1. Java基础
   1. Java内部类
      1. 常规内部类

常规内部类没有用static修饰且定义在在外部类类体中。

常规内部类中的方法可以直接使用外部类的实例变量和实例方法。  
 在常规内部类中可以直接用内部类创建对象

* + 1. 静态内部类

与类的其他成员相似，可以用static修饰内部类，这样的类称为静态内部类。静态内部类与静态内部方法相似，只能访问外部类的static成员，不能直接访问外部类的实例变量，与实例方法，只有通过对象引用才能访问。  
由于static内部类不具有任何对外部类实例的引用，因此static内部类中不能使用this关键字来访问外部类中的实例成员，但是可以访问外部类中的static成员。这与一般类的static方法想通

* + 1. 局部内部类

在方法体或语句块（包括方法、构造方法、局部块或静态初始化块）内部定义的类成为局部内部类。  
局部内部类不能加任何访问修饰符，因为它只对局部块有效。  
局部内部类只在方法体中有效，就像定义的局部变量一样，在定义的方法体外不能创建局部内部类的对象  
在方法内部定义类时，应注意以下问题：  
方法定义局部内部类同方法定义局部变量一样，不能使用private、protected、public等访问修饰说明符修饰，也不能使用static修饰，但可以使用final和   abstract修饰方法中的内部类可以访问外部类成员。对于方法的参数和局部变量，必须有final修饰才可以访问。  
static方法中定义的内部类可以访问外部类定义的static成员

* + 1. 匿名内部类

定义类的最终目的是创建一个类的实例，但是如果某个类的实例只是用一次，则可以将类的定义与类的创建，放到与一起完成，或者说在定义类的同时就创建一个类，以这种方法定义的没有名字的类成为匿名内部类。  
声明和构造匿名内部类的一般格式如下：  
   new ClassOrInterfaceName(){s

/\*类体\*/

}

 匿名内部类可以继承一个类或实现一个接口，这里的ClassOrInterfaceName是匿名内部类所继承的类名或实现的接口名。但匿名内部类不能同时实现一个接口和继承一个类，也不能实现多个接口。如果实现了一个接口，该类是Object类的直接子类，匿名类继承一个类或实现一个接口，不需要extends和implements关键字。  
由于匿名内部类没有名称，所以类体中不能定义构造方法，由于不知道类名也不能使用关键字来创建该类的实例。实际上匿名内部类的定义、构造、和第一次使用都发生在同样一个地方。此外，上式是一个表达式，返回的是一个对象的引用，所以可以直接使用或将其复制给一个对象变量。

* 1. final关键字
     1. final用法
     2. final的意义
  2. JAVA基类
     1. Java基类是？有哪些方法？

Java.lang.Object

对象相关{getClass()

protected clone()

toString()

equals()

hashCode()

private registerNatives()

}

线程相关{

notify()

notifyAll()

wait(空|long|long,int)

}

GC相关的{finalize()}。

* + 1. equals和==的区别？

"=="操作符的作用

1、用于基本数据类型的比较

2、判断引用是否指向堆内存的同一块地址。

equals的作用：

用于判断两个变量的内容是否相同，返回值为布尔类型

* + 1. hashCode()的作用？

hashcode方法返回该对象的哈希码值。支持该方法是为哈希表提供一些优点，例如，java.util.Hashtable 提供的哈希表。 Object.hashCode()实际上返回的是对象的引用地址，但子类可以覆盖此方法，实现不同的杂凑算法。

* + 1. hashCode()方法和equals()方法的关系

如果两个对象equals()相等，那么hashCode()必须相等；反之，则不一定。hashCode()相等，可能equals()不等，但这个概率不能太高，否则将增加HashMap冲突的可能性，而降低查找的效率。

* + 1. 重载与重写

首先我们来讲讲：**重载(Overloading)**

（1） 方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。

重载Overloading是一个类中多态性的一种表现。

（2） [Java](http://java.chinaitlab.com/)的方法重载，就是在类中可以创建多个方法，它们具有相同的名字，但具有不同的参数和不同的定义。

调用方法时通过传递给它们的不同参数个数和参数类型来决定具体使用哪个方法, 这就是多态性。

（3） 重载的时候，方法名要一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

然后我们再来谈谈 重写（Overriding）

（1） 父类与子类之间的多态性，对父类的函数进行重新定义。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。在Java中，子类可继承父类中的方法，而不需要重新编写相同的方法。

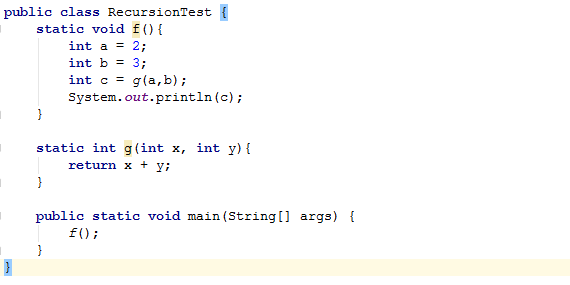
但有时子类并不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

（2）若子类中的方法与父类中的某一方法具有相同的方法名、返回类型和参数表，则新方法将覆盖原有的方法。如需父类中原有的方法，可使用super关键字，该关键字引用了当前类的父类。

（3）子类函数的访问修饰权限不能少于父类的；

* 1. 递归原理分析
     1. 函数调用原理

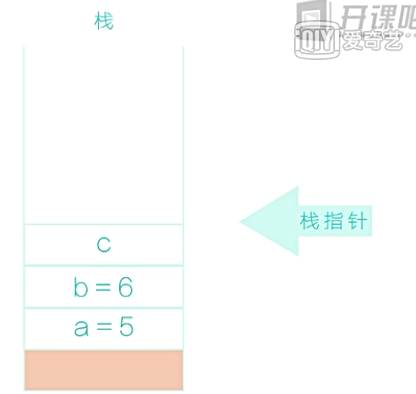
对于计算机而言递归调用和普通的其他的函数调用并没有本质的区别，只要我们明白了一般调用的过程也就能了解递归的过程。递归调用无非就是被调用函数是主调函数本身而已。



假设有函数f去调用g，首先要清楚两件事，第一函数调用的执行过程是串行的，也就是说当被调用的函数正在执行的时候，主调函数是停止工作的，它只是在等待被调用函数的返回，这一过程通常通过栈结构来完成；第二形式参数是调用函数时临时去创建的，被调函数执行完毕形式参数就消失了，如果我们调用多次，则多次创建临时的形式参数，下面来看看具体的过程。

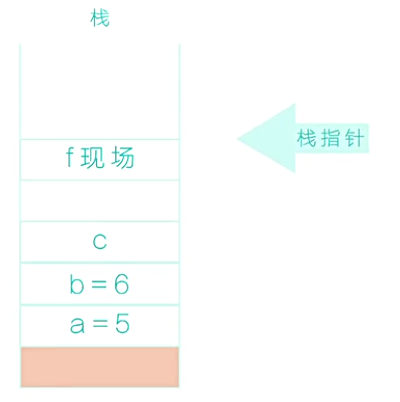
这里有两个函数f和g，f函数中定义了若干局部变量，并且调用了g函数。我们假设程序是从f开始执行的，f中定义了三个局部变量a b和c，这就需要在栈中为三个变量分配空间。

栈结构有一个被称为栈指针的单元，他指向栈顶元素，在栈指针以上是未分配的空间。在f的左大括号分配局部变量，在f的右大括号则释放已分配的局部变量。

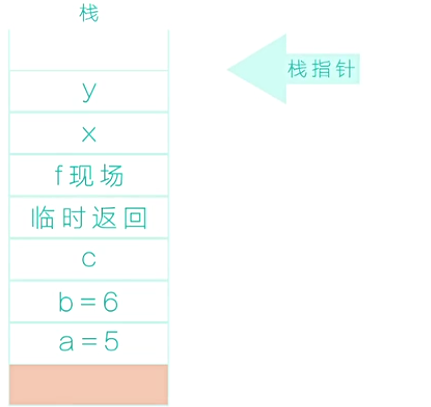


当程序转向去执行g函数的时候，f函数在做什么呢？它仅仅是停留在这句话上一直等待。直到被调用函数返回为止。这里就存在一个问题，当我们转向去执行f方法时，如果我们不做其他的任何动作，那么当调用结束时，我们如何知道该返回到哪里继续执行呢?那里原来的环境都还在吗？因为执行g方法很可能会破坏了原来的环境。

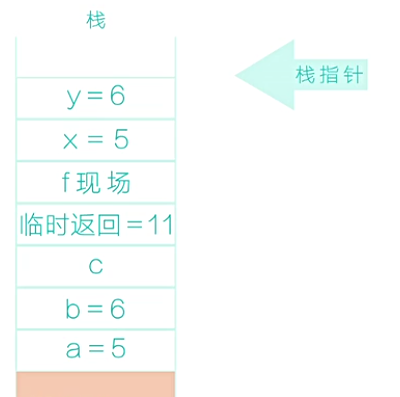
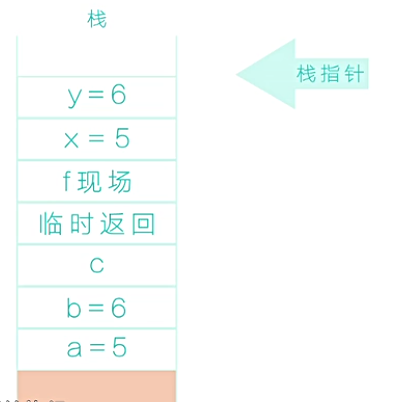
计算机采用了一种这样的策略，在转向另一个方法执行前，它可以在栈中保存当前函数的执行现场，当函数返回时再把这个现场恢复。



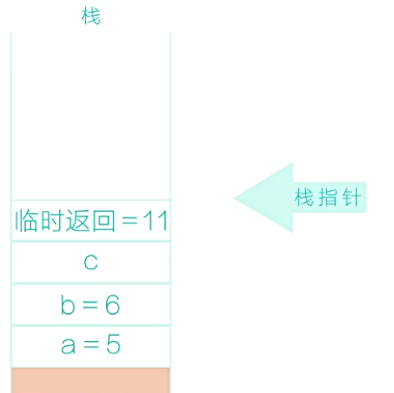
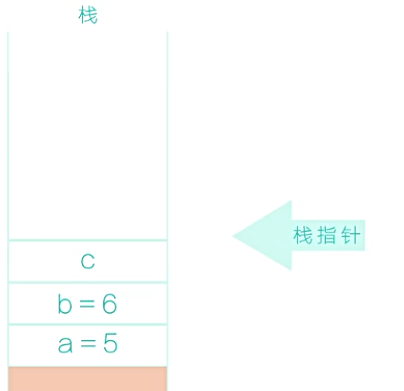
除了现场外，这里还有几个要素，a和b我们叫做实际参数，简称实参，他们已经存在于内存中了。而x和y叫做形式参数，简称形参，他们是不存在的，只是在函数调用的时候临时创建。这里还有一个函数的返回值需要我们去记录。



实际上需要在栈中创建这几个变量，返回值、形参。栈变量分配好以后，我们把a拷贝给x，b拷贝给y，然后就转向g方法开始执行。这里的逻辑很简单，x和y相加存入到临时变量中，临时变量是我们刚刚分配的。



到右括号这时函数就需要返回，释放掉开始分配的那些局部变量。栈指针一直向下落，释放变量。现在一切都恢复到g函数调用之前，只是多了一个临时的返回值放在栈顶。然后把它赋值给c，然后释放掉这个临时的返回值。

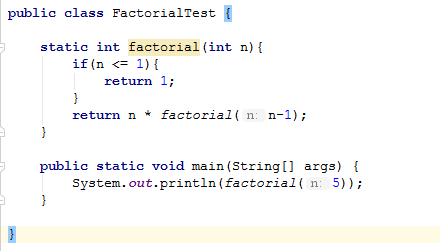
 

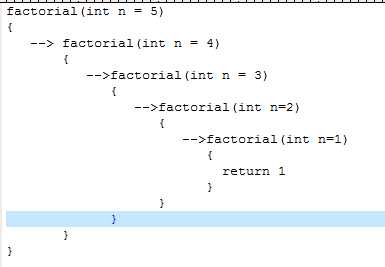
这时候就恢复到，这句话还没有执行时候的状态。继续执行，直到f的右大括号，所有分配的变量都已经释放了。整个函数的调用过程就结束了。

这里的关键是x和y的生命周期是很短暂的，它只存在于g函数被调用的一瞬间，如果g被调用多次，x和y也会出生和死亡多次。明白了这个道理我们再看递归现象就很容易了。

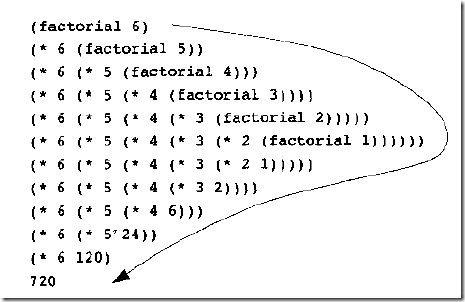
* + 1. 实例讲解

阶乘





理解递归的关键在于理解形式参数在栈中存在多层这个情况。在求解6的阶乘时，递归过程如下所示

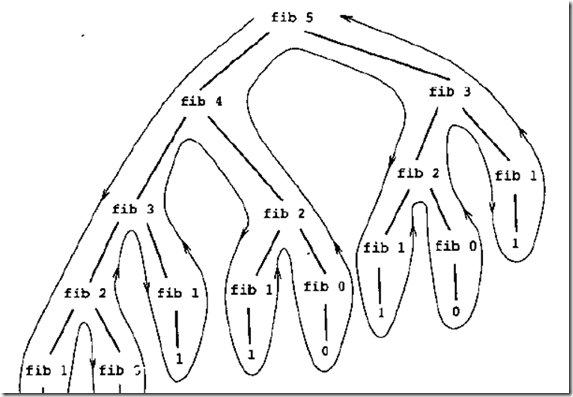


我们会惊奇的发现这个过程和栈的工作原理一致对，递归调用就是通过栈这种数据结构完成的。整个过程实际上就是一个栈的入栈和出栈问题。然而我们并不需要关心这个栈的实现，这个过程是由系统来完成的。那么递归中的“递”就是入栈，递进；“归”就是出栈，回归。

**斐波那契数列**

斐波那契数列的递推公式:Fib(n)=Fib(n-1)+Fib(n-2)，指的是如下所示的数列：

1、1、2、3、5、8、13、21.....



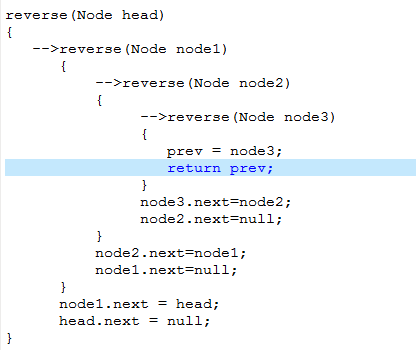
* + 1. 递归实现单链表反转

//反转

**public** LinkNode reverse(LinkNode node) {  
 LinkNode prev = **null**;  
 **if** (node == **null** || node.**next** == **null**) {  
 prev = node;  
 } **else** {  
 LinkNode tmp = reverse(node.**next**);  
 node.**next**.**next** = node;  
 node.**next** = **null**;  
 prev = tmp;  
 }  
 **return** prev;  
}



调用过程：



* 1. 值传递和引用传递

值传递：传递的是实际参数的一个副本，这个值可能是基本类型，也可能是引用类型的地址.

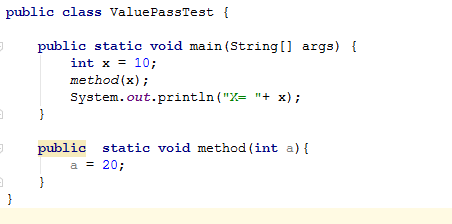
引用传递：传递的是实际参数的**地址**的一个副本.

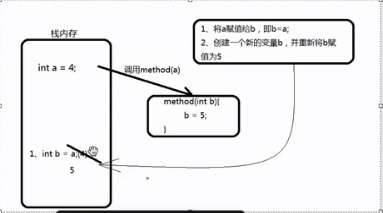
在java中，只有值传递.

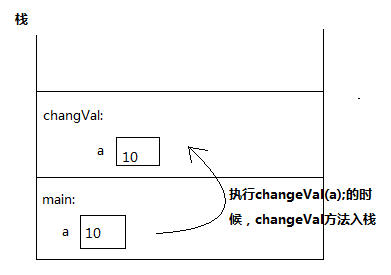
* + 1. 值传递

***基本数据类型***

值传递通常传递值得副本，而不会改变原有的值。

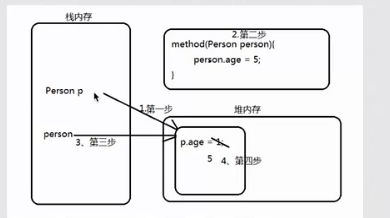


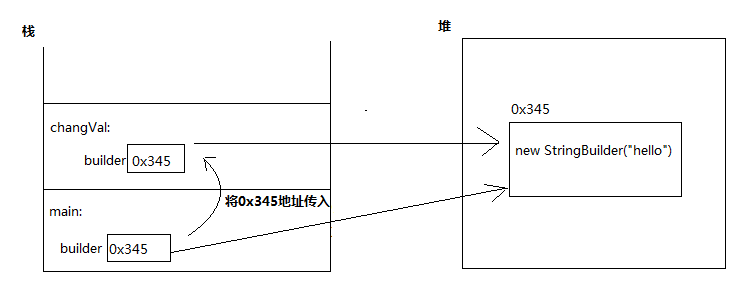




***引用类型***

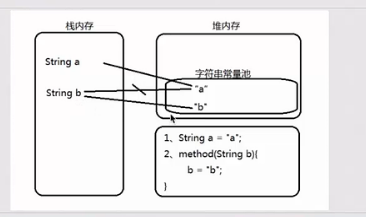
引用传递传递的是对象的引用，也就是传递的是堆内存中的地址，也就是将自己的对象共享给其他对象使用，这种情况下就会造成一个地方改变了，其他地方也会跟着变化。

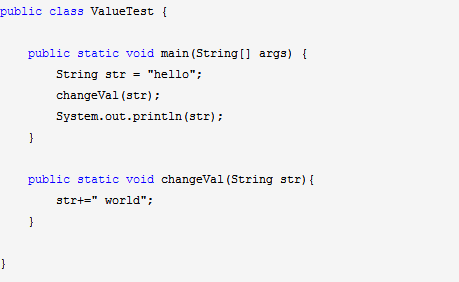


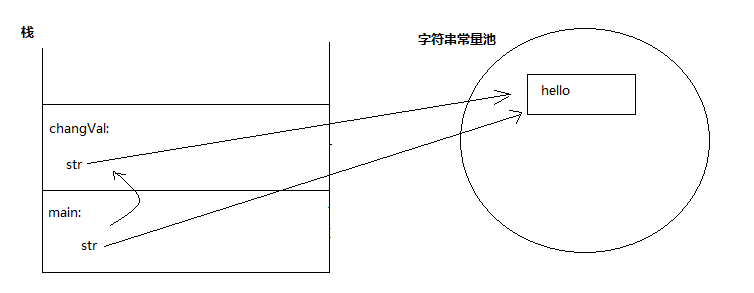


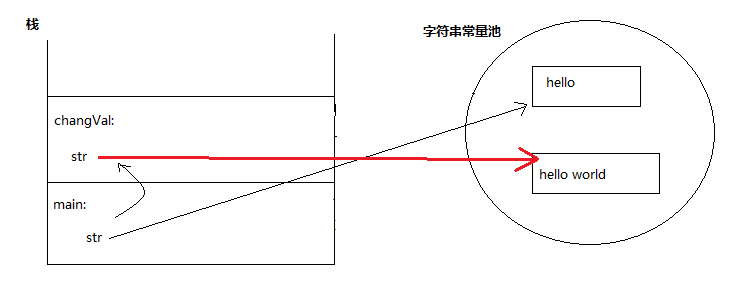
***String类型***

重新赋值时会重新创建对象，不会更改原有数据。









* + 1. 引用传递

引用传递，传递的是实际参数的地址.

如下图，有变量builder，值指向了堆内存地址(假设为0x456)。但是builder变量本身也有地址(假设为0x123)。传递的时候，传的就是0x123.

