**数组和对象的使用-数组-1**

如何写一个程序计算用户输入数字的平均数？

**package** javaArray;

**import** java.util.Scanner;

**public** **class** ArrayToCacularNumber {

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

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

Scanner in = **new** Scanner(System.***in***);

**int** x;

**int** cnt = 0;

**double** sum = 0;

x = in.nextInt();

**while**(x!=-1)

{

sum+=x;

cnt++;

x = in.nextInt();

}

**if**(cnt>0)

{

System.***out***.println(sum/cnt);

}

}

}

不需要记录每一个输入的数字。

如何写一个程序计算用户输入的数字的平均数，并输出输入数字中所有大于平均数的数？

必须先记录每一个输入的数字，计算平均数之后，再检查记录下来的每一个数字，与平均数比较，再考虑是否输出。

**package** javaArray;

**import** java.util.Scanner;

**public** **class** ArrayToCacularNumber {

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

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

Scanner in = **new** Scanner(System.***in***);

**int** x; 定义数组

**int**[] numbers = **new** **int**[100];

**int** cnt = 0;

**double** sum = 0;

x = in.nextInt();

**while**(x!=-1)

{ 对数组中的元素赋值

numbers[cnt]=x;

sum+=x;

cnt++;

x = in.nextInt();

}

**if**(cnt>0)

{

**for**(**int** i=0;i<cnt;i++)

{

**double** average = sum/cnt;

**if**(numbers[i]>average)

{ 使用数组中的元素

System.***out***.print(numbers[i]+" ");

}

}

System.***out***.println("|"+sum/cnt);

}

}

}

**1.1用类制造对象**

**1>对象与类**

* **对象是实体，需要被创建，可以为我们做事情**
* **类是规范，根据类的定义来创建对象**
* **对象（这只猫）**

表达东西或事件

运行时响应消息（提供服务）

* **类（猫）**

定义所有猫的属性

就是Java中的类型

可以用来定义变量

类定义了对象，对象是。。。的实体类

* **对象=属性+服务**

数据：属性或状态

操作：函数

把数据和对数据的操作放在一起————封装

* **OOP特性**

1. 一切都是对象
2. 程序就是一堆互相发送消息的对象
3. 每个对象有自己的存储空间，里面是其他的对象
4. 每个对象都有一个类型
5. 所有属于某个特定的对象可以提供相同的服务

* **自动售货机**

1. 它有什么？ Balance（余额） price total（总额）

Print balance show prompt（显示提示） insert money get food

**package** vendingmachine;

**public** **class** VendingMachine {

**int** price = 50;

**int** balance;

**int** total;

**void** showPrompt()

{

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

}

**void** insertMoney(**int** amount)

{

balance = balance+amount;

}

**void** showBalance()

{

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

}

**void** getFood()

{

**if**(balance >= price)

{

System.***out***.println("Here you are!");

balance = balance-price;

total = total+price;

}

}

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

VendingMachine vm = **new** VendingMachine();

vm.showPrompt();

vm.showBalance();

vm.insertMoney(100);

vm.getFood();

vm.showBalance();

}

}

* **创建对象**

new VendingMachine();

VendingMachine vm = new VendingMachine();

对象变量是对象的管理者

* **让对象做事**

**.**运算符

v.insertMoney(10);

v.getFood();

* **成员变量**

类定义了对象中所具有的变量，这些变量称作成员变量

每个对象有自己的变量，和同一个类的其他对象是分开的

* **函数与成员变量**

在函数中可以直接写成员变量的名字来访问成员变量

那么究竟访问的是哪个对象的呢？

函数是通过对象来调用的

v.insertMoney();

这次调用临时建立了insertMoney()和v之间的关系，让insertMoney()内部的成员变量指的是v的成员变量

* **this**

this是成员函数的一个特殊的固有的本地变量，它表达了调用这个函数的那个对象

* **调用函数**

通过.运算符调用某个对象的函数

在成员函数内部直接调用自己（this）的其他函数

* **本地变量**

定义在函数内部的变量是本地变量

定义在函数外部的变量是成员变量

本地变量的生存期和作用域都是函数内部

成员变量的生存期是对象的生存期，作用域是类内部的成员变量

我爱你，我的大宝贝，嘻嘻

* **成员变量定义初始化**

成员变量在定义的地方就可以给出初始值

没有给出初始值的成员变量会自动获得0值

对象变量的0值表示没有管理任何对象，也可以主动给null值

定义初始化可以调用函数，甚至可以使用已定义的成员变量

* **构造函数**

如果有一个成员函数的名字和类的名字完全相同，则在创建这个类的每一个对象的时候会自动调用这个函数—>构造函数

这个函数不能有返回类型

* **函数重载**

一个类可以有多个构造函数，只要他们的参数表不同

创建对象的时候给出不同的参数值，就会自动调用不同的构造函数

通过this()还可以调用其他构造函数

一个类里的同名但参数表不同的函数构成了重载关系

* **如何识别对象**

**public** **class** Display {

**private** **int** value = 0;

**private** **int** limit = 0;

**public** Display(**int** limit) {

**this**.limit = limit;

}

**public** **void** increase() {

value++;

**if**(value == limit){

value = 0;

}

}

**public** **int** getValue() {

**return** value;

}

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

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

Display d = **new** Display(24);

**for**(;;) {

d.increase();

System.***out***.println(d.getValue());

}

}

}

**public** **class** Clock {

**private** Display hour = **new** Display(24);

**private** Display minite = **new** Display(60);

**public** **void** start() {

**while**(**true**) {

minite.increase();

**if**(minite.getValue()==0) {

hour.increase();

} System.***out***.printf("%02d:%02d\n",hour.getValue(),minite.getValue());

}

}

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

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

Clock clock = **new** Clock();

clock.start();

}

}

* **private**

只有这个类内部可以访问

类内部指类的成员函数和定义初始化

这个限制是对类的而不是对对象的

* **public**

任何人都可以访问

任何人指的是在任何类的函数或定义初始化中可以使用

使用指的是调用、访问或定义变量

* **包**

在使用其他包内的类之前需要引入包

**import** display.led.Display;

* **Static**

它是类变量，静态类，只能通过类来调用或者访问它。

它不属于任何函数，只属于类。

Static函数只能调用Static函数只能访问Static的成员变量

Static函数和Static的成员变量都可以通过类的名字取访问也可以通过某个对象去访问

**3.1顺序容器-1**

* **记事本**

能存储记录

不限制能存储的记录的数量

能知道已经存储的记录的数量

能查看存进去的每一条记录

能删除一条记录

能列出所有的记录

* **接口设计**

add(String note);

getSize();

getNote(int index);

removeNote(int index);

list();

* **泛型类**

**private** ArrayList<String> notes = **new** ArrayList<String>();

* **容器类（它属于泛型类）**

ArrayList<String> notes = **new** ArrayList<String>();

容器类有两个类型：

1.容器的类型

2.元素的类型

**3.2对象数组-1**

* **对象数组中的每一个元素都是对象的管理者而非对象本身**

**3.2对象数组-2**

For-each循环

ArrayList<String> a = **new** ArrayList<String>();

a.add("first");

a.add("Second");

**for**(String s : a){

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

}

**3.3集合容器**

ArrayList<String> a = **new** ArrayList<String>();

a.add("first");

a.add("Second");

a.add("first");

**for**(String s : a){

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

}

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

HashSet<String> s = **new** HashSet<String>();

s.add("first");

s.add("Second");

s.add("first");

**for**(String k : s){

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

}

运行结果：

first

Second

first

------------

Second

first

ArrayList<String> a = **new** ArrayList<String>();

a.add("first");

a.add("Second");

a.add("first");

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

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

HashSet<String> s = **new** HashSet<String>();

s.add("first");

s.add("Second");

s.add("first");

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

运行结果：[first, Second, first]

------------

[Second, first]

**3.4 HASH表**

* **在存放相同key值时，hash表只保留最后一次输入的key值**

**public** **class** Coin {

**private** HashMap<Integer,String> coinnames = **new** HashMap<Integer,String>();

**public** Coin() {

coinnames.put(1, "peney");

coinnames.put(10, "dime");

coinnames.put(25, "quarter");

coinnames.put(50, "五毛");

coinnames.put(50, "half-dollar");

System.***out***.println(coinnames.keySet().size());

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

**for**(Integer k : coinnames.keySet()){

String s = coinnames.get(k);

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

}

}

**public** String getName(**int** amount){

**if**(coinnames.containsKey(amount)){

**return** coinnames.get(amount);

}

**else**{

**return** "NOT　FOUND";

}

}

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

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

Scanner in = **new** Scanner(System.***in***);

**int** amount = in.nextInt();

Coin coin = **new** Coin();

String name = coin.getName(amount);

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

}

}

运行结果：

1

4

{1=peney, 50=half-dollar, 25=quarter, 10=dime}

peney

half-dollar

quarter

dime

peney

**4.1继承-1、-2**

Database

DVD

CD

Item

**4.2子类父类关系-1、-2**

**Database.java**

**public** **class** Item {

**private** String title;

**private** **int** playingTime;

**private** **boolean** gotIt=**false**;

**private** String comment;

**public** Item(String title, **int** playingTime, **boolean** gotIt, String comment) {

**super**();

**this**.title = title;

**this**.playingTime = playingTime;

**this**.gotIt = gotIt;

**this**.comment = comment;

}

**public** Item(){

}

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

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

}

}

**DVD.java**

**public** **class** DVD **extends** Item {

**private** String director;

// private String title;

// private int playingTime;

// private boolean gotIt=false;

// private String comment;

**public** DVD(String title, String director, **int** playingTime, String comment) {

**super**(title,playingTime,**false**,comment);

**this**.director = director;

// this.title = title;

// this.playingTime = playingTime;

// this.comment = comment;

}

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

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

// byte a=1;

// byte b=1;

// int c=a+b;

//// byte b=a+1;

//// a+=1;

//

// System.out.println(c);

}

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

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

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

**super**.print();

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

}

}

**CD.java**

**public** **class** CD **extends** Item {

**private** String artist;

**private** **int** numOfTracks;

// private int playingTime;

// private boolean gotIt=false;

// private String comment;

//

**public** CD(String title, String artist, **int** numOfTracks, **int** playingTime, String comment) {

**super**(title,playingTime,**false**,comment);

**this**.artist = artist;

**this**.numOfTracks = numOfTracks;

// this.title = title;

// this.playingTime = playingTime;

// this.comment = comment;

}

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

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

}

// public void print() {

// System.out.println("CD:"+"《"+title+"》"+" is be played by "+artist);

//

// }

}

**Item.java**

**public** **class** Item {

**private** String title;

**private** **int** playingTime;

**private** **boolean** gotIt=**false**;

**private** String comment;

**public** Item(String title, **int** playingTime, **boolean** gotIt, String comment) {

**super**();

**this**.title = title;

**this**.playingTime = playingTime;

**this**.gotIt = gotIt;

**this**.comment = comment;

}

**public** Item(){

}

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

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

}

}

**5.1多态**

* **函数调用的绑定**

当通过对象变量调用函数的时候，调用哪个函数这件事情叫做绑定

静态绑定：根据变量的声明类型来决定

动态绑定：根据变量的动态类型来决定

在成员函数中调用其他成员函数也是通过this这个对象变量来调用的

* **覆盖override**

子类和父类中存在名称和参数表完全相同的函数，这一对函数构成覆盖关系

通过父类的变量调用存在覆盖关系的函数时，会调用变量当时所管理的对象所属的类的函数

你是一个shape，你就会draw，那么，你就去draw吧

**5.2多态变量和向上造型-1**

* **子类和子类型**

类定义了类型

子类定义了子类型

子类的对象可以被当作父类的对象来使用

赋值给父类的变量

传递给需要父类对象的函数

放进存放父类对象的容器里

* **子类型与赋值**

子类的对象可以赋值给父类的变量

Bicycle

Car

Vehicle

Vehicle v1 = new Vehicle();

Vehicle v2 = new Car();

Vehicle v3 = new Bicycle();

* **子类和参数传递**

子类的对象可以传递给需要父类对象的函数

public class Database

{

Public void add(Item theItem)

{

}

}

DVD dvd = new DVD(…);

CD cd = new CD(…);

Database.add(dvd);

Database.add(cd);

* **子类型和容器**

子类的对象可以放在父类对象的容器里

* **多态变量**

Java的对象变量是多态的，它们能保存不止一种类型的对象

它们可以保存的是声明类型的对象，或声明类型的子类的对象

当把子类的对象赋给父类的变量的时候，就发生了向上造型

**5.2多态变量和向上造型-2**

* **造型cast**

子类的对象可以赋值给父类的变量

注意! Java中不存在对象对对象的赋值！！

父类的对象不能赋值给子类的变量！

Vechilce v;

Car c = new Car();

v = c; //可以

c = v; 编译错误！

可以用造型：

c = (Car) v;

(只有当v这个变量实际管理的是Car才行)

* **造型（有点类似于强制类型转换，区别就是造型并没有改变写什么）**

用括号围起类型放在值的前面

对象本身并没有发生任何变化

所以不是“类型转换”

运行时有机制来检查这样的转化是否合理

ClassCastException

* **向上造型**

拿一个子类的对象，当作父类的对象来用

向上造型是默认的，不需要运算符

向上造型总是安全的

**5.3类型系统-1（OBJECT类）**

* **Object类**

所有的类都是继承自Object的

Car

Bicycle

Vehicle

Person

String

Object

* **Object类的函数**

toString();

equals();

**5.3类型系统-2（DOME的新媒体类型）**

在原有的程序上添加一个新的子类

**6.1城堡游戏**

**Game.java**

**package** castle;

**import** java.util.Scanner;

**public** **class** Game {

**private** Room currentRoom;

**public** Game()

{

createRooms();

}

**private** **void** createRooms()

{

Room outside, lobby, pub, study, bedroom;

// 制造房间

outside = **new** Room("城堡外");

lobby = **new** Room("大堂");

pub = **new** Room("小酒吧");

study = **new** Room("书房");

bedroom = **new** Room("卧室");

// 初始化房间的出口

outside.setExits(**null**, lobby, study, pub);

lobby.setExits(**null**, **null**, **null**, outside);

pub.setExits(**null**, outside, **null**, **null**);

study.setExits(outside, bedroom, **null**, **null**);

bedroom.setExits(**null**, **null**, **null**, study);

currentRoom = outside; // 从城堡门外开始

}

**private** **void** printWelcome() {

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

System.***out***.println("欢迎来到城堡！");

System.***out***.println("这是一个超级无聊的游戏。");

System.***out***.println("如果需要帮助，请输入 'help' 。");

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

System.***out***.println("现在你在" + currentRoom);

System.***out***.print("出口有：");

**if**(currentRoom.northExit != **null**)

System.***out***.print("north ");

**if**(currentRoom.eastExit != **null**)

System.***out***.print("east ");

**if**(currentRoom.southExit != **null**)

System.***out***.print("south ");

**if**(currentRoom.westExit != **null**)

System.***out***.print("west ");

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

}

// 以下为用户命令

**private** **void** printHelp()

{

System.***out***.print("迷路了吗？你可以做的命令有：go bye help");

System.***out***.println("如：\tgo east");

}

**private** **void** goRoom(String direction)

{

Room nextRoom = **null**;

**if**(direction.equals("north")) {

nextRoom = currentRoom.northExit;

}

**if**(direction.equals("east")) {

nextRoom = currentRoom.eastExit;

}

**if**(direction.equals("south")) {

nextRoom = currentRoom.southExit;

}

**if**(direction.equals("west")) {

nextRoom = currentRoom.westExit;

}

**if** (nextRoom == **null**) {

System.***out***.println("那里没有门！");

}

**else** {

currentRoom = nextRoom;

System.***out***.println("你在" + currentRoom);

System.***out***.print("出口有: ");

**if**(currentRoom.northExit != **null**)

System.***out***.print("north ");

**if**(currentRoom.eastExit != **null**)

System.***out***.print("east ");

**if**(currentRoom.southExit != **null**)

System.***out***.print("south ");

**if**(currentRoom.westExit != **null**)

System.***out***.print("west ");

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

}

}

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

Scanner in = **new** Scanner(System.***in***);

Game game = **new** Game();

game.printWelcome();

**while** ( **true** ) {

String line = in.nextLine();

String[] words = line.split(" ");

**if** ( words[0].equals("help") ) {

game.printHelp();

} **else** **if** (words[0].equals("go") ) {

game.goRoom(words[1]);

} **else** **if** ( words[0].equals("bye") ) {

**break**;

}

}

System.***out***.println("感谢您的光临。再见！");

in.close();

}

}

**Room.java**

**package** castle;

**public** **class** Room {

**public** String description;

**public** Room northExit;

**public** Room southExit;

**public** Room eastExit;

**public** Room westExit;

**public** Room(String description)

{

**this**.description = description;

}

**public** **void** setExits(Room north, Room east, Room south, Room west)

{

**if**(north != **null**)

northExit = north;

**if**(east != **null**)

eastExit = east;

**if**(south != **null**)

southExit = south;

**if**(west != **null**)

westExit = west;

}

@Override

**public** String toString()

{

**return** description;

}

}

**6.2消除代码复制**

* 代码复制是不良设计的一种表现

**6.3封装**

* 增加可扩展性

可以运行的代码！=良好的代码

对代码做维护的时候最能看出代码的质量

如果想要增加一个方向，如down或up

* 用封装来降低耦合

Room类和Game类都有大量的代码和出口相关

尤其是Game类中大量使用了Room类的成员变量

类和类之间的关系称作耦合

耦合越低越好，保持距离是形成良好代码的关键

**6.4可扩展性**

* 用接口来实现聚合

给Room类实现的新方法，把方向的细节彻底隐藏在Room类的内部了

今后方向如何实现就和外部无关了

* 用容器来实现灵活性

Room的方向是用成员变量表示的，增加或减少方向就要改变代码

如果用hush表来表示方向，那么方向就不是“硬编码”的了

**Room.java**

**package** castle;

**import** java.util.HashMap;

**public** **class** Room {

**private** String description;

**private** HashMap<String,Room> exits = **new** HashMap<String,Room>();

**public** Room(String description)

{

**this**.description = description;

}

**public** **void** setExit(String dir,Room room){

exits.put(dir, room);

}

@Override

**public** String toString()

{

**return** description;

}

**public** String getExitDesc(){

StringBuffer sb = **new** StringBuffer();

**for**(String dir : exits.keySet()){

sb.append(dir);

sb.append(" ");

}

**return** sb.toString();

}

**public** Room getExit(String direction){

**return** exits.get(direction);

}

}

**Game.java**

**package** castle;

**import** java.util.Scanner;

**public** **class** Game {

**private** Room currentRoom;

**public** Game()

{

createRooms();

}

**private** **void** createRooms()

{

Room outside, lobby, pub, study, bedroom;

// 制造房间

outside = **new** Room("城堡外");

lobby = **new** Room("大堂");

pub = **new** Room("小酒吧");

study = **new** Room("书房");

bedroom = **new** Room("卧室");

// 初始化房间的出口

outside.setExit("east", lobby);

outside.setExit("south", study);

outside.setExit("west", pub);

lobby.setExit("west", outside);

pub.setExit("east", outside);

study.setExit("north", outside);

study.setExit("east", bedroom);

bedroom.setExit("west", study);

lobby.setExit("up", pub);

pub.setExit("down", lobby);

currentRoom = outside; // 从城堡门外开始

}

**private** **void** printWelcome() {

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

System.***out***.println("欢迎来到城堡！");

System.***out***.println("这是一个超级无聊的游戏。");

System.***out***.println("如果需要帮助，请输入 'help' 。");

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

showPrompt();

}

// 以下为用户命令

**private** **void** printHelp()

{

System.***out***.print("迷路了吗？你可以做的命令有：go bye help");

System.***out***.println("如：\tgo east");

}

**private** **void** goRoom(String direction)

{

Room nextRoom = currentRoom.getExit(direction);

**if** (nextRoom == **null**) {

System.***out***.println("那里没有门！");

}

**else** {

currentRoom = nextRoom;

showPrompt();

}

}

**public** **void** showPrompt(){

System.***out***.println("你在" + currentRoom);

System.***out***.print("出口有: ");

System.***out***.println(currentRoom.getExitDesc());

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

}

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

Scanner in = **new** Scanner(System.***in***);

Game game = **new** Game();

game.printWelcome();

**while** ( **true** ) {

String line = in.nextLine();

String[] words = line.split(" ");

**if** ( words[0].equals("help") ) {

game.printHelp();

} **else** **if** (words[0].equals("go") ) {

game.goRoom(words[1]);

} **else** **if** ( words[0].equals("bye") ) {

**break**;

}

}

System.***out***.println("感谢您的光临。再见！");

in.close();

}

}

**6.5框架加数据**

* 以框架+数据来提高可扩展性

命令的解析是否可以脱离if-else

定义一个Handler来处理命令

用Hash表来保存命令和Handler之间的关系

**7.1抽象**

* Shape是什么形状？

Shape类的draw()函数应该如何写？

center

Shape

Square

Circle

Ellipse

Rectangle

move()

draw()

draw() draw()

draw() draw()

* 抽象函数/抽象类

抽象函数：表达概念而无法实现具体代码的函数

抽象类：表达概念而无法构造出实体的类

带有abstract修饰符的函数

有抽象函数的类一定是抽象类

抽象类不能制造对象

但是可以定义变量

任何继承了抽象类的非抽象类的对象可以付给这个变量

* 实现抽象函数

继承自抽象类的子类必须覆盖父类中的抽象函数

否则自己成为抽象类

* 两种抽象

与具体相对

表示一种概念而非实体

与细节相对

表示在一定程度上忽略细节而着眼大局

**7.2数据与表现分离：细胞自动机-1**

* 细胞自动机

死亡：如果活着的邻居的数量<2或>3，则死亡

新生：如果正好有3个邻居活着，则新生

其他情况则保持原状

**7.2数据与表现分离：细胞自动机-2**

* 程序的业务逻辑与表现无关
* 表现可以是图形也可以是文本的
* 表现可以是当地的也可以是远程的

View和Field的关系

* 表现与数据的关系
* View只管根据Field画出图形
* Field只管数据的存放
* 一旦数据更新以后，通知View重新画出整个画面
* 不去精心设计哪个局部需要更新
* 这样简化了程序逻辑
* 是在计算机运算速度提高的基础上实现的

责任驱动的设计

* 将程序要实现的功能分配到合适的类/对象中去是设计中非常重要的一环

网格化

* 图形界面本身有更高的解析度
* 但是将画面网格化以后，数据就更容易处理了

**7.3接口：狐狸与兔子**

Cell类的地位很尴尬

* 在Cell程序中它表达了细胞
* 但是同时它也表达了放在网格中的一个格子
* Fox和Rabbit是否应该从Cell继承？

**7.3接口：狐狸和兔子-2**

接口

* 接口是纯抽象类
* 所有的成员函数都是抽象函数
* 所有的成员变量都是public static final
* 接口规定了张什么样，但是不管里面有什么

**7.3 接口：狐狸和兔子-3（接口设计模式）**

实现接口

* 类用extends，接口用implements
* 类可以实现很多接口
* 接口可以继承接口，但不能继承类
* 接口不能实现接口

面向接口的编程方式

* 设计程序时先定义接口，再实现类
* 任何需要在函数间传入传出的一定是接口而不是具体的类
* 是Java成功的关键之一，因为极适合多人同时写一个大程序
* 也是Java被批评的要点之一，因为代码量膨胀起来很快

Cell和Field的关系

* Cell在Field中，但是Cell的很多操作需要Field的数据

方法一：

让每个Cell有一个Field的管理者（Cell知道Field）

方法二：

由外部第三方来建立两者之间的关系（Cell不知道Field）

**8.1控制反转-1（布局管理器）**

JButton btnStep = **new** JButton("单步"); frame.add(btnStep,BorderLayout.***NORTH***);

**8.1控制反转-2（布局管理器）**

btnStep.addActionListener(**new** ActionListener(){

@Override

**public** **void** actionPerformed(ActionEvent e) {

System.***out***.println("按下啦！！！");

step();

frame.repaint();

}

});

**8.1控制反转-3（内部类）**

* 内部类的好处:可以直接访问类里面的所有的成员函数
* 匿名类

在new对象的时候给出的类的定义形成了匿名类

匿名类可以继承某类，也可以实现某接口

Swing的消息机制广泛使用匿名类

* 内部类

定义在别的类内部、函数内部的类

内部类能直接访问外部的全部资源

包括任何私有的成员

外部是函数是，只能访问那个函数里final的变量

* 注入反转

由按钮公布一个守听者接口和一对注册/注销函数

你的代码实现那个接口，将守听者对象注册在按钮上

一旦按钮被按下，就会反过来调用你的守听者对象的某个函数

8.2 MVC设计模式-1（JTABLE）