

图 1 计算机硬件结构图

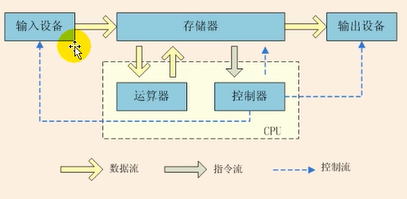


图 2 冯诺依曼体系结构



IT定律之计算机行业发展规律

* 摩尔定律
* 安迪-比尔定律
* 反摩尔定律

计算机的存储能力是以字节和多字节来衡量的。例如：

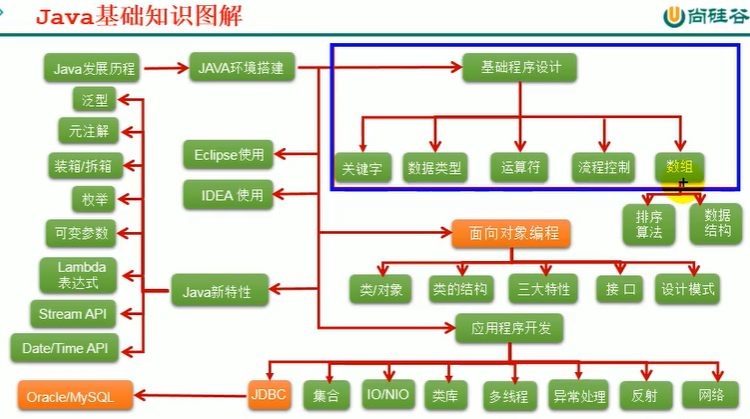
千字节（kilobyte,KB） = 1024B

兆字节（megabyte，MB） = 1024KB

千兆字节（gigabyte，GB） = 1024MB

万亿字节（terabyte，TB） = 1024GB

计算机中最基本的存储单元是字节（byte），每个字节由8个bit构成。



软件：一系列按照特定顺序组织的计算机数据和指令的集合。

分为：系统软件 和 应用软件

系统软件：Windows，os，Linux，unix，Android，iOS......

应用软件：用到的软件都可算



计算机语言：

第一代：机器语言，指令以二进制代码形式存在。

第二代：汇编语言，使用助记符表示一条机器指令。

第三代：高级语言，面向过程：C、Pascal、Fortran；面向对象：Java；C++过渡；

Dos命令框写hello world步骤：

Java程序编写---编译---运行的过程

编写：将编写的Java文件保存在“.java”结尾的源文件中；

编译：使用javac.exe命令编译我们的java源文件； 格式：javac 源文件名.java

运行：使用java.exe命令解释运行我们的字节码文件。 格式：java 类名



文档注释：注释内容可以被JDK提供的工具javadoc所解析，生成一套以网页文件形式体现的该程序的说明文档。

格式：/\*\*

@version 指定源文件的版本

\*/

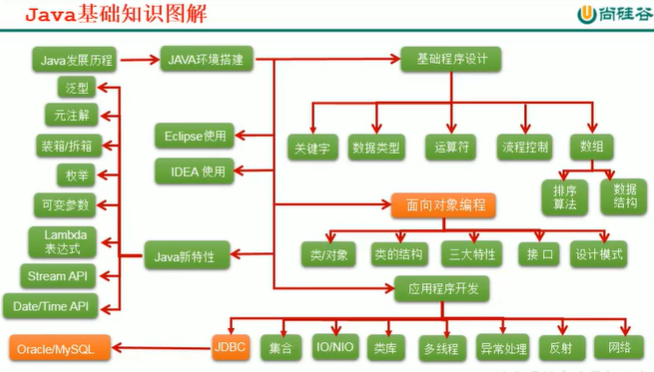
操作方式：javadoc -d mydoc -author -version HelloWorld.java

(进入代码所在目录)

Java API（Application Programming Interface，应用程序编程接口）文档

总结：

在一个Java源文件中可声明多个class，但最多只能有一个类声明为public，且要求声明为public的类的类名必须与源文件名相同。



项目一：家庭收支记账软件(讲完流程控制时可做)

项目二：客户信息管理软件(讲完面向对象第四章可做)

项目三：开发团队人员调度软件(第七章异常处理后可做)

附加项目一：银行业务管理软件(增量式开发，同三)

附加项目二：单机考试管理软件(I/O操作，讲完十一章I/O流后可做)

Java语言的特点：

>面向对象性：

两个要素：类和对象

三个特征：封装，继承，多态

>健壮性： ①去除了C语言中的指针；

②自动的垃圾回收机制-->仍然会出现内存溢出、内存泄漏

>跨平台性：

Java中名称命名规范：

* 包名：多单词组成时所有字母都小写：xxyyzz；
* 类名、接口名：多单词组成时，所有单词的首字母大写：XxxYyyZzz；
* 变量名、方法名：多单词组成时，第一个单词首字母小写，第二个单词开始

每个单词首字母大写：xxxYyyZzz；

* 常量名：所有字母都大写。多单词时每个单词用下划线连接：XXX\_YYY\_ZZZ。

基本数据类型之间的转换：

**自动类型提升**：当容量小的数据类型的变量与容量大的数据类型的变量做运算时，结果自动提升为容量大的数据类型。

byte、char、short -> short -> int -> long -> float -> double

特别地：当byte、char、short三种类型的变量做运算时，结果为int型

强制类型转换：

**说明：**此时的容量大小指的是，表示数的范围的大和小。比如：float容量要大于long的容量

# 程序运行

# 如何从键盘获取数据：使用Scanner类（demo1）

具体实现步骤：

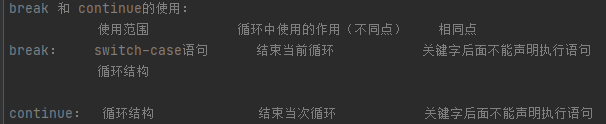
1、导包：import java.util.Scanner

2、Scanner实例化: Scanner scan = new Scanner(System.in);

3、调用Scanner类的相关方法，来获取指定类型的变量

注意：需要根据相应的方法，来输入指定类型的值，如果输入的类型与要求的类型不匹配时，会报异常：InputMisMatchException，导致程序终止。

# break和continue的使用：



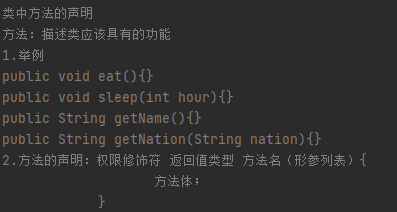
# 随机值获取：[10，99] 公式：int(Math.random() \* (99 - 10 + 1) + 10)

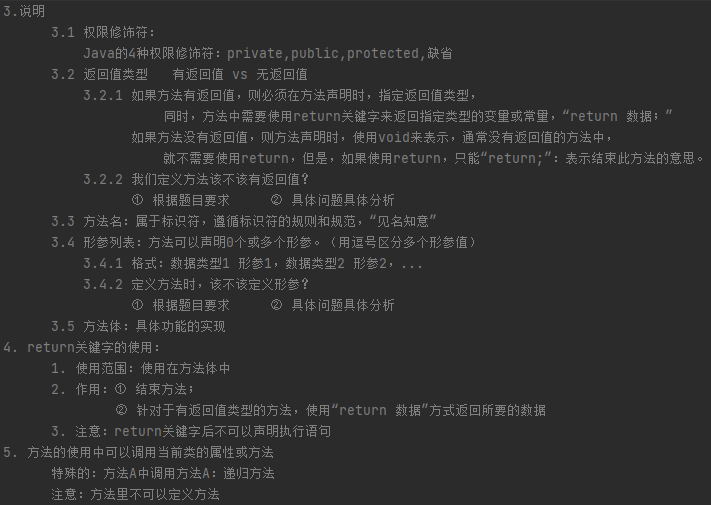
# 数组中的异常

(1) 数组角标越界异常：ArrayIndexOutOfBoundsException

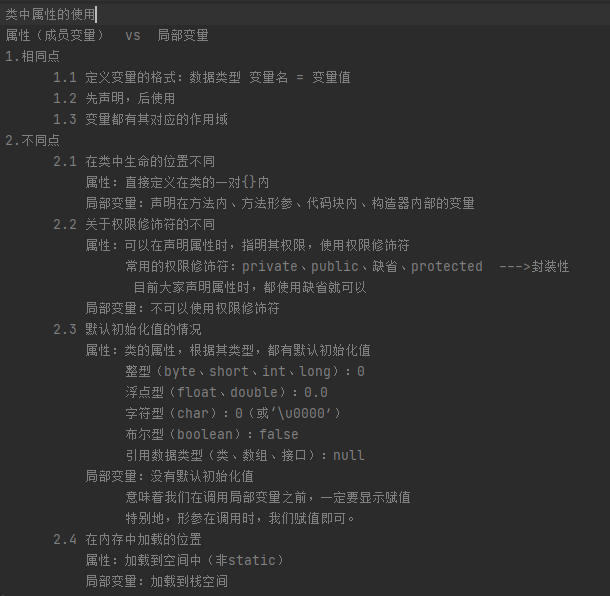
(2) 空指针异常：NullPointerException

# 类中方法的声明（demo3----1）

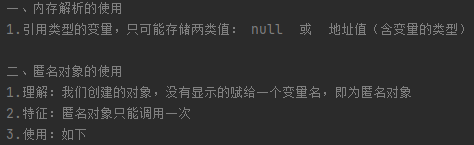




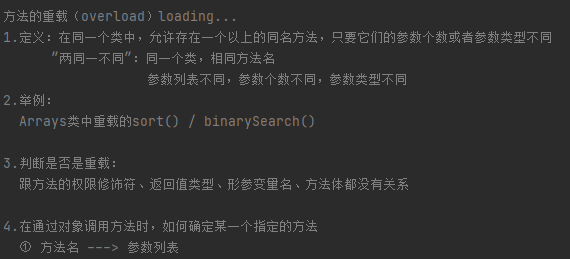
# 类中属性的使用（demo3----3）

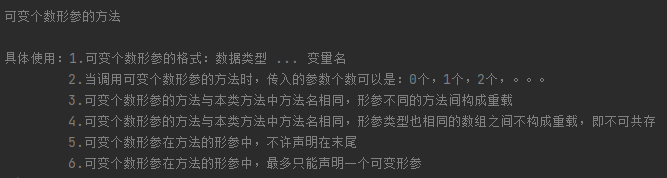


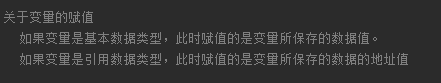
# 内存解析和匿名对象的使用（demo5----d）

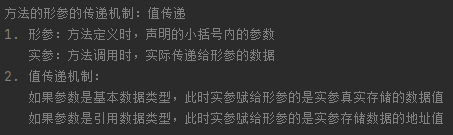


# 方法的重载（demo6----a c d d）

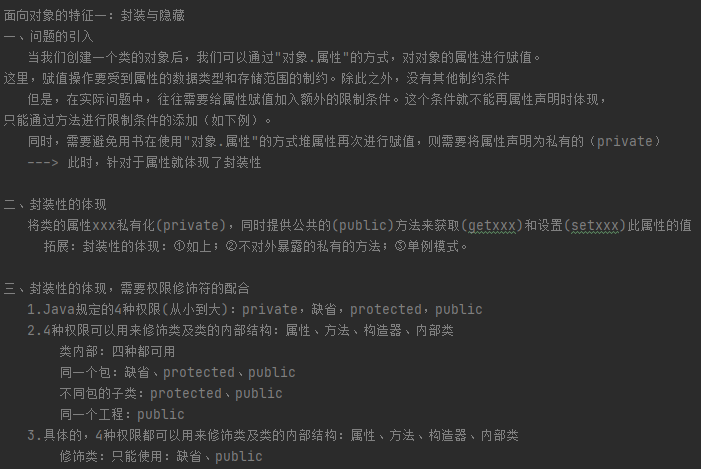




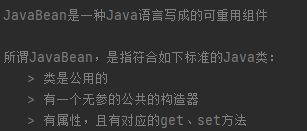
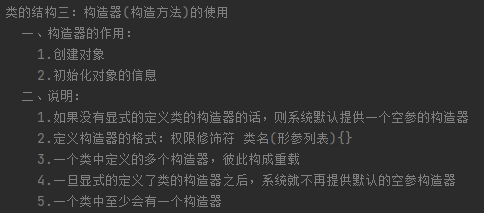




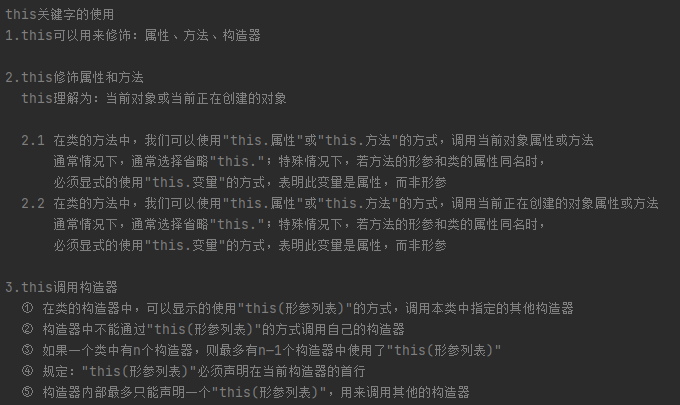
# 面向对象（demo7）



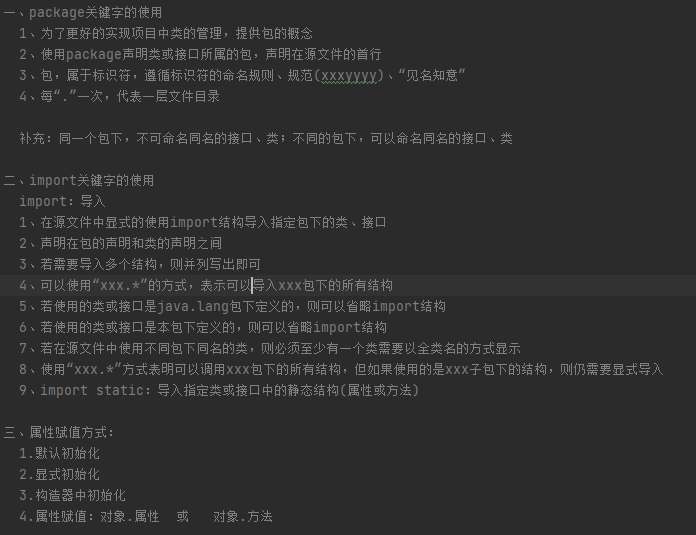
# 构造器的使用（demo8----a c）



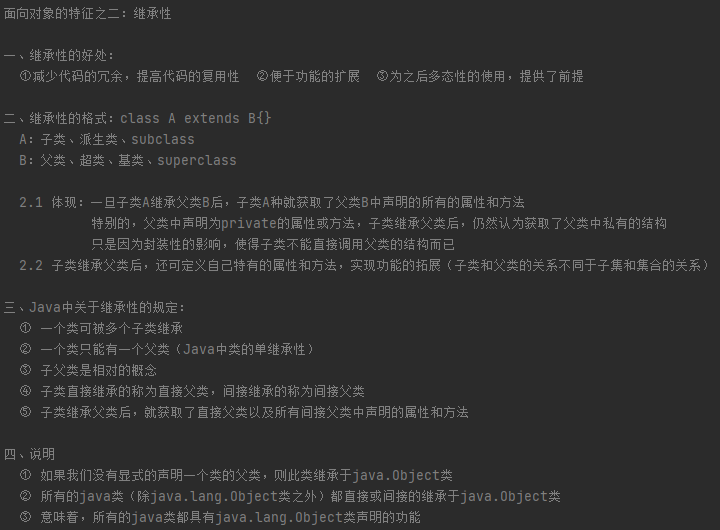
# this关键字的使用（demo9----a）



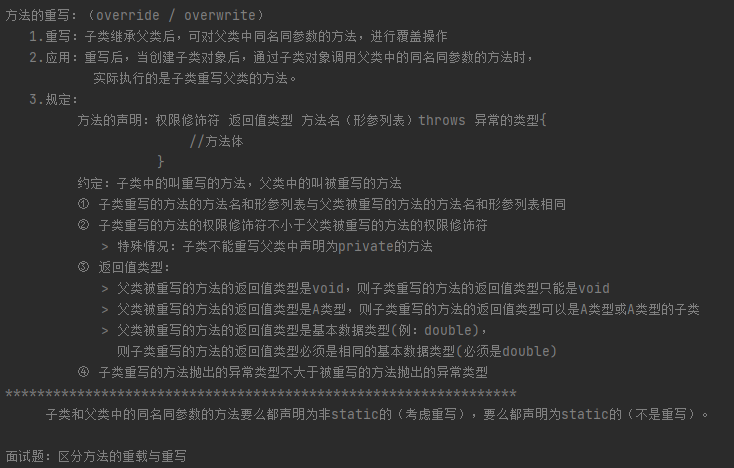
# package和import关键字的使用（demo12）



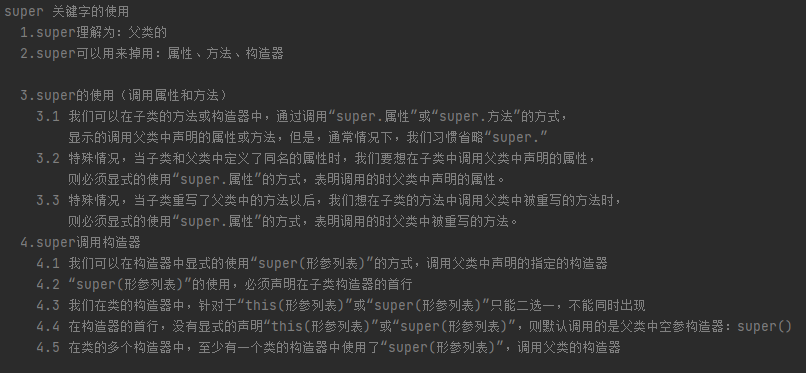
# 面向对象的特征之二：继承性（part2—demo1）



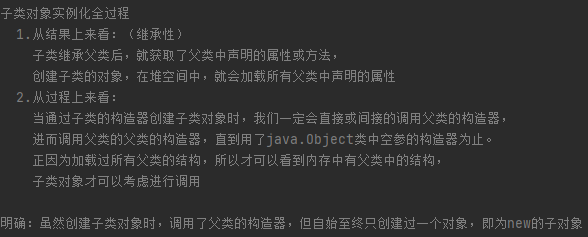
# 方法的重写（override / overwrite）（part2—demo5）

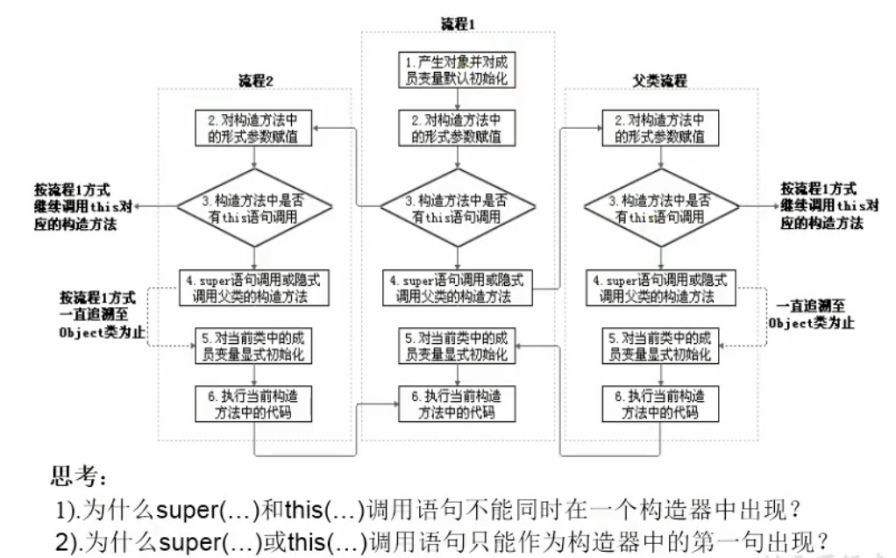


# super关键字的使用（part2—demo8）



# 子类对象实例化全过程（part2—demo8—Instance）

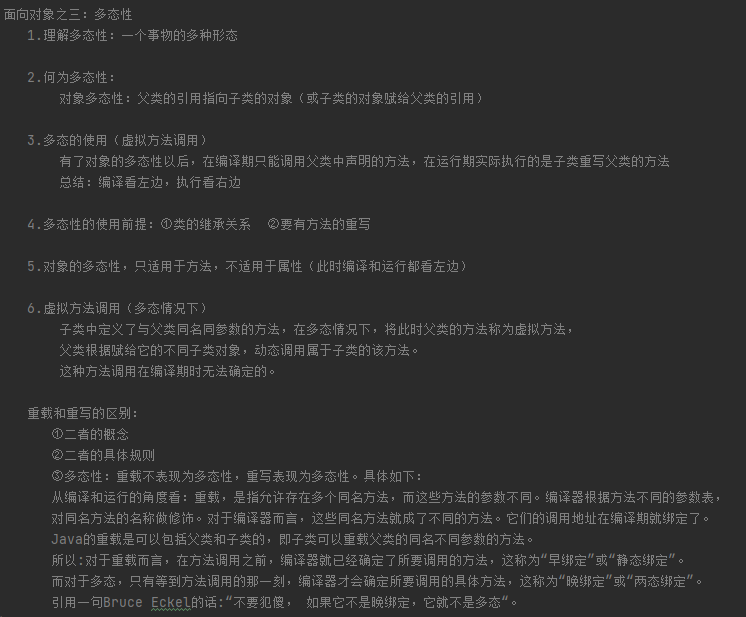




无论通过哪个构造器创建子类对象，需要保证先初始化父类

目的：当子类继承父类后，”继承“父类中所有的属性和方法，因此子类有必要知道父类如何为对象进行初始化。

# 面向对象的特征之三：多态性（part3—demo1）



# 关于向上转型和向下转型（part3—demo2）

（1）向上转型：多态

（2）向下转型：

①为什么使用向下转型：

有了对象的多态性后，内存中实际上是加载了子类特有的属性和方法的，但由于变量声明为父类类型，导致编译时，只能调用父类中声明的属性和方法，子类所特有的属性和方法不能调用。如何才能调用子类特有的属性和方法？向下转型：使用强制类型转换符

②如何实现向下转型：使用强制类型转换

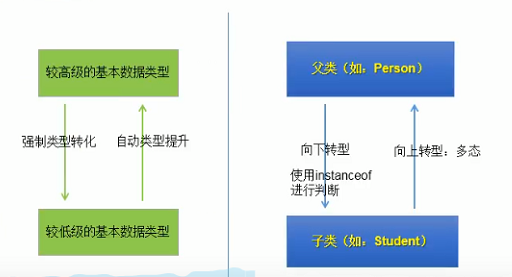
③使用时的注意点：

1. 使用强转时，可能出现ClassCastException(类型转换)的异常；
2. 为了避免在向下转型时出现ClassCastException的异常，我们在向下转型前先进行instanceof的判断，一旦返回true，就进行向下转型；若返回false，不进行向下转型。

④instanceof的使用：

* a instanceof A：判断对象a是否是类A的实例。若是返回true，否则返回false。
* 如果a instanceof A返回true，则a instanceof B也返回true，其中B是A的父类。
* 要求a所属的类与类A必须是子类和父类的关系，否则编译错误。

⑤图示：



# 面试题 （part3—demo2）

*（1）谈谈对多态性的理解？*

*①实现代码的通用性*

*②Object类中定义的public boolean equals(Object obj){ }*

*JDBC：使用java程序操作（获取数据库连接、增删改查）数据库（MySQL、Oracle、DB2、SQL Server）*

*③抽象类、接口的使用肯定体现了多态性（抽象类、接口不能实例化）*

*（2）多态时编译时行为还是运行时行为？*

# java.lang.Object类的说明：（part3—demo3）

* Object类是所有Java类的根父类
* 如果在类的生命中未使用extends关键字指明其父类，则默认父类为java.lang.Object类
* Object类中的功能（属性、方法）具有通用性
* 属性：无
* 方法：equals() / toString() / getClass() / hashCode() / clone() /finalize() / wait() / notify() / notifyAll()
* Object类中只声明了一个空参的构造器

# equals()方法（part3—demo3）

（1）equals()的使用

1. 是一个方法，而非运算符
2. 只能适用于引用数据类型
3. Object类中equals()的定义：

public boolean equals(Object obj) {

return (this == obj);

}

说明：Object类中定义的equals()和 == 的作用是相同的，比较两个对象的地址值是否相同，即两个引用是否指向同一个实体。

1. 像String、Date、File、包装类等都重写了Object类中的equals()方法。

* 重写后，比较的不是两个引用的地址是否相同，而是比较两个对象的“实体内容”是否相同
* 通常情况下，自定义的类如果使用equals()的话，指比较两个对象的“实体内容”是否相同。

1. 此时需要对Object类中的equals()进行重写。
2. 重写的原则：比较两个对象的实体内容(即name和age)是否相同

●对称性：如果x.equals(y)返回是"true"，那么yequals(x)也应该返回是"true"。

●自反性∶ x.equals(x)必须返回是"true"。

●传递性∶如果x.equals(y)返回是"true"，且y.equals(z)返回是"true"，则z.equals(x)也应该返回是"true"。

●一致性∶如果x.equals(y)返回是"true"，只要x和y内容一直不变，不管重复x.equals(y)多少次，返回都是"true"。

● 任何情况下，x.equals(null)，永远返回是"false";

x.equals（和x不同类型的对象）永远返回是"false"。

（2）如何重写equals()

①手动重写

@Override

public boolean equals(Object obj) {

if (this == obj){

return true;

}

if (obj instanceof Customer){

Customer cust = (Customer) obj;

// //比较两个对象的属性是否相同

// if (this.name.equals(cust.name) && this.age == cust.age){

// return true;

// }else {

// return false;

// }

return this.name.equals(cust.name) && this.age == cust.age;

}else {

return false;

}}

②开发中实现

@Override

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Customer customer = (Customer) o;

return age == customer.age && name.equals(customer.name);

}

（3）==运算符的使用

* 可以使用在基本数据类型变量和引用数据类型变量中
* 如果比较的是基本数据类型变量，比较两个变量保存的数据是否相等。

（类型不一定相同，boolean类型除外）

* 如果比较的是引用数据类型变量，比较两个对象的地址值是否相同，即两个引用是否指向同一个对象实体

补充：== 符号使用时，必须保证符号左右两边的变量类型一致

# toString()方法（part3—demo4）

（1）toString()的使用

* 当我们输出一个对象的引用时，实际上就是调用当前对象的toString();
* Object类中toString()的定义：

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

* 像String、Date、File、包装类等都重写了Object类中的toString()方法，使得再调用对象的toString()时，返回“实体内容”信息
* 自定义类也可重写toString()方法，当调用此方法时，返回对象的“实体内容”

（2）如何重写toString()

@Override

public String toString() {

return "Customer{name=" + name + '\'' +", age=" + age +'}';

}

# 面试题

*（1）final、finally、finalize的区别？*

*（2） == 和equals()的区别*

*①== 既可以比较基本类型也可以比较引用类型。对于基本类型就是比较值，对于引用类型就是比较内存地址*

*②equals的话，它是属于java.lang.Object类里面的方法，如果该方法没有被重写过默认也是==;我们可以看到String等类的equals方法是被重写过的，而且String类在日常开发中用的比较多，久而久之，形成了equals是比较值的错误观点。*

*③具体要看自定义类里有没有重写Object的equals方法来判断。*

*④通常情况下，重写equals方法，会比较类中的相应属性是否都相等。*

# Java中的JUnit单元测试

* 选中当前工程-右键选择∶build path- add libraries - Junit4 - 下一步；
* 创建Java类，进行单元测试；

此时的Java类要求：①此类是public的 ②此类提供公共的无参的构造器

* 此类中声明单元测试方法。

此时的单元测试方法∶ 方法的权限是public，没有返回值，没有形参

* 此单元测试方法上需要声明注解∶@Test，并在单元测试类中导入∶import org.junit.Test;
* 声明好单元测试方法以后，就可以在方法体内测试相关的代码；
* 写完代码以后，左键双击单元测试方法名，右键∶run as - JUnit Test。

说明∶

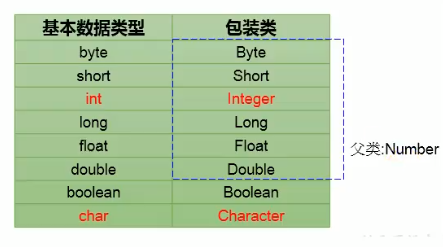
1. 如果执行结果没有任何异常：绿条
2. 如果执行结果出现异常：红条

# 包装类（或封装类）

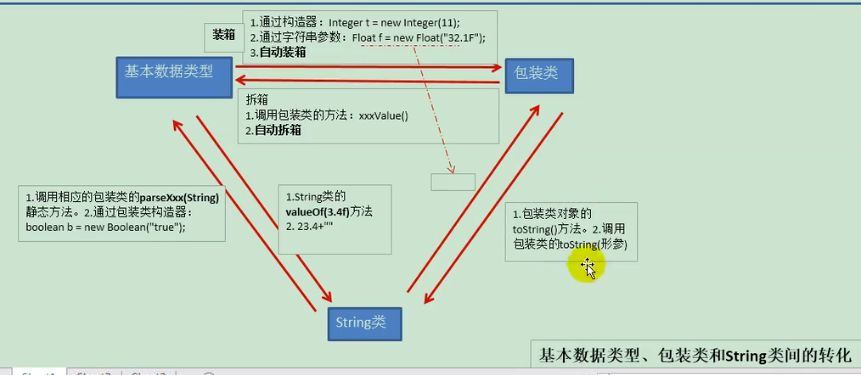
（1）为什么要有包装类（封装类）

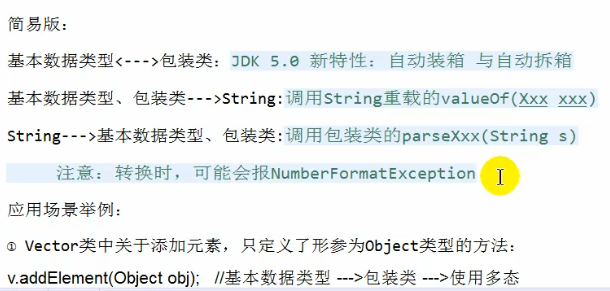
为了使基本数据类型的变量具有类的特征，引入包装类

（2）基本数据类型与对应的包装类



（3）需要掌握的类型间的转换





**具体实现：**

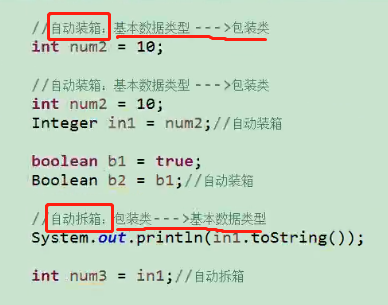
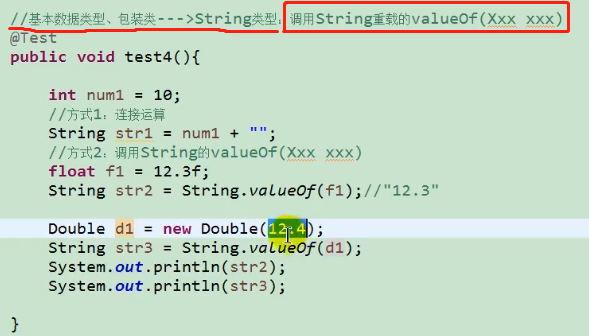
 

图 3 基本数据类型<--->包装类 图 4 基本数据类型、包装类<--->String类型

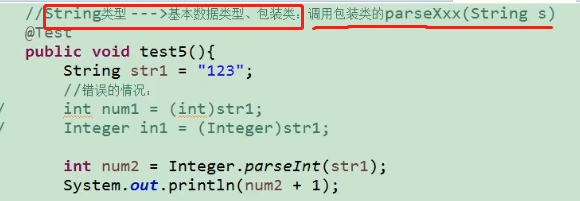
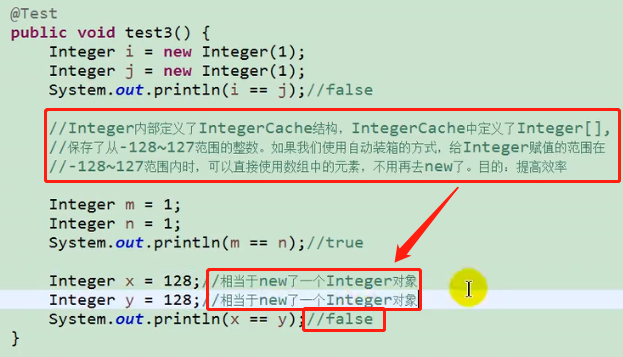


图 4 String类型<--->基本数据类型、包装类

**例题：**



# static关键字的使用（part4—demo1）

static：静态的

1. 可以用来修饰的结构：

主要用来修饰类的内部结构：属性、方法、代码块、内部类

1. static修饰属性：静态变量（或类变量）

① 属性：按是否使用static修饰分为：静态属性 vs 非静态属性（实例变量）

实例变量：我们创建了类的多个对象，每个对象都独立的拥有一套类中的非静态属性；当修改其中一个对象中的非静态属性时，不会导致其他对象中同样的属性值的修改。

静态变量：创建了类的多个对象，每个对象共享同一个静态变量。当通过某一个对象修改静态变量时，会导致其他对象调用此静态变量时，是修改过了的。

② static修饰属性的其他说明：

a、静态变量随着类的加载而加载，可以通过“类.静态变量”的方式进行调用

b、静态变量的加载要早于对象的创建

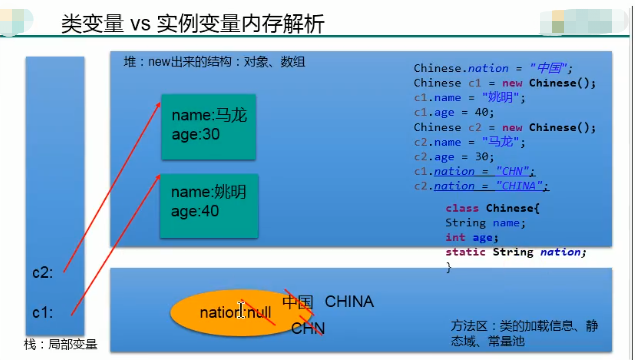
c、由于类只会加载一次，则静态变量在内存中也只会存在一份：存在方法区的静态域中。

d、 类变量 实例变量

类 yes no

对象 yes yes

1. 静态变量内存解析



1. static修饰方法：静态方法（类方法）

①随着类的加载而加载，可以通过“类.静态方法”的方式进行调用

② 静态方法 非静态方法

类 yes no

对象 yes yes

③静态方法中，只能调用静态的方法或属性。

非静态方法中，既可调用非静态的方法或属性，也可调用静态的方法或属性。

1. static的注意点：

* 在静态的方法内，不能使用this、super关键字
* 关于静态属性和静态方法使用，需从生命周期（类或对象的）的角度去理解

1. 如何判定属性和方法应该使用static关键字？

开发中，如何确定一个属性是否要声明为static的?

> 属性可以被多个对象所共享的，不会随着对象的不同而不同

> 类中的常量也常常声明位static

开发中，如何确定一个属性是否要声明为stαtic的?

> 操作静态属性的方法，通常设置为static的

> 工具类中的方法，习惯上声明为static的。比如∶ Math、Arrays、CoLlections

# 设计模式的说明（part4—demo3）

1. 理解

设计模式是在大量的实践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。

1. 常用设计模式—23种经典的设计模式 GOF

创建型模式，5种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式；

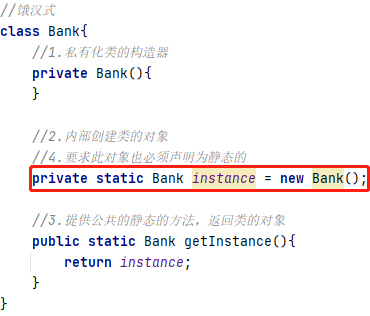
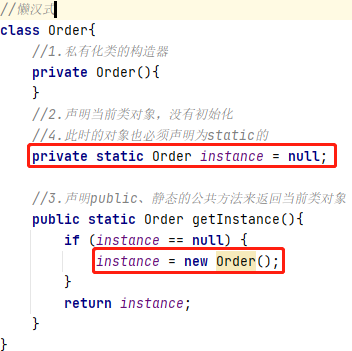
结构型模式，7种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式；

行为型模式，11种：策略模式、模板方法模式、观察者模式、迭代器模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

1. 单例模式

①要解决的问题：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。

②具体代码的实现：

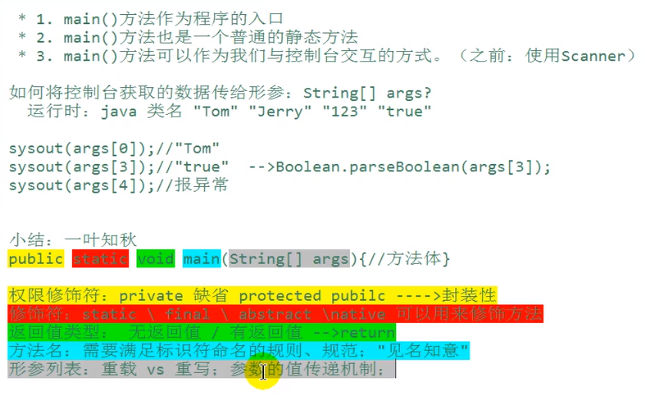
饿汉式 懒汉式

③两种方式的对比

饿汉式：坏处->对象加载时间过长 好处->保证线程安全

懒汉式：好处->延迟对象的创建 目前写法的坏处：线程不安全

# main()方法的使用



# 类的成员之四：代码块（初始代码块）（part4—demo4）

1. 代码块的作用：用来初始化类、对象的信息
2. 分类：代码块若使用修饰符，只能使用static（静态代码块 vs 非静态代码块）
3. 静态代码块

> 内部可以有输出语句

> 随着类的加载而执行，且只执行一次

> 作用：初始化类的信息

> 如果一个类中定义了多个静态代码块，则按照声明的先后顺序执行

> 静态代码块的执行要优先于非静态代码块的执行

> 静态代码块只能调用静态的属性、静态的方法，不能调用非静态结构

1. 非静态代码块

> 内部可以有输出语句

> 随着对象的创建而执行，且每创建一个对象就执行一次非静态代码块

> 作用：可以在创建对象时，对对象的属性等进行初始化

> 如果一个类中定义了多个非静态代码块，则按照声明的先后顺序执行

> 非静态代码块可以调用静态的属性、静态的方法，或非静态的属性、非静态的方法

1. 实例化子类对象时，涉及到父类、子类种静态代码块、非静态代码块、构造器的加载顺序：由父及子，静态先行
2. 对属性可以赋值的位置：

①默认初始化

②显式初始化 / ⑤代码块中赋值 (代码块和构造器时，先执行代码块)

③构造器中初始化

④有了对象后，可通过“对象.属性”或“对象.方法”的方式进行赋值

# final关键字（part4—demo4）

1. final可修饰的结构：类、方法、变量
2. 具体：

* final 用来修饰类 : 此类不可被其它类所继承，
* 比如：String类、System类、StringBuffer类
* final 用来修饰方法：表明此方法不可被重写。比如：Object类中getClass()
* final 用来修饰变量：此时的“变量”称为一个常量
* final 修饰属性：可以考虑赋值的位置有：显式初始化、代码块中初始化、构造器中初始化
* final修饰局部变量：尤其是使用final修饰形参时，表明此形参是一个常量。当调用此方法时，给常量形参赋一个参数。一旦赋值，就只能在方法体内使用此形参，但不可重新赋值
* static final 用来修饰属性：全局常量；

# abstract关键字的使用（part4—demo5）

1. abstract可以用来修饰的结构：类、方法
2. abstract可以用来修饰类：抽象类

> 此类不可实例化

> 抽象类中一定有构造器，便于子类对象实例化调用

（涉及：子类对象实例化的全过程）

> 开发中会提供抽象类的子类，让子类对象实例化，完成相关操作

1. abstract修饰方法：抽象方法

> 抽象方法：只有方法的声明，没有方法体

> 包含抽象方法的类，一定是个抽象类。反之，抽象类中可以没有抽象方法

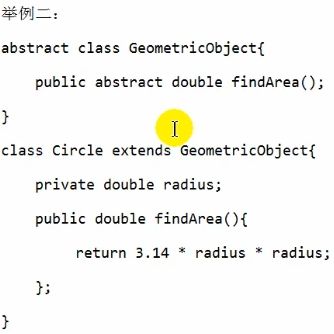
> 若子类重写了父类所有的抽象方法后，此类可实例化

若子类没有重写父类所有的抽象方法后，则此子类也是一个抽象类，需要使用abstract修饰

1. abstract使用上的注意点：

* abstract不能用来修饰：属性、构造器等结构
* abstract不能用来修饰私有方法、静态方法、final的方法、final的类

1. abstract的应用举例

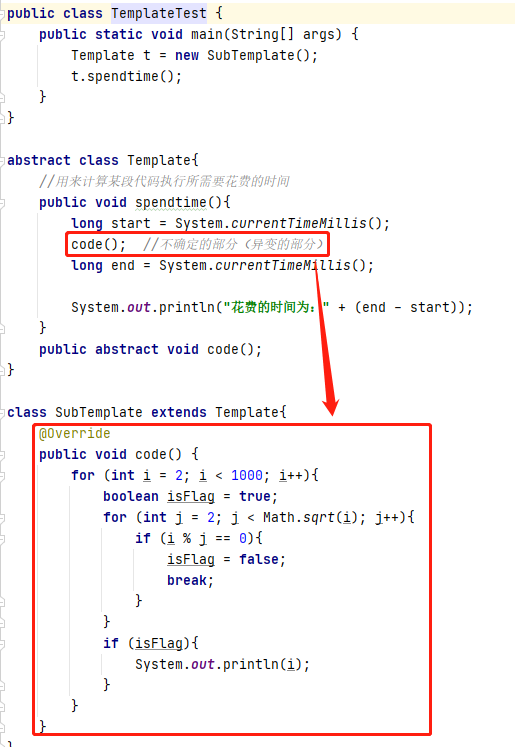


# 抽象类的匿名子类及抽象类的应用：模板方法的设计模式（part4—demo5）

1. 解决的问题

在软件开发中实现一个算法时，整体步骤很固定、通用，这些步骤已经在父类中写好了。但是某些部分易变，易变部分可以抽象出来，供不同子类实现。

1. 举例：



1. 应用场景

模板方法设计模式是编程中经常用得到的模式。各个框架、类库中都有他的影子，比如常见的有∶

● 数据库访问的封装

● Junit单元测试

● JavaWeb的Servlet中关于doGet/doPost方法调用

● Hibernate中模板程序

● Spring中JDBCTemlate、HibernateTemplate等

# interface关键字（part5—demo1）

1. 使用说明：
2. 接口使用interface关键字来定义
3. Java中，接口和类是并列的两个结构
4. 如何定义接口：定义接口中的成员

3.1 jdk7以前：只能定义全局常量和抽象方法

> 全局常量：public static final的，但书写时可省略

> 抽象方法：public abstract的

(略) 3.2 jdk8：除定义全局常量和抽象方法外，还可定义静态方法、默认方法

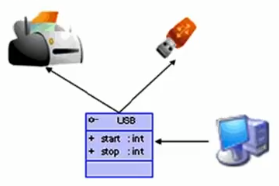
1. 接口中不能定义构造器，意味着接口不可以实例化
2. Java开发中，接口通过让类去实现（implements）的方式来使用

如果实现类覆盖了接口中所有抽象方法，则此实现类就可实例化

如果实现类没有覆盖接口中所有抽象方法，则此实现类仍未一个抽象类

1. Java类可以实现多个接口--->弥补了Java单继承性的局限性

格式：class AA extend BB implements CC,DD,EE{}

1. 接口与接口之间可以继承，且可以多继承
2. 接口的具体使用：体现了多态性
3. 接口，实际上可以看作是一种规范
4. 举例： （part5—demo1—USBTest）

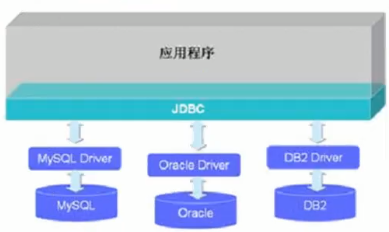
体会：

·接口使用上也满足多态性

·接口，实际上就是定义了一种规范

·开发中，体会面向接口编程

1. 体会面向接口编程的思想

面向接口编程：

我们在应用程序中，调用的接口都是JDBC中定义的接口，不会出现具体某一个数据库厂商的API。

1. Java8中关于接口的新规范（part5—demo3）

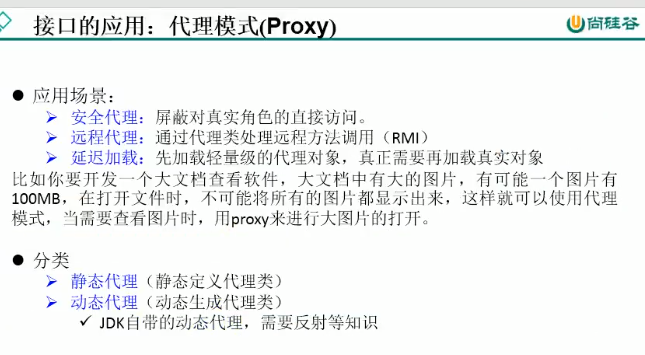
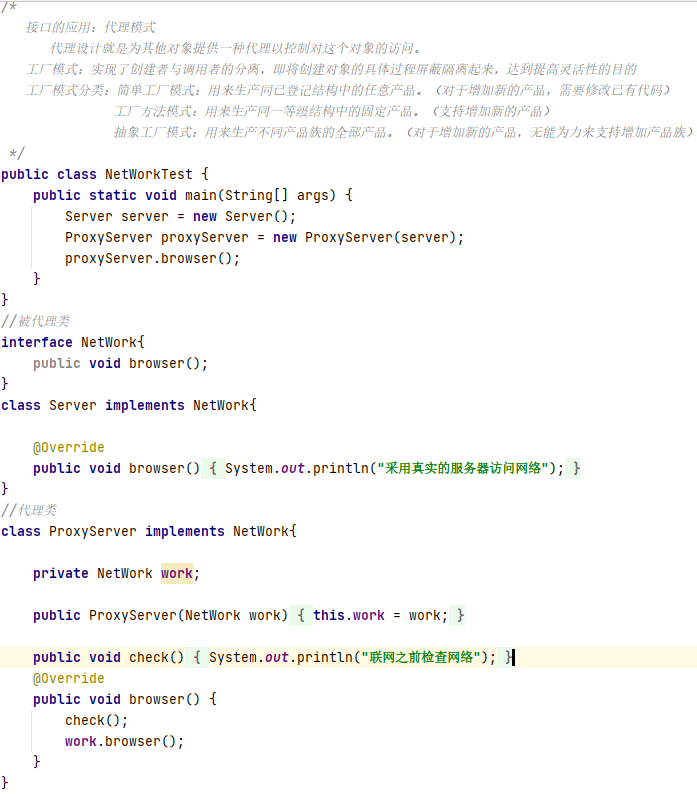


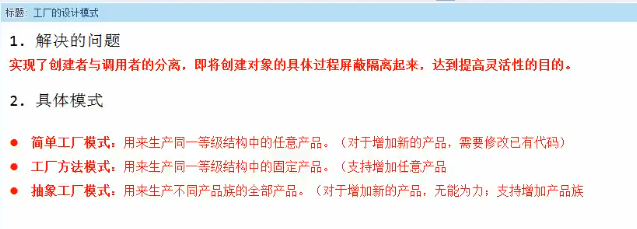
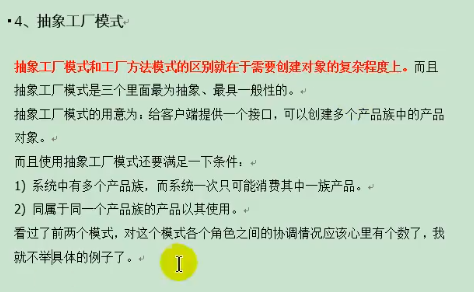
1. 面试题：抽象类和接口的异同

相同点：不能实例化；都可包含抽象方法

不同点：前者有构造器，后者没有；具体从抽象类和接口的定义、内部结构解释说明；前者单继承性，后者多继承；类和接口，多实现。

# 代理模式和工厂模式（part5—demo1）





# 类的成员之五：内部类（part5—demo4）

1. 定义：

Java中允许将一个类A声明在另一个类B中，则类A就是内部类，类B就是外部类。

1. 内部类的分类：

成员内部类(静态/非静态)

局部内部类（方法内、代码块内、构造器内）。

1. 成员内部类的理解：

一方面，作为外部类的成员

> 调用外部类的结构

> 可被static修饰（此时外部类作为内部类出现，可被static修饰）

> 可被4中不同的权限修饰

另一方面，作为一个类

> 类内可以定义属性、方法、构造器等

> 可被final修饰，表示此类不可被继承。即不使用final修饰时可被继承

> 可被abstract修饰，表明不可实例化

1. 成员内部类：

4.1 如何创建成员内部类的对象？（静态，非静态）

//创建静态的Dog内部类的实例（静态的成员内部类）

Person.Dog dog = new Person.Dog();

//创建非静态的Dog内部类的实例（非静态的成员内部类）

//此时不可同上述实例化过程

Person p = new Person();

Person.Bird bird = p.new Bird();

4.2 如何在成员内部类中调用外部类的结构？

class Person{

String name = “小明”；

//非静态成员内部类

class Bird{

String name = "黄鹂";

public void display(String name){

System.out.println(name); //此时调用的是方法的形参

//此时调用的时内部类Bird中的属性name

System.out.println(this.name);

//此时调:外部类Person的成员变量name

System.out.println(Person.this.name);

}

}

}

1. 局部内部类的使用

