**《尚学堂》马士兵J2SE视频笔记**

**-- VICTORY LOVES PREPARATION**

**@author Count Monte Cristo**

**@create D20150310**

**@version 2016/08/23 11:19**

**2016/08/23 10:21**

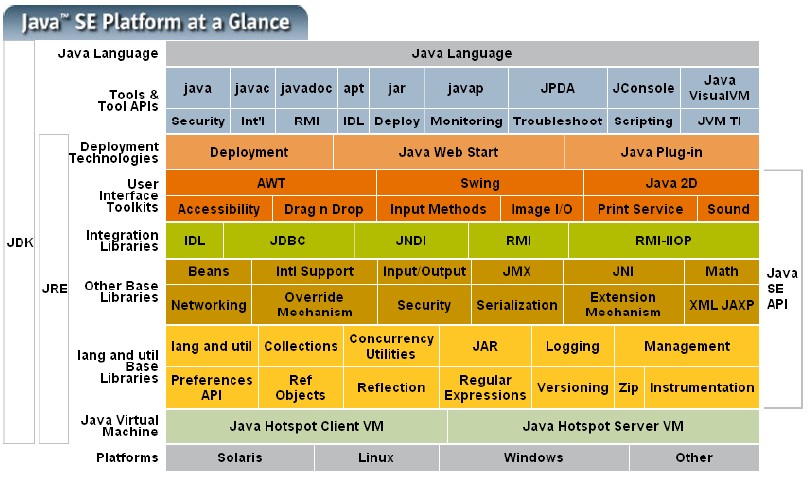
**D20150720**

**@description J2SE基本知识汇总**

**@criteria 凯旋基诺**

**@notice 按照J2SE结构进行系统化**

**No.0 Java SE Platform Structure @version 2016/08/22**

1. 图示
   1. 
2. 说明

**No.2 继承中的构造方法**

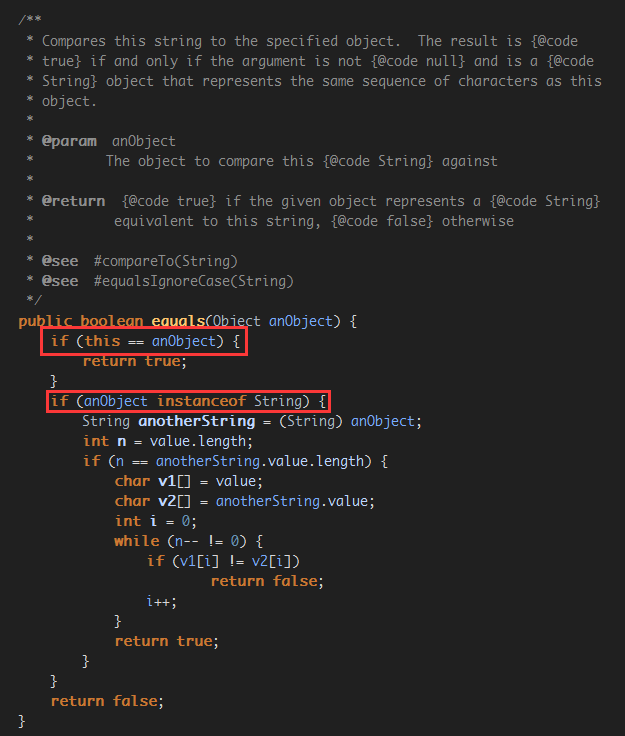
1. （无论是隐式还是显式，隐式也就说默认的无参构造方法，例如，父子类中都没有显示包含构造方法，那么其实子类的隐式构造方法是包含了父类的构造方法）子类中必须调用父类的构造方法（“没有父，哪有子”，如果要手写的话，必须要写（也就是super()）在第一行）；
2. 当类无构造方法的时候，Java会自动给类添加一个无参构造方法，但是当类有构造方法的时候，Java就不会默认添加一个无参数的构造方法。

**No.3 toString()**

1. String类型与其他数据类型连接的时候，将自动调用该类型的toString()方法。
   1. 如System.out.println(“info” + person)，将自动调用person的toString()方法。

**No.4 public boolean java.lang.String.equals(Object obj)**

1. 当obj是一个对象的引用的时候，比较的是地址，当obj是一个字符串序列的时候比较的是字符串序列本身；

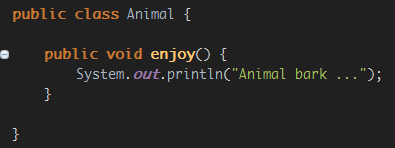
1. **@CODE**
   1. 
2. **注意**
   1. **对于equals()方法，如果重写object的equals方法就涉及到hashCode方法的重写，此时同一个Object的hashCode的值（整数）一定要保证相等**

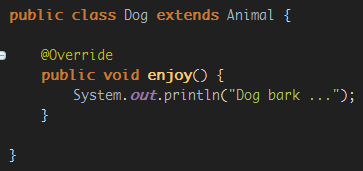
**No.5 动态绑定（又叫作迟绑定或者多态）详解（重点）**

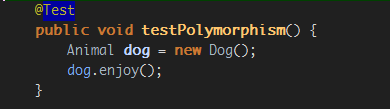
多态一句话：允许将子类类型的引用复制给父类类型的引用。

多态的条件：1.要有引用 2.要有重写 3.父类引用指向子类对象

**@CODE**







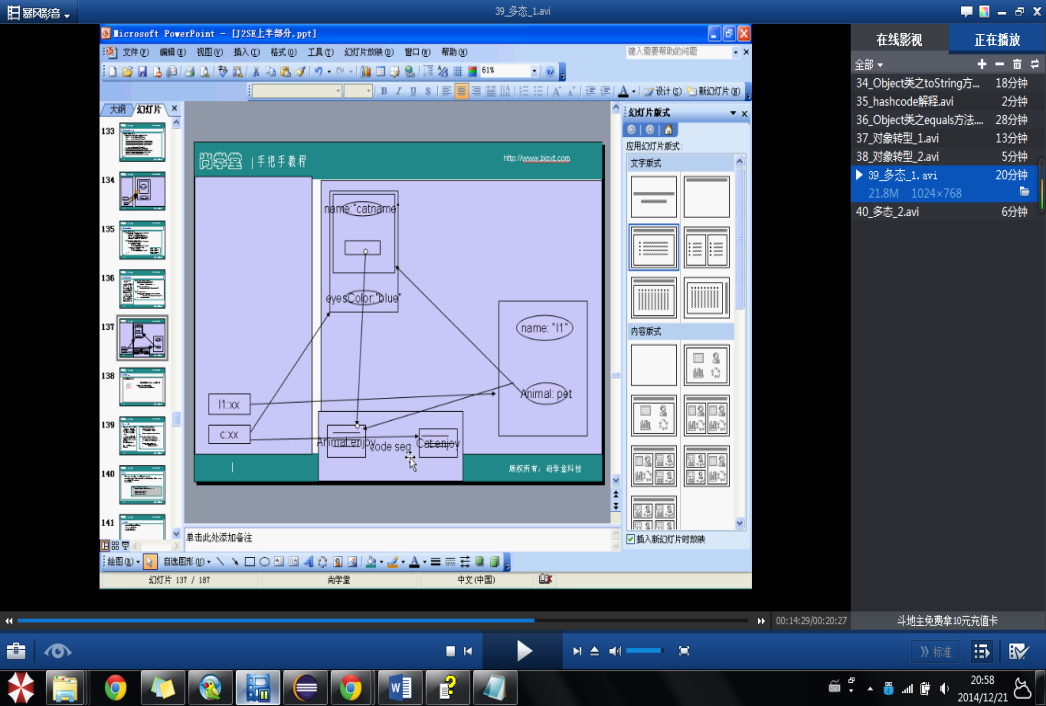
输出应该是Animal bark ...

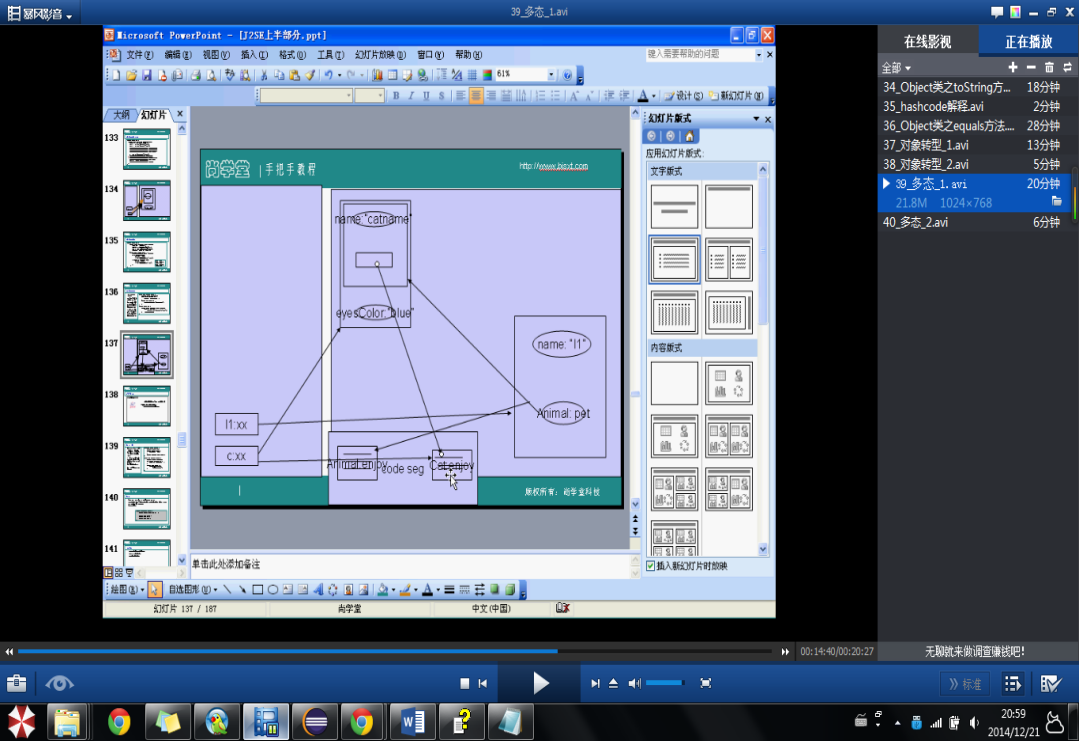
但是，最后的输出结果却是Dog bark ...

原因就是Java的多态机制（又称为动态绑定或者迟绑定）。

其原理解释如下：

先看两张图（程序在heap和stack中运行情况）





从图中可以看出原来指向Animal:enjoy()的指针变成指向Dog:enjoy()了，这就是动态的绑定，只有在Java程序运行的时候这种情况才会出现，实现父类的对象指向子类。

**No.6 final关键字**

1. final的变量不能够被改变
   1. final的成员变量
   2. final的局部变量（包括形参）
2. final的方法不能被重写
3. final的类不能被继承

**No.7 Java中的接口**

1. 多个无关的类可以实现同一个接口
2. 一个类可以实现多个无关的接口
3. 与继承相类似，接口与实现类之间存在多态性
4. （重要）从本质上讲，接口是一种特殊的抽象类，这种类中包含常量和方法的定义，而不包含变量（为了避免实现多个接口的时候出现重名变量，导致混淆）和方法的实现。例子如下：
   1. public interface Runner {

public static final int id = 9;

public void start();

public void run();

public void stop();

}

其中id必须定义为static final ,如果没有写，系统也会默认id是static final 类型；其中的方法start()、run()、stop()都是abstract的，写和不写接口中的方法都是abstract类型的。接口中的方法都是public类型的无论是否声明public了

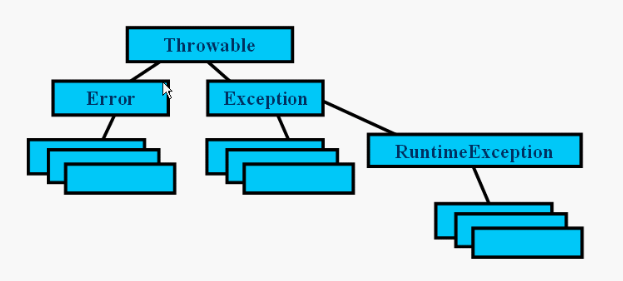
**No.8 Java中的面相对象总结**

1. 内存分析贯穿始终
2. 对象和类的概念
3. 类（对象）之间的关系
4. 面相对象设计思想
5. Class
6. New
   1. 引用的概念
   2. 构造方法的概念
7. 方法的重载
   1. 构造方法的概念
8. This
9. Static（static 变量属于类，非static变量属于对象）
10. Package（打包是为了解决类重名的问题）& import
11. Private default protected public
12. Extends（类和类之间、接口和接口之间相互继承）
13. Overwrite(override)
14. Final
15. Object
    1. toString()
    2. equals()
16. Upcasting downcasting
17. Polymorphism/dynamic binding/late binding
18. Abstract class
19. Interface（接口和接口之间是可以继承的）
    1. Implements

**No.9 Java中的异常（欠缺异常的总结，包含结构图、注意事项等）**

1. Java异常是Java提供的用于处理程序中的异常（实际生活中指的错误的一种）的一种机制
2. 设计良好的程序应该在异常发生的时候提供处理异常的方法，使程序不会应为异常的发生而阻断或产生不可预见的结果
3. Java程序在运行中如果出现异常事件，则会产生一个异常类对象，该对象封装了异常事件的信息并将被提交给Java运行时系统，这个过程叫做抛出（throw）异常
4. 当Java运行时系统接收到异常的时候，会寻找能处理这一异常的代码，并把当前的异常对象交给其处理，这一过程称为捕获（catch）异常

**No.10 Java中什么时候必须要捕获异常**



上面就是异常的分类，Throwable是异常的祖先（Object是所有类的祖先，包括Throwable）,其中Error抛出的是系统的错误（**表示系统错误或低层资源的错误**），不用程序员管，Exception抛出的是异常（**由程序员导致的错误**），其中RuntimeException（系统会自动捕获）代表运行时常出现的异常，这类异常可以捕获也可以不捕获，其他的异常程序必须捕获。

注意：异常和错误的区别和联系

**联系**：

都继承Throwable类

**区别**

**异常**

可以是可被控制的（checked）或者是不可控制的（unchecked）

**表示一个由程序员导致的错误**

应该在应用程序级别进行处理

**错误**

总是不可控制的（unchecked）

**经常表示系统错误或者底层资源的错误**

**一般程序员无法处理，也不需要进行处理**

**综上所述：异常是一种错误（由程序员导致，但是这种错误是实际生活中的错误，不是程序语言中的错误，后者指系统错误或者 底层资源的错误）**

**No.12 Java异常处理5个关键字（try、catch、finally、throw、throws）**

**No.13 Java中继承抛出异常类子类方法要不要抛出异常情况**

重写方法需要抛出与原方法所抛出异常类型一致异常或者不抛出异常，其余无论什么情况都不可以

**No.14 Java中数组初始化**

1. 数组初始化分为静态初始化和动态初始化
   1. 静态初始化：定义数组的时候就开始赋值
   2. 数组定义与数组元素分配空间和赋值操作分开

**No.15 Java中的length(数组中和String类中)**

1. 数组中，length是属性，所以调用的时候只需要a.length（其中a是数组对象的引用）
2. String类中，length的方法，所以调用的时候要a.length()（其中a是String类对象的引用）

**No.16 Java中main(String[] args)中args的用法和作用**

Public static void main(String[] args){}

其中的args的接受Java命令的命令行参数作用，把接收到的参数存在String[]类型的args变量中。

**No.17 Java中的exit()代表的意思**

1. exit(0)表示正常退出（退出当前虚拟机）
2. exit(-1)非正常退出（退出当前虚拟机）

在Windows的环境下，这两种情况看不出区别，所以在使用的时候用哪一个都行

**No.18 Java中的toString()介绍一处注意点**

1. 一般来说toString()返回的是字符串本身，例子如下：

Test test = new Test();

System.out.println(test);//此时会自动在test后面加上toString()方法

如果Test类中重写了toString()方法则自动调用（就算不指出调用toString()方法，也就是说此时的System.out.println(test)等价于System.out.println(test.toString())）Test中的toString()方法，否则调用父类的toString()也就是我们通常指的toString()方法。

**No.19 Java中的数组**

1. 多维数组就是一维数组的叠加
2. Int a[][] = new int[3][]是正确的然后可以a[0] = new int[2];a[1] =new int[4];a[3] = new int[3];但是不能写成inta[][] = new int[][4];也就是说Java中数组的初始化必须要按照先高维再低维的顺序（当然高维低维一起也行）
3. 二维数组初始化
   1. 静态初始化
      1. Int[][] a = {{1,2}，{2,3}，{3,4，5}}；是正确的
      2. Int[3][2] a ={{1,2},{2,2},{2,3}}是错误的，也就是说Java中属静态赋值的时候不需要指定大小，Java系统会自动指定，为了不必要的麻烦。

**No.20 Java中String的实例化**

1. String s1 =”hello”;String s2 = “hello”: s1==s2
   1. 原因：此时的”hello”被放在了Data segment中，而不是作为引用的对象放在堆中
2. s1 = new String(“hello”); s2= new String(“hello”); s1 != s2 s1.equals(s2) == true;
   1. 原因：此时的”hello” 是作为引用对象放在堆中的，s1和s2分别是指向”hello”对象的引用；而方法equals()只要字符串序列相同就相等。

**No.21 Java中String类常用的方法**

**@reference Java Platform, Standard Edition ? API Specification**

**No.22 Java中String和StringBuffer效率比较**

1. String类型一旦声明是不可以变化的，StringBuffer的内容和长度是可以变化的,其余没有什么太大的区别
   1. 例如String s1= “hello”;String s2 = “world”;

s1 += s2;

System.out.println(s1);//helloworld

疑问：为什么String类型的s1内容为什么还能变化呢，解释如下：

“world”

“hello”

“helloworld”

S2

S1

也就是说后来的s1已经不是原来的s1了，它已经是指向新的s1的引用了，而StringBuffer则不需要这么麻烦（效率较低），因为StringBuffer可以直接修改字符串的内容。

**No.23 Java中StringBuffer的用法**

1. 常见构造方法
   1. public String StringBuffer(){}
   2. public String StringBuffer(String str)
2. 常用方法
   1. Public StringBuffer append(…) StringBuffer对象后面加上…，常见形式有：
      1. Public StringBuffer append(String str)
      2. Public StringBuffer append(StringBuffer sbuf)
      3. Public StringBuffer append(char[] str)
      4. Public StringBuffer append(char[] str, int offset,int len)
      5. Public StringBuffer append(double d)
      6. Public StringBuffer append(Object obj)
      7. …..
   2. Public StringBuffer insert(int offset,…)
      1. Public StringBuffer insert(int offset, String str)
      2. Public StringBuffer insert(int offset, double d)
   3. Public StringBuffer delete(int start, int end)
   4. Public StringBuffer reverse()//字符串逆序
   5. StringBuffer和String类似的方法
      1. Public int indexOf(String str)
      2. Public int indexOf(String str, int fromIndex)
      3. Public String substring(int start)
      4. Public String substring(int start, int end)
      5. Public int length()
3. 实例

**package** demo;

**publicclass** TestStringBuffer {

**publicstaticvoid** main(String[] args) {

String str = "Fokuso";

**char**[] c = {'i', 's'};

StringBuffer sb1 = **new** StringBuffer(str);

sb1.append(' ').append(c);

System.***out***.println(sb1);

String str2 = " a good student.";

sb1.append(str2);

System.***out***.println(sb1);

sb1.delete(0, sb1.length() - 2).insert(1, "nihao");

System.***out***.println(sb1);

}

}

**No.24 Java中的Math类**

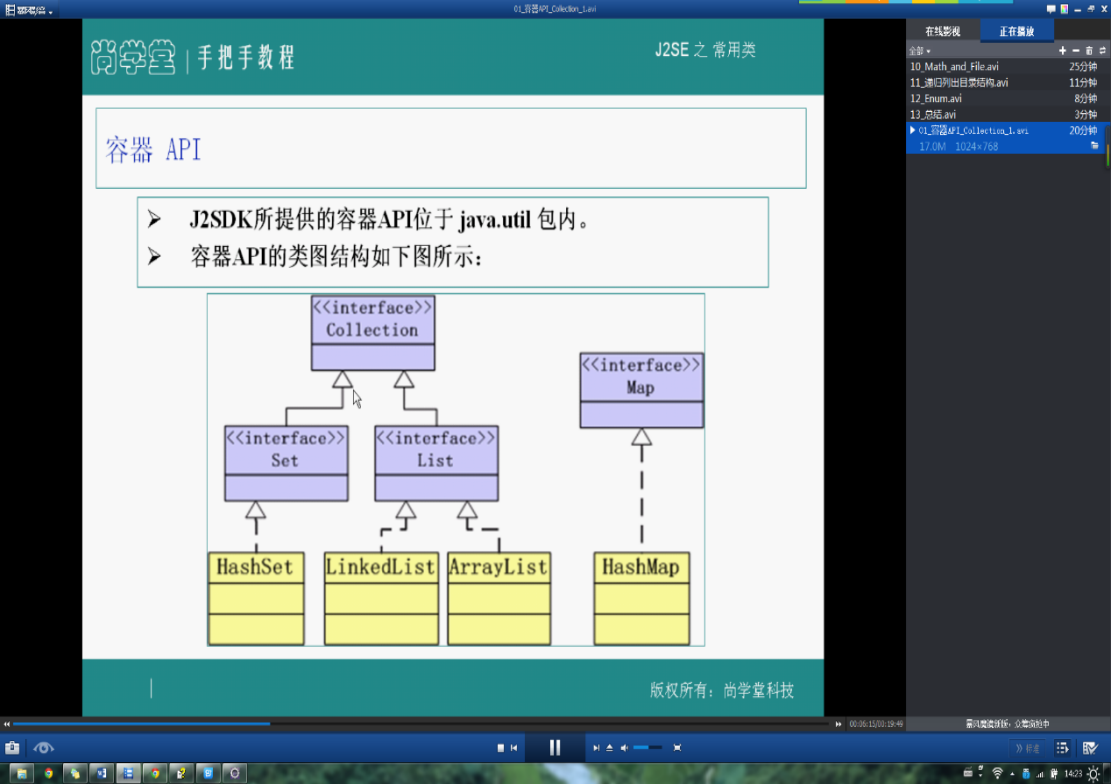
1. Math类中大部分方法都是静态方法，参数都是double类型
2. Public double random()随机产生大于等于0小于1的double类型数
3. Public double sqrt(double d)
4. Public static long round(double f)

**No25 Java中的文件操作**

1. 常用通过File对象访问文件的属相的方法：
   1. Public booean canRead()
   2. Public Boolean canWrite()
   3. Public booean exists()
   4. Public booean isDirectory()
   5. Public booean isFile()
   6. Public booean isHidden()
   7. Public long lastModified()
   8. Public long length()
   9. Public String getName()
   10. Public String getPath()
2. 通过File对象创建空文件或目录（目录不存在的
3. 况下）
   1. Public booean createNewFile() throws IOException
   2. Public breooean delete()
   3. Public Boolean mkdir()
   4. Public Boolean mkdirs()//创建一系列的目录

**No.26 Java中导入API文档，点击“attach source”然后添加src.zip即可**

**No.27 Java中的容器总结**

1. 容器关系
2. 详解
   1. Collection是一个一个地填充的，Map是一对一对地填充的
   2. Set中的数据没有顺序，不重复，List中的数据有顺序，可重复（equals()==ture）
   3. Map接口定义了存储“键（key）-值（value）映射对”方法
   4. Collection接口中常见的方法：
      1. Int size()
      2. Boolean isEmpty()
      3. Void clear()
      4. Boolean contains(Object element)
      5. Booean add(Object element)
      6. Boolean remove(Object element)
      7. Iterator iterator()
      8. Boolean containsAll(Collection c)
      9. Boolean addAll(Collection c)
      10. Boolean removeAll(Collection c)
      11. Boolean retainAll(Collection c)//求交集
      12. Object[] toArray()

**No.28 Java中的Iterator接口（对Collection对象做遍历的时候使用）**

1. 常用方法
   1. Boolean hasNext() //判断游标右边是否有元素
   2. Object next()//返回游标右边的元素并将游标移动到下个位置
   3. Void remove()//删除游标左边的元素，在执行完next之后该操作只能执行一次

**No.29 Java中增强版for(JDK1.5开始增强的，内部实质还是调用Iterator接口中的iterator()方法，除了简单的遍历，不建议使用该for循环)**

**int**[] arr = { 1, 2, 3, 4, 5 };

**for** (**int** i : arr) {

System.*out*.println(i);

}

Collection collection = **new**ArrayList();

collection.add(**new** String("abc"));

collection.add(**new** String("def"));

collection.add(**new** String("ghi"));

**for** (Object c : collection) {

System.*out*.println(c);

Result:会一一打印出来，集合的遍历，使用for进行，按照加入对象或者数据的顺序打印出来的

No.30 Java中的Set（按照上面的图顺序介绍的）

**import** java.util.\*;

**publicclass** TestSet {

**publicstaticvoid** main(String[] args) {

Set s1 = **new**HashSet();

Set s2 = **new**HashSet();

s1.add("a");

s1.add("b");

s1.add("c");

s2.add("a");

s2.add("b");

s2.add("d");

Set s3 = **new** HashSet(s1);

Set s4 = **new** HashSet(s1);

s3.retainAll(s2);

s4.addAll(s2);

System.*out*.println(s3);

System.*out*.println(s4);

}

}

No.30 Java中的List和Collection

1. List(ArrayList and LinkedList)
   1. List常用算法举例（注意Java.util.Collections类中封装了List操作的很多方法，大多都是静态的）
      1. Void sort(List) 排序（默认从小到大）
      2. Void shuffle(List) 随机排序
      3. Void reverse(List) 逆序
      4. Void fill(List, Object) 用一个特定的对象重写整个List容器
      5. Void copy(List dest, List src) 拷贝
      6. Int binarySearch(List, Object)
   2. 实例
      * 1. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Collections.binarySearch()失效\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/
        2. **import**java.util.Collections;
        3. **import** java.util.LinkedList;
        4. **import** java.util.List;
        5. **publicclass** TestList {
        6. @SuppressWarnings("unchecked")
        7. **publicstaticvoid** main(String[] args) {
        8. List list1 = **new**LinkedList();
        9. **for** (**int** i = 0; i < 10; i++) {
        10. list1.add("a" + i);
        11. }
        12. Collections.*shuffle*(list1);
        13. System.*out*.println(list1);
        14. Collections.*reverse*(list1);
        15. System.*out*.println(list1);
        16. Collections.*sort*(list1);
        17. System.*out*.println(list1);
        18. System.*out*.println(Collections.*binarySearch*(list1, "a2"));
        19. /\* 不能输出对应的a2编号，如果把Collections.sort(list1)和Collections.reverse(list1)换个位置就可以，这是为什么呢？\*/
        20. }
        21. }
2. Collection

No.31 Java中子类重写父类的方法应该注意的事项（特别要注意规则四）

A、重写规则之一：

重写方法不能比被重写方法限制有更严格的访问级别。

（但是可以更广泛，比如父类方法是包访问权限，子类的重写方法是public访问权限。）比如：Object类有个toString()方法，开始重写这个方法的时候我们总容易忘记public修饰符，编译器当然不会放过任何教训我们的机会。出错的原因就是：没有加任何访问修饰符的方法具有包访问权限，包访问权限比public当然要严格了，所以编译器会报错的。

B、重写规则之二：

参数列表必须与被重写方法的相同。

重写有个孪生的弟弟叫重载，也就是后面要出场的。如果子类方法的参数与父类对应的方法不同，那么就是你认错人了，那是重载，不是重写。

C、重写规则之三：

返回类型必须与被重写方法的返回类型相同。

父类方法A：void eat(){} 子类方法B：int eat(){} 两者虽然参数相同，可是返回类型不同，所以不是重写。

父类方法A：int eat(){} 子类方法B：long eat(){} 返回类型虽然兼容父类，但是不同就是不同，所以不是重写。

D、重写规则之四：

重写方法不能抛出新的异常或者比被重写方法声明的检查异常更广的检查异常。但是可以抛出更少，更有限或者不抛出异常。

import java.io.\*;

public class Test {

public static void main (String[] args) {

Animal h = new Horse();

try {

h.eat();

}

catch (Exception e) {

}

}

}

class Animal {

public void eat() throws Exception{

System.out.println ("Animal is eating.");

throw new Exception();

}

}

class Horse extends Animal{

public void eat() throws IOException{

System.out.println ("Horse is eating.");

throw new IOException();

}

}

这个例子中，父类抛出了检查异常Exception，子类抛出的IOException是Exception的子类，也即是比被重写的方法抛出了更有限的异常，这是可以的。如果反过来，父类抛出IOException，子类抛出更为宽泛的Exception，那么不会通过编译的。

注意：这种限制只是针对检查异常，至于运行时异常RuntimeException及其子类不再这个限制之中。

E、重写规则之五：

不能重写被标识为final的方法。

F、重写规则之六：

如果一个方法不能被继承，则不能重写它。

比较典型的就是父类的private方法。下例会产生一个有趣的现象。

public class Test {

public static void main (String[] args) {

//Animal h = new Horse();

Horse h = new Horse();

h.eat();

}

}

class Animal {

private void eat(){

System.out.println ("Animal is eating.");

}

}

class Horse extends Animal{

public void eat(){

System.out.println ("Horse is eating.");

}

}

这段代码是能通过编译的。表面上看来违反了第六条规则，但实际上那是一点巧合。Animal类的eat()方法不能被继承，因此Horse类中的 eat()方法是一个全新的方法，不是重写也不是重载，只是一个只属于Horse类的全新的方法！这点让很多人迷惑了，但是也不是那么难以理解。

main()方法如果是这样：

Animal h = new Horse();

//Horse h = new Horse();

h.eat();

编译器会报错，为什么呢？Horse类的eat()方法是public的啊！应该可以调用啊！请牢记，多态只看父类引用的方法，而不看子类对象的方法！

No.32 Java中比较任意数据类型集合中元素大小的方法

1. 具体思路：把数据封装到类中
2. 方法：
   1. 可以自定义写一个比较方法
   2. 可以在自己定义的类中实现Comparable接口中的CompareTo方法
3. 实例（只给出了第二种方法，第一种方法见笔记）
   * + 1. **import** java.util.Collections;
       2. **import** java.util.List;
       3. **import** java.util.ArrayList;
       4. **publicclass** TestComparable {
       5. **publicstaticvoid** main(String[] arsg) {
       6. List list = **new**ArrayList();
       7. list.add(**new** Name("fda", "d"));
       8. list.add(**new** Name("fa", "c"));
       9. list.add(**new** Name("fdas", "a"));
       10. System.*out*.println(list);
       11. Collections.*sort*(list);
       12. System.*out*.println(list);
       13. }
       14. }
       15. **class** Name **implements**Comparable {
       16. **private** String firstname;
       17. **private** String lastname;
       18. **public** Name(String firstname, String lastname) {
       19. **this**.firstname = firstname;
       20. **this**.lastname = lastname;
       21. }
       22. **public** String toString() {
       23. **return**firstname + " " + lastname;
       24. }
       25. **publicint** compareTo(Object obj) {
       26. // **TODO** Auto-generated method stub
       27. Name name = (Name) obj;
       28. **int** compareResult = lastname.compareTo(name.lastname);
       29. **return** compareResult != 0 ? compareResult : firstname
       30. .compareTo(name.firstname);
       31. }
       32. }

No.33 Java中Array、Linked和Hash之间的效率比较（主要是读和改的效率）

1. Array 读快改慢
2. Linked 改快读慢
3. Hash 两者之间

No.34 Java中Map的使用

1. Map常用方法
   1. Object put(Object key, Object value)
   2. Object get(Object key)
   3. Object remove(Object key)
   4. Boolean containsKey(Object key)
   5. Boolean containsValue(Object key)
   6. Int size()
   7. Boolean isEmpty()
   8. Void putAll(Map t)
   9. Void clear()
2. 应用实例
   * + 1. **import** java.util.\*;
       2. **publicclass** TestMap {
       3. **publicstaticvoid** main(String[] args) {
       4. Map m1 = **new**HashMap();
       5. Map m2 = **new**TreeMap();
       6. m1.put("one", **new** Integer(1));
       7. m1.put("two", **new** Integer(2));
       8. m1.put("three", **new** Integer(3));
       9. m2.put("A", **new** Integer(1));
       10. m2.put("B", **new** Integer(2));
       11. System.*out*.println(m1.size());
       12. System.*out*.println(m1.containsKey("one"));
       13. System.*out*.println(m1.containsValue(**new** Integer(2)));
       14. **if** (m1.containsKey("two")) {
       15. **int** i = (Integer) m1.get("two");
       16. System.*out*.println(i);
       17. }
       18. Map m3 = **new** HashMap(m1);
       19. m3.putAll(m2);
       20. System.*out*.println(m3);
       21. }
       22. }

No.35 Java中的打包和解包（Auto自动化进行是在JDK1.5之后（含）才行的）

1. 打包（Auto-Boxing）：
   1. 自动将类型转换成对象
2. 解包（Auto-UnBoxing）：
   1. 自动将对象转换成类型

No.36 Java中的Map自动打包和解包

**import** java.util.\*;

**publicclass** TestMap {

**publicstaticvoid** main(String[] args) {

Map<String, Integer> m1 = **new** HashMap<String, Integer>();

Map<String, Integer> m2 = **new** TreeMap<String, Integer>();

//m1.put("one", new Integer(1));

m1.put("one", 1);//此行是上一行的简写版（JDK1.5及以后开始）注意//此时m1中的两个参数依旧是对象（字符串对象和整型对象），存在堆中，而不是字//符串和整型数字，涉及到了自动打包原理（解包在下面）

m1.put("two", 2);

m1.put("three", 3);

m2.put("A", 1);

m2.put("B", 2);

System.*out*.println(m1.size());

System.*out*.println(m1.containsKey("one"));

System.*out*.println(m1.containsValue(**new** Integer(2)));

**if** (m1.containsKey("two")) {

**int** i = (Integer) m1.get("two");

System.*out*.println(i);

}

Map<String, Integer> m3 = **new** TreeMap<String, Integer>(m2);

m3.putAll(m2);

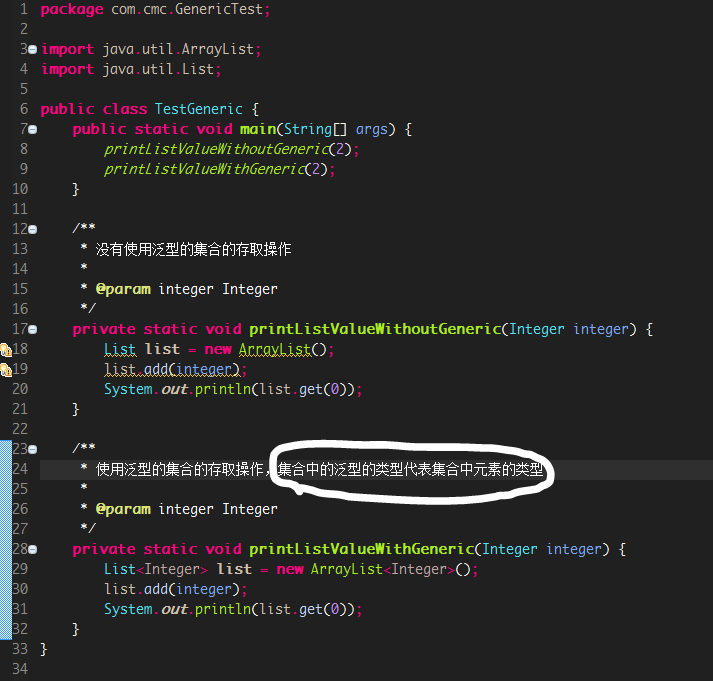
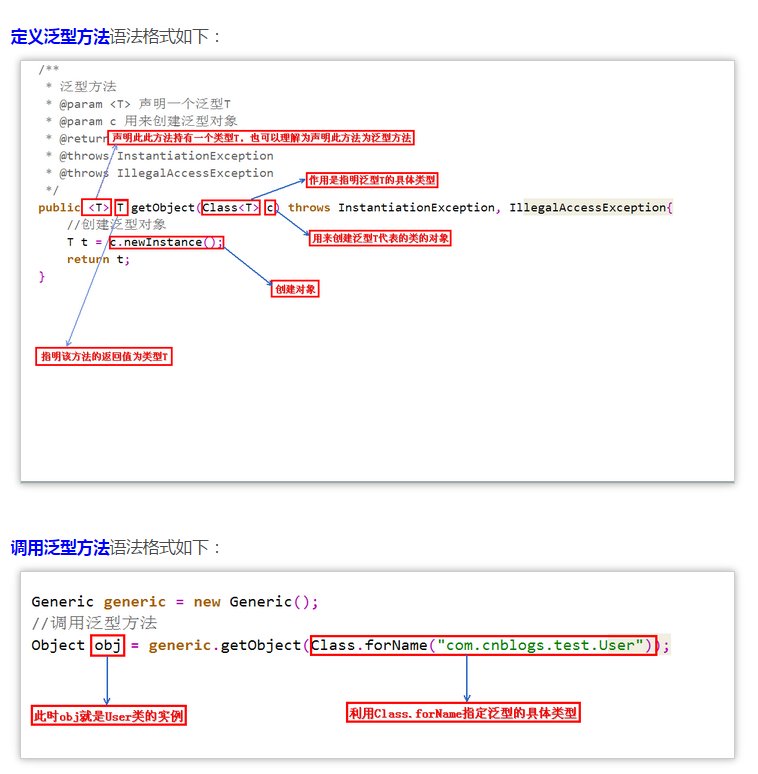
System.*out*.println(m3);

}

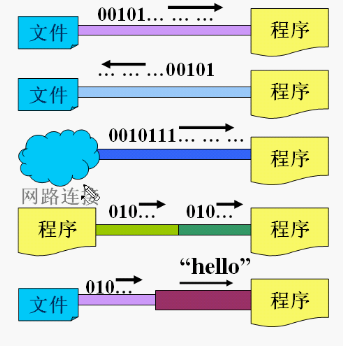
}

**No.37 Java中的泛型**

1. @reference <http://www.cnblogs.com/iyangyuan/archive/2013/04/09/3011274.html>
2. 起因
   1. JDK1.4以前类型不明确
      1. 装入集合的类型都被当做Object对待，从而失去自己的实际类型。
      2. 从集合中取出时往往需要转型，效率低，容易产生错误。
3. 解决方法
   1. 在定义集合的时候同时定义集合中对象的类型
   2. 实例：BasicGeneric.java
      1. 可以在定义Collection的时候制定
      2. 可以在循环时用Iterator指定
4. 优点
   1. 增强程序的可读性和稳定性
5. 实例
   1. 说明：警告的内容是List等类型采用了原始的数据类型，建议使用Generic（泛型）加上集合的参数
   2. 集合中泛型的类型代表集合中元素的类型



**No.38 Java中流（JDK中所有的流位于java.io包内）**

1. **本章内容**
   1. Java流式输入/输出原理
   2. java流式的分类
   3. 输入/输出流
   4. 常见的节点流和处理流
   5. 文件流
   6. 缓冲流
   7. 数据流
   8. 转换流
   9. Print流
   10. Object流
2. **Java流式输入/输出原理**
   1. Java程序中，对于数据的输入/输出操作以“流”（stream）方式进行；
   2. J2SE提供了各种各样的“流”类，用以获取不同种类的数据；程序中通过标准的方法输入或输出数据。
3. **Java流的分类**
   1. **输入流和输出流**
      1. 按照数据流的方向不同分为输入流和输出流

输入输出是以内存为主语的

* 1. **字节流和字符流**
     1. 按照处理数据单位不同分为字节流和字符（两个字节）流
  2. **节点流和处理流**
     1. 按照功能不同分为节点流（从一个特定的数据源（节点，如：文件，内存）读写数据）和处理流（“连接”在已经存在的流（节点流或者处理流）上，通过对数据的处理为程序提供更为强大的读写功能）

1. **节点流类型**
   * + 1. **节点流的分类（字节流和字符流）**



* + - 1. **程序实例**
         1. **FileInputStream和FileOutputStream(字节流)**

/\*\*

\* 实现复制文件的功能

\*/

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.FileOutputStream;

**import** java.io.IOException;

**publicclass** TestFileOutputStream {

**publicstaticvoid** main(String[] args) {

FileInputStream fileInputStream = **null**;

FileOutputStream fileOutputStream = **null**;

String separator = File.*separator*;

String fileInputPath = "." + separator + "file.java";

String fileOutputPath = "." + separator + "testss.java";

**int** data = 0;

**try** {

fileInputStream = **new** FileInputStream(fileInputPath);

fileOutputStream = **new** FileOutputStream(fileOutputPath);

**while** ((data = fileInputStream.read()) != -1) {

fileOutputStream.write(data);

}

fileInputStream.close();

fileOutputStream.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

System.*out*.println("文件已经被复制。");

}

}

* + - * 1. **FileReader和FileWriter（字符流）**

**package** practice.java.lcb.sxt;

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**publicclass** TestFileReaderWriter {

**publicstaticvoid** main(String[] args) {

FileReader fileReader = **null**;

FileWriter fileWriter = **null**;

**int** data = 0;

String separator = File.*separator*;

String fileReaderName = "FileReader.java";

String fileWriterName = "FileWriter.java";

String fileReaderPath = "." + separator + fileReaderName;

String fileWriterPath = "." + separator + fileWriterName;

**try** {

fileReader = **new** FileReader(fileReaderPath);

fileWriter = **new** FileWriter(fileWriterPath);

**while** ((data = fileReader.read()) != -1) {

fileWriter.write(data);

}

fileReader.close();

fileWriter.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

System.*exit*(-1);

} **catch** (IOException e) {

e.printStackTrace();

System.*exit*(-1);

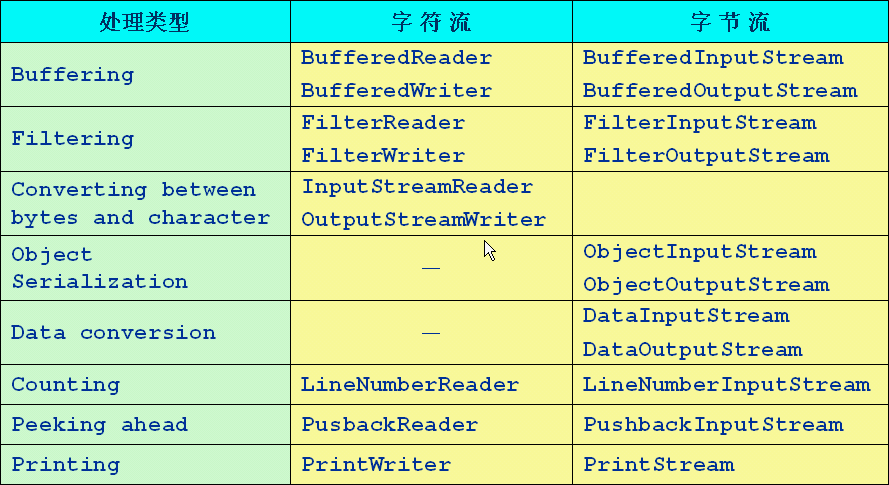
} **finally** {

System.*out*.println("文件复制成功！");

}

}

}

1. **处理流类型**
   * + 1. **处理流的分类**
          1. **缓冲流（减少直接写入硬盘的次数，加快速度，保护硬盘）**(bufferedInputStream+bufferedOutputStream+bufferedWriter+bufferedReader)

**“套接”在相应的节点流之上，对读写的数据提供了缓冲的功能，提高了读写的效率，同时增加了一些新的方法**

**J2SE提供了四种缓冲流，其常见的结构（构造方法）为：**

BufferedReader(Reader in)

BufferedReader(Reader in, int size)

BufferedWriter(Writer out)

BufferedWriter(Writer out, int size)

BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, int size)

BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out. Int size)

**注意事项**

缓冲输入流支持其父类的mark和reset方法

BufferedReader提供了readLine()方法用于读取一行字符串（以\r和\n分隔）

BufferedWriter提供了newLine()用于写入一个行分隔符

对于输出的缓冲流，写出的数据会先在内存中缓存，使用flush()方法将会使内存中的数据立刻写出。

**程序实例（BufferedInputStream）**

BufferedInputStream实例

**package** practice.java.lcb.sxt;

**import** java.io.BufferedInputStream;

**import** java.io.File;

**import** java.io.FileInputStream;

**import** java.io.FileNotFoundException;

**import** java.io.IOException;

**Public class** TestBufferedInputStream {

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

FileInputStream fileInputStream = **null**;

BufferedInputStream bufferedInputStream = **null**;

String filename = "BufferedInputStream.java";

String separator = File.*separator*;

String filePath = "." + separator + filename;

**int** data = 0;

**try** {

fileInputStream = **new** FileInputStream(filePath);

bufferedInputStream = **new** BufferedInputStream(fileInputStream);

System.*out*.println((**char**) bufferedInputStream.read());

System.*out*.println((**char**) bufferedInputStream.read());

**for** (**int** i = 0; i < 10 && (data = bufferedInputStream.read()) != -1; i++) {

System.*out*.print("" + (**char**) data);

}

fileInputStream.close();

bufferedInputStream.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

BufferedReader和BufferedWriter实例

**package** practice.java.lcb.sxt;

**import** java.io.BufferedReader;

**import** java.io.BufferedWriter;

**import** java.io.File;

**import** java.io.FileReader;

**import** java.io.FileWriter;

**import** java.io.IOException;

**publicclass** TestBufferedWriterBufferedReader {

**publicstaticvoid** main(String[] arsgs) {

BufferedWriter bufferedWriter = **null**;

BufferedReader bufferedReader = **null**;

FileWriter fileWriter = **null**;

FileReader fileReader = **null**;

String separator = File.*separator*;

String fileReaderName = "BufferedReader.java";

String fileWriterName = "BufferedReader.java";

String fileReaderPath = "." + separator + fileReaderName;

String fileWriterPath = "." + separator + fileWriterName;

**try** {

fileWriter = **new** FileWriter(fileWriterPath);

fileReader = **new** FileReader(fileReaderPath);

bufferedWriter = **new** BufferedWriter(fileWriter);

bufferedReader = **new** BufferedReader(fileReader);

String str = **null**;

**for** (**int** i = 0; i < 100; i++) {

str = String.*valueOf*(Math.*random*());

bufferedWriter.write(str);

bufferedWriter.newLine();

}

bufferedWriter.flush();

**while** ((str = bufferedReader.readLine()) != **null**) {

System.*out*.println(str);

}

fileWriter.close();

fileReader.close();

bufferedWriter.close();

bufferedReader.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

}

* + - * 1. **转换流(OutputSteamWriter+InputStreamReader)**

**注意事项**

InputStreamWriter和InputStreamReader用于字节数据到字符数据之间的转换；

InputStreamReader需要和InputStream“套接”，OutputStreamWriter需要和OutputStream“套接”；

转换流在构造的时候可以指定编码集合，例如：InputStreamReader inputStreamReader = new InputStreamReader(new InputStream(Obj),”GBK”);

程序实例(TestInputStreamReader)

package practice.java.lcb.sxt;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

public class TestInputStreamReader {

public static void main(String[] arsgs) {

InputStreamReader inputStreamReader = null;

BufferedReader bufferedReader = null;

String str = null;

inputStreamReader = new InputStreamReader(System.in);

bufferedReader = new BufferedReader(inputStreamReader);

try {

while ((str = bufferedReader.readLine()) != null) {

if (str.equalsIgnoreCase("exit"))

break;

System.out.println(str.toUpperCase());

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

* + - * 1. **数据流(DataInputStream+DataOutputStream)**

**注意事项**

DataInputStream和DataOutputStream分别继承InputStrem和OutputStream,是处理流，它们需要“套接”在InputStream和OutputStream节点流上面；

DataInputStream和DataOutputStream提供了可以存取与机器无关的Java原始类型数据的方法；也就是说，可以通过它们提供的方法进行Java原始数据的存取操作；

构造方法

DataOutputStream(OutputStrem out)

DataInputStream(InputStream in)

**程序实例**

package practice.java.lcb.sxt;

import java.io.ByteArrayInputStream;

import java.io.ByteArrayOutputStream;

import java.io.DataInputStream;

import java.io.DataOutputStream;

import java.io.EOFException;

import java.io.IOException;

public class TestDataOutputStreamDataInputStream {

public static void main(String[] args) {

ByteArrayOutputStream byteArrayOutputStream = null;

ByteArrayInputStream byteArrayInputStream = null;

DataOutputStream dataOutputStream = null;

DataInputStream dataInputStream = null;

byteArrayOutputStream = new ByteArrayOutputStream();

dataOutputStream = new DataOutputStream(byteArrayOutputStream);

try {

dataOutputStream.writeDouble(Math.random());

dataOutputStream.writeBoolean(true);

byteArrayInputStream = new ByteArrayInputStream(byteArrayOutputStream.toByteArray());

dataInputStream = new DataInputStream(byteArrayInputStream);

System.out.println(dataInputStream.available());

System.out.println(dataInputStream.readDouble());//此处一定要先readDouble()再readBoolean(),因为DataOutputStream存在内存中的队列当中的

System.out.println(dataInputStream.readBoolean());

} catch (IOException e) {

e.printStackTrace();

}

}

}

* + - * 1. **Print流(PrintWriter和PrintStream)(重新定义标准输出流，本来默认的输出目标是控制台的，通过Print流改变输出目标，例如可以把输出目标定义到一个文件上面或者是内存等上面。)**

**注意事项**

PrintWriter和PrintStream都属于输出流，分别针对于字符和字节；

PrintWriter和PrintStream提供了重载的print()和println()方法；

PrintWriter和PrintStream的输出操作不会抛出异常；

PrintWriter和PrintStream有自动flush()功能。

构造方法

PrintWriter(Writer out)

PrintWriter(Writer out,Boolean autoFlush)

PrintWriter(OutputStream out)

PrintWriter(OutputStream out, Boolean autoFlush)

PrintStream(OutputStream out)

PrintStream(OutputStream out, Boolean autoFlush)

**程序实例**

//重新分配标准输出流

package practice.java.lcb.sxt;

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.PrintStream;

public class TestPrintStream {

public static void main(String[] args) {

PrintStream printStream = null;

FileOutputStream fileOutputStream = null;

String filename = "TestPrintStream.txt";

String separator = File.separator;

String fileParentDirectory = "TxtFiles";

String filePath = "." + separator + fileParentDirectory + separator + filename;

try {

fileOutputStream = new FileOutputStream(filePath, true);

printStream = new PrintStream(fileOutputStream);

System.setOut(printStream);

String str = null;

for (int i = 0; i < 10; i++) {

str = String.valueOf(Math.random());

System.out.println(str);

}

} catch (IOException e) {

e.printStackTrace();

}

}

}

* + - * 1. **Object流**

**注意事项**

Object流是一种处理流；

Object流用于存储和读取Object数据（此时强调的是一组数据，可参考程序实例）

与Object流相关的字段

Transient关键字（意为“短暂的”，Object中的变量如果用transient修饰，则不会被存入Object流中）

Serializable接口（如果要存储一个Object（多种数据集合）类型，则必须要继承Serializable接口或其子接口，标记该类可以序列化，告诉编译器用的，如果不写编译器认为该类不能序列化）

Externalizable接口（Serializable的子接口，控制具体的序列化过程）（暂时用不到，高级开发人员，一般默认让jdk写序列化过程，不建议自定义。）

**程序实例**

package practice.java.lcb.sxt;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

public class TestObjectIO {

public static void main(String[] args) {

ObjectOutputStream objectOutputStream = null;

ObjectInputStream objectInputStream = null;

FileOutputStream fileOutputStream = null;

FileInputStream fileInputStream = null;

String filename = "ObjectIO.txt";

String separator = File.separator;

String fileParentDirectory = "." + separator + "TxtFiles";

String filePath = fileParentDirectory + filename;

ObjectT objectT = new ObjectT();

try {

fileOutputStream = new FileOutputStream(filePath);

objectOutputStream = new ObjectOutputStream(fileOutputStream);

objectOutputStream.writeObject(objectT);

objectOutputStream.flush();

objectOutputStream.close();

fileInputStream = new FileInputStream(filePath);

objectInputStream = new ObjectInputStream(fileInputStream);

objectT = (ObjectT) objectInputStream.readObject();

System.out.println(objectT.a);

System.out.println(objectT.b);

System.out.println(objectT.c);

} catch (IOException e) {

e.printStackTrace();

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

class ObjectT implements Serializable {

/\*\*

\*

\*/

private static final long serialVersionUID = 1913650176329361426L;

int a = 9;

double b = 10;

float c = 9;

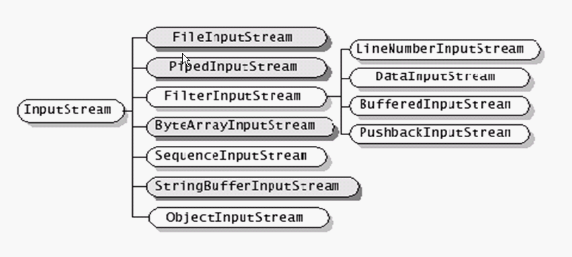
}

1. **J2SDK所提供的所有流类型都位于java.io包内，分别继承以下四种抽象流(都是抽象类)类型**
   1. **流的总结(常用IO流)**
      1. InputStream/OutputStream
      2. Reader/Writer
      3. FileInputStream/FileOutputStream
      4. FileReader/FileWriter
      5. BufferedInputStream/BufferedOutputStream
      6. BufferedReader/BufferedWriter
      7. ByteArrayInputStream/ByteArrayOutputStream
      8. InputStreamReader/OutputStreamWriter
      9. DataInputStream/DataOutputStream
      10. PrintStream/PrintWriter
      11. ObjectInputStream/OjectOutputStream

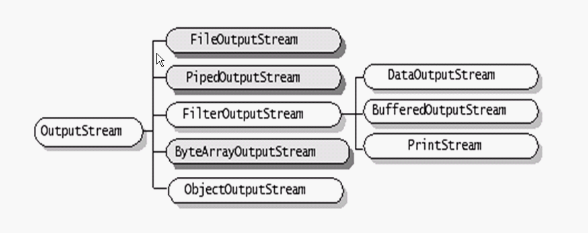


注意：凡是含“Stream”的都是字节流，”Reader/Writer”都是字符:

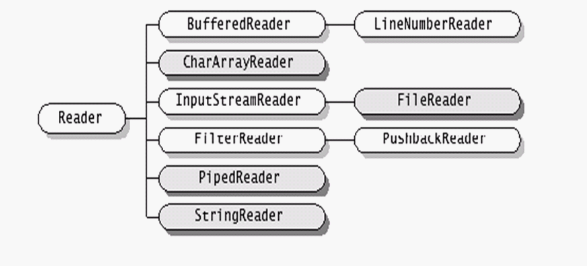
* 1. **InputStream**
     1. 继承自InputStream的流都是用于向程序中输入数据，且数据的单位为字节（8bit）；下图中深色的为字节流，浅色的为处理流



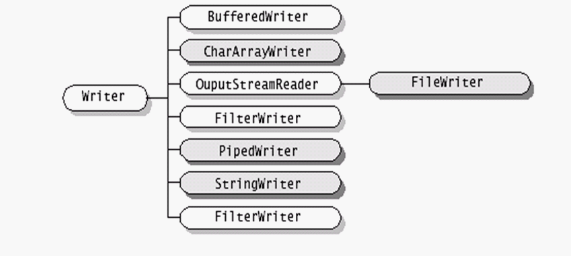
* + 1. **InputStream的基本方法**
       1. Int read() throws IOException
          1. 读取一个字节并以整数的形式返回（0-255）
          2. 如果返回-1则已经到了输入流的末尾
       2. Int read(byte[] buffer) throws IOException
          1. 读取一系列字节并存储到一个数组buffer中
          2. 返回实际读取的字节数目，如果读取前已经到了输入流的末尾则返回-1
       3. Int read(byte[] buffer, int offset, int length) throws IOException
          1. 读取length个字节(此时的length仅代表要读的字节长度，并不代表实际的情况，实际情况以返回值的形式返回)
          2. 存储一个字节数组buffer，从offset位置开始
          3. 返回实际读取的字节数目，如果读取前已经到了输入流的末尾则返回-1
       4. Void close() throws IOException
          1. 关闭（输入）流，释放内存资源
       5. Long skip(long n) throws IOException（不常使用）
          1. 跳过n个字节不读。返回实际跳过的字节数目
  1. **OutputStream**
     1. 继承自OutputStream的流是用于程序中输出数据（到硬盘），且数据的单位为字节（8bit）,下图中深色的为字节流，浅色的胃处理流



* + 1. **OutputStream的基本方法（基本与InputStream对称）**
       1. Void write(int b) throws IOException
          1. 向输出流中写入一个字节数据，该字节数据为参数b的低8位
       2. Void write(byte[] b) throws IOException
          1. 将一个字节类型的数组中的数据写入输出流
       3. Void write(byte[] b, int off, int len) throws IOException
          1. 将一个字节类型的数组中的数据从指定位置（off）开始的len个字节写入到输出流
       4. Void close() throws IOException
          1. 关闭（输出）流，释放内存资源
       5. Void flush() throws IOException
          1. 将输出流中缓冲的数据全部写入到目的地（因为输出流在写的时候是需要时间的，但是这个写的动作后面是紧跟着close()的，所以有可能还没有写完，输出流就已经关闭了，所以为了避免这种事情的发生，则使用flush()(在close()的前面)把缓冲区里面的数据全都写到目的地）
          2. Flush()写在close()前面是一种良好的编程习惯
  1. **Reader**
     1. 继承自Reader的流都是用于向程序中输入数据，且数据的单位都是字符（16bit，考虑到有的数据单位是16bit的，如：汉字）；下图中深色的为节点流，浅色的胃处理流



* + 1. **Reader的基本方法**
       1. Int read() throws IOException
          1. 读取一个字符并以整数的形式返回(0-255)
          2. 如果返回-1则已经到输入流的末尾
       2. Int read(char[] cbuf) throws IOExcepton
          1. 读取一系列字符并存储到一个数组cbuf中
          2. 返回实际读取的字符数
          3. 如果读取前已经到了输入流的末尾则返回-1
       3. Int read(char[] cbuf, int offset, int length) throw IOException
          1. 从offset开始读取length个字符，并存储到数组cbuf中
          2. 返回实际读取的字符数
          3. 如果读取前已经到了输入流的末尾则返回-1
       4. Void close() throws IOException
          1. 关闭输入流，释放内存资源
       5. Long skip(long n) throws IOException（不经常使用）
          1. 跳过n个字符不读，返回实际跳过的字节数
  1. **Writer**
     1. **继承自Writer的流都是用于程序中输出数据，且数据单位为字符（16bit）；下图中深色为节点流，浅色为处理流**



* + 1. **Writer的基本方法**
       1. Void write(int c) throws IOException
          1. 向输出流中写入一个字符数据，该字符数据为参数c的低16位
       2. Void write(char[] cbuf) throws IOException
          1. 将一个字符类型数组中的数据写入到输出流
       3. Void write(char[] cbuf, int offset, int length) throws

IOException

* + - * 1. 将一个字符类型的数组中的数据从指定位置开始写入length个字符数据
      1. Void write(String str) throws IOException
         1. 将一个字符串中的字符写入到输出流
      2. Void write(String str, int offset, int length) throws IOException
         1. 将一个字符串中的字符从off色调位置开始的length个字符写入到输出流
      3. Void close() throws IOException
         1. 关闭（输出）流，释放内存资源
      4. Void flush() throws IOException
         1. 将输出流中缓冲的数据全部写出到目的地

No.39 Java中的序列化（把Object类型转换成字节流）

1. 序列化 (Serialization)将对象的状态信息转换为可以存储或传输的形式的过程。因为网路传输是通过流（一个连续的字节队列）的形式。

No.40 Java中的线程

1. 线程
   1. 基本概念
      1. 线程是一个程序内部的程序执行的顺序控制流
      2. Main()方法是主线程
      3. 线程和进程的区别
      4. 进程就是“正在进行的程序”
      5. 进程包含了线程
      6. 每个进程都有独立的代码和数据空间（进程上下文），进程间的切换会有较大的开销；
      7. 线程可以看成是轻量级的进程，同一类线程共享代码和数据空间，每个线程有独立的运行栈和程序计数器（PC），线程切换开销小
      8. 多进程：在操作系统中能同时运行多个任务（程序）
      9. 多线程：在同一应用程序中有多个顺序流同时执行
   2. 注意事项
      1. Java的线程是通过java.lang.Thread类来实现的
      2. VM启动的时候会有一个主方法（public static void main(String[] args){}）所定义的线程
      3. 可以通过创建Thread的实例来创建新的线程
      4. 每个线程都是通过某个特定Thread对象所对应的run()方法来完成其操作的，run()方法称为线程体
      5. 通过调用Thread类的start()方法来启动一个线程
      6. 线程的优先级
      7. 如何正确的让线程停止（不建议使用stop()方法，因为这样太“暴力”会造成一些意想不到的结果，如文件读取一半等。使用一个变量flag，初始值为true，当要停止线程的时候设其值为false即可
         1. Java提供一个线程调度器来监控程序中启动后进入就绪状态的所有线程。线程调度器按照线程的优先级决定应该调度哪个线程来执行
         2. 线程的优先级用数字表示，范围从1到10，一个线程的缺省优先级是5
            1. Thread.Min\_PRIOTIRY = 1
            2. Thread.MAX\_PRIORITY = 10
            3. Thread.NORM\_PRIORITY = 5;
         3. 使用下述的线程方法获得或设置线程对象的优先级
            1. Int getPriority()
            2. Void setPriority(int newPriority)
   3. 控制线程的基本方法
      1. isAlive()
         1. 判断线程是否还“活”着，即线程是否还未终止
         2. 程序实例

package practice.java.lcb.sxt;

public class TestThread {

public static void main(String[] args) {

Thread thread = null;

Runnable threadForTestingIsAlive = null;

threadForTestingIsAlive = new ThreadForTestingIsAlive();

thread = new Thread(threadForTestingIsAlive);

thread.start();

}

}

class ThreadForTestingIsAlive implements Runnable {

public void run() {

System.out.println(Thread.currentThread().isAlive());

for (int i = 0; i < 100; i++) {

System.out.println("ThreadForTestingIsAlive" + i);

}

}

}

* + 1. getPriority()
       1. 获得线程的优先级数值
       2. 程序实例

package practice.java.lcb.sxt;

public class TestThread {

public static void main(String[] args) {

Thread thread1 = new Thread(new ThreadTestForPriority("ThreadOne"));

Thread thread2 = new Thread(new ThreadTestForPriority("ThreadTwo"));

thread1.start();

thread1.setPriority(Thread.MAX\_PRIORITY);

thread2.start();

}

}

class ThreadTestForPriority implements Runnable {

private String threadName;

public ThreadTestForPriority(String threadName) {

this.threadName = threadName;

}

public String getName() {

return threadName;

}

public void run() {

for (int i = 0; i < 100; i++) {

System.out.println(threadName + i);

}

}

}

* + 1. setPriority()
       1. 设置线程的优先级数值
       2. 程序实例（参考getPrioriry()的程序实例）
    2. Thread.sleep()
       1. 使线程“睡眠”一段时间，单位是毫秒
       2. 程序实例

package practice.java.lcb.sxt;

import java.util.Date;

public class TestThread {

public static void main(String[] args) {

MyThreadForTestingSleep myThreadForTestingSleep = new MyThreadForTestingSleep();

Thread thread = new Thread(myThreadForTestingSleep);

thread.start();

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

thread.interrupt();

myThreadForTestingSleep.setBoolean(false);

}

}

class MyThreadForTestingSleep implements Runnable {

boolean flag = true;

public void run() {

while (flag) {

System.out.println("====" + new Date() + "====");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

System.exit(-1);

}

}

}

public void setBoolean(boolean flag) {

this.flag = flag;

}

}

* + 1. Join()
       1. 调用某线程的该方法，将当前线程（就是join()方法所在的线程）和该线程“合并”，即等待该线程结束，再恢复当前线程的运行
       2. 程序实例

package practice.java.lcb.sxt;

public class TestThread {

public static void main(String[] args) {

MyThreadForTestingJoin myThreadForTestingJoin = new MyThreadForTestingJoin("James Bonde.");

myThreadForTestingJoin.start();

for (int i = 0; i < 10; i++) {

System.out.println("I am thread main.");

try {

Thread.sleep(100);

myThreadForTestingJoin.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

myThreadForTestingJoin.interrupt();

}

}

class MyThreadForTestingJoin extends Thread {

public MyThreadForTestingJoin(String str) {

super(str);

}

public void run() {

for (int i = 0; i < 10; i++) {

System.out.println("I am thread " + getName());

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

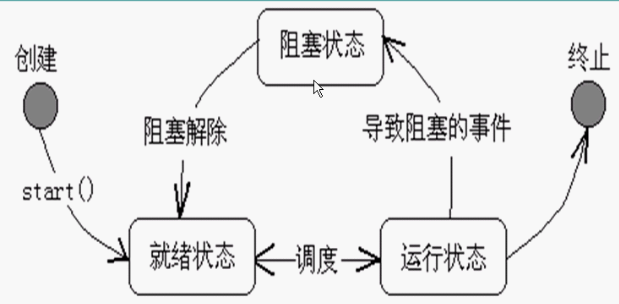
}

* + 1. Yield()
       1. 让出CPU，当前线程进入就绪队列，等待调度
       2. 程序实例

和Join()相仿

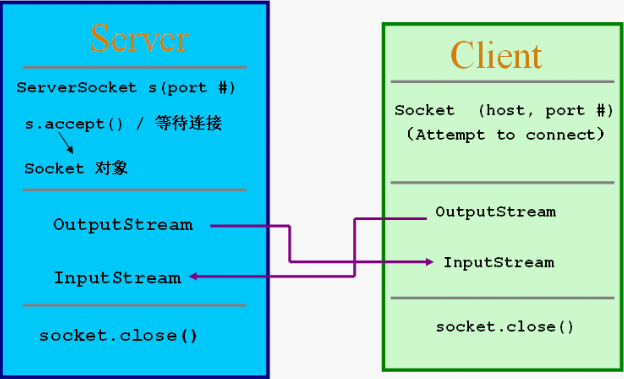
* + 1. Wait()
       1. 当前线程进入对象的wait pool
    2. Notify()/notifyAll()
       1. 唤醒对象的wait pool一个/所有等待线程

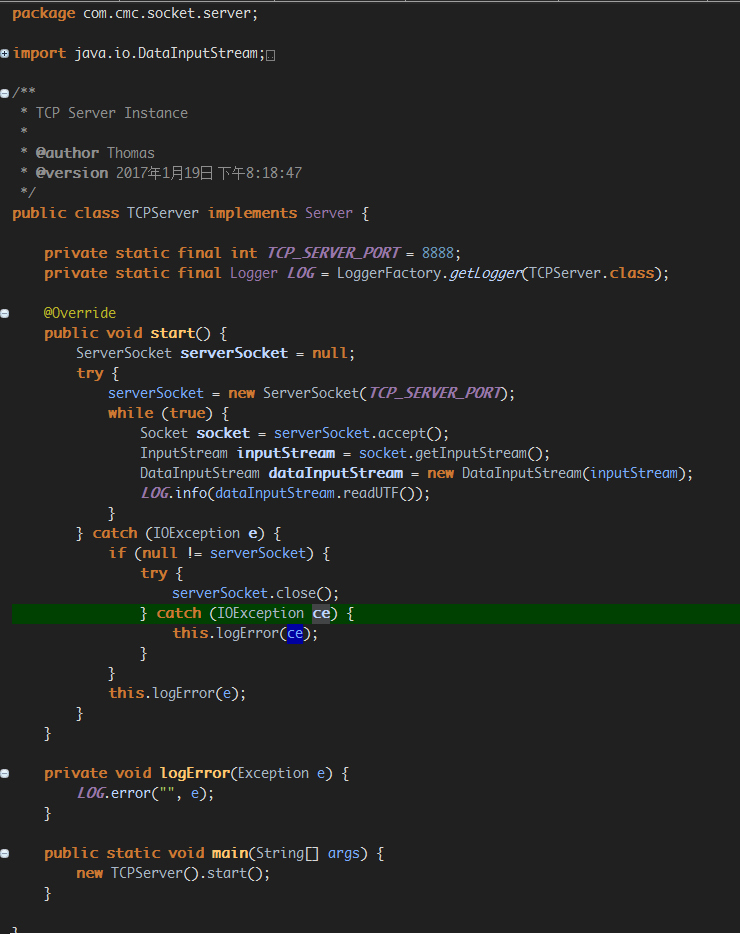
1. 线程的创建和启动
2. 线程的调度和优先级
3. 线程的状态控制
   1. 线程的状态转换



1. 线程同步
   1. Fda
   2. 程序实例
      1. 简单的线程同步程序(TestSync.java)
         1. package practice.java.lcb.sxt;
         2. public class TestSync implements Runnable {
         3. Timer timer = new Timer();
         4. public static void main(String[] args) {
         5. Runnable runner = new TestSync();
         6. Thread thread1 = new Thread(runner);
         7. Thread thread2 = new Thread(runner);
         8. thread1.setName("Thread1");
         9. thread2.setName("Thread2");
         10. thread1.start();
         11. thread2.start();
         12. }
         13. public void run() {
         14. timer.add(Thread.currentThread().getName());
         15. }
         16. }
         17. class Timer {
         18. private int nums;
         19. public void add(String name) {
         20. nums++;
         21. System.out.println(name + "已经被访问了" + nums + "次。");
         22. }
         23. }
      2. 线程的加锁(TestSync.java)
      3. 线程的死锁(TestDeadLock.java)
         1. package practice.java.lcb.sxt;
         2. public class TestDeadLock implements Runnable {
         3. private int flag;
         4. Object object1 = new Object();
         5. Object object2 = new Object();
         6. public static void main(String[] args) {
         7. TestDeadLock testDeadLock1 = new TestDeadLock();
         8. TestDeadLock testDeadLock2 = new TestDeadLock();
         9. testDeadLock1.flag = 1;
         10. testDeadLock2.flag = 0;
         11. Thread thread1 = new Thread(testDeadLock1);
         12. Thread thread2 = new Thread(testDeadLock2);
         13. thread1.start();
         14. thread2.start();
         15. }
         16. public void run() {
         17. if (flag == 1) {
         18. synchronized (object1) {
         19. try {
         20. Thread.sleep(100);
         21. } catch (InterruptedException e) {
         22. e.printStackTrace();
         23. }
         24. }
         25. synchronized (object2) {
         26. System.out.println(flag);
         27. }
         28. } else if (flag == 0) {
         29. synchronized (object2) {
         30. try {
         31. Thread.sleep(100);
         32. } catch (InterruptedException e) {
         33. e.printStackTrace();
         34. }
         35. }
         36. synchronized (object1) {
         37. System.out.println(flag);
         38. }
         39. }
         40. }
         41. }
2. 程序实例
   1. 通过实现Runnable接口创建线程
      * 1. /\*
        2. \* Simple case for Thread by implementing Runnable
        3. \*/
        4. package practice.java.lcb.sxt;
        5. public class TestThread {
        6. public static void main(String[] args) {
        7. MyThread myThread = new MyThread();
        8. Thread thread = new Thread(myThread);
        9. thread.start();
        10. for (int i = 0; i < 1000; i++) {
        11. System.out.println("main...");
        12. }
        13. }
        14. }
        15. class MyThread implements Runnable {
        16. public void run() {
        17. for (int i = 0; i < 1000; i++) {
        18. System.out.println("MyThread...");
        19. }
        20. }
        21. }
   2. 通过继承Thread类创建线程
      * 1. /\*
        2. \* Simple case for Thread by extending Thread
        3. \*/
        4. package practice.java.lcb.sxt;
        5. public class TestThread {
        6. public static void main(String[] args) {
        7. MyThread myThread = new MyThread();
        8. myThread.start();
        9. for (int i = 0; i < 1000; i++) {
        10. System.out.println("main..." + i);
        11. }
        12. }
        13. }
        14. class MyThread extends Thread {
        15. public void run() {
        16. for (int i = 0; i < 1000; i++) {
        17. System.out.println("MyThread..." + i);
        18. }
        19. }
        20. }
   3. 生产者消费者问题

**No.41 Java中的网络编程**

1. **本章内容**
   1. **网络基础**
   2. **TCP/IP协议**
   3. **IP地址**
   4. **Socket通信(TCP/UDP)（本质是编程接口对TCP/IP的封装）**
      1. **基本知识**
         1. 两个Java应用程序可以通过一个的网络通信连接实现数据交换，这个双向连接的一端称为一个Socket
         2. Socket通常用来实现Client-Server的连接
         3. Java.net中定义了两个类Socket和ServerSocket，分别用来实现双向连接的client和server端
         4. 建立连接所需要的寻址信息为远程计算机的IP地址和端口号(Port Number)
      2. **注意事项**
         1. TCP Socket通讯模型
         2. TCP和UDP端口是分开的，每个都有65536个
            1. 
      3. **程序实例**
         1. Socket通信（接受的时候不是很完美）
            1. **Server端**



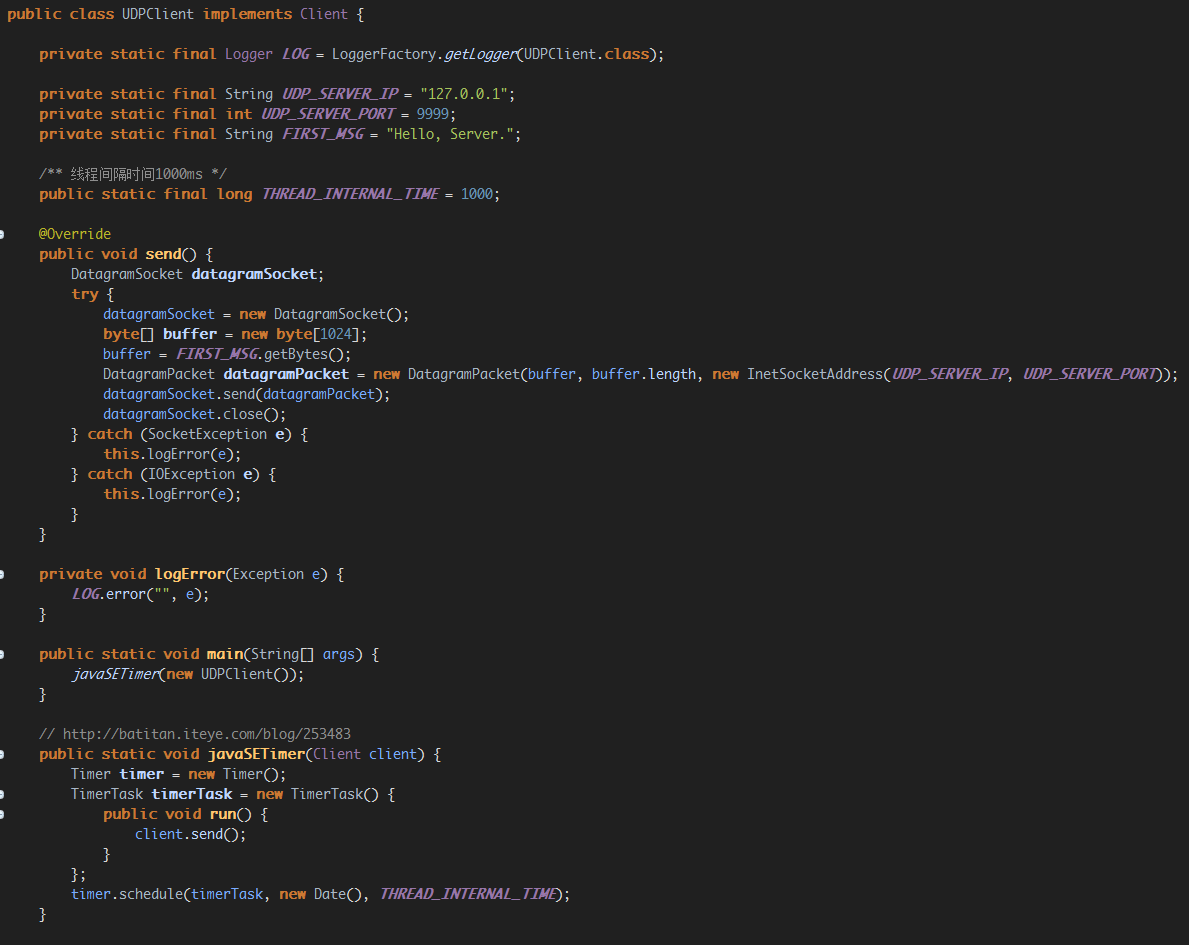
* + - * 1. **Client端**



* + - 1. **UDP通信**
         1. **Server端**



* + - * 1. **Client端**



**No.42 Java中的GUI**

1. 本章内容
   1. AWT（Abstract Window Toolkit）（并没有完全的跨平台，因为用到了系统本身的东西，是Java比较旧的图形开发包，较新的是javax.Swing开发包，javax是java extend的意思，java扩展开发包）
      1. 包括了很多类和接口，用于Java Application的GUI（Graphics User Interface 图形用户界面）编程
      2. GUI的各种元素（如：窗口、按钮、文本框等）由Java类来实现
      3. 使用AWT所涉及的类一般在java.awt包及其子包中
      4. Container和Component是AWT中的两个核心类
         1. 具体关系
            1. Java的图形用户界面的最基本组成部分是Component，Component类及其子类的对象用来描述以图形化的方式显示在屏幕上并能与用户进行交互的GUI元素，例如一个按钮、一个标签等
            2. 一般的Component对象不能独立地显示出来，必须“放在”某一个Container对象中才可以显示出来
            3. Container是Component的子类，Container子类对象可以“容纳”别的Component对象
            4. Container对象可以使用add(…)向其中添加其他Component对象
            5. Container是Component的子类，因此Container对象也可以被当做Component对象添加到其他Container对象中
            6. 有两种常用的Container

Window：其对象表示自由停泊的顶级窗口

Frame是Window的子类，由Frame或其子类创建的对象的一个窗体

Frame常用的构造方法：

Frame()

Frame(String s)创建标题栏为字符串s的窗口

Frame默认的布局方式是BorderLayout，特别注意的是如果没有指定其具体的布局方式，也就是说采用的是BorderLayout，则如果在其上面添加其他组件，无论组件有没有设置大小都会占满Frame因为BorderLayout默认把组件放在中间那一块，且没有其他块的时候中间那块会占满整个Frame，这就是为什么没有设置Frame的Layout的时候，其中的组件会占满整个Frame的原因。

Panel：其对象表示作为容纳啊Component对象，但不能独立存在，必须被添加到其它Container中（如Window或Applet）

Panel对象可以看成可以容纳Component的控件

Panel对象可以拥有自己的布局管理器，默认的是FlowLayout

Panel类拥有从其父类继承来的

SetBounds(int x,int y,int width,int height)

setSize(int width,int height)

setLocation(int x,inty)

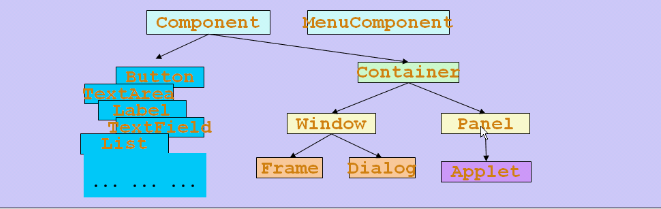
setBackground(Color color)

setLayout(LayoutManager layoutManager)等

Panel的构造方法

Panel() 使用默认的FlowLayout类布局管理器初始化

Panel(LayoutManager layoutManager)使用指定的布局管理器初始化

* + - 1. 关系图如下：
  1. 组件和容器
     1. TextField
        + 1. Java.awt.TextField类用来创建文本框对象
          2. 常用方法

构造方法

TextField()

TextField(int columns)

TextField(String text)

TextField(String text, int columns)

其他常用成员方法

Public void setText(String str)

Public String getText()

Public void setEchoChar(char c)

Public Boolean isEditable()

Public void setBackground(Color c)

Public void select(int selectionStart, int selectionEnd)

Public void selectAll()

Public void addActionListener(ActionListener actionListener)//添加动作监听器

* + - * 1. 程序实例
  1. 布局管理器
     1. 基本知识
        1. Java语言中，提供了布局管理器类的对象可以管理
           1. 挂历Component在Container中的布局，不必直接设置Component的位置和大小
           2. 每个Container都有一个布局管理器对象，当容器需要对某个组件进行定位或判断其大小尺寸的时候，就会调用其对应的布局管理器，调用Container的setLayout()方法改变其布局管理器对象
           3. 使用布局管理器的时候，布局管理器负责各个组件的大小和位置，因此用户无法在这种情况下设置组件大小和位置属性，如果试图使用java语言提供的setLocatin()、setSize()、setBounds()等方法，则都会被布局管理器覆盖
           4. 如果用户确实需要亲自设置组件大小或位置，则应该取消该容器的布局管理器，方法为setLayout(null)也就是说该容器不使用布局管理器。
           5. Frame是一个顶级窗口，Frame的缺省布局管理器是BorderLayout
           6. Panel无法单独显示，必须添加到某个容器中，Panel的缺省布局管理器是FlowLayout
           7. 当Panel作为一个组件添加到某个容器中后，该Panel仍然可以有自己的布局管理器
     2. 布局管理器的类型
        1. FlowLayout
           1. FlowLayout是Panel类的默认布局管理器

FlowLayout布局管理器对组件逐行定位，行内从左到右，一行排满后换行

不改变组件的大小，按照组件原有的尺寸显示组件，可设置不同的组件间距、行距以及对齐方式

* + - * 1. FlowLayout布局管理器默认的对齐方式是居中
        2. FlowLayout的构造方法

New FlowLayout(FlowLayout.RIGHT, 20, 40)

右对齐，组件之间水平间距20个像素，垂直间距40个像素

New FlowLayout(FlowLayout.LEFT)

左对齐，水平和垂直间距默认为的5

New FlowLayout()

使用默认的居中对齐方式，水平和垂直的间距默认为5

* + - * 1. 程序实例

package pratice.java.lcb.sxt.GUI;

import java.awt.Button;

import java.awt.FlowLayout;

import java.awt.Frame;

public class TestFlowLayout {

public static void main(String[] args) {

Frame frame = new Frame();

FlowLayout flowLayout = new FlowLayout(FlowLayout.RIGHT, 20, 40);

frame.setLayout(flowLayout);

Button button1 = new Button("Hello");

Button button2 = new Button("World");

Button button3 = new Button("I am coming.");

frame.add(button1);

frame.add(button2);

frame.add(button3);

frame.setBounds(100, 100, 200, 200);

frame.setVisible(true);

}

}

* + - 1. BorderLayout
         1. BorderLayout是Frame类的默认布局管理器
         2. BorderLayout将整个容器划分成：

EAST

WEST

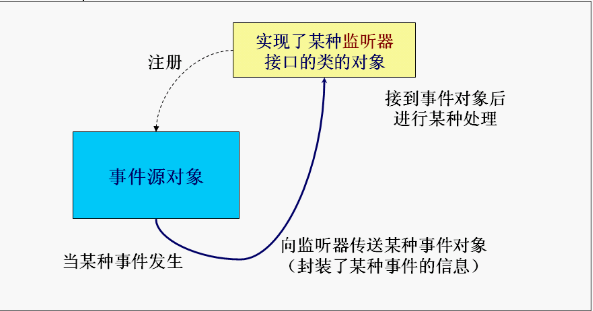
SOUTH

NORTH

CENTER

* + - * 1. 如果不指定组件加入的位置，则默认添加到CENTER区
        2. 每个区域只能加入一个组件，如果加入多个，则会覆盖先加入的那一个
        3. SetLayout(null)的意思不是设置默认的布局管理器而是设置为没有布局管理器的情况，此时组件只要按照相对于其上一层容器的左上角设置位置即可。
      1. GridLayout
         1. GridLayout布局管理器将组件划分为恶规则的矩形网络，每个单元格大小相等。组件被添加到每个单元格中，按照从左到右，从上到下的顺序依次添加组件
         2. 在GridLayout构造方法中指定行数和列数

GridLayout gridLayout = new GridLayout(3, 4);

* + - 1. CardLayout(自己看)
      2. GridBagLayout(自己看)
  1. Java图形
  2. Window事件
  3. 事件监听（不仅仅在AWT中使用，其他很多技术里面都有事件监听，框架也有）
     1. 图说
        1. 事件源对象：发生事件的对象，例如：Button、TextField等
        2. 事件：发生的事情，例如：单击按钮、双击按钮等，所以监听的接口有很多，根据事件的不同而不同
        3. 监听器：awt中组件里面都有自己的监听器，即已经写好了，只要自己写一个具体的类实现监听器接口（实现了某种监听器接口的类的对象）就行了
     2. 事件监听的类要实现ActionListener接口
     3. 内部类（暂时总结在此）
        1. 含义
           1. 写在其他类内部的类
           2. 优点

可以方便的访问包装类的成员

可以更清楚的组织逻辑，防止不应该被其它类访问的类进行访问，即通过这种方式控制类的访问控制权限(更安全)

* + - * 1. 使用时候

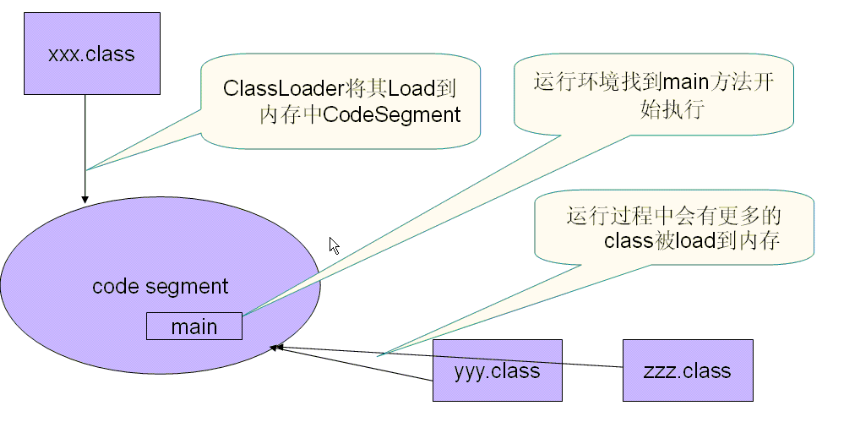
该类不允许或者不需要其它类进行访问

* + 1. 匿名内部类（局部类）
       1. 写在方法内部的类，该类的内容只在该类所在的方法内部使用
    2. Paint(Graphics c)方法
       1. 基本知识
          1. 该方法用于画图形（画图形专用）
          2. 该方法一般用在组件中（该结论没有经过验证，但是Frame中可以使用）
          3. 调用问题（当组件需要被重画的时候，比如第一次显示的时候、改变窗口大小的时候、当组件被遮盖重新出现的时候等）

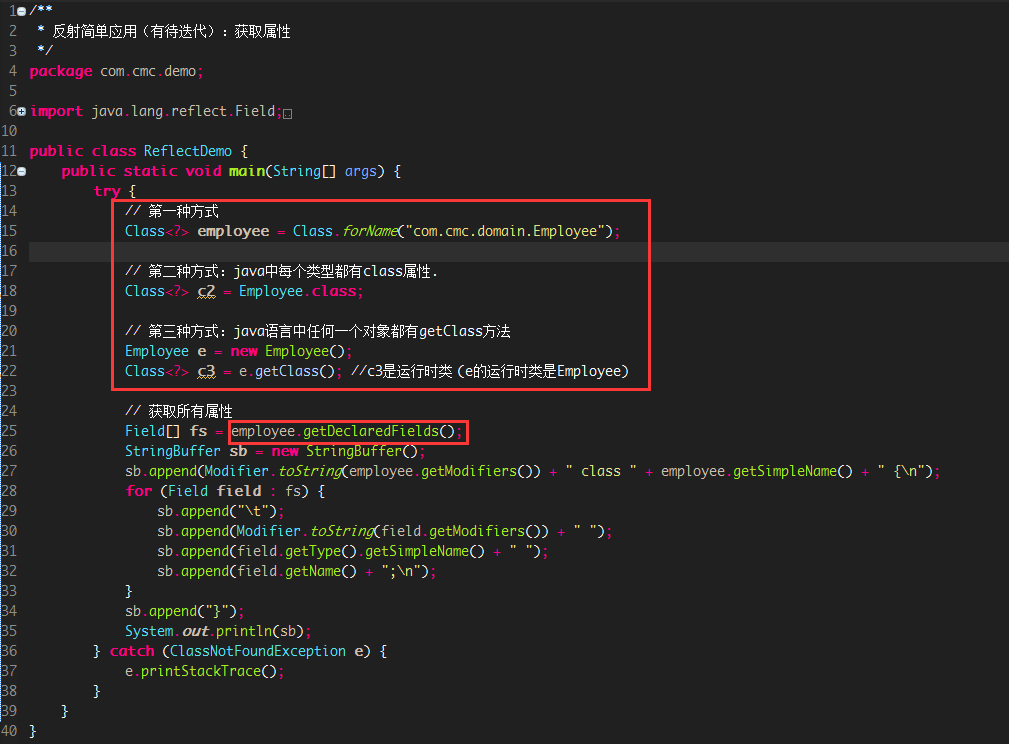
1. 程序实例
   1. 简单的Frame
      * 1. package pratice.java.lcb.sxt.GUI;
        2. import java.awt.Frame;
        3. import java.awt.Color;
        4. public class TestFrame {
        5. public static void main(String[] args) {
        6. Frame frame = new Frame("Frame name.");
        7. frame.setLocation(300, 300);x
        8. frame.setSize(200, 200);
        9. frame.setResizable(true);
        10. frame.setBackground(Color.red);
        11. frame.setVisible(true);
        12. }
        13. }
   2. 定义自己的Frame
      * 1. package pratice.java.lcb.sxt.GUI;
        2. import java.awt.Frame;
        3. import java.awt.Color;
        4. public class TestFrame {
        5. public static void main(String[] args) {
        6. MyFrame myFrame1 = new MyFrame(100, 100, 200, 200, Color.RED);
        7. MyFrame myFrame2 = new MyFrame(300, 100, 200, 200, Color.GREEN);
        8. MyFrame myFrame3 = new MyFrame(100, 300, 200, 200, Color.YELLOW);
        9. MyFrame myFrame4 = new MyFrame(300, 300, 200, 200, Color.MAGENTA);
        10. }
        11. }
        12. class MyFrame extends Frame {
        13. static int id;
        14. public MyFrame(int x, int y, int width, int height, Color color) {
        15. super("MyFrame" + id);
        16. setBounds(x, y, width, height);
        17. setBackground(color);
        18. setVisible(true);
        19. }
        20. }
   3. 简单的Panel实例
      * 1. package pratice.java.lcb.sxt.GUI;
        2. import java.awt.Color;
        3. import java.awt.Frame;
        4. import java.awt.Panel;
        5. public class TestFrame {
        6. public static void main(String[] args) {
        7. Frame frame = new Frame("Frame with Panel.");
        8. Panel panel = new Panel();
        9. frame.setLayout(null);
        10. panel.setLayout(null);
        11. panel.setBounds(50, 50, 50, 50);
        12. panel.setBackground(Color.YELLOW);
        13. frame.setBounds(100, 100, 200, 200);
        14. frame.setBackground(Color.MAGENTA);
        15. frame.add(panel);
        16. frame.setVisible(true);
        17. }
        18. }
   4. 自定义Panel
      * 1. package pratice.java.lcb.sxt.GUI;
        2. import java.awt.Color;
        3. import java.awt.Frame;
        4. import java.awt.Panel;
        5. public class TestFrame {
        6. public static void main(String[] args) {
        7. MyFrame myFrame = new MyFrame("Frame with Panel", 300, 300, 400, 300);
        8. }
        9. }
        10. class MyFrame extends Frame {
        11. private Panel panel1, panel2, panel3, panel4;
        12. public MyFrame(String str, int x, int y, int width, int height) {
        13. super(str);
        14. setLayout(null);
        15. panel1 = new Panel(null);
        16. panel2 = new Panel(null);
        17. panel3 = new Panel(null);
        18. panel4 = new Panel(null);
        19. panel1.setBounds(0, 0, width / 2, height / 2);
        20. panel2.setBounds(width / 2, 0, width / 2, height / 2);
        21. panel3.setBounds(0, height / 2, width / 2, height / 2);
        22. panel4.setBounds(width / 2, height / 2, width / 2, height / 2);
        23. panel1.setBackground(Color.RED);
        24. panel2.setBackground(Color.YELLOW);
        25. panel3.setBackground(Color.GREEN);
        26. panel4.setBackground(Color.BLUE);
        27. add(panel1);
        28. add(panel2);
        29. add(panel3);
        30. add(panel4);
        31. setBounds(x, y, width, height);
        32. setBackground(Color.MAGENTA);
        33. setVisible(true);
        34. }
        35. }

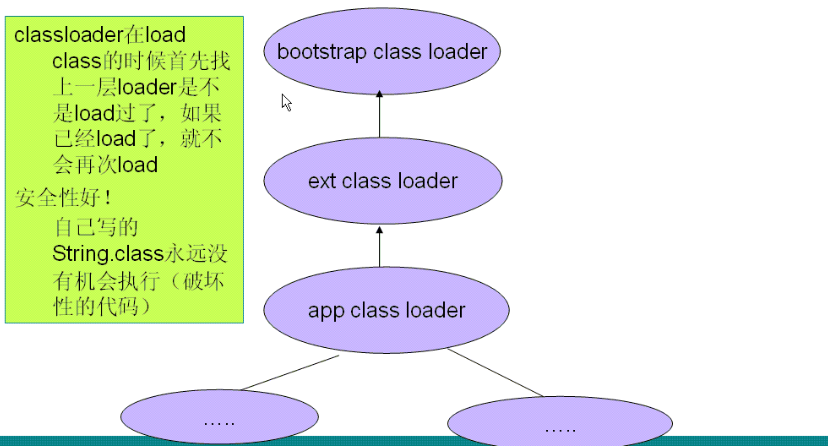
**No.43 Java中的反射机制（动态加载类，动态调用类的方法就是反射，在其他框架中经常用到）**

1. Java程序的运行过程



1. **ClassLoader的类的加载机制（动态加载机制）**
   1. 基本知识
      1. 并非一次加载
      2. 需要的时候加载（也就是说运行的时候动态加载）
      3. Static语句块在加载后执行一次
      4. Dynamic语句块每次new新的对象都会执行
         1. 等同于构造方法中的语句
         2. 用的较少
   2. 实例



1. JDK内置的ClassLoader（主要有四大类）
   1. Bootstrap class loader
      1. Implemented by native language(不是Java语言写的，最基本的classLoader)
      2. Load the core classes of jdk
   2. Extension class loader
      1. Load the class from jre/lib/ext
   3. Application class loader
      1. Load user-defined classes
      2. ClassLoader.getSystemClassLoader()
   4. Other class loader
      1. SecureClassLoader
      2. URLClassLoader
2. JDK中ClassLoader的层次结构
   1. 它们之间只是层次上的关系，并非继承关系
3. 具体使用说明
   1. 类.class
      1. 类名.class是Class对象的句柄，每个被加载的类，在JVM中都会有一个Class对象与之相对应，如果要创建新的对象，直接使用Class对象的局部class.forName就可以了，不需要用new类名。
      2. 在java中，每个class都有一个相应的Class对象，当编写好一个类，编译完成后，在生成的.class文件中，就产生一个class对象，用来表示这个类的类型信息。获得Class实例的三中方式：
      3. 1.利用对象调用getClass()方法获取该对象的Class实例
      4. 2ame()，用类的名字获取一个Class实例
      5. 3.运用.calss的方式获取Class实例，对基本数据类型的封装类，还可以采用.TYPE来获取对应的基本数据类型的Class实例
4. @reference <http://www.cnblogs.com/rollenholt/archive/2011/09/02/2163758.html>

**No.44 Java中的拆箱和装箱**

1. **拆箱：把引用数据类型转为对应的基本数据类型**
   1. 实例
      1. Integer –> int
   2. 说明
      1. 引用数据类型到基本数据类型，（区别）前者默认值为null后者为0
2. **装箱：把基本数据类型转换为对应的引用数据类型**
   1. 实例
      1. int -> Integer
   2. 说明
      1. 基本数据类型 ->引用数据类型，区别同上