[Math 1](#_Toc528662847)

[求随机数 2](#_Toc528662848)

[求幂数 2](#_Toc528662849)

[求四舍五入后的值 3](#_Toc528662850)

[求最小整数 4](#_Toc528662851)

[自动装箱和拆箱 4](#_Toc528662852)

[类关系 5](#_Toc528662853)

[UML介绍(了解下就行) 5](#_Toc528662854)

[Java中的类关系(必须掌握) 6](#_Toc528662855)

[泛化 6](#_Toc528662856)

[实现 6](#_Toc528662857)

[关联 7](#_Toc528662858)

[组合 8](#_Toc528662859)

[聚合 8](#_Toc528662860)

[依赖 9](#_Toc528662861)

[OOD的5大原则 9](#_Toc528662862)

[SRP原则 9](#_Toc528662863)

[OCP原则 10](#_Toc528662864)

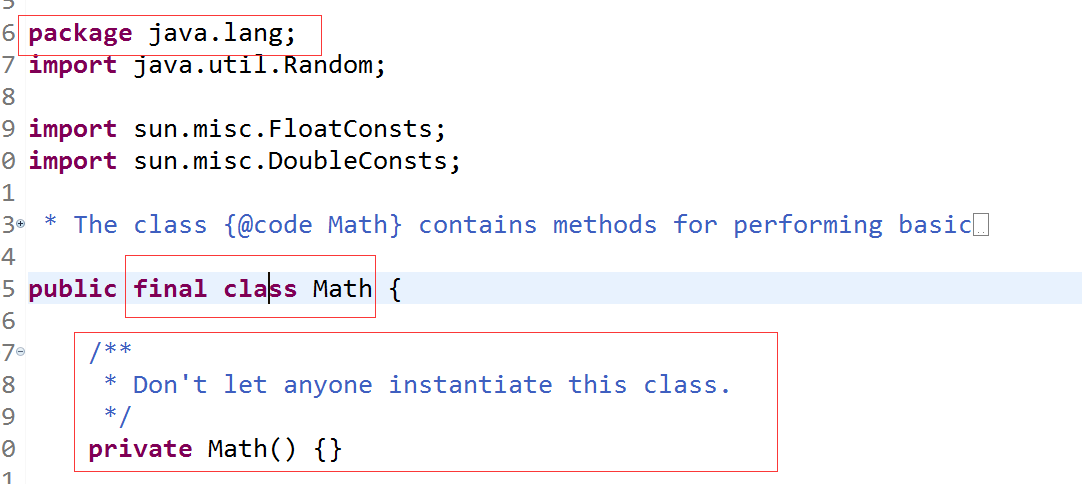
[LSP原则 10](#_Toc528662865)

[DIP法则 10](#_Toc528662866)

[ISP原则 11](#_Toc528662867)

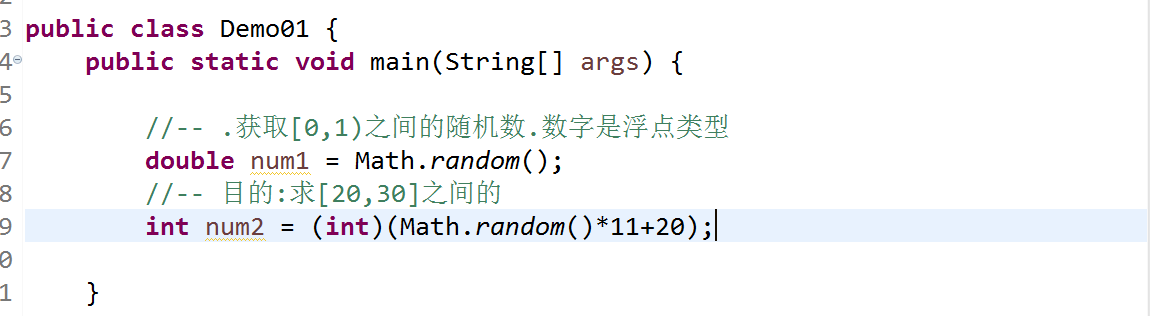
[参考 11](#_Toc528662868)

# Math

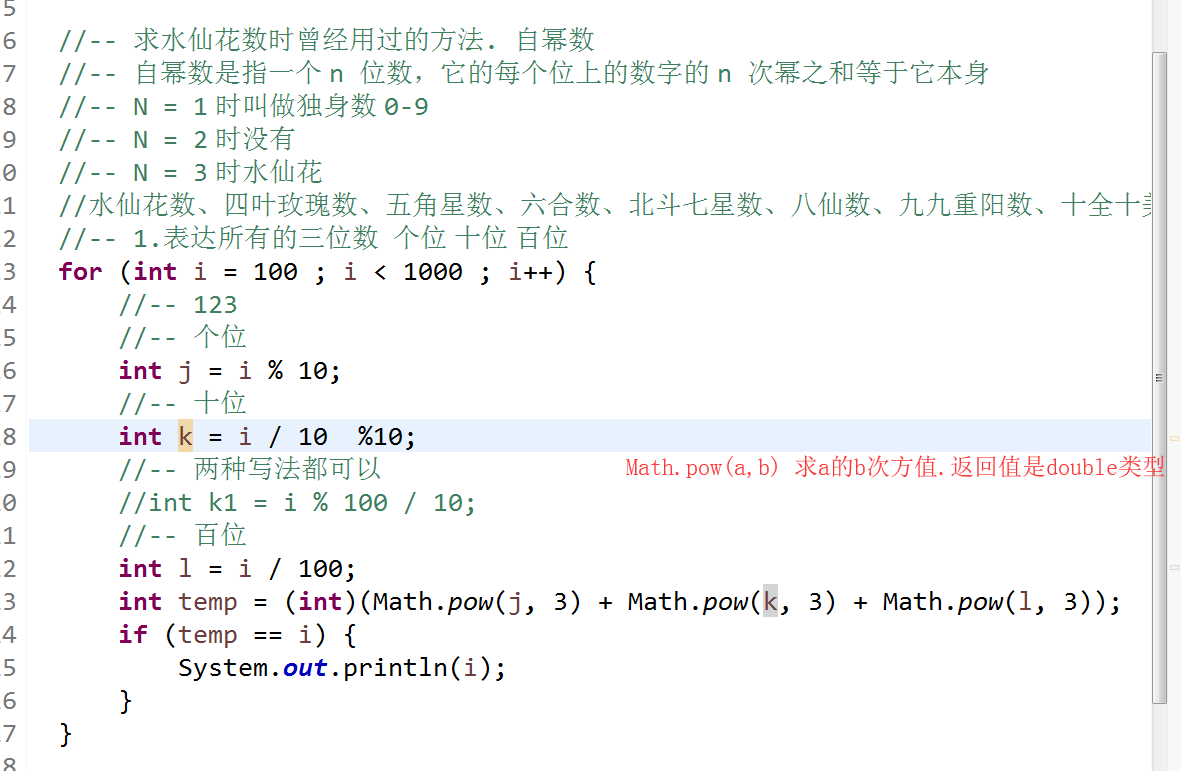


这个类中定义几乎所有的数学公式.有几个常用的是我们必须要掌握的

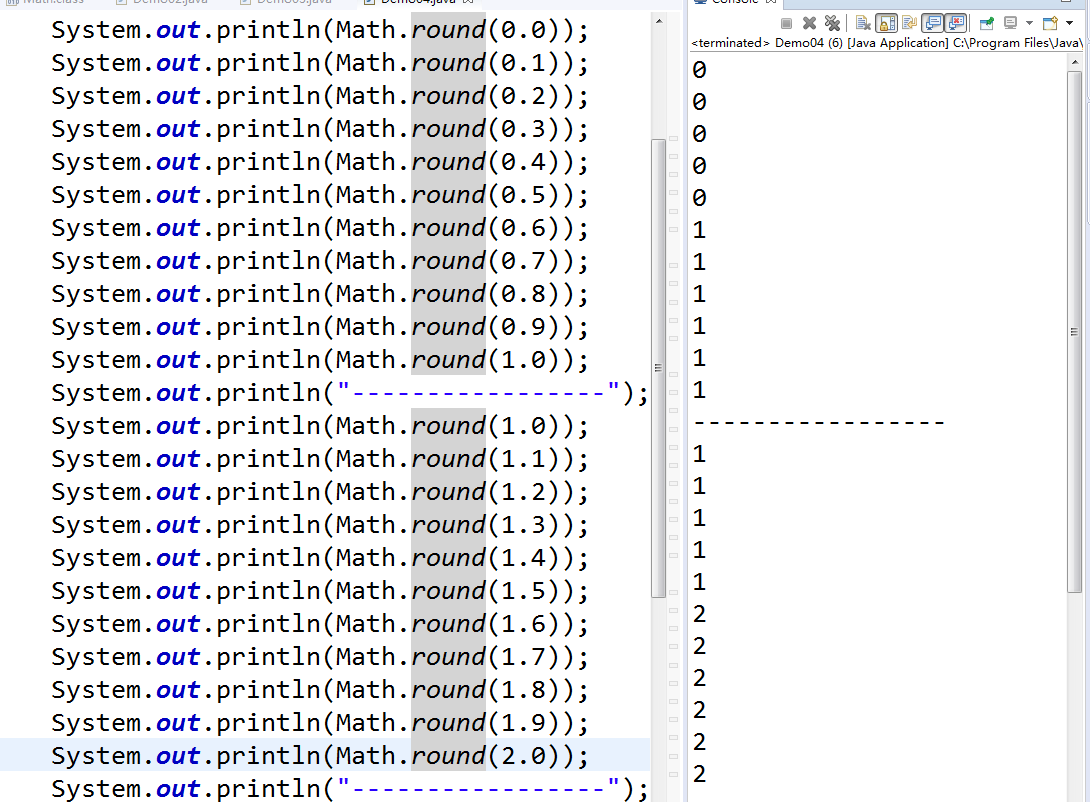
## 求随机数



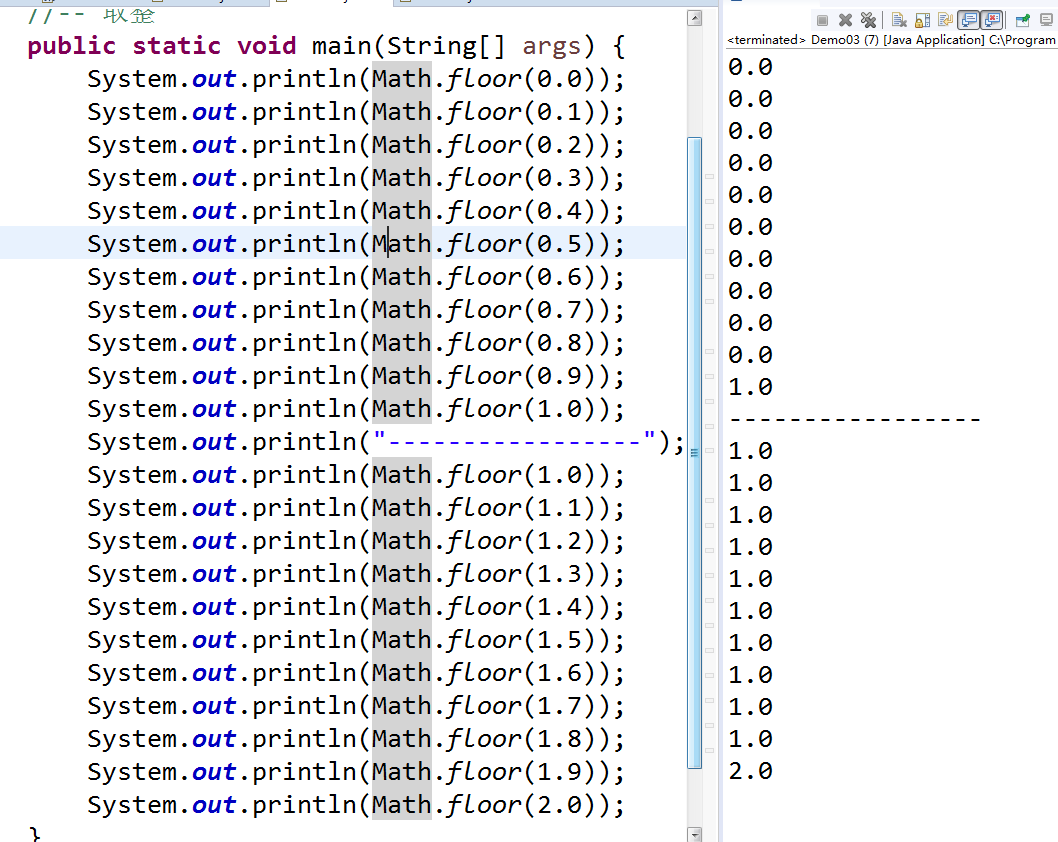
## 求幂数



## 求四舍五入后的值



## 求最小整数



Floor会直接取整数.不会进行四舍五入的.

# 自动装箱和拆箱

在Java中有原生数据类型和引用数据类型.原生数据有与之对应8个引用数据类型.我们把原生转成引用的过程叫做装箱.反之叫做拆箱.

在JDK1.5之前所有的装箱和拆箱需要手工完成.在JDK1.5之后是自动完成.

|  |  |
| --- | --- |
| 原生类型 | 引用类型 |
| byte | Byte |
| short | Short |
| int | Integer |
| long | Long |
| char | Character |
| float | Float |
| double | Double |
| boolean | Boolean |

还有一个细节需要注意.这8个引用类型都是被final修饰的,即都不可以被继承.并且所有的数字都是继承Number,只有Boolean和Character没有继承Number.

# 类关系

## UML介绍(了解下就行)

有一个标准叫做UML,统一建模语言或者叫做标准建模语言.(Unified Modeling Language).在Java中同样支持UML.

UML定义了5类10种模型图

|  |  |  |
| --- | --- | --- |
| 类型 | 具体种类 | 含义 |
| 用例图 | 用例图 | 从用户的角度指出系统的功能和各功能的操作者. |
| 静态图 | 类图 | 定义系统中的类,描述类的结构和类与类之间的关系 |
| 对象图 | 是类图的实例 |
| 包图 | 由包或类组成.用于表示分层结构 |
| 行为图 | 状态图 |  |
| 活动图 |  |
| 交互图 | 顺序图 |  |
| 合作图 |  |
| 实现图 | 构件图 |  |
| 配置图 |  |

第一类是用例图，用例图从用户角度描述系统的功能，并指出各功能的操作者。

第二类是静态图，包括类图，对象图和包图。其中类图用于定义系统中的类，包括描述类之间的联系（如关联、依赖、聚合等）以及类的内部结构，即类的属性和操作。因此类图描述系统中类的静态结构，它描述的是一种静态关系，在系统的整个生命周期都是有效的。对象图所使用的表示符号与类图几乎相同，它们的不同点在于对象图只显示类的对象实例，而不是实际的类。 一个对象图是类图的一个实例。由于对象有生命周期，所以对象图只在系统的某段时间有效。包图由包或类组成，主要表示包与包或包与类之间的关系，包图用于描述系统的分层结构。

第三类是行为图，描述系统的动态模型和对象间的交互关系。一种是状态图，它描述一类对象可能的状态以及事件发生时状态迁移的条件。通常状态图是对类图的补充，我们并不需要对所有的类绘制状态图，而只要为那些有多个状态，并且其行为受外界影响而会发生改变的类绘制状态图。另一种称为活动图，它描述为满足用例要求所要进行的活动及活动见的约束关系，使用活动图可以方便地表示并进行活动。

第四类是交互图，它描述对象间的交互关系。一种称之为顺序图，用以描述对象间的动态合作关系，它强调对象之间消息发送的顺序，同时也显示对象之间的交互过程。另一种是合作图，它着重描述对象间的协作关系。合作图和顺序图类似，显示对象间的动态合作关系。除了显示信息交换外，合作图还显示对象以及对象之间的关系。如果强调时间和顺序，应当使用顺序图；如果强调通信关系，则可以选择合作图，这两种图合称为交互图。

第五类是实现图，包括构件图和配置图。构件图描述代码部件的物理结构和各部件之间的依赖关系。一个部件肯能是一个资源代码部件、一个二进制部件或一个可执行部件，它包含逻辑类和实现类的有关信息。构件图有助于分析和理解部件之间的相互影响程度。配置图定义系统中的软硬件物理体系结构，它可以显示实际的计算机和设备（用节点表示）以及它们之间的连接关系，也可以显示连接的类型和部件或部件之间的依赖性。在节点内部，放置可执行的部件和对象，以显示节点和可执行单元之间的对应关系。

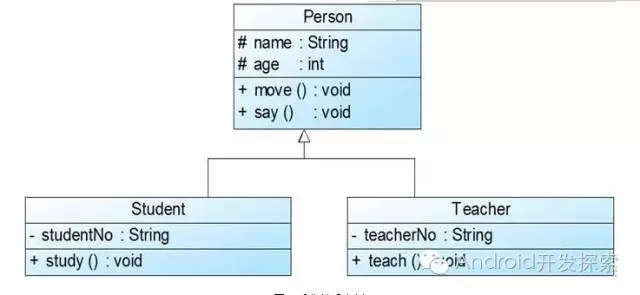
## Java中的类关系(必须掌握)

Java是面向对象的.我们在设计项目时更多的是思考项目中有哪些类,以及这些类之间的关系.我们一般会用一下几种说法来描述这些类关系

* Is – a
* Like – a
* Has – a
* Use – a
* Contains - a

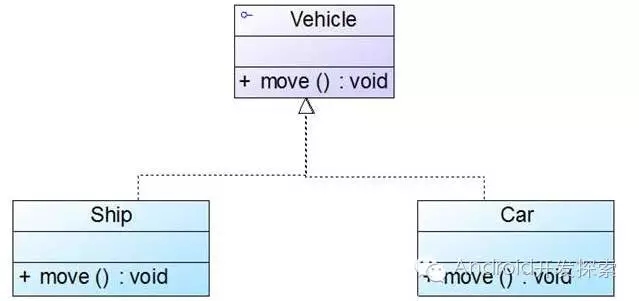
### 泛化

泛化就是继承的意思对应的是is-a关系.在UML类图中使用空心三角加实线来表示.箭头指向的是父类



### 实现

就是实现一个接口的意思.对应的是like-a关系.在UML类图中使用空心三角加虚线来表示.箭头指向的是接口



### 关联

关联关系  关联体现的是两个类之间语义级别的一种强依赖关系，比如我和我的朋友，这种关系比依赖更强、不存在依赖关系的偶然性、关系也不是临时性的，一般是长期性的，而且双方的关系一般是平等的。关联可以是单向、双向的。表现在代码层面，为被关联类B以类的属性形式出现在关联类A中，也可能是关联类A引用了一个类型为被关联类B的全局变量。在UML类图设计中，关联关系用由关联类A指向被关联类B的带箭头实线表示，在关联的两端可以标注关联双方的角色和多重性标记。

和依赖相比.关联的依赖性更强,也更加稳定.并且关联还需要负责被关联的对象的生命周期.

关联可以分为三种

1. 单向关联
2. 双向关联
3. 自关联

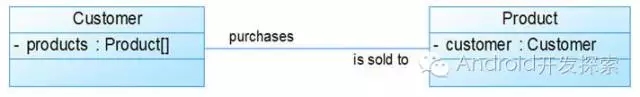
#### 单向关联

一方知道另外一方.比如学生和Java课程之间的关系就是单向关联.学生要学习Java. 但是Java不知道要被谁学习了.



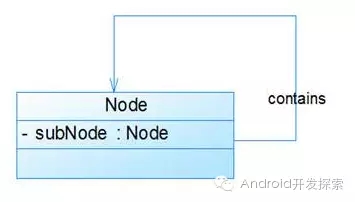
#### 双项关联

简单说就是双方互相知道对方的存在.比如学生和老师.学生知道他/她的上课老师是谁.老师也知道他/她要教哪些学生.



#### 自关联

就是自己关联自己了.类似于像递归那样.在自己的类中用含有一个自己作为成员变量



无论是哪一种关联.它表达都是一种has-a关系.在代码中的体现都是以成员变量的形式存在.

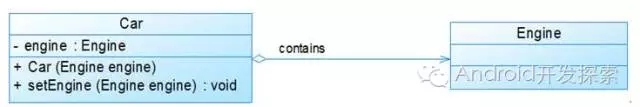
### 组合

组合也是关联关系的一种特例，它体现的是一种contains-a的关系，这种关系比聚合更强，也称为强聚合。它同样体现整体与部分间的关系，但此时整体与部分是不可分的，整体的生命周期结束也就意味着部分的生命周期结束，比如人和人的大脑。表现在代码层面，和关联关系是一致的，只能从语义级别来区分。在UML类图设计中，组合关系以实心菱形加实线箭头表示。



### 聚合

聚合是关联关系的一种特例，它体现的是整体与部分的关系，即has-a的关系。此时整体与部分之间是可分离的，它们可以具有各自的生命周期，部分可以属于多个整体对象，也可以为多个整体对象共享。比如计算机与CPU、公司与员工的关系等，比如一个航母编队包括海空母舰、驱护舰艇、舰载飞机及核动力攻击潜艇等。表现在代码层面，和关联关系是一致的，只能从语义级别(逻辑)来区分。在UML类图设计中，聚合关系以空心菱形加实线箭头表示。

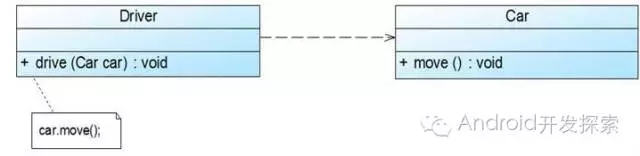


聚合是一种强关联.它的代码体现和关联一样.所以想从代码层次区分关联和聚合是不现实的.只有从逻辑上来进行区分.

### 依赖

简单的理解，依赖就是一个类A使用到了另一个类B，而这种使用关系是具有偶然性的、临时性的、非常弱的，但是类B的变化会影响到类A。比如某人要过河，需要借用一条船，此时人与船之间的关系就是依赖。表现在代码层面，为类B作为参数被类A在某个method方法中使用。在UML类图设计中，依赖关系用由类A指向类B的带箭头虚线表示。

再比如人类和氧气.人类的生存依赖氧气.



被依赖的类一般以参数的形式出现.如上图Driver是不需要负责Car的生命周期的(Car对象的创建和销毁).

对应的是use-a关系.

#### 这些关系的强弱顺序

几种关系所表现的强弱程度依次为：泛化 = 实现 > 组合 > 聚合 > 关联 > 依赖。

# OOD的5大原则

## SRP原则

职责单一原则.简单说就是只做一件事情.Student类中只放Student的特征不放Teacher的特征.

## OCP原则

软件实体(类,模块,函数等)应该是可以扩展的,但是不可修改.

OCP的两个主要特征:

(1):对于扩展是开放的(open for extension):意味着模块的行为是可以扩展的,当应用的需求改变时,我们可以对模块进行扩展,使其具有满足那些改变的新行为,换句话说,我们可以改变模块的功能.

(2):对于修改是关闭的(closed for modification):对模块行为进行扩展时,不必改动模块的源代码或者二进制代码,无论是可链接的库,DLL或者EXE文件,都不必修改.

 那如何去实现呢,答案就是抽象.由于模块依赖于抽象,所以它对于更改可以是封闭的,同时,通过从这个抽象体派生,可以扩展此模块的行为.

在此常用的两个设计模式是:STRATEGY模式跟TEMPLATE METHOD模式.

其实是不可能做到完全封闭,那么就必须有策略地对待这个问题,也就是说,设计人员必须对于他设计的模块应该对哪种变化封闭做出选择,他必须先猜测出最有可能发生的变化种类,然后构造抽象来进行隔离.

结论:OCP是面向对象设计的核心所在,遵循这个原则可以带来面向对象技术所声称的巨大好处:灵活性,可重用性以及可维护性.对于应用程序中的每个部分都肆意地进行抽象同样不是一个好主意.正确的做法是:设计人员应该仅仅对程序中呈现出频繁变化的那些部分做抽象.

## LSP原则

这是评判是否为OO的标准之一.也是代码中评判是否可以使用继承的标准之一.因为继承会打破封装性.并且父类的改变所有的子类都会受影响.所以继承的关系链一定不可以过长.

LSP的含义是:所有出现父类的地方都可以使用其子类来代替.如果满足就可以使用继承.如果不满足就不推荐用.推荐使用组合来代替继承.

Class A{

B b;

b.属性

b.方法

}

## DIP法则

依赖倒置原则,简单说就是面向抽象,不要依赖实现.在编码时多使用多态.即用父类或者接口为类型来创建子类型对象.

 DIP的两个主要原则:

(1):高层模块不应该依赖于低层模块,二者都应该依赖于抽象;

(2):抽象不应该依赖于细节,细节应该依赖于抽象;

Booch曾经说过:"所有结构良好的面向对象构架都具有清晰的层次定义,每个层次通过一个定义良好的,受控的接口向外提供了一组内聚的服务."其实每个较高层次都为它所需要的服务声明一个抽象接口,较低的层次实现这些抽象接口.每个高层类都通过该抽象接口使用下一层,这样高层就不依赖于低层,低层反而依赖于高层中声明的抽象服务接口.

这里的倒置不仅仅是依赖关系的倒置,它也是接口所有权的倒置,我们通常会认为工具库拥有它们自己的接口.但是当应用了DIP时,我们发现往往是客户拥有抽象接口,而它们的服务者则从这些抽象接口派生.这也就是著名的Hollywood原则:"Don't call us,we'll call you.(不要调用我们,我们会调用你)".

依赖于抽象,这是一个简单的陈述,该启发规则建议不应该依赖于具体类,也就是说,程序中所有的依赖关系都应该终止于抽象或者接口.它提醒我们应该这样来编码:a,任何变量都不该持有一个指向具体类的引用;b,任何类都不该从具体类派生;c,任何方法都不该重写它的任何基类中已经实现了的方法.

## ISP原则

接口隔离原则.

胖类会导致它们的客户程序之间产生不正常的并且有害的耦合关系,当一个客户程序要求该胖类进行一个改动时,会影响到所有其它的客户程序.因此,客户程序应该仅仅依赖于它们实际调用的方法.通过把胖类的接口分解为多个特定于客户程序的接口,可以实现这个目标.每个特定于客户程序的接口仅仅声明它的特定客户或客户组调用的那些函数,接着,该胖类就可以继承所有特定于客户程序的接口,并实现它们.这就解除了客户程序和它们没有调用的方法间的依赖关系,并使客户程序间互不依赖.

# 泛型

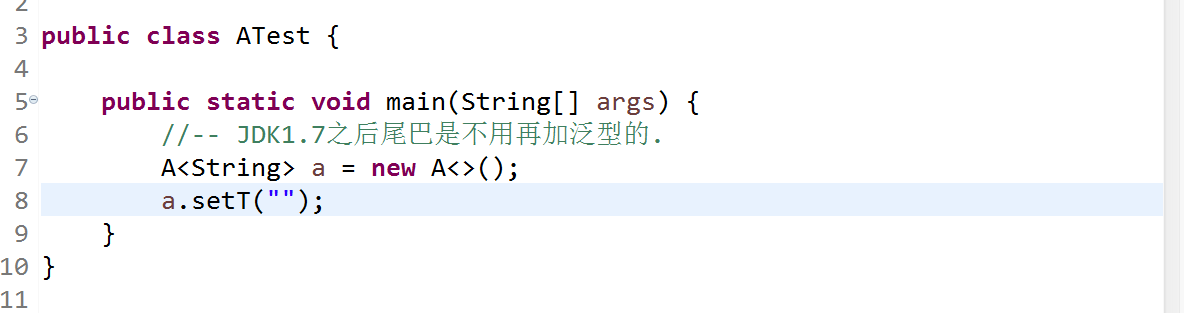
泛型的本质就是类型参数化.在JDK1.5之前是没有泛型的.如果想要在容器中保存数据.无论放入之前是什么类型.放入后都是Object类型.放入是没有什么大问题的.但是取出会有严重问题.

会有很大的概率导致类型转换异常.使用instanceof关键字进行类型验证是可以避免类型转换错误,但是验证是很消耗资源的.

为了解决这个问题.在JDK1.5的时候借鉴了C++中模板引入了泛型机制.在Java中使用<>来表示泛型.

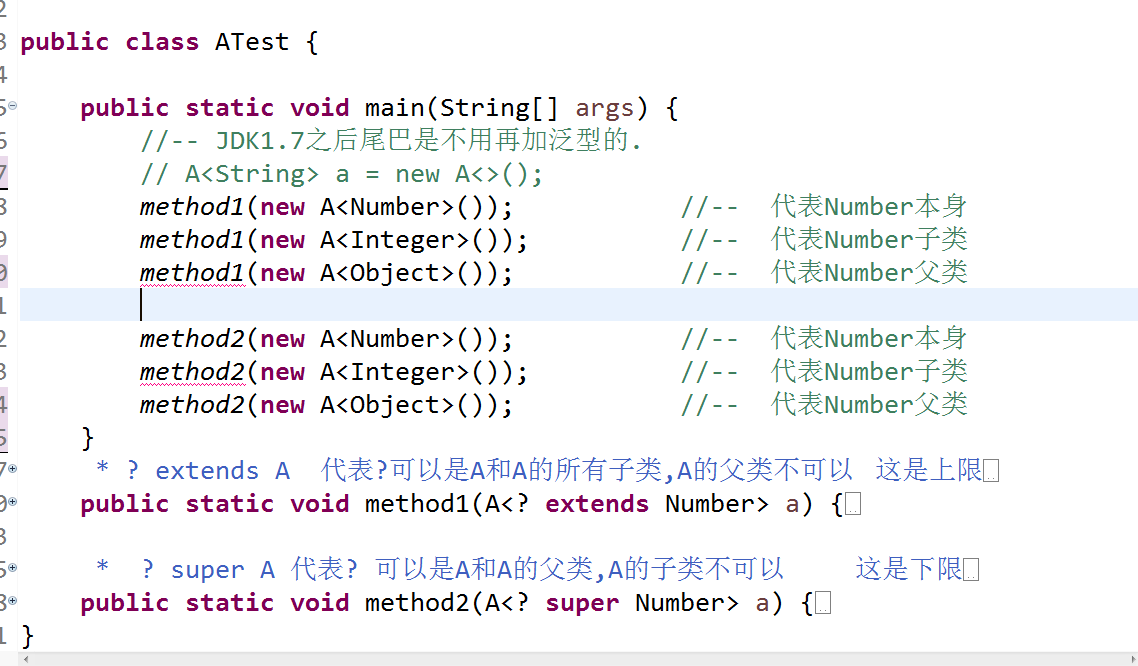
泛型可以引用在变量.方法和类上.对应的我们把它叫做泛型类.泛型变量和泛型方法.





## 泛型的上限和下限

在使用泛型时.可以配合关键字:super或者extends.比如A<? super String> 和 A<? extends String>



## 泛型的可擦除性

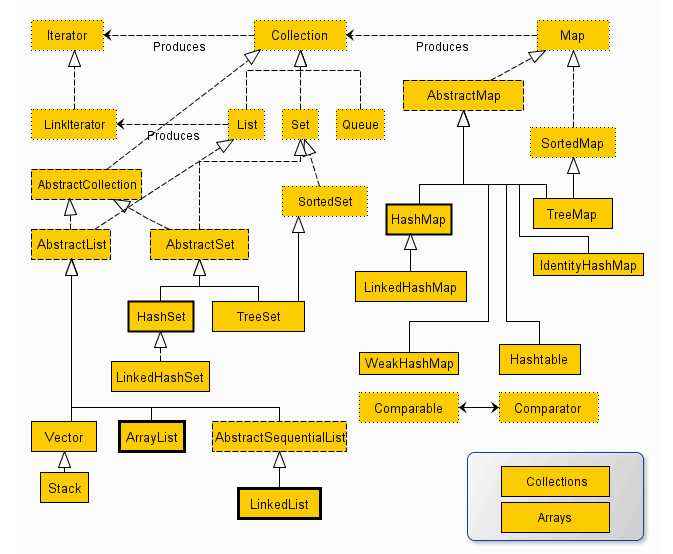
如果父类有泛型.在子类继承父类时,父类的泛型可能会丢失.所以不要使用泛型类.

# 集合

JavaSE在面向对象之后一共有5大块内容

1. 集合
2. 线程
3. IO
4. 网络
5. GUI(过时了.Swing/AWT)

## Java的集合体系图



## Collection

作为线性结构的顶层接口在Collection中定义了很多方法.所有Collectiond的子类或者实现类.都会拥有其中声明过的方法.

### ArrayList

叫做动态数组.是完全基于数组扩展来的一种集合.数组所具备的所有特性它都拥有.

### LinkedList

# 家庭作业

1. 使用ArrayList.重写双色球
   1. 100期
      1. 红球 1-33
      2. 蓝球 1-16
   2. 每一期红球不重复
   3. 统计一百期每个号码的出现次数
   4. 按照次数排序

# 参考

1. [类关系介绍](https://www.cnblogs.com/shindo/p/5579191.html)
2. [类关系介绍](https://blog.csdn.net/pj36536/article/details/53101764)
3. [数据结构](https://blog.csdn.net/qq_31196849/article/details/78529724)