1. **面向对象**

面向对象：万事万物皆对象，将具体的属性和方法封装为一个对象，程序员关注设计对象以及处理对象关系，特点是封装、继承和多态。

类：是一个模板，是对象的类型，确定对象拥有的属性和方法，以Class为单元。

属性：对象具有的各种特征

方法：对象执行的操作

对象：类的具体实例即为对象

**成员变量和局部变量**

成员变量 也就是属性，不同方法内相互独立。

局部变量 仅出现和作用于方法内部，外部无法调用

初始值不同：java会给成员变量一个初始值，不会给局部变量赋初始值

变量同名时，局部变量具有更高的优先级

局部变量前不能放置访问修饰符 (private，public，和protected)

**构造方法**

构造方法用来创建对象，并且可以初始化对象的值

构造方法与类同名且没有返回值

构造方法的重载：方法名相同，参数不同，创建对象时需要传参来调用不同的构造方法

构造方法无法被继承

1.一个类没有添加构造方法时，系统会自动添加无参的构造方法

若写了有参构造，系统就不会自动添加无参构造

2.子类实例化会自动调用父类的构造方法

**This**

this.属性：调用当前对象的属性，用来区分成员变量和局部变量

this.方法：调用当前对象的方法

this（）：构造方法内调用其他构造方法，只能在第一行，只能调用一次；

**static静态变量/方法**

类名直接调用

静态变量用一次变一次，仅有一个固定值；

静态方法中不能直接调用非静态成员和方法，需要创建对象来调用；

static初始化块只在类加载时执行一次，只能给静态变量赋值

程序初始化执行顺序：static 静态初始化块——普通初始化块——构造方法

static常驻内存，程序加载时就会被分配空间，执行效率高，但释放内存时间长

**static final必须声明的同时初始化，通过类名访问**

1. **封装**

概念：将类的信息隐藏在类的内部，不允许外部程序直接访问，而是通过该类提供的特定的方法（setter.getter）来实现对隐藏信息的操作和访问

类：封装的是对象属性和行为

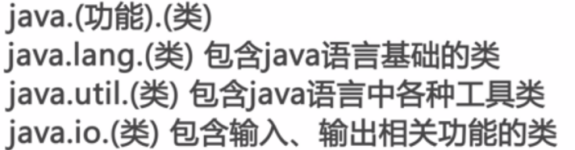
方法：封装的是功能具体实现

访问修饰符：封装的访问的权限

**使用包管理java中的类**

全小写，作用：管理java文件；解决同名文件冲突

必须放在java源程序第一行

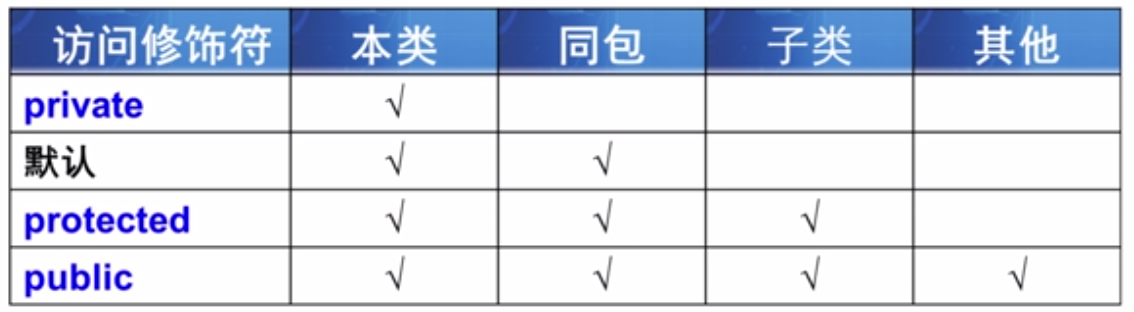


包的使用：如何在其他包中使用我们自定义的类

Import com.imooc.\*; 导入包下所有文件

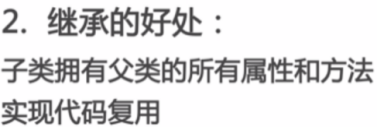
Import com.imooc.hello; 导入hello

**访问修饰符—修饰属性和方法的访问范围**

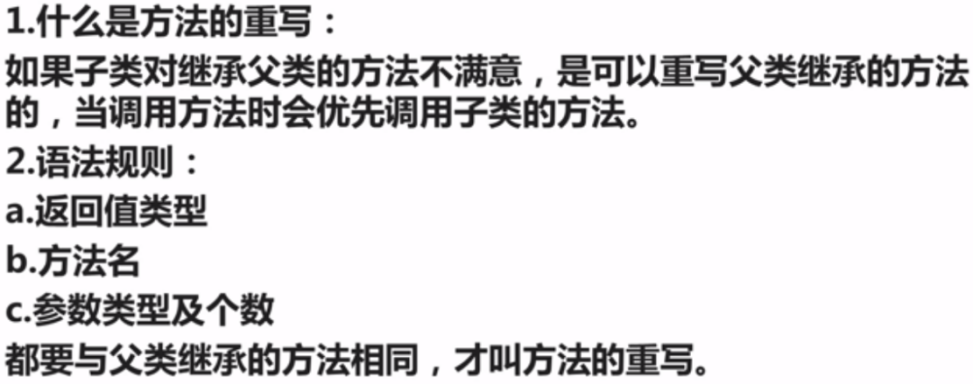


1. **继承 extends**

\*属性和方法修饰符不能是private，Java是单继承



**方法的重写**



方法重载：只看参数列表，遵循编译期绑定，根据参数的类型调用不同的方法

方法重写：遵循运行期绑定，根据对象的类型调用不同的方法

方法重写规则：

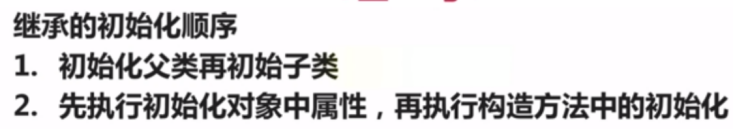
两同：a.方法名相同 b.参数列表相同

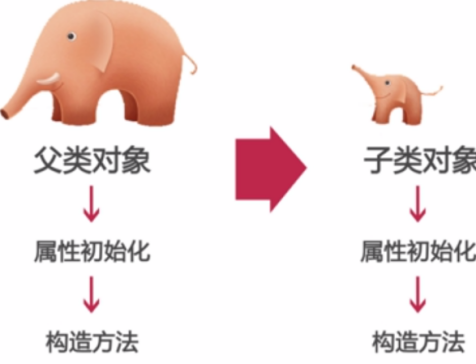
两小：a.子类的返回值类型小于等于父类【基本类型和void必须相同】

b.子类抛出的异常小于等于父类

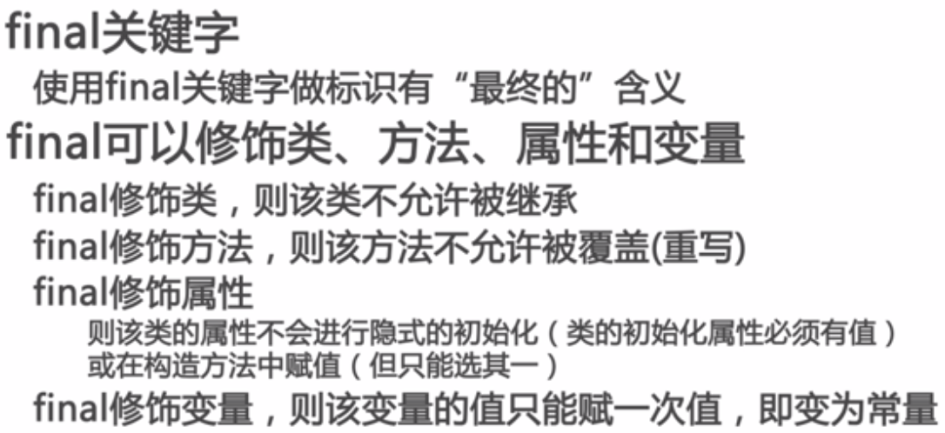
一大：子类的访问权限大于等于父类

**继承的初始化顺序**





**final的使用**



定义的常量名用大写字母

\*如果final声明属性的时候没有初始化，那么在构造方法里一定要进行初始化

\*如果final修饰方法，则子类不允许覆盖（修改）父类的方法，但允许继承调用

Public static final int NUM = 2;

该句在编译时，就直接将NUM替换为数值2 ，在类.class加载时存放进方法区，

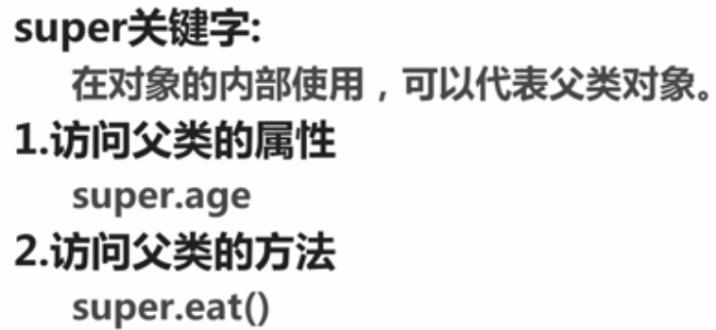
Public static int NUM = 2;

该句与上句相比，在调用时多了一个赋值的过程，将2赋值给num存放进方法区

Public final int NUM =2;

成员变量，存放在堆中

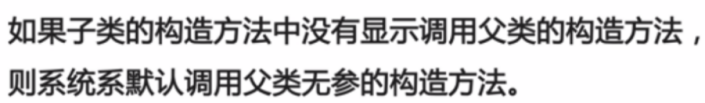
**Java中super的使用**

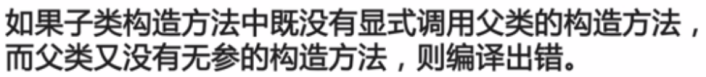


**Super与构造方法的关系：**

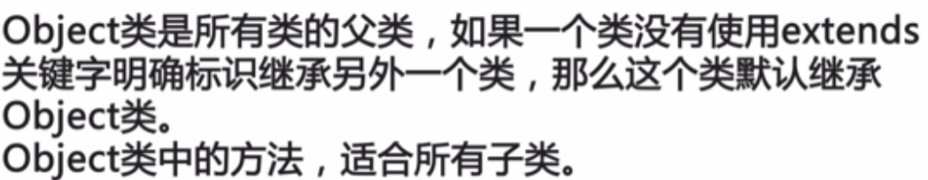
子类的构造过程必须调用父类的构造方法，且必须在第一行







**Java中的object类**



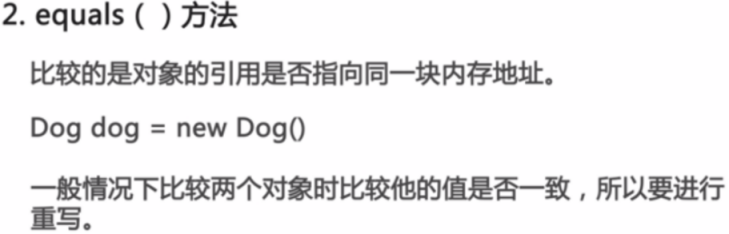


**Java很多地方会默认调用对象的toString()方法。**

比如对象是String类型，String类内部已经封装了自己重写了的toString()方法，所以输出是字符串而不是哈希码

如果对象是自定义的类的类型，就需要重写toString（）方法

哈希码用于区分不同的对象



equals方法默认情况下比较的是对象的引用地址是否相同，所以要比较两个对象的值是否相同，需要重写equals方法，在source-general equals中添加后修改。

\*\*Equals方法的返回值是一个布尔类型，所以适合使用if else

\*\*equals原本和==运算符是相同的，比较的都是两个引用的内存地址。

\*\*String类的内部已经重写了equals方法，所以直接比较字符串内容

**内部类**

内部类总结：

**内部类不允许其他类访问**

内部类可访问外部类的所有数据，包括private

创建内部类对象：

Outer o = new outer();

Inner i = o. New Inner();

成员内部类：

**inner 类可以使用任意访问控制符，**如 public 、 protected 、 private 等

编译后，会发现产生了两个 .class 文件

**外部类是不能直接使用内部类的成员和方法**

静态内部类

创建静态内部类的对象时，不需要外部类的对象，可以直接创建

内部类 对象名= new 内部类();

方法内部类

中的方法只在方法内部可见和使用，不能使用访问控制符和static修饰

public class OuterInner {

public void show(){

class InnerT{

}

InnerT t = new InnerT();

t.print();

}

}

匿名内部类

场景：想创建一个类的对象，并且该类只被创建一次对象，该类不必命名

条件：该类必须是子类 （继承父类，实现接口）

**匿名内部类访问外部类的数据，该数据必须是final修饰的**

创建：

Thread t1 = **new** Thread(){

**public** **void** run(){

**for**(**int** i=0; i<10000;i++){

System.***out***.println("");

}

}

};

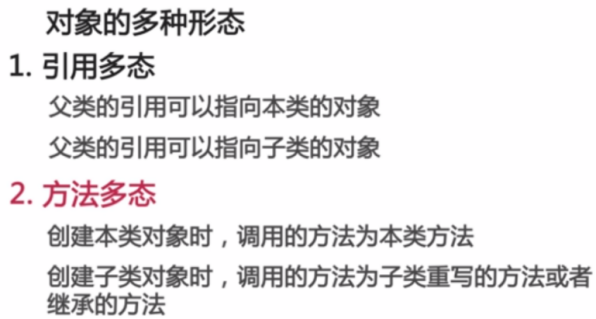
1. **多态**

重载：同一个类中，方法名相同，参数列表不同，遵循编译器绑定，根据参数类型绑定方法

重写：父子类中，方法名相同，参数列表相同，遵循运行期绑定，根据对象的类型调用

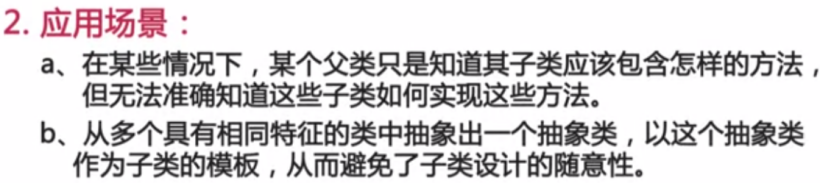
继承是多态的实现基础

**多态的定义：**



**抽象类**

**abstract**修饰的类，可不包含抽象方法；





抽象方法只有方法的定义，无方法体｛｝，抽象类不能创建对象

**接口：interface**

接口：只能有全局常量（**static final**修饰的）和抽象方法

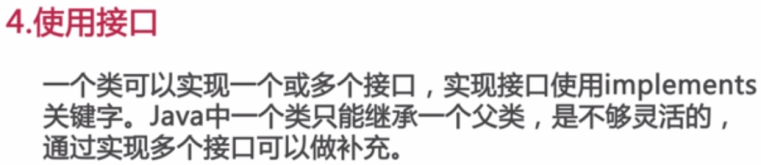
不能被实例化，只能有抽象方法public abstract和全局常量public staitc final，属于特殊的类

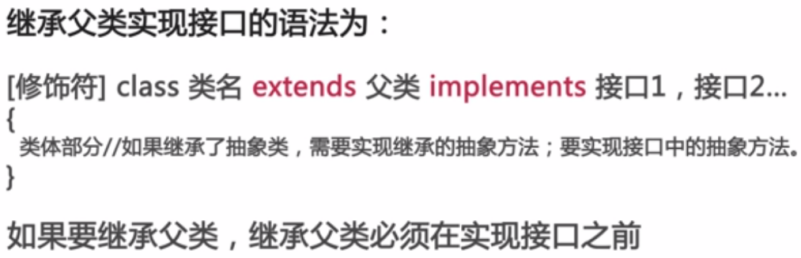
接口可以继承多个接口，不能继承类或抽象类。

类实现接口，必须进行方法的重写

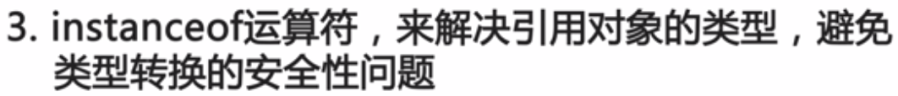
类是单继承的，接口是多继承的







**引用类型转换 instanceof，返回值boolean**



1. **堆、栈、方法区**

内存泄漏：不再使用的内存没有被及时释放，不再使用的对象引用及时设置为null

栈内存：存放方法的局部变量和引用（物理地址），使用栈中的引用访问堆内存中的对象

每一个线程都包含一个栈区，一个栈区分为三部分：基本类型变量区，执行环境上下文，操作指令区（存放操作指令）。 速度仅次于寄存器register，快于堆

调用方法时为方法分配对应的栈帧，方法执行结束是栈帧清楚，局部变量消失

堆内存heap：存放对象【成员变量 数组 final GC】

成员变量的生命周期：对象被回收时一并消失

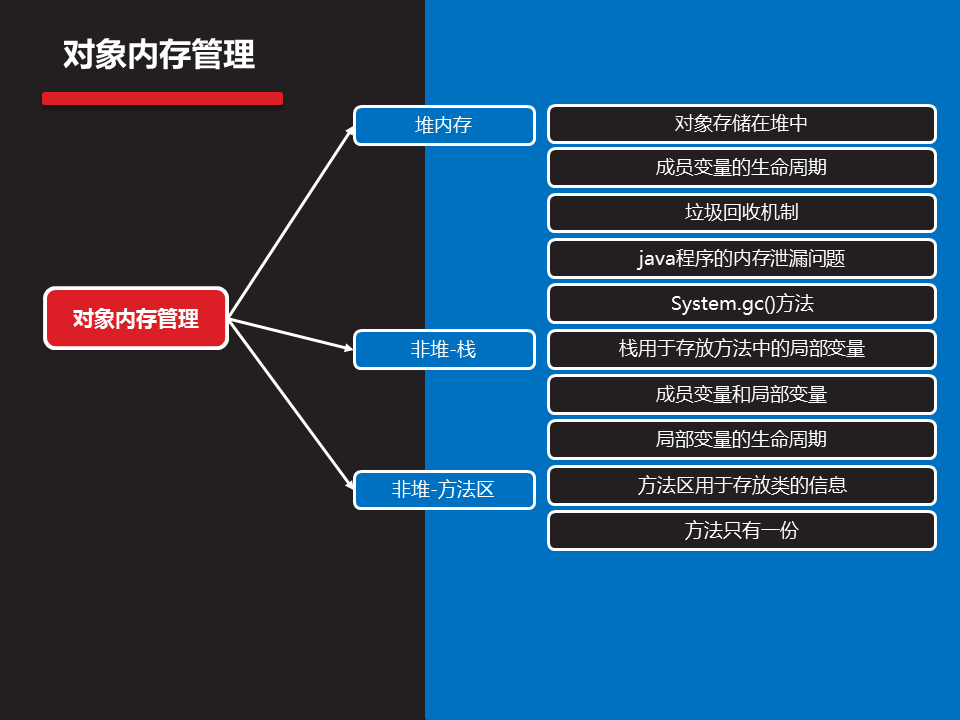
Jvm只有一个堆区被所有线程共享

没有任何引用指向的对象称之为垃圾，GC不定时回收，调用system.gc()建议虚拟机尽快回收

方法区：存放.class文件以及方法，是类信息，方法只有一份。方法区被所有线程共享。包含所有的class和static变量，是整个程序中永远唯一的元素

引用之间划等号指向同一个对象

引用类型变量，可以赋值为null，表示还没有指向任何对象，当引用为null时，调用对象的成员变量和方法会返回空指针异常



垃圾回收器是GC 是jvm自带的一个线程（自动运行着的程序），用于回收没有任何引用指向的对象 。调用system.gc()方法

内存泄漏是指，不再使用的内存没有及时的回收，严重的内存泄漏会因过多的内存占用而导致程序的崩溃

GC线程判断对象是否可以回收的依据是该对象是否有引用指向，因此当确定该对象不再使用时，应该及时将其引用设置为null.

所有实例化对象是在公用方法区中的class定义的

finalize方法名

Java 技术允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作。

这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的。

它是在 Object 类中定义的，因此所有的类都继承了它。

子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作。

finalize() 方法是在垃圾收集器删除对象之前对这个对象调用的。

**public** **class** F {

**public** **static** **void** main(String[] args) **throws** InterruptedException {

F f = **new** F();

f = **null**;

System.*gc*(); //gc在回收的时候调用finalize()方法

Thread.*sleep*(10);//休息10秒

System.***out***.println("main方法调用完毕");

}

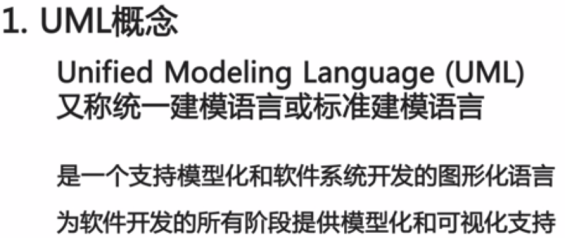
**protected** **void** finalize()**throws** Throwable{

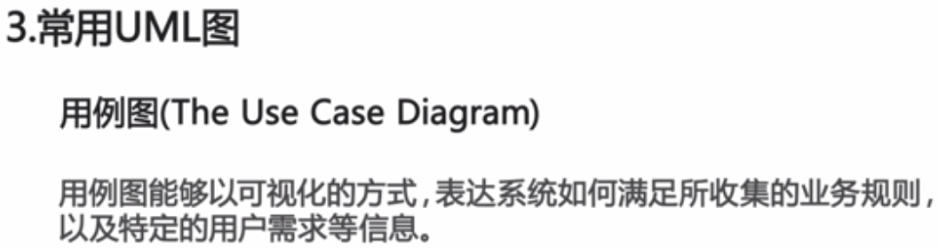
System.***out***.println("当前对象即将被释放");

}

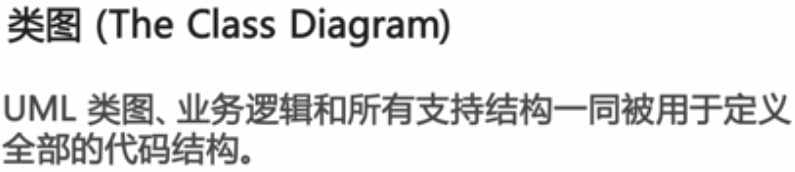
}

**UML简介：**











**开发过程：**

1. **需求分析**
2. **概要设计 ：总体结构设计；概要设计说明书；数据库设计说明书**
3. **详细设计说明书**
4. **代码实现**
5. **测试**
6. **打包部署上线**