String 类没有提供⽤于修改字符串的⽅法，所以我们将 String 类对象称为不可变字符串，它被声明为 final  
class，所有的属性也被定义为 final 的。但是我们可以修改字符串变量，让它指向另外⼀个字符串。  
为了提⾼内存利⽤率，JVM 有⼀个字符串常量池，每次使⽤双引号定义字符串，JVM 会先到该常量池中来检  
测是否已经存在，存在则直接该对象的引⽤；否则在常量池中创建⼀个新增并返回该值的引⽤。  
使⽤ new 创建字符串（ new String("字符串"); ）时，会直接在堆中创建该字符串并返回其引⽤。从 Java  
6 开始，String 类提供了 intern() ⽅法，调⽤该⽅法时，JVM 去字符串常量池检测是否已存在该字符串，  
如果已经存在则直接返回引⽤；如果不存在则在常量池中添加并返回其引⽤。  
Java 6 中，字符串常量池存在 PermGen ⾥，也就是(“永久代”)，⽽这个空间是有限的，基本不会被 FullGC  
之外的垃圾收集机制扫描。如果使⽤不当，经常会发⽣ OOM。在后续版本中，将字符串常量池放在了堆中，  
⽽且默认缓存⼤⼩也在不断扩⼤。在 Java 8 中永久代 PermGen 也被元数据区 MetaSpace 替代。

格式化  
为了让字符串拼接更简洁直观，我们可以使⽤字符串格式化⽅法 String.format  
%s 字符串  
%c 字符类型  
%b 布尔类型  
%d 整数类型（⼗进制数）  
%x 整数类型（⼗六进制数）  
%o 整数类型（⼋进制数）  
%f 浮点类型  
%a 浮点类型（⼗六进制数）  
%% 百分⽐类型  
%n 换⾏  
示例：  
System.out.printf("hello, %s %n", "world");  
System.out.printf("⼤写a：%c %n", 'A');  
System.out.printf("100 > 50：%b %n", 100 > 50);  
System.out.printf("100除以2：%d %n", 100 / 2);  
System.out.printf("100的16进制数是：%x %n", 100);  
System.out.printf("100的8进制数是：%o %n", 100);  
System.out.printf("100元打8.5折扣是：%f 元%n", 50 \* 0.85);  
System.out.printf("上述价格的16进制数是：%a %n", 50 \* 0.85);  
System.out.printf("上⾯的折扣是%d%% %n", 85);

**String / StringBuilder / StringBuffer**String 在拼接过程中或操作不当时，可能会产⽣⼤量的中间对象。⽽ StringBuffer 就是为了解决这个问题⽽提  
供的⼀个类，StringBuffer 是线程安全的，如果没有线程安全的需要则使⽤ StringBuilder（Java 1.5 中新  
增）。  
StringBuffer 和 StringBuilder 都继承⾃ AbstractStringBuilder 类，⽽ StringBuffer 类的所有⽅法都使  
⽤关键字 synchronized 来保证线程安全。它们的底层都是通过可修改的 char 数组（Java 9 以后改为 byte  
数组实现）来实现修改。以下内容没有特别说明则均基于 Java 8。

System.out.println("3. Java 8新增的 Lambda 表达式");  
Arrays.asList(arr).forEach(numb -> System.out.println(numb));

// 拷⻉数组 arr1 的前 3 个元素  
Integer[] arr3 = Arrays.copyOf(arr1, 3);  
System.out.println(Arrays.toString(arr3));  
// 拷⻉数组 arr1 的第 1 位到第 3 位的元素（不包括第 3 位）  
Integer[] arr4 = Arrays.copyOfRange(arr1, 1, 3);  
System.out.println(Arrays.toString(arr4));  
// 拷⻉数组 arr2 的后 3 位到 arr1 到后 3 位  
System.arraycopy(arr2, 2, arr1, 2, 3);  
System.out.println(Arrays.toString(arr1));

数组排序

Arrays.sort(arr);

举例说明 **Arrays** ⼯具类的常⽤⽅法  
copyOf  
copyOfRange  
fill  
sort  
toString  
equals  
asList  
binarySearch

int[] arr = new int[5]; arr.length

简化格式：

\* 数据类型[] 数组名 = {元素1,元素2,…};

\* a:ArrayIndexOutOfBoundsException:数组索引越界异常

\* 原因：你访问了不存在的索引。

\* b:NullPointerException:空指针异常

\* 原因：数组已经不在指向堆内存了。而你还用数组名去访问元素。

\* int[] arr = {1,2,3};

\* arr = null;

\* System.out.println(arr[0]);

'\x' x表示任意，\是转义符号,这种做法叫转移字符。

'\t' tab键的位置

'\r' 回车

'\n' 换行

\* return和break以及continue的区别?

\* return是结束方法

\* break是跳出循环

\* continue是终止本次循环继续下次循环

面向对象特征

\* 封装(encapsulation)

\* 继承(inheritance)

\* 多态(polymorphism)

**在将⽗类转换成⼦类之前，应该使⽤ instanceof 进⾏类型检查**

**下⾯我们来看⼀下接⼝有哪些特性：  
接⼝不是类，不能使⽤ new 操作符进⾏实例化。  
接⼝变量必须引⽤实现了接⼝的类对象。  
可以使⽤ instanceof 来检查⼀个对象是否实现了某个特定的接⼝。  
接⼝不能包含实例域和静态⽅法，但可以包含常量；接⼝中的⽅法都⾃动被设置为 public，⽽接⼝中  
的域则被⾃动设置为 public static final。  
接⼝中定义的⽅法不能有⽅法体。  
类通过 implements 关键字来实现⼀个或多个接⼝。  
特别注意：Java 8 版本开始对接⼝做了以下增强  
在⽅法声明前添加 default 关键字时，则可以为该⽅法编写⼀个默认实现，这个特征叫默认⽅法或  
者扩展⽅法；但是默认⽅法不能覆盖 Object 中的⽅法，却可以重载 Object 中的⽅法。  
接⼝⾥可以声明静态⽅法，静态⽅法必须有⽅法体。**

⽅法参数  
Java 语⾔总是采⽤按值调⽤（call by value），⽅法得到的是所有参数值的⼀个拷⻉，⽅法是不能修改传递给他  
的任何参数变量的内容。⽅法参数总共有两种类型：  
1. 基本数据类型  
2. 对象引⽤  
⼀个⽅法不能修改⼀个基本数据类型的参数，也不能修改引⽤类型参数的指向，但是却可以修改引⽤类型参数  
指向的对象的值。

隐式参数 this，Java 编译器在编译时会将对象⾃⼰放在第⼀个参数上，我们称之为隐式参数；第⼆个参数开  
始才是位于⽅法名后边括号中的参数，这些是显示参数。使⽤关键字 this 来表示第⼀个隐式参数，因此可以  
在⽅法内使⽤ this. 来访问对象⾃⼰的其他成员变量或⽅法。  
当没有使⽤ this. 前缀来访问⼀个变量时，⾸先看是否存在该名字的局部变量，如果不存在则再去看该对象是  
否存在该名字的成员变量。

当我们需要传同类型的⼀组参数，但是却不知道参数个数，这时可以使⽤可变参数语法：参数类型... 变量  
名。  
⼀个⽅法只能有⼀个可变参数，并且可变参数只能是最后⼀个。可变参数变量我们可以当成是⼀个数组来使  
⽤。

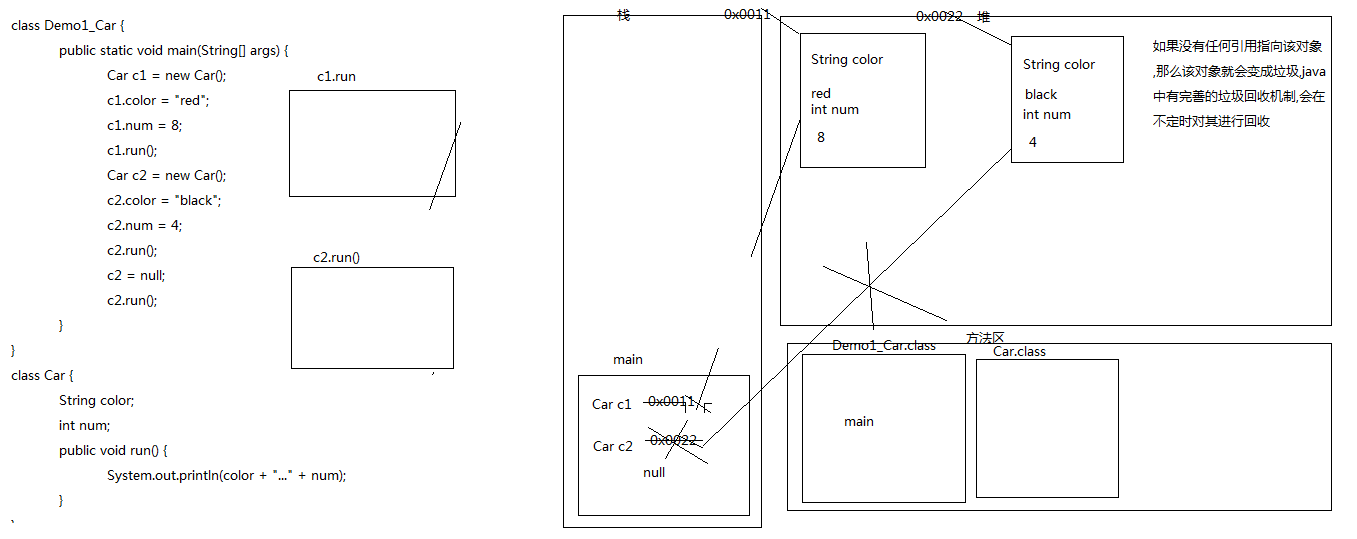
什么是重载、⽅法签名  
重载（overloading）只是⼀个类中可以多个名字相同的⽅法，但这些⽅法的参数个数或类型不同。⽅法签名是  
指⽅法名和参数列表，⽽⽅法返回值类型不是⽅法签名的⼀部分，⼀个类中的⽅法签名必须唯⼀。

new Car().run(); //匿名对象调用方法

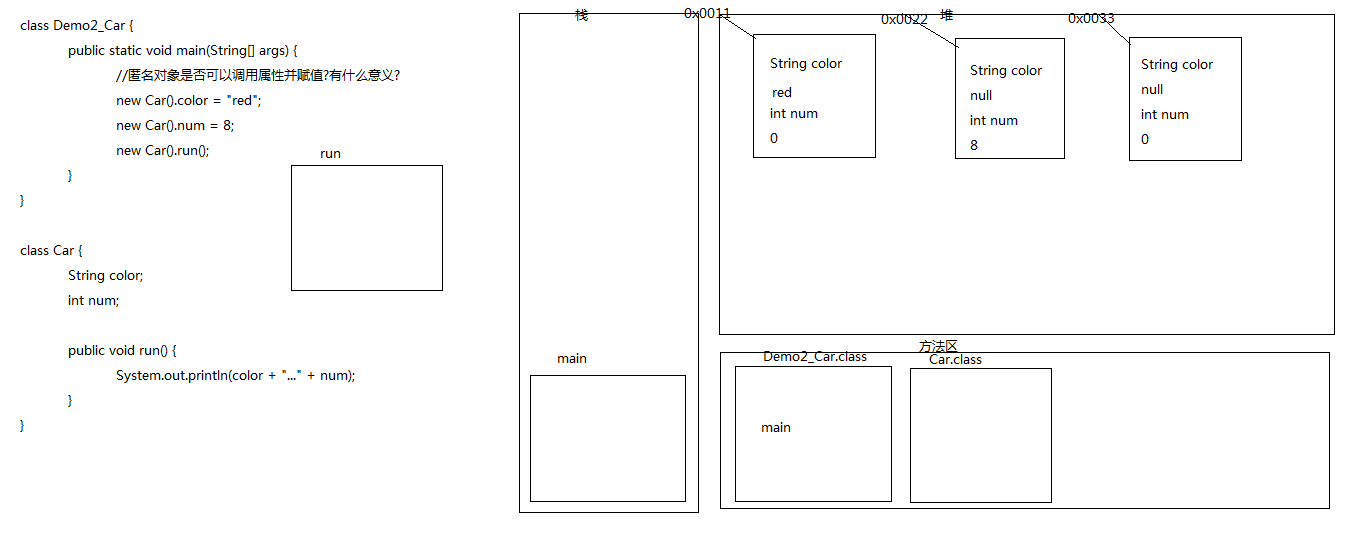
new Car().run(); //匿名对象只适合对方法的一次调用,因为调用多次就会产生多个对象,不如用有名字的对象

method(**new** Car()); //匿名对象可以当作参数传递

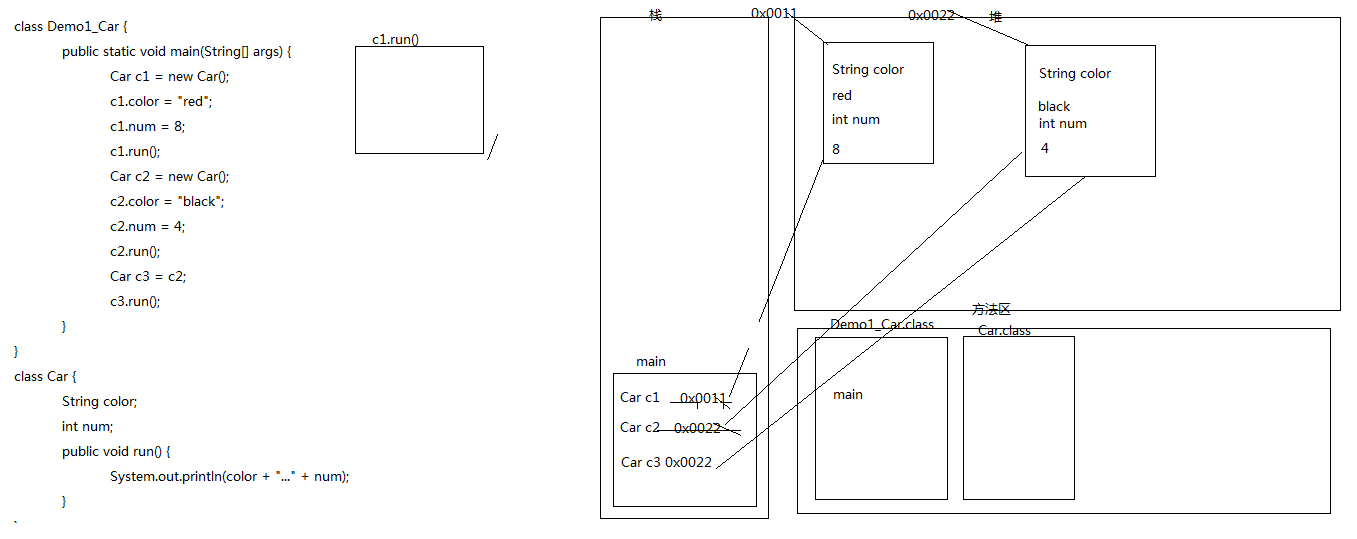
两个对象的内存图.png



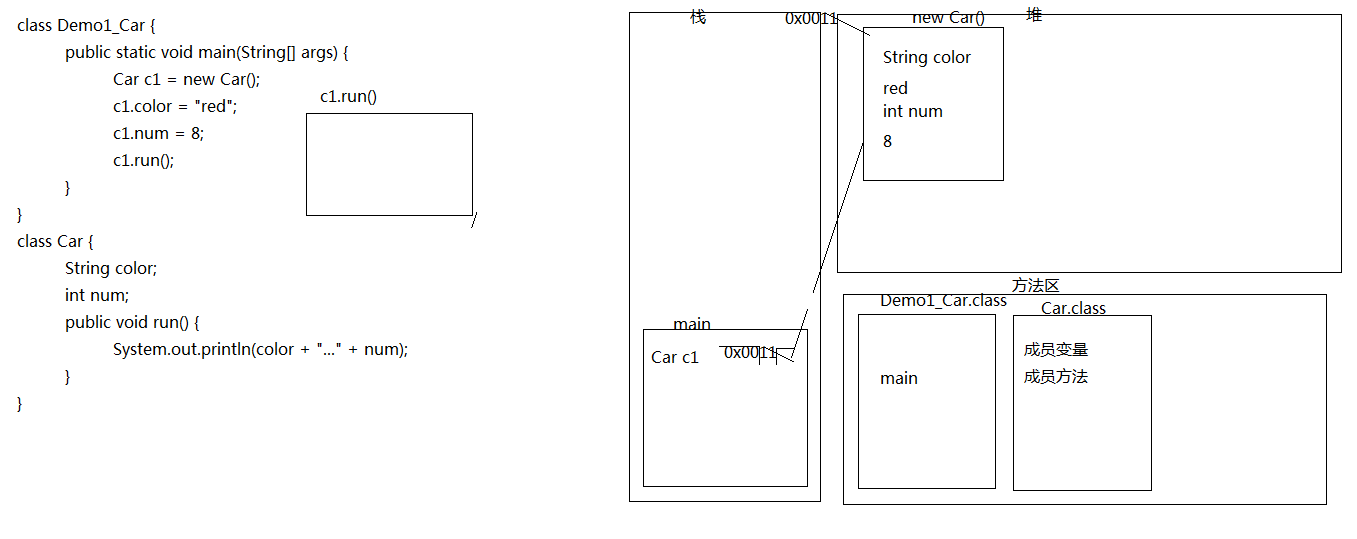
匿名对象内存图



三个引用两个对象内存图



一个对象的内存图



构造方法格式特点

\* a:方法名与类名相同(大小也要与类名一致)

\* b:没有返回值类型，连void都没有

\* c:没有具体的返回值return;

A:案例演示

\* 构造方法的重载

\* 重载:方法名相同,与返回值类型无关(构造方法没有返回值),只看参数列表

\* B:构造方法注意事项

\* a:如果我们没有给出构造方法，系统将自动提供一个无参构造方法。

\* b:如果我们给出了构造方法，系统将不再提供默认的无参构造方法。

\* 注意：这个时候，如果我们还想使用无参构造方法，就必须自己给出。建议永远自己给出无参构造方法

\* A:static关键字的特点

\* a:随着类的加载而加载

\* b:优先于对象存在

\* c:被类的所有对象共享

\* 举例：咱们班级的学生应该共用同一个班级编号。

\* 其实这个特点也是在告诉我们什么时候使用静态?

\* 如果某个成员变量是被所有对象共享的，那么它就应该定义为静态的。

\* 举例：

\* 饮水机(用静态修饰)

\* 水杯(不能用静态修饰)

\* 共性用静态,特性用非静态

\* d:可以通过类名调用

\* 其实它本身也可以通过对象名调用。

\* 推荐使用类名调用。

\* 静态修饰的内容一般我们称其为：与类相关的，类成员

在静态方法中是没有this关键字的

\* 如何理解呢?

\* 静态是随着类的加载而加载，this是随着对象的创建而存在。

\* 静态比对象先存在。

\* b:静态方法只能访问静态的成员变量和静态的成员方法

\* 静态方法：

\* 成员变量：只能访问静态变量

\* 成员方法：只能访问静态成员方法

\* 非静态方法：

\* 成员变量：可以是静态的，也可以是非静态的

\* 成员方法：可是是静态的成员方法，也可以是非静态的成员方法。

\* 简单记：

\* 静态只能访问静态。

###07.12\_面向对象(静态变量和成员变量的区别)(掌握)

\* 静态变量也叫类变量 成员变量也叫对象变量

\* A:所属不同

\* 静态变量属于类，所以也称为为类变量

\* 成员变量属于对象，所以也称为实例变量(对象变量)

\* B:内存中位置不同

\* 静态变量存储于方法区的静态区

\* 成员变量存储于堆内存

\* C:内存出现时间不同

\* 静态变量随着类的加载而加载，随着类的消失而消失

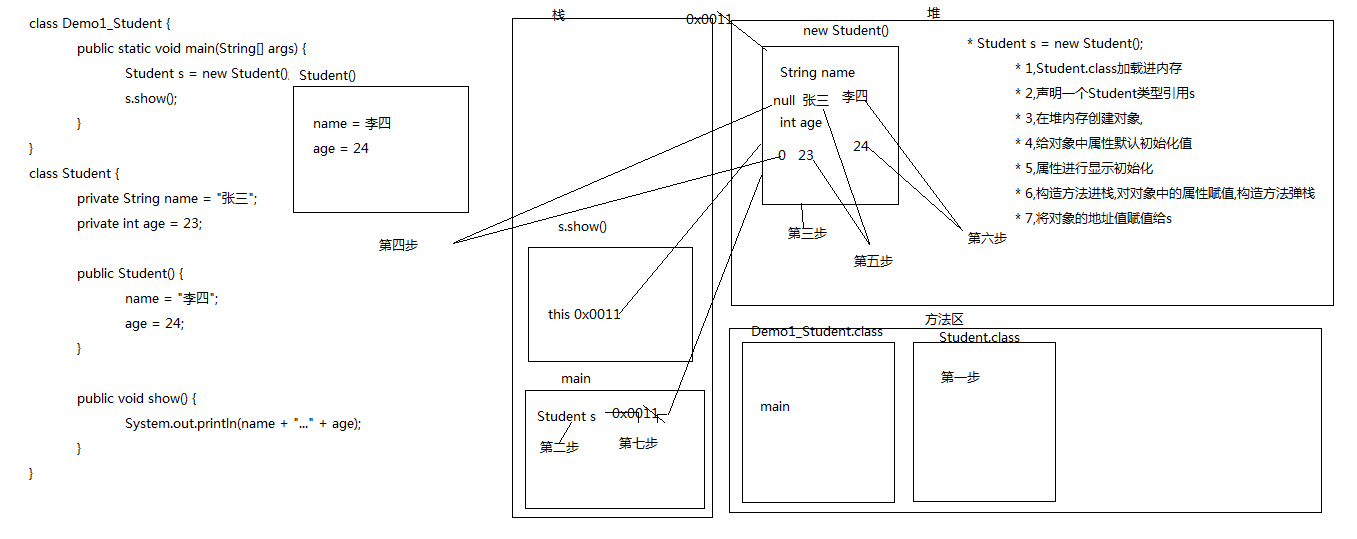
\* 成员变量随着对象的创建而存在，随着对象的消失而消失

\* D:调用不同

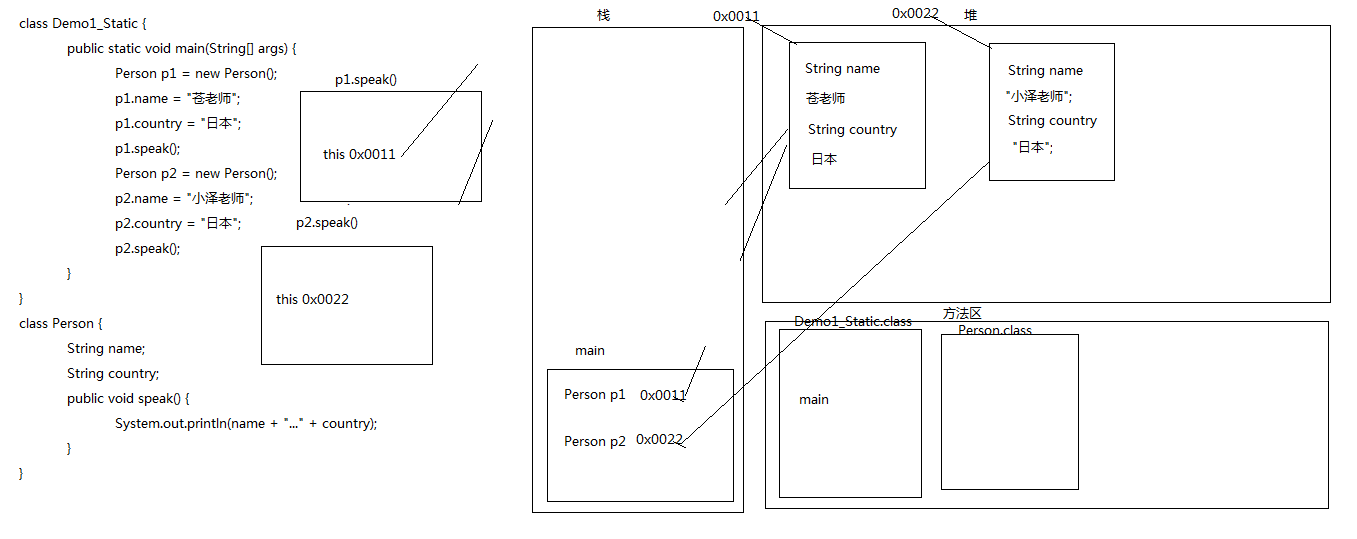
\* 静态变量可以通过类名调用，也可以通过对象调用

\* 成员变量只能通过对 象名调用

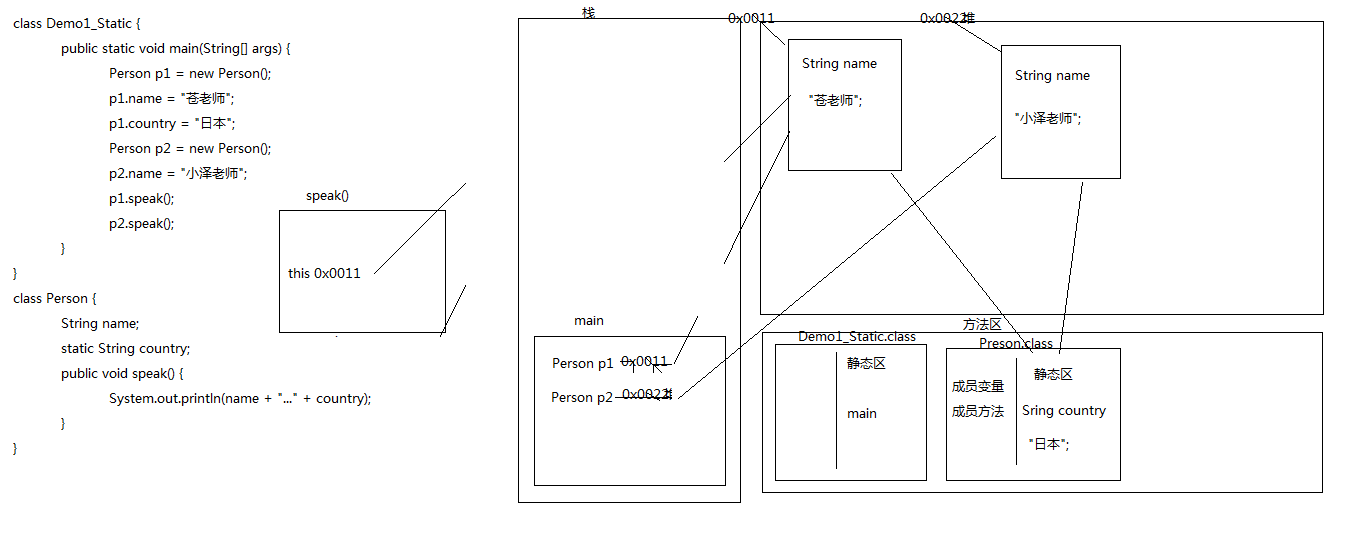
创建对象步骤



创建对象内存图



静态内存图



//如果一个类中所有的方法都是静态的,需要再多做一步,私有构造方法,目的是不让其他类创建本类对象

//直接用类名.调用即可

私有构造方法

**private** ArrayTool(){}

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

//method(); //错误: 无法从静态上下文中引用非静态 方法 method()

Demo3\_Static.print();//在主方法中调用本类的静态方法,可以省略类名.,系统会默认加上

Demo3\_Static d = **new** Demo3\_Static(); //非静态方法在调用的时候必须创建对象调用

d.method();

}

C:常见代码块的应用

\* a:局部代码块

\* 在方法中出现；限定变量生命周期，及早释放，提高内存利用率

\* b:构造代码块 (初始化块)

\* 在类中方法外出现；多个构造方法方法中相同的代码存放到一起，每次调用构造都执行，并且在构造方法前执行

\* c:静态代码块

\* 在类中方法外出现，并加上static修饰；用于给类进行初始化，在加载的时候就执行，并且只执行一次。

\* 一般用于加载驱动

\* A:继承的好处

\* a:提高了代码的复用性

\* b:提高了代码的维护性

\* c:让类与类之间产生了关系，是多态的前提

\* B:继承的弊端

\* 类的耦合性增强了。

\* 开发的原则：高内聚，低耦合。

\* 耦合：类与类的关系

\* 内聚：就是自己完成某件事情的能力

A:Java中类的继承特点

\* a:Java只支持单继承，不支持多继承。(一个儿子只能有一个爹)

\* 有些语言是支持多继承，格式：extends 类1,类2,...

\* b:Java支持多层继承(继承体系)

Java中类的继承特点

\* 如果想用这个体系的所有功能用最底层的类创建对象

\* 如果想看这个体系的共性功能,看最顶层的类

A:继承的注意事项

\* a:子类只能继承父类所有非私有的成员(成员方法和成员变量)

\* b:子类不能继承父类的构造方法，但是可以通过super(马上讲)关键字去访问父类构造方法。

\* c:不要为了部分功能而去继承

08.08\_面向对象(this和super的区别和应用)(掌握)

\* A:this和super都代表什么

\* this:代表当前对象的引用,谁来调用我,我就代表谁

\* super:代表当前对象父类的引用

\* B:this和super的使用区别

\* a:调用成员变量

\* this.成员变量 调用本类的成员变量,也可以调用父类的成员变量

\* super.成员变量 调用父类的成员变量

\* b:调用构造方法

\* this(...) 调用本类的构造方法

\* super(...) 调用父类的构造方法

\* c:调用成员方法

\* this.成员方法 调用本类的成员方法,也可以调用父类的方法

\* super.成员方法 调用父类的成员方法

08.09\_面向对象(继承中构造方法的关系)(掌握)

\* A:案例演示

\* 子类中所有的构造方法默认都会访问父类中空参数的构造方法

\* B:为什么呢?

\* 因为子类会继承父类中的数据，可能还会使用父类的数据。

\* 所以，子类初始化之前，一定要先完成父类数据的初始化。

\* 其实：

\* 每一个构造方法的第一条语句默认都是：super() Object类最顶层的父类。

###08.10\_面向对象(继承中构造方法的注意事项)(掌握)

\* A:案例演示

\* 父类没有无参构造方法,子类怎么办?

\* super解决

\* this解决

\* B:注意事项

\* super(…)或者this(….)必须出现在构造方法的第一条语句上

\* a:父类中私有方法不能被重写

\* 因为父类私有方法子类根本就无法继承

\* b:子类重写父类方法时，访问权限不能更低

\* 最好就一致

\* c:父类静态方法，子类也必须通过静态方法进行重写

\* 其实这个算不上方法重写，但是现象确实如此，至于为什么算不上方法重写，多态中我会讲解(静态只能覆盖静态)

\* 子类重写父类方法的时候，最好声明一模一样。

A:什么是方法重写

\* 重写:子父类出现了一模一样的方法(注意:返回值类型可以是子父类,这个我们学完面向对象讲)

\* B:方法重写的应用：

\* 当子类需要父类的功能，而功能主体子类有自己特有内容时，可以重写父类中的方法。这样，即沿袭了父类的功能，又定义了子类特有的内容。

###08.15\_面向对象(方法重写的面试题)(掌握)

\* A:方法重写的面试题

\* Override和Overload的区别?Overload能改变返回值类型吗?

\* overload可以改变返回值类型,只看参数列表

\* 方法重写：子类中出现了和父类中方法声明一模一样的方法。与返回值类型有关,返回值是一致(或者是子父类)的

\* 方法重载：本类中出现的方法名一样，参数列表不同的方法。与返回值类型无关。

\* 子类对象调用方法的时候：

\* 先找子类本身，再找父类。

###08.19\_面向对象(final关键字修饰类,方法以及变量的特点)(掌握)

\* A:final概述

\* B:final修饰特点

\* 修饰类，类不能被继承

\* 修饰变量，变量就变成了常量，只能被赋值一次

\* 修饰方法，方法不能被重写

\* A:修饰符：

\* 权限修饰符：private，默认的，protected，public

\* 状态修饰符：static，final

\* 抽象修饰符：abstract

\* B:类：

\* 权限修饰符：默认修饰符，public

\* 状态修饰符：final

\* 抽象修饰符：abstract

\* 用的最多的就是：public

\* C:成员变量：

\* 权限修饰符：private，默认的，protected，public

\* 状态修饰符：static，final

\* 用的最多的就是：private

\* D:构造方法：

\* 权限修饰符：private，默认的，protected，public

\* 用的最多的就是：public

\* E:成员方法：

\* 权限修饰符：private，默认的，protected，public

\* 状态修饰符：static，final

\* 抽象修饰符：abstract

\* 用的最多的就是：public

\* F:除此以外的组合规则：

\* 成员变量：public static final

\* 成员方法：

\* public static

\* public abstract

\* public final

\* B:内部类访问特点

\* a:内部类可以直接访问外部类的成员，包括私有。

\* b:外部类要访问内部类的成员，必须创建对象。

\* 外部类名.内部类名 对象名 = 外部类对象.内部类对象;

//外部类名.内部类名 = 外部类对象.内部类对象

Outer.Inner oi = **new** Outer().**new** Inner(); //创建内部类对象

oi.method();

###10.10\_面向对象(静态成员内部类)(了解)

\* static

\* B:成员内部类被静态修饰后的访问方式是:

\* 外部类名.内部类名 对象名 = 外部类名.内部类对象;

//外部类名.内部类名 对象名 = 外部类名.内部类对象;

Outer.Inner oi = **new** Outer.Inner();

oi.method();

Outer.Inner2.print();

**class** Test1\_InnerClass {

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

Outer.Inner oi = **new** Outer().**new** Inner();

oi.show();

}

}

//要求：使用已知的变量，在控制台输出30，20，10。

//内部类之所以能获取到外部类的成员,是因为他能获取到外部类的引用外部类名.this

**class** Outer {

**public** **int** num = 10;

**class** Inner {

**public** **int** num = 20;

**public** **void** show() {

**int** num = 30;

System.out.println(num);

System.out.println(**this**.num);

System.out.println(Outer.**this**.num);

}

}

}

###10.12\_面向对象(局部内部类访问局部变量的问题)(掌握)

\* A:案例演示

\* 局部内部类访问局部变量必须用final修饰

\* 局部内部类在访问他所在方法中的局部变量必须用final修饰,为什么?

因为当调用这个方法时,局部变量如果没有用final修饰,他的生命周期和方法的生命周期是一样的,当方法弹栈,这个局部变量也会消失,那么如果局部内部类对象还没有马上消失想用这个局部变量,就没有了,如果用final修饰会在类加载的时候进入常量池,即使方法弹栈,常量池的常量还在,也可以继续使用

但是jdk1.8取消了这个事情,所以我认为这是个bug

**class** Demo1\_InnerClass {

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

Outer o = **new** Outer();

o.method();

}

}

//局部内部类

**class** Outer {

**public** **void** method() {

**final** **int** num = 10;//局部内部类访问局部变量必须用final修饰

**class** Inner {

**public** **void** print() {

System.out.println(num);//局部内部类访问局部变量必须用final修饰

}

}

Inner i = **new** Inner();

i.print();

}

/\*public void run() {

Inner i = new Inner(); //局部内部类,只能在其所在的方法中访问

i.print();

}\*/

}

###10.13\_面向对象(匿名内部类的格式和理解)

\* A:匿名内部类

\* 就是内部类的简化写法。

\* B:前提：存在一个类或者接口

\* 这里的类可以是具体类也可以是抽象类。

\* C:格式：

\*

new 类名或者接口名(){

重写方法;

}

\* D:本质是什么呢?

\* 是一个继承了该类或者实现了该接口的子类匿名对象。

Inter i = **new** Inter(){

**public** **void** show1() {

System.out.println("show1");

}

**public** **void** show2() {

System.out.println("show2");

}

/\*public void show3() {

System.out.println("show3");

}\*/

};

i.show1();

i.show2();

//i.show3();  **//匿名内部类是不能向下转型的,因为没有子类类名**

**###11.20\_常见对象(==号和equals方法的区别)(掌握)**

**\* ==是一个比较运算符号,既可以比较基本数据类型,也可以比较引用数据类型,基本数据类型比较的是值,引用数据类型比较的是地址值**

**\* equals方法是一个方法,只能比较引用数据类型,所有的对象都会继承Object类中的方法,如果没有重写Object类中的equals方法,equals方法和==号比较引用数据类型无区别,重写后的equals方法比较的是对象中的属性**

###13.01\_常见对象(StringBuffer类的概述)

\* A:StringBuffer类概述

\* 通过JDK提供的API，查看StringBuffer类的说明

\* 线程安全的可变字符序列

\* B:StringBuffer和String的区别

\* String是一个不可变的字符序列

\* StringBuffer是一个可变的字符序列

###13.02\_常见对象(StringBuffer类的构造方法)

\* A:StringBuffer的构造方法：

\* public StringBuffer():无参构造方法

\* public StringBuffer(int capacity):指定容量的字符串缓冲区对象

\* public StringBuffer(String str):指定字符串内容的字符串缓冲区对象

\* B:StringBuffer的方法：

\* public int capacity()：返回当前容量。 理论值(不掌握)

\* public int length():返回长度（字符数）。 实际值

\* C:案例演示

\* 构造方法和长度方法的使用

###13.03\_常见对象(StringBuffer的添加功能)

\* A:StringBuffer的添加功能

\* public StringBuffer append(String str):

\* 可以把任意类型数据添加到字符串缓冲区里面,并返回字符串缓冲区本身

\* public StringBuffer insert(int offset,String str):

\* 在指定位置把任意类型的数据插入到字符串缓冲区里面,并返回字符串缓冲区本身

###13.04\_常见对象(StringBuffer的删除功能)

\* A:StringBuffer的删除功能

\* public StringBuffer deleteCharAt(int index):

\* 删除指定位置的字符，并返回本身

\* public StringBuffer delete(int start,int end):

\* 删除从指定位置开始指定位置结束的内容，并返回本身

StringBuffer sb = **new** StringBuffer();

//sb.deleteCharAt(5); //当缓冲区中这个索引上没有元素的时候就会报StringIndexOutOfBoundsException

sb.append("heima");

//sb.deleteCharAt(4); //根据索引删除掉索引位置上对应的字符

//sb.delete(0, 2); //删除的时候是包含头,不包含尾

//System.out.println(sb);

//sb.delete(0, sb.length()); //清空缓冲区

//System.out.println(sb);

sb = **new** StringBuffer();//不要用这种方式清空缓冲区,原来的会变成垃圾,浪费内存

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

###13.05\_常见对象(StringBuffer的替换和反转功能)

\* A:StringBuffer的替换功能

\* public StringBuffer replace(int start,int end,String str):

\* 从start开始到end用str替换

\* B:StringBuffer的反转功能

\* public StringBuffer reverse():

\* 字符串反转

###13.06\_常见对象(StringBuffer的截取功能及注意事项)

\* A:StringBuffer的截取功能

\* public String substring(int start):

\* 从指定位置截取到末尾

\* public String substring(int start,int end):

\* 截取从指定位置开始到结束位置，包括开始位置，不包括结束位置

\* B:注意事项

\* **注意:返回值类型不再是StringBuffer本身**

###13.07\_常见对象(StringBuffer和String的相互转换)

\* A:String -- StringBuffer

\* a:通过构造方法

\* b:通过append()方法

\* B:StringBuffer -- String

\* a:通过构造方法

\* b:通过toString()方法

\* c:通过subString(0,length);

###13.10\_常见对象(StringBuffer和StringBuilder的区别)

\* A:StringBuilder的概述

\* 通过查看API了解一下StringBuilder类

\* B:面试题

\* String,StringBuffer,StringBuilder的区别

\* StringBuffer和StringBuilder的区别

\* StringBuffer是jdk1.0版本的,是线程安全的,效率低

\* StringBuilder是jdk1.5版本的,是线程不安全的,效率高

\* String和StringBuffer,StringBuilder的区别

\* String是一个不可变的字符序列

\* StringBuffer,StringBuilder是可变的字符序列

###13.11\_常见对象(String和StringBuffer分别作为参数传递)

\* A:形式参数问题

\* String作为参数传递

\* StringBuffer作为参数传递

\* B:案例演示

\* String和StringBuffer分别作为参数传递问题

基本数据类型的值传递,不改变其值

引用数据类型的值传递,改变其值

String类虽然是引用数据类型,但是他当作参数传递时和基本数据类型是一样的

\* A:Arrays类概述

\* 针对数组进行操作的工具类。

\* 提供了排序，查找等功能。

\* B:成员方法

\* public static String toString(int[] a)

\* public static void sort(int[] a)

\* public static int binarySearch(int[] a,int key)