[Java面向对象引入 3](#_Toc70606768)

[1.1面向对象的思想 3](#_Toc70606769)

[1.2 面向对象的特点 3](#_Toc70606770)

[1.3 类与对象 4](#_Toc70606771)

[1.3.1 匿名对象 6](#_Toc70606772)

[1.3.2 成员变量和局部变量的区别 6](#_Toc70606773)

[面向对象的三个重要特征：封装、继承、多态 7](#_Toc70606774)

[2.1 封装（Encapsulation） 7](#_Toc70606775)

[2.1.1 封装的思想和特点 7](#_Toc70606776)

[2.1.2 基本数据类型和引用数据类型的参数传递 8](#_Toc70606777)

[2.1.3 构造函数 9](#_Toc70606778)

[2.1.4 静态——static 12](#_Toc70606779)

[2.1.5 设计模式——单例设计模式 16](#_Toc70606780)

[2.2 继承（Extends） 18](#_Toc70606781)

[2.2.1 继承的思想和特点 18](#_Toc70606782)

[2.2.2 单继承和多继承 19](#_Toc70606783)

[2.2.3 继承关系中父类和子类的成员 20](#_Toc70606784)

[2.2.4 final关键字 23](#_Toc70606785)

[2.2.5 抽象类 24](#_Toc70606786)

[2.2.6 接口——interface 26](#_Toc70606787)

[2.3 多态——（Polymorphism） 29](#_Toc70606788)

[2.3.1 多态的思想和特点 29](#_Toc70606789)

[2.3.2 向上向下转型 30](#_Toc70606790)

[2.3.3 多态中对成员（变量和函数）的调用 32](#_Toc70606791)

[Object 对象 34](#_Toc70606792)

[3.1 boolean equals(Object obj) 35](#_Toc70606793)

[3.2 String toString() 37](#_Toc70606794)

[内部类 37](#_Toc70606795)

[4.1 内部类的思想和特点 37](#_Toc70606796)

[4.2 非静态非私有内部类 38](#_Toc70606797)

[4.3 静态非私有内部类 38](#_Toc70606798)

[4.4 内部类细节 39](#_Toc70606799)

[4.5 匿名内部类 40](#_Toc70606800)

# Java面向对象引入

## 1.1面向对象的思想

过去我们解决问题通常是用面向过程的思想，那么什么是面向过程，什么是面向对象呢？举个例子，把大象装进冰箱里，分别用面向过程和面向对象的方式来解决。

面向过程：

1. 打开冰箱。
2. 存储大象。
3. 关闭冰箱。

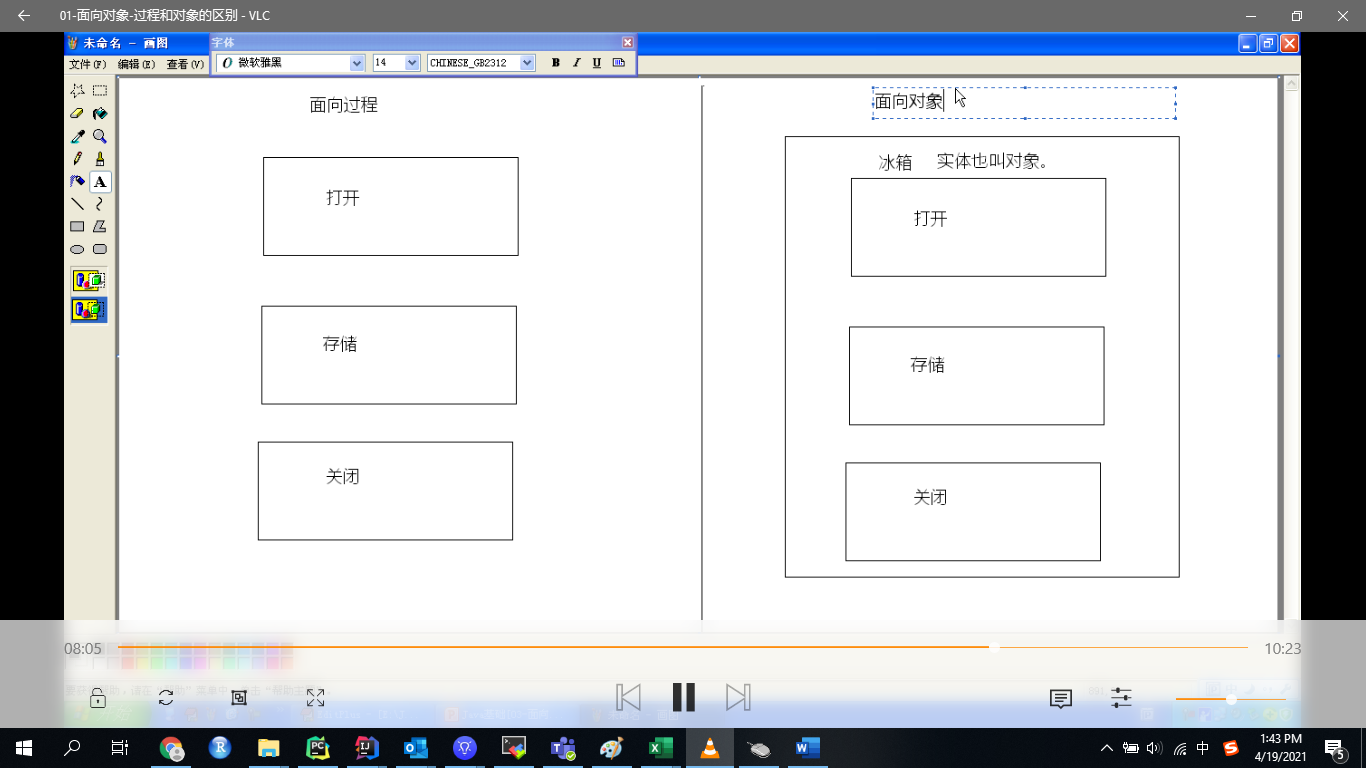
这种解决方案，它关注的是过程，或者说是关注一些行为，如打开，存储，关闭。那么对应在程序开发里就是，我们主要定义解决问题要做的行为：

打开(); 存储(); 关闭();

面向对象：

1. 冰箱打开。
2. 冰箱存储。
3. 冰箱关闭。

这个看起来虽然和面向过程很像，但是有一个非常大的区别，我们现在是基于冰箱这个对象开始思考的，这所有的行为都是冰箱的，而不是我们的。我们一样还是有打开，存储，关闭这些行为，但是它们现在被冰箱封装起来了，我们知道冰箱拥有这些行为，那么我们就不必再关注行为本身，而是关注冰箱这个对象。



## 1.2 面向对象的特点

面向对象有以下特点：

1. 面向对象是一种思想，它更符合生活中的思考习惯。
2. 从过程的执行者，转变成对象的指挥者。
3. 将复杂的问题简单化。

来举个例子加深一下理解，如果你要去电脑城买组装机，你会怎么做？

1. 自己先在网上查一下价格
2. 自己去电脑城看各个部件，货比三家。
3. 挑中要买的那一家，询价。
4. 讨价还价
5. 付款，拿到电脑。

上面的步骤通常就是你自己打算去买电脑时会做的，整个过程当中，你都是执行者，自己去做很多步骤，这就是面向过程的解决方案。

那么如果你身边有一个很懂电脑的朋友呢？他对电脑部件和价格都非常熟悉，你会怎么做？当然是请他跟你一起喽：

1. 他本人很了解电脑和电脑价格。
2. 请他带你去划算的电脑城。
3. 请他帮你询价。
4. 请他帮你讨价还价。
5. 付款，拿到电脑。

现在有了专家的帮助，你几乎都不用再自己做什么事情了，你就从一系列步骤的执行者变成了这个专家的指挥者，而且整个买电脑的过程会非常的简单和快速。这就是面向对象的解决方案。

## 1.3 类与对象

先不考虑java代码，从生活出发思考一个问题，生活中我们常有一些事物，比如汽车。我们如何描述一台汽车呢？一般而言，汽车会有一些属性，比如颜色，轮胎个数，车灯数等等，除了属性，汽车还有一些行为，比如启动，运行，停止等等。那么描述汽车就是：

汽车：

属性：颜色，轮胎个数。

行为： 启动，运行，停止。

这种描述转换为java语言中的描述，也是很类似的，首先说明这是描述汽车，然后列出汽车的属性和行为即可。属性是不确定的，所以定义成变量，行为就用函数来表示。

class Car{

String color;

int num;

void run(){}

}

生活中我们说到汽车，其实是指一个概念，它只是一种描述，比如你去汽车店买车时，你说买汽车其实是指要买一个汽车对象，像是买辆奔驰车，或者宝马车。

Java语言定义的class Car 也是一样的，它只是一种描述，如果你想创建一个真正的实体，必须要通过 new Car() 来实现，这就创建了一个对象。

那么就总结出了类与对象的定义：

类：描述某一类事务。

对象：该类实物具体的个体。

创建类实际上就是描述事物，描述事物其实就是描述事物的属性和行为：

属性对应类中的成员变量。

行为对应类中的成员方法。

到这里我们就理解了类和对象的概念和关系，那么观察这个汽车类，发现它和我们之前学习的东西好像不太一样，我们之前总是用到main方法啊，那么为什么这个汽车类没有main方法呢？

这里需要解释一下main方法，main方法的目的是为了运行，如果想让汽车类也可以运行，那么可以在汽车类里添加main方法，但是实际上汽车不会自己运行的，它一般是被人调用才能运行。main方法所在的类就是可以运行的类，那么就要用另一个包含main方法的类来调用汽车类，达到运行的目的。

class CarDemo {  
  
 // 定义这个类的目的是为了定义主函数，让该类运行，在该类中调用Car类中的内容。

// Car 类只是一种虚拟的描述，要想使用里面的内容，必须先创建一个Car类型的对象.

// 如何创建对象呢？通过关键字new来创建。  
 public static void main(String[] args) {  
 Car c = new Car();

// Car c 意思是创建了一个Car 类型的变量c. new Car() 意思是创建一个Car类型的对象。

// 上面代码的用意是，创建一个Car类型的变量c，指向了一个Car类型的对象。

// 其实它和普通的变量定义很像，int a = 4; 都是：数据类型 变量名 = 初始化值  
 }  
  
}

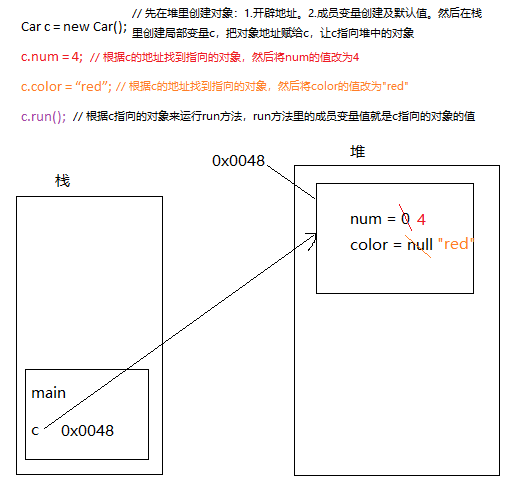
为了加深理解，试着画一下下面代码的内存图解：

Car c = new Car();

c.num = 4;

c.color = “red”;

c.run();



### 1.3.1 匿名对象

匿名对象是指没有名字的对象，如 new Car(); 就是一个匿名对象，它是对象的简化定义方式。

用处：1. 匿名对象可以作为实际参数进行传递。

2. 如果对象方法只进行一次调用，可以简化为匿名对象的形式。

如：Car c = new Car(); c.run();

要是上例中c.run()仅执行一次的话，就可以简化为 new Car().run();

### 1.3.2 成员变量和局部变量的区别

1. 源代码中定义的位置不同：

成员变量定义在类中，局部变量定义在方法中或者语句（如for语句）中（只要是类的下一级大括号都是局部的）

2. 内存中的存储位置不同：

成员变量存储在堆内存的对象中，局部变量存储在栈内存的方法中。

3. 生命周期不同：

成员变量随着对象的出现而出现，局部变量随着所属区间的运行而出现，随着所属区间的结束而释放。

# 面向对象的三个重要特征：封装、继承、多态

## 2.1 封装（Encapsulation）

封装就是隐藏实现细节，并对外提供公共的访问方式（接口）。

### 2.1.1 封装的思想和特点

生活中也有很多封装的例子，如电脑的主机，其实主机包括很多零部件，但是都用机箱包起来了，你看不到机箱里面的东西，但是你知道按了机箱上的开机电源，电脑就可以开机，而且通过插入USB接口，你就可以使用鼠标和键盘。

那么如何体现代码中的封装呢？有好几种方式。

1. 通过权限修饰符来规定成员变量和成员方法可以被访问的范围。

如：

class Person{

private int age; // 设置权限修饰符为private，意味着只能在当前类内部访问age

public int getAge(){ return age; } // 提供公共的接口来访问age的值

public void setAge(int a){ // 提供公共的接口来设置age的值并对传入的值进行判断

if(a < 0 || a >130){

throw new RunTimeException(“传入的值错误！”);

}

age = a;

}

}

这样做的好处是什么？

1. 提高安全性：因为不允许直接访问细节，而是通过公共的接口访问，可以实现可控。
2. 提高易用性：不需要知道细节，使用简单。比如机箱里面是什么我都不知道，但是按键就可以开机。
3. 提高复用性：方法本身就是封装了固定功能的代码，通过重复调用方法就可以复用这个功能。
4. 隔离了变化：比如一个方法，无论它内部的功能怎么变化，我都不关注，我只关注怎么使用这个方法。

### 2.1.2 基本数据类型和引用数据类型的参数传递

class Demo{

public static void main(String[] args){

int x = 4;

show(x);

System.out.println(“x=” + x); // 这里打印的结果是什么？

}

Public static void show(int x){

X = 5;

}

}

class Demo{

int x = 6;

public static void main(String[] args){

Demo d = new Demo();

d.x = 8;

show(d); // 如果这里是show(new Demo())，打印结果是什么？

System.out.println(“x=” + d.x); // 这里打印的结果是什么？

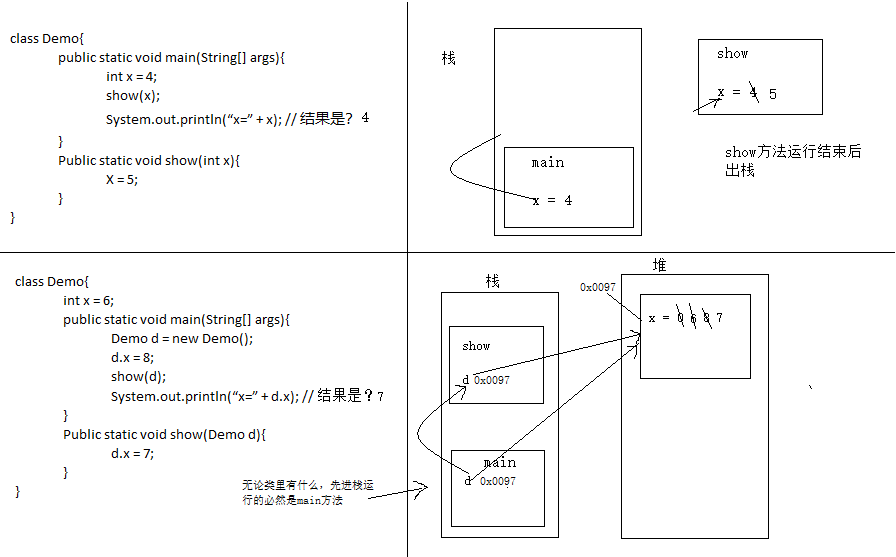
}

Public static void show(Demo d){

d.x = 7;

}

}



### 2.1.3 构造函数

我们前面学过，当创建一个对象时，里面的成员变量会有默认值，我们只能创建对象后再给对象里面的变量赋值，如：

Person p = new Person();

p.age = 12;

p.name = “张三”;

那么，能否在创建对象的同时，就给成员变量赋值呢？当然可以。就相当于要对对象进行初始化，怎么实现对象初始化呢？Java为我们提供了构造函数，就是为了解决这个问题。

我们之前也学过自己定义函数，为了封装一些需要复用的功能，这些都是一般函数。构造函数与一般函数类似，它也是功能，只不过这个功能是用于给对象进行初始化。

#### 1.构造函数定义

构造函数格式是固定的：

1. 函数名和类名相同
2. 没有返回值类型
3. 没有具体的返回值，但是构造函数有return; 这里的return是为了结束构造函数。

那么如果让每一个创建的person都有给定的名字，那么Person类的构造函数就是：

Person(String n){

name = n;

}

现在再创建对象时，就可以给初始化值了：Person p = new Person(“张三”);

#### 2.构造函数和一般函数比较

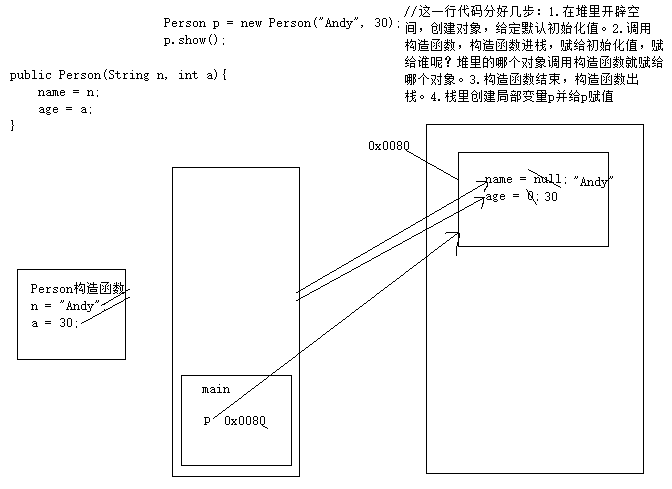
构造函数和一般函数的区别：

构造函数在对象创建时就执行了，且只执行一次。

一般函数是在对象创建后，需要时才被对象调用，可以调用多次。

构造函数和一般函数一样，都可以重载（overload）。

为了加深对构造函数的理解，我们也要画一下构造函数的内存图：



#### 3.默认构造函数

学习了构造函数，我们再回头看之前写的代码就会有疑问，之前我们new对象的时候我们也没定义构造函数啊，那么为什么还可以new对象呢？

这是因为每一个类中，都有一个默认的空参数构造函数。

但是需要注意的是，如果我们自己在类里定义了构造函数，默认的构造函数就没有了。

#### 4.私有构造函数

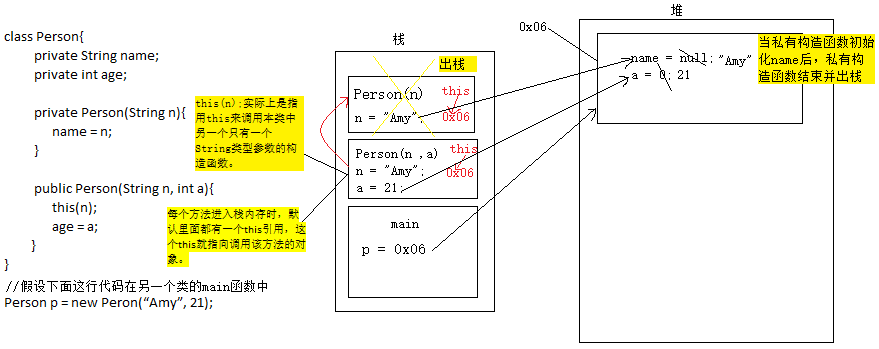
可以有私有化的构造函数，但是一旦私有化，外部就不能再调用这个构造函数创建对象，所以可以用来限制创建对象。

#### 5.构造函数间调用——this关键字

注意1：构造函数只能被构造函数调用，不能被一般方法调用。为什么？一般方法必须在对象被创建后用对象调用，而构造方法则是创建对象的，必须先用构造方法创建对象然后再调用一般方法。

这里有一个的例子，其中Person类有两个构造函数，一个是私有构造函数，另一个是公共构造函数，但是呢，公共构造函数中有一行特殊代码：this(n); 这里需要先解释一下this关键字：

this 是代表一个对象，相当于对象的一个引用，代表哪个对象呢？哪个对象调用this所在的函数，this就代表哪个对象。是不是有点难理解，试着画个图就明白了。



上图里有几个知识点要注意：

重点1：每个方法进入栈内存时，默认里面都有一个this引用，这个this就指向调用该方法的对象。

重点2： this调用构造函数这种用法只能放在构造函数里，且一定要放第一行。

除了this(n); 以外，this还有另一种用法。

我们上面的构造函数中，是这么写的：

public Person(String n, int a){

}

这里参数中的n和a都是局部变量名，正常情况下这个名字是可以任意取的，那么如果我想让这个名字更有意义，一眼就能知道表示什么，我也可以这么写：

public Person(String name, int age){

name = name;

age = age;

}

像这种写法可以吗？当然不可以，我们的目的是想要用局部变量name给成员变量name赋值，但是实际上，在这个构造函数中，两个name都会被认为是局部变量，所以没法儿给成员变量赋值。那么怎么才能给成员变量赋值呢？

我们知道构造函数中默认有一个this引用，指向对象，那么我们就可以用this来访问到成员变量，所以构造函数应该这么写：

public Person(String name, int age){

this.name = name; // 这里右侧的name是局部变量，左侧用this.name 表示成员变量

this.age = age;

}

### 2.1.4 静态——static

先抛开静态这个概念，我们考虑一种实际情况，类中有可能有一些方法，这些方法只是做一些特殊的事情而不用访问类的成员变量，比如，我定义一个sleep() 方法，在方法里打印一行字。这种情况下，还有必要创建对象去调用它吗？答案是没有必要，我又不用访问成员变量，我也不调用别的成员方法，只是做了一个简单的打印，那我干嘛要创建对象？

那么这个时候，我们就希望不用创建对象就可以调用它，在java里能实现吗？能的，只要把它定义成静态方法即可，也就是在前面加上 static 关键字，如： public static void sleep(){}

这时，就可以直接用类名来访问了，如：Person.sleep();

#### 1.静态方法

什么时候需要静态方法呢？

当该方法不需要访问对象的属性时，可以通过添加static把方法设置为静态方法。

#### 2.静态变量

static不仅可以修饰方法，也可以修饰成员变量。

同样的思考一个例子，Person这个类有一个属性country，表示新建对象的国籍，在中国境内，这个country是不是固定的？全部都是”China”，是的。那么当新建对象时，对象的name和age不一定，但是country肯定是一样的，那么就可以把country当成一个通用的属性，每个对象都是一样的，这时，就可以用static修饰它，把它变为静态变量。如：static String country;

所以，静态变量时所有对象共享的变量。那么它就必须在创建对象前就出现了，否则，对象都来了还没有它，那还说啥共享。

#### 3. 静态的特点

我们想要了解静态，还是要从它在内存中的表现形式着手。

之前我们已经知道两种内存区，栈和堆，栈和堆都是运行时创建的。但是静态成员比较特殊，它在对象被创建前就已经存在了，所以就涉及另外一个重要的内存区域：方法区，又叫数据共享区它是存放代码的一块区域。

方法区是什么时候创建的呢？我们从编写一个java文件到运行它有两个过程：

1. 编写HelloWorld.java 文件，包含一个main方法。
2. javac HelloWorld.java 来编译java文件（JDK里的工具来做的），创建一个HelloWorld.class文件。
3. java HelloWorld 来运行 HelloWorld.class文件。

上面的第3个步骤中，运行过程都做了什么呢？我们知道HelloWorld.class是一个编译后的字节码文件，运行这个文件的第一步，就是先把这个文件（或者说这个类）加载到内存中的方法区里面。这就是我们经常听到的类加载过程。

方法区里有两个部分：1. 普通方法区，它以类为单位存放类中的非静态成员。2. 静态方法区，它也以类为单位，存放类中的静态成员。

了解了以上的内容后，我们来试着画一下下面代码运行时的内存分配：

class Person{

private String name;

private int age;

static String country;

Person(String name, int age){

this.name = name;

this.age = age;

}

public void show(){

System.out.println(“name = “ + name + “, age = “ + age);

}

public static void sleep(){

System.out.println(“我正在睡觉：呼呼~呼呼~”);

}

}

public class myDemo{

public static void main(String[] args){

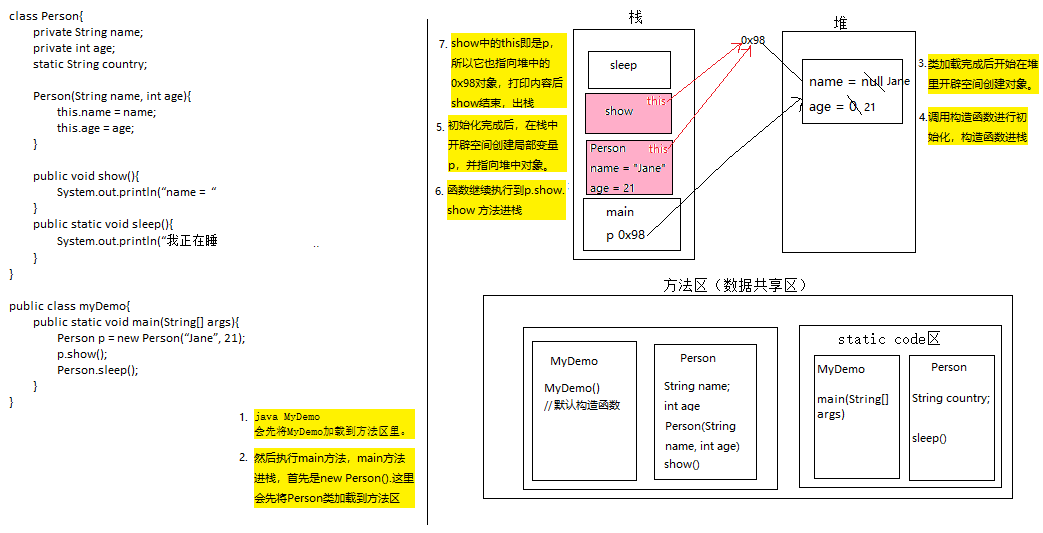
Person p = new Person(“Jane”, 21);

p.show();

Person.sleep();

}

}



至此，总结一下，静态成员的特点：

1. 可以修饰成员变量和成员方法，静态成员可以直接用类名调用。
2. 静态成员优先于对象被创建。
3. 静态成员随着类的加载而加载，随着类的消失而消失。所以静态成员的生命周期很长。

但是使用静态成员时，也有一些要注意的地方：

1. 静态方法只能访问静态成员，不能访问非静态成员。
2. 静态的方法中没有默认的this和super，也不能使用this或者super。
3. 主函数是静态函数。

什么时候用静态？

1. 如果成员变量在所有对象中都一样，直接把它设置为静态变量。
2. 如果成员函数没有访问成员变量或者其他的成员函数，可以把它设置为静态函数。

成员变量和静态变量的区别：

1. 名称不同：成员变量又叫实例变量，静态变量又叫类变量。
2. 内存存储方式不同：成员变量存储在堆内存中的对象中，静态变量存储在方法区的静态区中。
3. 生命周期不同：成员变量随着对象的出现而出现，随着对象的消失而消失。静态变量随着类的出现而出现，随着类的消失而消失。

#### 4. 静态代码块

代码块是指类中独立的一块代码，用{}括起来，静态代码块是指用static修饰的代码块，如：

class Person{

static { // 静态代码块

System.out.println(“A”);

}

staic void show(){

System.out.println(“show”);

}

}

静态代码块的特点：随着类的加载而执行，且只执行一次。是为了给类进行初始化的。

应用场景：类不需要创建对象，但是需要初始化，这时可以将部分代码存储在静态代码块中。

#### 5. 构造代码块

类中的一个代码块，为了给对象进行初始化。每次创建对象都会执行。如：

class Person{

{ // 构造代码块

System.out.println(“A”);

}

Person(){

}

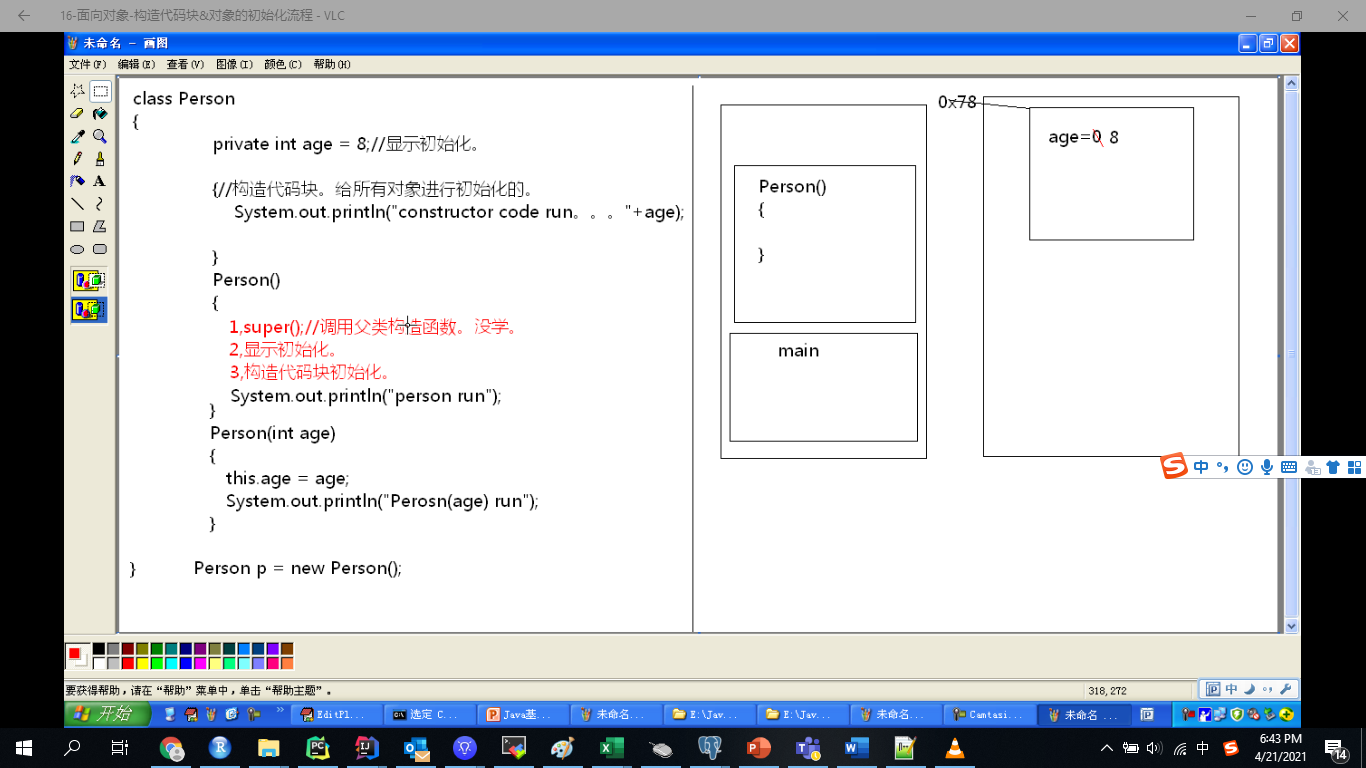
Person(){

……

}

}

但是有个特殊的内容，参考下面的内存解析图：



总结一下创建对象的流程：

1. 加载对应类的字节码文件进入内存。
2. 通过new在堆内存中开辟空间，分配首地址值。
3. 对对象中的属性进行默认初始化。
4. 调用与之对应的构造函数，构造函数压栈。
5. 构造函数中会先调用隐式的super() 来调用父类的构造函数。
6. 对属性进行显示初始化。
7. 调用类中的构造代码块。
8. 执行构造函数中自定义的初始化代码。
9. 初始化完毕，将地址赋值给指定的引用。

#### 6. 局部代码块

是在方法里面的一个代码块，用来控制局部变量的生命周期。如：

public static void main(String[] args){

int a = 3;

{

int b = 4;

System.out.println(“局部代码块”);

}

System.out.println(“a = “ + a);

System.out.println(“b = “ + b);

}

上面代码的输出顺序和结果是什么？

首先，运行会报错，因为b是局部代码块中的局部变量，外部无法访问。

修改错误后，会先打印“局部代码块”，再打印a = 3。

### 2.1.5 设计模式——单例设计模式

设计模式本来是建筑行业的一种用语，在盖房子之前先进行设计，提前想好一些方案，以免房子改好之后要进行返工。比如，盖别墅有盖别墅的设计图，欧式建筑有欧式建筑的设计图，那么以后盖类似房子的时候，就可以参考已有的设计图了。

延伸到程序开发领域，也是一样的，有一些问题是可以被归纳出解决思路的。所以设计模式就是解决某一类问题的一种思想。Java当中归纳出了23种有效的设计模式。

我们这里先介绍一种，叫做单例设计模式。

#### 1. 单例设计模式——思想

解决的问题是： 保证一个类（Single）的对象在内存中只有一个。

应用场景： 多个程序都要操作同一个对象时（如，配置文件对象），程序A 修改配置文件的话，程序B能马上知道并且能拿到程序A修改后的数据。

怎么实现呢： 保证这个类只能创建一个对象。

解决思路： 1. 正常情况下，一个类只要暴露构造函数，那么其他任何程序都可以通过new来新建对象。那么解决思路就是不要暴露构造函数，让别的程序能用new 创建对象。代码如下：

class Single{

private Single(){} // 私有构造函数

}

2. 通过1的思路，别的程序已经不能创建对象了，但是我们的需求是让内存中只有一个对象，所以该类本身要自己要用new来创建这个唯一的对象。

class Single{

private Single(){} // 私有构造函数

Single s = new Single(); // 本类自身创建的对象

}

3. 对外提供访问这个对象的方法，就可以了。

class Single{

private Single(){} // 私有构造函数

Single s = new Single(); // 本类自身创建的对象

public Single getInstance(){ // 公共方法可以获取对象

return s;

}

}

至此，我们以为我们已经解决了问题，但是让我们来假设从别的类的main方法中要如何获取这个对象？

我们要调用方法的话，必须要先有对象，但是这个方法本身就是为了获取唯一的对象，所以不可能使用对象获取，那么就想到用类名来获取，类名可以调用的方法必须是静态的，所以，getInstance() 方法必须是静态方法。

这个方法里返回了s，如果方法是静态的话，它不能访问非静态的变量，所以s也要是静态的，那么代码就是：

class Single{

private Single(){} // 私有构造函数

static Single s = new Single(); // 本类自身创建的对象

public static Single getInstance(){ // 公共方法可以获取对象

return s;

}

}

#### 2. 单例设计模式——细节

上面的代码，我们现在想在另一个类的main方法中获取Single对象的话要这么写：

Single single = Single.getInstance(); 这当然是可以的，但是聪明的你也许已经发现了，这并不是唯一

的方式，我还可以这么写： Single single = Single.s; s是Single类的静态成员变量啊，我当然也可以

直接用类名调用它。

这在代码上来说是没问题的，但是在设计上来讲，不太好。就跟我们要把成员变量私有化，而是

通过get和set方法操作成员变量是一个道理。所以这里，也要把s私有化，最终代码是：

class Single{

private Single(){} // 私有构造函数

// 本类自身创建的对象，类加载时就创建。私有化，外部不能直接访问。

private static Single s = new Single();

public static Single getInstance(){ // 公共方法可以获取对象

return s;

}

}

#### 3. 单例设计模式——懒汉和饿汉

上面的那种方式在类加载时就实例化对象了，实际上单例设计模式还有另一种写法：

class Single{

private Single(){} // 私有构造函数

// 类加载时这个对象显示初始化值设置为null.

private static Single s = null;

public static Single getInstance(){ // 外部第一次要获取对象时，才初始化对象。

if(s == null){

s = new Single();

}

return s;

}

}

这种方式是在外部第一次调用方法时才初始化对象，而不是类加载时就初始化，所以又叫做延迟加载方式。那么猜一下，这两种方式，哪个是懒汉哪个是饿汉呢？

第一种是饿汉，因为表现比较急切，类刚加载就初始化对象，第二种叫懒汉，拖到有人用时才初始化。

## 2.2 继承（Extends）

### 2.2.1 继承的思想和特点

老规矩，介绍概念之前，还是先从生活出发思考。我们的身份是 学生，可以创建学生类，我们还有老师，也可以创建老师类。那么这两个类的内容分别是：

|  |  |  |
| --- | --- | --- |
| 类 | 属性 | 方法 |
| 学生类 | 姓名，年龄，性别，身高，生活费。。。 | 吃饭，睡觉，学习。。。 |
| 老师类 | 姓名，年龄，性别，身高，工资。。。 | 吃饭，睡觉，教课。。。 |

分析上面的两个类，发现了什么？老师类和学生类有很多共同的属性和方法啊，那么有共同的东西我们写代码的时候是不是会有很多重复的代码？是的。怎么才能不要这些重复的东西呢？有一个很好的办法就是把共同的东西抽出来单独放一块，然后分享给需要的类。这种思想反映在代码的世界里，就是继承。

学生和老师共用的东西抽出来也要放在一个地方吧？当然了，那我们就另一个更通用的类来存放，学生和老师都属于人，那么就创建一个人类，来存储学生和老师公有的属性和方法，让老师类和学生类都继承人类。继承在代码中用 extends 表示，学生类和老师类就叫做子类，而人类叫做父类。

那么用代码表示：

class Person{

private String name;

private int age;

public void eat(){

}

}

class Student extends Person{

……

}

了解了继承的思想之后，我们来总结一下继承有什么特点：

1. 提高代码复用性。子类不用再重写父类的代码。
2. 让类与类之间产生了关联。

### 2.2.2 单继承和多继承

单继承就像字面意思一样，意味着一个类只能有一个父类，相当于一个人只能有一个父亲。

多继承是指一个类可以有多个父类。

Java是只支持单继承的，不支持直接多继承。

多继承会存在问题，比如学生类如果同时继承人类和生物类，人类和生物类都有吃饭方法，那么当学生类调用吃饭方法时，会产生不确定性。正是如此Java才规定，一个类只能有一个父类。

但是Java支持多层次继承：比如学生类继承人类，人类继承生物类。

多层次继承会产生继承体系，就像家谱一样，有好多层，

### 2.2.3 继承关系中父类和子类的成员

**1. 父类和子类中的成员变量**

举例说明：

class Father{

int num = 4;

}

class Child extends Father{

private int num2 = 5;

public void show(){

int num3 = 6;

System.out.println(num); // 输出什么？

System.out.println(num2); // 输出什么？

System.out.println(num3); // 输出什么？

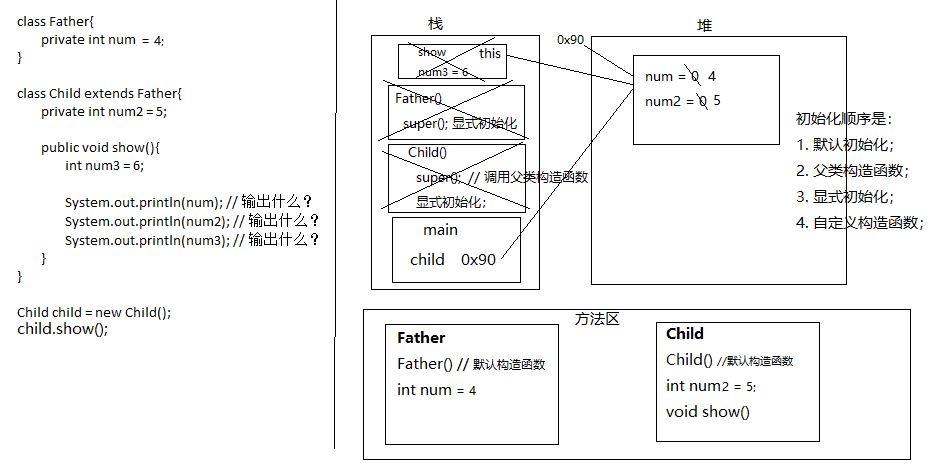
}

}

Child child = new Child();

child.show();

上面这整个过程在内存中的体现？画图表示：



如果我改变了代码，如下，那么如何才能打印6,5,4不同的值呢？

class Father{

int num = 4;

}

class Child extends Father{

private int num = 5;

public void show(){

int num = 6;

System.out.println(num); //猜猜这里输出什么？思考一下内存存储就知道了

}

}

Child child = new Child();

child.show();

总结：

1. 当本类中局部变量和成员变量同名时，用this关键字来获取成员变量。
2. 当子类和父类中出现了同名的成员变量时，用super关键字来获取父类中的成员变量。

**2. super关键字**

super 和 this关键字的用法很相似：

1. this代表对象的引用.可以用System.out.println(this) 来打印this，它是一个地址
2. super 代表父类的那片空间，并不代表对象。不能打印super，会编译报错。

每个类加载进内存时，都会有一个super，super指向父类。所以用super. 就可以访问父类的成员。

**3. 父类和子类中的成员函数**

class Father{

void show(){

SOUT(“Father show run”);

}

}

class Child extends Father{

void show1(){

SOUT(“Child show run”);

}

}

Child child = new Child();

child.show();

child.show1();

Child 类在方法区中持有super，指向Father类。调用方法时会现在Child类中寻找有没有该方法，如果找到就直接加载进栈执行，否则就去找父类，然后加载进栈执行。

当子类和父类中出现一模一样的方法时，子类会运行自己类中的方法，这种情况称之为覆盖/重写（override）。但在内存中，父类的方法区内实际上还是持有自己的方法。

我们之前还学习过重载（overload），那么重载和重写有什么区别呢？

1. 重载一般发生在同个类里，是指方法被重复载入，只跟参数列表有关
2. 重写发生在子父类之间，子父类有共同的方法，返回类型，方法名，和参数列表完全一样。

**重写的应用：**

当子类需要更先进的方法来实现同一个功能时，就不要再定义新方法了，可以重写父类的方法，而且父类已经实现的功能也不需要再重写了，直接调用super.func() 就可以，然后下面再写子类独特的功能。如：老式手机的来电显示只显示电话号码，而新式手机的来电显示可以显示电话号、姓名和头像。

父类的来电显示：

show(){

System.out.println(number);

}

字类的来电显示：

show(){

super.show();

System.out.println(name);

System.out.println(picture);

}

重写的注意事项：

1. 子类重写父类的方法时，要保证子类的该方法的权限要大于等于父类的权限。否则会编译报错。
2. 静态方法只能重写静态方法，或者被静态方法重写。

**4. 子类的实例化过程**

子类所有的构造函数第一行默认都有一个隐式的super(); 来调用父类的空参数构造函数。

子类的对象初始化时的过程是：

1. 内存中默认初始化：int类型默认值为0，boolean为false，对象为null
2. super(); 隐式调用父类构造函数。
3. 显示初始化：构造函数中可能会显示的给定成员变量的值。
4. 本构造函数内自定义的初始化内容。

当子类继承父类时，默认都会调用父类的空参数构造函数，但是如果父类中没有空参数构造函数时，编译就会报错，那么这时怎么办呢？

当然可以在父类中创建一个空参数构造函数，但是不建议这么做，因为构造函数的目的是为了初始化成员变量值。如果不需要空的构造函数就没必要特意创建一个。可以用显示指定super的方式来调用父类中的构造函数，如：super(5);

用来调用父类构造函数的super语句在子类的构造函数中，必须定义在第一行，因为父类的初始化要先完成。

扩展内容：

this() 必须放在构造函数第一行，super()也必须放在构造函数第一行，有了this()就没有super() 了。因为this会访问本类另一个构造函数，那个构造函数会有隐式的super();在第一行。

为什么它们要放在第一行？因为初始化过程必须先完成才能进行别的操作。

### 2.2.4 final关键字

继承在某些方面有一定的风险，子类继承父类可能会重写父类的方法，但是有可能父类的方法里访问了电脑系统数据或电脑系统文件，子类重写之后可能会修改这些系统内容而导致系统崩溃。

所以为了降低这种风险，父类可以用final关键字来限制子类可以重写的内容。

1**. final关键字的特点：**

1. final关键字可以修饰类，也可以修饰方法和变量，包括成员的和局部的。
2. final修饰的类不可以被继承。
3. 被final修饰的方法不可以被重写。
4. 被final修饰的变量是常量，一定要被显示初始化，且被赋值后不可以被修改。final最终化的不是默认值，而是显示初始化的值。

所以开发中用到的那些不变的数据，需要用阅读性强的名称来定义，并把该数据final化，而名称的定义规范是：全部子母都大写，如果由多个单词组成，用\_连接。

**2. final关键字的应用**

final关键字会被应用在很多地方，但是有一个经典的地方，就是我们之前学过的饿汉式单例模式.我们之前是这么写饿汉模式的：

class Single{

private static Single s = new Single();

public Single getInstance(){

return s;

}

}

这时，当别人通过getInstance() 拿到唯一的Single对象后，可能对它进行修改，导致一些问题。我们现在学了final了，就知道可以用final修饰对象，保证它不会被修改：

class Single{

private static final Single SINGLE\_INSTANCE = new Single();

public Single getInstance(){

return SINGLE\_INSTANCE;

}

}

### 2.2.5 抽象类

老规矩先不要考虑抽象类这个概念，从生活中思考，我们都知道狗具有吼叫的功能，狼也具有吼叫功能，但是呢？狼不可能继承狗，狗也不可能继承狼，所以我们就把它们的共性抽离出来，作为犬科，都有吼叫功能。所以抽离如下：

狗：吼叫——汪汪

狼：吼叫——嗷嗷

犬科：吼叫？犬科的吼叫怎么写呢？

你现在可能会想到，那我们就让犬科的吼叫是空方法，反正它的子类会重写它。但是如果这么做，子类是不是也可以选择不重写，假如狗没有重写吼叫方法，那么狗就没法叫了？显然不合理。所以，我们要让狼和狗都必须重写犬科的这个方法，但是又会发现，犬科的吼叫这个行为很难去具体的描述，不同的犬科动物吼叫的方式也不一样啊，这就引申出了抽象的概念。

所以描述一个事物，却没有足够的信息，这时就把这个事物定义为抽象事物。

生活中其实很多这种难以描述的抽象事物，它们也有特定的属性和行为，但是行为很难进行具体的描述，所以行为方法就没有具体内容，也就是没有{}，那么定义就是：

void bark();

但是这种定义会和正常的方法搞混掉，为了区别，给它一个特定的关键字，就是abstract，那么方法定义就是：

abstract void bark();

**1. 抽象的特点**

1. 抽象方法一定定义在抽象类中，且一定要用abstract关键字。
2. 抽象类不能实例化，不能用new关键字创建对象。
3. 只有子类覆盖了所有的抽象方法后，子类就具体化了，子类就可以创建对象。

如果子类没有覆盖所有的抽象方法，那么子类还是抽象类。

接下来有几个问题，思考一下：

1. 抽象类中有构造函数吗？

有，抽象类的构造函数虽然不能给抽象类实例化，但是抽象类可能有子类，它的构造函数可以给子类实例化。

抽象类和一般类的异同点：

相同点：都是对事物进行描述。

不同点：抽象类描述事物的信息不具体，一般类描述事物的具体信息。

抽象类可以定义抽象方法，一般类不行，抽象类不能实例化，一般类可以。

1. 抽象类一定是父类吗？

是的，抽象类可能是别的抽象类的子类，但是它一定会是一些类的父类，因为只有被子类覆盖抽象方法后，才能实例化并使用这些方法。

1. 抽象类中可以不定义抽象方法吗？

可以。仅仅是为了让该类不能被实例化。

1. 抽象关键字abstract不能和哪些别的关键字共存呢？

final：final修饰的类不能有子类，而abstract修饰的类一定会有子类。

private：private修饰的东西不能被子类访问。

static：静态修饰的东西会直接进方法区，用类名直接访问，但是抽象类直接访问没有意义。

**2. 抽象类练习**

需求：公司中程序员有姓名，薪水，工号，工作内容。项目经理除了这些内容外还有奖金。

对给出的需求进行数据建模。

分析：这个需求中，可以描述的具体事物有：

程序员：

属性：姓名、薪水、工号

行为：工作

项目经理：

属性：姓名、薪水、工号、奖金

行为：工作

分析后发现，程序员和项目经理有共性，但是程序员继承项目经理？不合适，程序员没有奖金啊。那么项目经理继承程序员？不合适啊，项目经理不是程序员的一种啊，没啥所属关系。

那么这种情况下，就可以向上抽取，在公司里面，程序员和项目经理都属于什么呢？合适的说法叫员工。

员工：

属性：姓名、薪水、工号

行为：工作

但是员工的工作是具体的吗？不同员工工作肯定不同啊，所有这里的工作是抽象的。转换成代码：

abstract class Employee{

private String name;

private String id;

private double salary;

Employee(String name, String id, double salary){

this.name = name;

this.id = id;

this.salary = salary;

}

abstract void work();

}

// 描述程序员

class Programmer extends Employee{

Programmer(String name, String id, double salary){

super(name, id, salary);

}

public void work(){

System.out.println(“write code”);

}

}

// 描述经理

class Manager extends Employee{

private double bonus;

Manager(String name, String id, double salary, double bonus){

super(name, id, salary);

this.bonus = bonus;

}

public void work(){

System.out.println(“manage”);

}

}

### 2.2.6 接口——interface

学过了抽象类，让我们思考一种特殊的情况，如果一个抽象类中只有抽象方法时，那么就引入了另一种描述事物的方式，接口。

如：

abstract class Person{

abstract void eat();

abstract void sleep();

}

这时可以转换为接口：

interface Person{

public abstract void eat();

public abstract void sleep();

}

如果你自己尝试编译interface，你会发现生成的二进制文件依然是.class。这说明它依然是个类，只是表现形式是interface。

**1. 接口的特点**

interface定义和class定义的不同：

接口中常见的成员有两种：1.全局变量：public static。2.抽象方法：public abstract。观察发现它们有固定的修饰符，实际开发中就算你不写，编译时会自动补上。

1. 接口不可以实例化。
2. 接口是用来被实现（implements）的，而不是被继承的。
3. 实现了接口的类必须覆盖接口中所有的抽象方法，然后该类才能实例化。

class Man implements Person{

public void eat(){

}

public void sleep(){

}

}

此时，你就可以实例化Man了。

**2. 接口的好处**

我们之前讲过一个概念叫多继承，即使JAVA不支持，但是它有自己的特点：多继承可以让子类具备更多的功能，但是多继承可能会导致调用的不确定性，比如两个父类都有同一个方法，那么子类调用时就不确定调用哪个父类的方法。

在JAVA中，就通过接口来解决这个问题，实现既能多实现，又不会有调用的不确定性：

接口来定义功能（方法），让子类来实现内容（方法体）：

interface InterA{

public void show1();

}

interface InterB{

public void show2();

}

calss SubInter implements InterA, InterB{

public void show1(){}

public void show2(){}

}

上面这种方式我们很好理解，就是一个类实现多个接口：多实现。

但是如果接口中有同样的功能呢？

interface InterA{

public void show();

}

interface InterB{

public void show();

}

calss SubInter implements InterA, InterB{

public void show(){

System.out.println(“SubInter show”);

}

}

这里是不会产生调用的不确定性的，因为实现是在子类中，唯一且独立。

重点：子类可以同时继承一个父类并实现多个接口。

继承是为了获取的基本功能，想要扩展功能可以通过实现接口来完成。

**3. 接口可以多继承**

类与类之间的关系：继承关系，意味着 is a

类与接口之间的关系：实现关系，意味着 like a

那么接口与接口之间是什么关系呢？是继承关系，一个接口也可以继承多个别的接口。

**4. 没有抽象方法的抽象类**

当我有一个接口，有4个方法，我需要一个类A只用该接口的方法1，另一个类B只用该接口的方法3，那么该怎么办？

初步的思路是类A和类B都实现该接口，就可以解决问题。

但是这种方式，很浪费，因为A和B虽然只用一个方法，但是实现接口后必须重写接口的所有方法。代码复用性很差。有没有更好的办法？当然有。

A和B都有很多重复的代码，那么就把重复内容抽取出来，成为一个父类P，让P实现接口，P实现了方法，但是方法体为空，让A和B继承P即可，A和B这时只需要重写自己需要的方法。

那么既然父类P的方法都是空方法体，他就没有必要创建对象了，既然不用创建对象，就可以把P作为抽象类。

这时我们就知道了，没有抽象方法的抽象类是存在的，它的作用就是为了方便实现接口，内部的方法全空实现。

**5. 接口的思想**

狭义的接口就是指java中的interface。

广义来讲，凡是对外提供的都可以叫做接口，比如USB接口，机箱，暴露的公共方法等。

1. 接口的出现扩展了功能。
2. 接口其实就是暴露出来的规则。
3. 接口的出现降低了耦合性：耦合性是指事物之间的紧密程度，接口的出现使得本来需要紧密相连的东西，如笔记本要想使用鼠标本来是要把鼠标线焊在笔记本上，USB接口的出现使得笔记本和鼠标独立，鼠标只要通过接口插入就可以和笔记本联合使用。

接口的出现，一方在实现接口，一方在使用接口。

**6. 抽象类和接口的区别**

通过前面的学习，我们知道抽象类可以定义抽象方法，接口也可以定义抽象方法，那么什么时候用抽象类什么时候用接口呢？

比如，犬这一类动物，有抽象的行为吃，也有抽象的行为叫。那么我们用抽象类定义犬呢，还是用接口定义犬呢？

在代码实现方面，两种都可以，但是根据现实生活分析，当我们要扩展功能时，比如我们有一种犬叫缉毒犬，那么它适合继承呢？适合实现呢？我们知道继承意味着is a，实现意味着like a。那么缉毒犬显然是is a犬。所以犬更适合定义为abstract class.

但是，如果是定义具有缉毒功能的事物，取名缉毒者，比如缉毒犬，缉毒机器人，缉毒警察等等。那么要抽取成类呢？还是接口呢？

这时以缉毒犬为例，它已经继承了犬类，那么就不能再继承缉毒者类了，所以这时，缉毒者就要抽离成接口。缉毒犬就继承犬类，实现缉毒者接口。

分析就发现，原来类是描述某种事物的基本且共同的功能，而接口用于定义事物的额外功能。

那么抽象类和接口的区别就是：

1. 类与抽象类之间是继承关系，is a。类与接口是实现关系，like a。
2. 抽象类中可以定义抽象和非抽象方法，子类可以直接使用或者覆盖使用。接口中定义的都是抽象方法，子类必须实现才能使用。
3. 抽象类是描述某种事物的基本且共性的功能，接口描述的是事物的额外功能。

## 2.3 多态——（Polymorphism）

多态，字面意思就是多种形态。一个事物的多种形态，是不是还挺熟悉的？是的，我们之前学过的重载，就是方法多态的表现，重写也是一种多态，是父类和子类的多态。

我们这里引入的多态，是指对象的多态。

### 2.3.1 多态的思想和特点

老样子，还是先举例说明。比如狗类，我们之前怎么创建对象？ Dog dog1 = new Dog();

但是学过继承后，我们知道狗可能会继承动物类，Animal.那么狗就是动物的一种了，可不可以这么写？Animal dog = new Dog();

当然可以，我们new Dog() 创建的是Dog类型的对象，这是一种形态，但是同时它又是一种Animal，这就是另一种形态，这就是对象的多态的一种体现。

多态在程序中的体现：父类或者接口的引用指向了子类的对象。

多态的好处：

1. 提高代码的扩展性：只要是共性的对象，都可以接收。

多态的弊端：

1. 不能使用子类的特有方法。

多态的前提：

1. 必须有关系，继承，实现。
2. 通常有覆盖。

### 2.3.2 向上向下转型

我们上面写的 Animal a= new Dog(); 这个专业术语叫做向上转型。跟我们最初学的基础数据类型的自动类型提升类似。

向上转型的好处：隐藏了子类型，提高了代码的扩展性。

向上转型的弊端：只能使用父类中的功能，不能使用子类特有功能，功能被限定了。

所以，当我们不需要面对子类型，只需要使用父类的功能即可完成操作时就可以使用向上转型。

如果我们希望上面的a对象拥有Dog类特有的功能呢？这时意味着我们需要子类型，那么就使用向下转型，这相当于我们之前学的强制类型转换：

Dog d = (Dog) a;

向下转型的好处：可以使用子类特有的功能。

向下转型的弊端：必须得面对具体的子类型。向下转型有风险：只要转换类型和对象类型不匹配，运行时会有问题。

万一你做了 Cat c = (Cat) a; 编译没问题，因为编译时不产生对象，只检查语法。但是运行时会报错，因为运行时就知道a实际上不是Cat 而是Dog.

为了避免这个问题，强制类型转换之前，先判断转换类型和对象类型是否匹配，使用instanceof 关键字判断：

If(a instanceof Cat){

Cat c = (Cat) a;

}

记住一点：这里的转型，转的都是引用，而不是对象。对象还是那个对象，只是引用变了。

根据前面的内容，我们来做一个练习，分阶段来进行：

1. 笔记本电脑运行。

class Laptop{

public void run(){

System.out.println(“Laptop run”);

}

}

1. 想使用一个鼠标，说明笔记本中多了一个使用鼠标的功能，鼠标是另一种事物，所以创建鼠标类。

class Mouse{

public void open(){ System.out.println(“Mouse open”); }

public void close(){ System.out.println(“Mouse close”); }

}

class Laptop{

public void run(){

System.out.println(“Laptop run”);

}

public void useMouse(Mouse m){

m.open();

m.close();

}

}

1. 如果还想使用其他设备呢？比如键盘、U盘、外置硬盘、耳机等。

首先想到的是，可以像2一样，通过在笔记本对象中继续添加对应的方法。可行吗？可行，但是扩展性非常差，每次要用东西就要修改笔记本类。

思考，笔记本要使用其他设备，这个设备确定吗？不确定，但是每多一个设备笔记本就增加一个功能，那么笔记本和设备间的耦合性很强，如果才能解耦呢？

我们不需要面对具体的设备，只需要指定一个规则，让这些设备都满足这个规则，这样笔记本只需要面对规则就可以。

JAVA中可以通过定义接口来指定规则，实现解耦。

思路：

1. 定义规则：

interface USB{

void open();

void close();

}

1. 描述笔记本：

class Laptop{

public void run(){

System.out.println(“Laptop run”);

}

// 定义了一个接口类型的引用，可以接收任何实现了USB接口的对象

Public void useUSB(USB usb){

usb.open();

usb.close();

}

}

1. 描述外围设备：

class Mouse implements USB{

public void open(){ System.out.println(“Mouse open”); }

public void close(){ System.out.println(“Mouse close”); }

}

### 2.3.3 多态中对成员（变量和函数）的调用

**1. 多态中对成员变量的调用**

class Father{

int num = 4;

}

class Child extends Father{

int num = 6;

}

class Main{

public static void main(String[] args){

Father f = new Child();

System.out.println(f.num); // 这里输出什么？答案是4，因为类型提升过后，现在是Father 类型了，自然去找Father的变量。

}

}

再思考，如果Father没有定义num属性时，上面会输出什么？

会编译报错，因为Father类型没有num属性，所以f.num会编译失败。

总结：在多态调用时，如果子父类有同名成员变量，只看等号左边的引用所属类中的变量。这是因为在运行时，堆内存中的对象里会持有两个同名的变量，一个是父类的，一个是子类的，既然引用是父类的引用，所以自然拿到了父类的变量。

**2. 多态中对成员函数的调用**

class Father{

void show(){ SOUT(“Father”);} // SOUT 是打印函数的缩写

}

class Child extends Father{

void show(){ SOUT(“Child”);}

}

class Main{

public static void main(String[] args){

Father f = new Child();

f.show(); // 这里输出什么？答案是”Child”

}

}

为什么输出的是Child？实际上这里编译时看的还是Father中的show()，因为编译时没有对象产生。运行时，调用方法是去方法区找方法，子类重写了父类的方法，所以直接调用的是子类的show()。

总结：当子父类有同样的成员函数，多态调用时，编译时看的是引用所属类中的方法，运行时看的是对象所属类中的方法。

**3. 多态中对静态函数的调用**

class Father{

void static show(){ SOUT(“Father”);}

}

class Child extends Father{

void static show(){ SOUT(“Child”);}

}

class Main{

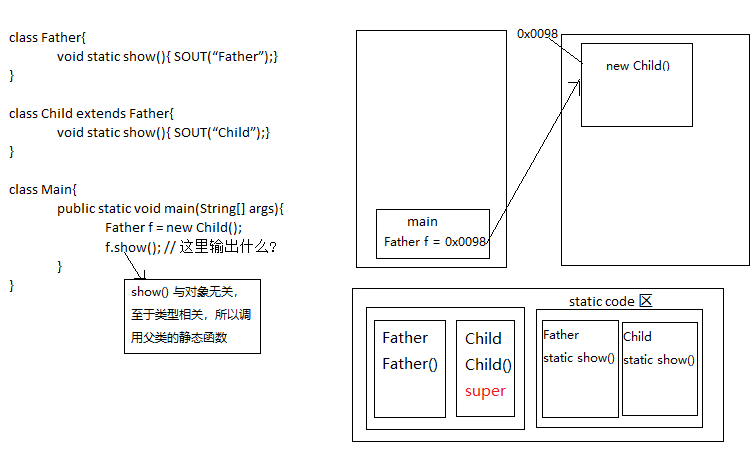
public static void main(String[] args){

Father f = new Child();

f.show(); // 这里输出什么？答案是”Father”，用下面的内存图解释

}

}



总结：当子父类有同样的静态函数，多态调用时，编译时看的是引用所属类中的方法，运行时看的也是引用所属类中的方法。因为静态函数与对象无关，只和类有关。

来做个练习：

class Father{

int num = 5;

void show(){

SOUT(this.num);

}

}

class Child extends Father{

int num = 6;

}

class Main{

public static void main(String[] args){

Father f = new Child();

f.show(); // 这里打印什么？打印5，因为this表示的是引用，是本类类型的引用，这里的本类是Father，因为show() 方法来自于Father。

}

}

# Object 对象

我们之前有了解过，所有的类都有父类，最顶层的父类就是Object.其中定义了所有对象都具有的功能，我们要对这个基础类中的功能有点了解。

JAVA对外提供了API（Application Program Interface）文档，其中就包括Java内部的很多类的介绍，这里我们关注Object 类，它是在java.lang包中，java.lang 包是指java的语言包，里面包括了java一些核心内容的介绍。

类Object是java类层次结构的根类，所有类最终都以Object作为父类。

Java语言是开源的，所以你可以到你安装的jdk目录下找到一个src.zip文件，打开它可以查看源代码。查看Object.java 会有一些发现：

1. Object.java 中并没有自己定义构造函数，所以它只有默认的空参数构造函数。
2. Object.java中有很多native方法，它的第一行就是一个native 方法，然后接下来是静态代码块执行这个native方法。这里先对native做一个简单的介绍，它也是一个修饰符，意思是本地方法，实现方式跟系统底层语言相关，比如windows系统，这些native方法就是用C实现的。

native方法与abstract方法一样，是没有方法体的，因为要在本地系统实现。

它在内存里的存储有另一个独特的内存区域：本地方法区。

## 3.1 boolean equals(Object obj)

判断其他对象是否与此对象“相等”。equals方法在非空对象引用上实现相等关系。

对于任何非空引用值，当且仅当x和y指向同一个对象时，此方法才返回true，此时x.equals(y) 相当于 x == y。

但是任何非空引用值x，x.equals(null) 都返回false。

那我们知道，Object是所有类的超类，现在我们自己来创建一个类：

class Person{

}

public class Main{

public static void main(String[] args){

Person p1 = new Person();

Person p2 = new Person();

SOUT(p1.equals(p2)); // 这么写可以吗？

}

}

问：p1.equals(p2) 可以直接用吗？

答：可以，因为Person类默认继承Object类，所以自然也拥有父类的equals 方法。

问：结果是什么？

答：false，继承自Object 的equals方法默认比较的是引用指向的对象，p1和p2指向的是不同对象，所以结果是false。

人都是有姓名和年龄的，那么我们改进一下Person类。

class Person{

private String name;

private String age;

public Person(String name, int age){

this.name = name;

this.age = age;

}

}

public class Main{

public static void main(String[] args){

Person p1 = new Person(“Ady”, 21);

Person p2 = new Person(“Ady”, 21);

SOUT(p1.equals(p2)); // 结果是什么？

}

}

问：上面打印的结果是什么？

答：是false。 p1 和 p2指向两个不同的对象，即使这两个对象的属性是相同的。

在实际的应用当中，我们很多时候并不关注两个引用是不是指向同一个对象，而关注的是具体的值。比如，我们知道张三和李四是不同的人，但是我们就是想知道他们是不是有相同的年龄，该怎么办？

首先想到的是，我们可以在Person 类中定义一个方法，比较年龄就好了。这当然是可行的。

但是我们既然已经学过继承和重写了，就知道我们也可以在Person类中重写equals方法，让它不要比较是否是同一个对象了，而是比较对象的值。那么我们在Person类里来重写equals方法：

// 既然是重写，那么返回值类型，方法名和参数列表都要相同，所以这里的参数必须是Object 类型的

public boolean equals(Object obj){

return this.name == obj.name && this.age == obj.age;

}

问：上面重写的方法有问题吗？

答：有问题。编译就会报错：obj对象根本没有name和age属性。

我们调用上面的方法时，传入的对象是Person类型的对象，但是由于参数类型定义的是Object类型，所以Person对象被自动提升成了Object对象，这就是多态中的向上类型转换。

那么为了能作比较，我们就必须把它转换回Person类型，做向下类型转换，但是有可能我们调用时传入的不是Person对象呢？那么强制转换前必须要做判断：

public boolean equals(Object obj){

if(!(obj instanceof Person)){

return false;

}

Person p = (Person) obj;

return this.name == p.name && this.age == p.age;

}

问：上面重写的方法还有问题吗？

答：有问题。Person 类的成员变量name是String类型的，String是一个类，所以

this.name == p.name 实际上是在比较这两个name是否指向同一个String对象，但是我们实际

要比较的是值，所以对于String类型的值比较，不要用 ==，而要用String 类重写后的equals

方法：

public boolean equals(Object obj){

if(!(obj instanceof Person)){

return false;

}

Person p = (Person) obj;

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

}

那么到了这里，我们已经完成了Person类中equals方法的重写，不过在效率上还可以做一点改善。思考，如果我们调用equals方法时传入的对象是同一个对象呢？那么还要对它进行类型转换然后再比较值，这就没必要了，同一个对象那肯定是相同的啊。所以我们在类型转换前进行判断，下面就是最终的equals方法了：

public boolean equals(Object obj){

if(this == obj){

return true;

}

if(!(obj instanceof Person)){

return false;

}

Person p = (Person) obj;

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

}

## 3.2 String toString()

toString() 返回该对象的字符串表示。

当我们用System.out.println() 打印一个对象时，比如打印Person类的对象p1。得到的结果会是一个像：Person@1afae45 的字符串。这是为什么呢？

当System.out.println(p1) 时，会默认调用p1.toString() 方法，也就是说实际上这个语句执行的是：

System.out.println(p1.toString()).

对象.toString()的值有固定的格式：类名@对象哈希码的无符号十六进制表示。也就是说，值等于：

getClass().getName() + @ + Integer.toHexString(hashCode())

默认的这个值的形式对我们而言没啥意义，我们打印一个对象通常希望得到的是该对象的内容信息，而不是它的hash值。所以，最好是重写这个方法，还是以刚刚的Person类为例：

public String toString(){

return “Person[name = ” + this.name + “, age = “ + this.age + “]”;

}

# 内部类

内部类是指把类定义在别的类的内部。什么时候需要用内部类呢？

## 4.1 内部类的思想和特点

当A类需要直接访问B类中的成员时，可以将A类定义在B类内部，作为B类的内部类存在。

这么说可能有点难以理解，那么从生活的角度来看内部类，我们知道人体内部有很复杂的器官组成，心脏，胃，肺等等，那么它们是不是就相当于与人的内部类？可以直接访问人的血液啊、食物啊等等成员。

内部类和外部类的访问规则：

内部类可以直接访问外部类的成员，包括成员变量和成员函数。

外部类要想访问内部类，需要创建内部类的对象来访问。

内部类编译后也会生成对应的class文件，但是文件名会是：外部类$内部类.class

## 4.2 非静态非私有内部类

其实整体上来看，内部类就相当于外部类的一个特殊成员，那么它就可以被成员修饰符来修饰：

public， private，static等。

如果内部类权限是非私有的，就可以在外部其他程序中被访问到，如，有一个外部类Outer， 内部有一个非私有的内部类 Inter，那么在外部就可以用：

Outer.Inter in = new Outer().new Inter() 访问内部类。

这里需要注意的是：new Outer()时，内存里的方法区只加载Outer类，而不会加载Inter类，因为Inter类有自己独立的class文件.

## 4.3 静态非私有内部类

静态内部类相当于一个外部类，因为static修饰的类会最先被加载进静态方法区。

这里需要明确一点，如果一个类声明为static，它只能是内部类。外部类是无法用static修饰的，编译会失败。

如：

class Outer{

private static int num = 4;

static class Inter{

public void show(){

SOUT(“num = ” + num);

}

}

}

那么在外部，如何访问show呢？

Outer.Inter in = new Outer.Inter() 就可以创建一个内部类对象，然后用in.show() 调用。

注意，这时我们不需要实例化外部类就可以创建内部类对象。因为内部类是静态的，一旦外部类被加载，它也马上被加载。但是show是非静态方法，所以必须创建内部类对象才能访问。

上面的show方法时非静态的，那么如果静态内部类里有个静态方法呢？

class Outer{

private static int num = 4;

static class Inter{

public void show(){

SOUT(“num = ” + num);

}

public static void show2(){

SOUT(“num = ” + num);

}

}

}

这时，可以直接用Outer.Inter.show2()来访问。

注意：非静态内部类中只能定义非静态成员和静态的常量，不允许定义别的静态成员（静态变量和静态函数）。

为什么？非静态内部类如果要被访问，必须要在外部创建对象，而静态成员不需要创建对象就能访问，所以两者相悖。那么为什么又能定义静态常量呢？这是因为静态常量是存储在常量池里，以后不会被更改，所以静态常量在编译时就直接用值表示了。

## 4.4 内部类细节

1. 内部类为什么可以访问外部类成员？因为内部类默认持有外部类.this。举例：

class Outer{

private int num = 4;

class Inner{

public void show(){

System.out.println(“num…” + num);// 这里其实默认是 Outer.this.num

}

}

}

那如果show内定义了局部的 int num 呢？打印的就会是局部变量。

如果Inner类定义了自己的成员变量int num呢？如果才能访问这个num？用this.num访问。

1. 局部内部类不允许访问局部变量，为什么？因为生命周期不同。局部变量的生命周期比内部类对象的短。举例：

class Outer{

private int num = 4;

public void method(){

int num = 7;

class Inner{

public void show(){

System.out.println(“num…” + num);// 这里会怎样？编译失败

}

}

}

}

## 4.5 匿名内部类

匿名内部类就是简化书写的内部类。

想写匿名内部类必须有前提：

内部类需要继承外部类或实现外部接口。

匿名内部类格式：

new 父类or接口(){子类的内容}

匿名内部类其实就是一个子类对象。

使用匿名内部类时，最好父类或者接口里的方法不超过两个。

练习题：

class Outer{

public void method(){

new Object(){

public void show(){

SOUT(“show”);

}

}.show();

}

}

问：上述写法编译通过否？运行通过否？

答：编译通过，没有语法问题，匿名内部类就是个子类对象，然后改对象调用自己的方法。

运行通过，没有逻辑错误。

class Outer{

public void method(){

Object obj = new Object(){

public void show(){

SOUT(“show”);

}

};

obj.show();

}

}

问：上述写法编译通过否？运行通过否？

答：编译失败，obj对象是Object类型，Object类有show方法吗？没有，所以obj.show()编译失败。Object obj = 子类对象，这句话编译没问题的，因为相当于给子类对象做向上转型。