

面向对象及Java实践

适用班级：软件设计师

主讲：邓少勋

Q Q：154913686

网址：[www.bitpx.com](http://www.bitpx.com)

E-Mail:bitpx@163.com

分值说明：早上试题考5-9分

下午试题考15分

比特培训中心

贵州·贵阳

目 录

[第1节 面向对象程序设计概述 1](#_Toc463559348)

[1.1 结构化程序设计方法学 1](#_Toc463559349)

[1.2 面向对象程序设计方法学 1](#_Toc463559350)

[1.3 面向对象VS面向结构化 1](#_Toc463559351)

[1.3.1 活字印刷，面向对象 1](#_Toc463559352)

[1.4 面向对象的基本概念 1](#_Toc463559353)

[1.5 类及类的实例化 1](#_Toc463559354)

[1.5.1 JAVA中类的定义 2](#_Toc463559355)

[1.5.2 JAVA中类的实例化 2](#_Toc463559356)

[1.5.3 构造函数和析构函数 2](#_Toc463559357)

[1.5.4 Java中类型的组织方式-包(package) 3](#_Toc463559358)

[1.6 对象与封装 4](#_Toc463559359)

[1.6.1 对象三要素 4](#_Toc463559360)

[1.6.2 面向对象封装特性 4](#_Toc463559361)

[1.7 面向对象之继承特性 5](#_Toc463559362)

[1.8 面向对象之多态特性 7](#_Toc463559363)

[1.9 消息通信 7](#_Toc463559364)

[1.10 动态绑定 7](#_Toc463559365)

[1.11 面向对象方法学的优点总结 7](#_Toc463559366)

[第2节 面向对象之Java语言实践 7](#_Toc463559367)

[2.1 动态存储 7](#_Toc463559368)

[2.2 static静态成员 7](#_Toc463559369)

[2.3 Java中的继承与类的派生 8](#_Toc463559370)

[2.3.1 继承编码关键字 8](#_Toc463559371)

[2.3.2 Object祖先类 9](#_Toc463559372)

[2.3.3 继承结构代码运行情况 9](#_Toc463559373)

[2.3.4 继承中方法的覆盖(重写) 10](#_Toc463559374)

[2.4 Java中的多态机制 11](#_Toc463559375)

[2.4.1 函数重载实现编译时的多态 11](#_Toc463559376)

[2.4.2 Java中最终类和方法 11](#_Toc463559377)

[2.4.3 Java中抽象类和抽象方法 12](#_Toc463559378)

[2.4.4 Java语言中的接口类型 13](#_Toc463559379)

[2.4.5 Java中的泛型(类属)类型 15](#_Toc463559380)

[2.5 Java中List集合 16](#_Toc463559381)

[2.5.1 ArrayList和Vector实现类 16](#_Toc463559382)

[2.5.2 List集合的泛型编码 16](#_Toc463559383)

[2.6 函数的缺省参数 17](#_Toc463559384)

[2.7 this作为函数实参 17](#_Toc463559385)

[2.8 Java中的异常处理 18](#_Toc463559386)

[2.8.1 异常和异常处理 18](#_Toc463559387)

[2.8.2 自定义异常 19](#_Toc463559388)

# 面向对象程序设计概述

## 结构化程序设计方法学

自顶向下，逐步求精。程序结构按功能划分为若干个基本模块，这些模块形成一个树状结构；各模块之间的关系尽可能简单，在功能上相对独立；每一模块内部均由顺序、选择和循环三种基本结构组成；其模块化实现的具体方法是使用子程序。

* c语言、basic语言、Pascal语言、汇编等语言都是结构化语言。

## 面向对象程序设计方法学

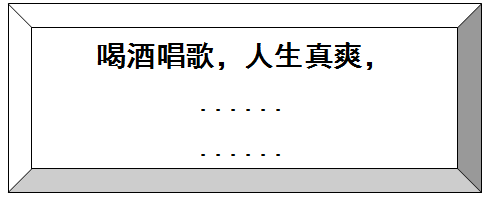
面向对象程序设计吸取了结构化程序设计的一切优点，又考虑了现实世界与面向对象解空间的映射关系，它所追求的目标是将现实世界问题的求解尽可能的简单化，将现实世界的问题空间映射到面向对象的解空间。

面向对象程序设计将数据及对数据的操作放在一起，作为一个相互依存、不可分割的整体来处理，采用数据抽象和信息隐藏技术。它将对象及对对象的操作抽象成一种新的数据类型—类，并且考虑不同对象之间的联系和对象类的重用性。

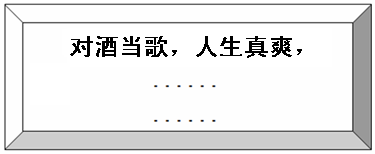
## 面向对象VS面向结构化

### 活字印刷，面向对象

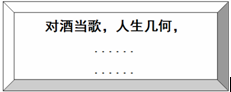
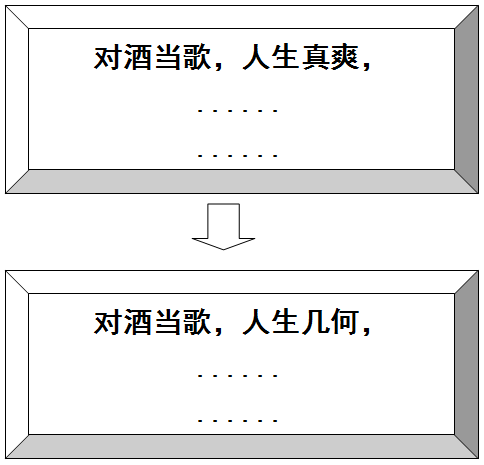
三国时期，曹操吟道：“喝酒唱歌，人生真爽。。。。”。于是一臣速命印刷工匠刻版印刷，以便流传天下。



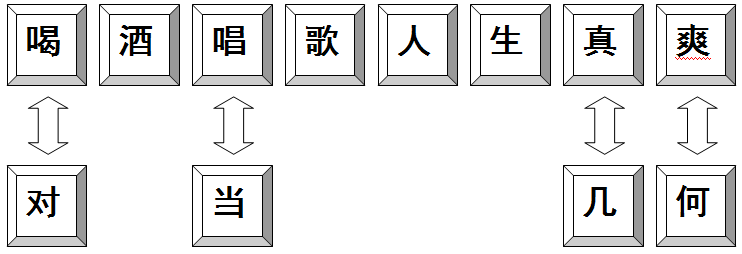
样张出来给曹操一看，曹操感觉不爽，修改为：“对酒当歌，人生真爽”，于是此臣就命工匠重新来过。工匠眼看连夜刻版之工，彻底白费，心中叫苦连天，但是只得照办。



样张再次出来，曹操再要求修改为“对酒当歌，人生几何”。当大臣转告工匠之时，工匠昏倒....!



工匠昏倒的原因：三国时期还没有活字印刷技术，所以要改字，就必须要整个刻板全部重新刻。如果采用活字印刷技术：



##### 采用活字印刷(面向对象)的好处

* 要改，只需要改要改的字，此为可维护;
* 这些字并非用完一次就无用，完全可以在后来的印刷中重复使用，此为可复用;
* 此诗若要加字，只需要另刻字加入即可，此为可扩展;
* 字的排版其实可能是竖排，也可能是横排，此时只需要活字移动就可以做到满足排列需求，此为灵活性好。

##### 采用刻版印刷(结构化)

要修改、要加字、要重新排列必须重刻，而且印完此版没有任何利用价值。

总结：面向对象好处具有可维护、可复用、可扩展，灵活性好。

●(2006年下半年)面向对象分析的第一步是 33 。

（33）A．定义服务 B.确定附加的系统约束 C．确定问题域 D.定义类和对象

解析：服务=行为或操作，实际就是类中函数或方法

## 面向对象的基本概念

* 面向对象＝**类+继承+对象+消息通信**。
* 面向对象系统最突出的特性:封装性、继承性和多态性。

## 类及类的实例化

类是对具有相同属性和服务的一个或同一类事物的抽象定义。如对所有的学生群体进行抽象可定义学生类Student，分析所有教师的共同特性和行为可抽象出教师类Teacher等。

### Java运行环境配置

##### 安装JDK 1.8

登录比特网www.bitpx.com学习系统，在相应的链接处下载。

##### 配置JDK环境变量

JAVA\_HOME: C:\Program Files\Java\jdk1.8.0\_201 采取新建的方式

PATH: %JAVA\_HOME%\bin;%JAVA\_HOME%\jre\bin; 将路径值复制到系统PATH路径值首部

CLASSPATH: .;%JAVA\_HOME%\lib;%JAVA\_HOME%\lib\dt.jar;%JAVA\_HOME%\lib\tools.jar; 采取新建的方式

##### 使用IDEA或Eclipse运行Java程序

### JAVA中类的定义

在Java源代码文件Student.java中定义类如下:

**public** **class** Student {//定义Student类

//定义类的数据成员（属性、特征）

**private** String no;//学号

**private** String name; //姓名

**private** **float** score; //成绩

**protected** **double** money; //钱

//......

//定义类的函数成员（行为、方法、操作）

**public** String getNo() {

**return** no;

}

**public** **void** setNo(String no) {

**this**.no = no;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** **float** getScore() {

**return** score;

}

**public** **void** setScore(**float** score) {

**this**.score = score;

}

**public** **double** getMoney() {

**return** money;

}

**public** **void** setMoney(**double** money) {

**this**.money = money;

}

//......

}

### JAVA中类的实例化

使用关键字“new”进行对象实例化,如StudentManager.java文件中代码:

**public** **class** StudentManager {

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

Student stu1;//stu1为空引用，相当于C或C++中的指针变量,此刻并无真正的对象诞生

Student stu2=**new** **Student**();//实例化类为对象,stu为对象引用,其存储在内存栈区,而实际的对象空间由new在堆区申请

//......

}

}

* 类的实例化过程是一种实例的合成过程，而不仅仅是根据单个类型进行的空间分配、初始化和绑定。指导编译程序进行这种合成的是多态的种类。

**类与对象的关系:**

* 类是对对象的抽象,对象是类的实例化。对象是客观存在的，包含看得见摸得着的或存在于意识形态的。

### 构造函数和析构函数

* 它们都没有返回值说明，定义它们的时候不需指出函数返回值类型，如果程序员未定义构造函数和析构函数，则系统会自动给当前类加上无任何参数值的构造函数和析构函数；而一旦程序员自己编写了构造函数或析构函数，系统便不会自动添加无参的构造函数和析构函数。
* 它们不能被继承。
* 和大多数函数一样，构造函数可以用缺省参数。
* 析构函数可以是虚的，但构造函数不行。
* 不可取它们的地址。
* 构造函数的作用是用来构建对象，当定义对象时，编译程序自动调用构造函数且可通过构造函数给对象内的成员赋初值；当删除对象时，编译程序自动地调用析构函数释放对象所占用的内存空间。
* 构造函数的名字必须与类名相同。
* 构造函数是对象生命周期中第一个被系统自动调用的函数，而析构函数是最后一个被系统自动调用的函数。

class Student

{

private String no;

private String name;

public Student(String no,String name)//带参数的构造函数

{

this.no=no;

this.name=name;

}

public Student()//无参数的构造函数

{

}

public String getNo() {

return no;

}

public void setNo(String no) {

this.no = no;

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

}

public class Test {

public static void main(String[] args)

{

Student s1=new Student("100", "王大丫");//通过构造函数赋值

s1.setNo("101");//修改学号

s1.setName("王小丫");//修改姓名

Student s2=new Student();//调用无参的构造函数

s1.setNo("101");//设置学号

s1.setName("王小丫");//设置姓名

}

}

以上Java代码中，Student定义了两个构造函数，一个有参数，一个无参数，形成了函数重载。C++语言中有析构函数，而Java语言无析构函数，因为Java语言有自动的垃圾回收机制。

### Java中类型的组织方式-包(package)

为了更好地组织类，Java提供了包机制，用于区别类名的命名空间。包的作用如下：

#### 把功能相似或相关的类或接口组织在同一个包中，方便类的查找和使用。

#### 如同文件夹一样，包也采用了树形目录的存储方式。同一个包中的类名字是不同的，不同的包中的类的名字是可以相同的，当同时调用两个不同包中相同类名的类时，应该加上包名加以区别。因此，包可以避免名字冲突。

#### 包也限定了访问权限，拥有包访问权限的类才能访问某个包中的类。

#### Java使用包这种机制是为了防止命名冲突，访问控制，提供搜索和定位类（class）、接口、枚举（enumerations）和注释（annotation）等。

包语句的语法格式为：

package pkg1[．pkg2[．pkg3…]];包名之间用点号“.”分割。

引用已经存在包使用“import”关键字。代码文件“A.java”中的内容为：

**package** com.ba; //定义包“com.ba”，在eclipse环境中，其对应的物理文件夹路劲为：src/com/ba

**import** java.lang.\*;//引用系统自带的包,其中“\*”表示包java.lang中的所有类型。

public class A {

String de="default";

private String pri="private";

protected String pro="protected";

public String pub="public";

public void Print()

{

System.out.println(de+" "+pri+" "+pro+" "+pub);

}

}

源文件“A.java”的物理位置为com.ba.A，其他包中的代码如要使用类A，则需要加“import com.ba.A”或“import com.ba.\*”指令到文件中。

**在Java语言中，main函数必须放置于公共类中，而公共类必须与源代码文件名称相同，一个源代码文件中，只能包含一个公共类。**

## 对象与封装

对象(object)是系统中用来描述客观事物的一个实体，它是构成系统的一个基本单位，是类的实例。类与对象是抽象与具体实例的关系，一个具体的对象被称做类的一个实例。

面向对象的软件系统是由对象组成的，复杂的对象是由比较简单的对象组合而成。也就是说，面向对象方法学使用对象分解取代了传统方法的功能分解。

### 对象三要素

对象三要素：对象标志、属性和服务。

* 对象标志：也就是对象的名字，供系统内部唯一地识别对象。
* 属性：也称状态或数据，用来描述对象的静态特征。在某些面向对象的程序设计语言中，属性通常被称为成员变量或简称变量。
* 服务：也称操作、行为或方法等，用来描述对象的动态特征。某些面向对象程序设计语言中，服务通常被称为成员函数或简称函数。

### 面向对象封装特性

对象的一个重要原则，目的是使对象的定义与实现分离。它有两层含义：

对象是其全部属性和全部服务紧密结合而形成的一个不可分割的整体；

对象是一个不透明的黑盒子，表示对象状态的数据和实现操作的代码都被封装在黑盒子里面。使用一个对象的时候，只需知道它向外界提供的接口形式(公有段的函数定义)，无须知道它的数据结构细节和实现操作的算法。从外面看不见，更不可能从外面直接修改对象的私有属性、保护属性。

在Java语言中,权限修饰符号描述如表 1.1所示:

表 1.1 Java中权限修饰符号

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 权限修饰符合 | 当前类 | 当前包 | 子类 | 其他包 |
| private(私有) | √ | × | × | × |
| default(默认，包权限) | √ | √ | × | × |
| protected(保护) | √ | √ | √ | × |
| public(公有) | √ | √ | √ | √ |

* private(私有) ：处于私有段(private)的属性或行为是不对外公开的，只能在当前类定义体的函数体中访问，当前包、子类及其他包中都不能直接访问，即使是“对象名.私有属性或行为”这种写法都是错误的。
* default(默认) ：类的成员前不加任何权限修饰符号即默认为default(默认)权限,default成员在当前类、当前包中可直接访问，子类和其他包中不能直接访问。
* protected(保护)：处于保护段(protected)的属性或行为主要对其派生类对象提供服务，在当前类定义体的函数体中，当前包中也可直接访问，其他包中不能直接访问，即使“对象名.保护属性或行为”这种写法都是错误的。
* public(公有) ：处于公有段(public)的属性或行为是向所有外界公开的功能，它可以响应外界对象的请求，通过“对象名.公有属性或行为”这种方式访问。公有段主要是向外界提供一个访问私有段和保护段的一种接口。

源代码文件“B.java”(与A.java文件属于同一个包)内容为:

package com.ba;//定义包“com.ba”

import java.lang.System;//引用包java.lang中的类System

public class B {//与类A属于同一个包com.ba

public static void main(String[] args)

{

A a=new A();

//System.out.println(a.pri);//private成员只能在当前来中访问,此处无权访问

System.out.println(a.de);//访问同一个包中的default成员

System.out.println(a.pro); //访问同一个包中的protected成员

System.out.println(a.pub); //访问同一个包中的public成员

a.Print();//访问同一个包中的public成员

}

}

源代码文件“B.java”(与A.java文件不属于同一个包)内容为:

**package com.bb**;

**package** com.bb;

**import** java.lang.System;

**import** com.ba.A;

**public** **class** B {

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

{

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

//System.out.println(a.pri);//private成员只能在当前来中访问

//System.out.println(a.de);//default成员不能直接跨包引用

//System.out.println(a.pro);//protected成员不能直接跨包引用

System.***out***.println(a.pub);//public成员可跨包访问

a.Print();//public成员可跨包访问

**new** C().Print();

}

}

**class** C **extends** A{ //类C派生于类A

**public** **void** Print()

{

System.***out***.println(pro+" "+**super**.pub);//派生类中可直接访问其他包的protected和public段成员

**super**.Print();

//System.out.println(super.pri+" "+super.de);//派生类中不能直接访问其他包的default和private段成员

}

}

从以上代码可看出,包com.ba与com.bb中都包含同名的类B,但由于可使用包名前缀来区分,所以不会出现任何问题。在类C中,pro和pub都是从父类A中继承下来的,派生类可直接访问(虽然类C与类A不在同一个包)。在继承结构代码中，super表示对父类对象的引用，this表示对当前类对象的引用。

问题：如果去掉“**super**.Print()”中的super会出现什么问题？为什么？学完本课程继承知识自然就懂了！！！

## 面向对象之继承特性

是面向对象程序设计语言不同于其它语言的主要特点，是类之间共享属性和操作的机制。其定义为：特殊类（或称子类、派生类）的对象拥有其一般类（或称父类、基类）的全部属性与服务，称做特殊类对一般类的继承(分为单继承和多继承)。在面向对象的方法学中，继承是提高软件开发效率的重要机制之一。继承有私有继承、保护继承、公有继承，三种继承图如图 1.1所示。

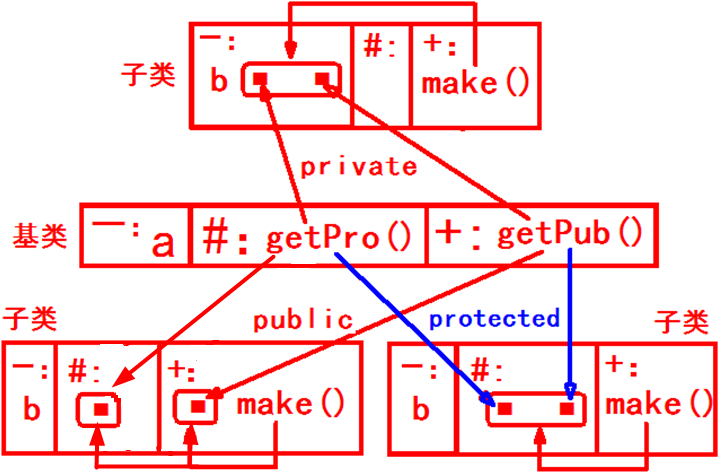


图 1.1面向对象继承图

* 私有继承：由私有派生得到的派生类，对它的基类的公有成员(public)和保护段成员只能是私有继承。也就是说基类的所有公有成员和保护段成员只能成为私有派生类的私有成员(private)，这些私有成员只能被派生类的成员函数访问，而派生类的使用者无权访问。基类中的私有段数据不能被继承。
* 保护继承：在基类标志符前加一个protected关键字。由保护继承得到的派生类，把它的基类的公有成员(public)和保护段成员变成自己的保护段成员。也就是说基类的所有公有成员和保护段成员变成为派生类的保护段成员。基类中的私有段数据不能被继承。
* 公有继承：在基类标志符前加一个public关键字。由公有派生得到的派生类，对它的基类的公有成员(public)和保护段成员原样继承。也就是说基类的所有公有成员和保护段成员变成为公有派生类的公有成员和保护段成员。基类中的私有段数据不能被继承。

继承是一种“is-a”的关系，如计算机类书籍是一种书籍，数学类书籍也是一种书籍，其中书籍是父类型，计算机类书籍和数学类书籍是派生类型。

##### C++语言中的继承代码机制

C++语言支持单继承及多继承于类,且有私有继承、保护继承和公有继承实现方式。

class A{

//……

};

class E:**private A**{//单继承,私有继承

//….

};

class B:public A{//单继承,公有继承

//……

};

class C:public B{//单继承,公有继承

//……

};

class D:protected A{//单继承,保护继承

//……

};

class F:private A,protected B,C,public D//多继承

{

//…

};

类B是从类A中派生出来的，也就是说A是B的基类，而且是直接基类，B是A的派生类；类C是从类B中派生出来的，也就是说B是C的直接基类，C是B的派生类。对于C类来讲，除去B是它的直接基类外，A也是它的基类，但是是间接基类,同理类A是类D的直接基类。由此，A,B,C,D这四个类之间的继承关系就形成了一个类的层次。

●【2006年上半年】已知3个类O、P和Q，类O中定义了一个私有方法F1、一个公有方法F2和一个受保护的方法F3；类P和类Q是类O的派生类，其继承方式如下所示：

class P : protected O {…};

class Q : public O {…};

关于方法F1的描述中正确的是 (34) ；关于方法F2描述中正确的是 (35) ；关于方法F3的描述中正确的是 (36) 。

(34)A.方法F1无法被访问 B.只有在类o内才能访问方法F1

C.只有在类P内才能访问方法F1 D．只有在类Q内才能访问方法F1

(35)A．类O、P和Q的对象都可以访问方法F2 B．类P和Q的对象都可以访问方法F2

C.类o和Q的对象都可以访问方法F2 D．只有在类P内才能访问方法F2

(36)A．类0、P和Q的对象都可以访问方法F3 B.类o、p和Q的对象都不可以访问方法F3

C．类0和Q的对象都可以访问方法F3 D．类P和Q的对象都可以访问方法F3

##### 继承结构中构造函数和析构函数的执行顺序

#include<iostream>

using namespace std;

class Parent

{

public:

Parent()//无参的构造函数

{

cout << "父类对象诞生了!" << endl;

}

~Parent()//析构函数

{

cout << "父类对象死翘翘!" << endl;

}

};

class Son:public Parent

{

public:

Son()**:Parent()** //无参的构造函数

{

cout << "子类对象诞生了!" << endl;

}

~Son()//析构函数

{

cout << "子类对象死翘翘!" << endl;

}

};

void main()

{

Son \*son = new Son();//诞生派生类对象

**delete** son;//释放派生类对象空间

}

输出结果为：

父类对象诞生了!

子类对象诞生了!

子类对象死翘翘!

父类对象死翘翘!

由以上C++代码输出结果可知，构建派生类对象时要先执行父类的构造函数构建父类对象空间，然后才执行派生构造函数构造子类对象空间。而在释放对象占用内存空间时，系统先执行子类析构函数释放子类对象空间，然后执行父类析构函数释放父类对象空间。

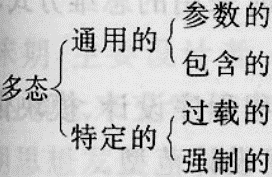
## 面向对象之多态特性

在收到消息时，对象要予以响应。不同的对象收到同一消息可以产生完全不同的结果，这一现象称为多态。在使用多态的时候，用户可以发送一个通用的消息，而实现的细节由接收对象自行决定。这样，同一消息就可以调用不同的方法。

多态的实现受到继承的支持，利用类继承的层次关系，把具有通用功能的消息存放在高层次，而不同的实现这一功能的行为放在较低层次，在这些低层次的对象能够给通用消息以不同的响应。

多态总体上分为：编译时的多态(静态多态)和运行时的多态（动态多态）。

Cardelli和、Wegner把多态细分为4类：



* 参数多态：采用参数化模板，通过给出不同的类型参数，使得一个结构有多种类型。如C++语言中的函数模板和类模板属于参数多态。参数多态又叫静态多态，它的执行速度快，异常少，调用在编译时已经确定。参数多态是应用比较广泛的一种多态，被称为最纯的多态。
* 包含多态：在许多语言中都存在，最常见的例子就是子类型化，即一个类型是另外一个类型的子类型。一般需要进行运行时的类型检查，属于动态多态。包含多态的基础是虚函数。虚函数是引入了派生概念后用来表现基类和派生类的成员函数之间的一种关系。
* 过载多态：同一个名字在不同的上下文中所代表的含义不同。典型的例子是运算符重载和函数重载，属于静态多态。
* 强制多态：编译程序通过语义操作，把操作对象的类型强行加以变换，以符合函数或操作符的要求。程序设计语言中基本类型的大多数操作符，在发生不同类型的数据进行混合运算时，编译程序一般都会进行强制多态。程序员也可以显示地进行强制多态的操作。如int+double，编译系统一般会把int转换为double，然后执行double+double运算，这个int->double的转换，就实现了强制多态，即可是隐式的，也可显式转换。强制多态属于静态多态。

多态性不仅增加了面向对象软件系统的灵活性，进一步减少了信息冗余，而且显著提高了软件的可重用性和可扩充性。

**编写代码时，对动态多态的体现为：父类型的指针(引用)可以指向任何派生类对象。**

## 消息通信

* 消息：就是向对象发出的服务请求，它包含供服务的对象标志、方法名、输入信息和回答信息。
* 消息通信：也是面向对象方法学中的一个重要原则，它与对象的封装原则密不可分，一个对象通过发送消息来请求另一对象为其服务。

## 动态绑定

绑定是一个把过程调用和相应调用所需要执行的代码加以结合的过程。

* 静态绑定：绑定是在编译时进行的。
* 动态绑定：是在程序运行时进行的，在运行过程中，当一个对象发送消息请求服务时，要根据接收对象的具体情况将请求的操作与实现的方法进行连接，即动态绑定。

## 面向对象方法学的优点总结

* 与人类的思维方法一致
* 稳定性
* 可重用性强
* 容易开发大型软件
* 可维护性好

总之，面向对象软件设计方法灵活性强、可重用性高、可维护性好、扩展性强!

# 面向对象之Java语言实践

## 动态存储

在C语言中，允许动态地分配存储空间及动态地释放已分配的存储空间，它所使用的是函数malloc()和free()。

* 申请：int \*p=(int \*)malloc(5\*sizeof(int))
* 释放：free(p)

在C++使用new来动态分配存储空间，使用delete来动态释放存储空间。Java中也使用new来分配内存单元,但无须手工写代码释放空间,Java的垃圾回收机制会自动释放空间。使用new分配空间具备如下优势：

* new自动计算要分配的类型的大小，这样既省事，又可以避免存储量分配的偶然性错误；
* 它自动返回正确的指针类型，不必对返回指针进行类型转换；
* 可以用new将分配的对象初始化。

## static静态成员

静态成员：除了静态变量，还包括静态函数，用static来定义,是用来实现同一个类的不同对象之间的数据共享。同一个类的不同对象共享静态数据成员值，当通过一个对象改变了静态数据成员的值时，同类的其他对象可以看到这个修改。

例如：

class Counter {

private static int count;//静态的成员变量

private int objNo;

public int getObjNo() {

return this.objNo;**//this为对当前对象自身的引用,通过this访问对象中的objNo毫无问题**

}

public void setObjNo(int objNo) {

this.objNo = objNo;

}

public static int getCount() { //静态成员函数

return count;**//此处的count不能写成this.count,因为count不属于任何对象而是属于类,故可写成Counter.count**

}

public static void setCount(int count) { //静态成员函数

Counter.count = count;

}

public static void printNo(Counter o)//静态成员函数访问非静态成员变量时需要传入具体对象

{

System.out.println(o.objNo);

}

public Counter(int objNo) { //构造函数

this.objNo = objNo;

Counter.count++;

}

};

public class Test {

public static void main(String[] args) {

Counter.setCount(0);//此刻还未诞生任何对象

System.out.println("目前对象个数:"+Counter.getCount());

Counter o1 = new Counter(100);

Counter o2 = new Counter(101);

System.out.println("目前对象个数:"+Counter.getCount());//也可使用o1.getCount()或o2.getCount()来访问count

Counter.printNo(o2);//静态成员函数访问非静态成员变量时需要传入具体对象

}

}

静态成员函数

当一个成员函数被说明为static时，作为静态的成员函数与静态成员变量一样,同属于整个类，称为此类所有对象所共享的成员，而不属于类中的某个对象。

静态成员函数的访问

* 对象名.静态成员函数(“.”并不表示所属关系，而是调用关系)
* 类名.静态成员函数(既是所属关系，又是调用关系)
* 静态成员函数不能像一般的成员函数那样随意地访问对象中的非静态的数据内容，一般情况下，静态的成员函数里面只访问静态的成员变量，如果一定要使用非静态成员变量，需传入相应的对象,如代码中printNo函数。
* 在一般的成员函数中含有一个this指针，用来指向对象自身；而在静态成员函数中是没有this指针，因为它无需判别执行它的是哪一个对象，它是同类的所有对象共享的资源，只有一个共享的副本。
* this:对象自身引用，是面向对象程序设计语言中特有的、十分重要的机制。每个对象都有属于自己的对象自身引用值。只能使用this来引用实例级别成员，不能引用static成员(类级别成员)。

**问题：**在一个实例级别函数中，如何找到该函数所属的对象？

## Java中的继承与类的派生

### 继承编码关键字

由于面向对象中多继承于类机制弊大于利，故JAVA语言只支持单继承于类，但支持多继承于接口类型。Java中只有一种公有继承(原样继承)方式，继承于类(包含普通类和抽象类)使用关键字extends，继承于接口类型使用关键字implements。代码如下所示:

interface ICpu { //... }

interface IMemory { //... }

abstract class Parent1 {//... }

class Parent2 {//... }

class Son1 **extends** Parent1 { //... } //继承于抽象类

class Son2 **extends** Parent2 { //... }//继承于一般类

class Son3 **extends** Parent2 **implements** ICpu, IMemory { //... }单继承于类而多继承于接口类型

**Java中的继承即公有继承，父类的保护段及公有段被派生类继承下来，仍然放置于派生类的保护段及公有段，父类的私有段成员不能被继承。**

### Object祖先类

在Java语言中，Object类是系统自带的其他任何类型的默认父类，一个类如没有显示继承于某个父类，则系统默认让该类继承于Object类型。如下代码中A类的的父类型都为Object类型，故A类将Object中的equals()、toString()等方法直接下来使用。

class A{//... } //默认继承于Object类

class A **extends** Object{ //显示继承于Object类

//...

}

### 继承结构代码运行情况

class Parent {

private int a;

protected int b;

public Parent(int a) {

this.a = a;

System.out.println("父类对象诞生了!");

}

public int getA() {

return this.a;

}

}

class Son **extends** Parent {

private int b;

public Son(int a, int b) {

**super(a);**// 先执行父类构造函数构建父类对象,必须放置于派生类构造函数体中第一行

**this**.b = b;//this引用本类中的变量b

**super**.b = ++b;//super引用从父类继承下来的变量b

System.out.println("派生类对象诞生了!");

}

public void print() {

System.out.println(String.format( "a=%s,b of son is:%s,b of derived from parent is:%s", super.getA(), this.b, super.b));

}

}

public class DerivedTest {

public static void main(String[] args) {

Son son = new Son(1, 2);

son.print();

}

}

输出结果:

父类对象诞生了!

派生类对象诞生了!

a=1,b of son is:2,b of derived from parent is:3

以上代码中,要实例化派生类对象,必须先实例化父类对象,故super(a)必须放置于派生类构造函数体中第一行, 先执行父类构造函数构建父类对象,因为没有“老子”怎么可能有“儿子”呢？！this与super关键字使用对比如下：

* super是对父类的一个引用，在一个对象的内部，如果想得到该对象父类的引用,要用super,而不是父类名称；子类引用父类的成员：super.变量名，super.方法名（实参列表）。
* 如果调用的父类构造函数是无参的构造函数，即super()方式调用，则此调用可省略不写。
* 子类自己定义的成员，只能使用this引用，在不会引起歧义的情况下可省略this。
* 子类从父类继承下来的成员，如果未与子类自定义成员同名，可使用this或super引用，也可省略，但推荐使用super关键字，因为一看便知此成员是继承下来的，含义清晰。
* 如果子类存在与从父类继承下来的同名成员，则使用this只能引用子类自定义的成员，而从父类继承下来的成员需要使用super引用，直接写成员名相当于省略了this，即与写this效果一样。
* this是对当前对象的一个引用，即代表当前对象自身；在一个对象的内部，如果想得到该对象的引用,要用this,而不是它的名称。
* this可以在普通成员方法和构造方法中使用，不能在静态方法中使用。
* 对于static属性和方法，不能有this引用*。*
* 调用同一个类中的其他构造函数，使用this关键字。

class Person {

String name;

String sex;

public Person(String name,String sex)

{

this.name=name;

this.sex=sex;

}

public Person()//无参的构造函数

{

**this**("dsx","male");//调用有参数的构造函数

}

public void show() {

System.out.println(name +" "+ sex);

}

}

上述代码中出现了构造函数的重载，在无参的构造函数中，使用“this("dsx","male");”调用有参数的构造函数。

### 继承中方法的覆盖(重写)

在一个类的子类中,可以定义与父类中的方法同名的方法,并且它们的参数列表,返回类型也完全相同,子类中的方法就会**覆盖掉**父类中的方法,这种行为就叫方法覆盖。子类中方法的访问控制不能比父类中被覆盖方法的访问控制范围窄。

**class** Person {

String name;

String sex;

**public** Person(String name, String sex) {

**this**.name = name;

**this**.sex = sex;

}

**public** **void** print() {//直接被派生类继承

System.***out***.println("print中:" + name + " " + sex);

}

**public** **void** show() {//被覆盖方法

System.***out***.println(name + " " + sex);

}

}

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

**int** num;

**double** score;

**public** Student(**int** num, **double** score, String name, String sex) {

**super**(name, sex);

**this**.num = num;

**this**.score = score;

}

**public** **void** show() {// 覆盖父类同名方法

System.***out***.println(name + " " + sex + " " + num + " " + score);

}

**public** **void** anotherShow() {

**super**.show();// 调用父类中的show()

}

}

**public** **class** Test {

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

Person p = **new** Student(100, 98.99, "dsx", "male");// 父类型引用指向派生类对象

Student s = **new** Student(100, 98.99, "dsx", "male");// 派生类引用指向派生类对象

p.print();// 派生类继承下来的

p.show();

s.print();// 派生类继承下来的

s.show();

s.anotherShow();

// p.anotherShow();//错误

}

}输出结果:

dsx male 100 98.99

dsx male 100 98.99

dsx male

以上的代码中,子类Student中**show()方法隐藏了从父类Person中继承下来的show()方法**，故不论是父类型的引用还是派生类型的引用调用的show()方法都是派生类中的版本；在派生中，如果要调用从父类中继承下来的show()版本，则必须使用super前缀，如anotherShow()方法中的super.show()。

* 问题1：“s.anotherShow()”能运行正常，而“p.anotherShow()”则编译错误，为什么？

答案：**一个引用要能访问某成员，要求此引用的类型定义体中至少包含此被引用成员的声明！！**

* **问题2:** Student类中并非定了print函数,为什么“s.print()”不会出错？

答案：Student将print方法从父类Person继承下来了。

* **问题3:** Student类中show方法修改为“**public** **void** show(int a)”后会发生什么现象？

答案：其与继承下来的“**public** **void** show()”形成了函数重载现象。

●重置的基本思想是通过 (48) 机制的支持，使得子类在继承父类界面定义的前提下，用适合于自己要求的实现去置换父类中的相应实现。

(48)A．静态绑定 B．对象引用 C．类型匹配 D．动态绑定

## Java中的多态机制

### 函数重载实现编译时的多态

两个或两个以上的函数，取同一名字，只要使用不同类型的参数或参数个数不同，编译器便知道在什么情况下该调用哪个函数，这叫做函数重载。

import java.lang.\*;

class Number {

private int i;

private String s;

public int Max(int i) {

return this.i > i ? this.i : i;

}

public String Max(String s) {

return this.s.compareTo(s) > 0 ? this.s : s;

}

public Number(int i, String s) {

this.i = i;

this.s = s;

}

}

public class OverLoadExample {

public static void main(String[] args) {

Number num = new Number(10, "china");

System.out.println(num.Max(5));

System.out.println(num.Max("chinese"));

}

}

输出结果：

10

chinese

上例中的多态通过函数重载来实现，重载(编译时的多态性通过重载来实现)了两个函数，均为max，它们的功能为将函数的参数分别与类中各私有数据比较大小。函数名相同，它们的区别在于函数参数的类型不同。

**在定义函数重载时，函数名字相同，但函数所带的参数的个数或类型必须有所区别，否则就会出现二义性。两个无关：与参数名称无关；与函数返回值无关。**

在面向对象系统中除了函数可以重载外，运算符也可以重载。

**编译时的多态一般由函数重载来实现，而运行时的多态在C++中通过虚函数来使用,但在Java中,运行时的多态则通过抽象类、接口等进行综合实现。动态多态编码体现为：**

* 父类型的引用可指向任何派生类型对象，故引用需要定义为父类型，而被实例化的是实现了具体功能的子类型。
* 父类型的引用要能引用(调用)某成员函数,则此成员函数在父类型中至少要有函数的声明存在。

### Java中最终类和方法

##### final修饰最终类

在Java语言中，一个类可以是最终的，最终类使用关键字final来描述，最终类不能被继承。以下代码写法错误：

public **final** class A{

… …

}

public class B extends A{ //错误写法，最终类A不能被继承

… …

}

##### final修饰最终方法

在Java语言中，一个方法也可以是最终的，最终方法也是使用关键字final来描述。最终方法不能被子类中的方法覆盖。

public class A{

public **final** void fun()

{ … }

}

public class B extends A{

public void fun(){…} //错误,不能覆盖父类中继承下来的最终方法

}

##### final常量

在声明成员变量或局部变量的时候加上final修饰符，则相当于将其定义为常量。Java规定，final常量没有默认值，必须在定义时指定其数据值。常量一旦定义，其值就再不能改变，即不能被重新赋值。

final double PI ; //错误！常量必须指定数据值

final double PI = 3.14;//正确。定义了常量PI固定代表实数3.14

PI = 3.1415926; //错误！常量不能被重新赋值

### Java中抽象类和抽象方法

抽象类是对类的抽象，以关键字abstract来说明,抽象类不能用来创建对象，是专门设计来让子类继承的;定义抽象类和抽象方法的目的是建立抽象模型，它们只关心类应具备的功能，而不关心功能具体实现的细节。

有时候在父类中可能无法给出对所有子类都有意义的共同程序代码，抽象方法的意义就是即使无法给出方法的具体实现内容，还是**可以定义出一组子类共同的协议(函数头)**。

**抽象类本身不具备实际功能，它只是制定了“规范”，仅用于衍生子类。抽象方法是没有方法体的，也称没有实现的方法，以一个紧跟在声明后的“；”结束；**抽象类不是必须包含抽象方法;如果一个类里有抽象的方法，则这个类就必须声明成抽象的;不能生成抽象类的实例对象，只能使用实现抽象类中抽象方法的子类对象。

##### 抽象类不能被实例化

abstract class A { //抽象类

abstract void fun1(); //抽象方法

}

public class App{

public static void main(String[] s){

A a = **new** A();//编译错误:抽象类不能使用new运算符来生成它的实例对象

……

}

}

##### 一个类里面有抽象方法,这个类也必须是抽象的。

**abstract** **class** A {//包含抽象方法的类必须定义为抽象类

**int** a;

**public** A()//构造函数

{

a=0;

}

**abstract** **void** fun(); // 抽象方法

**public** **void** print()//未被覆盖

{

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

}

}

**class** B **extends** A { // B派生于抽象类A

**public** B()//无参数

{

**super**();//调用父类构造函数

}

**void** fun() {**//子类B实现抽象方法，或称为覆盖抽象类中的抽象方法**

System.***out***.println("子类B实现抽象方法");

}

}

**public** **class** Test {

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

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

o.fun();

o.print();//派生类继承下来的

}

}

在以上代码中，父类的fun()方法被派生类B实现(覆盖)，抽象类中print()方法直接被派生类B继承使用。由类A的定义可知，抽象类中可包含抽象函数、一般函数及数据成员。

抽象类一般是用来生成子类的；在抽象类的子类中，需要覆盖抽象类中的抽象方法，也叫做实现抽象方法。如果在子类中没有实现抽象父类中的所有抽象方法，则这个子类也是一个抽象的类。private和static的方法不能声明为抽象的。

abstract class A { //抽象类

abstract void fun1(); //抽象方法

}

**abstract** class B extends A{//子类没有实现抽象类中的抽象方法fun1(),所以子类也是抽象类

void fun2(){

System.out.println("子类B自己的方法");

}

}

##### 抽象类及抽象方法小结

* 抽象类中可以包含抽象方法和非抽象方法。
* 一旦某个类中包含了抽象方法，则该类必须使用abstract说明为抽象类。
* 抽象方法必须用abstract来说明，抽象方法没有方法体，在方法首部后以分号结束。
* 抽象方法必须存在于抽象类中，不能出现在普通类里。
* abstract不能用来修饰实例变量。
* abstract不能与private、static、final等同时修饰一个成员方法。
* **抽象类代表此类必须要被extends过才能使用，抽象方法代表此方法必须被覆盖过才能使用。**
* **抽象类引用指向实现(派生)类对象时,其只能调用抽象类中实现的或声明过的方法。**

### Java语言中的接口类型

与类相同,一个接口也是一种数据类型；Java中的接口interface是一种特殊的结构，它像是一个100%抽象的类,因为接口里面的方法都是抽象方法；接口的定义与类的定义很相似,只是所用的关键字不同。

[修饰符]  **interface**  接口名{

//定义成员变量

//定义成员方法

}

##### 接口中的成员

* 变量：接口中的变量都是**public,static,final**的,不论在这些变量的前面写不写这些关键字,变量一律都是公共的,静态的,最终的,都必须在声明的时候赋值。
* 方法：接口中的方法都是public,abstract的,不论在这些方法的前面写不写这些关键字,方法一律都是**公共的,抽象的（无方法体）**。

##### 接口的实现

一个父类可以有多个子类；一个子类只能有一个父类；一个类可以实现一个或者多个接口,类实现接口时使用implements这个关键字。

interface IA {…}

interface IB {…}

interface IC {…}

class A **implements** IA {…} //类A,实现了接口IA

class B **extends** A **implements** IB, IC {…}//类B继承了类A,又实现了接口IB和IC

* 一个类实现一个接口,就是要实现接口中所有的方法。
* 实现多个接口时,接口名之间用逗号隔开。
* 哪怕只有一个接口中的方法没有实现,则实现接口的类也必须定义成抽象的。
* 接口与类一样也可以继承。接口的继承也是使用关键字extends。
* 接口可以多重继承，多个接口用逗号分开，如interface ID extends IA, IB, IC {…}

interface IA {

public static final double width = 100;

public static final double height = 50;

public double getArea();

}

class B implements IA {

public double getArea()

{ return width \* height; }

}

public class InterfaceDemo {

public static void main(String[] s) {

B b = new B();**//更应写为IA b = new B();**

System.out.println("Area is "+b.getArea());

}

}

##### 接口的使用

因为接口是一种特殊的“类” ，一种特殊的“抽象类”，所以它也创建了一种数据类型。**可以用接口的变量来引用一个类的对象，就像用父类的变量来引用子类的对象一样，只要这个类实现了这个接口。**

public class InterfaceDemo {

public static void main(String[] s) {

**IA ia = new B();** //用接口IA的变量ia来引用类B的实例对象

System.out.println(“Area is ”+ia.getArea());

}

}

##### 接口的作用

* 接口是对行为的抽象，定义了各子类共同的合约协议，实现接口的类相当于保证了必然会履行这个合约(行为)。
* 接口弥补了Java类仅能单继承的局限性。类可以extends继承一个父类，并使用implements实现多个接口。同时，其他的类也可以实现这些接口，这就让我们可以为不同的需求组合出不同的继承关系。
* 接口是实现多态的重要手段。接口有无比的适用性，使用接口作为参数或返回类型，则可以传入任何实现了该接口的东西。

面向对象设计是针对接口的设计，面向对象编程是针对接口的编程。以下代码为组装国际标准电脑的代码写法。

import java.awt.Menu;

interface ICPU {

public abstract void makeCPU();//生产CPU的国际标准

}

interface IMemory {

public void makeMemory();//生产Memory的国际标准

}

// 此处省略主板、硬盘等接口定义

class Computer {

private ICPU cpu;

private IMemory memory;

public Computer(ICPU cpu, IMemory memory) {

this.cpu = cpu;

this.memory = memory;

}

public ICPU getCpu() {

return cpu;

}

public void setCpu(ICPU cpu) {

this.cpu = cpu;

}

public IMemory getMemory() {

return memory;

}

public void setMemory(IMemory memory) {

this.memory = memory;

}

public void Show()

{

System.out.println("电脑配置信息为:");

cpu.makeCPU();

memory.makeMemory();

}

}

//以上代码规定了生产一台计算机的国际标准(契约),其中还包含了CPU、内存条等电脑部件的生产规范，以上代码完全是针对接口编程。

class Intel implements ICPU {//Intel厂商生产CPU，必须遵循国际标准

public void makeCPU() {//遵循国际标准生产CPU

System.out.println("Intel品牌CPU!");

}

}

class AMD implements ICPU {//AMD厂商生产CPU，必须遵循国际标准

public void makeCPU() {//遵循国际标准生产CPU

System.out.println("AMD品牌CPU!");

}

}

class Kingston implements IMemory {//金士顿厂商生产内存，必须遵循国际标准

public void makeMemory() {//遵循国际标准生产Memory

System.out.println("金士顿品牌内存条!");

}

}

class Samsung implements IMemory {//三星厂商生产内存，必须遵循国际标准

public void makeMemory() {//遵循国际标准生产Memory

System.out.println("三星品牌内存条!");

}

}

//此处省略主板、硬盘等实现类定义

**//以上代码为各厂商遵循国际标准规范生产电脑部件,也还是针对接口编程(继承相应的接口即为遵循接口规定的标准)。**

public class Test {

public static void main(String[] args) {

//组装1台电脑

ICPU cpu=new Intel();//购买1个Intel品牌CPU

IMemory memory=new Samsung();//购买1个三星品牌内存

Computer computer=new Computer(cpu, memory);//用购买的部件组装电脑，将子类型对象传递给父类型引用

computer.Show();

//用了2年后,此电脑内存坏了,于是更换了另外一个品牌金士顿的内存

memory=new Kingston();//新买1颗金士顿内存

computer.setMemory(memory);//更换内存条

computer.Show();

}

}

在面向对象多态编程中，父类型引用可指向任何派生类型对象，故可将派生类型实例对象作为实际参数传递给父类型引用(变量)，如：

Computer computer=new Computer(**cpu, memory**);

试想以下，如果金士顿厂商生产的内存不遵循国际标准，而采取如下的生产方式会出现什么问题？

class Kingston {//金士顿厂商生产内存，未遵循国际标准

public void makeMemory() {

System.out.println("金士顿品牌内存条!");

}

}

如果不遵循国际标准生产内存条，则如下代码会出错，对应的事实就是无人购买金士顿内存条，因为其兼容性差。

memory=new Kingston();//新买1颗金士顿内存

computer.setMemory(memory);//更换内存条

### Java中的泛型(类属)类型

泛型，即“参数化类型”。在定义方法时有形式参数，然后调用此方法时传递对应类型的实际参数。而参数化类型是把参数的类型由原来的具体类型进行参数化，用一个广泛的类型标志“T”表示，“T”称为泛型变元或类型形参，然后在使用/调用时传入具体的类型(类型实参)。

class Box<T> {//定义泛型类Box,T为泛型变元

private T data;//存储的data类型未知,需要在使用时确定

public Box(T data) {

this.data = data;

}

public T getData() {

return data;

}

}

public class Test {

public static void main(String[] args) {

Box<String> b1=new Box("字符串");//使用时将T转变为String类型

Box<Integer> b2=new Box(5);//使用时将T转变为Integer类型

System.out.println(b1.getData());//b1存储一个字符串对象

System.out.println(b2.getData());//b2存储一个整型对象

}

}

泛型(类属)机制是程序设计语言中普遍注重的一种**参数多态**机制。一个类属类(泛型类)是关于一组类的一个特性抽象，它强调的是这些类的成员特征中与具体类型无关的那些部分，而用变元来表示与具体类型有关的那些部分。

## Java中List集合

List集合代表一个元素有序、可重复的集合，集合中每个元素都有其对应的顺序索引。List集合允许使用 重复元素，可以通过索引来访问指定位置的集合元素。List集合默认按元素的添加顺序设置元素的索引，例如第一次添加的元素索引为0，第二次添加的元素索引为1。

**List是一个接口类型，**其中定义的函数如下：

* void add(Object element):将元素element插入到集合的末尾。
* void add(int index,Object element):将元素element插入到集合的index位置处。
* boolean addAdd(Collection c):将集合c所包含的所有元素都插入到List集合末尾。
* boolean addAdd(int index,Collection c):将集合c所包含的所有元素都插入到List集合的index处。
* Object get(int index):返回集合index索引处的元素。
* int size():返回集合存储的实际元素个数。
* int indexOf(Object o):返回对象o在List集合中第一次出现的位置索引。
* int lastIndexOf(Object o):返回对象o在List集合中最后一次出现的位置索引。
* Object remove(int index):删除并返回index索引处的元素。
* Object set(int index,Object element)将index索引处的元素替换成element对象，返回被替换的旧元素。
* List subList(int fromlndex,int tolndex):返回从索引fromlndex(包含)到索引tolndex(不包含)处所有集合元素组成的子集合。

### ArrayList和Vector实现类

ArrayList类和Vector类是List类的两个典型实现，完全支持List接口的全部功能，可将它们简单理解为长度会自动扩充的动态数组，两种集合默认的数据存储类型为Object类型。但由于Vector性能较低，其被ArrayList淘汰。

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** Test {

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

List objList = new ArrayList();

objList.add(new Integer(1));

objList.add(new String("软件工程"));

objList.add(new Float(1.2f));

objList.add(new Double(2.5));

objList.add(0,new String("面向对象"));

**for** (**int** i = 0; i < objList.size(); i++) {

System.***out***.println(objList.get(i));

}

}

}

输出结果为：

面向对象

1

软件工程

1.2

2.5

请问以上代码中，添加到集合objList中的对象类型是否是同一种类型？“objList.add(new Integer(1));”能否改为“objList.add(1);”，为什么？

### List集合的泛型编码

2.5.1小节中，由于集合List默认存储Object类型，故编码时可向集合objList中添加任意子类型对象，这种编码方式不规范，也容易让人混淆(感觉一个集合同时存储了多种数据类型数据)或出错，可结合泛型类型将List集合存储的数据类型指定为某一固定类型。

import java.util.ArrayList;

import java.util.List;

public class Test {

public static void main(String[] args) {

//规定list只能存储String类型数据,将泛型变元T转变成String类型

List<String> list = new ArrayList<String>();

list.add("How ");

list.add("are ");

list.add("you?");

//list.add(5);//不能存储整型数据

for (String s : list) {

System.out.print(s);

}

}

}

以上代码中，“List<String> list = new ArrayList<String>();” 将泛型变元T转变成String类型，规定list只能存储String类型数据，故 “list.add(5);”错误。

## 函数的缺省参数

在定义函数时，某些形式参数可设置一个默认值，如果实际参数未传递数据给该形参，则形参取默认值，否则取实参传入的值。

#include<iostream>

using namespace std;

void **print(int a, int b = 2, int c = 3)**

{//b、c分别取默认值(缺省值)2、3

cout << a << " " << b << " " << c << endl;

}

void main()

{

//print();//出错

print(1);//1传给形参a

print(1, 3);//分别传值给形参a,b

print(1, 3, 4);//分别传值给形参a,b,c

}

输出结果：

1 2 3

1 3 3

1 3 4

以上代码为C++代码代码，Java不支持函数的缺省参数值。需要注意的是，一个函数的形式参数列表中，如一部分有缺省参数值，一部分没有，则**无缺省参数值的形参必须防止于有缺省值参数的左边**。

## this作为函数实参

在Java语言中，实例级别的函数成员有一个this变量(引用)指向该函数所属的对象，于是this有如下的用途：

* 在该函数中使用this可访问函数所属对象的所有成员。
* 若在该函数中需要其所属对象作为实际参数传递到另外一个函数，则实际参数使用this。

class ClassForPrint {

public void print(**Student stu**) {

System.out.println("学号为:" + stu.no);

}

}

class Student {

String no;

private ClassForPrint cfp;//ClassForPrint类对象作为Student类对象成员

public String getNo() {

return this.no;//通过this访问当前类成员

}

public void setNo(String no) {

this.no = no; //通过this访问当前类成员

}

public void print()

{

cfp=new ClassForPrint();

cfp.print(**this**);//通过this将当前对象传递给cfp的print函数

}

}

public class Test {

public static void main(String[] args) {

Student stu=new Student();

stu.setNo("100");

stu.print();

}

}

输出结果:

学号为:100

在类Student中的print函数中，“cfp.print(**this**);”将this作为实际参数传递给cfp对象的print函数。

**问题:**在ClassForPrint类中，为什么可以通过stu.no直接访问类Student对象的no成员?

同样的功能也可采取如下代码写法，ClassForPrint类对象cfp并非作为Student类对象成员，而是作为Student类对象中封装的函数print的形参，然后在对象stu的print中将通过this将stu它自己传递给cfp的函数print，从而实现输出功能。

class ClassForPrint {

public void print(Student stu) {

System.out.println("学号为:" + stu.no);

}

}

class Student {

String no;

public String getNo() {

return this.no;

}

public void setNo(String no) {

this.no = no;

}

//ClassForPrint类对象作为print函数的形参

public void print(**ClassForPrint cfp**)

{

cfp.print(**this**);//通过this将当前对象传递给cfp的print函数

}

}

public class Test {

public static void main(String[] args) {

ClassForPrint cfp=new ClassForPrint();

Student stu=new Student();

stu.setNo("100");

stu.print(cfp);//将对象cfp传入print中,然后在该print中调用cfp对象中的print函数

}

}

比特学员必须牢牢掌握以上两种代码写法，软件设计师下午试题六常考！

## Java中的异常处理

### 异常和异常处理

* Java中的异常是用来处理在运行时发生的异常情况的，比如数组出界、被零除、输入输出错误、堆栈溢出等。
* 与异常有关的关键字有五个：try，catch，throw，throws，finally。
* 与异常处理有关的类主要有Exception和Error，它们都是Throwable的子类。
* 处理异常使用try-catch-finally结构。

try

{ //可能产生异常的代码 }

catch(someException e)

{ //处理异常的代码 }

……

catch(anotherException e)

{ //处理异常的代码 }

finally

{ //必须执行的代码 }

以上代码中,先执行try中的代码,如果发生异常,则某个catch分支捕获相应的异常进行处理,最后执行finally代码块。如果不发生异常,则catch块中的代码不会被执行；不论是否发生异常finally块中的代码一定会被执行。

public class ExceptionDemo {

public static void main(String[] s) {

try {

int d = 0;

d = 100 / d;//产生异常

System.out.println("Try部分执行完毕,完全没有问题!");

} catch (Exception e) {

System.out.println("被零除了!");

} finally {

System.out.println("程序运行结束!");

}

}

}

输出结果：

被零除了!

程序运行结束!

在执行try代码块的“d=100/d;”语句时发生了除数为0异常，此异常被catch部分捕获后执行catch部分代码，而try代码块中产生异常的“d=100/d;”语句后的代码将无机会执行；执行完catch代码块后，执行finally代码块，最后程序执行完毕。

### 自定义异常

自定义异常步骤如下:

#### 创建一个类，必须继承Exception类。

#### 在可能抛出自定义异常方法的方法头中声明该异常。

#### 在方法体中指明需要抛出自定义异常对象的位置。

* 创建异常对象。
* 使用throw子句抛出异常。

案例:编写一个程序，在命令行输入一个学生的成绩，输出相应的等级。

class NumberRangeException extends Exception {

NumberRangeException(String msg) {

super(msg);

}

}

public class ExceptionDemo {

public static void show(int score) throws NumberRangeException {//使用throws声明异常

if (score < 0 || score > 100) {

NumberRangeException e = new NumberRangeException(

"输入的数字不在指定的范围！请重新输入。");

throw e; //使用throw抛出异常

} else {

int s = score / 10;

switch (s) {

case 10:

case 9:

System.out.println("A");

break;

case 8:

System.out.println("B");

break;

case 7:

System.out.println("C");

break;

case 6:

System.out.println("D");

break;

default:

System.out.println("E");

}

}

}

public static void main(String[] args) {

int score = -40;

try {

show(score);

} catch (NumberRangeException e) {

System.out.println(e.getMessage());

}

}

}

输出结果:

输入的数字不在指定的范围！请重新输入。

在自定义异常中,关键字throw用来实现“抛出”异常这个动作，关键字throws用于相应的方法声明异常。