# Java课程笔记

## Java数据类型表（两大类）

### 1.1基本数据类型：

数值型：表示整数或小数

整型：byte、short、int、long; -> 默认值 0 ;c\c++整型没有默认值，需要初始化默认值

浮点型：float、double; ->默认值0.0

字符型：char; ->默认值’\u0000’

布尔型：boolean。 ->默认值：false

### 引用数据类型：数组、类、接口 默认值 ： null

选用原则：

整数用int ，小数用double

Long用于描述日期时间，内存或者文件大小

如果需要进行编码转换或者二进制数据传输用byte -128~127

Char基本在描述中文时使用（基本会忽略）

Boolean在描述程序逻辑时使用

## 1.2整型：(java所有整数常量都是默认int型)

在进行整型数据操作的过程之中也会存在数据溢出问题，指的是当已经到达了整型的最大或最小值，如果继续进行数学运算，而产生的错误而数据

关于变量的命名要求：第一个单词的首字母小写，后面每个单词首字母大写（驼峰法）

观察数据溢出：

发现超过数据局限后，会出现偏差，一定要预估数据大小。

解决溢出问题的方案：更换更大的数据类型

所有程序的运算顺序都是从”=”右侧开始；

方式一：将int数据变量赋值给long

范围小的数据类型能自动转变成大的数据类型，范围大转变成范围小的数据类型需要强制转换（自己不要使用）

整型变量未初始化时，报错。

两种解决方案：输出前定义值或者声明时初始化变量值。

Byte-128~127,如果说赋值范围大于这范围，将会报错

## 1.3浮点数：默认使用double

如果要声明float在声明时初始化时在数字后面加f或者在之前强制转换

## 1.4字符型：char

ASCII表 ‘A’=65 ‘a’=97 ‘0’ =48 ‘9’ = 57

但是java使用Unicode，包含ASCII码

## 1.5布尔型：true false

Java不支持0，1表示布尔型

## 1.6String类型

之中的“+”是连接的意思，当运算中出现了String所有类型变成String与String连接，可以用括号

## 1.7转义字符：对一些字符串进行操作

\n

## 2.0Java运算符

自增预算 ++a:先自增后运算；a++先运算后加加，和其他运算无关

## 2.1三目运算（重点）

## int a1 = b>a?b:a;//不会用，使用if判断

## 2.2关系运算符

> ,< , >= ,<= ,!= ,== .

可以直接将字符符和数字进行判断

## 2.3逻辑运算符：与、或、非 &&（&）、||（|）、！

在逻辑运算中，与和或比较麻烦，因为有两种写法

使用（&）时，多个条件都要判断

使用（&&）时，只要有个条件false,后续条件不再执行

## 2.4位运算

关键问题时十进制和二进制的转换

位运算实现>>,<<

Int a = 023;//八进制

Int a=0x23;//十六进制

## JAVA 类与对象的使用

面对对象三大特征：封装，集成，多态

### 3.1

类声明后是不能直接去使用的，如果要使用类，那必须产生对象，而对象的定义：

1. 声明并实例化对象 类名称 对象名称 = new 类名称（）；
2. 分布式进行对象实例化
   * + 1. 声明对象：类名称 对象名称 = null
       2. 实例化对象：对象名称 =new 类名称（）；

New代表开辟内存

所有对象只有实例化之后才能真正使用：

调用类中属性 对象.属性

调用类中方法 对象.方法名称

### 3.2

对象的产生分析。

引用类型指的是内存空间的操作。而对于现在的内存，主要使用两块内存空间：

1. 堆内存空间：保存真正的数据；
2. 栈内存空间：保存的堆内存的地址，堆内存操作权；

所有的引用数据类型，必须在其开辟空间后使用。

## 3.3引用传递分析

引用的本质是别名，这个别名放在了栈内存，一块堆内存可以被多块栈内存指向

public class User {

public static void main(String args[]) {

person a = new person();

a.info();

person p1 =new person();

person p2 = new person();

p1.name = "blue";

p1.age = 19;

p2 = p1;

p2.name = "yellow";

System.out.println("年龄是"+p1.age+"姓名是"+"\n"+p1.name);

System.out.println("年龄是"+a.age+"姓名是"+"\n"+a.name);

}

}

class person{

String name = "huangyiming";

int age = 22;

public void info() {

System.out.println("年龄是"+age+"姓名是"+"\n"+name);

}

}

结果为：（实际概念类似c指针的作用，记录了相同的堆地址，对相同的内存空间进行操作）

年龄是22姓名是

huangyiming

年龄是19姓名是

yellow

年龄是22姓名是

huangyiming

## 3.4private实现封装：只允许本类访问不允许外部访问

类的属性通过private前缀进行封装，外部无法直接进行操作，想要进行属性的访问使用setter\getter

1. setter主要用于属性的内容的设置

Private String name：public void setName(String n)

1. getter方法主要用于属性内容的取得

Private String name:public String getName()

类中属性都需使用private封装

## 3.5构造方法和匿名对象

实例化对象为例：

* + 1. 类名称 每个对象都有其对应的类，不知道类就不知道对象具备的功能
    2. 对象名称 是一个唯一的标记，表示的是操作的属性的标记
    3. New 开辟新的堆内存空间
    4. 类名称（）构造方法

所谓的构造方法就是在使用关键字new来实例化对象时调用的操作方法，但对构造方法的定义必须遵从以下原则 ：

1. 构造方法名称必须与类名称相同，且没有返回值声明
2. 一个类必须至少有一个构造方法，如果没有声明将自动生成一个无参的什么都不做的构造方法

构造方法在new开辟新内存时被执行

Q&A ：

Q:既然没有返回值，构造方法为什么不用void声明？

A:现在的类中的组成：属性、普通方法、构造方法

1. 属性是在对象开辟堆内存就开辟的空间
2. 构造方法是在使用关键字new时同时使用的，使用一次//出生
3. 普通方法是在对象已经实例化完成后使用的，使用多次//吃饭

普通方法和类名相同语法可行的但不是好习惯。

对于类自动生成的无参构造方法，前提是你没有定义任何构造方法。如果定义了一个有参构造不会自动生成无参构造方法，并且会报错，可以通过有参构造方法来传递参数。

**构造方法的调用和对象内存的分配几乎是同时完成的，构造方法可以为类中属性进行初始化处理。**

**通过构造方法进行处理，可以避免重复的setter方法调用**

**在实际开发中，setter还承担了修改数据的作用**

构造方法能够重载，因为方法名称，所以只能实现参数的属性和个数的区别

写构造方法时，应当按照参数的个数进行排序

类定义时应当按照如下顺序：

1. 属性
2. 构造方法
3. 普通方法

### 匿名对象：

Person per = new person(“张三”，20)；

new person(“张三”，20)//匿名对象,在只调用一次的类方法时使用

由于匿名对象不会有任何栈空间指向，会变成垃圾

## Java实例1：简单java类

假设有这样一个要求，定义一个雇员类，该类包含雇员编号，姓名，职位、基本工资、佣金几个属性信息。

那么这种就是简单java类

要求：

1. 类的名称要有意义，明确描述事物
2. 类中所有属性必须使用private封装，提供setter,getter 方法
3. 可以定义有参构造方法，但必须保留一个无参构造方法
4. 类中的所有方法不能出现System.Out，输出交给调用处完成

5、类中应该提供一个返回类完整信息的方法，暂名为getinfo

### 简单java类补遗：

1.一个java文件可以包含多个java类，但是只能包含一个public类，并且public类的类名必须与java文件名相同。

2.如果一个java文件只写一个类，所以看类名就知道这个java文件里面的类了。

3.一个文件多个类和一个文件一个类的效果是一样的，同样不能访问其它类的private方法。

## 4.1数组 ：属于引用类型（动态初始化）

一系列相关类型的变量集合，而且这些变量可以按照统一的方式进行操作。数组属于引用数据类型，所以会牵扯到内存分配。

定义语法：

数组的动态初始化:

|-声明并开辟数组

数据类型 [] 数组名称 = new 数据类型[]

|-分步进行数组空间开辟

1. 声明数组 数据类型 数组名称[] = null;
2. 开辟数组空间 数组名称 = new 数据类型[长度]

数组访问从索引进行 从0开始；范围是0~数组长度-1；

Java中数组属于引用数据类型，一定要提前开辟内存空间，和对象开辟空间一样。

## 4.2数组引用传递（java只有引用传递没有指针概念）

数组存放的是地址，通过地址对堆内存进行操作，（有点像指针，但不是）

## 4.3数组的静态初始化

在之前的数组定义，有明显特点：先开辟内存空间，都叫做动态初始化

而在定义的时候同时设置内容，叫做静态初始化。

**静态初始化：（推荐完整格式）**

简化格式：数据类型 数组名称[]={值，值，，，，}

完整格式：数据类型 数组名称[] = new 数据类型[] {值，值，，，，}

System.out.print(new int[] {1,2,3,41,2,314,123,4,1,243,22}.length)

**数组最大的缺点：长度固定**

## 4.4二维数组

需要两个索引进行访问的数组

## 4.5数组与方法调用

数组是引用数据类型，那么所有的引用数据类型都可以为其设置多个占内存指向，(同一个数组由多个不同的数组名指向)，所以在进行数组操作，也可以将其通过方法进行处理。

数组和方法发生引用传递，所以方法是可以直接修改数组的（修改堆内存中的数组数据）

## 4.6java对数组的支持

Java的类库中存在数组的支持方法

1. 数组排序 java.util.Arrays.sort(数组名称)//基本类型数组，默认升序
2. 数组拷贝：将一个数组的部分内容替换掉另外一个数组的部分内容：System.arraycopy(源宿主名称，源数组开始点，目标数组名称，目标数组开始点，拷贝长度)

## 4.7数组的定义与使用

### （案例1：数组数据统计）

### （案例2：数组排序）冒泡排序（和c语言逻辑共通）

### （案例3：数组转置）（首尾交换）

1. 开辟新数组，将原数组倒序保存，再赋值
2. 在一个数组上完成 p[i]\p[p.length-i]转换//保证中间轴不动

主方法不需要涉及到过于复杂的程序逻辑，只需要关注结果

### （案例4：数组二分法查找）//数据结构课程内容

重要前提：数组已经排序

## 4.8对象数组（核心）(对象数组和结构体数组有点像)

对象也可以将其定义为一个数组，这种就叫对象数组，往往是由引用类型数据为主的定义

对象数组动态初始化：默认值为null

类名称 对象数组名称[] = new 类名称 [长度]

对象数组静态初始化

类名称 对象数组名称[] = new 类名称 []{实例化对象}

对象数组存储的数据更多，所以应用很多

## 5.1 String类的基本特点

项目开发一定存在string类，但string类本身一定有差别

## 5.1.2string类的两种实例化方法

1、String可以直接赋值和基本数据类型类似

2、但是，String本身是一个类，类中则一定有一个构造方法，String类恰好提供了一个构造方法：

Public String （string str）；

也可以使用String 变量名 = new String(“实例化”)；

## 5.1.3字符串比较

String a1 = **new** String("hello");//指向堆中一个新的空间，里面是hello

String a3 = “hello”; //指向堆中常量字符串hello

System.***out***.println(a1==a3);

使用“==”比较时为false

“==”本身是进行数值的比较的，如果用在对象的比较重，实际上比较的是两个对象的地址的数组而不是内容

System.***out***.println(a1.equals(a3));

内容比较使用public boolean equals(string str);//实际上（）内的数据类型是object类型所有对象

## 5.1.4字符串匿名对象

在任何语言的底层上面都不会提供直接的string类型，现在java的string类是高级语言提供给用户开发的支持。所以java本身没有提供字符常量的概念，所以“”的内容实际上是string的匿名对象

所以

String string = "hello";

System.***out***.println("hello".equals(a1));

所以之前的string string本质上是给匿名对象“hello”赋予名字。

**Tips：**

**日后开发中，进行string比较时一定要把用户输入的字符串放在equals（）中因为要考虑用户输入null的情况。**

## 5.1.5string两种实例化方法的区别（chrome收藏夹相关链接）//一般都是使用直接赋值的方法

实际开发中，那种方式更好？

1. 直接赋值

此时多个string如果他们初始化值相同，实际上他们指向的是同一个地址

**实际上string类的设计使用了一个共享设计模式：**

在**jvm**的底层实际上维护了一个对象池（字符串对象池），如果采用直接赋值实现了string类实例化操作，那么该对象会保存在对象池中，下次有人继续有人使用了直接赋值的模式声明了string类的一个对象，如果对象池中有对象，那么直接引用，如果没有，那么开辟新的对象以便下次使用。

**对象池本身是一个对象数组**

1. 采用构造方法

类对象采用构造方法进行实例化才属于标准做法。

但实际上此时即便不同对象名拥有相同的实例化值，他们也是指向不同的内存空间而且并没有指向对象池中的对象,而且不会自动留在对象池之中必须通过下列方法

## 5.1.6字符串常量无法改变

所有语言对字符串的底层实现都是字符数组，数组的最大缺陷就是长度固定，所以在定义字符串常量时，内容无法改变

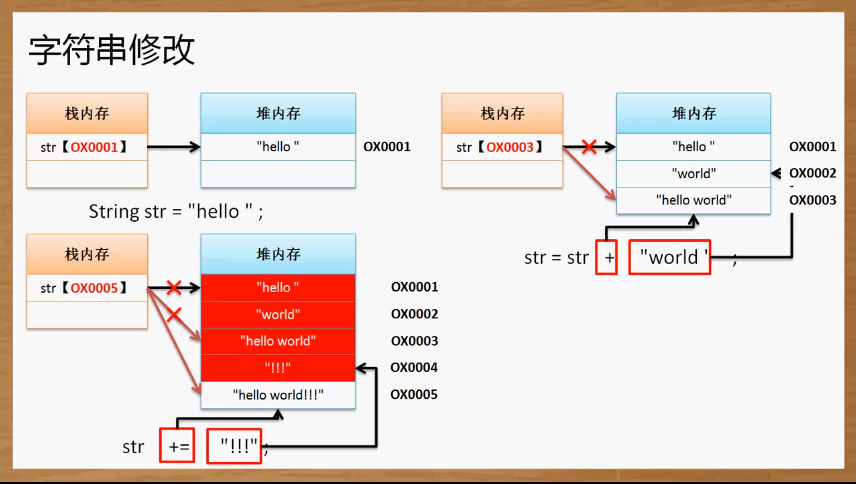
String str2 = "hello";

str2 += " world";

str2 += " !!!";

System.***out***.println(str2);

“world” 字符串在+时与str2指向的字符串“hello”进行组合在内存中new了一个新的空间变为“hello world”，然后str2改变指向只向为“hello world”。

“ world”就是string匿名对象 如图所示：

**所以“hello ”本身没有改变，并且产生大量的内存垃圾，类似代码不应该出现。**

## 5.1总结

1. 字符串使用直接赋值的模式完成
2. 字符串的比较使用equals（）
3. 字符串不要轻易改变

## 5.2string类的常用方法

String类在开发中比用，并且基本概念必然不够用，所以必须熟练掌握string类的方法

## 5.2.1DOC文档组成

1. 任何一个类的文档有如下几个部分组成
   1. 类的相关定义，包括这个类的名字，有哪些父类，哪些接口
   2. 类的相关简介基本使用
   3. 成员（field）摘要：属性就是其中一种，会列出所有出现的成员的信息向、
   4. 构造方法（constructor）说明： 列出所有的构造方法说明
   5. 方法摘要（method）：方法构造，参数和成员信息：左边一列返回值右边名称和参数

## 5.2.2字符串和字符数组

字符串就是一个字符数组，所以string类中就有支持字符串和字符数组的相互转换

char [] str11 = string.toCharArray();

String a = str11.toString();

字符串和字符数组的转换

## 5.2.3字节和字符串

字节byte不适合处理中文-128 ~ 127

字节只适合处理二进制文件

## 5.2.4字符串比较

之前使用过equals(),该方法可以实现区分大小写的比较操作，string类还有实现比较的方法

|  |  |  |
| --- | --- | --- |
| 1 | Public boolean equals (string object) | 区分大小写 |
| 2 | Public boolean equalsIgdnoreCase(string object) | 不区分大小写 |
| 3 | Public int compareTo(string anotherstring) |  |

在string中compareTo()方法(支持中文)是一个最为重要的操作方法，该方法会根据大小返回三种不同的整型值：是唯一一个区分大小关系的方法

1. 相等：返回0
2. 小于：<0
3. 大于：>0

## 5.2.5字符串查找

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | Public boolean contains(string str) | 判断一个子字符串是否存在 |  |
| 2 | Public int indexOf(string str) | 从头查找制定字符串的位置，查到了返回开始索引，查不到返回-1 | 只返回第一个 |
| 3 | Public int indexOf(string str,int fromIndex) | 从指定位置开始查找 |  |
| 4 | Public int lastIndexOf(string str) | 从后向前开始查找 |  |
| 5 | Public int lastIndexOf(string str,int fromIndex) | 从指定位置开始从后向前查找 |  |
| 6 | public boolean startsWith(String prefix) | 是否以指定字符串开头 |  |
| 7 | public boolean startsWith(String prefix,int toffset) | 从指定位置开始判断是否以指定位置字符串开始 |  |
| 8 | public boolean endsWith(String suffix) | 是否以指定字符串结尾 |  |

## 5.2.6字符串替换

使用一个新的字符串替换已有字符串的数据

|  |  |  |
| --- | --- | --- |
| 1 | public String replaceAll(String regex,String replacement) | 替换所有指定内容 |
| 2 | public String replaceFirst(String regex,String replacement) | 替换首个指定内容 |

替换操作与正则有关

## 5.2.7字符串拆分

|  |  |  |
| --- | --- | --- |
| 1 | public String[] split(String regex) | 完全拆分 |
| 2 | public String[] split(String regex, int limit) | 将字符串部分拆分limit是字符串数组长度 |

Str.split(“ ”)//按空格拆分

## 5.2.8字符串截取

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | public String substring(int beginIndex) | 普通 | 由指定索引截取到结尾 |
| 2 | public String substring(int beginIndex, int endIndex) | 普通 | 截取指定长度 |

## 5.2.9字符串的其他方法

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | public String trim() | 普通 | 去掉左右空格，中间空格表格 |
| 2 | public String toUpperCase() | 普通 | 字符串转大写 |
| 3 | public String toLowerCase() | 普通 | 字符串传小写 |
| 4 | public String intern() |  | 字符串入池 |
| 5 | public String concat(String str) |  | 字符串连接 |
| 6 | Public int length(string str) |  | 求长度 |
| 7 | Boolean isEmpty(string str) |  | 判断长度是否为0，但不是null |

## 6.1 this

1. 调用本类属性

类中，当参数和属性名相同时，在属性名之前加上this.表示这是当前类的属性名，只要在类中的方法之中访问类的属性，属性名之前一定要加this

**private** **int** emId = 0;

**private** String emname;

public void setemId(int emId) {

This.emId = emId;

}

public void setemname(String emname) {

This.emname = emname;

}

1. 调用本类方法（普通、构造）

普通方法：//不是必须，但是是好习惯

This.方法名称（参数）

构造方法：

This(参数)

范例：

构造方法与普通方法最大的区别：使用关键字new实例化类对象时使用一次，而普通方法是在对象实例化完成了（构造方法已经执行过了）之后可以调用多次

程序中出现重复的代码，必须消除。

此时使用this解决问题：

public emp() {

System.out.println("这时一个新的emp对象出现了");

}

public emp(int emId) {

this();

this.emId = emId;

}

public emp(int eno,String name,String job,double sa,double com) {

this();

this.setemId(eno);

setemname(name);

seteJob(job);

setSal(sa);

setComm(com);

}

虽然使用this可以实现构造方法的互相调用

This()调用构造方法的语句必须放在构造方法的首行

使用this调用构造方法时请留有出口（别形成死循环）

1. this表示当前对象（最难）

一个类会产生若干个对象，那么程序类在分辨的时候不会记得有多少个对象，他只记得当前操作类的对象是哪一个。

范例：person p1 = new person();

System.out.println("【main方法】"+p1);//对象地址

p1.fun();//两者输出一样

public void fun() {

System.out.println("【fun方法】"+this);

}

This没变代指现在正在使用类的对象

## 7.1 JAVA引用传递

Java的核心内容，使用三个简单程序进行引用传递分析。

**1：最简单：**

class Message{

private int num ;

public void setNum(int num) {

this.num = num;

}

public int getNum() {

return this.num;

}

}

public class TestDemo {

public static void main(String args[]) {

Message msg = new Message();

msg.setNum(100);

fun(msg);// == message temp = msg;

System.out.println(msg.getNum());

}

public static void fun(Message temp) {

temp.setNum(30);//temp被释放，结果被保留

}

}//输出是30

**2：//关键是因为string指向一个字符常量**

public class TestDemo {

public static void main(String args[]) {

String str = “hello”;

fun(msg);

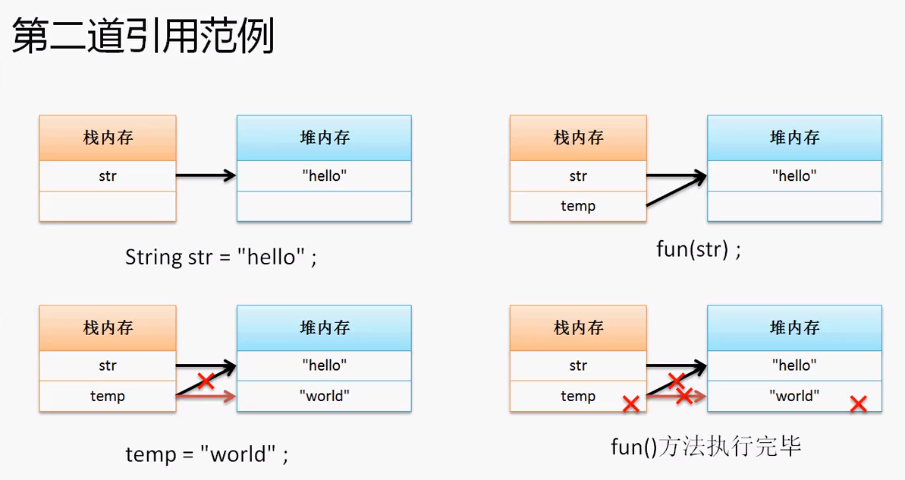
System.out.println(str);

}

public static void fun(String temp) {

Temp = “world”;

}

}//输出hello

**3：前提：**

class Message{

private String note;

public String getNote() {

return note;

}

public void setNote(String note) {

this.note = note;

}

}

public class TestDemo {

public static void main(String args[]) {

Message msg = new Message();//开辟了新空间

msg.setNote("hello");

fun(msg);

System.out.println(msg.getNote());

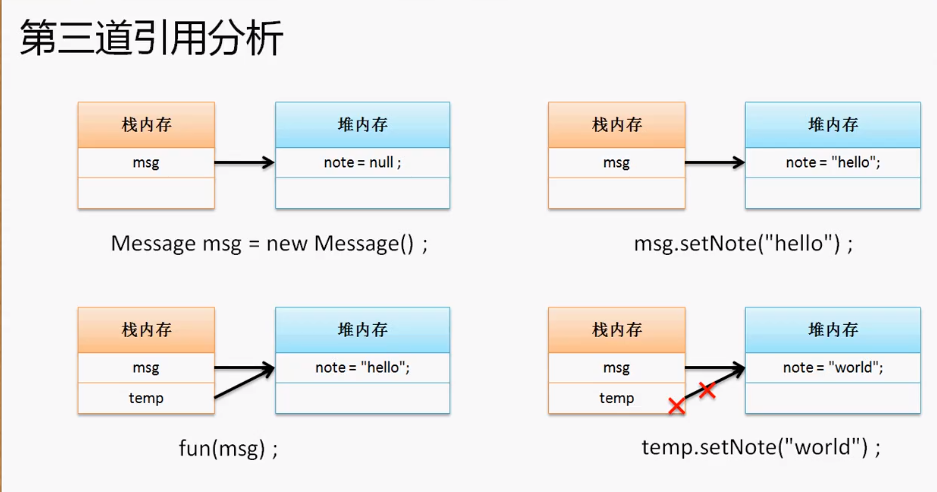
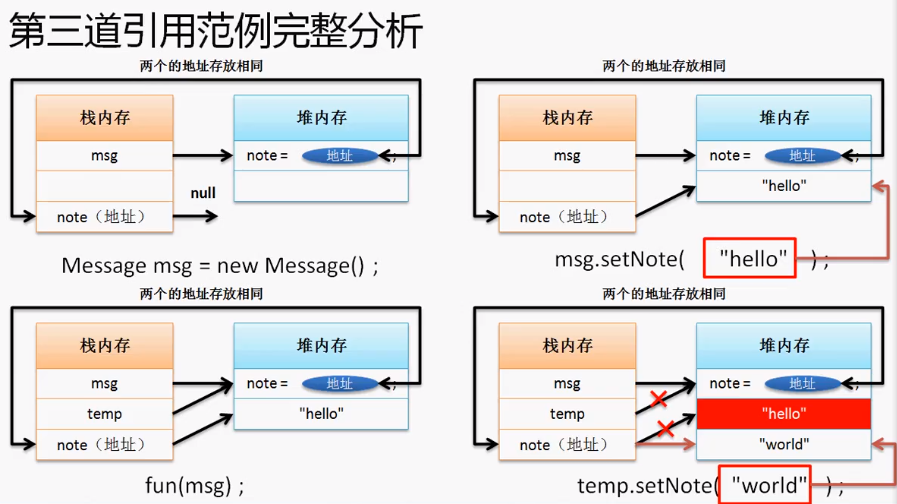
}

public static void fun(Message temp) {

temp.setNote("world");

}

把string当做基本数据类型

String是引用数据类型

总结 ：最简单的方法是把字符串当做基本数据类型

## 7.2引用传递实际应用

引用传递能把更好的表现出现实世界的抽象

**例子：表现一个人有或者没有车**

进一步设计人的后代，后代有车

两个类：一个人（member）、一个（车）、车能找人，人能找车

后代公用member类

注意，单个引用是单向的，甲能引用乙，乙不一定能引用甲

可得出如下设计：MemberCar java程序

**例子2：描述电脑组成**

只有将这些细小的类合并到一起才能描述具体的概念，而且这些细小的类都可以替换，所以这种设计叫做合成设计

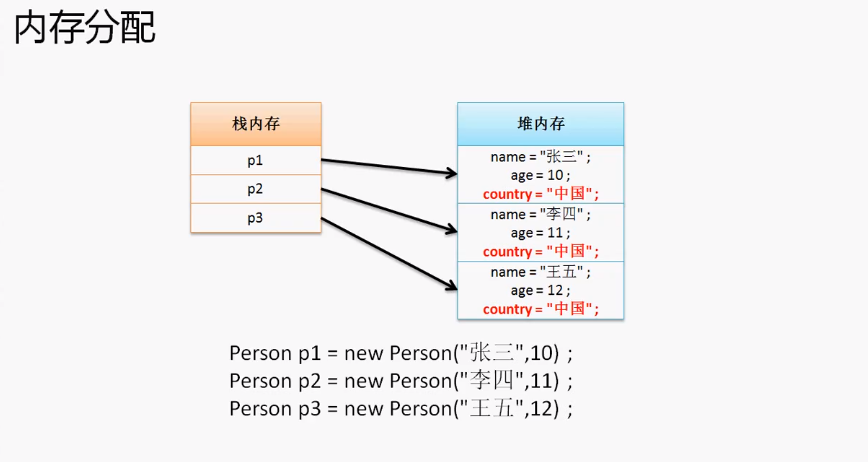
总结：以后的项目没开发中，所有的类必须自定义

## Static关键字

所有定义的方法以及属性上实际上都可以使用Static关键字进行定义。

## 8.1static属性

构建一个person类，属性是name,age,country. 其中country都是中国



描述的都是中国人所以country会重复保存，如果要改变这个属性值需要改变，此时person类如果产生了大量的对象，会需要注意改变

会发现传统属性所具备的特点就是保存在堆内存中，每个对象独享此属性，所以此时不适合使用传统属性，需要将该属性改成共享属性，只要一个对象改变属性，所有对象的该属性都会改变，方法就是在属性前加上static关键字。

Static修饰的属性不会保存在堆内存中，而是会保存在一个全局数据区的内存空间中，并且所有对象都可以对此数据区进行访问

但是既然使用的是共享属性，代码此时会出现一个问题，共享属性能通过一个对象修改吗？

所以所有static属性都称为类属性，所有类属性都可以通过类名称直接修改

结论：所有static属性都使用类名称进行访问

所有非static属性都要通过实例化对象进行访问，static属性和类是否实例化无关

选择：是否选择static属性

定义类中绝大多数情况下不考虑static属性，如果要考虑到共享属性或者不想被实例化对象调用时使用static属性

## 8.2 static方法

使用static定义的方法也可以通过类名称直接访问不受到实例化对象的影响

类中存在static方法和非static方法，会存在要求：

1. 所有static方法无法使用非static方法或属性
2. 非static方法允许调用static方法

语言：static方法可以在没有实例化对象的情况下使用，而非static方法不能在没有实例化对象的情况下使用。

使用static方法的原因：不希望方法受到实例化对象的限制

## 8.3 主方法组成

之前说过：一个方法定义在主类之中并且受到主类调用，该方法形式：

Public static 返回值类型 方法名称

因为主方法本身是static修饰所以根据static方法无法使用非static方法或属性，主类的其它方法必须加static

非静态方法必须用对象调用

组成意义：

Public：表示公共的，主方法作为起点必须能够随意访问

Static：执行java程序执行类名称，表示不受实例化对象限制

Void：主方法是一切的起点

Main：是系统定义的一个方法名称

String args[] 表示该类执行时需要的参数，java Demo执行时在后面加语句就是该参数

## 8.4Static应用

Static的最大功能是进行共享的操作，所以在这基础上static可以做一个对象产生的技术统计，所有新对象的产生一定要通过构造方法，所以在构造方法中实现计数的增加

那么也可以在以上的程序基础上做个简单的扩充，假设说，现在对于一个无参构造函数希望他可以给内部变量赋予默认值，可以通过static属性赋值，自动为属性附上不同的默认值

1、Static属性和方法不是类设计的首要选择

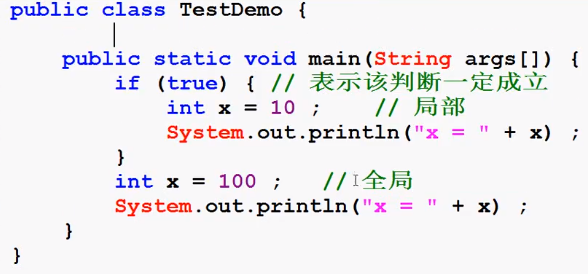
2、static属性和方法不受到类的实例化对象限制，可以通过类名称直接调用

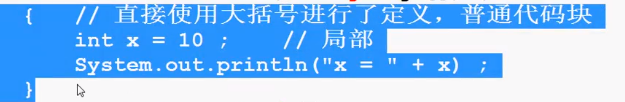
## 代码块

## 9.1普通代码块

指的是定义在方法中的代码块

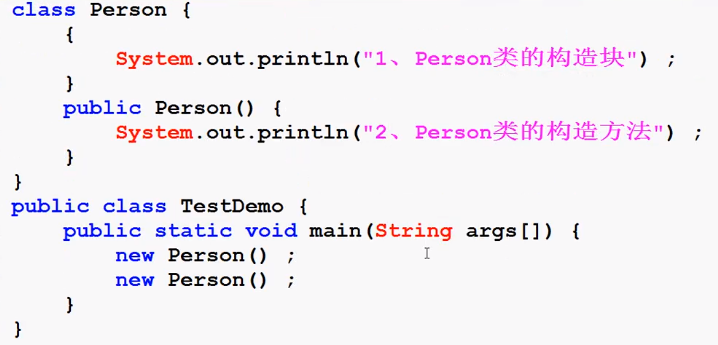
范例：



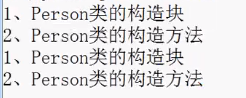
如果将if语句拿掉，此时就是简单代码块。

如果代码写的过长，又要避免变量重名问题，会使用简单代码块

## 9.2、构造块

指定义在类中的代码块。

执行结果：



使用关键字new实例化对象时一定会调用构造方法，但是有了构造块后会发现构造块优先于构造方法执行，多个构造块按顺序执行

构造块可以进行简单的逻辑操作。

## 9.3静态代码块

指的是使用了static定义的代码块，分为两种：

1. 在非主类中定义的静态块
2. 在主类中定义的静态块

总结：

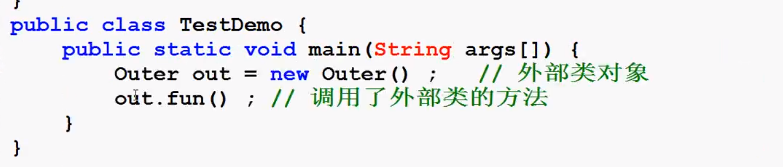
如果需要在一些属性使用做处理就使用构造块或者静态代码块

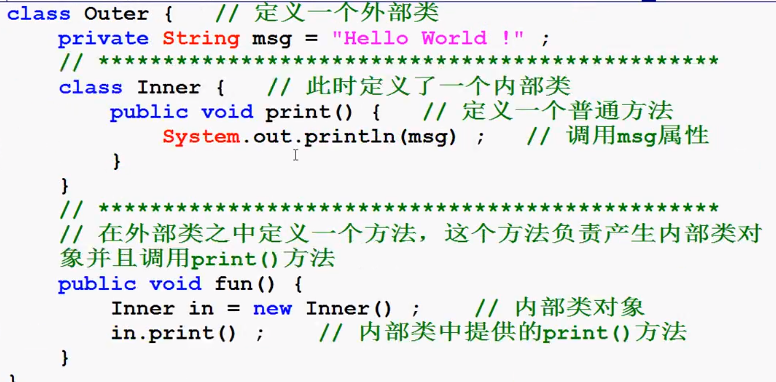
## 10、内部类的定义及使用

暂时不作为首要的设计原则

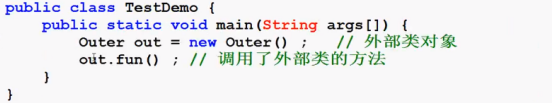
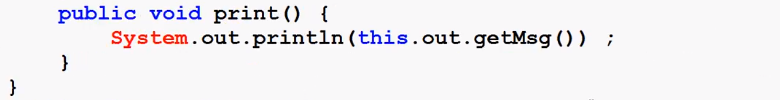
## 10.1内部类的基本概念

内部类指的是一个类内部进行其它类结构嵌套的语法操作形式。理论上一层层进行类的嵌套。但是如果嵌套的结构太多，会让代码复杂。



程序结构开始混乱，虽然破坏了结构，但是实现了外部类的私有访问，如果把类提取的到外部实现相同功能



上述代码实现了访问外部类的私有属性。

Outer和inner 是两个独立的类，而且，msg已经采用了private封装，所以外部类想要访问msg必须使用getmsg方法

在inner中的print方法中，想要访问msg必须使用getmsg方法，但是getmsg是个普通方法必须使用outer对象进行访问，此时需要将主方法中的out对象传递进inner

对于内部类的操作远远不止于此

1. 从上方代码，当前内部类的访问必须通过外部类的方法完成，如果不想通过外部类方法而是想在程序外部类调用必须按如下形式进行内部类实例化对象：

（1）外部类.内部类 内部类对象 = new 外部类().new 内部类();//必须实例化外部类才能实例化内部类

2、如果一个内部类只想被外部类使用不想产生实例化对象，用private修饰

3、在进行属性访问时习惯性加上this如果需要在内部类中使用this形式应该变为外部类.this.属性

## 10.2static定义内部类

内部类如果使用了static定义就是表示其就是一个外部类的形式，但是这个外部类的名称是“外部类.内部类”，该内部类只能访问外部类中的static

class Outer{

private static String msg = "helloworld";

static class Inner{

public void print(){

System.out.println(msg);

}

}

}

public class DemoMain{

public static void main(String args[]){

Outer.Inner in = new Outer.Inner();

in.print();

}

}

## 10.3在方法中定义类

理论上内部类可以定义在类内部的任何地方包括：方法中，类中，代码块中。

从实用角度讲，在方法中定义内部类形式是最多的。

class Outer{

private String msg = "hello world";

public void fun(final int num) {

class Inner{

public void print() {

System.out.println("num = "+num);

System.out.println("msg = "+msg);

}

}

new Inner().print();

}

}

public class TestDemo {

public static void main(String[] args) {

// TODO Auto-generated method stub

Outer outer = new Outer();

outer.fun(100);

}

}

此时的代码在jdk1.8之前是错误的。

在jdk1.8之前，如果一个方法内内部类想要访问方法的参数，那么这个参数一定要用final修饰，在1.8之后为了推广函数式编程，将final取消了。

总结：内部类的使用暂时不作为你设计的首选至少应该知道内部类的特点：

1. 破坏了程序的结构
2. 方便的进行私有属性的访问
3. 如果类名称出现了“.” ，应该想到是内部类

# 继承的定义与使用

面向对象的第二大特点就是继承，继承的一大特点就是在已有的基础上进行功能的扩充

## 11.1、 继承问题的引出

解释继承的作用：

定义两个类 //人类// 学生类//

class person{

private String name;

private int age;

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return this.name;

}

public int getAge() {

return this.age;

}

}

class Student{

private String name;

private int age;

private String school;

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public void setSchool(String school) {

this.school = school;

}

public String getName() {

return this.name;

}

public int getAge() {

return this.age;

}

public String getSchool() {

return this.school;

}

}

出现了重复的代码，又因为学生类相较于人类更为具体，具备的属性和方法也会更多。

此时想要消除结构定义上的重复，就需要使用继承

## 11.2、继承的实现

java使用extends关键字来进行实现继承：

继承关系 ：class 子类 extends 父类

子类又被称为派生类

父类又被称为超类

继承的基本实现：

class person{

private String name;

private int age;

public void setName(String name) {

this.name = name;

}

public void setAge(int age) {

this.age = age;

}

public String getName() {

return this.name;

}

public int getAge() {

return this.age;

}

}

class Student extends person{

private String school;

public void setSchool(String school) {

this.school = school;

}

public String getSchool() {

return this.school;

}

}

当发生了类继承之后，子类可以实现父类的操作，可以实现代码的重用，子类最低也能维持和父类相同的功能，子类也可以经行功能的扩充包括方法和属性

## 11.3、继承使用限制

虽然从本质上继承子类可以对父类操作进行共享，但是继承本身也是有限制的。

1、子类对象在实例化前，一定会先实例化父类对象，默认调用父类的构造方法后再调用子类的构造方法进行子类实例化，但是此时在子类，实际上在构造方法中隐藏了一个语句super(); ，此语句在无参是写与不写一样，但是如果此时你的父类中没有提供构造方法，那么要用super();明确你要调用的是父类构造方法。

格式： super(构造方法的参数);

2、java中值允许单继承，不允许多继承

一个子类只允许有一个父类

错误的多继承

class A;

class B;

class C extends A,B;

实际上，c类要同时继承A和B类的内容，此时可以用多层继承，但不能太多层

不允许多重继承，允许多层继承

3、继承时子类会继承父类的所有结构，所有的非私有操作属于显示继承，所有的私有操作属于隐式继承

## 12、覆写

如果子类定义了与父类相同的方法，或者属性，这就叫做覆写

## 12.1、方法的覆写

父类中使用private修饰的方法子类无法覆写！

所谓方法的覆写，指的是，子类定义了与父类方法名称、参数类型及个数

完全相同的方法，但是覆写的方法不能比父类拥有更严格的访问控制权限。

简单的覆写：

class Person{

private String name;

private int age;

public void printInfo() {

System.out.println("person类printInfo方法");

}

}

class Student extends Person{

public void printInfo() {

System.out.println("student类printInfo方法");

}

}

在以后，进行覆写方法使用时该关注一下两点：

1、当前的对象是哪个类的实例化对象

2、当调用某个方法，如果该方法已经被子类覆写，调用的是被子类覆写的方法

如果实例化的是被覆写的那个类，方法还是执行的原方法。

覆写的方法不能拥有比被覆写的方法拥有更严格的访问控制权限。关于访问控制权限才是封装的全部内容：private(私有)<default(空着默认)<public(公共)

结论 ：方法用public方法，至少保证绝大多数情况正确。

问题：如果父类的方法使用了private定义，子类使用了public覆写正确吗？

不正确，private方法，属性不能被任何外部类直接访问即便是子类也不行。

此时public定义的方法是子类定义的新方法。

补充：重载和覆写的区别：

1、overloading、override

2、重载方法名称，参数的个数和类型不同。覆写所有东西完全相同

3、重载发生在一个类中、覆写发生在继承关系中

4、重载没有要求，覆写的方法不能比父类拥有更严格的访问权限

5、方法重载返回值可以不同，但是良好的设计要求相同

## 12.2属性覆盖

当子类定义了与父类属性名称完全相同的属性时，就是属性覆盖

结论：在定义属性别整重名

## 12.3 super关键字

在子类对象实例化时使用过super，当时是子类调用父类的构造方法时使用的，那么在进行覆写的操作过程中，子类也可以用super.方法或者super.属性明确的调用父类的方法和属性。

观察程序：

class A {

public void print() {

System.out.println("Hello World .");

}

}

class B extends A {

public void print() {

super.print();

System.out.println("世界，你好！");

}

}

public class Test {

public static void main(String args[]) {

B b = new B();

b.print(); // 方法从父类继承而来

}

}

输出结果：

Hello World

世界，你好！

this和super的区别.



总结：

1、子类覆写父类的方法是因为父类方法的功能不足，才进行覆写

2、方法覆写使用public权限

## 13、final关键字

final被称为终接器

1、final修饰的类不能有子类

2、使用final定义的方法不允许被子类覆写

3、使用final定义的变量就变成了常量，必须在声明时赋值，且不允许被修改。

public static final int 变量名（全部大写）。 全局常量

## 14、多态性

在java中对多态的核心表现：

1、方法的多态性：

方法的重载：同一个方法名称根据参数类型和个数不同调用不同的方法体

方法的覆写：同一个父类的方法根据调用他的子类不同也有不同的实现

2、对象的多态性（前提：方法覆写）：

【自动 90%】对象的向上转型：父类 父类对象 = 子类实例；

【强制 1%】对象的向下转型：子类 子类对象 = （子类）父类实例；

9%是不进行转型。例如string

转型实例：

**class** A{

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

// **TODO** Auto-generated method stub

System.***out***.println("A");

}

}

**class** B **extends** A{

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

// **TODO** Auto-generated method stub

System.***out***.println("B");

}

**public** **void** funB() {

System.***out***.println("BfunB");

}

}

**public** **class** Demo {

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

// **TODO** Auto-generated method stub

A a = **new** B();//实例化子类对象

System.***out***.println(a **instanceof** A);

System.***out***.println(a **instanceof** B);

//这时候父类能调用的方法只能是本类定义好的方法

a.print();//调用被覆写过的方法

B b = (B)a;//向下转型

b.funB();

}

}

不管是是否发生了向上转型，本质核心还是你使用了哪个子类（new 后面）以及你使用的方法是否被覆写。

向下转型：将父类对象变成子类对象，在此之前明确，当需要用到子类扩充操作时，就使用向下转型

不是所有父类对象都可以向下转型：如果要进行向下转型之前一定要发生向上转型。

向下转型可能存在隐患：

最好的做法是先进行判断，可以通过instanceof关键字来判断：

子类对象 instanceof 类，返回boolean型数据

实现了子类和父类对象的转换，其意义：

要求定义一个方法，这个方法接收person类的所有子类实例，并调用person类的方法

对象的向上转型，有一个最核心的用途：操作参数统一

**class** Person{

**public** **void** takeOff() {

System.***out***.println("脱掉。。。");

}

}

**class** Student **extends** Person{

**public** **void** takeOff() {

System.***out***.println("一件一件脱掉。。。");

}

}

**class** Worker **extends** Person{

**public** **void** takeOff() {

System.***out***.println("直接脱掉、、、");

}

}

**public** **class** Demo {

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

*in*(**new** Worker());//Person a = new Worker

*in*(**new** Student());//向上转型

}

**public** **static** **void** in(Person per) {

per.takeOff();

}

}

1、对象的多态性实现的核心在于方法的腹泻

2、对象向上转型实现了接受参数的统一，向下转型实现了子类拓展函数的调用

3、两个没有关系的对象是不能够进行转型，会发生classcastexception，所以向下转型有安全隐患

## 15、抽象类的定义与使用

【90%的正确代码】在项目开发中，不要出现一个类去继承一个实现好的类，而只能继承抽象类和接口

对象多态性的核心本质在于方法的覆写，那么如果说子类没有进行方法的覆写，这样的操作就有些不符合要求了，所以如果要对子类的方法进行一些强制的要求就必须使用抽象类。

## 15.1、抽象类的基本概念

抽象类只是在普通类的基础上扩充了一些抽象方法的类，所谓的抽象方法指的是只声明不实现的方法，所有的抽象方法要求使用abstract关键字来进行定义，并且抽象方法所在的类也一定要用abstract来定义

**package** abstractClass;

**abstract** **class** A{

**private** String msg = "www.baidu.com";

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

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

}

**public** **abstract** **void** fun();//{}为方法体，所有的抽象方法没有方法体

}

**public** **class** abstractDemo {

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

// **TODO** Auto-generated method stub

}

}

由上图可知抽象类就是比普通类多了一些抽象方法而已。

抽象类中包含抽象方法，而抽象方法与普通方法最大的区别抽象方法没有方法体，即不知道具体的实现，如果抽象类的能够实例化意味着能够调用类中所有方法，所以抽象类不能实例化。

对于抽象类的使用原则：

·所有抽象类必须要有子类；

·抽象类的子类（不是抽象类）必须覆写抽象类中所有的抽象方法。

|-方法覆写一定要考虑权限问题：抽象方法可以使用任意权限，要求权限尽量使用public

·抽象类的对象可以理由对象多态性，利用子类为其实例化

**package** abstractClass;

**abstract** **class** A{

**private** String msg = "www.baidu.com";

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

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

}

**public** **abstract** **void** fun();//{}为方法体，所有的抽象方法没有方法体

}

**class** B **extends** A{

**public** **void** fun() {

**super**.print();

}

}

**public** **class** abstractDemo {

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

// **TODO** Auto-generated method stub

A a = **new** B();

a.fun();

}

}

从正常开发角度来讲，以上代码没有任何问题，也是使用最多的形式，但是对于抽象类还有一点说明，以后可能会出现以下形式

**package** abstractClass;

**abstract** **class** A{

**private** String msg = "www.baidu.com";

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

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

}

**public** **abstract** **void** fun();//{}为方法体，所有的抽象方法没有方法体

**public** **static** A getInstance() {

**class** B **extends** A{**public** **void** fun(){**super**.print();}}**return** **new** B();}}

**public** **class** abstractDemo {

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

// **TODO** Auto-generated method stub

A a = A.*getInstance*();

a.fun();

}}

将b类做成了一个内部类，通过getInstance方法取得这个类的对象。

## 15.2、抽象类使用限制

1、抽象类只是比普通类多了一些抽象定义而已，所以在抽象类中允许提供构造方法，并且在子类也会遵守子类对象的实例化流程：实例化子类之前调用父类的构造方法

观察：

其实抽象类含有构造方法很好理解，因为抽象类有属性，所有属性在对象实例化时进行开辟空间，对象实例化必须要构造方法。

构造方法没有执行，对象中的属性时其数据类型的默认值

2、抽象类中可以没有任何抽象方法，但是此时也无法实例化

3、抽象类不能用final声明

抽象方法不能够用private定义

4、抽象类也分为内部抽象类和外部抽象类，外抽象类不允许static修饰，内部抽象类才允许使用static

## 15.3、模板设计模式

抽象类的最大特点是规定了子类的结构，抽象类还起到了模板的作用。

抽象类在实际使用过程中会定义一些固化的模式，他只能接收几种特定指令。

但是每种指令的具体实现由子类实现。

最具代表性的就是servlet程序。

总结：

1、抽象类虽然定义了子类必须要做的事情，但是抽象类依然具有单继承局限

2、抽象类的使用必须要通过子类进行对象实例化处理

## 15、接口的定义与使用

抽象类相比于普通类的最大特点就是约定了子类的实现要求，但是抽象类有单继承局限，如果想要约定子类的实现要求，又要避免单继承局限请使用接口。在开发中，接口优先。

## 15.1、 接口的基本概念

接口就是一个抽象方法和全局常量的结合。

java中用interface关键字定义

子类想使用接口必须使用implements关键字来实现接口，一个子类可以实现多个接口即可以用接口实现多继承的概念，对于接口的子类如果不是抽象类，必须覆写所有的接口中所有的抽象方法，再通过子类的向上转型来实现接口的对象实例化

接口例子：

**interface** IMessage{

**public** **static** **final**/\*全局变量\*/ String ***MSG*** = "www.baidu.com";

**public** **abstract** **void** print();}

**interface** INews{

**public** **abstract** String get();

**public** **abstract** **void** print1();}

**class** MessageImpl **implements** IMessage,INews{

@Override

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

System.***out***.println(**this**.get());}

@Override

**public** **void** print1() {

System.***out***.println(**this**.get());}

@Override

**public** String get() {

**return** IMessage.***MSG***;//访问常量建议加上类名称

}}

**public** **class** IDemo {

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

// **TODO** Auto-generated method stub

IMessage aIMessage = **new** MessageImpl();//向上转型

aIMessage.print();//调用被子类覆写过的方法

MessageImpl aI = (MessageImpl)aIMessage;//向下转型

//上面如果是MessageImpl实际上对象是子类对象，如果是INews实际上是父类对象

aI.print1();

}

}

当一个子类继承了多个接口后，并且接口对象通过子类实例化，那么多个父接口之间是允许互相转化的，这个子类必须同时继承多个父类

## 15.2接口使用限制

接口一旦定义完成了之后，就需要对其有个核心的说明：接口里面只允许存在public权限不管是属性还是方法。（即便你不写public也是public，此时因为子类覆写不允许比父类方法拥有更加严格的权限，所以子

类的方法也只能是public）

另外由于接口中只是全局常量和抽象方法的结合，所以一下的两种定义形式最终效果都是一样的：

|  |  |
| --- | --- |
| 完整格式： | **interface** IMessage{  **public** **static** **final** String ***MSG*** = "www.baidu";  **public** **abstract** **void** print();  } |
| 简便格式： | **interface** IMessage{  String ***MSG*** = "www.baidu";  **void** print();  } |
| 好习惯格式： | **interface** IMessage{  public String ***MSG*** = "www.baidu";  public print();  } |

大部分接口中只有抽象方法，没有全局常量

为了避免开发者混乱一般在接口定义方法时都加上public

如果一个类既要实现接口又需要继承抽象类时，一般先extends继承一个抽象类再使用implements实现多个接口。