ganbin@tarena.com.cn

《Core Java 课件》

Day 01

一、 从面向过程编程到面向对象编程的思维转变

我们知道所有的计算机程序都是由两类元素组成：代码和数据。此外从概念上将讲，程序还可以以他的代码或是数据为核心进行组织编写。也就是说一些程序围绕“正在发生什么编写“，而另一些程序则围绕”谁将被影响“编写。这两种范型决定程序的构建方法。第一种方法被称为面向过程的模型，用他编写的程序都具有线性执行的特点。面向过程的模型可以认为是代码作用于数据，用 C 写的程序就是典型的面向过程模型。第二种方法也就是我们现在正在学习的面向对象编程，面向对象编程围绕她的数据（即对象）和为这个数据严格定义的接口来组织程序。面向对象的程序实际上就是用数据控制对代码的访问。CoreJava 就是一门纯面向对象编程的语言。

学习方法很简单，就是模仿、改进、创新，循环往复。

二、 什么是字节码和虚拟机：

字节码是一套设计用来在 Java 运行时系统下执行的高度优化的指令集。该 Java 运行时系统称为 Java 虚拟机(JVM)。JVM 其实就是一个字节码解释器。虚拟机将字节码解释成机器码给 CPU 执行，所以我们在 java 中通过虚拟机的这种解释执行方式来屏蔽底层操作系统的差异。

JRE = JVM+编译器

JDK= JVM+编译器+类库

查看类库源码在：JDK/src.zip 压缩包里

三、 环境变量的设置：

需要新加的两个环境变量

1、JAVA_HOME:指名 JDK 的位置。

2、CLASSPATH:指名到哪里去找运行时需要用到的类代码（字节码）

原有的环境变量

1、 PATH:指名可执行程序的位置。

2、 EXPORT :将指定的环境变量声明为全局的。

我们在.bash_profile 下的环境变量设置。

JAVA_HOME=/opt/jdk1.5

CLASSPATH=.

PATH＝$PATH:$JAVA_HOME/bin:.

注： “.”代表当前目录

当我们把环境变量都配置好了以后在终端敲入”java -version”命令如果出现 JDK 版本号信息就表示我们环境变量设置成功了。

Bin 目录下放的是一些 JDK 命令

四、 kate 工具的使用

这个就不多做描述了，大家多试试就清楚了

五、 我们的第一个 Java 程序

```
public class MyFirstJava{
    public static void main(String[] args){
        System.out.println(“Hello World”);
    }
}
```

注：

1、我们要求类名必须和文件名一致，只不过文件名多了个.java 的后坠。

2、main 函数是我们整个程序的执行入口所以必须是静态公开的。

编译：javac MyFirstJava.java

编译后我们可以看到目录下多了一个 MyFirstJava.class 文件。这就是 Java 编译原文件后生成的字节码文件。

执行：java MyFirstJava

将字节码文件交给 JVM 去解释执行。

思考：

1、为什么编译后不生成可执行文件（注：所有的可执行文件都是机器代码）

2、Java 的跨平台特性(SUM 口号：一次编译到处运行)；

六、 包结构

为了根据需要将不同的原文件放在不同的目录下我们引入了包。包我们可以看作就是一个存放 java 原文件的目录。

在源码的基础上声明一个包名：package sd0604

加包后我们用“javac –d . MyFirstJava.java”编译后生成的字节码文件就会放在我们指定的包（目录）结构下.

如果我们想指定多级目录那么在目录明之间我们可以用.作为分隔符。

例如：package sd0604.najing.xuanwu

七、 CoreJava 中的注释

1、//单行注释

2、/* */多行注释

3、/** */文档注释

文档注释可以由 java doc 命令单独提取出来生成注释文档。

例：javadoc MyFirstJava.java

我们在生成的注释文件中可以打开 index 入口页面来查看我们刚刚生成的注释文档。

文档注释一般写在类、方法、属性定义之前

前两种注释和 C++相同。

注意：1、我们可以用 javadoc –help 命令来查看该命令的其他

用法。

2、同样我们可以用“javadoc –d 路径名 *.java”来指定生成文档注释的位置。

3、可以在文档注释中加入 HTML 标签来控制生成注释文档的显示格式。

八、 jar 命令的用法

我们可以用“jar –cvf m n”命令来将文件打包成 jar 压缩包。

m: 要生成 jar 包的名字

n: 要压缩文件的文件名（可以是多个文件或一个目录）

生成的 jar 文件实际上就是一个普通的 zip 压缩文件

顺便说一下解包的命令：unzip *.jar

九、计算机运行一个 java 程序的过程

1、 启动 JVM

2、 通过 JAVA_HOME\jre\lib 目录下找到对应的类

3、 如果的 2 不找不到则在环境变量中配置的 CLASSPATH 配置的路径中找类，这就是为什么我们要在 CLASSPATH 中配上当前路径的原因。

十、import 声明的作用

当我们要用一个 JDK 定义好的类时我们需要在 Java 程序中配上该

类对应的 jar 包。（类似于 C++中了＃include 预处理指令）

例：import java.util.*;

这里大家注意一点：由于 java.lang.*包中的类是我们编程中经常要使用的，所以这个包下的类我们不用特别用 import 在程序中声明。但当我们要用到其他包中的类似我们就必须用 import 声明了。

awt：抽象窗口工具.

如果大家想了解什么包做什么用的话大家课以看看我给大家的 API 文档，上面有对所有这些包用法的解释。

十一、java 中的垃圾回收器

因为有了垃圾回收器，我们可以不用顾虑对象创建后占用系统资源的问题。我们只用负责对象的创建，而对象销毁和资源释放的问题就可以留给垃圾回收器做了。这里我们需要注意的是垃圾回收器一般只会在内存空间不够的情况下进行资源回收。

十二、java 中标识符的命名规则

1、 只能以字母、下划线或”$”开头，严格区分大小写，且长度不限。

2、 类名的每个单词的首字母大写

3、 方法名属性名的第一个单词小写，以后的每个单词首字母大写

4、 所有的包结构名字都是小写

5、 常量名所有字母大写

注：第一点是必须遵守的，2～5 点不遵守也不会出错，但建议大家严格遵守以上命名规范。

十三、java 中的关键字和保留字

查书

十四、java 中的八中基本数据类型

1. **boolean:占 1 个字节**
2. **byte :占 1 个字节**
3. **char : 占 2 个字节(可以用来保存汉字)**
4. **short :占 2 个字节**
5. **int :占 4 个字节**
6. **long : 占 8 个字节**
7. **float : 占 4 个字节**
8. **double : 占 8 个字节**

十五、正负数在内存空间中的存放

正数在内存空间中存发的是源码。

负数在内存空间中存发的是正数源码对应的补码。

补码：在源码的基础上取反后末位加 1。

十六、基本数据类型之间的转换

1、 正向过程：从低字节到高字节可以自动转换。

byte->short->int->long->float->double

注：boolean 不能转。

2、 逆向过程：从高字节到低字节用强制类型转换

例：int a = (int)3.12

注：逆向转换将丢失精度。

十七、java 中的转义字符

1. 表示格式控制的转义字符(如：\n \t)与 C++相同。
2. Java 中用“\u 四位十六进制的数字”表是将字符转换成对应的 unicode 编码。

十八、表达式和流程控制

instanceof()：用户判断某一个对象是否属于某一个类的实例。

运算符：单目运算符、双目运算符、位运算符

注：1、>>>是 corejava 中的位移运算符，表示右移，左边空出的位以 0 添充。>>右移

2、将一个数右移 n 位相当于将该数除以 2 的 n 次方；

3、 将一个数左移 n 位相当于将该数乘以 2 的 n 次方；

4、 &&、||是短路运算符（左边条件不符合时不会执行右边的判断）

Day02

一、流程控制语句

1、两路分支选择

If else 配对原则：else 一定会和离他最近的且没有与别的 else 配对的 if 配对；

2、多路分支选择

```
switch (byte,short,int,char){
case   xxx:
            break;
case    xxx:
            break;
default:
}
```

3、循环结构

3.1 for(初始化；条件；调整){语句块} 该结构在知道循环次数的时候使用

初始化语句只执行一次；

判断条件→执行语句块→执行调整→判断条件

for（;;）表示为无限循环

注：for 循环中两个分号是不能省略的。

3.2 While(条件){语句块}改结构在不知道循环次数的时候使用

先判断后执行，调整语句在代码块中体现。

while(1)表示为无限循化。

3.3 Do {语句块}while(条件);同 3.2；

唯一的区别是该循环语句代码块至少要被执行一次；

先执行后判断

continue:提前终止本次循环直接进入下一次循环；

3.4 java 中的循环标号

```
labe:for(int i=0;i<10;i++){
        for(int a=0;a<10;a++){
            break labe;
        }
    }
```

break labe:直接跳出同标号循环层

4、条件运算符的自动类型提升问题。

二、java 中的数组

1. 数组的定义

   分为两部分：数组引用（声明） int[] a; int a[];

   数组空间（初始化）a = new int[5];

2. 数组在分配存储空间后，系统会自动为数组的每个元素初始化为 0；

3. 定义数组，分配存储空间和初始化数组可以放在一个语句中，如：

   int[] a = {10,20,30}

   对比：int[] a = new int[3];a[0] = 10;a[1] = 20;a[2] = 30;

4. 在 java 中一个数组就是一个对象，只要是对象就是在堆空间存放。

   注：在 java 中只有堆空间，栈空间，代码空间。

5. 数组长度可以用 数组名.length 来取得.

   注：二维数组用.length 取得的长度是其一维数组的长度。

6. 两个数组之间的拷贝：System.arraycopy(a,0,b,0,length)

   表示将 a 数组从 0 号位置开始的 length 个元素依次拷贝到 b 数组中（从 0 号位置开始）。

7. 在 java 中二维数组本质上就是一维数组的数组。所以 java 中的二维数组可以是不对称的。Java 中只有一维数组的内存地址空间是连续的而二维数组的空间可以不连续。

   （1）、二维数组声明和初始化

   ```
   int[][] a;//声明一个二维数组
   a = new int[3][];//该二维数组包含三个一维数组对象
       而每个一维数组对象长度可以不同所以第二个【】中的长度可以不填
   a[0] = new int[5];//第一列数组长度为 5
   a[1] = new int[3]; //第二列数组长度为 3
   a[0][0] = 1;//将第一个数组的第一个元素初始化 1;
   ```

8. 介绍一个关于数组的小技巧

   在我们使用数组时，我们可以用一个 index 辅助变量来配合数组使用，表示其有效数据的个数，同时用来表示

数组下一个可插入位置的下标。

三、 java 中的对象

声明：Student s ;

这时我们只是说明 s 是一个能够指向 Student 类型的引用（相当与 C++中的针），并没有创建一个对象。所以我们不能对 s 做任何操作。

初始化：s = new Student();

向系统申请一块存储空间（地址空间），该地址空间保存的是一个 Student 类型的数据。而 s 中保存的就是该地址空间的首地址。

这里大家可能还是不太好理解，那么我们给变量来下一个定义

什么叫变量：变量就是内存空间中一块具有固定长度的，用来保存数据的地址空间。（s 也是一个变量）

一个对象可以有多个引用指向。

Student[] s = new Student[3]只是相当于声明一个长度为 3 的 Student 类型的数组。

四、 实例变量和局部变量

实例变量：1、在一个类中，任何方法之外定义的变量；

2、从面向对象的思想来说我们又把实例变量成为一个类的属性。

3、实例变量在没有符初值时系统会自动帮我们做初始化：整型数据初始化为 0，布尔型数据初始化为 false，对象类型初始化为 null。

局部变量：1、在方法内定义的变量叫局部变量。

2、局部变量使用前必须初始化，系统不会自动给局部变量做初始化。

3、局部变量的生命范围在他所在的代码块，在重合的作用域范围内不允许两个局部变量命名冲突。

注：局部变量与实例变量允许同名，在局部变量的作用域内，其优先级高于实例变量。我们可以用 this.实例变量名以区分局部变量。

Day03

一、 java 中的自动类型提升问题。

```
public class test1{
    public static void main(String[] args){
        byte a = 1;
        byte b = 2;
        byte c = (byte)(a+b);
        System.out.println(c);
    }
}
```

二进制是无法精确的表示 0.1 的。

进行高精度运算可以用 java.math 包中 BigDecimal 类中的方法。

自动类型提升又称作隐式类型转换。

二、在 java 中对面向对象（OO）的要求

1．对象是客观存在的，万物皆对象。

. (注：看不见的对象并不表示该对象不存在，比如说事件)；

2． 简单性：采用面向对象方法可以使系统各部分各司其职各尽所能。

3． 复用性：对象的功能越简单其可重用性越高。

4． 弱耦合性：各司其职各尽所能。

5． 高内聚性：一个对象独立完成一个功能的能力

6． 类是一类事务的共性，是人类主观认识的一种抽象，是对象的模板。

三、面向过程与面向对象的对比

面向过程：先有算法，后有数据结构。先考虑怎么做。

面向对象：先有数据结构，后有算法。先考虑用什么做。

四、java 中方法的声明（分为五个部分）

1． 方法的修饰符（可以有多个，且顺序无关）

2． 方法的返回值类型

3． 方法名

4． 方法的参数表

5． 方法允许抛出的列外（异常）

注：编译器只能做语法上的检查，而不能进行逻辑上的检查。

Java 中不允许有废话。

五、 java 中的重载(Overload)

1、相同方法名，不同参数表。

2、方法重载时，对于参数的匹配有个向上就近原则。（这样可以节省栈空间资源）；

3、为什么面向对象中要有方法重载？

方法的重载使同一类方法由于参数造成的差异对于对象的使用者是透明的。对象的使用者只负责把参数交给对象，而具体怎么实现由对象内部决定。

4、Java 中的运算符重载

java 中唯一重载的运算符是 String 类型的“+”号，任何类型+String 类型结果都为 Stirng 类型。

5、注意点：重载不仅出现在同一个类中，也可以出现在父子类中。

六．Java 中创建对象的步骤

1、 分配空间

2、 初始化属性

3、 调用构造方法

注：构造方法不能手工调用，在对象的生命周期内构造方法只

调用一次。

七、java 中的构造方法

1、 特点：没有返回值，方法名与类名相同。

2、 在不写构造方法时，系统会自动生成一个无参的构造方法。

3、 请养成在每个类中自己加上无参构造方法的习惯。

八、对象和对象引用的区别

对象好比一台电视机，对象引用好比电视机遥控。对象引用中存的是对象的地址。多个对象引用中存放的是同一个地址，表示该对象被多个对象引用所引用。

九、 this 表示当前对象

谁调用该方法，在这一时刻谁就是该方法的当前对象

用 this 来区分实例变量和局部变量。

this()表示调用本类的其他构造方法，且只能放在一个方法中的第一行第一句。

十、 参数传递

在 java 方法传参过程中简单类型是按值传递，对象类型是按引用传递。

按值传递传递的是数据的副本。

按引用传递传递的是保存该数据的地址

十一、封装

1．定义：封装指的是一个对象的内部状态对外界是透明的，对象与对象之间只关心对方有什么方法，而不关心属性。

封装使实现的改变对架构的影响最小化。

2．原则：装使对象的属性尽可能的私有，根据需要配上相应的 get/set 方法，对象的方法尽可能的公开。该隐藏的一定要隐藏，该公开的一定要公开。

3．方法公开的使声明而不是实现。使方法实现的改变对架构的影响最小化。

4．访问控制从严到宽

private　:仅本类成员可见

default　:本类＋同包类可见（默认）

protected:本类＋同包＋不同包的子类

public　:公开

注：这里的同包指的是跟父类所在的包相同。

5、完全封装：属性全部私有，并提供相应的 get/set 方法。

Day04

一、 继承

1.　定义：基于一个已存在的类构造一个新类。继承已存在的

类就是复用这些类的方法合属性，在此基础上，还可以在新类中添加一些新的方法和属性。

2. 父类到子类是从一般到特殊的关系。

3. 继承用关键字 extends

   dog extends Animal :表示狗类继承了动物类

4. Java 中只允许单继承（java 简单性的体现）

   父子类之间的关系是树状关系。（而多继承是网状关系）

5. 父类中的私有属性可以继承但是不能访问。

   也可以说父类中的私有属性子类不能继承。

6. 原则：父类放共性，子类放个性。

7. 构造方法不能被子类继承。

二、带继承关系的对象创建的过程

1. 递归的构造父类对象
2. 分配空间
3. 初始化属性
4. 调用本类的构造方法

三、super 关键字

1. Super（）表示调用父类的构造方法
2. Super（）也和 this 一样必须放在方法的第一行第一句。
3. Super.表示调用父类的方法或属性。例：super.m();

4. Super 可以屏蔽子类属性和父类属性重名时带来的冲突

5. 在子类的构造函数中如果没有指定调用父类的哪一个构造方法，那么就会调用父类的无参构造方法，即 super（）。

四、白箱复用和黑箱复用

1．白箱复用：又叫继承复用，子类会继承父类所有的东西，从某种程度上说白箱复用破坏了封装。是一种 is a 的关系。

例：class Liucy{

```
 public void teachCpp(){
     System.out.println("Teach Cpp");
 }
 public void chimogu(){
 }
}

class Huxy extends Liucy{
}
```

2、黑箱复用：又叫组合复用，是一种 has a 的关系。

例：class Liucy{

```
 public void teachCpp(){
     System.out.println("Teach Cpp");
 }
 public void chimogu(){
 }
}

class Huxy {
 private Liucy liucy = new Liucy();
 public void teachCpp(){
     liucy.teachCpp();
 }
}
```

原则：组合复用取代继承复用原则。

使我们可以有机会选择该复用的功能。

五、多态

1. 定义：所谓多态是指一个对象可以有多种形态，换句话说多态使我们可以把一个子类对象看作是一个父类对象类型（ 例：father A = new child() ） 。多态指的是编译时的类型变化，而运行时类型不变。

2. 多态分为两种：编译时多态和运行时多态。

   编译时类型：定义时类型（主观概念）把它看作什么。

   运行时类型：真实类型（客观概念） 实际上他是什么。

   重载又叫编译时多态，覆盖又称运行时多态。

   在方法重载的情况下，参数类型决定于编译时类型。

3. 多态的作用：在我们需要一类对象的共性时，我们可以很容易的抽取。并且可以屏蔽不同子类对象之间我们所不关心的差异。多态使我们有机会写出更通用的代码，以适应需求的不断变化

4. 多态常见的用法：

   (1)、多态用在方法的参数上

   (2)、多态用在方法的返回类型上

5. 运行时多态的三原则：

   （1)、对象不变（改变的是主观认识）

   （2)、对于对象的调用只能限于编译时类型的方法。

   （3)、在程序的运行时，动态类型判定。运行时调用运行时类

型，即他调用覆盖后的方法。

六、java 中的覆盖（Override）

1、 参数表、方法名必须完全相同，访问修饰符要求子类宽于父类。返回值类型在 JDK5.0 以前要求完全相同，5.0 以后可以父类返回一个对象 a,子类返回一个该对象 a 的子类也是覆盖。子类方法覆盖父类方法时要求子类方法的访问修饰符宽于或等于父类的访问修饰符。

2、 为什么面向对象中要有方法覆盖？

覆盖允许子类用自己特色的方法去替换调父类已有的方法。

3、 父类中的私有方法与子类中任何方法不够成覆盖关系，也就是说只有父类被子类继承过来的方法，才有可能与子类自己的方法构成覆盖关系。

4、少覆盖原则：如果子类覆盖了父类过多的方法，那么我们要重新思考一下这两个类之间到底是不是继承关系。

注：子类的属性和父类的属性同名时叫遮盖（区覆盖）

属性的遮盖是没有多态的。

七、关系运算符:instanceof

1. boolean c = a instanceof b;

a:对象变量；b:类名；c:逻辑型返回值。

如果可以把 a 对象看作是 b 类型，那么返回真。否则返回假。

2. instanceof 一般用于在强制类型转换之前判断对象变量是否可以强制转换为指定类型。

String [] args 命令行参数在用 java 命令运行程序时输入:

如：java TestOverLoad 参数 1 参数 2 参数 3。。。。。。

Day05

一、static 修饰符

1. 可以修饰属性、方法、初始代码块，成为类变量、静态方法、静态初始化代码块。

   注：初始代码块是在类中而不再任何方法之内的代码块。

2. 类变量、静态方法、静态初始化代码块与具体的某个对象无关，只与类相关，是全类公有的。在类加载时初始化。

3. 类加载：虚拟机通过 CLASSPATH 从磁盘上找到字节码文件，并将字节码文件中的内容通过 I/O 流读到虚拟机并保存的过程。在虚拟机的生命周期中一个类只被加载一次。

   注：Java 命令的作用是启动 JVM。

4. Static 定义的时一块为整个类共有的一块存储区域，其发生变化时访问到的数据都是经过变化的。

5. 为什么主方法必须是静态的？

主方法是整个应用程序的入口，JVM 只能通过类名去调用主方法。

6. 类变量和静态方法可以在没有对象的情况下用：类名.方法名（或属性名）来访问。

7. 静态方法不可被覆盖（允许在子类中定义同名的静态方法，但是没有多态）；父类如果是静态方法，子类不能覆盖为非静态方法。父类如果是非静态方法，子类不能覆盖为静态方法。

争论：静态方法可以覆盖但是没有多态。

思考：没有多态的覆盖叫覆盖吗？

在静态方法中不允许调用本类中的非静态成员。

8. 静态初始化代码块只在类加载的时候运行一次，以再也不执行了。所以静态代码快一般被用来初始化静态成员。

9. 不加 static 为动态初始化代码块，在创建对象时被调用（在构造函数之前）。

10. 最后要注意的一点就是 Static 不能修饰局部变量。

二、 什么时候类加载

第一次需要使用类信息时加载。

类加载的原则：延迟加载，能不加载就不加载。

触发类加载的几种情况：

（1）、调用静态方法时会加载静态方法真正所在的类。

例：通过子类调用父类的静态方法时，只会加载父类而不

会加载子类。

（2）、调用静态初始化代码块时要加载类。

（3）、加载子类时必定会先加载父类。

（4）、构造对象的时候会加载。

（5）、调用静态属性时会加载类。

注：如果静态属性有 final 修饰时，则不会加载。

例：public static final int a =123;

但是如果上面的等式右值改成表达式（且该表达式在编译时不能确定其值）时则会加载类。

例：public static final int a = math.PI

三、final 修饰符

1. final 可以用来修饰类、属性和方法。
2. final 修饰一个属性时，该属性成为常量。

（1）对于再构造方法中利用 final 进行赋值时候，此时在构造之前系统设置的默认值相对于构造方法失效。

（2）对于实例常量的赋值有两次机会

在初始化的时候通过声明赋值

在构造的时候（构造方法里）赋值

注：不能在声明时赋值一次，在构造时再赋值一次。

注意：当 final 修饰实例变量时，实例变量不会自动初始化为 0；

3. Final 修饰方法时，该方法成为一个不可覆盖的方法。这样可以保持方法的稳定性。

   如果一个方法前有修饰词 private 或 static，则系统会自动在前面加上 final。即 private 和 static 方法默认均为 final 方法。

4. Final 常常和 static、public 配合来修饰一个实例变量，表示为一个 全类公有的公开静态常量。

   例：pubic static final int a = 33;

   在这种情况下属性虽然公开了，但由于是一个静态常量所以并不算破坏类的封装。

5. Final 修饰类时，此类不可被继承，即 final 类没有子类。

   一个 final 类中的所有方法默认全是 final 方法。

   Fianl 不能修饰构造方法，构造方法不能被继承更谈不上被子类方法覆盖。

四、 关于 final 的设计模式：不变模式

1、不变模式：一个对象一旦产生就不可能再修改（string 就是典型的不变模式）；

通过不变模式可以做到对象共享；

2、池化思想：用一个存储区域来存放一些公用资源以减少存储空间的开销。

例：在 String 类中有个串池（在代码区）。

（1）如果用 Stirng str = “abc”来创建一个对象时，则系统会先

在串池中寻找有没有“abc”这个字符串。如果有则直接将对象指向串池中对应的地址，如果没有则在串池中创建一个“abc”字符串。

所以：String str1 = “abc”;

String str2 = “abc”;

Str1 = = str2 返回值是 ture;他们的地址是一样的。也就是说 str1 和 str2 都指向了代码空间中相同的一个地址，而这个地址空间保存就是是字符串”abc”;

（2）如果用 String str = new String(“abc”)则直接在堆空间开辟一块存储空间用来存放”abc”这个字符串。

所以：String str1 = new String(“abc”);

String str2 = new String(“abc”);

Str1 = = str2 返回值是 false;他们的地址是不一样的。也就是说 str1 和 str2 分别指向了堆空间中不同的两个地址，而这两个地址空间保存的都是字符串”abc”;

4、 java.lang 下的 StringBuffer 类。

对于字符串连接

String str=”1”+”2”+”3”＋”4”;

产生：

12
123
1234

会在串池中产生多余对象，而真正我们需要的只有最后

一个对象，用这种方式进行字符串连接时，不管在时间上还是在空间上都会造成相当大的浪费。所以我们应该使用 StringBuffer(线程安全的) 或者 StringBuilder（线程不安全的）

解决方案：

```
String s;
StringBuffer sb = new StringBuffer(“1”);
Sb.append(“2”);
Sb.append(“3”);
Sb.append(“4”);
S = sb.toString();
```

解决后的方案比解决前在运行的时间上相差 2 个数量级。

五、abstract 修饰符

1. 可用来修饰类、方法
2. abstract 修饰类时，则该类成为一个抽象类。抽象类不可生成对象（但可以有构造方法留给子类使用），必须被继承使用。

   抽象类可以声明，作为编译时类型，但不能作为运行时类型。

   Abstract 永远不会和 private,static,final 同时出现。
3. Abstract 修饰方法时，则该方法成为一个抽象方法，抽象方法没有实现只有定义，由子类覆盖后实现。

   比较：private void print(){};表示方法的空实现

   abstract void print();表示方法为抽象方法，没有实现
4. 抽象方法从某中意义上来说是制定了一个标准，父类并不实现，留给子类去实现。

注：抽象类中不一定要有抽象方法，但有抽象方法的类一定是抽象类。

六、 关于抽象类的设计模式：模板方法

灵活性和不变性

在下面这个例子种父类（抽象类）指定了一个标准，而子类根据自己的需求做出不同的实现。

例：/**********/

```
public class TestTemplateMethod{
    public static void main(String[] args){
        XiaoPin x1 = new ShuoShi();
        XiaoPin x2 = new DaPuKe();
        x1.act();
        x2.act();
    }
}

abstract class XiaoPin{
    abstract void jiaoliu();
    abstract void xushi();
    abstract void gaoxiao();
    abstract void shanqing();
    public final void act(){
        jiaoliu();
        xushi();
        gaoxiao();
        shanqing();
    }
}

class ShuoShi extends XiaoPin{
    void jiaoliu(){
        System.out.println("顺口溜");
    }
    void xushi(){
        System.out.println("写书");
```

```
        }
        void gaoxiao(){
            System.out.println("打招呼");
        }
        void shanqing(){
            System.out.println("二人传");
        }
    }

    class DaPuKe extends XiaoPin{
        void jiaoliu(){
            System.out.println("大家好");
        }
        void xushi(){
            System.out.println("老同学见面");
        }
        void gaoxiao(){
            System.out.println("打扑克");
        }
        void shanqing(){
            System.out.println("马家军");
        }
    }
    /*************************************************************/
```

Day06

一、接口（interface）

1、 定义：接口不是类，而是一组对类需求的描述，这些类要遵从接口描述的统一格式进行定义。定义一个接口用关键字 interface。

例：public interface a{……}

2、 接口是一种特殊的抽象类。在一个接口中，所有的方法为公开、抽象的方法，所有的属性都是公开、静态、常量。所以接口中的所有属性可省略修饰符：public static

final，接口中所有的方法可省略修饰符：public abstract。

3、 一个类实现一个接口必须实现接口中所有的方法，否则其为一抽象类。并且在实现类中的方法要加上 public(不能省略)。实现接口用关键字 implements.

所谓实现一个接口就是实现接口中所有的方法。

例：class Aimple implements A{……..};

4、 一个类除了继承另一个类外（且只能继承一个类），还可以实现多个接口(接口之间用逗号分割)。这样可以实现变相的多继承。

例：class Aimple extends Arrylist implements A,B,C{…}

5、 不能用“new 接口名”来实例化一个接口，但可以声明一个接口。

6、 接口与接口之间可以多继承。

例：interface face1 extends face2,face3{}

7、 接口的作用

（1）、间接实现多继承。

用接口来实现多继承并不会增加类关系的复杂度。因为接口终归不是类，与类不在一个层次上，是在类的基础上进行再次抽象。

父类：主类型　　接口：副类型

典例：相声届师父（主）和干爹（副）

（2）、允许我们为一个类定义出混合类型。

（3）、通过接口制定标准

接　　口：标准的定义　定义标准

接口的调用者：标准的使用　使用标准

接口的实现类：标准的实现　实现标准

接口的回调:先有接口的使用者，再有接口的实现者，最后把接口的实现者的对象传到接口的使用者中，并由接口的使用者通过接口来调用接口实现者的方法。这就是接口回调。

例：sun 公司提供一套访问数据库的接口（标准），java 程序员访问数据库时针对数据库接口编程。接口由各个数据库厂商负责实现。

（4）、解耦合作用：采用接口可以最大限度的做到弱耦合，将标准的实现者与标准的制定者隔离（例：我们通过 JDBC 接口来屏蔽底层数据库的差异）

8、 接口的编程设计原则

（1）、尽量针对接口编程（能用接口就尽量用接口）

（2）、接口隔离原则（用若干个小接口取代一个大接口）这样可以只暴露想暴露的方法，实现一个更高层次的封装。

9、 注意点：

（1）、一个文件只能有一个 public 接口，且与文件名相同。

（2）、在一个文件中不能同时定义一个 public 接口和一个 public 类。

（3）、接口与实体类之间只有实现关系，没有继承关系；抽象类与类之间只有继承关系没有实现关系。接口与接口之间只有继承关系，且允许多继承。

（4）、接口中可以不写 public,但在子类实现接口的过程中 public 不可省略。

二、接口 VS 抽象类

1、接口中不能有具体的实现，但抽象类可以。

2、一个类要实现一个接口必须实现其里面所有的方法，而抽象类不必。

3、通过接口可以实现多继承，而抽象类做不到。

4、接口不能有构造方法，而抽象类可以。

5、实体类与接口之间只有实现关系，而实体类与抽象类只有继承关系，抽象类与接口之间既有实现关系又有继承关系。

6、接口中的方法默认都是公开抽象方法，属性默认都是公开静态常量，而抽象类不是。

三、Object 类

1、 object 类是类层次结构的根类，他是所有类默认的父类。

2、 object 类中的三个方法。

（1）、finalize()

当一个对象被垃圾收集的时候，最后会由 JVM 调用这个对象的 finalize 方法；

注意：这个方法一般不用，也不能将释放资源的代码放在这个方法里；

（2）、toString()

返回一个对象的字符串表示形式。打印一个对象其实就是打印这个对象 toString 方法的返回值。

我们可以在自己的类时覆盖 toString()方法，从而打印我们需要的数据。Public String toString(){……}

(3)、equals(Object o)

该方法用来判断对象的值是否相等。但前提是类中覆盖了 equals 方法。Object 类中的 equals 方法判断的其实还是地址。这里注意：String 类已经覆盖了 equals 方法了，所以我们能使用 equals 来判断 String 对象的值是否相等。

下面是覆盖 equals 方法的标准流程：

```
public Boolean equals(Object o){
/**第一步：现判断两个对象地址是否相等*/
    if(this = = o)    return      true;
/**第二步：如果参数是 null 的话直接返回 false;*/
    if(o = = null)    return      false;
```

```
/**第三步：如果两个对象不是同一个类型直接返回false*/
    if(  !(o  instanceof  Student)  )  return false;
/**第四步：将待比较对象强转成指定类型，然后自定义比较规则*/
    Student s = (Student)o;
    If(s.name.equals(this.name)&&s.age==this.age)
     return  true;
     else return false
}
```

（4）、equals 的特性：自反性、对称性、一致性、传递性。

Day07

一、封装类

JAVA 为每一个简单数据类型提供了一个封装类，使每个简单数据类型可以被**Object** 来装载。

除了 **int** 和 **char**，其余类型首字母大写即成封装类

int Integer

char Character

int Integer String 之间的类型转换(最常用的)

int a =12;

int 到 Integer Integer aa = new Integer(a);

Integer 到 int int i = aa.intValue();

Int 到 String String str = String.valueOf(i);

String 到 int int ii = Integer.parseInt(str);

Integer 到 String String str = aa.toString()

String 到 Integer Integer bb = Integer.valueOf(str)

二、 内部类（非重点）

1．定义：定义在其他类中的类，叫内部类。内部类是一种编译时的语法。编译后生成的两个类时独立的两个类。内部类配合接口使用，来强制做到弱耦合（局部内部类，或私有成员内部类）。

2．内部类存在的意义在于可以自由的访问外部类的任何成员（包括私有成员），所有使用内部类的地方都可以不用内部类，使用内部类可以使程序更加的简洁（以牺牲程序的可读性为代价），便于命名规范和划分层次结构。

3．内部类作为外部类的一个成员，并且依附于外部类而存在的。

4．内部类可为静态，可用**PROTECTED** 和**PRIVATE** 修饰。（而外部类不可以：外部类只能使用**PUBLIC**和**DEFAULT**）。

5．内部类的分类：成员内部类、局部内部类、静态内部类、匿名内部类。

① **成员内部类**：作为外部类的一个成员存在，与外部类的属性、方法并列。

内部类和外部类的实例变量可以共存。

在内部类中访问实例变量：**this.**属性

在内部类访问外部类的实例变量：外部类名**.this.**属性。

对于一个名为**outer** 的外部类和其内部定义的名为**inner** 的内部类。

编译完成后出现**outer.class** 和**outer$inner.class** 两类。

成员内部类不可以有静态属性，这是因为静态属性是在加载类的时候创建，这个时候内部类还没有被创建。

如果在外部类的外部访问内部类，使用**out.inner.**

建立内部类对象时应注意：

在外部类的内部可以直接使用**inner s=new inner();**（因为外部类知道**inner** 是哪个类，所以可以生成对象。）

而在外部类的外部，要生成（**new**）一个内部类对象，需要首先建立一个外部类对象（外部类可用），然后在生成一个内部类对象。

Outer.Inner in=Outer.new.Inner()。

相当于：

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

错误的定义方式：

Outer.Inner in=new Outer.Inner()。

② **局部内部类**：在方法中定义的内部类称为局部内部类。

与局部变量类似，在局部内部类前不加修饰符**public**和**private**，其范围为定义它的代码块。

注意：局部内部类不仅可以访问外部类实例变量，还可以访问外部类

的局部变量（但此时要求外部类的局部变量必须为**final**）

在类外不可直接生成局部内部类（保证局部内部类对外是不可见的）。

要想使用局部内部类时需要生成对象，对象调用方法，在方法中才能调用其局部内部类。

③ **静态内部类：**（注意：前三种内部类与变量类似，所以可以对照参考变量）

静态内部类定义在类中，任何方法外，用**static** 定义。

静态内部类只能访问外部类的静态成员。

生成（**new**）一个静态内部类不需要外部类成员：这是静态内部类和成员内部类的区别。静态内部类的对象可以直接生成：

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

对比成员内部类：Outer.Inner in = Outer.new Inner();

而不需要通过生成外部类对象来生成。这样实际上使静态内部类成为了一个顶级类。

静态内部类不可用**private** 来进行定义。例子：

对于两个类，拥有相同的方法：

```
/**********************************************************/
public class TestRobot{
    public static void main(String[] args){
        Robot r = new Robot();
        r.run();
        r.getHeart().run();
    }
}
```

```
abstract class People{
    abstract void run();
}

interface Machine{
    void run();
}

class Robot extends People{
    class Heart implements Machine{
        public void run(){
            System.out.println("发动机跑");
        }
    }
    public void run(){
        System.out.println("机器人跑");
    }
    public Machine getHeart(){
        return new Heart();
    }
}
/*************************************************************************/
```

此时**run()**不可直接实现。

注意：当类与接口（或者是接口与接口）发生方法命名冲突的时候，此时必须使用内部类来实现。**这是唯一一种必须使用内部类的情况**。用接口不能完全地实现多继承，用接口配合内部类才能实现真正的多继承。

④ **匿名内部类：**

【1】匿名内部类是一种特殊的局部内部类，它是通过匿名类实现接口。

【2】不同的是他是用一种隐含的方式实现一个接口或继承一个类，而且他只需要一个对象

【3】在继承这个类是，根本就没有打算添加任何方法。

【4】匿名内部类大部分情况都是为了实现接口的回调。

注：一个匿名内部类一定是在**new** 的后面，用其隐含实现一个接口或实现一个类，没有类名，根据多态，我们使用其父类名。

因其为局部内部类，那么局部内部类的所有限制都对其生效。

匿名内部类是唯一一种无构造方法类。

注:这是因为构造器的名字必须合类名相同,而匿名内部类没有类名。

匿名内部类在编译的时候由系统自动起名**Out$1.class**。

因匿名内部类无构造方法，所以其使用范围非常的有限。

接口＋内部类才能真正实现多继承。

一个关于匿名内部类的例子：

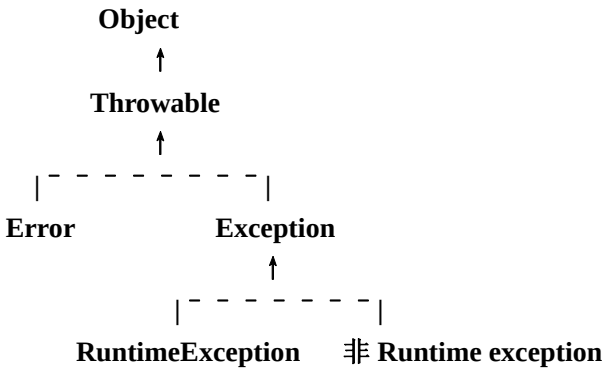
```
/**********************************************************/
            public class test{
                public static void main(String[] args){
                    B.print(new A(){
                            public void getConnection(){
                                System.out.println("Connection....");
                            }
                    });
                }
            }

            interface A{
                void getConnection();
            }

            class B{
                public static void print(A a){
                    a.getConnection();
                }
            }
/**********************************************************/
```

三、 异常 Exception

1. 概念： **JAVA** 会将所有的错误封装成为一个对象，其根本父类为 **Throwable**。异常处理可以提高我们系统的容错性。

   Throwable 有两个子类：**Error** 和 **Exception**。

   Error:一般是底层的不可恢复的错误。



2. **Exception** 分类：**Runtime exception**（未检查异常）和非 **Runtime exception**（已检查异常）。

   未检查异常是因为程序员没有进行必要的检查，因为他的疏忽和错误而引起的异常。一定是属于虚拟机内部的异常（比如空指针）。

   几个常见的未检查异常：

   1) java.lang.ArithmeticException

      如：分母为 0;

   2) java.lang.NullPointerException

      如：空指针操作;

   3) java.lang.ArrayIndexoutofBoundsExceptio

如：数组越界；

4） java. lang. ClassCastException

如：类型转换异常；

已检查异常是不可避免的，对于已检查异常必须处理。

3、异常对象的传递。

当一个方法中出现了异常而又没做任何处理，那么这个方法会返回该异常对象。依次向上层调用者传递，直到传到 JVM，虚拟机终止运行。（用一句话说就是沿着方法调用链反向传递）

4、如何来处理异常（这里主要是针对已检查异常）

【1】throws 消极处理异常的方式。

方法名（参数表）throws 后面接要往上层抛的异常。

表示该方法对指定的异常不作任何处理，直接抛往上一层。

【2】积极处理方式 try、catch

try {可能出现错误的代码块**} catch(exception e){**进行处理的代码**}** ；

一个异常捕获只会匹配一次 try,catch.

一个异常一旦被捕获就不存在了。

Catch 中要求必须先捕获子类异常再捕获父类异常。

【3】finally （紧接在 catch 代码块后面）

finally 后的代码块是无论如何都会被执行的(除非虚

拟机退出)，所以在 finally 后的代码快里我们一般写的是释放资源的代码。

```
Public static int fn(int b){
    Try{
        Return b/2;
    }catch(Exception e){
        return 0;
    }finally{
        return b;
    }
}
```

返回的结果是一定是 b;

5、 自定义异常(与一般异常的用法没有区别)

```
class MyException extends Exception{
  public MyException(String message){
      super(message);
  }
  public MyException(){}
}
```

6、如何控制 try 的范围：根据操作的连动性和相关性，如果前面的程序代码块抛出的错误影响了后面程序代码的运行，那么这个我们就说这两个程序代码存在关联，应该放在同一个 try 中。

7、不允许子类比父类抛出更多的异常。

8、断言：只能用于代码调试时用。（一般没什么用）

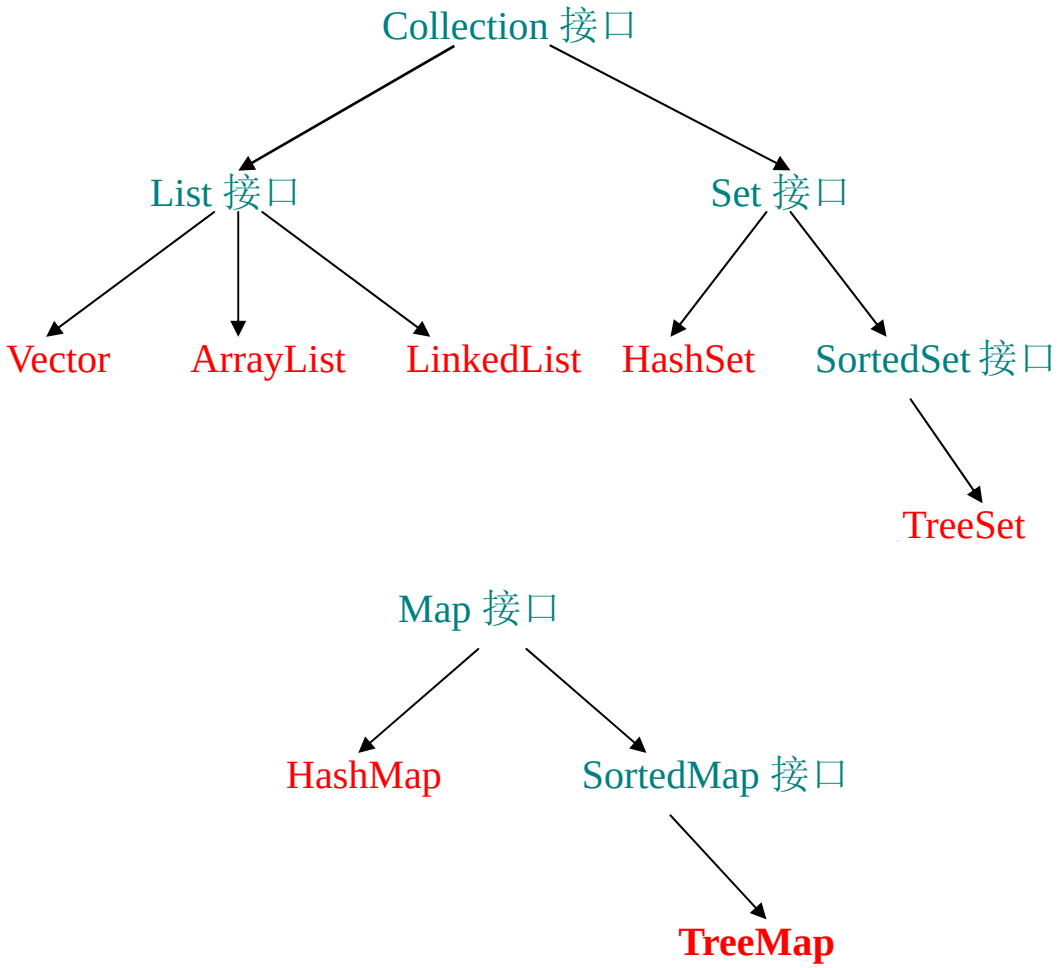
一个关于断言的列子：

```
/*********************************************************/
    public class TestAssertion {
       public static void main(String[] args){
          int i = Integer.parseInt(args[0]);
          assert i==1:"ABCDEFG";
          /*
```

```
            断言语句（表示断言该boolean语句返回值一定为真，
              如果断言结果为false就会报Error错误）
            ":"后面跟出现断言错误时要打印的断言信息。
            */
            System.out.println(i);
        }
    }

    //java -source 1.4 TestAssertion.java
    //表示用1.4的新特性来编译该程序。
    //java -ea TestAssertion 0
    //表示运行时要用到断言工具

/*************************************************************/
```

Day08

《集合框架》



一、 **集合：**集合是一个用于管理其他多个对象的对象

1、Collection 接口：集合中每一个元素为一个对象，这个接口将这些对象组织在一起，形成一维结构。

2、List 接口：代表按照元素一定的相关顺序来组织（在这个序列中顺序是主要的），**List** 接口中数据可重复。

3、Set 接口：是数学中集合的概念：其元素无序，且不可重复。

（正好与**List** 对应）

4、SortedSet 接口 ：会按照数字将元素排列，为“可排序集合”。

5、Map 接口 ：接口中每一个元素不是一个对象，而是一个键对象和值对象组成的键值对（**Key-Value**）。

6、SortedMap接口：如果一个**Map** 可以根据**key** 值排序，则称其为**SortedMap**。

注意：在“集合框架”中，Map 和Collection 没有任何亲缘关系。

- Map 的典型应用是访问按关键字存储的值。它支持一系列集合操作的全部，但操作的是键-值对，而不是单个独立的元素。因此Map 需要支持get() 和put() 的基本操作，而Set 不需要。
- 返回Map 对象的Set 视图的方法：

```
Set set = aMap.keySet()
```

《常用集合列表》

	List	Set	SortedSet	Map	SortedMap
存放元素	Object	Object	Object	Object(key)—Object(value)	Object(key)—Object(value)
存放顺序	有序	无序	无序	Key 无序	无序，有排序
元素可否重复	可	不可	不可	Key 不可，value 可	Key 不可，value 可
遍历方式	迭代	迭代	迭代	对 Key 迭代	对 Key 迭代
排序方式	Collections.sort()	SortedSet	已排序	SortedMap	已对键值排序
各自实现类	ArrayList LinkedList	HashSet	TreeSet	HashMap	TreeMap

注：以上有序的意思是指输出的顺序与输入元素的顺序一致

HashSet、HashMap 通过 hashCode()，equals()来判断重复元素

在 java 中指定排序规则的方式只有两种：1、实现 java.util 包下的 Comparator 接口

2、实现 java.lang 包下的 Comparable 接口

二、 迭代器：Iterator

1、使用 Iterator 接口方法，您可以从头至尾遍历集合，并安全的从底层 Collection 中除去元素

2、remove() 方法可由底层集合有选择的支持。当底层集合调用并支持该方法时，最近一次 next() 调用返回的元素就被除去

3、Collection 接口的iterator() 方法返回一个Iterator

4、Iterator中的hasNext()方法表用于判断元素右边是否还有数据，返回True 说明有。然后就可以调用next（）动作。

5、Iterator中的next()方法会将游标移到下一个元素，并把它所跨过的元素返回。（这样就可以对元素进行遍历）

6、用于常规 Collection 的 Iterator 接口代码如下：

```
/*迭代遍历*/
   List l = new ArrayList();
   Iterator it = l.iterator();
   while(it.hasNext()){
      Object o = it.next();
      System.out.println(o);
}
```

注：工具类是指所有的方法都是公开静态方法的类。

Java.util.collections 就是一个工具类；

三、 对集合的排序

1、 我们可以用 Java.util.collections 中的 sort(List l)方法对指定的 List 集合进行排序；但是如果 List 中存放的是

自定义对象时，这个方法就行不通了，必须实现 Comparable 接口并且指定排序规则。

这里我们再来看一下 sort(List l)方法的内部实现；

/***/

```
class Collections2{
   public static void sort(List l){
      for(int i=0;i<l.size()-1;i++){
         for(int j=i+1;j<l.size();j++){
            Object o1 = l.get(i);
            Object o2 = l.get(j);
            Comparable c1 = (Comparable)o1;
            Comparable c2 = (Comparable)o2;
            if(c1.compareTo(c2)>0){
               Collections.swap(l,i,j);
            }
         }
      }
   }
}
```

注：其实用的算法就是个冒泡排序。

/***/

2、 实现 Java.lang.Comparable 接口，其实就是实现他的 public int compareTo(Object o)方法；

比较此对象与指定对象的顺序。如果该对象小于、等于或大于指定对象，则分别返回负整数、零或正整数。

其规则是当前对象与o 对象进行比较，其返回一个int 值，系统根据此值来进行排序。

如当前对象>o 对象，则返回值>0；

如当前对象=o 对象，则返回值=0；

如当前对象<o 对象，则返回值〈0。

注意：String 类型已经实现了这个接口，所以可以直接排序；

```
/*************************************************************/
   class Student implements Comparable{
         private String name;
         private int age;
         public Student(String name, int age) {
            this.name = name;
            this.age = age;
         }
         public int compareTo(Object o) {
            Student s = (Student)o;
            return s.age-this.age;
         }
   }
/*************************************************************/
```

四、 ArrayList 和 LinkedList 集合

1、ArrayList 底层是**object** 数组，所以**ArrayList** 具有数组的查询速度快的优点以及增删速度慢的缺点。

Vector 底层实现也是数组，但他是一个线程安全的重量级组件。

2、而在**LinkedList** 的底层是一种双向循环链表。在此链表上每一个数据节点都由三部分组成：前指针（指向前面的节点的位置），数据，后指针（指向后面的节点的位置）。最后一个节点的后指针指向第一个节点的前指针，形成一个循环。

3、双向循环链表的查询效率低但是增删效率高。所以**LinkedList** 具

有查询效率低但增删效率高的特点。

4、ArrayList 和**LinkedList** 在用法上没有区别，但是在功能上还是有区别的。

LinkedList 经常用在增删操作较多而查询操作很少的情况下：队列和堆栈。

队列：先进先出的数据结构。

堆栈：后进先出的数据结构。

（堆栈就是一种只有增删没有查询的数据结构）

注意：使用堆栈的时候一定不能提供方法让不是最后一个元素的元素获得出栈的机会。

LinkedList 提供以下方法：（**ArrayList** 无此类方法）

addFirst();

removeFirst();

addLast();

removeLast();

在堆栈中，**push** 为入栈操作，**pop** 为出栈操作。

Push 用**addFirst()**；**pop** 用**removeFirst()**，实现后进先出。

用**isEmpty()--**其父类的方法，来判断栈是否为空。

在队列中，**put** 为入队列操作，**get** 为出队列操作。

Put 用**addFirst()**，**get** 用**removeLast()**实现队列。

List 接口的实现类（**Vector**）（与**ArrayList** 相似，区别是**Vector** 是重量级的组件，使用使消耗的资源比较多。）

结论：在考虑并发的情况下用**Vector**（保证线程的安全）。

在不考虑并发的情况下用 **ArrayList**（不能保证线程的安全）。

5、面试经验（知识点）：

java.util.stack（**stack** 即为堆栈）的父类为**Vector**。可是**stack** 的父类是最不应该为**Vector** 的。因为**Vector**的底层是数组，且**Vector** 有 **get** 方法（意味着它可能访问到并不属于最后一个位置元素的其他元素，很不安全）。

对于堆栈和队列只能用**push** 类和**get** 类。

Stack 类以后不要轻易使用。

！！！实现堆栈一定要用**LinkedList**。

（在**JAVA1.5** 中，**collection** 有**queue** 来实现队列。）

五、 HashSet 集合

1.、HashSet 是无序的，没有下标这个概念。HashSet 集合中元素不可重复（元素的内容不可重复）；

2、HashSet 底层用的也是数组。

3、**HashSet 如何保证元素不重复**？Hash 算法和 equals 方法。

当向数组中利用**add(Object o)**添加对象的时候，系统先找对象的 **hashCode**：

int hc=o.hashCode(); 返回的**hashCode** 为整数值。

Int I=hc%n;（**n** 为数组的长度），取得余数后，利用余数向数组中相应的位置添加数据，以**n** 为**6** 为例，

如果**I=0** 则放在数组**a[0]**位置，如果**I=1,**则放在数组**a[1]**位置。如

果**equals()**返回的值为**true**，则说明数据重复。如果**equals()**返回的值为**false**，则再找其他的位置进行比较。这样的机制就导致两个相同的对象有可能重复地添加到数组中，因为他们的**hashCode**不同。如果我们能够使两个相同的对象具有相同**hashcode**，才能在**equals()**返回为真。

在实例中，定义**student** 对象时覆盖它的**hashcode**。

因为**String** 类是自动覆盖的，所以当比较**String** 类的对象的时候，就不会出现有两个相同的**string** 对象的情况。

现在，在大部分的**JDK** 中，都已经要求覆盖了**hashCode**。

结论：如将自定义类用**hashSet** 来添加对象，一定要覆盖**hashcode()**和**equals()**，覆盖的原则是保证当两个对象**hashcode** 返回相同的整数，而且**equals()**返回值为**True**。

如果偷懒，没有设定**equals()**，就会造成返回**hashCode** 虽然结果相同，但在程序执行的过程中会多次地调用**equals()**，从而影响程序执行的效率。

我们要保证相同对象的返回的**hashCode** 一定相同，也要保证不相同的对象的**hashCode** 尽可能不同（因为数组的边界性，**hashCode** 还是可能相同的）。例子：

```
public int hashCode(){
return name.hashcode()+age;
}
```

六、 TreeSet 集合

1、 TreeSet 是 SortedSet 的实现类 TreeSet 通过实现 Comparable 接口来实现元素不重复。

2、 TreeSet 由于每次插入元素时都会进行一次排序，因此效率不高。

3、 java.lang.ClassCastException 是类型转换异常。

4、 在我们给一个类用 CompareTo()实现排序规则时

5、 从集合中以有序的方式抽取元素时，可用 TreeSet，添加到 TreeSet 的元素必须是可排序的。“集合框架”添加对 Comparable 元素的支持。一般说来，先把元素添加到 HashSet，再把集合转换为 TreeSet 来进行有序遍历会更快。

Day09

一、 HashMap 集合

1. **HashMap** 就是用 hash 算法来实现的 Map
2. 在实际开发中一般不会用自定义的类型作为 Map 的 Key。做 Key 的无非是八中封装类。
3. **HashMap** 的三组操作：

【1】改变操作允许您从映射中添加和除去键-值对。键和值都可以为null。但是，您不能把Map 作为一个键或值添加给自身。

–Object put(Object key, Object value)

–Object remove(Object key)

–void clear()

【2】查询操作允许您检查映射内容：

–Object get(Object key)

–intsize()

–boolean isEmpty()

【3】最后一组方法允许您把键或值的组作为集合来处理。

–public Set KeySet();

–public Collection values()

4、 **HashMap** 和 **HashTable** 的区别等同于 **ArrayList** 和 **Vector** 的区别。只不过 **HashTable** 中的 **Key** 和 **Value** 不能为空，而 **HashMap** 可以。

5、 **HashMap** 底层也是用数组，**HashSet** 底层实际上也是 **HashMap**，**HashSet** 类中有 **HashMap** 属性（我们如何在 **API** 中查属性）。**HashSet** 实际上为**(key.null)**类型的 **HashMap**。有 **key** 值而没有 **value** 值。

二、HashMap 类和TreeMap 类

•集合框架提供两种常规Map 实现：HashMap和TreeMap。

•在Map 中插入、删除和定位元素，HashMap 是最好选择。

•如果要按顺序遍历键，那么TreeMap 会更好。

•根据集合大小，先把元素添加到HashMap，再把这种映射转换成一个

用于有序键遍历的TreeMap 可能更快。

•使用HashMap 要求添加的键类明确定义了hashCode() 实现。

•有了TreeMap 实现，添加到映射的元素一定是可排序的

•HashMap 和 TreeMap 都实现 Cloneable 接口。

三、图型界面（不重要）

1、Awt：抽象窗口工具箱，它由三部分组成：

①组件：界面元素；

②容器：装载组件的容器（例如窗体）；

③布局管理器：负责决定容器中组件的摆放位置。

2、图形界面的应用分四步：

① 选择一个容器：

⑴window:带标题的容器（如Frame）；

⑵Panel:面板通过add()向容器中添加组件。

注：Panel不能作为顶层容器。

Java 的图形界面依然是跨平台的。但是在调用了一个窗体之后只生成一个窗体，没有事件的处理，关闭按钮并不工作。此时只能使用CTRL+C 终止程序。

②设置一个布局管理器：用setLayout()；

③向容器中添加组件；

jdk1.4用getContentPare()方法添加主件。

③ 添加组件的事务处理。

Panel 也是一种容器：但是不可见的。在设置容易的时候不要忘记设置它们的可见性。

Panel pan=new Panel;

Fp.setLayout(null);表示不要布局管理器。

3、五种布局管理器：

（1）、Flow Layout(流式布局)：按照组件添加到容器中的顺序，顺序排放组件位置。默认为水平排列，如果越界那么会向下排列。排列的位置随着容器大小的改变而改变。

Panel 默认的布局管理器为Flow Layout。

（2）、BorderLayout：会将容器非常五个区域：东西南北中。

语句：

Button b1=new Botton(“north”);//botton 上的文字

f.add(b1,”North”);//表示b1 这个botton 放在north 位置

注：一个区域只能放置一个组件，如果想在一个区域放置多个组件就需要使用Panel 来装载。

Frame 和Dialog 的默认布局管理器是Border Layout。

（3）、Grid Layout（网格布局管理器）：将容器生成等长等大的条列格，每个块中放置一个组件。

f.setLayout GridLayout(5,2,10,10)//表示条列格为5行2 类，后面为格间距。

（4）、CardLayout（卡片布局管理器）:一个容器可以放置多个组

件，但每次只有一个组件可见（组件重叠）。

使用first()，last()，next()可以决定哪个组件可见。可以用于将一系列的面板有顺序地呈现给用户。

(5)、GridBag Layout（复杂的网格布局管理器）:在Grid 中可指定一个组件占据多行多列，GridBag 的设置非常的烦琐。

注：添加滚动条：JScrollPane jsp = new JScrollPane(ll);

4、常用的组件：

（1）、JTextArea：用作多行文本域

（2）、JTextField：作单行文本编辑空件

（3）、JButton:按钮

（4）、JComboBox：从下拉框中选择记录

（5）、JList：在界面上显示多条记录并可多重选择的列表

Day10

一．事件模型(重点)

1．定义：事件模型指的是对象之间进行通信的设计模式。

事件模型是在观察者模式基础上发展来的。

2．对象1 给对象2 发送一个信息相当于对象1 引用对象2 的方法。

3．对象对为三种：

（1）事件源：发出事件者；

（2）事件对象：发出的事件本身(事件对象中会包含事件源对象)

事件对象继承:java.util.EventObjct类.

（3）事件监听器：提供处理事件指定的方法。

标记接口：没有任何方法的接口；如EventListene接口

监听器接口必须继承java.util.EventListener接口。

监听接口中每一个方法都会以相应的事件对象作为参数。

4．授权：Java AWT 事件模型也称为授权事件模型，指事件源可以和监听器之间事先建立一种授权关系：约定那些事件如何处理，由谁去进行处理。这种约定称为授权。

当事件条件满足时事件源会给事件监听器发送一个事件对象，由事件监听器去处理。事件源和事件监听器是完全弱偶合的。

一个事件源可以授权多个监听者（授权也称为监听者的注册）；事件源也可以是多个事件的事件源。监听器可以注册在多个事件源当中。监听者对于事件源的发出的事件作出响应。

在java.util 中有**EventListener** 接口：所有事件监听者都要实现这个接口。

java.util 中有**EventObject** 类：所有的事件都为其子类。

事件范例在**\CoreJava\Girl.java** 文件中。**(**文件已加注释**)**

注意：接口因对不同的事件监听器对其处理可能不同，所以只能建立监听的功能，而无法实现处理。

下面程序建立监听功能：

```
//监听器接口要定义监听器所具备的功能，定义方法


/**********************************************************/
```

```
import java.util.ArrayList;
import java.util.EventListener;
import java.util.EventObject;
import java.util.Iterator;
import java.util.List;

public class TestShare {

   public static void main(String[] args) {
      Share s = new Share();
      Man1 man1 = new Man1("Huxz");
      Man2 man2 = new Man2("Liucy");
      s.addListener(man1);
      s.addListener(man2);
      s.fire();
      s.removeListener(man1);
   }
}
/**第一步:先写一个事件对象继承EventObject类*/
class ShareEvent extends EventObject{
   public ShareEvent(Object source) {
      super(source);
   }

}
/**第二步：写一个事件监听器接口继承EventListener接口*/
interface ShareListener extends EventListener{
   void whenShareRise(ShareEvent s);
   void whenShareDrop(ShareEvent s);
}


/**第三步：写一个事件源*/
class Share{
   private List listener = new ArrayList();
   //注册一个监听者
   public void addListener(ShareListener s){
      listener.add(s);
   }
   //删除一个监听者
```

```
   public void removeListener(ShareListener s){
      listener.remove(s);
      System.out.println(s.getClass().getName()+" has
been removed!");
   }
   public void fire(){
      ShareEvent shareEvent = new ShareEvent(this);
      for(int i=0;i<10;i++){
         Iterator it = listener.iterator();
         int index = i%2;
         if(index==0){
            System.out.println("Share has Rised");
         }else{
            System.out.println("Share has Droped");
         }
         while(it.hasNext()){
            ShareListener s = (ShareListener)it.next();
            if(index==0){
               s.whenShareRise(shareEvent);
            }else{
               s.whenShareDrop(shareEvent);
            }
         }
      }
   }
}


/**第四步：写一个事件监听器的实现类*/
class Man1 implements ShareListener{
   private String name;

   public Man1(String name) {
      this.name = name;
   }
   public void whenShareRise(ShareEvent s){
      System.out.println(name+" sad:buy");
   }
   public void whenShareDrop(ShareEvent s){
      System.out.println(name+" sad:sell");
   }
}

class Man2 implements ShareListener{
```

```
    private String name;
    public Man2(String name) {
        this.name = name;
    }
    public void whenShareRise(ShareEvent s){
        System.out.println(name+" sad:sell");
    }
    public void whenShareDrop(ShareEvent s){
        System.out.println(name+" sad:buy");
    }
}
/*********************************************************/
```

注意查看参考书：事件的设置模式，如何实现授权模型。

事件模式的实现步骤：

开发事件对象（事件发送者）——接口——接口实现类——设置监听对象

一定要理解透彻**Gril.java** 程序。

重点：学会处理对一个事件源有多个事件的监听器（在发送消息时监听器收到消息的排名不分先后）。

事件监听的响应顺序是不分先后的，不是谁先注册谁就先响应。

事件监听由两个部分组成（接口和接口的实现类）。

事件源事件对象事件监听

gril EmotinEvent EmotionListener(接口**)**、**Boy(**接口的实现类**)**

鼠标事件：**MouseEvent**，接口：**MouseListener**。

P235 ActionEvent。

注意在写程序的时候：**import java.awt.*;**以及**import java.awt.event.***

注意两者的不同。

在生成一个窗体的时候，点击窗体的右上角关闭按钮激发窗体事件的

方法：窗体**Frame** 为事件源，

WindowsListener 接口调用**Windowsclosing()**。

为了配合后面的实现，我们必须将**WindowsListener** 所有的方法都实现，除了**Windowsclosing** 方法，

其余的方法均为空实现。

（练习：写一个带**button** 窗体，点关闭按钮退出。）

上面程序中实现了许多不必要的实现类，虽然是空实现。

为了避免上面那些无用的实现，可以利用**WindowEvent** 的一个**WindowEvent** 类，还是利用

windowsListener。还有**WindowAdapter** 类，它已经实现了**WindowsListener**。它给出的全部都是空实

现，那就可以只写想要实现的类，去覆盖其中的类，就不用写空实现。

注意：监听过多，会抛**tooManyListener** 例外。

缺省试配设计模式：如果一个接口有太多的方法，我们可以为这个接口配上一个对应的抽象类。

Day11

《多线程》

一. 线程:线程是一个并发执行的顺序流, 一个进程包括多个顺序执行流程, 这执行流程称为线程. 线程是一个操作系统创建并维护的一个资源, 对操作系统来说JVM就是一个进程. 对于单

CPU系统来说，某一个时刻只可能由一个线程在运行。

一个Thread对象就表示一个线程。

线程由三部分组成:

(1).CPU分配给线程的时间片

(2).线程代码(写在run方法中)

(3).线程数据

进程是独立的数据空间,线程是共享的数据空间.

线程对象存在于虚拟机对空间的一块连续的地址空间(静态)

注意:1.线程是动态的,与线程对象是两回事.

2.线程对象与其他对象不同的是线程对象能够到底层去申请管理一个线程资源。

3.只有对线程对象调用start()方法才是到底层去申请管理一个线程资源。

4.任务并发执行是一个宏观概念，微观上是串行的。

二.进程的调度

进程的调度是由OS 负责的（有的系统为独占式，有的系统为共享式，根据重要性，进程有优先级）。由OS 将时间分为若干个时间片。JAVA 在语言级支持多线程。分配时间的仍然是OS。

三.线程由两种实现方式：

第一种方式：

```
class MyThread extends Thread{
public void run(){
```

需要进行执行的代码，如循环。

```
}
}
public class TestThread{
main(){
Thread t1=new Mythread();
T1.start();
}
}
```

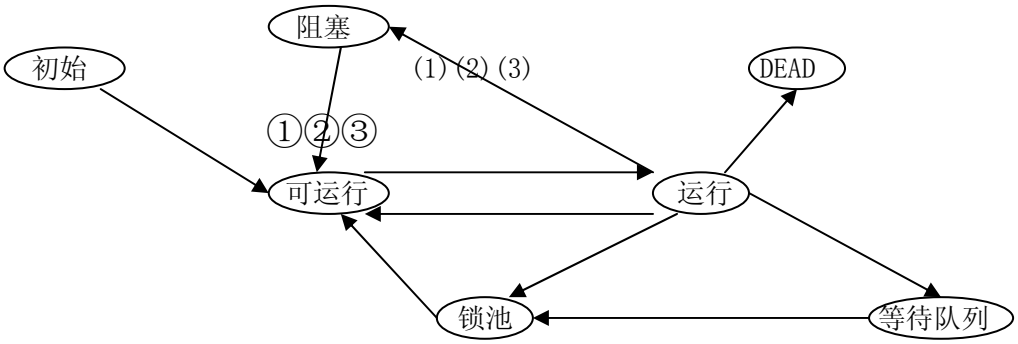
只有等到所有的线程全部结束之后，进程才退出。

第二种方式：通过接口实现继承

```
Class MyThread implements Runnable{
Public void run(){
Runnable target=new MyThread();
Thread t3=new Thread(target);
Thread.start();//启动线程
}
}
```

四．下面为线程中的7 中非常重要的状态：

（有的书上也只有认为前五种状态：而将“锁池”和“等待队列”都看成是“阻塞”状态的特殊情况：这种认识也是正确的，但是将“锁池”和“等待队列”单独分离出来有利于对程序的理解）



注意：图中标记依次为

①输入完毕；②wake up③t1 退出

⑴如等待输入（输入设备进行处理，而CUP 不处理），则放入阻塞，直到输入完毕。

⑵线程休眠sleep（）

⑶t1.join()指停止main()，然后在某段时间内将t1 加入运行队列，直到t1 退出，main()才结束。

特别注意：①②③与⑴⑵⑶是一一对应的。

进程的休眠：Thread sleep(1000);//括号中以毫秒为单位

当**main()**运行完毕，即使在结束时时间片还没有用完，**CPU** 也放弃此时间片，继续运行其他程序。

Try{Thread.sleep(1000);}

Catch(Exception e){e.printStackTrace(e);}

T1.join()表示运行线程放弃执行权，进入阻塞状态。

当**t1** 结束时，**main()**可以重新进入运行状态。

T1.join 实际上是把并发的线程编程并行运行。

五．线程的优先级：

越大优先级越高，优先级越高被**OS** 选中的可能性就越大。（不建议使用，因为不同操作系统的优先级并不相同，使得程序不具备跨平台性，这种优先级只是粗略地划分）。

设置线程优先级：setPriority（Thread. MAX_PRIORITY）；

注：程序的跨平台性：除了能够运行，还必须保证运行的结果。

当一个现成对象调用**yield()**方法时会马上交出执行权，回到可运行状态，等待**OS** 的再次调用。由一个高优先级的线程进入运行状态。

注：设置线程优先级只有在独占式系统中有效。

程序员需要关注的线程同步和互斥的问题。

多线程的并发一般不是程序员决定，而是由容器决定。

六．多线程出现故障的原因：

(1).两个线程同时访问一个数据资源（该资源称为临界资源），形成数据发生不一致和不完整。

(2).数据的不一致往往是因为一个线程中的两个关联的操作只完成了一步。

避免以上的问题可采用对数据进行加锁的方法

七．对象锁Synchronized

1. 互斥锁标记:每个对象除了属性和方法，都有一个monitor（互斥锁标记),用来将这个对象交给一个线程,只有拿到monitor 的线程才能够访问这个对象。
2. Synchronized:这个修饰词可以用来修饰方法和代码块

```
Object obj;
Obj.setValue(123);
```

Synchronized用来修饰代码块时，该代码块成为同步代码块。

Synchronized 用来修饰方法，表示当某个线程调用这个方法之后，其他的事件不能再调用这个方法。只有拿到obj 标记的线程才能够执行代码块。

注意：（1）Synchronized 一定使用在一个方法中。

（2）锁标记是对象的概念，加锁是对对象加锁，目的是在线程之间进行协调。

（3）当用Synchronized 修饰某个方法的时候，表示该方法都对当前对象加锁。

（4）给方法加Synchronized 和用Synchronized 修饰对象的效果是一致的。

（5）一个线程可以拿到多个锁标记，一个对象最多只能将monitor 给一个线程。

（6）构造方法和抽象方法不能加synchronized;

（7）一般方法和静态方法可以加synchronized同步。

（8）Synchronized 是以牺牲程序运行的效率为代价的，因此应该尽量控制互斥代码块的范围。

（9）方法的Synchronized 特性本身不会被继承，只能覆盖。

八．锁池

1．定义：线程因为未拿到锁标记而发生的阻塞不同于前面五个基本状态中的阻塞，称为锁池。

2．每个对象都有自己的一个锁池的空间，用于放置等待运行的线程。这些线程中哪个线程拿到锁标记由系统决定。

3．死锁：锁标记如果过多，就会出现线程等待其他线程释放锁标记，而又都不释放自己的锁标记供其他线程运行的状况。就是死锁。

死锁的问题通过线程间的通信的方式进行解决。

九．线程间通信：

1．线程间通信机制实际上也就是协调机制。

线程间通信使用的空间称之为对象的等待队列，则个队列也是属于对象的空间的。

注：那么学到这里，我们已经知道一个对象除了由属性和方法之外还有互斥锁标记、锁池空间和等待队列空间。

2．Object 类中又一个wait()，在运行状态中，线程调用wait()，此时表示着线程将释放自己所有的锁标记，同时进入这个对象的等待队列。等待队列的状态也是阻塞状态，只不过线程释放自己的锁标记。

3．Notify()

如果一个线程调用对象的notify()，就是通知对象等待队列的一个线程出列。进入锁池。如果使用notifyall()则通知等待队列中所有的线程出列。

注意：只能对加锁的资源进行wait()和notify()。

我们因该用notifyall取代notify，因为我们用notify释放出的一个线程是不确定的，由OS决定。

释放锁标记只有在Synchronized 代码结束或者调用wait()。

注意锁标记是自己不会自动释放，必须有通知。

注意在程序中判定一个条件是否成立时要注意使用WHILE 要比使用IF 要严密。

WHILE 会放置程序饶过判断条件而造成越界。

4.补充知识：

suspend（）是将一个运行时状态进入阻塞状态（注意不释放锁标记）。恢复状态的时候用resume()。Stop()

指释放全部。

这几个方法上都有Deprecated 标志，说明这个方法不推荐使用。

一般来说，主方法main()结束的时候线程结束，可是也可能出现需要中断线程的情况。对于多线程一般

每个线程都是一个循环，如果中断线程我们必须想办法使其退出。

如果主方法main()想结束阻塞中的线程（比如sleep 或wait）

那么我们可以从其他进程对线程对象调用interrupt()。用于对阻塞（或锁池）会抛出例外Interrupted

5. Exception。

这个例外会使线程中断并执行catch 中代码。

多线程中的重点： 实现多线程的两种方式， Synchronized, 以及生产者和消费者问题

（ProducerConsumer.java 文件）。

练习：

① 存车位的停开车的次序输出问题；

② 写两个线程，一个线程打印1-52，另一个线程答应字母A-Z。打印顺序为12A34B56C……5152Z。通

过使用线程之间的通信协调关系。

注：分别给两个对象构造一个对象**o**，数字每打印两个或字母每打印一个就执行**o.wait()**。在**o.wait()**之

前不要忘了写**o.notify()**。

补充说明：通过Synchronized，可知Vector 较ArrayList 方法的区别就是Vector 所有的方法都有

Synchronized。所以Vector 更为安全。

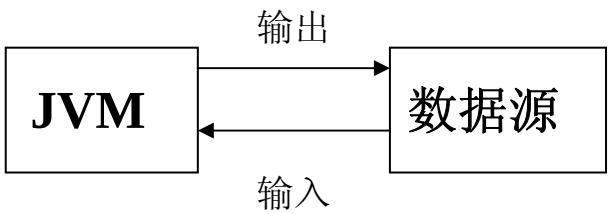
同样：Hashtable 较HashMap 也是如此。

Day12

《I/O 流》

一．I/O 流（java 如何实现与外界数据的交流）

1．Input/Output：指跨越出了**JVM** 的边界，与外界数据的源头或者目标数据源进行数据交换。



注意：输入/输出是针对**JVM** 而言。

2．流的分类：

按流向分为输入流和输出流；

按传输单位分为字节流和字符流；

以Stream结尾的类都是字节流。

按功能还可以分为节点流和过滤流。

节点流：负责数据源和程序之间建立连接；（相当于裸枪）

过滤流：用于给节点增加功能。（相当于功能零部件）

过滤流的构造方式是以其他流位参数构造（这样的设计模式称为装饰模式）。

注：I/O流是一类很宝贵的资源，使用完后必须调用close()方法关闭流并释放资源。在关闭流时只用关闭最外层的流。

3．File 类（**java.io.***）可表示一个文件，也有可能是一个目录（在 **JAVA** 中文件和目录都属于这个类中，而且区分不是非常的明显）。

4．Java.io 下的方法是对磁盘上的文件进行磁盘操作，但是无法读取文件的内容。

注意：创建一个文件对象和创建一个文件在**JAVA** 中是两个不同的概念。前者是在虚拟机中创建了一个文件，但却并没有将它真正地创建到**OS** 的文件系统中，随着虚拟机的关闭，这个创建的对象也就消失了。而创建一个文件才是在系统中真正地建立一个文件。

例如：**File f=new File(“11.txt”);**//创建一个名为**11.txt** 的文件对象

f.CreateNewFile(); //真正地创建文件

f.CreateMkdir()：创建目录

f.delete()；删除文件

getAbsolutePath()；打印文件绝对路径

getPath（）；打印文件相对路径

f.deleteOnExit();在进程退出的时候删除文件，这样的操作通常用在临时文件的删除。

5．对于命令：**File f2=new file(“d:\\abc\\789\\1.txt”)**

这个命令不具备跨平台性，因为不同的**OS** 的文件系统很不相同。如果想要跨平台，在**file** 类下有**separtor()**，返回锁出平台的文件分隔符。

File.fdir=new File(File.separator);

String str=”abc”+File.separator+”789”;

使用文件下的方法的时候一定注意是否具备跨平台性。

6．List()：显示文件的名（相对路径）

ListFiles()：返回**Files** 类型数组，可以用**getName()**来访问到文件名。

使用**isDirectory()**和**isFile()**来判断究竟是文件还是目录。

使用**I/O**流访问**file**中的内容。

JVM与外界通过数据通道进行数据交换。

二．字节流

1．字节输入流：**io**包中的**InputStream**为所有字节输入流的父类。

Int read();读入一个字节（每次一个）；

可先使用**new byte[]=**数组，调用**read(byte[] b)**

read (byte[])返回值可以表示有效数；**read (byte[])**返回值为**-1** 表示结束。

2．字节输出流：**io**包中的**OutputStream**位所有字节输入流的父类。

Write和输入流中的**read**相对应。

3．在流中**close()**方法由程序员控制。因为输入输出流已经超越了J**VM**的边界，所以有时可能无法回收资源。

原则：凡是跨出虚拟机边界的资源都要求程序员自己关闭，不要指望垃圾回收。

4．以**Stream**结尾的类都是字节流。

如果构造**FileOutputStream**的同时磁盘会建立一个文件。如果创建的文件与磁盘上已有的文件名重名，就会发生覆盖。

用**FileOutputStream**中的**boolean**，则视，添加情况，将数据覆盖重名文件还是将输入内容放在文件的后面。

DataOutputStream:输入数据的类型。可以对八种基本类型加上String类型进行写入。

因为每中数据类型的不同，所以可能会输出错误。所有对于：**DataOutputStream　DataInputStream**两者的输入顺序必须一致。

过滤流：

bufferedOutputStream

bufferedInputStream

用于给节点流增加一个缓冲的功能。（典型的牺牲空间换时间）

在**VM**的内部建立一个缓冲区，数据先写入缓冲区，等到缓冲区的数据满了之后再一次性写出，效率很高。

使用带缓冲区的输入输出流的速度会大幅提高，缓冲区越大，效率越高。（这是典型的牺牲空间换时间）

切记：使用带缓冲区的流，如果数据数据输入完毕，使用**flush**方法将缓冲区中的内容一次性写入到外部数据源。用**close()**也可以达到相同的效果，因为每次**close**都会使用**flush**。一定要注意关闭外部的过滤流。

（非重点）管道流：也是一种节点流，用于给两个线程交换数据。

PipedOutputStream

PipedInputStream

输出流：**connect(**输入流**)**

RondomAccessFile 类允许随机访问文件同时拥有读和写的功能。

GetFilepoint()可以知道文件中的指针位置，使用**seek()**定位。

Mode(“r”:随机读；**”w”**：随机写；**”rw”**：随机读写**)**

字符流：**reader\write** 只能输纯文本文件。

FileReader 类：字符文件的输出

三、字节流的字符编码：

字符编码把字符转换成数字存储到计算机中，按**ASCii** 将字母映射为整数。

把数字从计算机转换成相应的字符的过程称为解码。

乱码的根源在于编解码方式不统一。在世界上任何一种编码方式中都会向上兼容ASCII码。所以英文没有乱码。

编码方式的分类：

ASCII（数字、英文）**:1** 个字符占一个字节（所有的编码集都兼容**ASCII**）

ISO8859-1（欧洲）：**1** 个字符占一个字节

GB-2312/GBK：**1** 个字符占两个字节。GB代表国家标准。

GBK是在GB－2312上增加的一类新的编码方式，也是现在最常用的汉字编码方式。

Unicode: 1 个字符占两个字节（网络传输速度慢）

UTF-8：变长字节，对于英文一个字节，对于汉字两个或三个字节。

原则：保证编解码方式的统一，才能不至于出现错误。

I/O学习种常范的两个错误：1。忘了flush

2．没有加换行。

四、 字符流

以Reader或write结尾的流为字符流。

Io 包的**InputStreamread** 称为从字节流到字符流的桥转换类。这个类可以设定字符转换方式。

OutputStreamred:字符到字节

Bufferread 有**readline()**使得字符输入更加方便。

在**I/O** 流中，所有输入方法都是阻塞方法。

Bufferwrite 给输出字符加缓冲，因为它的方法很少，所以使用父类**printwrite**，它可以使用字节流对象构造，省了桥转换这一步，而且方法很多。注：他是带缓冲的。最常用的输出流。

最常用的读入流是BufferedReader.

Day13

一．对象序列化

1. 定义：把一个对象通过I/O流写到文件（持久性介质）上的过程叫做对象的序列化。
2. 序列化接口：**Serializable**

   此接口没有任何的方法，这样的接口称为标记接口。
3. 不是所有对象都能序列化的，只有实现了Serializable的类，他的实例对象才是可序列化的。
4. 在Java种定义了一套序列化规范，对象的编码和解码方式都是已经定义好的。
5. **class ObjectOutputStream** 和**ObjectInputStream**也是过滤流，使节点流直接获得输出对象。

最有用的方法：

（1）writeObject(Object b)

（2）readObject();该方法返回的是读到的一个对象，但是需要我们注意的是,该方法不会以返回null表示读到文件末尾。而是当读到文件末尾时会抛出一个IOException；

6. 序列化一个对象并不一定会序列化该对象的父类对象
7. 瞬间属性（临时属性）不参与序列化过程。
8. 所有属性必须都是可序列化的，特别是当有些属性本身也是对象的时候，要尤其注意这一点。序列化的集合就要求集合中的每一个元素都是可序列化的。
9. 用两次序列化把两个对象写到文件中去（以追加的方式），和用一次序列化把两个对象写进文件的大小是不一样的。因为每次追加时都会要在文件中加入一个开始标记和结束标记。所以对于对象的序列化不能以追加的方式写到文件中。

二、transient关键字

1. 用transient修饰的属性为临时属性。

三．分析字符串工具java.util. StringTokenizer;

1. string tokenizer 类允许应用程序将字符串分解为

标记

2. 可以在创建时指定，也可以根据每个标记来指定分隔符（分隔标记的字符）集合。

3. **StringTokenizer(s,":")** 用“：”隔开字符，**s** 为String对象。

例：

```
/*************************************************************/
      import java.util.StringTokenizer;
      public class TestStringTokenizer {
         public static void main(String[] args) {
            String s = "Hello:Tarena:Chenzq";
            StringTokenizer st = new
      StringTokenizer(s,":");
            while(st.hasMoreTokens()){
               String str = st.nextToken();
               System.out.println(str);
            }
         }
      }
/*************************************************************/
```

四、网络的基础知识：

1、 ip：主机在网络中的唯一标识，是一个逻辑地址。

127．0．0．1表示本机地址。（没有网卡该地址仍然可以用）

IP 地址分类：

1. A类地址

A类地址第1字节为网络地址，其它3个字节为主机地址。另外第1个字节的最高位固定为0。

A类地址范围：1.0.0.1到126.155.255.254。

A类地址中的私有地址和保留地址：

10.0.0.0到10.255.255.255是私有地址（所谓的私有地址就是在互联网上不使用，而被用在局域网络中的地址）。

127.0.0.0到127.255.255.255是保留地址，用做循环测试用的。

2. B类地址

B类地址第1字节和第2字节为网络地址，其它2个字节为主机地址。另外第1个字节的前两位固定为10。

B类地址范围：128.0.0.1到191.255.255.254。

B类地址的私有地址和保留地址

172.16.0.0到172.31.255.255是私有地址

169.254.0.0到169.254.255.255是保留地址。如果你的IP地址是自动获取IP地址，而你在网络上又没有找到可用的DHCP服务器，这时你将会从169.254.0.0到169.254.255.255中临得获得一个IP地址。

3. C类地址

C类地址第1字节、第2字节和第3个字节为网络地址，第4个个字节为主机地址。另外第1个字节的前三位固定为110。

C类地址范围：192.0.0.1到223.255.255.254。

C类地址中的私有地址：

192.168.0.0到192.168.255.255是私有地址。

4. D类地址

D类地址不分网络地址和主机地址，它的第1个字节的前四位固定为1110。

D类地址范围：224.0.0.1到239.255.255.254

2、 端口：端口是一个软件抽象的概念。如果把Ip地址看作是一个电话号码的话，端口就相当于分机号。进程一定要和一个端口建立绑定关系。

3、 协议：通讯双方为了完成预先制定好的功能而达成的约定。

4、 TCP/IP网络七层模型：

物理层

数据链路层

网络层：IP协议负责寻止和路由

传输层：TCP协议，UDP协议

会话层：

表示层：

应用层：HTTP,FTP,TELET,SMTP,POPS,DNS

注：层与层之间是一个单向依赖的关系。对等层之间会有一条虚连接。Java中的网络编程就是针对传输层编程；

中国人 美国人

秘书 秘书

翻译　　翻译

发报员　　发报员

5、 网络通信的本质是进程间通信。

6、 Tcp协议和UDP协议

TCP：开销大，用于可靠性要求高的场合。

TCP的过程相当于打电话的过程

UDP：用在对实时性要求比较高的场合。

UDP的过程相当于写信的过程。

五、 网络套节字Socket（TCP）

1、 一个Socket相当于一个电话机。

OutputStream相当于话筒

InputStream相当于听筒

2、 服务器端要创建的对象：java。Net。ServerSocket

3、 创建一个TCP服务器端程序的步骤：

1). 创建一个 ServerSocket

2). 从 ServerSocket 接受客户连接请求

3). 创建一个服务线程处理新的连接

4). 在服务线程中，从 socket 中获得 I/O 流

5). 对 I/O 流进行读写操作，完成与客户的交互

6). 关闭 I/O 流

7). 关闭 Socket

ServerSocket server = new ServerSocket(post)

Socket connection = server.accept();

ObjectInputStream put=new

ObjectInputStream(connection.getInputStream());

ObjectOutputStreamo

put=newObjectOutputStream(connection.getOutputStream());

处理输入和输出流;

关闭流和 socket。

4、建立 TCP 客户端

创建一个 TCP 客户端程序的步骤：

1).创建 Socket

2). 获得 I/O 流

3). 对 I/O 流进行读写操作

4). 关闭 I/O 流

5). 关闭 Socket

Socket connection = new Socket(127.0.0.1, 7777);

ObjectInputStream input=new

ObjectInputStream(connection.getInputStream());

ObjectOutputStream utput=new

ObjectOutputStream(connection.getOutputStream());

处理输入和输出流;

关闭流和 socket。

六、 网络套节字Socket（UDP）

1. UDP编程必须先由客户端发出现息。
2. 一个客户端就是一封信，Socket相当于美国式邮筒（信件的收发都在一个邮筒中）。
3. 端口与协议相关，所以TCP的300端口与UDP的3000端口不是同一个端口

Day14

一、反射

1、 反射主要用于工具的开发。所有的重要 Java 技术底层都会用到反射。反射是一个底层技术。是在运行时动态分析或使用一个类的工具（是一个能够分析类能力的程序）

2、 反射使我们能够再运行时决定对象的生成和对象的调用。

3、 Class

（1）定义：在程序运行期间，Java 运行时系统始终为所有的对象维护一个被称为运行时的类型标识。虚拟机利用类型标识选用相应的方法执行。我们可以通过专门的 Java 类访问这些信息。保存这些信息的类被称为 Class（类类）

（2）类对象（类类，用于存储和一个类有关的所有信息），用来描述一个类的类。类信息通过流读到虚拟机中并以类

对象的方式保存。

注：简单类型也有类对象。

在反射中凡是有类型的东西，全部用类对象来表示。

5. 获得类对象的 3 种方式：

（1）通过类名获得类对象 Class c1 = String.class;

（2）通过对象获得类对象 Class c2 = s.getClass();

（3）通过 Class.forName(“类的全名”)来获得类对象

Class c3 = Class.forName(“Java.lang.String”);

注：第三种方式是最常用最灵活的方式。第三种方式又叫强制类加载。

6. java.lang.reflect .Field 对象用来描述属性信息

7. java.lang.reflect .Constructor 用来描述构造方法信息。

8. java.lang.reflect .Method 用来描述方法信息。

9. 在反射中用什么来表示参数表？

Class[] cs2 = {StringBuffer.class};//表示一个参数表

Constructor c = c1.getConstructor(cs2);//返回一个唯一确定的构造方法。

Class[] cs2 = {String.class,int.class}

Method m = c1.getMethod(methodName,cs3);

10. 可以通过类对象来生成一个类的对象。

Object o = c.newInstance();

10、反射是一个运行时的概念。反射可以大大提高程序的通用性。

一个关于反射的例子：

```
/*************************************************************/

import java.lang.reflect.*;
public class TestClass2 {
    public static void main(String[] args) throws Exception{
        Class    c=Class.forName(args[0]);
        //Object o=c.newInstance();
        //1.得到构造方法对象
        Class[] cs1={String.class};
        Constructor con=c.getConstructor(cs1);
        //2.通过构造方法对象去构造对象
        Object[] os1={args[1]};
        Object o=con.newInstance(os1);
        //3.得到方法对象
        String methodName=args[2];
        Class[] cs2={String.class};
        Method m=c.getMethod(methodName,cs2);
        //4.调用方法
        Object[] os2={args[3]};
        m.invoke(o,os2);

        // Student s=new Student("Liucy");
        // s.study("CoreJava");
    }
}

/*************************************************************/
```

Day16

《Java5.0 新特性》

一、自动封箱和自动解箱技术

1、 自动封箱技术：编译器会自动将简单类型转换成封装类型。

2、 编译器会自动将封装类型转换成简单类型

3、 注：自动封箱和自动解箱只会在必要的情况下执行。

二、静态引用概念：

如：import static java.lang.System.*;

Out.println(“a”);

三、新型 for 循环 for—each，用于追求数组与集合的遍历方式统一

1、 数组举例：

```
String[]  ss  =  {“a”,”b”,”c”};
for(String  s : ss){   // s:遍历元素类型  ss:要遍历的对象
        System.out.println(s);
}
```

2、 集合举例：

```
List  ll  =  new  ArrayList();
 for(Object  o : ll ){
     System.out.println(o);
}
```

注：凡是实现了 java.lang.Iterable 接口的类就能用 for—each 遍历；

四、可变长的参数

一个关于变长参数的例子：

```
/*************************************************************/
import static java.lang.System.*;
public class TestVararg {
   public static void main(String... args){
      m();
      m("Liucy");
      m("Liucy","Hiloo");
   }
   static void m(String... s){
//s可以看作是一个字符串数组String[] s
      out.println("m(String...)");
   }
```

```
    static void m(){
        out.println("m()");
    }
    static void m(String s){
        out.println("m(String)");
    }
}
/*************************************************************/
```

注：一个方法的参数列表中最多只能又一个变长参数，而且这个变长参数必须是最后一个参数。方法调用时只会在必要时去匹配变长参数。

五、枚举 enum

1、 定义：枚举是一个具有特定值的类型，对用户来说只能任取其一。对于面向对象来说时一个类的对象已经创建好，用户不能新生枚举对象，只能选择一个已经生成的对象。

2、 枚举本质上也是一个类。枚举值之间用逗号分开，以分号结束。

3、 枚举分为两种：类型安全的枚举模式和类型不安全的枚举模式

4、 枚举的超类是:Java.lang.Enum。枚举的构造方法默认也必须是 private 并且枚举是一个 final 类所以不能有子类。

5、 一个枚举值实际上是一个公开静态的常量，也是这个类的一个对象。

6、 枚举中可以定义抽象方法，其实现在个个枚举值中（匿名内部类的方式隐含继承）

```
/*************************************************************/
final class Season1{
    public static final Season1 SPRING=new Season1("春");
    public static final Season1 SUMMER=new Season1("夏");
    public static final Season1 AUTUMN=new Season1("秋");
```

```
    public static final Season1 WINTER=new Season1("冬");
    private Season1(){}
    String name;
    private Season1(String name){
        this.name=name;
    }
    public String getName(){
        return this.name;
    }
}
/*****************************************************************/

/*****************************************************************/
enum Season2{
    SPRING("春"),

    SUMMER("夏"),

    AUTUMN("秋"),

    WINTER("冬");
    String name;
    Season2(String name){
        this.name=name;
    }
    public String getName(){
        return this.name;
    }
}
/*****************************************************************/
```

7、 一个关于枚举的例子

```
/*****************************************************************/
import static java.lang.System.*;
public class TestTeacher {
    public static void main(String[] args) {
        for(TarenaTeacher t:TarenaTeacher.values()){
```

```
            t.teach();
        }
    }
}

enum TarenaTeacher{
    LIUCY("liuchunyang"){
    void teach(){out.println(name+" teach UC");}},
    CHENZQ("chenzongquan"){
    void teach(){out.println(name+" teach C++");}},
    HAIGE("wanghaige"){
    void teach(){out.println(name+" teach OOAD");}};
    String name;
    TarenaTeacher(String name){
        this.name=name;
    }
    abstract void teach();
}
/*************************************************************/
```

六、泛型

1、 为了解决类型安全的集合问题引入了泛型。

泛型是一个编译时语法。

2、 List<String> l = new ArrayList<String>();

<String>:表示该集合中只能存放 String 类型对象。

3、 使用了泛型技术的集合在编译时会有类型检查，不再需要强制类型转换。String str = l.get(2);

注：一个集合所允许的类型就是这个泛型的类型或这个泛型的子类型。

4、 List<Object> l = new ArrayList<Integer>×

必须类型一致，泛型没有多态

5、 泛型的通配符<?>

泛型的通配符表示该集合可以存放任意类型的对象。

```
static void print( Cllection<?> c ){
        for( Object o : c )
                out.println(o);
}
```

6、 带范围的泛型通配符

(1)、向下匹配：<? extends Number>

表示该集合元素可以为 Number 类型及其子类型(包括接口)

(2)、向上匹配：<? super Number>

表示该集合元素可以为 Number 类型及其父类型

7、 泛型方法

在返回值与修饰符之间可以定义一个泛型方法

```
public static <T,E extends T> void copy (T[] array,Stack<E> sta){…..}
```

8、 不能使用泛型的情况：

（1)、带泛型的类不能成为 Throwable 类和 Exception 类的子类

因为 cathc()中不能出现泛型。

（2)、不能用泛型来 new 一个对象

如：T t = new T();

（3)、静态方法不能使用类的泛型，因为静态方法中没有对象的概念。

9、 在使用接口的时候指明泛型。

```
class Student implements Comparable<Student>{…….}
```

Day17

一、CoreJava 5.0 的注释

1、 定义：Annotation 描述代码的代码（区：描述代码的文字）

给机器看 ← 给人看的 ←

2、 注释的分类：

（1）、标记注释：没有任何属性的注释。@注释名

（2）、单值注释：只有一个属性的注释。@注释名(value=___)

在单值注释中如果只有一个属性且属性名就是 value，则“value＝”可以省略。

（3）、多值注释：有多个属性的注释。多值注释又叫普通注释。

@注释名（多个属性附值，中间用逗号隔开）

3、 内置注释：

（1）、Override（只能用来注释方法）

表示一个方法声明打算重写超类中的另一个方法声明。如果方法利用此注释类型进行注解但没有重写超类方法，则编译器会生成一条错误消息。

（2）、Deprecated

用 @Deprecated 注释的程序元素，不鼓励程序员使用这样的元素，通常是因为它很危险或存在更好的选择。在使用不被赞成的程序元素或在不被赞成的代码中执行重写时，编译器会发出警告。

（3）、SuppressWarnings（该注释无效）

指示应该在注释元素（以及包含在该注释元素中的所有程序元素）中取消显示指定的编译器警告。

4、 自定义注释

5、 注释的注释：java.lang. annotation 包中

（1）、Target：指示注释类型所适用的程序元素的种类。

例：@Target(value = {ElementType.METHOD});

说明该注释用来修饰方法。

（2）、Retention：指示注释类型的注释要保留多久。如果注释类型声明中不存在 Retention 注释，则保留策略默认为 RetentionPolicy.CLASS。

例：Retention(value = {RetentionPolicy.xxx}

当 x 为 CLASS 表示保留到类文件中，运行时抛弃。

当 x 为 RUNTIME 表示运行时仍保留

当 x 为 SOURCE 时表示编译后丢弃。

（3）、Documented：指示某一类型的注释将通过 javadoc 和类似的默认工具进行文档化。应使用此类型来注释这些类型的声明：其注释会影响由其客户端注释的元素的使用。

（4）、Inherited：指示注释类型被自动继承。如果在注释类型声明中存在 Inherited 元注释，并且用户在某一类声明中查询该注释类型，同时该类声明中

没有此类型的注释，则将在该类的超类中自动查询该注释类型。

注：在注释中，一个属性既是属性又是方法。

二、Java5.0 中的并发

1、 所在的包：Java.util.concurrent

2、 重写线程的原因：

（1）、何一个进程的创建（连接）和销毁（释放资源）的过程都是一个不可忽视的开销。

（2）、run 方法的缺陷：没有返回值，没有抛例外。

3、 对比 1.4 和 5.0 的线程

5.0		1.4
ExecutorService	取代	Thread
Callable Future	取代	Runnable
Lock	取代	Synchronized
SignalAll	取代	notifyAll()
await()	取代	wait()