类的使用,需要导包 : Scanner, Random, Arrays, 导包使用import关键字, 将需要使用的目标类型所在的具体路径导入进来
3. java.lang包下的所有类型, 使用不需要导包
4. import java.util.\*; \*表示全部, 导入当前util文件夹路径下的全部, 实际开发中不建议导包使用\*, 使用哪个类型, 就将对应类型的包导入即可, 尽量不要使用\*导入方式

## 权限修饰符

1. 权限修饰 : 就是用于表示可以使用的范围
2. 在java中一共有4种权限修饰符:

public : 关键字, 公共权限, 修饰类,修饰方法最常用

protected : 关键字, 受保护权限, 主要修饰变量和方法, 不修饰类; 受保护权限在外包的子类中, 相当于private私有使用, 因此protected 也是封装的一种体现

默认权限 : 定义类, 定义方法, 定义变量, 没有给出任何权限修饰符, 就表示使用默认权限修饰, 在类中, 默认权限不能写出来, 写出来会报错

private : 关键字, 私有, 最小权限, 只能在当前类型中使用

1. 4种权限使用范围:

public > protected > 默认 > private

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 本类中 | 本包中 | 外包子类内部 | 外包无关类 |
| public | 可以 | 可以 | 可以 | 可以 |
| protected | 可以 | 可以 | 可以 |  |
| 默认 | 可以 | 可以 |  |  |
| private | 可以 |  |  |  |

package com.ujiuye.quanxian; 包中的代码

|  |
| --- |
| package com.ujiuye.quanxian;  // 类就是一个默认权限的类  class 默认权限Class {  // 成员变量i也是默认权限  int i = 10;    public static void main(String[] args) {  System.out.println(new 默认权限Class().i);  }  } |

|  |
| --- |
| package com.ujiuye.quanxian;  public class 受保护权限Class {  protected int pro = 10;    public static void main(String[] args) {  System.out.println(new 受保护权限Class().pro);  }  } |

|  |
| --- |
| package com.ujiuye.quanxian;  public class TestQuanXian {  public static void main(String[] args) {  // 1. 默认权限可以在本包中使用  System.out.println(new 默认权限Class().i);  // 2. 受保护前线可以在本包中使用  System.out.println(new 受保护权限Class().pro);  }  } |

package com.ujiuye.other; 包中的代码

|  |
| --- |
| package com.ujiuye.other;  import com.ujiuye.quanxian.受保护权限Class;  public class 受保护子类 extends 受保护权限Class {  // 子类中可以继承大到受保护权限, 但是仅仅只能在子类中使用  // 受保护权限在外包的子类中, 相当于private私有使用  public void usePro() {  System.out.println(pro);  System.out.println(super.pro);  }  } |

|  |
| --- |
| package com.ujiuye.other;  import com.ujiuye.quanxian.受保护权限Class;  public class TestQX {  public static void main(String[] args) {  // 1. 默认权限只能在本包中使用,外包无法使用  // 默认权限Class c = new 默认权限Class();    // 2. 创建出一个受保护类型的外包子类对象  受保护子类 cl = new 受保护子类();  // System.out.println(c1.pro);    // 3. public修饰的类受保护权限Class可以在外包的无关类中使用,  // public只要是当前项目工程下, 都可以使用  System.out.println(new 受保护权限Class());  }  } |