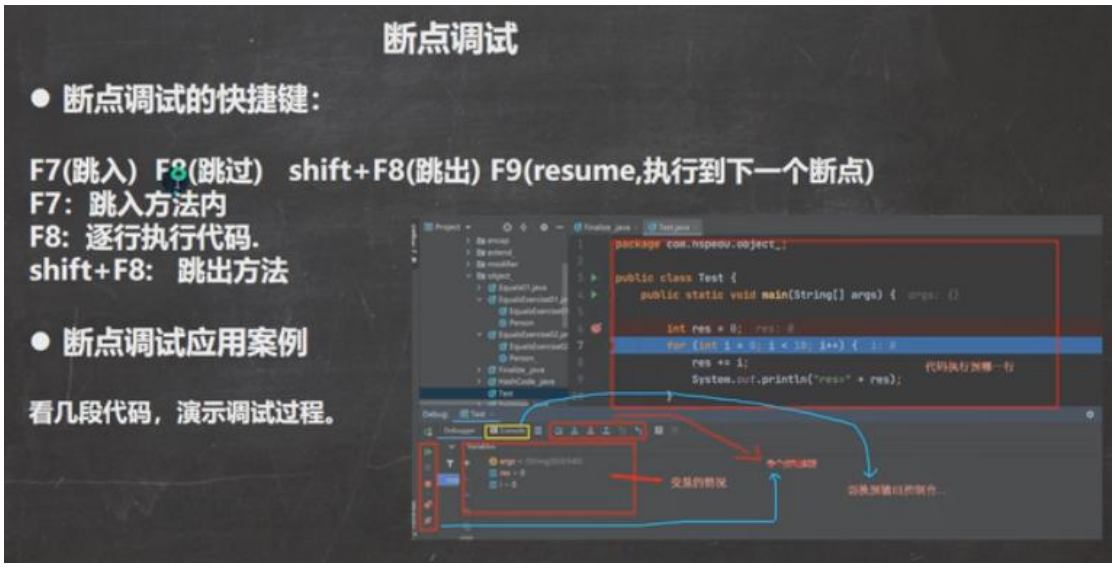
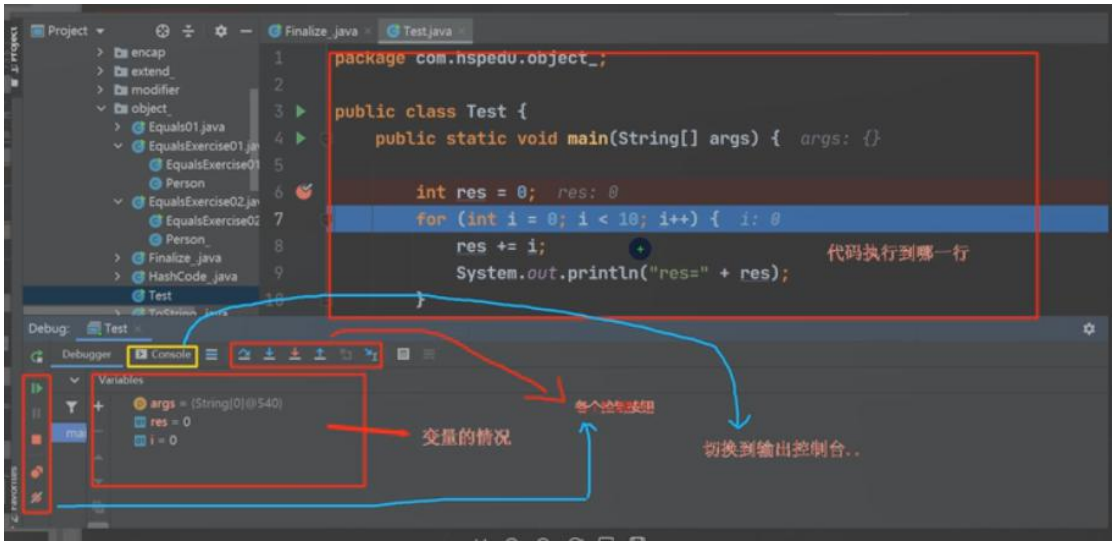
finalize 方法

1. 当对象被收回时，系统自动调用该对象的 finalize 方法，子类可以重写该方法，做一些释放资源的操作；
2. 什么时候回收：当某个对象没有任何引用时，则 JVM 就认为这个对象是一个垃圾对象，就会使用垃圾回收机制来销毁对象，在销毁该对象前，会先调用 finalize 方法；
3. 垃圾回收机制的调用，是由系统来决定（即有自己的 GC 算法），也可以通过 System.gc()主动触发垃圾回收机制，测试：Car [name]。

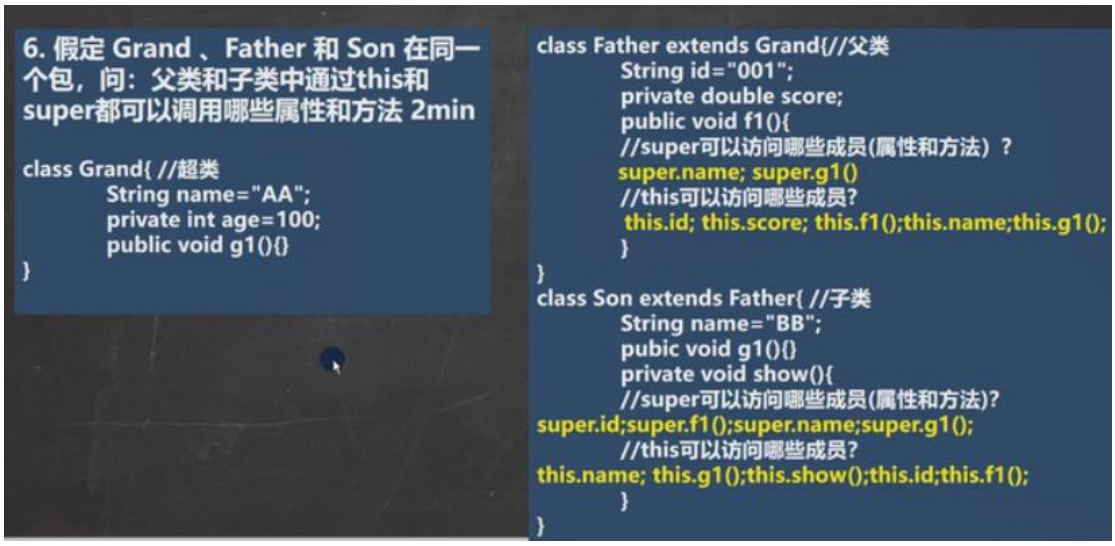
断点测试（deBug）：

1. 断点测试是指在程序的某一行设置一个断点，调试时，程序运行到这一行就会停住，然后你可以一步一步往下调试，调试过程中可以看各个变量当前的值，出错的话，调试到出错的代码行即显示错误，停下。进行分析进而找到这个 Bug；
2. 断点调试是程序必须掌握的技能；
3. 断点调试也能帮助查看 java 底层源代码的执行过程。

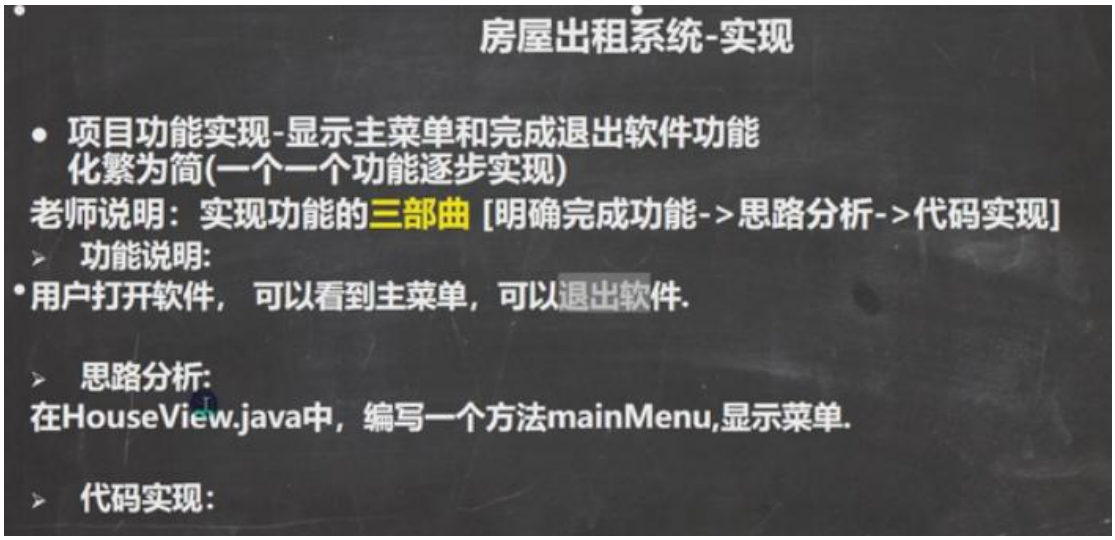




在进行 DeBug 过程中可以在自身的代码中设置断点，也可以在系统的代码中设置断点



```
class Person { //父类
 public void run() {System.out.println("person run"); }
 public void eat() {System.out.println("person eat"); }
}
class Student extends Person {//子类
 public void run() {System.out.println("student run"); }
 public void study() {System.out.println("student study.."); }
}
//向上转型: 父类的引用指向子类对象
Person p = new Student();
p.run();//student run
p.eat();//person eat
//向下转型: 把指向子类对象的父类引用，转成指向子类对象的子类引用
Student s = (Student)p;
s.run();//student run
s.study();//student study..
s.eat();//
```



面向对象高级阶段
类变量
（1）static 变量是同一个类所有对象共享（2）static 类变量在类加载的时候就生成了
类变量的定义：
　　类变量也叫静态变量/静态属性，是该类的所有对象共享的变量，任何一个该类的对象去访问它时，取到的都是相同的值，同样任何一个该类的对象去修改它时，修改的也是同一个变量。

如何定义类变量：
语法：
访问修饰符 static 数据类型 变量名; [推荐]
static 访问修饰符 数据类型 变量名;

如何访问类变量：
类名.类变量名； [推荐使用]
或者 对象名.类变量名 [静态变量的访问修饰符的访问权限和范围和普通属性是一样的]

类变量的使用细节：
1. 什么时候需要用类变量：当我们需要让某个类的所有对象都共享一个变量时，就可以考虑使用类变量（静态变量）；
2. 类变量与实例变量（普通属性）的区别：类变量是该类的所有对象共享的，而实例变量是每个对象独享的；
3. 加上 static 称为类变量或静态变量，否则称为实例变量/普通变量/非静态变量
4. 类变量可以通过 类名.类变量名 或者 对象名.类变量名 来访问，但 java 设计者推荐使用 类名.类变量名 来进行访问[前提是满足访问修饰符的访问权限和范围]；
5. 实例变量不能通过 类名.变量名 的方式访问
6. 类变量是在类加载时就初始化了，也就是说，即使你没有创建对象，只要类加载了，就可以使用类变量了；
7. 类变量的生命周期是随着类的加载开始，随着类的消亡而消亡。

类方法基本介绍：
类方法也叫静态方法，其形式如下：
访问修饰符 static 数据返回类型 方法名（）{}   [推荐]
static 访问修饰符 数据返回类型 方法名（）{}

类方法的调用：
类名.类方法名 或者 对象名.类方法名 [前提是 满足访问修饰符的访问权限]

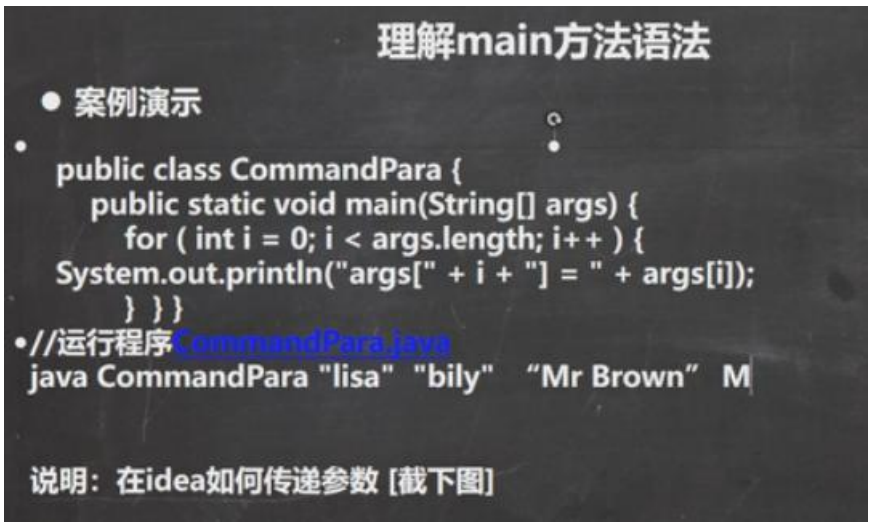
类方法使用的注意事项和细节：
1）类方法和普通方法都是随着类的加载而加载，将结构信息存储在方法区：类方法中无 this 的参数，普通方法中隐含着 this 的参数；
2）类方法可以通过类名调用，也可以通过对象名调用；
3）普通方法和对象有关，需要通过对象名调用，比如对象名.方法名(参数)，不能通过类名调用；
4）类方法中不允许使用和对象有关的关键字，比如 this 和 super，普通的 成员方法可以；
5）类方法（静态方法）中只能访问静态变量或静态方法；
6）普通成员方法，既可以访问普通变量（方法），也可以访问静态变量（方法）。

总结：（1）静态方法只能访问静态的成员；（2）非静态的方法，可以访问静态成员和非静态的成员；（3）在编写代码时，仍然要遵守访问权限的规则。
只有 new 了一个对象之后该类才会走构造器

main 方法的语法：
解释 main 方法的形式：public static void main(String[] args){}

1. java 虚拟机需要调用类的 main()方法，所以该方法的访问权限必须是 public；
2. java 虚拟机在执行 main()方法时不必创建对象，所以该方法必须是 static；
3. 该方法接收 String 类型的数组参数，该数组中保存执行 java 命令时传递给所运行的类的参数，接收参数；
4. java 执行的程序 参数 1 参数 2 参数 3；
5. main 方法是由 java 虚拟机调用的。

特别提示：
（1）在 main()方法，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）但是不能直接去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



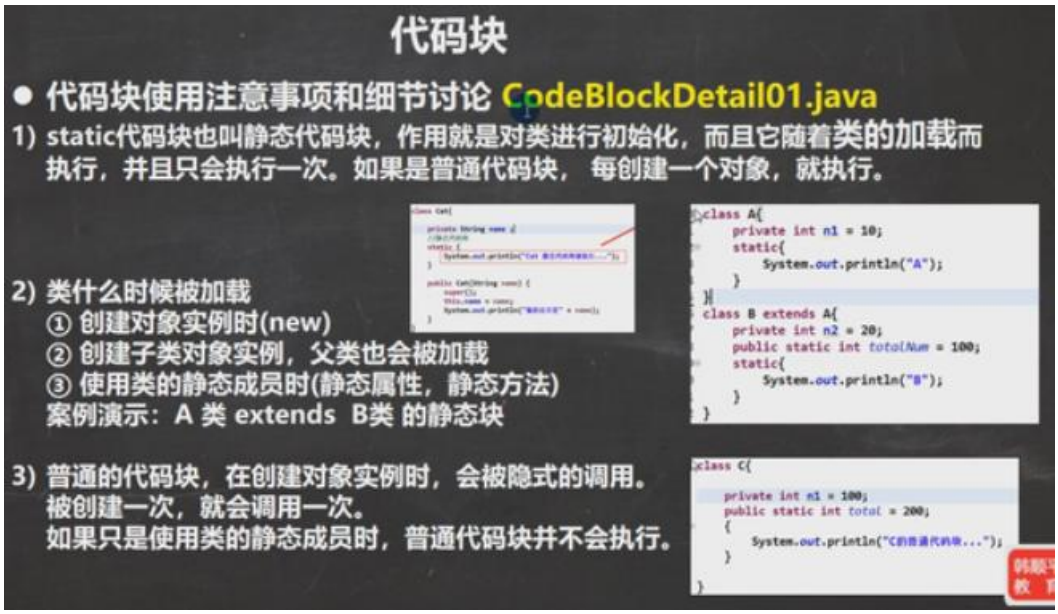
代码块：代码化块又称为初始化块，属于类中的成员[即是类的一部分]，类似于方法，将逻辑语句封装在方法体中，通过{}包围起来；

但和方法不同，没有方法名，没有返回，没有参数，只有方法体，而且不用通过对象或类显式调用，而是加载类时或创建对象时隐式调用；

基本语法：
[修饰符]{
　　代码
};

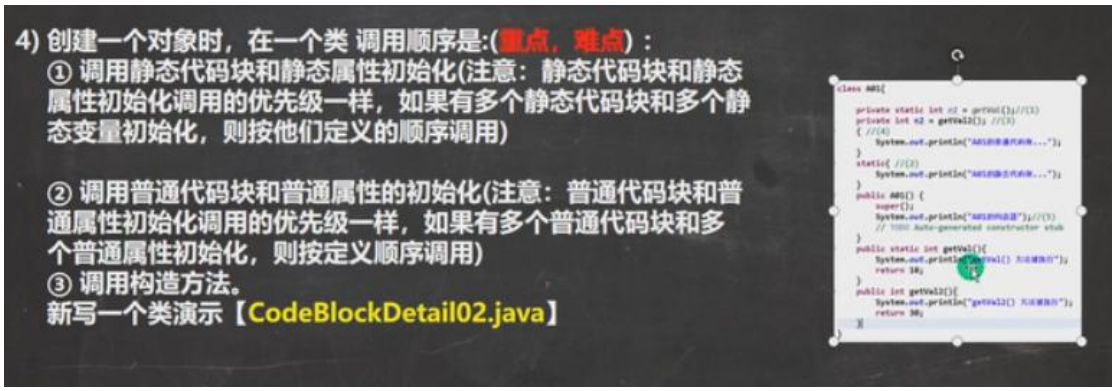
注意：
1）修饰符可选，要写的话也只能写 static
2）代码块分为两类，使用 static 修饰的叫静态代码块，没有 static 修饰的叫普通代码块
3）逻辑语句可以为任何逻辑语句（输入、输出、方法调用、循环、判断等）
4）;号可以写上，也可以省略；
5）代码块调用的顺序优先于构造器。

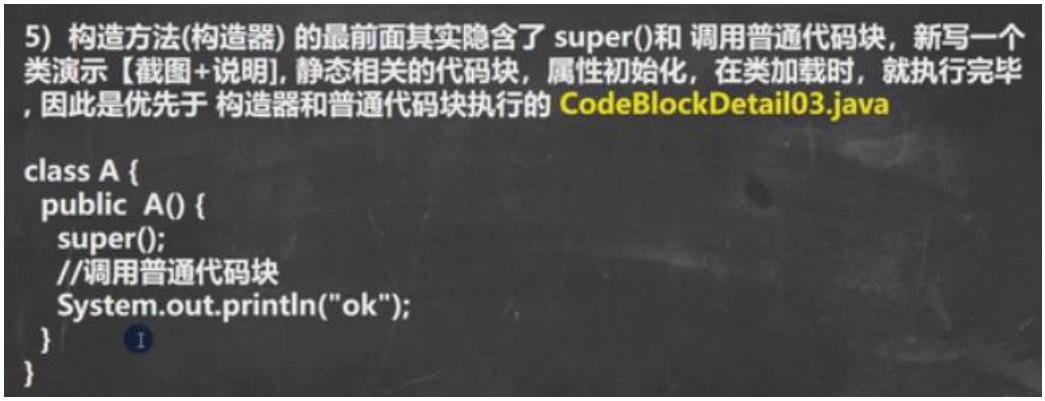
```
private String director;

//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
{
    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) {

    this.name = name;
}

public Movie(String name, double price) {
```



小结：

1. static 代码块是类加载时，执行，只会执行一次
2. 普通代码块是在创建对象时调用的，创建一次，调用一次
3. 类加载的 3 种情况，需要记住



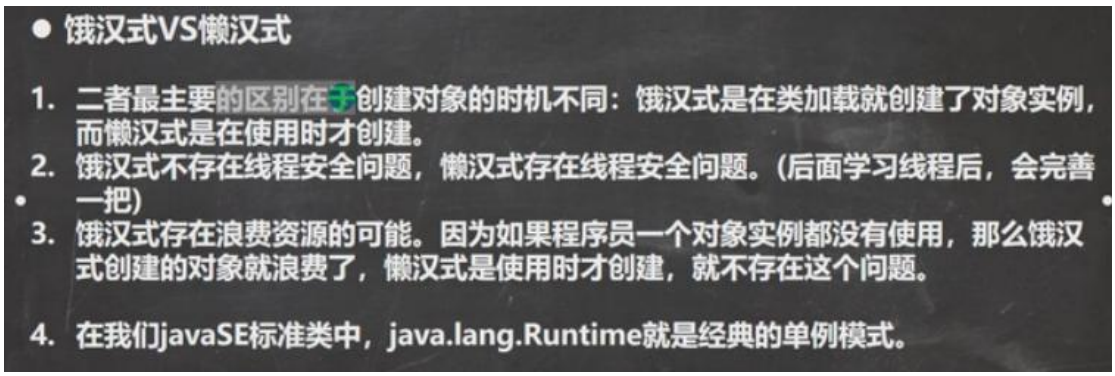



```
public class CodeBlockDetail04 {
public static void main(String[] args) {
//老师说明
//(1) 进行类的加载
//1.1 先加载 父类 A02 1.2 再加载 B02
//(2) 创建对象
//2.1 从子类的构造器开始
//new B02();//对象 new C02();
}
}
class A02 { //父类
private static int n1 = getVal01();
```

```
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
}
}
```

设计模式：
1. 静态方法和属性的经典使用；
2. 设计模式是在大量的时间中总结和理论化之后优选的代码结构、编程风格以及解决问题的思考方式。
单例设计模式：单例（单个的实例）
1. 所谓类的单例设计模式，就是采取一定的方法保证在整个的软件系统中，对某个类只能存在一个对象实例，并且该类只提供一个区的其对象实例的方法
2. 单例模式有两种方式：1）饿汉式 2）懒汉式
饿汉式可能造成创建了对象但是没有使用
单例模式——饿汉式步骤如下：
（1）构造器私有化->防止对象直接 new
（2）类的内部创建对象（该对象是 static 的）
（3）向外暴露一个静态的公共方法（为了在外部能够使用该对象）
（4）代码实现




```
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.");
```

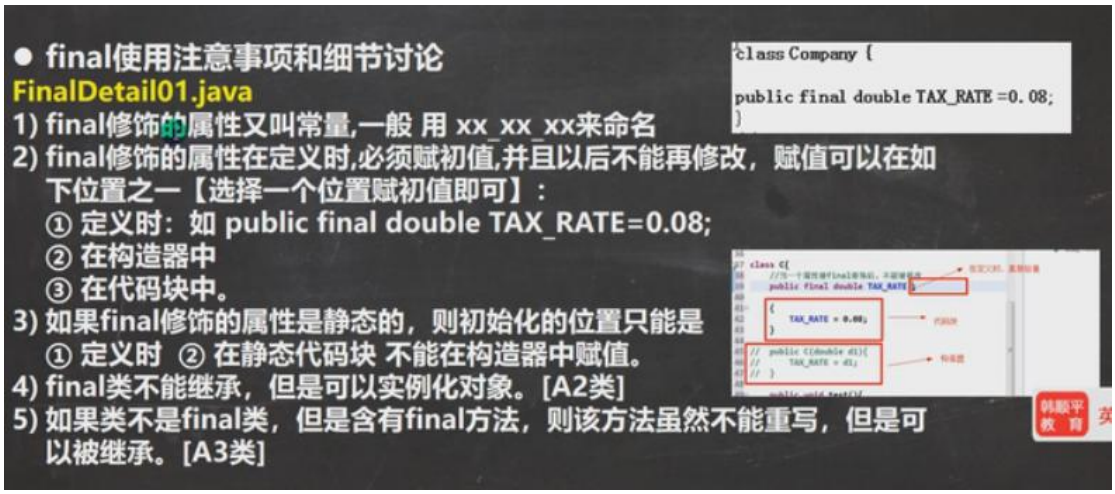
```
this.name = name;
}
public static GirlFriend getInstance() {
return gf;
}
@Override
public String toString() {
return "GirlFriend{" +
"name='" + name + '\'' +
'}';
}
}
```

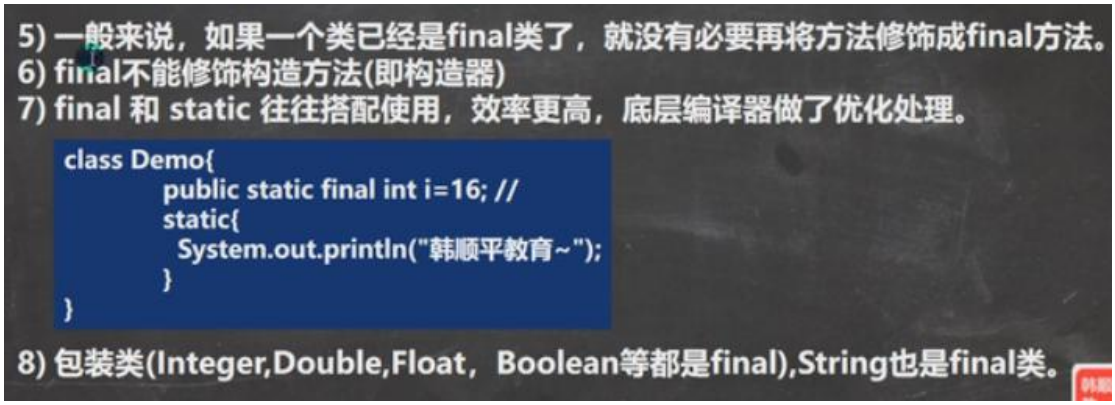
==========

```
/**
* 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 + '\'' +
'}';
}
}
```

final 关键字
final 可以修饰类、属性、方法和局部变量
在某些情况下，可能会有以下需求，就会使用到 final：
1）当不希望类被继承时，可以用 final 修饰；
2）当不希望父类的某个方法被子类覆盖/重写（override）时，可以用 final 关键字修饰；
3）当不希望类的某个属性的值被修改，可以用 final 修饰
4）当不希望某个局部变量被修改，可以用 final 修饰





抽象类

当父类的某些方法，需要声明，但是又不确定如何实现时，可以将其声明为抽象方法，那么这个类就是抽象类

介绍：

1）用 abstract 关键字来修饰一个类时，这个类就叫抽象类：

访问修饰符 abstract 类名{}

2）用 abstract 关键字来修饰一个方法时，这个方法就是抽象方法：

访问修饰符 abstract 返回类型 方法名（参数列表）;//没有方法体

3）抽象类的价值更多作用是在于设计，是设计者设计好后，让子类继承并实现抽象类()

抽象类使用的注意事项和细节讨论：

1）抽象类不能被实例化；

2）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；

3）一旦类包含了 abstract 方法，则这个类必须声明为 abstract；

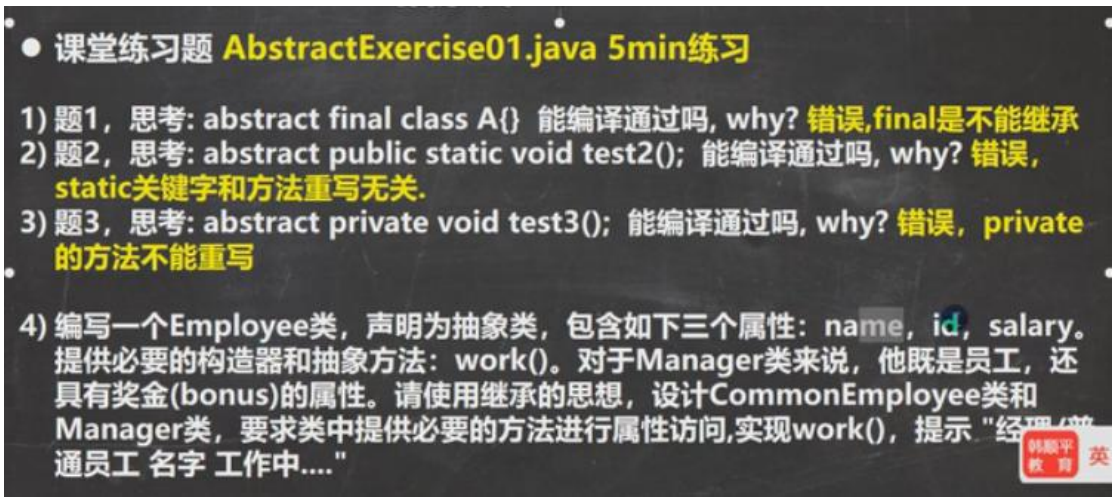
4）abstract 只能修饰类和方法，不能修饰属性和其他的；

5）抽象类可以有任意成员[抽象类的本质还是类]，比如：非抽象方法、构造器、静态属性等；

6）抽象方法不能有主体，即不能实现；

7）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；

8）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。




```
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() ;
}
```

```
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;
}


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");
}
}
```

抽象类的最佳设计模式——模板设计模式

```
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));
}
}

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;
}
}
}

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();
}
}
```

接口
实现接口的实质，就是实现接口的方法
通过接口来调用方法

接口就是给出一些没有实现的方法，封装到一起，到某个类要使用的时候，再根据具体情况把这些方法写出来。
语法：

```
interface 接口名{
        //属性
        //方法（1.抽象方法 2.默认实现方法 3.静态方法）
}
class 类名 implements 接口{
        自己属性;
        自己方法;
        必须实现的接口的抽象方法;
}
```

在接口中可以省略 abstract 方法

接口的注意事项和使用细节：
1）接口不能被实例化
2）接口中所有的方法是 public 方法，接口中的抽象方法可以不用 abstract 修饰
3）一个普通类实现接口就必须将该类的所有方法都实现，可以使用 alt+enter 快速实现
4）抽象类实现接口时，可以不用实现接口的方法
5）一个类同时可以实现多个接口
6）接口中的属性，只能是 final 的，而且是 public static final 修饰符。比如：int a=1;实际上是 public static final int a=1;（必须初始化）
7）接口中属性的访问形式：接口名.属性名
8）一个接口不能继承其他的类，但是可以继承多个别的接口
9）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

接口的多态特性：
1）多态参数
2）多态数组
3）接口存在多态传递现象