# Java 类型变量

**自动类型转换遵循下面的规则：**

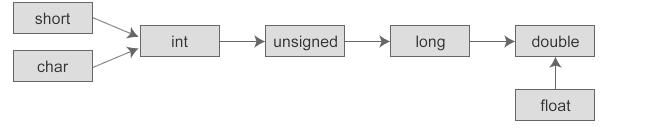
1.若参与运算的数据类型不同，则先转换成同一类型，然后进行运算。

2.转换按数据长度增加的方向进行，以保证精度不降低。例如int型和long型运算时，先把int量转成long型后再进行运算。

3.所有的浮点运算都是以双精度进行的，即使仅含float单精度量运算的表达式，也要先转换成double型，再作运算。

4.char型和short型参与运算时，必须先转换成int型。

5.在赋值运算中，赋值号两边的数据类型不同时，需要把右边表达式的类型将转换为左边变量的类型。如果右边表达式的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度。



注意：

boolean类型不能和任何类型进行转换，会报出类型异常错误。

byte的取值范围是-128—127

合法转化：byte, short, int, char, float, long, double

优先级：double > float > long > int

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

单精度和双精度精确的范围不一样，单精度，也即float，一般在计算机中存储占用4字节，也32位，有效位数为7位；双精度（double）在计算机中存储占用8字节，64位，有效位数为16位。

# 编译执行原理

**类的初始化顺序是：**

1、初始化父类中的静态成员变量和静态代码块。

2、初始化子类中的静态成员变量和静态代码块。

3、初始化父类中的普通成员变量和代码块，在执行父类中的构造方法。

4、初始化子类中的普通成员变量和代码块，在执行子类中的构造方法。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

byte b1=1,b2=2,b3,b6;

final

byte b4=4,b5=6;

b6=b4+b5;

b3=(b1+b2);

System.out.println(b3+b6);

关于上面代码片段叙述正确的是（）

语句：b3=b1+b2编译出错

被final修饰的变量是常量，这里的b6=b4+b5可以看成是b6=10；在编译时就已经变为b6=10了，而b1和b2是byte类型，java中进行计算时候将他们提升为int类型，再进行计算，b1+b2计算后已经是int类型，赋值给b3，b3是byte类型，类型不匹配，编译不会通过，需要进行强制转换。

# Java 基础知识

**值传递与引用传递：**

Java中原始数据类型都是值传递，传递的是值的副本，形参的改变不会影响实际参数的值，引用传递传递的是引用类型数据，包括String,数组，列表, map,类对象等类型，形参与实参指向的是同一内存地址，因此形参改变会影响实参的值。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Final关键字：**

final修饰类、方法、属性！不能修饰抽象类，因为抽象类一般都是需要被继承的，final修饰后就不能继承了。

final修饰属性：

此属性就是一个常量，不能被再次赋值。编译期常量的计算可带入任何计算式中，减轻运行时负担。

final修饰实例域：

对于不可变类的实例域声明为final，则在构建对象后，无法修改其值。对于可变类（如StringBuilder，可通过内部方法对值进行修改），final关键字只是表示存储在变量中的对象引用不会再指示其他对象，但是这个对象的值是可以改变的。因此，final在修饰对象的时候实际上是指对象引用不能被改变。

final修饰类和方法：

将一个类声明为final，这个类不能被继承；

类中方法被声明为final，则其子类不能覆盖（重写）这个方法，而当前类中只有其中的方法会自动成为final，域不会。

final修饰的方法，不允许被子类覆盖。

final修饰的类，不能被继承。

final修饰的成员变量为基本数据类型是，在赋值之后无法改变。当final修饰的成员变量为引用数据类型时，在赋值后其指向地址无法改变，但是对象内容还是可以改变的。

final修饰的引用类型，不能再指向别的东西，但是可以改变其中的内容。

补充：

static, random修饰情况：static保证只有一个值，例如：类中属性static final int I5 = rand.nextInt(20)，即使再创建一个对象，对象中I5的值也不会变。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**值的初始化：**

类变量在不设置初始值时，会进行默认值赋值，而局部方法中声明的变量则必须进行初始化，他不会进行默认值赋值。

Java中基本数据类型默认值为： short: 0 int: 0 long:0 float: 0.0 double: 0.0 char:0 String:null Boolean: false Java中对象引用默认值为null

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**包访问权限：**

默认访问权限无任何修饰词，当前包中所有其他类对那个成员都有访问权限，但是对这个包之外的所有类，这个成员是private。

protected：相同包内的其他类可以访问protected元素。

private：除了包含该成员类之外，其他任何类都无法访问这个成员。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Static关键字：**

被static声明的为静态方法，可以直接通过类名调用而不需要通过对象调用。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Java内存**

Java把内存分成两种，一种叫做**栈内存**，一种叫做**堆内存**。

**在函数中定义的一些基本类型的变量和对象的引用变量都是在函数的栈内存中分配**。当在一段代码块中定义一个变量时，java就在栈中为这个变量分配内存空间，当超过变量的作用域后，java会自动释放掉为该变量分配的内存空间，该内存空间可以立刻被另作他用。

**堆内存用于存放由new创建的对象和数组**。在堆中分配的内存，由java虚拟机自动垃圾回收器来管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，在栈中的这个特殊的变量就变成了数组或者对象的引用变量，以后就可以在程序中使用栈内存中的引用变量来访问堆中的数组或者对象，引用变量相当于为数组或者对象起的一个别名，或者代号。

引用变量是普通变量，定义时在栈中分配内存，引用变量在程序运行到作用域外释放。而数组＆对象本身在堆中分配，即使程序运行到使用new产生数组和对象的语句所在地代码块之外，数组和对象本身占用的堆内存也不会被释放，数组和对象在没有引用变量指向它的时候（比如先前的引用变量x=null时），才变成垃圾，不能再被使用，但是仍然占着内存，在随后的一个不确定的时间被垃圾回收器释放掉。这个也是java比较占内存的主要原因。

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

面向对象的五个基本原则：   
单一职责原则（Single-Resposibility Principle）：一个类，最好只做一件事，只有一个引起它的变化。单一职责原则可以看做是低耦合、高内聚在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。   
开放封闭原则（Open-Closed principle）：软件实体应该是可扩展的，而不可修改的。也就是，对扩展开放，对修改封闭的。   
Liskov替换原则（Liskov-Substituion Principle）：子类必须能够替换其基类。这一思想体现为对继承机制的约束规范，只有子类能够替换基类时，才能保证系统在运行期内识别子类，这是保证继承复用的基础。   
依赖倒置原则（Dependecy-Inversion Principle）：依赖于抽象。具体而言就是高层模块不依赖于底层模块，二者都同依赖于抽象；抽象不依赖于具体，具体依赖于抽象。   
接口隔离原则（Interface-Segregation Principle）：使用多个小的专门的接口，而不要使用一个大的总接口

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**异常抛出**

**Throw和throws**

**1、throws出现在方法头，throw出现在方法体**

**2、throws表示出现异常的一种可能性，并不一定会发生异常；throw则是抛出了异常，执行throw则一定抛出了某种异常。**

**3、两者都是消极的异常处理方式，只是抛出或者可能抛出异常，是不会由函数处理，真正的处理异常由它的上层调用处理。**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**方法的重写（override）两同两小一大原则：**

**方法名相同，参数类型相同**

**子类返回类型小于等于父类方法返回类型，**

**子类抛出异常小于等于父类方法抛出异常，**

**子类访问权限大于等于父类方法访问权限。**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**接口里面定义方法时都用public abstract修饰**

**接口里面定义属性时都用public static final修饰**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

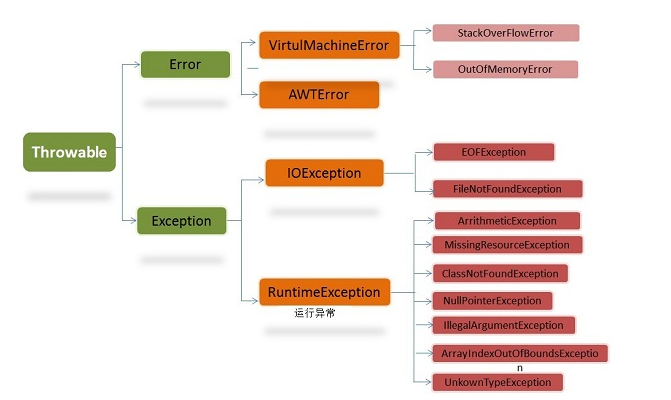
**抽象类;**

**抽象类不能实例化,** **java中，接口和抽象类，都可以声明为对象，只需要在实例化的时候，用一个该接口/抽象类的实现类实例化就行了。**

**接口是一种特殊的抽象类。**

**抽象类的方法，可以是抽象abstract的，也可以是普通的。**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

****

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**ArryList和LinkedList都实现了List接口。**

**ArrayList的内存结构是数组，本质是顺序存储的线性表，插入和删除操作都会引起后续元素移动，效率低，但是随机访问效率高。**

**LinkedList的内存结构是双向链表存储的，链式存储结构插入和删除效率高，不需要移动，但是随机访问效率低，需要从头开始向后依次访问。**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**1. 关于HashMap的一些说法：**

**a) HashMap实际上是一个“链表散列”的数据结构，即数组和链表的结合体。HashMap的底层结构是一个数组，数组中的每一项是一条链表。**

**b) HashMap的实例有俩个参数影响其性能： “初始容量” 和 装填因子。**

**c) HashMap实现不同步，线程不安全。 HashTable线程安全**

**d) HashMap中的key-value都是存储在Entry中的。**

**e) HashMap可以存null键和null值，不保证元素的顺序恒久不变，它的底层使用的是数组和链表，通过hashCode()方法和equals方法保证键的唯一性**

**f) 解决冲突主要有三种方法：定址法，拉链法，再散列法。HashMap是采用拉链法解决哈希冲突的。**

**注： 链表法是将相同hash值的对象组成一个链表放在hash值对应的槽位；**

**用开放定址法解决冲突的做法是：当冲突发生时，使用某种探查(亦称探测)技术在散列表中形成一个探查(测)序列。 沿此序列逐个单元地查找，直到找到给定 的关键字，或者碰到一个开放的地址(即该地址单元为空)为止（若要插入，在探查到开放的地址，则可将待插入的新结点存人该地址单元）。**

**拉链法解决冲突的做法是： 将所有关键字为同义词的结点链接在同一个单链表中 。若选定的散列表长度为m，则可将散列表定义为一个由m个头指针组成的指针数 组T[0..m-1]。凡是散列地址为i的结点，均插入到以T[i]为头指针的单链表中。T中各分量的初值均应为空指针。在拉链法中，装填因子α可以大于1，但一般均取α≤1。拉链法适合未规定元素的大小。**

**2. Hashtable和HashMap的区别：**

**a) 继承不同。**

**public class Hashtable extends Dictionary implements Map**

**public class HashMap extends AbstractMap implements Map**

**b) Hashtable中的方法是同步的，而HashMap中的方法在缺省情况下是非同步的。在多线程并发的环境下，可以直接使用Hashtable，但是要使用HashMap的话就要自己增加同步处理了。**

**c) Hashtable 中， key 和 value 都不允许出现 null 值。 在 HashMap 中， null 可以作为键，这样的键只有一个；可以有一个或多个键所对应的值为 null 。当 get() 方法返回 null 值时，即可以表示 HashMap 中没有该键，也可以表示该键所对应的值为 null 。因此，在 HashMap 中不能由 get() 方法来判断 HashMap 中是否存在某个键， 而应该用 containsKey() 方法来判断。**

**d) 两个遍历方式的内部实现上不同。Hashtable、HashMap都使用了Iterator。而由于历史原因，Hashtable还使用了Enumeration的方式 。**

**e) 哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。**

**f) Hashtable和HashMap它们两个内部实现方式的数组的初始大小和扩容的方式。HashTable中hash数组默认大小是11，增加的方式是old\*2+1。HashMap中hash数组的默认大小是16，而且一定是2的指数。**

**注： HashSet子类依靠hashCode()和equal()方法来区分重复元素。**

**HashSet内部使用Map保存数据，即将HashSet的数据作为Map的key值保存，这也是HashSet中元素不能重复的原因。而Map中保存key值的,会去判断当前Map中是否含有该Key对象，内部是先通过key的hashCode,确定有相同的hashCode之后，再通过equals方法判断是否相同。**