第 10 章面向对象编程(高级部分)

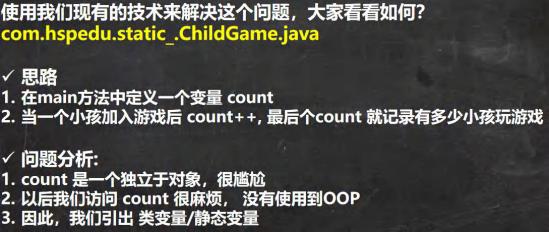
10. 1 类变量和类方法

10.1.1 类变量-提出问题

提出问题的主要目的就是让大家思考解决之道，从而引出我要讲的知识点.

说：有一群小孩在玩堆雪人,不时有新的小孩加入,请问如何知道现在共有多少人在玩? ，编写程序解决。

10.1.2 传统的方法来解决



10.1.3 类变量快速入门

思考: 如果,设计一个 int count 表示总人数,我们在创建一个小孩时，就把 count 加 1,并且 count 是所有对象共享的

就 ok 了! ，我们使用类变量来解决 ChildGame.java 改进

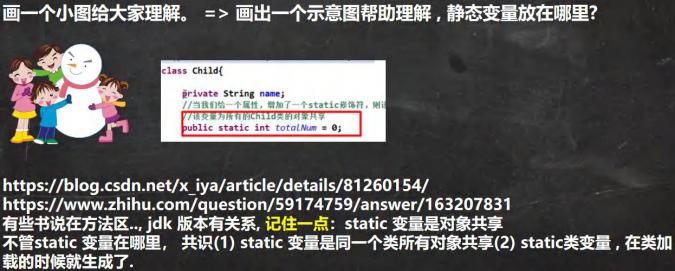
代码.

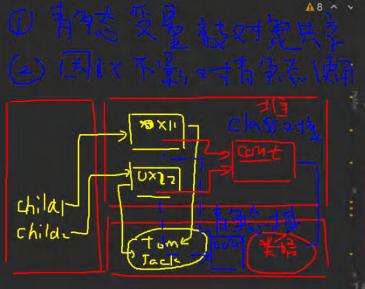
|  |
| --- |
| package com.hspedu.static\_; |

|  |
| --- |
| public class ChildGame {  public static void main(String[] args) {  //定义一个变量 count, 统计有多少小孩加入了游戏  int count = 0;  Child child1 = new Child(" 白骨精");  child1.join();  //count++;  child1.count++;  Child child2 = new Child("狐狸精");  child2.join();  //count++;  child2.count++;  Child child3 = new Child("老鼠精");  child3.join();  //count++;  child3.count++;  //===========  //类变量，可以通过类名来访问  System.out.println("共有" + Child.count + " 小孩加入了游戏...");  //下面的代码输出什么? |

|  |
| --- |
| System.out.println("child1.count=" + child1.count);//3  System.out.println("child2.count=" + child2.count);//3  System.out.println("child3.count=" + child3.count);//3  }  }  class Child { //类  private String name;  //定义一个变量 count ,是一个类变量(静态变量) static 静态  //该变量最大的特点就是会被 Child 类的所有的对象实例共享  public static int count = 0;  public Child(String name) {  this.name = name;  }  public void join() {  System.out.println(name + " 加入了游戏..");  }  } |

10.1.4 类变量内存布局

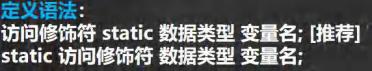




10.1.5 什么是类变量



10.1.6 如何定义类变量



10.1.7 如何访问类变量 VisitStatic.java

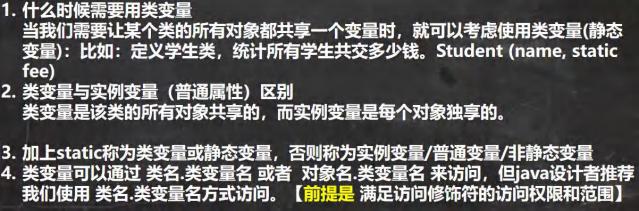


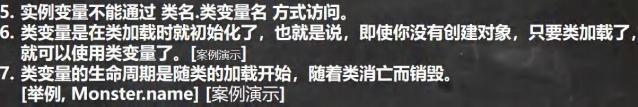
代码

|  |
| --- |
| package com.hspedu.static\_;  public class VisitStatic {  public static void main(String[] args) {  //类名.类变量名  //说明：类变量是随着类的加载而创建，所以即使没有创建对象实例也可以访问  System.out.println(A.name);  A a = new A();  //通过对象名.类变量名  System.out.println("a.name=" + a.name);  }  }  class A {  //类变量  //类变量的访问，必须遵守 相关的访问权限.  public static String name = "韩顺平教育";  //普通属性/普通成员变量/非静态属性/非静态成员变量/实例变量  private int num = 10; |

|  |
| --- |
| } |

10.1.8 类变量使用注意事项和细节讨论 StaticDetail.java



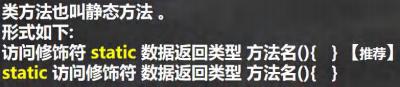


代码:

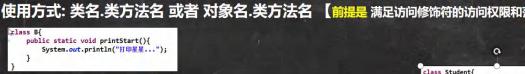
|  |
| --- |
| package com.hspedu.static\_;  public class StaticDetail {  public static void main(String[] args) {  B b = new B();  //System.out.println(B.n1);  System.out.println(B.n2);  //静态变量是类加载的时候，就创建了,所以我们没有创建对象实例  //也可以通过类名.类变量名来访问 |

|  |
| --- |
| System.out.println(C.address);  }  }  class B {  public int n1 = 100;  public static int n2 = 200;  }  class C {  public static String address = "北京";  } |

10.1.9 类方法基本介绍



10.1. 10 类方法的调用



10.1. 11 类方法应用案例 StaticMethod.java

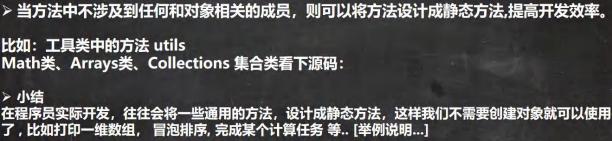
代码:

|  |
| --- |
| package com.hspedu.static\_;  public class StaticMethod { |

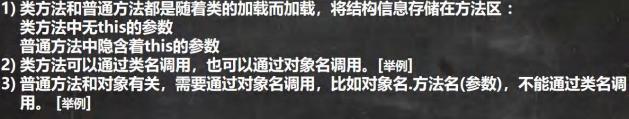
|  |
| --- |
| public static void main(String[] args) {  //创建 2 个学生对象，叫学费  Stu tom = new Stu("tom");  //tom.payFee(100);  Stu.payFee(100);//对不对?对  Stu mary = new Stu("mary");  //mary.payFee(200);  Stu.payFee(200);//对  //输出当前收到的总学费  Stu.showFee();//300  //如果我们希望不创建实例，也可以调用某个方法(即当做工具来使用)  //这时，把方法做成静态方法时非常合适  System.out.println("9 开平方的结果是=" + Math.sqrt(9));  System.out.println(MyTools.calSum(10, 30));  }  }  //开发自己的工具类时，可以将方法做成静态的，方便调用  class MyTools {  //求出两个数的和  public static double calSum(double n1, double n2) { |

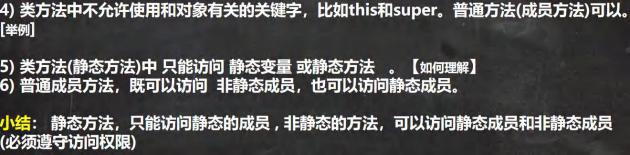
|  |
| --- |
| return n1 + n2;  }  //可以写出很多这样的工具方法...  }  class Stu {  private String name;//普通成员  //定义一个静态变量，来累积学生的学费  private static double fee = 0;  public Stu(String name) {  this.name = name;  }  //说明  //1. 当方法使用了 static 修饰后，该方法就是静态方法  //2. 静态方法就可以访问静态属性/变量  public static void payFee(double fee) {  Stu.fee += fee;//累积到  }  public static void showFee() {  System.out.println("总学费有:" + Stu.fee);  }  } |

10.1. 12 类方法经典的使用场景



10.1. 13 类方法使用注意事项和细节讨论 StaticMethodDetail.java





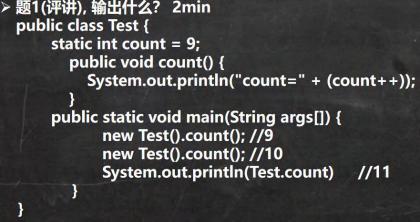
代码:

|  |
| --- |
| package com.hspedu.static\_;  public class StaticMethodDetail {  public static void main(String[] args) {  D.hi();//ok  //非静态方法，不能通过类名调用  //D.say();, 错误，需要先创建对象，再调用  new D().say();//可以 |

|  |
| --- |
| }  }  class D {  private int n1 = 100;  private static int n2 = 200;  public void say() {//非静态方法,普通方法  }  public static void hi() {//静态方法,类方法  //类方法中不允许使用和对象有关的关键字，  //比如 this 和 super 。普通方法(成员方法)可以。  //System.out.println(this.n1);  }  //类方法(静态方法)中 只能访问 静态变量 或静态方法  //口诀:静态方法只能访问静态成员.  public static void hello() {  System.out.println(n2);  System.out.println(D.n2);  //System.out.println(this.n2);不能使用  hi();//OK  //say();//错误  }  //普通成员方法，既可以访问 非静态成员，也可以访问静态成员 |

|  |
| --- |
| //小结: 非静态方法可以访问 静态成员和非静态成员  public void ok() {  //非静态成员  System.out.println(n1);  say();  //静态成员  System.out.println(n2);  hello();  }  } |

10.1. 14 课堂练习 StaticExercise01.java



10.1. 15 题 2(评讲) ，看看下面代码有没有错误,如果有错误，就修改，看看输出什么?

代码:

|  |
| --- |
| class Person { //StaticExercise02.java 2min 时间  private int id; |

|  |
| --- |
| private static int total = 0;  public static int getTotalPerson() {  //id++;//错误, 注销  return total;  }  public Person() {//构造器  total++; //total = 1  id = total;//id = 1  }  }  public class TestPerson {  public static void main(String[] args) {  System.out.println("Number of total is " +Person.getTotalPerson()); //0  Person p1 = new Person();  System.out.println( "Number of total is "+ Person.getTotalPerson()); //1  }  } |

10.1. 16 题 3(评讲) ，看看下面代码有没有错误,如果有错误，就修改，看看total 等于多少 4?

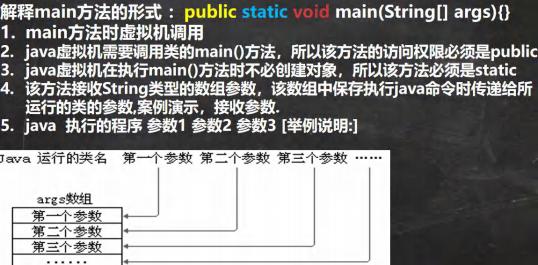
代码:

|  |
| --- |
| class Person { //StaticExercise03.java 2min 看  private int id;  private static int total = 0;  public static void setTotalPerson(int total){ |

|  |
| --- |
| // this.total = total;//错误，因为在 static 方法中，不可以使用 this 关键字  Person.total = total;  }  public Person() {//构造器  total++;  id = total;  }}  public class TestPerson {  public static void main(String[] args) {  Person.setTotalPerson(3);  new Person(); //最后 total 的值就是 4  }  }  小结：记住两句话 (1) 静态方法，只能访问静态成员 (2) 非静态方法，可以访问所有的成员  (3) 在编写代码时，仍然要遵守访问权限规则 |

10.2 理解 main 方法语法

10.2.1 深入理解 main 方法



10.2.2 特别提示：

1) 在 main()方法中，我们可以直接调用 main 方法所在类的静态方法或静态属性。

2) 但是，不能直接访问该类中的非静态成员，必须创建该类的一个实例对象后，才能通过这个对象去访问类中的非静

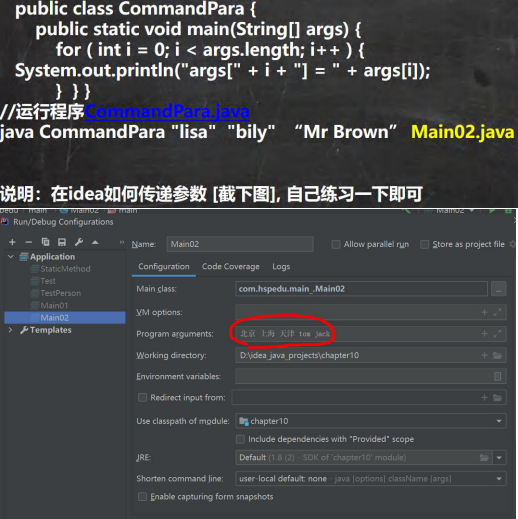
态成员，[举例说明] Main01.java

3) 代码

|  |
| --- |
| package com.hspedu.main\_;  public class Main01 {  //静态的变量/属性  private static String name = "韩顺平教育";  //非静态的变量/属性  private int n1 = 10000;  //静态方法  public static void hi() {  System.out.println("Main01 的 hi 方法"); |

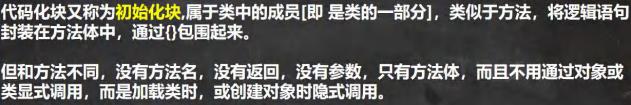
|  |
| --- |
| }  //非静态方法  public void cry() {  System.out.println("Main01 的 cry 方法");  }  public static void main(String[] args) {  //可以直接使用 name  //1. 静态方法 main 可以访问本类的静态成员  System.out.println("name=" + name);  hi();  //2. 静态方法 main 不可以访问本类的非静态成员  //System.out.println("n1=" + n1);//错误  //cry();  //3. 静态方法 main 要访问本类的非静态成员，需要先创建对象 , 再调用即可  Main01 main01 = new Main01();  System.out.println(main01.n1);//ok  main01.cry();  }  } |

10.2.3 案例演示

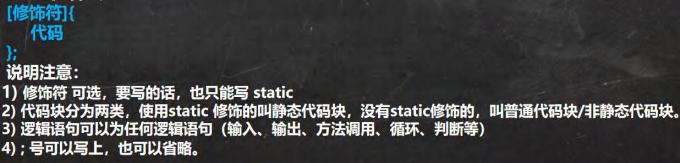


10.3 代码块

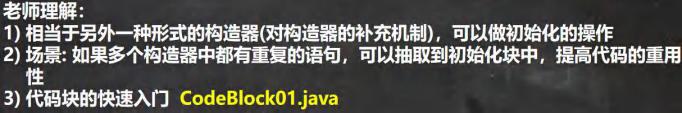
10.3.1 基本介绍



10.3.2 基本语法



10.3.3 代码块的好处和案例演示





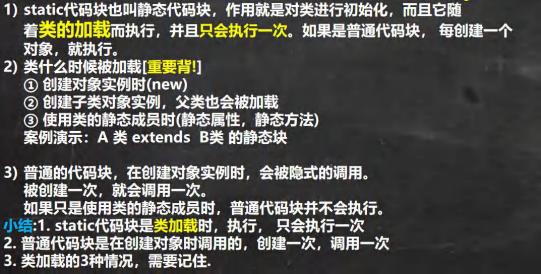
代码:

|  |
| --- |
| package com.hspedu.codeblock\_;  public class CodeBlock01 {  public static void main(String[] args) {  Movie movie = new Movie("你好，李焕英");  System.out.println("===============");  Movie movie2 = new Movie("唐探 3", 100, "陈思诚");  } |

|  |
| --- |
| }  class Movie {  private String name;  private double price;  private String director;  //3 个构造器-》重载  //老韩解读  //(1) 下面的三个构造器都有相同的语句  //(2) 这样代码看起来比较冗余  //(3) 这时我们可以把相同的语句，放入到一个代码块中，即可  //(4) 这样当我们不管调用哪个构造器，创建对象，都会先调用代码块的内容  //(5) 代码块调用的顺序优先于构造器..  {  System.out.println(" 电影屏幕打开...");  System.out.println("广告开始...");  System.out.println(" 电影正是开始...");  };  public Movie(String name) {  System.out.println("Movie(String name) 被调用...");  this.name = name;  }  public Movie(String name, double price) { |

|  |
| --- |
| this.name = name;  this.price = price;  }  public Movie(String name, double price, String director) {  System.out.println("Movie(String name, double price, String director) 被调用...");  this.name = name;  this.price = price;  this.director = director;  }  } |

10.3.4 代码块使用注意事项和细节讨论 CodeBlockDetail01.java

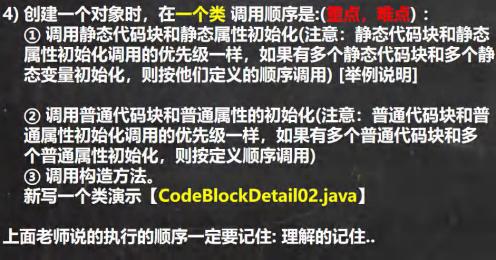


代码:

|  |
| --- |
| package com.hspedu.codeblock\_;  public class CodeBlockDetail01 {  public static void main(String[] args) {  //类被加载的情况举例  //1. 创建对象实例时(new)  // AA aa = new AA();  //2. 创建子类对象实例，父类也会被加载, 而且，父类先被加载，子类后被加载  // AA aa2 = new AA();  //3. 使用类的静态成员时(静态属性，静态方法)  // System.out.println(Cat.n1);  //static 代码块，是在类加载时，执行的，而且只会执行一次.  // DD dd = new DD();  // DD dd1 = new DD();  //普通的代码块，在创建对象实例时，会被隐式的调用。  // 被创建一次，就会调用一次。  // 如果只是使用类的静态成员时，普通代码块并不会执行  System.out.println(DD.n1);//8888, 静态模块块一定会执行  }  }  class DD {  public static int n1 = 8888;//静态属性  //静态代码块  static {  System.out.println("DD 的静态代码 1 被执行...");//  }  //普通代码块, 在 new 对象时，被调用，而且是每创建一个对象，就调用一次 |

|  |
| --- |
| //可以这样简单的，理解 普通代码块是构造器的补充  {  System.out.println("DD 的普通代码块...");  }  }  class Animal {  //静态代码块  static {  System.out.println("Animal 的静态代码 1 被执行...");//  }  }  class Cat extends Animal {  public static int n1 = 999;//静态属性  //静态代码块  static {  System.out.println("Cat 的静态代码 1 被执行...");//  }  }  class BB {  //静态代码块  static {  System.out.println("BB 的静态代码 1 被执行...");//1  }  }  class AA extends BB {  //静态代码块 |

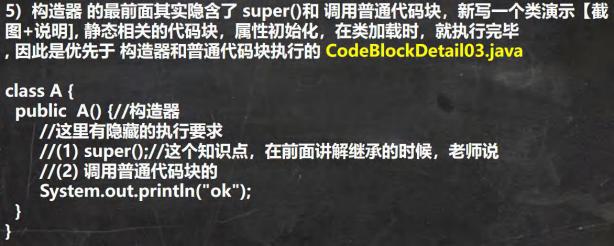
|  |
| --- |
| static {  System.out.println("AA 的静态代码 1 被执行...");//2  }  } |



代码:

|  |
| --- |
| package com.hspedu.codeblock\_;  public class CodeBlockDetail02 {  public static void main(String[] args) {  A a = new A();// (1) A 静态代码块 01 (2) getN1 被调用...(3)A 普通代码块 01(4)getN2 被调用...(5)A() 构造器被调  用  }  }  class A {  { //普通代码块 |

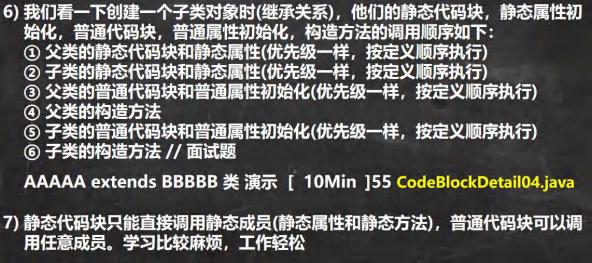
|  |
| --- |
| System.out.println("A 普通代码块 01");  }  private int n2 = getN2();//普通属性的初始化  static { //静态代码块  System.out.println("A 静态代码块 01");  }  //静态属性的初始化  private static int n1 = getN1();  public static int getN1() {  System.out.println("getN1 被调用...");  return 100;  }  public int getN2() { //普通方法/非静态方法  System.out.println("getN2 被调用...");  return 200;  }  //无参构造器  public A() {  System.out.println("A() 构造器被调用");  }  } |



代码

|  |
| --- |
| package com.hspedu.codeblock\_;  public class CodeBlockDetail03 {  public static void main(String[] args) {  new BBB();//(1)AAA 的普通代码块(2)AAA() 构造器被调用(3)BBB 的普通代码块(4)BBB() 构造器被调用  }  }  class AAA { //父类 Object  {  System.out.println("AAA 的普通代码块");  }  public AAA() {  //(1)super()  //(2)调用本类的普通代码块  System.out.println("AAA() 构造器被调用....");  } |

|  |
| --- |
| }  class BBB extends AAA {  {  System.out.println("BBB 的普通代码块...");  }  public BBB() {  //(1)super()  //(2)调用本类的普通代码块  System.out.println("BBB() 构造器被调用....");  }  } |



代码:

|  |
| --- |
| package com.hspedu.codeblock\_; |

|  |
| --- |
| public class CodeBlockDetail04 {  public static void main(String[] args) {  //老师说明  //(1) 进行类的加载  //1. 1 先加载 父类 A02 1.2 再加载 B02  //(2) 创建对象  //2. 1 从子类的构造器开始  //new B02();//对象  new C02();  }  }  class A02 { //父类  private static int n1 = getVal01();  static {  System.out.println("A02 的一个静态代码块..");//(2)  }  {  System.out.println("A02 的第一个普通代码块..");//(5)  }  public int n3 = getVal02();//普通属性的初始化  public static int getVal01() {  System.out.println("getVal01");//(1)  return 10;  } |

|  |
| --- |
| public int getVal02() {  System.out.println("getVal02");//(6)  return 10;  }  public A02() {//构造器  //隐藏  //super()  //普通代码和普通属性的初始化......  System.out.println("A02 的构造器");//(7)  }  }  class C02 {  private int n1 = 100;  private static int n2 = 200;  private void m1() {  }  private static void m2() {  } |

|  |
| --- |
| static {  //静态代码块，只能调用静态成员  //System.out.println(n1);错误  System.out.println(n2);//ok  //m1();//错误  m2();  }  {  //普通代码块，可以使用任意成员  System.out.println(n1);  System.out.println(n2);//ok  m1();  m2();  }  }  class B02 extends A02 { //  private static int n3 = getVal03();  static {  System.out.println("B02 的一个静态代码块..");//(4)  }  public int n5 = getVal04();  {  System.out.println("B02 的第一个普通代码块..");//(9) |

|  |
| --- |
| }  public static int getVal03() {  System.out.println("getVal03");//(3)  return 10;  }  public int getVal04() {  System.out.println("getVal04");//(8)  return 10;  }  //一定要慢慢的去品..  public B02() {//构造器  //隐藏了  //super()  //普通代码块和普通属性的初始化...  System.out.println("B02 的构造器");//(10)  // TODO Auto-generated constructor stub  }  } |

10.3.5 课堂练习题 CodeBlockExercise01.java

|  |
| --- |
| 题 1：下面的代码输出什么？1min  class Person {  public static int total;//静态变量 |

|  |
| --- |
| static {//静态代码块  total = 100;  System.out.println("in static block!");//(1)  }  }  public class Test {  public static void main(String[] args) {  System.out.println("total = "+ Person.total); //100  System.out.println("total = "+ Person.total); //100  }  } |

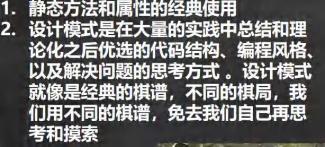
10.3.6 题 2：下面的代码输出什么？

|  |
| --- |
| 题 2：下面的代码输出什么？CodeBlockExercise02.java  class Sample  {  Sample(String s)  {  System.out.println(s);  }  Sample()  {  System.out.println(“Sample 默认构造函数被调用");  }  }  //==== |

|  |
| --- |
| class Test{  Sample sam1=new Sample("sam1 成员初始化");//  static Sample sam=new Sample("静态成员 sam 初始化 ");//  static {  System.out.println("static 块执行");//  if(sam==null)System.out.println("sam is null");  }  Test()//构造器  {  System.out.println("Test 默认构造函数被调用");//  }  }  //主方法  public static void main(String str[])  {  Test a=new Test();//无参构造器  }  //运行结果， 输出什么内容，并写出. 2min 看看  1. 静态成员 sam 初始化  2. static 块执行  3. sam1 成员初始化  4. Test 默认构造函数被调用 |

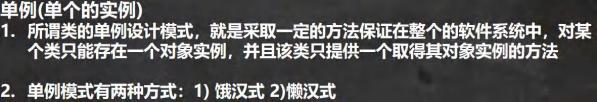
10.4 单例设计模式

10.4.1 什么是设计模式

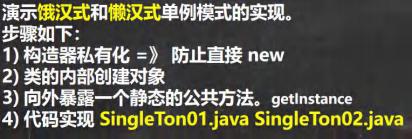




10.4.2 什么是单例模式



10.4.3 单例模式应用实例



代码:

|  |
| --- |
| package com.hspedu.single\_;  public class SingleTon01 { |

|  |
| --- |
| public static void main(String[] args) {  // GirlFriend xh = new GirlFriend("小红");  // GirlFriend xb = new GirlFriend("小白");  //通过方法可以获取对象  GirlFriend instance = GirlFriend.getInstance();  System.out.println(instance);  GirlFriend instance2 = GirlFriend.getInstance();  System.out.println(instance2);  System.out.println(instance == instance2);//T  //System.out.println(GirlFriend.n1);  //...  }  }  //有一个类， GirlFriend  //只能有一个女朋友  class GirlFriend {  private String name; |

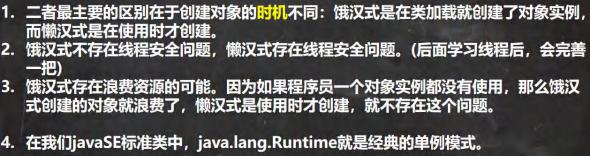
|  |
| --- |
| //public static int n1 = 100;  //为了能够在静态方法中，返回 gf 对象，需要将其修饰为 static  //對象，通常是重量級的對象, 餓漢式可能造成創建了對象，但是沒有使用.  private static GirlFriend gf = new GirlFriend("小红红");  //如何保障我们只能创建一个 GirlFriend 对象  //步骤[单例模式-饿汉式]  //1. 将构造器私有化  //2. 在类的内部直接创建对象(该对象是 static)  //3. 提供一个公共的 static 方法，返回 gf 对象  private GirlFriend(String name) {  System.out.println("構造器被調用.");  this.name = name;  }  public static GirlFriend getInstance() {  return gf;  }  @Override  public String toString() {  return "GirlFriend{" +  "name='" + name + '\'' +  '}';  } |

|  |
| --- |
| } |
| package com.hspedu.single\_;  /\*\*  \* 演示懶漢式的單例模式  \*/  public class SingleTon02 {  public static void main(String[] args) {  //new Cat("大黃");  //System.out.println(Cat.n1);  Cat instance = Cat.getInstance();  System.out.println(instance);  //再次調用 getInstance  Cat instance2 = Cat.getInstance();  System.out.println(instance2);  System.out.println(instance == instance2);//T  }  }  //希望在程序運行過程中，只能創建一個 Cat對象 |

|  |
| --- |
| //使用單例模式  class Cat {  private String name;  public static int n1 = 999;  private static Cat cat ; //默認是 null  //步驟  //1.仍然構造器私有化  //2.定義一個 static 靜態屬性對象  //3.提供一個 public 的 static 方法，可以返回一個 Cat 對象  //4.懶漢式，只有當用戶使用 getInstance 時，才返回 cat 對象, 後面再次調用時，會返回上次創建的 cat 對象  // 從而保證了單例  private Cat(String name) {  System.out.println("構造器調用...");  this.name = name;  }  public static Cat getInstance() {  if(cat == null) {//如果還沒有創建 cat 對象  cat = new Cat("小可愛");  }  return cat;  }  @Override  public String toString() { |

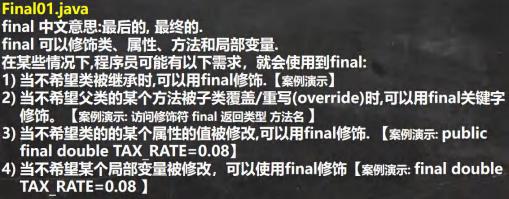
|  |
| --- |
| return "Cat{" +  "name='" + name + '\'' +  '}';  }  } |

10.4.4 饿汉式 VS 懒汉式



10.5 final 关键字

10.5.1 基本介绍



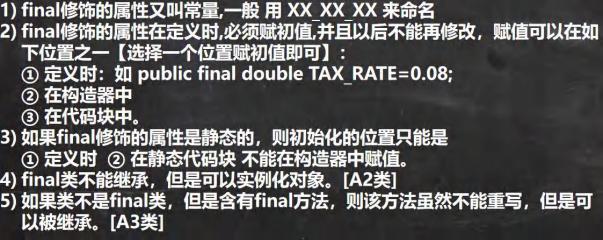
代码:

|  |
| --- |
| package com.hspedu.final\_;  public class Final01 {  public static void main(String[] args) { |

|  |
| --- |
| E e = new E();  //e.TAX\_RATE = 0.09;  }  }  //如果我们要求 A 类不能被其他类继承  //可以使用 final 修饰 A 类  final class A { }  //class B extends A {}  class C {  //如果我们要求 hi 不能被子类重写  //可以使用 final 修饰 hi 方法  public final void hi() {}  }  class D extends C {  // @Override  // public void hi() {  // System.out.println("重写了 C 类的 hi 方法..");  // }  }  //当不希望类的的某个属性的值被修改,可以用 final 修饰  class E {  public final double TAX\_RATE = 0.08;//常量 |

|  |
| --- |
| }  //当不希望某个局部变量被修改，可以使用 final 修饰  class F {  public void cry() {  //这时，NUM 也称为 局部常量  final double NUM = 0.01;  //NUM = 0.9;  System.out.println("NUM=" + NUM);  }  } |

10.5.2 final 使用注意事项和细节讨论

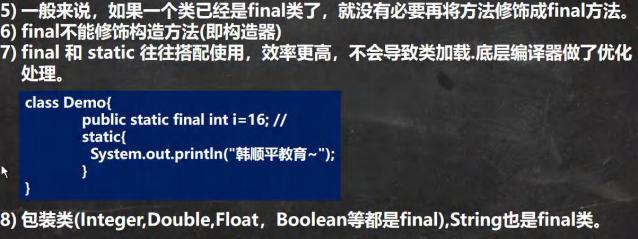


代码:

|  |
| --- |
| package com.hspedu.final\_;  public class FinalDetail01 { |

|  |
| --- |
| public static void main(String[] args) {  CC cc = new CC();  new EE().cal();  }  }  class AA {  /\*  1. 定义时：如 public final double TAX\_RATE=0.08;  2. 在构造器中  3. 在代码块中  \*/  public final double TAX\_RATE = 0.08;//1.定义时赋值  public final double TAX\_RATE2 ;  public final double TAX\_RATE3 ;  public AA() {//构造器中赋值  TAX\_RATE2 = 1. 1;  }  {//在代码块赋值  TAX\_RATE3 = 8.8;  }  }  class BB { |

|  |
| --- |
| /\*  如果 final 修饰的属性是静态的，则初始化的位置只能是  1 定义时 2 在静态代码块 不能在构造器中赋值。  \*/  public static final double TAX\_RATE = 99.9;  public static final double TAX\_RATE2 ;  static {  TAX\_RATE2 = 3.3;  }  }  //final 类不能继承，但是可以实例化对象  final class CC { }  //如果类不是 final 类，但是含有 final 方法，则该方法虽然不能重写，但是可以被继承  //即，仍然遵守继承的机制.  class DD {  public final void cal() {  System.out.println("cal()方法");  }  }  class EE extends DD { } |



代码:

|  |
| --- |
| package com.hspedu.final\_;  public class FinalDetail02 {  public static void main(String[] args) {  System.out.println(BBB.num);  //包装类,String 是 final 类，不能被继承  }  }  //final 和 static 往往搭配使用，效率更高，不会导致类加载.底层编译器做了优化处理  class BBB {  public final static int num = 10000;  static {  System.out.println("BBB 静态代码块被执行");  } |

|  |
| --- |
| }  final class AAA{  //一般来说，如果一个类已经是 final 类了，就没有必要再将方法修饰成 final 方法  //public final void cry() {}  } |

10.5.3 final 应用实例



代码:

|  |
| --- |
| package com.hspedu.final\_;  public class FinalExercise01 {  public static void main(String[] args) {  Circle circle = new Circle(5.0);  System.out.println("面积=" + circle.calArea());  }  }  class Circle {  private double radius;  private final double PI;// = 3. 14;  //构造器  public Circle(double radius) {  this.radius = radius;  //PI = 3. 14; |

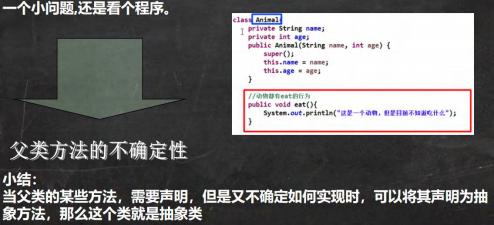
|  |
| --- |
| }  {  PI = 3. 14;  }  public double calArea() {  return PI \* radius \* radius;  }  } |

题 2

|  |
| --- |
| public int addOne(final int x) { //下面的代码是否有误，为什么? 1min  ++x; //错误,原因是不能修改 final x 的值  return x + 1; //这里是可以.  }  } |

10.6 抽象类

10.6.1 先看一个问题 Abstract01.java

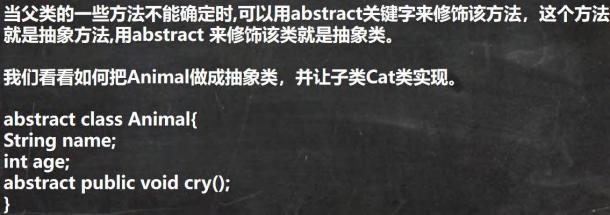


代码:

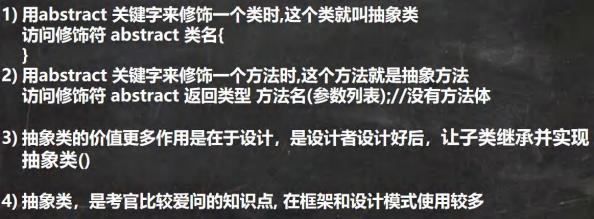
|  |
| --- |
| package com.hspedu.abstract\_;  public class Abstract01 {  public static void main(String[] args) {  }  }  abstract class Animal {  private String name;  public Animal(String name) {  this.name = name;  }  //思考：这里 eat 这里你实现了，其实没有什么意义  //即： 父类方法不确定性的问题  //===> 考虑将该方法设计为抽象(abstract)方法  //===> 所谓抽象方法就是没有实现的方法  //===> 所谓没有实现就是指，没有方法体  //===> 当一个类中存在抽象方法时，需要将该类声明为 abstract 类  //===> 一般来说，抽象类会被继承，有其子类来实现抽象方法.  // public void eat() {  // System.out.println("这是一个动物，但是不知道吃什么..");  // } |

|  |
| --- |
| public abstract void eat() ;  } |

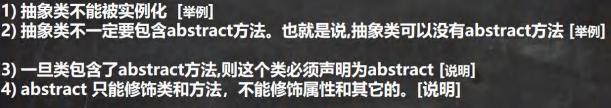
10.6.2 解决之道-抽象类快速入门



10.6.3 抽象类的介绍



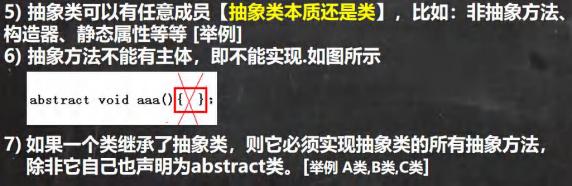
10.6.4 抽象类使用的注意事项和细节讨论 AbstractDetail01.java



代码:

|  |
| --- |
| package com.hspedu.abstract\_;  public class AbstractDetail01 {  public static void main(String[] args) {  //抽象类，不能被实例化  //new A();  }  }  //抽象类不一定要包含 abstract 方法。也就是说,抽象类可以没有 abstract 方法  // ，还可以有实现的方法。  abstract class A {  public void hi() {  System.out.println("hi");  }  }  //一旦类包含了 abstract 方法,则这个类必须声明为 abstract  abstract class B {  public abstract void hi();  }  //abstract 只能修饰类和方法，不能修饰属性和其它的  class C {  // public abstract int n1 = 1;  } |

10.6.5 抽象类使用的注意事项和细节讨论 AbstractDetail02.java



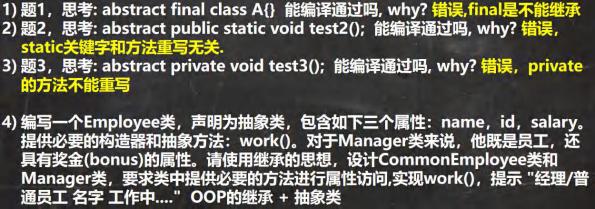


代码:

|  |
| --- |
| package com.hspedu.abstract\_;  public class AbstractDetail02 {  public static void main(String[] args) {  System.out.println("hello");  }  }  //抽象方法不能使用 private 、final 和 static 来修饰，因为这些关键字都是和重写相违背的  abstract class H {  public abstract void hi();//抽象方法  }  //如果一个类继承了抽象类，则它必须实现抽象类的所有抽象方法，除非它自己也声明为 abstract 类  abstract class E {  public abstract void hi();  }  abstract class F extends E {  }  class G extends E {  @Override  public void hi() { //这里相等于 G 子类实现了父类 E 的抽象方法，所谓实现方法，就是有方法体 |

|  |
| --- |
| }  }  //抽象类的本质还是类，所以可以有类的各种成员  abstract class D {  public int n1 = 10;  public static String name = "韩顺平教育";  public void hi() {  System.out.println("hi");  }  public abstract void hello();  public static void ok() {  System.out.println("ok");  }  } |

10.6.6 课堂练习题 AbstractExercise01.java 5min 练习



代码:

|  |
| --- |
| package com.hspedu.abstract\_;  public class AbstractExercise01 {  public static void main(String[] args) {  //测试  Manager jack = new Manager("jack", 999, 50000);  jack.setBonus(8000);  jack.work();  CommonEmployee tom = new CommonEmployee("tom", 888, 20000);  tom.work();  }  }  package com.hspedu.abstract\_;  abstract public class Employee {  private String name;  private int id;  private double salary;  public Employee(String name, int id, double salary) {  this.name = name;  this.id = id;  this.salary = salary; |

|  |
| --- |
| }  //将work 做成一个抽象方法  public abstract void work();  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public int getId() {  return id;  }  public void setId(int id) {  this.id = id;  }  public double getSalary() {  return salary;  }  public void setSalary(double salary) {  this.salary = salary;  } |

|  |
| --- |
| }  package com.hspedu.abstract\_;  public class Manager extends Employee {  private double bonus;  public Manager(String name, int id, double salary) {  super(name, id, salary);  }  public double getBonus() {  return bonus;  }  public void setBonus(double bonus) {  this.bonus = bonus;  }  @Override  public void work() {  System.out.println("经理 " + getName() + " 工作中...");  }  }  package com.hspedu.abstract\_;  public class CommonEmployee extends Employee{ |

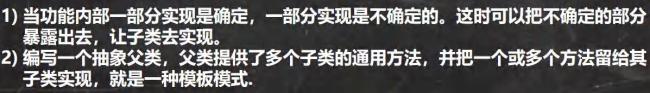
|  |
| --- |
| public CommonEmployee(String name, int id, double salary) {  super(name, id, salary);  }  @Override  public void work() {  System.out.println("普通员工 " + getName() + " 工作中...");  }  } |

10.7 抽象类最佳实践-模板设计模式

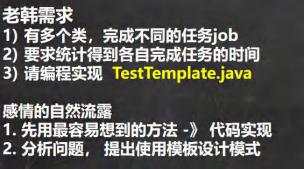
10.7.1 基本介绍

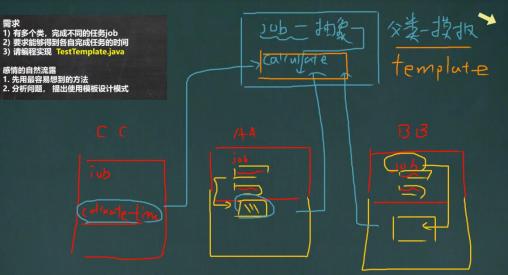


10.7.2 模板设计模式能解决的问题



10.7.3 最佳实践





代码:

|  |
| --- |
| package com.hspedu.abstract\_;  abstract public class Template { //抽象类-模板设计模式  public abstract void job();//抽象方法  public void calculateTime() {//实现方法，调用job 方法  //得到开始的时间  long start = System.currentTimeMillis(); |

|  |
| --- |
| job(); //动态绑定机制  //得的结束的时间  long end = System.currentTimeMillis();  System.out.println("任务执行时间 " + (end - start));  }  } |
| package com.hspedu.abstract\_;  public class AA extends Template {  //计算任务  //1+....+ 800000  @Override  public void job() { //实现 Template 的抽象方法job  long num = 0;  for (long i = 1; i <= 800000; i++) {  num += i;  }  }  // public void job2() {  // //得到开始的时间  // long start = System.currentTimeMillis();  // long num = 0; |

|  |
| --- |
| // for (long i = 1; i <= 200000; i++) {  // num += i;  // }  // //得的结束的时间  // long end = System.currentTimeMillis();  // System.out.println("AA 执行时间 " + (end - start));  // }  }  package com.hspedu.abstract\_;  public class BB extends Template{  public void job() {//这里也去，重写了 Template 的job 方法  long num = 0;  for (long i = 1; i <= 80000; i++) {  num \*= i;  }  }  } |
| package com.hspedu.abstract\_;  public class TestTemplate {  public static void main(String[] args) { |

|  |
| --- |
| AA aa = new AA();  aa.calculateTime(); //这里还是需要有良好的 OOP 基础，对多态  BB bb = new BB();  bb.calculateTime();  }  } |

10.8 接口

10.8.1 为什么有接口



10.8.2 接口快速入门



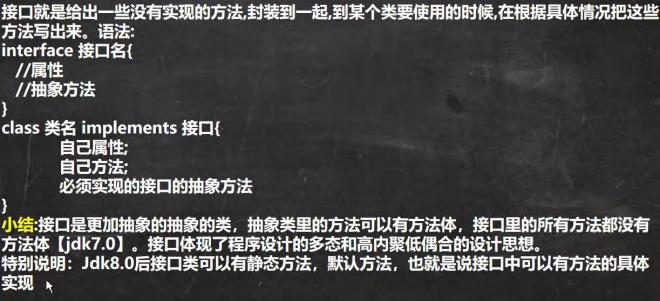
代码:

|  |
| --- |
| package com.hspedu.interface\_;  public interface UsbInterface { //接口  //规定接口的相关方法,老师规定的. 即规范...  public void start();  public void stop();  }  package com.hspedu.interface\_;  public class Camera implements UsbInterface{//实现接口,就是把接口方法实现  @Override  public void start() {  System.out.println("相机开始工作...");  }  @Override  public void stop() {  System.out.println("相机停止工作....");  }  } |

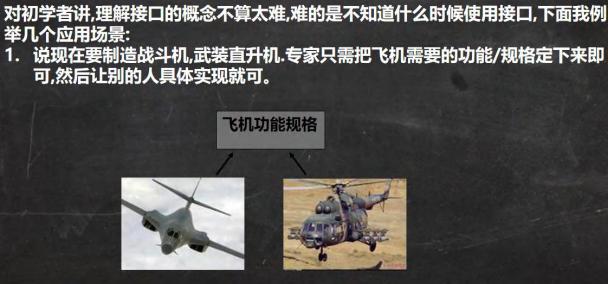
|  |
| --- |
| package com.hspedu.interface\_;  //Phone 类 实现 UsbInterface  //解读 1. 即 Phone 类需要实现 UsbInterface 接口 规定/声明的方法  public class Phone implements UsbInterface {  @Override  public void start() {  System.out.println("手机开始工作...");  }  @Override  public void stop() {  System.out.println("手机停止工作.....");  }  }  package com.hspedu.interface\_;  public class Interface01 {  public static void main(String[] args) {  //创建手机，相机对象  //Camera 实现了 UsbInterface  Camera camera = new Camera();  //Phone 实现了 UsbInterface  Phone phone = new Phone();  //创建计算机  Computer computer = new Computer(); |

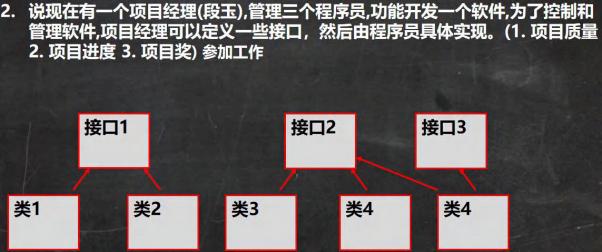
|  |
| --- |
| computer.work(phone);//把手机接入到计算机  System.out.println("===============");  computer.work(camera);//把相机接入到计算机  }  } |

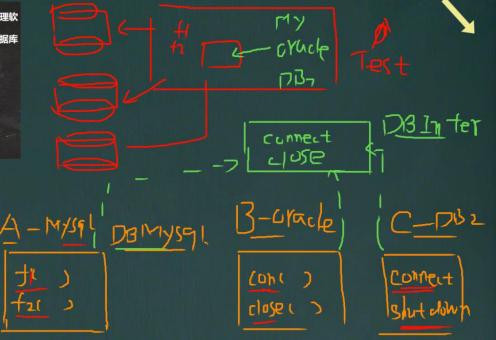
10.8.3 基本介绍



10.8.4 深入讨论







代码:

|  |
| --- |
| package com.hspedu.interface\_;  public interface DBInterface { //项目经理  public void connect();//连接方法  public void close();//关闭连接  }  package com.hspedu.interface\_;  //A 程序  public class MysqlDB implements DBInterface {  @Override  public void connect() {  System.out.println("连接 mysql"); |

|  |
| --- |
| }  @Override  public void close() {  System.out.println("关闭 mysql");  }  }  package com.hspedu.interface\_;  //B 程序员连接 Oracle  public class OracleDB implements DBInterface{  @Override  public void connect() {  System.out.println("连接 oracle");  }  @Override  public void close() {  System.out.println("关闭 oracle");  }  }  package com.hspedu.interface\_;  public class Interface03 {  public static void main(String[] args) { |

|  |
| --- |
| MysqlDB mysqlDB = new MysqlDB();  t(mysqlDB);  OracleDB oracleDB = new OracleDB();  t(oracleDB);  }  public static void t(DBInterface db) {  db.connect();  db.close();  }  } |

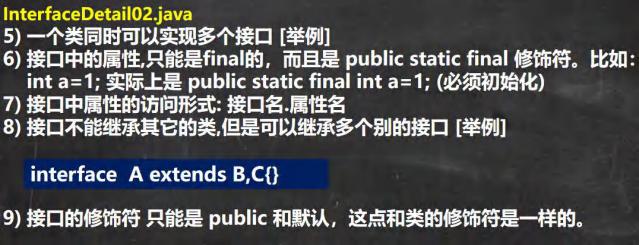
10.8.5 注意事项和细节



代码:

|  |
| --- |
| package com.hspedu.interface\_;  public class InterfaceDetail01 {  public static void main(String[] args) {  //new IA(); |

|  |
| --- |
| }  }  //1.接口不能被实例化  //2.接口中所有的方法是 public 方法, 接口中抽象方法，可以不用 abstract 修饰  //3.一个普通类实现接口,就必须将该接口的所有方法都实现,可以使用 alt+enter 来解决  //4.抽象类去实现接口时，可以不实现接口的抽象方法  interface IA {  void say();//修饰符 public protected 默认 private  void hi();  }  class Cat implements IA{  @Override  public void say() {  }  @Override  public void hi() {  }  }  abstract class Tiger implements IA {  } |

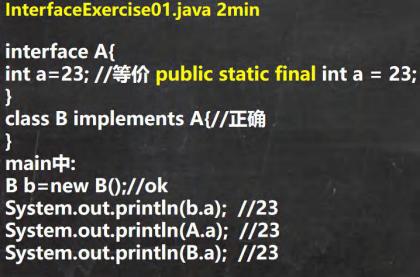


代码

|  |
| --- |
| package com.hspedu.interface\_;  public class InterfaceDetail02 {  public static void main(String[] args) {  //老韩证明 接口中的属性,是 public static final  System.out.println(IB.n1);//说明 n1 就是 static  //IB.n1 = 30; 说明 n1 是 final  }  }  interface IB {  //接口中的属性,只能是 final 的，而且是 public static final 修饰符  int n1 = 10; //等价 public static final int n1 = 10;  void hi();  }  interface IC {  void say();  } |

|  |
| --- |
| //接口不能继承其它的类,但是可以继承多个别的接口  interface ID extends IB,IC {  }  //接口的修饰符 只能是 public 和默认，这点和类的修饰符是一样的  interface IE{}  //一个类同时可以实现多个接口  class Pig implements IB,IC {  @Override  public void hi() {  }  @Override  public void say() {  }  } |

10.8.6 课堂练习



10.8.7 实现接口 vs 继承类

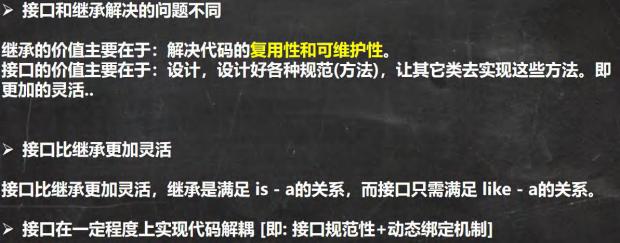


代码:

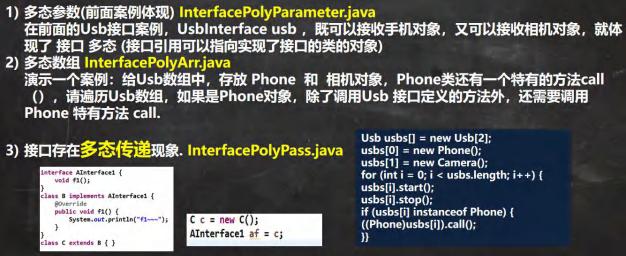
|  |
| --- |
| package com.hspedu.interface\_;  public class ExtendsVsInterface {  public static void main(String[] args) {  LittleMonkey wuKong = new LittleMonkey("悟空");  wuKong.climbing();  wuKong.swimming();  wuKong.flying();  }  }  //猴子  class Monkey { |

|  |
| --- |
| private String name;  public Monkey(String name) {  this.name = name;  }  public void climbing() {  System.out.println(name + " 会爬树...");  }  public String getName() {  return name;  }  }  //接口  interface Fishable {  void swimming();  }  interface Birdable {  void flying();  }  //继承  //小结: 当子类继承了父类，就自动的拥有父类的功能  // 如果子类需要扩展功能，可以通过实现接口的方式扩展.  // 可以理解 实现接口 是 对java 单继承机制的一种补充. |

|  |
| --- |
| class LittleMonkey extends Monkey implements Fishable,Birdable {  public LittleMonkey(String name) {  super(name);  }  @Override  public void swimming() {  System.out.println(getName() + " 通过学习，可以像鱼儿一样游泳...");  }  @Override  public void flying() {  System.out.println(getName() + " 通过学习，可以像鸟儿一样飞翔...");  }  } |



10.8.8 接口的多态特性



代码:

|  |
| --- |
| package com.hspedu.interface\_;  public class InterfacePolyParameter {  public static void main(String[] args) {  //接口的多态体现  //接口类型的变量 if01 可以指向 实现了 IF 接口类的对象实例  IF if01 = new Monster();  if01 = new Car();  //继承体现的多态  //父类类型的变量 a 可以指向 继承 AAA 的子类的对象实例  AAA a = new BBB();  a = new CCC();  }  } |

|  |
| --- |
| interface IF {}  class Monster implements IF{}  class Car implements IF{}  class AAA {  }  class BBB extends AAA {}  class CCC extends AAA {} |
| package com.hspedu.interface\_;  public class InterfacePolyArr {  public static void main(String[] args) {  //多态数组 -> 接口类型数组  Usb[] usbs = new Usb[2];  usbs[0] = new Phone\_();  usbs[1] = new Camera\_();  /\*  给 Usb 数组中，存放 Phone 和 相机对象，Phone 类还有一个特有的方法 call() ,  请遍历 Usb 数组，如果是 Phone对象，除了调用 Usb 接口定义的方法外，  还需要调用 Phone 特有方法 call  \*/  for(int i = 0; i < usbs.length; i++) { |

|  |
| --- |
| usbs[i].work();//动态绑定..  //和前面一样，我们仍然需要进行类型的向下转型  if(usbs[i] instanceof Phone\_) {//判断他的运行类型是 Phone\_  ((Phone\_) usbs[i]).call();  }  }  }  }  interface Usb{  void work();  }  class Phone\_ implements Usb {  public void call() {  System.out.println("手机可以打电话...");  }  @Override  public void work() {  System.out.println("手机工作中...");  }  }  class Camera\_ implements Usb {  @Override |

|  |
| --- |
| public void work() {  System.out.println("相机工作中...");  }  } |
| package com.hspedu.interface\_;  /\*\*  \* 演示多态传递现象  \*/  public class InterfacePolyPass {  public static void main(String[] args) {  //接口类型的变量可以指向，实现了该接口的类的对象实例  IG ig = new Teacher();  //如果 IG 继承了 IH 接口，而 Teacher 类实现了 IG 接口  //那么，实际上就相当于 Teacher 类也实现了 IH 接口.  //这就是所谓的 接口多态传递现象.  IH ih = new Teacher();  }  }  interface IH {  void hi();  }  interface IG extends IH{ }  class Teacher implements IG { |

|  |
| --- |
| @Override  public void hi() {  }  } |

10.8.9 课堂练习 InterfaceExercise02.java

|  |
| --- |
| package com.hspedu.interface\_;  public class InterfaceExercise02 {  public static void main(String[] args) {  }  }  interface A { // 1min 看看  int x = 0;  } //想到 等价 public static final int x = 0;  class B {  int x = 1;  } //普通属性  class C extends B implements A {  public void pX() {  //System.out.println(x); //错误，原因不明确 x |

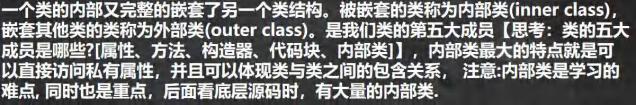
|  |
| --- |
| //可以明确的指定 x  //访问接口的 x 就使用 A.x  //访问父类的 x 就使用 super.x  System.out.println(A.x + " " + super.x);  }  public static void main(String[] args) {  new C().pX();  }  } |

10.9 内部类

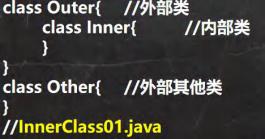
如果定义类在局部位置(方法中/代码块) :(1) 局部内部类 (2) **匿名内部类**

定义在成员位置 (1) 成员内部类 (2) 静态内部类

10.9.1 基本介绍



10.9.2 基本语法



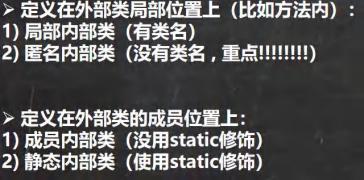
10.9.3 快速入门案例

代码:

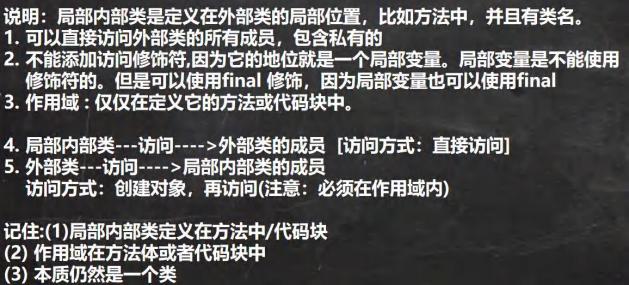
|  |
| --- |
| package com.hspedu.innerclass;  public class InnerClass01 { //外部其他类  public static void main(String[] args) {  }  }  class Outer { //外部类  private int n1 = 100;//属性  public Outer(int n1) {//构造器  this.n1 = n1;  }  public void m1() {//方法  System.out.println("m1()");  }  {//代码块  System.out.println("代码块...");  }  class Inner { //内部类, 在 Outer 类的内部 |

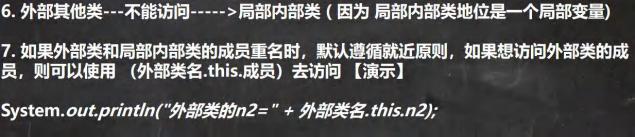
|  |
| --- |
| }  } |

10.9.4 内部类的分类



10.9.5 局部内部类的使用 LocalInnerClass.java



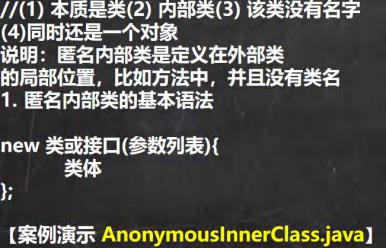


代码:

|  |
| --- |
| package com.hspedu.innerclass;  /\*\*  \* 演示局部内部类的使用  \*/  public class LocalInnerClass {//  public static void main(String[] args) {  //演示一遍  Outer02 outer02 = new Outer02();  outer02.m1();  System.out.println("outer02 的 hashcode=" + outer02);  }  }  class Outer02 {//外部类  private int n1 = 100;  private void m2() {  System.out.println("Outer02 m2()");  }//私有方法  public void m1() {//方法  //1.局部内部类是定义在外部类的局部位置,通常在方法  //3.不能添加访问修饰符,但是可以使用 final 修饰  //4.作用域 : 仅仅在定义它的方法或代码块中  final class Inner02 {//局部内部类(本质仍然是一个类)  //2.可以直接访问外部类的所有成员，包含私有的 |

|  |
| --- |
| private int n1 = 800;  public void f1() {  //5. 局部内部类可以直接访问外部类的成员，比如下面 外部类n1 和 m2()  //7. 如果外部类和局部内部类的成员重名时，默认遵循就近原则，如果想访问外部类的成员，  // 使用 外部类名.this.成员）去访问  // 老韩解读 Outer02.this 本质就是外部类的对象, 即哪个对象调用了 m1, Outer02.this 就是哪个对象  System.out.println("n1=" + n1 + " 外部类的 n1=" + Outer02.this.n1);  System.out.println("Outer02.this hashcode=" + Outer02.this);  m2();  }  }  //6. 外部类在方法中，可以创建 Inner02 对象，然后调用方法即可  Inner02 inner02 = new Inner02();  inner02.f1();  }  } |

10.9.6 匿名内部类的使用(重要!!!!!!!)



代码:

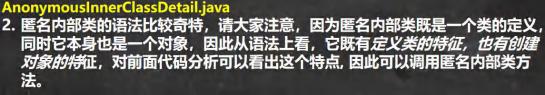
|  |
| --- |
| package com.hspedu.innerclass;  /\*\*  \* 演示匿名内部类的使用  \*/  public class AnonymousInnerClass {  public static void main(String[] args) {  Outer04 outer04 = new Outer04();  outer04.method();  }  }  class Outer04 { //外部类  private int n1 = 10;//属性  public void method() {//方法  //基于接口的匿名内部类  //老韩解读  //1.需求： 想使用 IA 接口,并创建对象  //2.传统方式，是写一个类，实现该接口，并创建对象  //3.老韩需求是 Tiger/Dog 类只是使用一次，后面再不使用  //4. 可以使用匿名内部类来简化开发  //5. tiger 的编译类型 ? IA  //6. tiger 的运行类型 ? 就是匿名内部类 Outer04$1  /\*  我们看底层 会分配 类名 Outer04$1 |

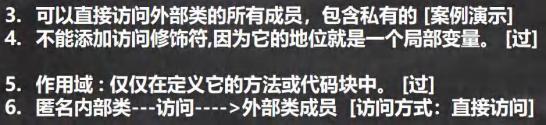
|  |
| --- |
| class Outer04$1 implements IA {  @Override  public void cry() {  System.out.println("老虎叫唤...");  }  }  \*/  //7.jdk 底层在创建匿名内部类 Outer04$1,立即马上就创建了 Outer04$1 实例，并且把地址  // 返回给 tiger  //8. 匿名内部类使用一次，就不能再使用  IA tiger = new IA() {  @Override  public void cry() {  System.out.println("老虎叫唤...");  }  };  System.out.println("tiger 的运行类型=" + tiger.getClass());  tiger.cry();  tiger.cry();  tiger.cry();  // IA tiger = new Tiger();  // tiger.cry();  //演示基于类的匿名内部类  //分析 |

|  |
| --- |
| //1. father 编译类型 Father  //2. father 运行类型 Outer04$2  //3. 底层会创建匿名内部类  /\*  class Outer04$2 extends Father{  @Override  public void test() {  System.out.println("匿名内部类重写了 test 方法");  }  }  \*/  //4. 同时也直接返回了 匿名内部类 Outer04$2 的对象  //5. 注意("jack") 参数列表会传递给 构造器  Father father = new Father("jack"){  @Override  public void test() {  System.out.println("匿名内部类重写了 test 方法");  }  };  System.out.println("father 对象的运行类型=" + father.getClass());//Outer04$2  father.test();  //基于抽象类的匿名内部类  Animal animal = new Animal(){  @Override |

|  |
| --- |
| void eat() {  System.out.println("小狗吃骨头...");  }  };  animal.eat();  }  }  interface IA {//接口  public void cry();  }  //class Tiger implements IA {  //  // @Override  // public void cry() {  // System.out.println("老虎叫唤...");  // }  //}  //class Dog implements IA{  // @Override  // public void cry() {  // System.out.println("小狗汪汪...");  // }  //}  class Father {//类 |

|  |
| --- |
| public Father(String name) {//构造器  System.out.println("接收到 name=" + name);  }  public void test() {//方法  }  }  abstract class Animal { //抽象类  abstract void eat();  } |







代码:

|  |
| --- |
| package com.hspedu.innerclass;  public class AnonymousInnerClassDetail { |

|  |
| --- |
| public static void main(String[] args) {  Outer05 outer05 = new Outer05();  outer05.f1();  //外部其他类---不能访问----->匿名内部类  System.out.println("main outer05 hashcode=" + outer05);  }  }  class Outer05 {  private int n1 = 99;  public void f1() {  //创建一个基于类的匿名内部类  //不能添加访问修饰符, 因为它的地位就是一个局部变量  //作用域 : 仅仅在定义它的方法或代码块中  Person p = new Person(){  private int n1 = 88;  @Override  public void hi() {  //可以直接访问外部类的所有成员，包含私有的  //如果外部类和匿名内部类的成员重名时，匿名内部类访问的话，  //默认遵循就近原则，如果想访问外部类的成员，则可以使用 （外部类名.this.成员）去访问  System.out.println("匿名内部类重写了 hi 方法 n1=" + n1 +  " 外部内的 n1=" + Outer05.this.n1 );  //Outer05.this 就是调用 f1 的 对象  System.out.println("Outer05.this hashcode=" + Outer05.this); |

|  |
| --- |
| }  };  p.hi();//动态绑定, 运行类型是 Outer05$1  //也可以直接调用, 匿名内部类本身也是返回对象  // class 匿名内部类 extends Person {}  // new Person(){  // @Override  // public void hi() {  // System.out.println("匿名内部类重写了 hi 方法,哈哈...");  // }  // @Override  // public void ok(String str) {  // super.ok(str);  // }  // }.ok("jack");  }  }  class Person {//类  public void hi() {  System.out.println("Person hi()");  }  public void ok(String str) { |

|  |
| --- |
| System.out.println("Person ok() " + str);  }  }  //抽象类/接口... |

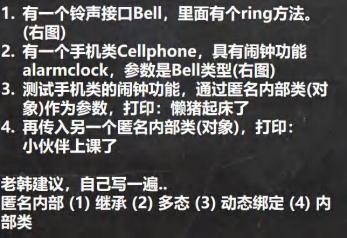
10.9.7 匿名内部类的最佳实践

当做实参直接传递，简洁高效。 InnerClassExercise01.java

|  |
| --- |
| package com.hspedu.innerclass;  import com.hspedu.abstract\_.AA;  public class InnerClassExercise01 {  public static void main(String[] args) {  //当做实参直接传递，简洁高效  f1(new IL() {  @Override  public void show() {  System.out.println("这是一副名画~~...");  }  });  //传统方法  f1(new Picture());  } |

|  |
| --- |
| //静态方法,形参是接口类型  public static void f1(IL il) {  il.show();  }  }  //接口  interface IL {  void show();  }  //类->实现 IL=> 编程领域 (硬编码)  class Picture implements IL {  @Override  public void show() {  System.out.println("这是一副名画 XX...");  }  } |

10.9.8 匿名内部类课堂练习



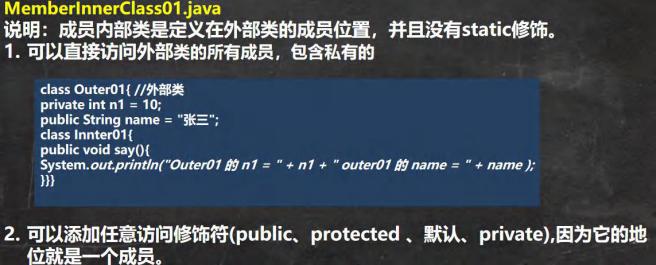
代码:

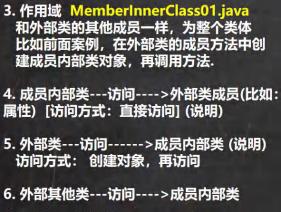
|  |
| --- |
| package com.hspedu.innerclass;  public class InnerClassExercise02 {  public static void main(String[] args) {  /\*  1.有一个铃声接口 Bell ，里面有个ring 方法。(右图)  2.有一个手机类 Cellphone ，具有闹钟功能 alarmClock ，参数是 Bell 类型(右图)  3.测试手机类的闹钟功能，通过匿名内部类(对象)作为参数，打印：懒猪起床了  4.再传入另一个匿名内部类(对象) ，打印：小伙伴上课了  \*/  CellPhone cellPhone = new CellPhone();  //老韩解读  //1. 传递的是实现了 Bell 接口的匿名内部类 InnerClassExercise02$1  //2. 重写了 ring  //3. Bell bell = new Bell() {  // @Override  // public void ring() { |

|  |
| --- |
| // System.out.println("懒猪起床了");  // }  // }  cellPhone.alarmClock(new Bell() {  @Override  public void ring() {  System.out.println("懒猪起床了");  }  });  cellPhone.alarmClock(new Bell() {  @Override  public void ring() {  System.out.println("小伙伴上课了");  }  });  }  }  interface Bell{ //接口  void ring();//方法  }  class CellPhone{//类  public void alarmClock(Bell bell){//形参是 Bell 接口类型  System.out.println(bell.getClass());  bell.ring();//动态绑定  } |

|  |
| --- |
| } |

10.9.9 成员内部类的使用







代码：

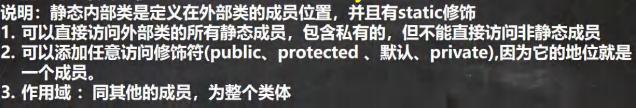
|  |
| --- |
| package com.hspedu.innerclass;  public class MemberInnerClass01 { |

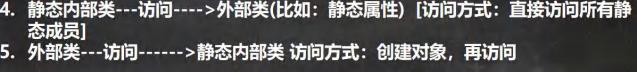
|  |
| --- |
| public static void main(String[] args) {  Outer08 outer08 = new Outer08();  outer08.t1();  //外部其他类，使用成员内部类的三种方式  //老韩解读  // 第一种方式  // outer08.new Inner08(); 相当于把 new Inner08()当做是 outer08 成员  // 这就是一个语法，不要特别的纠结.  Outer08.Inner08 inner08 = outer08.new Inner08();  inner08.say();  // 第二方式 在外部类中，编写一个方法，可以返回 Inner08 对象  Outer08.Inner08 inner08Instance = outer08.getInner08Instance();  inner08Instance.say();  }  }  class Outer08 { //外部类  private int n1 = 10;  public String name = "张三";  private void hi() {  System.out.println("hi()方法...");  } |

|  |
| --- |
| //1.注意: 成员内部类，是定义在外部内的成员位置上  //2.可以添加任意访问修饰符(public 、protected 、默认、private), 因为它的地位就是一个成员  public class Inner08 {//成员内部类  private double sal = 99.8;  private int n1 = 66;  public void say() {  //可以直接访问外部类的所有成员，包含私有的  //如果成员内部类的成员和外部类的成员重名，会遵守就近原则.  // ，可以通过 外部类名.this.属性 来访问外部类的成员  System.out.println("n1 = " + n1 + " name = " + name + " 外部类的 n1=" + Outer08.this.n1);  hi();  }  }  //方法，返回一个 Inner08 实例  public Inner08 getInner08Instance(){  return new Inner08();  }  //写方法  public void t1() {  //使用成员内部类  //创建成员内部类的对象，然后使用相关的方法  Inner08 inner08 = new Inner08();  inner08.say(); |

|  |
| --- |
| System.out.println(inner08.sal);  }  } |

10.9. 10 静态内部类的使用 StaticInnerClass01.java





5。外部其他类…访问…-静态内部类



代码:

|  |
| --- |
| package com.hspedu.innerclass;  public class StaticInnerClass01 {  public static void main(String[] args) {  Outer10 outer10 = new Outer10();  outer10.m1();  //外部其他类 使用静态内部类 |

|  |
| --- |
| //方式 1  //因为静态内部类，是可以通过类名直接访问(前提是满足访问权限)  Outer10.Inner10 inner10 = new Outer10.Inner10();  inner10.say();  //方式 2  //编写一个方法，可以返回静态内部类的对象实例.  Outer10.Inner10 inner101 = outer10.getInner10();  System.out.println("============");  inner101.say();  Outer10.Inner10 inner10\_ = Outer10.getInner10\_();  System.out.println("\*\*\*\*\*\*\*\*\*\*\*\*");  inner10\_.say();  }  }  class Outer10 { //外部类  private int n1 = 10;  private static String name = "张三";  private static void cry() {}  //Inner10 就是静态内部类  //1. 放在外部类的成员位置  //2. 使用 static 修饰  //3. 可以直接访问外部类的所有静态成员，包含私有的，但不能直接访问非静态成员  //4. 可以添加任意访问修饰符(public 、protected 、默认、private), 因为它的地位就是一个成员  //5. 作用域 ：同其他的成员，为整个类体 |

|  |
| --- |
| static class Inner10 {  private static String name = "韩顺平教育";  public void say() {  //如果外部类和静态内部类的成员重名时，静态内部类访问的时，  //默认遵循就近原则，如果想访问外部类的成员，则可以使用 （外部类名.成员）  System.out.println(name + " 外部类 name= " + Outer10.name);  cry();  }  }  public void m1() { //外部类---访问------>静态内部类 访问方式：创建对象，再访问  Inner10 inner10 = new Inner10();  inner10.say();  }  public Inner10 getInner10() {  return new Inner10();  }  public static Inner10 getInner10\_() {  return new Inner10();  }  } |

10.9. 11 课堂测试题

|  |
| --- |
| public class Test {//外部类  public Test() {//构造器  Inner s1 = new Inner();  s1.a = 10;  Inner s2 = new Inner();  System.out.println(s2.a);  }  class Inner { //内部类，成员内部类  public int a = 5;  }  public static void main(String[] args) {  Test t = new Test();  Inner r = t.new Inner();//5  System.out.println(r.a);//5  }  } |

10. 10卖油翁和老黄牛

