[Java基础 3](#_Toc18623014)

[1.基础 3](#_Toc18623015)

[1.1 面向对象和面向过程的区别 3](#_Toc18623016)

[面向过程 3](#_Toc18623017)

[面向对象 3](#_Toc18623018)

[1.2 Java 面向对象编程三大特性:封装、继承、多态 3](#_Toc18623019)

[封装 3](#_Toc18623020)

[继承 3](#_Toc18623021)

[多态 3](#_Toc18623022)

[1.3 接口和抽象类有什么区别 4](#_Toc18623023)

[1.4 字符型常量和字符串常量的区别 4](#_Toc18623024)

[1.5 重载和重写的区别 5](#_Toc18623025)

[1.6 String类 5](#_Toc18623026)

[1.6.1两种创建方式： 5](#_Toc18623027)

[1.6.2 字符串的比较： 6](#_Toc18623028)

[1.6.3 字符串的不可变： 6](#_Toc18623029)

[1.7 String,StringBuffer与StringBuilder的区别 7](#_Toc18623030)

[1.7.1 可变与不可变 7](#_Toc18623031)

[1.7.2 运行速度 7](#_Toc18623032)

[1.7.3 线程安全 7](#_Toc18623033)

[1.7.4 StringBuilder为什么线程不安全？ 7](#_Toc18623034)

[1.7.5 为什么返回ArrayIndexOutOfBoundsException异常 8](#_Toc18623035)

[1.7.6 StringBuffer如何保证线程安全 9](#_Toc18623036)

[1.7.7 总结 9](#_Toc18623037)

[1.8 == 与 equals(重要) 9](#_Toc18623038)

[1.8.1 ==： 9](#_Toc18623039)

[1.8.2 equals： 10](#_Toc18623040)

[1.8.3 equals和==的区别： 10](#_Toc18623041)

[1.8.4 对equals重新需要注意五点： 11](#_Toc18623042)

[1.9 hashCode与equals（重要） 11](#_Toc18623043)

[1.9.1 hashCode（）介绍 11](#_Toc18623044)

[1.9.2 为什么要有hashCode 11](#_Toc18623045)

[1.9.3 hashCode（）与equals（）的相关规定 12](#_Toc18623046)

[1.9.4为什么两个对象有相同的hashcode值，它们也不一定是相等的？ 12](#_Toc18623047)

[1.10为什么说Java中只有值传递 12](#_Toc18623048)

[1.10.1 实参与形参 12](#_Toc18623049)

[1.10.2 值传递与引用传递 13](#_Toc18623050)

[1.10.3 总结： 16](#_Toc18623051)

[1.11 final关键字 17](#_Toc18623052)

[1.11.1 关键字final 17](#_Toc18623053)

[1.11.2 类的final变量和普通变量有什么区别？ 17](#_Toc18623054)

[1.11.3 final参数 18](#_Toc18623055)

[1.12 static关键字 20](#_Toc18623056)

[1.13 访问控制 20](#_Toc18623057)

[1.13.1修饰类 20](#_Toc18623058)

[1.13.2修饰类的方法和变量 20](#_Toc18623059)

[2.集合 22](#_Toc18623060)

[2.1集合概述 22](#_Toc18623061)

[2.2 Collection架构 23](#_Toc18623062)

# Java基础

## 1.基础

### 1.1 面向对象和面向过程的区别

#### 面向过程

优点： 性能比面向对象高，因为类调用时需要实例化，开销比较大，比较消耗资源;比如单片机、嵌入式开发、Linux/Unix等一般采用面向过程开发，性能是最重要的因素。

*缺点*： 没有面向对象易维护、易复用、易扩展

#### 面向对象

优点： 易维护、易复用、易扩展，由于面向对象有封装、继承、多态性的特性，可以设计出低耦合的系统，使系统更加灵活、更加易于维护

*缺点*： 性能比面向过程低

### 1.2 Java 面向对象编程三大特性:封装、继承、多态

#### 封装

封装把一个对象的属性私有化，同时提供一些可以被外界访问的属性的方法，如果属性不想被外界访问，我们大可不必提供方法给外界访问。但是如果一个类没有提供给外界访问的方法，那么这个类也没有什么意义了。

#### 继承

继承是使用已存在的类的定义作为基础建立新类的技术，新类的定义可以增加新的数据或新的功能，也可以用父类的功能，但不能选择性地继承父类。通过使用继承我们能够非常方便地复用以前的代码。

关于继承如下 3 点请记住：

1.子类拥有父类非 private 的属性和方法。

2.子类可以拥有自己属性和方法，即子类可以对父类进行扩展。

3.子类可以用自己的方式实现父类的方法。（重写）。

#### 多态

所谓多态就是指程序中定义的引用变量所指向的具体类型和通过该引用变量发出的方法调用在编程时并不确定，而是在程序运行期间才确定，即一个引用变量倒底会指向哪个类的实例对象，该引用变量发出的方法调用到底是哪个类中实现的方法，必须在由程序运行期间才能决定。

在Java中有两种形式可以实现多态：继承（多个子类对同一方法的重写）和接口（实现接口并覆盖接口中同一方法）

### 1.3 接口和抽象类有什么区别

你选择使用接口和抽象类的依据是什么？

接口和抽象类的概念不一样。接口是对动作的抽象，抽象类是对根源的抽象。

抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。比如，男人，女人，这两个类（如果是类的话……），他们的抽象类是人。说明，他们都是人。

人可以吃东西，狗也可以吃东西，你可以把“吃东西”定义成一个接口，然后让这些类去实现它.

所以，在高级语言上，一个类只能继承一个类（抽象类）正如人不可能同时是生物和非生物)，但是可以实现多个接口(吃饭接口、走路接口)。

总结几句话来说：

1、抽象类和接口都不能直接实例化，如果要实例化，抽象类变量必须指向实现所有抽象方法的子类对象，接口变量必须指向实现所有接口方法的类对象。

2、抽象类要被子类继承，接口要被类实现。

3、接口只能做方法申明，抽象类中可以做方法申明，也可以做方法实现

4、接口里定义的变量只能是公共的静态的常量，抽象类中的变量是普通变量。

5、抽象类里的抽象方法必须全部被子类所实现，如果子类不能全部实现父类抽象方法，那么该子类只能是抽象类。同样，一个实现接口的时候，如不能全部实现接口方法，那么该类也只能为抽象类。

6、抽象方法只能申明，不能实现，接口是设计的结果 ，抽象类是重构的结果

7、抽象类里可以没有抽象方法

8、如果一个类里有抽象方法，那么这个类只能是抽象类

9、抽象方法要被实现，所以不能是静态的，也不能是私有的。

10、接口可继承接口，并可多继承接口，但类只能单根继承。

总结：

1、抽象类和接口都是用来抽象具体对象的. 但是接口的抽象级别最高

2、抽象类可以有具体的方法 和属性, 接口只能有抽象方法和不可变常量

3、抽象类主要用来抽象类别,接口主要用来抽象功能.

4、抽象类中，且不包含任何实现，派生类必须覆盖它们。接口中所有方法都必须是未实现的。

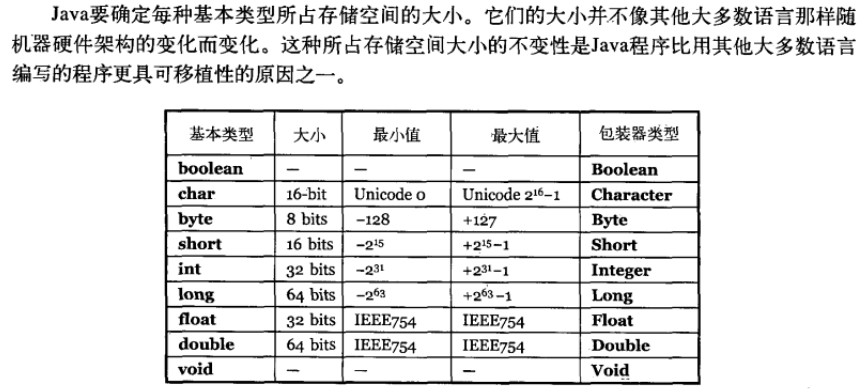
当你关注一个事物的本质的时候，用抽象类；当你关注一个操作的时候，用接口。 抽象类的功能要远超过接口，但是，定义抽象类的代价高。因为高级语言来说（从实际设计上来说也是）每个类只能继承一个类。在这个类中，你必须继承或编写出其所有子类的所有共性。虽然接口在功能上会弱化许多，但是它只是针对一个动作的描述。而且你可以在一个类中同时实现多个接口。在设计阶段会降低难度的。

### 1.4 字符型常量和字符串常量的区别

1.形式上: 字符常量是单引号引起的一个字符 字符串常量是双引号引起的若干个字符

2.含义上: 字符常量相当于一个整形值( ASCII 值),可以参加表达式运算，字符串常量代表一个地址值(该字符串在内存中存放位置)

3.占内存大小：字符常量只占2个字节，字符串常量占若干个字节(至少一个字符结束标志) (注意： char在Java中占两个字节)



### 1.5 重载和重写的区别

**重载**： 发生在同一个类中，方法名必须相同，参数类型不同、个数不同、顺序不同，方法返回值和访问修饰符可以不同，发生在编译时。

**重写**： 发生在父子类中，方法名、参数列表必须相同，返回值范围小于等于父类，抛出的异常范围小于等于父类，访问修饰符范围大于等于父类；如果父类方法访问修饰符为 private 则子类就不能重写该方法。

**注意**：父类的私有属性和构造方法并不能被继承，所以Constructor构造方法不能被override(重写)，但是可以overload(重载)，所以一个类里可以有多个构造函数，无参构造，有参数构造等等。

### 1.6 String类

String类代表字符串，所有的字符串都是这个类的实例实现，字符串是常量，在创建之后值不可变。

String 类中使用 final 关键字字符数组保存字符串，private　final　char　value[]，所以 String 对象是不可变的。

#### 1.6.1两种创建方式：

1.String str="hello";//直接赋值的方式

2.String str=new String("hello");//new关键字实例化的方式

在字符串中，如果采用***直接赋值***的方式进行对象的实例化，直接赋值会先检查常量池内有没有对应的字符串，有的话就直接指向，则会将“hello”放入常量，每当下一次对不同的对象进行直接赋值的时候会直接利用池中原有的匿名对象。

采用**构造实例化**的方式的时候，会像堆申请内存，来存储字符串，str在栈内存中指向堆内存中的匿名对象“hello”，当我们再一次的new一个同样内容的String对象时，原来的匿名对象将被，会产生新的匿名对象“hello”，原来的将被抛弃变成垃圾回收，字符串对象存在于堆空间中，与字符串常量池无关。

String str=new String("hello");

**栈内存**

**str**

**堆内存**

**hello**

**hello**

**生成了新的匿名对象，旧的抛弃**

String str=new String("hello");

**栈内存**

**str**

**堆内存**

**hello**

**生成了匿名对象**

**两种实例化的总结：**

1）直接赋值（String str = "hello"）：只开辟一块堆内存空间，并且会自动入池，不会产生垃圾。

2）构造方法（String str=  new String("hello");）:会开辟两块堆内存空间，其中一块堆内存会变成垃圾被系统回收，而且不能够自动入池，需要通过public  String intern();方法进行手工入池。在开发的过程中不会采用构造方法进行字符串的实例化。

例题：

String s1="hello"+"world"；

String s2="helloworld"；

其中s1==s2返回的是true。字符串相加的时候，如果相加的是静态字符串，则会去常量池中寻找对应的存在直接指向。

#### 1.6.2 字符串的比较：

**== 和public boolean equals()比较字符串的区别：**

==在对字符串比较的时候，对比的是**内存地址**，而equals比较的是**字符串内容**在开发的过程中，equals()通过接受参数，可以避免空指向。

*举例*：

String str **=** **null;**

**if(**str**.**equals**(**"hello"**)){**//此时会出现空指向异常

**...**

**}**

**if(**"hello"**.**equals**(**str**)){**//此时equals会处理null值，可以避免空指向异常

**...**

**}**

#### 1.6.3 字符串的不可变：

字符串从创建开始就不可以去变了，变化的的最多只是地址，内容是不会变的。

在设计的时候，被设计成**享元模式**，每当生成一个新的字符串的是，就会被放进共享池中，第二次生成同样的内容字符串的时候，则会共享这个对象，这些操作只限于**直接赋值==**。

在object中，equals()是用来比较内存地址的，但是String**重写了equals()方法，用来比较内容的，即使是不同地址，只要内容一致，也会返回true。**

不可变的好处：

1. 避免多次创建的开销
2. 出于安全性考虑，字符串在程序用应用太多了
3. 例如HashMap中可以用String做key，可变的话就会有问题
4. 在传参的时候，使用不可变类不需要去考虑谁可能会修改其内部的值，如果使用可变类的话，可能需要每次记得重新拷贝出里面的值，性能会有一定的损失

举例：

使用String s = new String("hello");会创建几个对象

　　会创建2个对象

　　首先，出现了字面量"hello"，那么去String Pool中查找是否有相同字符串存在，因为程序就这一行代码所以肯定没有，那么就在Java Heap中用字面量"hello"首先创建1个String对象。接着，new String("hello")，关键字new又在Java Heap中创建了1个对象，然后调用接收String参数的构造器进行了初始化。最终s的引用是这个String对象.

### 1.7 String,StringBuffer与StringBuilder的区别

#### 1.7.1 可变与不可变

String类中用字符数组保存字符，采用final修饰，所以**String对象不可变**。

StringBuffer与StringBuilder继承于AbstractStringBuilder，AbstractStringBuilder里面也是采用字符数组，但是没有final修饰，所以是**可变的**。

#### 1.7.2 运行速度

由于String类是不可变的，每次重新赋值也就是创建新的对象，原来的被回收，所以比较慢。

而StringBuilder和StringBuffer的对象是变量，对变量进行操作就是直接对该对象进行更改，而不进行创建和回收的操作，所以速度要比String快很多。

#### 1.7.3 线程安全

String是不可变的，显然是线程安全的。

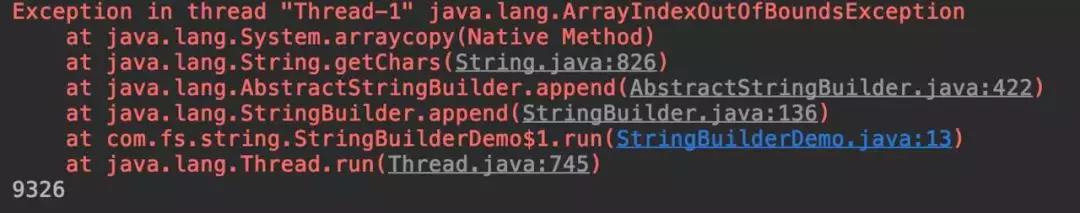
StringBuilder和StringBuffer，StringBuffer内部对字符串的操作都采用了synchronized关键字，所以是线程安全的；StringBuilder是非线程安全的。

由于，StringBuilder非线程安全，所以加锁释放锁的操作就没有，就相对于StringBuffer更快。

#### 1.7.4 StringBuilder为什么线程不安全？

1. **public** **class** StringBuilderDemo {
2. **public** **static** **void** main(String[] args) **throws** InterruptedException {
3. StringBuilder stringBuilder = **new** StringBuilder();
4. **for** (**int** i = 0; i < 10; i++){
5. **new** Thread(**new** Runnable() {
6. @Override
7. **public** **void** run() {
8. **for** (**int** j = 0; j < 1000; j++){
9. stringBuilder.append("a");
10. }
11. }
12. }).start();
13. }
14. Thread.sleep(100);
15. System.out.println(stringBuilder.length());
16. }
17. }

这段代码生成了10个线程，每个线程向StringBuilder里插入1000次a，到最后应该有10000个长度，但是实际结果输出是：



结果小于预期的10000，并且还抛出了一个ArrayIndexOutOfBoundsException异常（异常不是必现，需要多试试）

这时候需要看看append方法，StringBuilder的append()方法调用的父类AbstractStringBuilder的append()方法：

1. **public** AbstractStringBuilder append(String str) {
2. **if** (str == **null**)
3. **return** appendNull();
4. **int** len = str.length();
5. ensureCapacityInternal(count + len);  //判断扩容的操作
6. str.getChars(0, len, value, count);  //将string对象的char拷贝到builder里
7. count += len;
8. **return** **this**;
9. }

直接看第七行，假设这个时候count值为10，len值为1，两个线程同时执行到了第七行，拿到的count值都是10，执行完加法运算后将结果赋值给count，所以两个线程执行完后count值为11，而不是12。我们调用length方法的时候，其实就是返回count这个值，count并没有达到预期的结果，这就是最终结果小于10000的原因

#### 1.7.5 为什么返回ArrayIndexOutOfBoundsException异常

我们看回AbstractStringBuilder的append()方法源码的第五行，ensureCapacityInternal()方法是检查StringBuilder对象的原char数组的容量能不能盛下新的字符串，如果盛不下就调用expandCapacity()方法对char数组进行扩容。

1. **private** **void** ensureCapacityInternal(**int** minimumCapacity) {
2. // 传入的值是原本长度加上新增的字符长度的和，减去字符存储的长度，大于0，需要扩容
3. **if** (minimumCapacity - value.length > 0)
4. expandCapacity(minimumCapacity);
5. }

扩容的逻辑就是new一个新的char数组，新的char数组的容量是原来char数组的两倍再加2，再通过System.arryCopy()函数将原数组的内容复制到新数组，最后将指针指向新的char数组。

1. **void** expandCapacity(**int** minimumCapacity) {
2. //计算新的容量
3. **int** newCapacity = value.length \* 2 + 2;
4. //中间省略了一些检查逻辑
5. ...
6. value = Arrays.copyOf(value, newCapacity);
7. }

两个线程同时在插入数据的时候，，两个线程都执行完了第五行ensureCapacityInternal()方法，此时长度为5，线程1获得时间片，得到长度5，准备插入，线程2又竞争到时间片，插入，长度变为6，线程1继续执行第六行的str.getChars()方法的时候拿到的count值就是6了，执行char数组拷贝的时候就会抛出ArrayIndexOutOfBoundsException异常

#### 1.7.6 StringBuffer如何保证线程安全

StringBuffer内部对字符串的操作都采用了synchronized关键字，所以是线程安全的。

可以点进去StringBuffer的内部查看，关键方法，都在方法前加上了synchronized，例如append方法等

#### 1.7.7 总结

**String：**适用于少量的字符串操作的情况

**StringBuilder：**适用于单线程下在字符缓冲区进行大量操作的情况

**StringBuffer：**适用多线程下在字符缓冲区进行大量操作的情况

### 1.8 == 与 equals(重要)

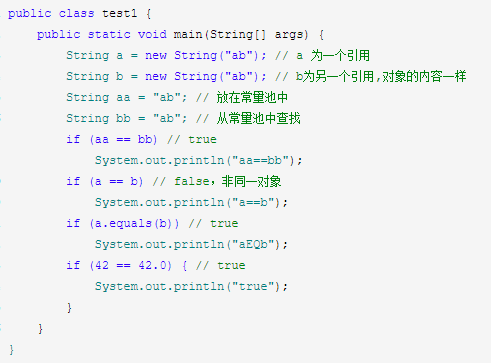
#### 1.8.1 ==：

== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指相同一个对象。比较的是真正意义上的指针操作。

1. 比较的是操作符两端的操作数是否是同一个对象。  
   2、两边的操作数必须是同一类型的（可以是父子类之间）才能编译通过。  
   3、比较的是地址，如果是具体的阿拉伯数字的比较，值相等则为true，如：  
   int a=10 与 long b=10L 与 double c=10.0都是相同的（为true），因为他们都指向地址为10的堆。

#### 1.8.2 equals：

　　equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承自java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，**而Object中的equals方法返回的却是==的判断**。



#### 1.8.3 equals和==的区别：

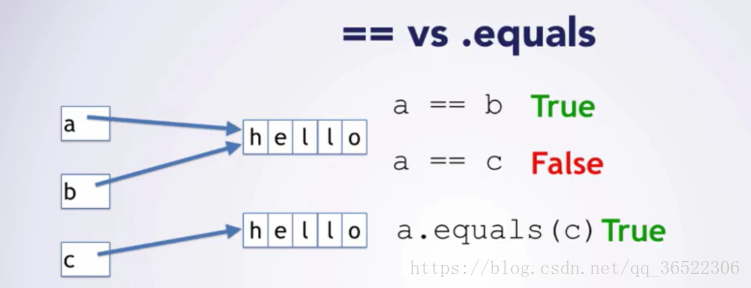
equals方法最初是在所有类的基类Object中进行定义的，源码是

public boolean equals(Object obj) {

    return (this == obj);

}

由equals的源码可以看出这里定义的equals与==是等效的（Object类中的equals没什么区别），不同的原因就在于有些类（像String、Integer等类）对equals进行了**重写**，但是没有对equals进行重写的类（比如我们自己写的类）就只能从Object类中继承equals方法，其equals方法与==就也是等效的，除非我们在此类中重写equals。



== 比较的是变量(栈)内存中存放的对象的(堆)内存地址，用来判断两个对象的地址是否相同，即是否是指相同一个对象。比较的是真正意义上的指针操作。

equals用来比较的是两个对象的内容是否相等，由于所有的类都是继承java.lang.Object类的，所以适用于所有对象，如果没有对该方法进行覆盖的话，调用的仍然是Object类中的方法，而Object中的equals方法返回的却是==的判断。

String s="abcd"是一种非常特殊的形式,和new 有本质的区别。它是java中唯一不需要new 就可以产生对象的途径。

以String s="abcd";形式赋值在java中叫直接量,它是在常量池中而不是象new一样放在压缩堆中。

这种形式的字符串，在JVM内部发生字符串拘留，即当声明这样的一个字符串后，JVM会在常量池中先查找有有没有一个值为"abcd"的对象,如果有,就会把它赋给当前引用.即原来那个引用和现在这个引用指点向了同一对象, 如果没有,则在常量池中新创建一个"abcd",下一次如果有String s1 = "abcd";又会将s1指向"abcd"这个对象,即以这形式声明的字符串,只要值相等,任何多个引用都指向同一对象.

而String s = new String("abcd");和其它任何对象一样.每调用一次就产生一个对象，只要它们调用。

也可以这么理解: String str = "hello"; 先在内存中找是不是有"hello"这个对象,如果有，就让str指向那个"hello".如果内存里没有"hello"，就创建一个新的对象保存"hello". String str=new String ("hello") 就是不管内存里是不是已经有"hello"这个对象，都新建一个对象保存"hello"。

"=="比"equals"运行速度快,因为"=="只是比较引用。

#### 1.8.4 对equals重新需要注意五点：

自反性：对任意引用值X，x.equals(x)的返回值一定为true；

对称性：对于任何引用值x,y,当且仅当y.equals(x)返回值为true时，x.equals(y)的返回值一定为true；

传递性：如果x.equals(y)=true, y.equals(z)=true,则x.equals(z)=true ；

非空性：任何非空的引用值X，x.equals(null)的返回值一定为false 。

一致性：如果参与比较的对象没任何改变，则对象比较的结果也不应该有任何改变；

### 1.9 hashCode与equals（重要）

#### 1.9.1 hashCode（）介绍

hashCode() 的作用是获取哈希码，也称为散列码；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在哈希表中的索引位置。hashCode() 定义在JDK的Object.java中，这就意味着Java中的任何类都包含有hashCode() 函数。另外需要注意的是： Object 的 hashcode 方法是本地方法，也就是用 c 语言或 c++ 实现的，该方法通常用来将对象的 内存地址 转换为整数之后返回。散列表存储的是键值对(key-value)，它的特点是：能根据“键”快速的检索出对应的“值”。这其中就利用到了散列码！（可以快速找到所需要的对象）。

#### 1.9.2 为什么要有hashCode

我们以“HashSet如何检查重复”为例子来说明为什么要有hashCode：

当你把对象加入HashSet时，HashSet会先计算对象的hashcode值来判断对象加入的位置，同时也会与其他已经加入的对象的hashcode值作比较，如果没有相符的hashcode，HashSet会假设对象没有重复出现。但是如果发现有相同hashcode值的对象，这时会调用equals（）方法来检查hashcode相等的对象是否真的相同。如果两者相同，HashSet就不会让其加入操作成功。如果不同的话，就会重新散列到其他位置。这样我们就大大减少了equals的次数，相应就大大提高了执行速度。

#### 1.9.3 hashCode（）与equals（）的相关规定

1. 如果两个对象相等，则hashcode一定也是相同的
2. 两个对象相等,对两个对象分别调用equals方法都返回true
3. 两个对象有相同的hashcode值，它们也不一定是相等的
4. 因此，equals方法被覆盖过，则hashCode方法也必须被覆盖
5. hashCode()的默认行为是对堆上的对象产生独特值。如果没有重写hashCode()，则该class的两个对象无论如何都不会相等（即使这两个对象指向相同的数据）

#### 1.9.4为什么两个对象有相同的hashcode值，它们也不一定是相等的？

因为hashCode() 所使用的杂凑算法也许刚好会让多个对象传回相同的杂凑值。越糟糕的杂凑算法越容易碰撞，但这也与数据值域分布的特性有关（所谓碰撞也就是指的是不同的对象得到相同的 hashCode）。我们刚刚也提到了 HashSet,如果 HashSet 在对比的时候，同样的 hashcode 有多个对象，它会使用 equals() 来判断是否真的相同。也就是说 hashcode 只是用来缩小查找成本。

### 1.10为什么说Java中只有值传递

首先纠正一下一些**错误理解**：

错误理解一：值传递和引用传递，区分的条件是传递的内容，如果是个值，就是值传递。如果是个引用，就是引用传递。

错误理解二：Java是引用传递。

错误理解三：传递的参数如果是普通类型，那就是值传递，如果是对象，那就是引用传递。

#### 1.10.1 实参与形参

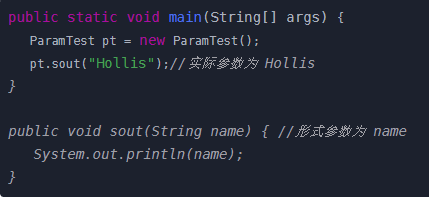
我们都知道，在Java中定义方法的时候是可以定义参数的。比如Java中的main方法：



这里面的args就是参数。参数在程序语言中分为形式参数和实际参数。

**形式参数**：是在定义函数名和函数体的时候使用的参数,目的是用来接收调用该函数时传入的参数。

**实际参数**：在调用有参函数时，主调函数和被调函数之间有数据传递关系。在主调函数中调用一个函数时，函数名后面括号中的参数称为“实际参数”。



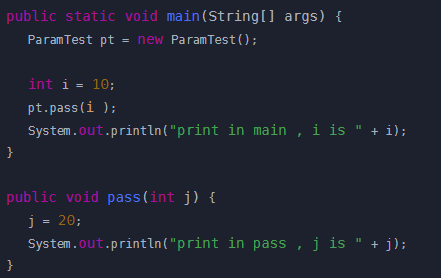
实际参数是调用有参方法的时候真正传递的内容，而形式参数是用于接收实参内容的参数。

#### 1.10.2 值传递与引用传递

上面提到了，当我们调用一个有参函数的时候，会把实际参数传递给形式参数。但是，在程序语言中，这个传递过程中传递的两种情况，即值传递和引用传递。我们来看下程序语言中是如何定义和区分值传递和引用传递的。

值传递（pass by value）是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

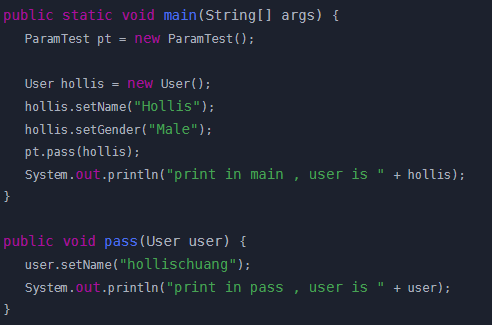
引用传递（pass by reference）是指在调用函数时将实际参数的地址直接传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。



我们在pass方法中修改了参数j的值，然后分别在pass方法和main方法中打印参数的值.

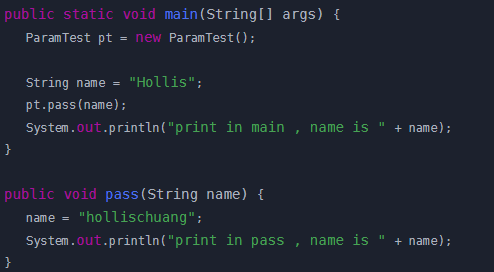


pass方法内部对name的值的修改并没有改变实际参数i的值。那么，按照上面的定义，有人得到结论：Java的方法传递是值传递。但是不能轻易下结论，我们操作的只是基础类型。





经过pass方法执行后，实参的值竟然被改变了，那按照上面的引用传递的定义，实际参数的值被改变了，这不就是引用传递了么。于是，根据上面的两段代码，有人得出一个新的结论：Java的方法中，在传递普通类型的时候是值传递，在传递对象类型的时候是引用传递。这个还是错误的结论，再看看下面的代码。



输出结果：

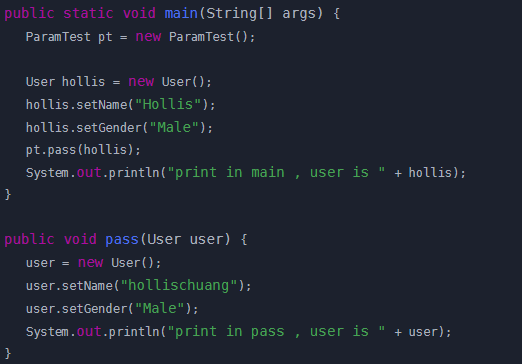


同样传递了一个对象，但是原始参数的值并没有被修改。

值传递（pass by value）是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

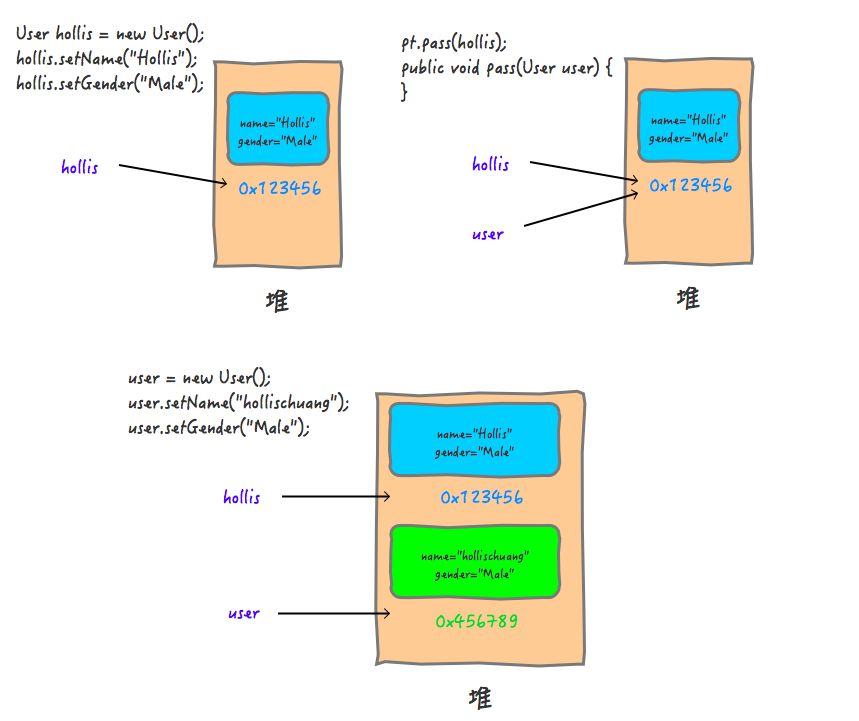
引用传递（pass by reference）是指在调用函数时将实际参数的地址直接传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。







其实关于hollis的内容并没有变这是为什么呢，因为pt.pass(hollis),这时候将user还有hollis指向同一个地址，当调用user=new User()时候，会另外开辟一个内存空间的，这时候修改hollis还有user是没有有相互关联的。



通过上图我们能看出这不是个引用传递，如果是引用传递的话，user = new User()的时候，实际参数的引用也应该改为指向0X456789，但是实际上并没有。

这里是把实际参数的引用的地址**复制**了一份，传递给了形式参数。所以，**上面的参数其实是值传递，把实参对象引用的地址当做值传递给了形式参数。**

如果没有user = new User()，直接set的话，并没有对形参本身进行修改，而是修改的形参持有的地址中存储的内容。**所以，值传递和引用传递的区别并不是传递的内容。而是实参到底有没有被复制一份给形参。**在判断实参内容有没有受影响的时候，要看传的的是什么，如果你传递的是个地址，那么就看这个地址的变化会不会有影响，而不是看地址指向的对象的变化。就像钥匙和房子的关系。（钥匙是地址，房子才是地址指向的引用）**所以说，Java中其实还是值传递的，只不过对于对象参数，值的内容是对象的引用。**

#### 1.10.3 总结：

无论是值传递还是引用传递，其实都是一种求值策略(Evaluation strategy)。在求值策略中，还有一种叫做按共享传递(call by sharing)。其实Java中的参数传递严格意义上说应该是按共享传递。

按共享传递，是指在调用函数时，传递给函数的是实参的地址的拷贝（如果实参在栈中，则直接拷贝该值）。在函数内部对参数进行操作时，需要先拷贝的地址寻找到具体的值，再进行操作。如果该值在栈中，那么因为是直接拷贝的值，所以函数内部对参数进行操作不会对外部变量产生影响。如果原来拷贝的是原值在堆中的地址，那么需要先根据该地址找到堆中对应的位置，再进行操作。因为传递的是地址的拷贝所以函数内对值的操作对外部变量是可见的。

简单点说，Java中的传递，是值传递，而这个值，实际上是对象的引用。

而按共享传递其实只是按值传递的一个特例罢了。所以我们可以说Java的传递是按共享传递，或者说Java中的传递是值传递。

### 1.11 final关键字

#### 1.11.1 关键字final

在Java中，final关键字可以用来**修饰类、方法和变量**（包括成员变量和局部变量），变量也可以叫数据。

1、修饰类

当用final修饰一个类时，表明这个类不能被继承。也就是说，如果一个类你永远不会让他被继承，就可以用final进行修饰。final类中的成员变量可以根据需要设为final，但是要注意final类中的所有成员方法都会被隐式地指定为final方法。

**在使用final修饰类的时候，要注意谨慎选择，除非这个类真的在以后不会用来继承或者出于安全的考虑，尽量不要将类设计为final类。**

1. 修饰方法

下面这段话摘自《Java编程思想》第四版第143页：

　 “使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。在早期的Java实现版本中，会将final方法转为内嵌调用。但是如果方法过于庞大，可能看不到内嵌调用带来的任何性能提升。在最近的Java版本中，不需要使用final方法进行这些优化了。“

如果只有在**想明确禁止 该方法在子类中被覆盖的情况下才将方法设置为final的**。即**父类的final方法是不能被子类所覆盖的**，也就是说子类是不能够存在和父类一模一样的方法的。

此处需要注意的一点是：**因为重写的前提是子类可以从父类中继承此方法，如果父类中final修饰的方法同时访问控制权限为private，将会导致子类中不能直接继承到此方法**，因此，此时可以在子类中定义相同的方法名和参数，此时不再产生重写与final的矛盾，而是在子类中重新定义了新的方法。（注：**类的private方法会隐式地被指定为final方法**。）

1. 修饰变量（常用）

final成员变量表示常量，**只能被赋值一次，赋值后值不再改变**。

当final修饰一个**基本数据类型**时，表示该基本数据类型的值**一旦在初始化后便不能发生变化**；如果final修饰一个**引用类型**时，则在对**其初始化之后便不能再让其指向其他对象了，但该引用所指向的对象的内容是可以发生变化的**。本质上是一回事，因为**引用的值是一个地址，final要求值，即地址的值不发生变化**。

**final修饰一个成员变量（属性），必须要显示初始化**。一种是在变量声明的时候初始化；第二种方法是在声明变量的时候不赋初值，但是要在这个变量所在的类的所有的构造函数中对这个变量赋初值。

　　当函数的参数类型声明为final时，说明该参数是只读型的。即你可以读取使用该参数，但是无法改变该参数的值。

#### 1.11.2 类的final变量和普通变量有什么区别？

     当用final作用于类的成员变量时，成员变量（注意是类的成员变量，局部变量只需要保证在使用之前被初始化赋值即可）必须在定义时或者构造器中进行初始化赋值，而且final变量一旦被初始化赋值之后，就不能再被赋值了。

public class Test {

public static void main(String[] args) {

String a = "hello2";

final String b = "hello"; //编译器常量，b+2的时候直接去常量池找到了a对应的地址

String d = "hello";

String c = b + 2;

String e = d + 2;

System.out.println((a == c));

System.out.println((a == e));

}

}

输出结果是 true，false

里面就是final变量和普通变量的区别了，当final变量是**基本数据类型以及String类型**时，如果在编译期间能知道它的确切值，则编译器会把它当做编译期常量使用。也就是说在用到该final变量的地方，相当于直接访问的这个常量，不需要在运行时确定。这种和C语言中的**宏替换**有点像。因此在上面的一段代码中，由于变量b被final修饰，因此会被当做编译器常量，所以在使用到b的地方会直接将变量b 替换为它的值。而对于变量d的访问却需要在运行时通过链接来进行。想必其中的区别大家应该明白了，不过要注意，只有在编译期间能确切知道final变量值的情况下，编译器才会进行这样的优化。

public class Test {

public static void main(String[] args) {

String a = "hello2";

final String b = getHello();

String c = b + 2;

System.out.println((a == c));

}

public static String getHello() {

return "hello";

}

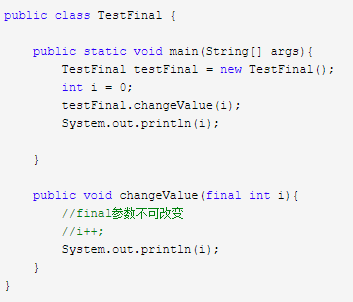
}

输出false。这里要注意一点就是：不要以为某些数据是final就可以在编译期知道其值，通过变量b我们就知道了，在这里是使用getHello()方法对其进行初始化，他要在运行期才能知道其值。

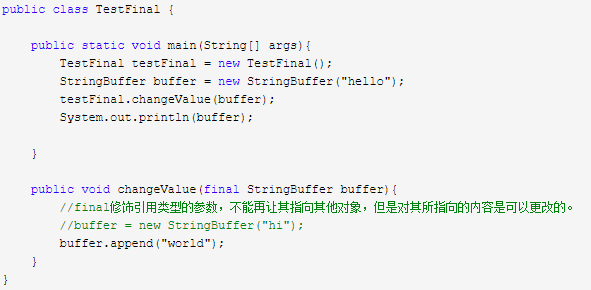
#### 1.11.3 final参数

     在实际应用中，我们除了可以用final修饰成员变量、成员方法、类，还可以修饰参数、若某个参数被final修饰了，则代表了该参数是不可改变的。如果在方法中我们修改了该参数，则编译器会提示你：The final local variable i cannot be assigned. It must be blank and not using a compound assignment。

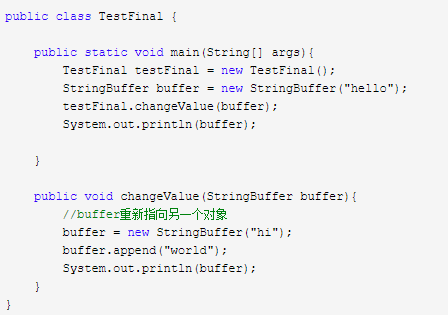
其实参数也分为基本数据类型还有对象类型的。



changeValue方法中的参数i用final修饰之后，就不能在方法中更改变量i的值了。值得注意的一点，方法changeValue和main方法中的变量i根本就不是一个变量，因为java参数传递采用的是值传递，对于基本类型的变量，相当于直接将变量进行了拷贝。所以即使没有final修饰的情况下，在方法内部改变了变量i的值也不会影响方法外的i。



用final进行修饰虽不能再让buffer指向其他对象，但对于buffer指向的对象的内容是可以改变的。现在假设一种情况，如果把final去掉，结果又会怎样？看下面的代码：



结果是

hiworld

hello

原因在于java采用的是值传递，对于引用变量，传递的是引用的值，也就是说让实参和形参同时指向了同一个对象，因此让形参重新指向另一个对象对实参并没有任何影响。

### 1.12 static关键字

Static可以**修饰成员变量，修饰方法，静态块，还有静态导包四种用法**。

1.static最常用的功能就是修饰类的属性和方法，让他们成为类的成员属性和方法，我们通常将用static修饰的成员称为类成员、静态成员或者静态变量。

2.static的另一个作用，就是修饰成员方法。相比于修饰成员属性，修饰成员方法对于数据的存储上面并没有多大的变化，因为我们从上面可以看出，方法本来就是存放在类的定义当中的。

3.static关键字还有一个比较关键的作用就是 用来形成静态代码块以优化程序性能。static块可以置于类中的任何地方，类中可以有多个static块。在类初次被加载的时候，会按照static块的顺序来执行每个static块，并且只会执行一次。为什么说static块可以用来优化程序性能，是因为它的特性:只会在类加载的时候执行一次。

4.用import static代替import静态导入包是JDK1.5中的新特性。一般我们导入一个类都用 import com…..ClassName;而静态导入是这样：import static com…..ClassName. \* ;这里的多了个static，还有就是类名ClassName后面多了个.\* ，意思是导入这个类里的静态方法。当然，也可以只导入某个静态方法，只要把 .\* 换成静态方法名就行了。然后在这个类中，就可以直接用方法名调用静态方法，这种方法的好处就是可以简化一些操作，例如打印操作System.out.println(…);就可以将其写入一个静态方法print(…)，在使用时直接print(…)就可以了。

### 1.13 访问控制

在Java中，提供了四种访问权限控制：默认访问权限（包访问权限），public，private以及protected。

　　注意，上述四种访问权限，只有默认访问权限和public能够用来修饰类，类既不可以是private的（这样会使得除该类之外，其他任何类都不可以访问它），也不可以是protected的。修饰类的变量和方法四种权限都可以。（本处所说的类针对的是外部类，不包括内部类）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 访问权限修饰 | 类内部调用 | 同包下的外部类调用 | 不同包下的外部类调用 | 继承调用 |
| public | √ | √ | √ | √ |
| protected | √ | √ |  | √ |
| 包访问权限 | √ | √ |  |  |
| private | √ |  |  |  |

（PS：其中private和protected不能修饰一般的类，否则编译就会报“modifier private not allowed here”，如果是内部类就另当别论了）

#### 1.13.1修饰类

**默认访问权限（包访问权限）**：用来修饰类的话，表示该类只对同一个包[子包也不行,

例如com.example.class1.java为默认访问权限,它不能在com.example.test.class2.java中import导入]中的其他类可见。

**public**：用来修饰类的话，表示该类对其他所有的类都可见。

#### 1.13.2修饰类的方法和变量

**默认访问权限（包访问权限）**：如果一个类的方法或变量被包访问权限修饰，也就意味着只能在同一个包中的其他类中显示地调用该类的方法或者变量，在不同包中的类中不能显示地调用该类的方法或变量[子包也是不可见的]。

**private**：如果一个类的方法或者变量被private修饰，那么这个类的方法或者变量只能在该类本身中被访问，在类外以及其他类中都不能显示地进行访问。

**protected**：如果一个类的方法或者变量被protected修饰，对于同一个包的类，这个类的方法或变量是可以被访问的。对于不同包的类，只有继承于该类的类才可以访问到该类的方法或者变量。

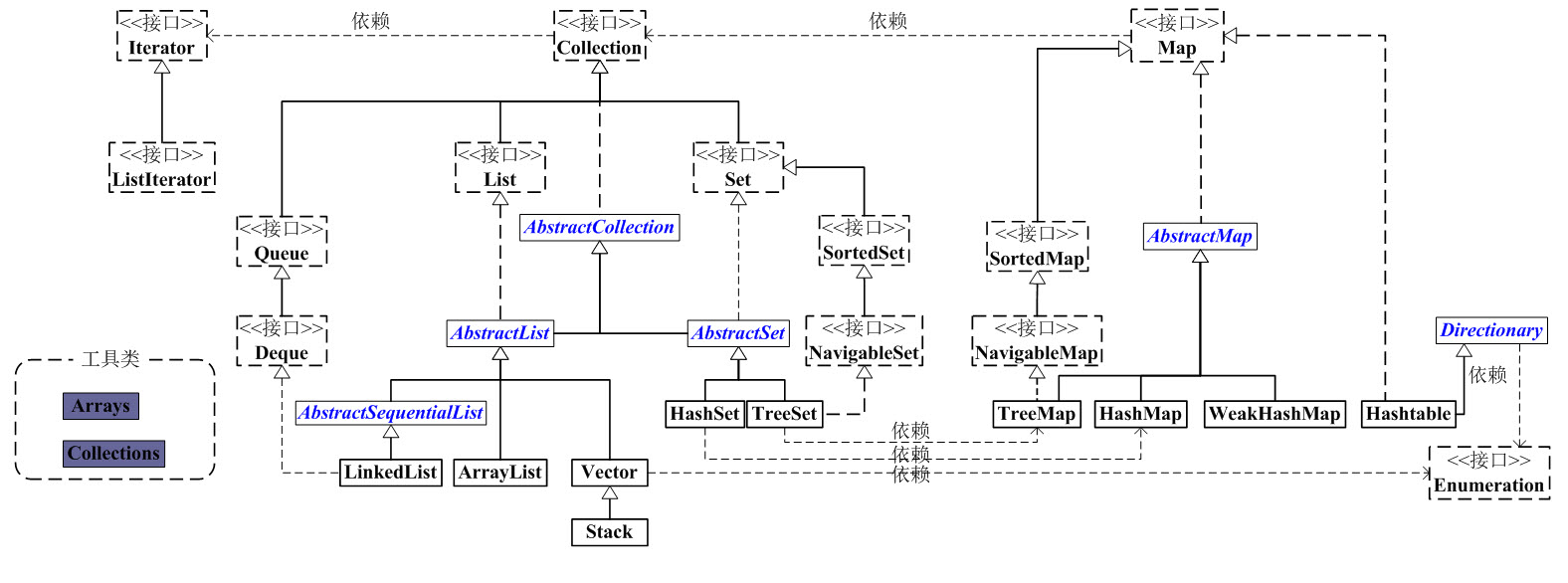
**public**：被public修饰的方法或者变量，在任何地方都是可见的

## 2.集合

### 2.1集合概述



简化版如下



Java集合是java提供的工具包，包含了常用的数据结构：集合、链表、队列、栈、数组、映射等。Java集合工具包位置是java.util.\*  
Java集合主要可以划分为4个部分：List列表、Set集合、Map映射、工具类(Iterator迭代器、Enumeration枚举类、Arrays和Collections)。

### 2.2 [Collection架构](https://www.cnblogs.com/skywang12345/p/3308513.html)

<https://www.cnblogs.com/skywang12345/p/3308498.html>