Java SE课程详细DEMO

Day01//基础//

Demo01

计算机 = 硬件　＋ 软件

硬件：

cpu : cpu是计算机中的运算核心与控制核心。负责了解析计算机的指令、与计算机运算的执行。

内存: 数据与cpu交互的桥梁。 程序所需要的数据都会先加载到内存中，cpu是直接读取内存中数据运算的。

特点： 存取速度快。 缺点：断电或者是程序退出的时候，内存中的数据都会丢失。

硬盘（机械硬盘、固态硬盘）：可以永久性保存数据， 缺点： 存取的速度慢。 固态硬盘： 读取速度非常快。

软件：软件就是按照一定顺序组织起来的计算机指令与数据的集合。 软件 = 数据 + 指令

软件的类别：

系统软件：

比如： windows 、 linux 、 mac os 、 ios 、 android（改UI）

应用软件: 应用软件必须要在系统软件的基础上才能使用的。

比如： QQ、 微信、 迅雷、 飞信...。。。

软件的本质（软件为我们解决了什么问题）： 提高了人类与计算机的交互效率。

人类与计算机的交互方式：

1. 图形化界面的交互方式： 简单 、 功能强大。

2.通过dos命令的交互方式： 特点： 需要记忆大量的dos命令， 功能比较弱小。

首先具备一个控制台(命令行提示符窗口)用于输入dos命令：

打开一个控制台的方式：

方式一：开始-------> 所有程序--------->附件----------->命令提示符

方式二：开始------->运行-------->输入cmd即可.

常见的dos命令：

盘符： 进入指定的盘下面。

操作文件夹:

dir 列出当前控制台所在的路径下的所有文件以及文件夹。

cd 路径 进入指定的路径下。

cd .. 回退到上一级目录上。

cd \ 回退到当前路径下的根目录。

md 文件夹的名字 创建一个文件夹

rd 文件夹的名字 删除一个空文件夹。 注意： 不能用于删除非空到的文件夹。

操作文件：

echo 数据>文件名 创建一个文件并且指定写入的数据。

type 文件名 查看指定的文件内容。

del 文件名 删除指定的文件。 注意： del后面如果跟一个文件夹，那么是删除一个文件夹中的所有文件。

\* 通配符（可以匹配任意的文件名）

tab 内容补全.

cls 清屏

上下方向键 找回上一次所写过的命令。

exit 关闭控制台。

计算机语言(更好的与计算机进行交互):

计算机的发展历史：

机器语言：010101010101010

汇编语言 通过大量的英文单词代表计算机的指令。 add 1 1

高级计算机语言: C \ JAVA \ C++ \ PHP

1993 , JAVA初衷： 机 1994 年互联网刚刚兴起,.（高司令\ 高斯林），改造成了面向互联网的计算机语言。

java重要特性之 ------- 跨平台(一次编译，到处运行).

平台： 操作系统。

java的环境的搭建：

jre: java运行环境。 jre = java虚拟机 + 核心类库(辅助java虚拟机运行的文件)

jdk： java开发工具集 jdk = jre + java开发工具。

安装jdk的路径要注意： 不能带有中文路径或者是空格。

jdk目录的介绍：

bin: 存放的是java的开发工具。

db : JDK7附带的一个轻量级的数据库，名字叫做Derby。

include ：存放的调用系统资源的接口文件。

jre ： java运行环境

lib : 核心类库。

src.zip : java源代码

java.exe 启动java虚拟机解释并执行指定的class文件。

java工具的使用格式：

java 指定的class文件

javac.exe 启动java编译器对指定的java源文件进行编译、。

编写第一个java的程序：

第一步： 新建一个txt文件。把后缀名改成java。

第二步： java 代码全部都是写到类上面的。

如何编写一个类：

使用一个关键字class即可定义个类。

格式：

class 类名{

} 备注： 大括号表示是一个类的范围，在类的范围之内都可以写代码，

第三 步： 编写一个主方法。 主方法的写法是固定的，是一个程序的入口，代码从主方法开始执行，

如何启动java编译对java的源文件进行编译？

javac的使用格式：

javac java源文件。

编译的过程中我们就面临 的问题：每次编译 的时候都需要写上java源文件的完整路径。 烦！！！

解决方案： 如果可以在任意的路径下都可以找到java的开发工具，那么该问题就解决了。

在控制台写一个可执行文件的文件名的执行原理：

在控制台写一个可执行文件的文件名,那么系统首先会在控制台当前路径下去搜索是否存在该文件，如果可以找到指定的文件，

那么就执行该文件，如果找不到该文件，那么系统还会根据path的环境变量所保存的路径信息下去搜索是否有指定的文件，如果

能找到，那么就执行该文件。

配置path环境变量：

如何找到path的环境变量:

我的电脑-------属性------------>高级系统设置---------->高级--------->环境变量------------path

把javac所在的路径拷贝到path环境变量中即可。

Day04//函数//

Demo01

/\*

函数（方法）:

函数的作用： 提高功能代码的复用性。

需求：做两个数的加法功能。

目前存在的问题：以下的代码都是在做一个加法功能，而

这里加法功能 的代码目前是没有任何的复用性的。

解决方案： 如果一个功能的代码要被复用起来，那么这时候可以把

这里的功能代码封装起来，在java中把功能代码封装起来的是以函数的形式体现的。

函数的定义格式：

修饰符 返回值类型 函数名(形式参数..){

需要被封装的功能代码；

return 结果;

}

分析函数：

public static int add(){

int a =2;

int b =3;

return a+b;

}

修饰符: public static

返回值类型： int 返回值类型就是指函数运行完毕后，返回的结果的数据类型。

注意： 某些函数是没有结果返回给调用者的,那么这时候返回值类型是void。

函数名： add 函数名的作用：如果需要调用该函数就需要使用的函数名。 函数名只要符合标识符的命名规则即可。

函数名的命名规范： 首单词全部小写，其他单词的首字母大写，其他小写。

形式参数： 如果一个函数在运行的时候，存在着数据是要调用者确定 的，那么这时候就应该定义形式参数。

return ： 把一个结果返回给调用者。

函数的特点：

1. 函数定义好之后，是需要被调用才会执行的。 main函数是有jvm调用的，不需要我们手动调用。

\*/

class Demo1

{

public static void main(String[] args){

//int sum = add(2,3); //调用了add名字 的函数。

//System.out.println("结果："+ sum);

add(2,3);

}

/\*

//做加法功能的函数

public static int add(int a,int b){ // int a,int b 形式参数: 形式参数的值是交给调用者确定的。

return a+b;

}

需求： 定义一个函数做加法功能，不需要返回结果给调用者，直接输出结果即可。

\*/

public static void add(int a ,int b){

int sum = a+b;

System.out.println("结果："+ sum);

}

}

Demo02

/\*

函数的定义格式：

修饰符 返回值类型 函数名(形式参数..){

需要被封装的功能代码；

return 结果;

}

如何定义一个函数：

1. 返回值类型。

2. 是否存在未知的参数(是否存在要由调用者确定的参数)。---->形式参数

\*/

class Demo2

{

public static void main(String[] args)

{

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

//int max = getMax(14,5); //调用了函数 实际参数

//System.out.println("最大值："+ max);

getMax(3,7);

}

//需求2：定义一个函数比较两个int类型的数据大小， 不需要把最大值返回给调用者,直接打印即可。

public static void getMax(int a, int b){

int max = 0; //定义一个变量用于保存最大值的

if(a>b){

max = a;

}else{

max = b;

}

System.out.println("最大值："+ max);

}

/\*

//需求1： 定义一个函数比较两个int类型的数据大小， 把最大值返回给调用者。

public static int getMax(int a, int b){ // 形式参数

int max = 0; //定义一个变量用于保存最大值的

if(a>b){

max = a;

}else{

max = b;

}

return max; //把结果返回给调用者

}

\*/

}

Demo03

/\*

需求1： 定义一个函数判断一个分数的等级，把分数的等级返回给调用者。

"A等级" ”B等级“

如何 定义函数：

1. 返回值类型。 String

2. 未知的参数--->形式参数。 分数

需求2： 定义一个函数打印一个乘法表，不需要返回任何数据。

1. 返回值类型。 void

2. 未知的参数--->形式参数。 到底是什么乘法表

函数的特点：

1. 函数的作用就是把一个功能代码给封装起来，已达到提高功能代码的复用性。

2. 函数定义好之后是需要被调用才会执行的。

3. 如果一个函数没有返回值返回给调用者，那么返回值类型必须是使用void表示。

\*/

class Demo3

{

public static void main(String[] args) {

//String result = getGrade(189);

//System.out.println(result);

print(7);

}

//需求2： 定义一个函数打印一个乘法表，不需要返回任何数据。

public static void print(int row){

for(int i = 1 ; i<= row ; i++){

for (int j = 1 ;j<=i ;j++ ){

System.out.print(i+"\*"+j+"="+i\*j+"\t");

}

//换行

System.out.println();

}

}

//需求1： 定义一个函数判断一个分数的等级，把分数的等级返回给调用者。

public static String getGrade(int score){

String grade = ""; //定义一个变量存储等级

if(score>=90&&score<=100){

grade = "A等级";

}else if(score>=80&&score<=89){

grade = "B等级";

}else if(score>=70&&score<=79){

grade = "C等级";

}else if(score>=60&&score<=69){

grade = "D等级";

}else if(score>=0&&score<=59){

grade = "E等级";

}else{

grade = "补考等级";

}

return grade; //把等级返回给调用者

}

}

Day04//控制流程语句//

Demo01

/\*

控制流程语句之---if 判断语句

格式一： 只适用于一种情况下去使用。

if(判断条件){

符合条件执行的代码;

}

格式二：适用于两种情况下去使用

if(判断条件){

符合条件执行的代码

}else{

不符合条件执行 的 代码

}

]

格式3： 适用于多种情况使用的

if(判断条件1){

符合条件1执行的 语句;

}else if(判断条件2){

符合条件2执行 的语句;

}else if(判断条件3){

符合条件3执行 的语句;

}else if(判断条件4){

符合条件4执行 的语句;

}......else{

都不符合上述 条件执行的代码...

}

\*/

Demo02

/\*

控制流程语句之----switch选择判断语句

switch语句的格式：

switch(你的选择){

case 值1：

符合值1执行的代码

break;

case 值2：

符合值 2执行的代码

break;

case 值3：

符合值 3执行的代码

break;

case 值4：

符合值 4执行的代码

break;

......

default:

你的选择都符合上述的选项时执行的代码;

break;

}

switch语句要注意的事项：

1. switch语句使用的变量只能是byte、 char、 short、int、 String数据类型，String数据类型是从jdk7.0的时候开始支持的。

2. case后面跟 的数据必须是一个常量。

3. switch的停止条件：

switch语句一旦匹配上了其中的一个case语句，那么就会执行对应的case中的语句代码，执行完毕之后如果没有

遇到break关键字或者是结束switch语句的大括号，那么switch语句不会再判断，按照代码的顺序从上往下执行

所有的代码。直到遇到break或者是结束siwitch语句的大括号为止。

4. 在switch语句中不管代码的顺序如何，永远都是会先判断case语句，然后没有符合的情况下才会执行default语句。

if--else if---else if 语句与switch语句非常的相似：

switch语句的优点：switch语句的结构清晰。

switch缺点：如果 判断的条件是一个区间范围的，使用switch操作就非常的麻烦了。

判断以下那些不是计算机语言( D )

A java B C# C javascript D android

\*/

class Demo2{

public static void main(String[] args)

{

int option = 13; //定义一个变量存储你的选择

switch(option){

case 1:

System.out.println("java");

case 2:

System.out.println("C#");

case 3:

System.out.println("javascript");

case 4:

System.out.println("android");

default:

System.out.println("你的选择有误");

}

/\*

String str = "world";

switch(str){

case "hello":

System.out.println("hello");

break;

case "world":

System.out.println("world");

break;

}

\*/

}

}

/\*

需求： 接受键盘录入一个月份， 根据对应的月份输出对应的季节。

345 春天

678 夏天

9 10 11 秋天

1 2 12 冬天

要求使用switch语句实现。

\*/

import java.util.\*;

class Demo4

{

public static void main(String[] args)

{

System.out.println("请输入一个月份：");

//创建一个扫描器

Scanner scanner = new Scanner(System.in);

//调用扫描器的nextInt方法

int month = scanner.nextInt();

switch(month){

case 3:

case 4:

case 5:

System.out.println("春天");

break;

case 6:

case 7:

case 8:

System.out.println("夏天");

break;

case 9:

case 10:

case 11:

System.out.println("秋天");

break;

case 12:

case 1:

case 2:

System.out.println("冬天");

break;

default:

System.out.println("没有对应的季节");

break;

}

}

}

Demo03

/\*

循环语句----while循环语句

while循环 语句的格式:

while(循环的条件){

循环语句；

}

while循环语句要注意的事项：

1. while循环语句一般是通过一个变量控制其循环的次数。

2. while循环语句的循环体代码如果只有一个语句的时候，那么可以省略大括号。但是也是不建议大家省略。

3. while循环语句的判断条件后面不能跟有分号，否则会影响到执行的效果。

需求： 在控制上打印五句hello world.

\*/

class Demo5

{

public static void main(String[] args)

{

int count = 0;

while(count<5){

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

count++;

}

}

}

/\*

需求： 计算1+2+3+....+ 100的总和。

\*/

class Demo6{

public static void main(String[] args)

{

int num = 1;

int sum = 0; //定义一个变量用于保存每次相加的结果

while(num<=100){

sum = sum+num; // sum = 1

num++;

}

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

}

}

/\*

需求1:计算1-100,7的倍数总和。 7 14 21

如何产生一个随机数。

步骤：

1. 创建一个随机数对象。

2. 调用随机数对象的nextInt方法。

3. 导包。

\*/

class Demo7 {

public static void main(String[] args){

int num = 1;

int sum = 0; //定义一个变量用于保存每次相加的总和。

while(num<=100){ // num = 1

if(num%7==0){

sum = sum+num;

}

num++;

}

System.out.println("总和是："+ sum);

}

}

需求2： 实现猜数字游戏， 如果没有猜对可以继续输入你猜的数字，如果猜对了停止程序。

最多只能猜三次，如果还剩下最后一次机会的时候要提醒用户。

import java.util.\*;

class Demo8 {

public static void main(String[] args)

{

//创建一个随机数对象

Random random = new Random();

//调用随机数对象的nextInt方法产生一个随机数

int randomNum = random.nextInt(10)+1; //要求随机数是 1~10

//创建一个扫描器对象

Scanner scanner = new Scanner(System.in);

while(true){

System.out.println("请输入你要猜的数字:");

//调用扫描器的nextInt方法扫描一个数字

int guessNum = scanner.nextInt();

if (guessNum>randomNum){

System.out.println("猜大了..");

}else if(guessNum<randomNum){

System.out.println("猜小了..");

}else{

System.out.println("恭喜你，猜对了`..");

break;

}

}

}

}

控制流程语句----do while循环语句

格式：

do{

}while(判断条件);

需求： 在控制上打印五句hello world.

while循环语句与do-while循环语句的区别：

while循环语句是先判断后执行循环语句的，do-while循环语句

是先执行，后判断。不管条件是否满足至少会执行一次。

\*/

class Demo9{

public static void main(String[] args)

{

/\*

int count =0;

while(count<5){

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

count++;

}

boolean flag = false;

while(flag){

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

}

boolean flag = false;

do{

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

}while(flag);

\*/

int count = 0;

do

{

System.out.println("hello world");

count++;

}while(count<5);

}

}

/\*

需求： 使用do-while算出1-100之间偶数的总和。

\*/

class Demo10

{

public static void main(String[] args)

{

int num = 1;

int sum = 0; //定义一个变量用于保存每次相加的总和

do{

if(num%2==0){

sum += num;

}

num++;

}while(num<101);

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

}

}

Demo04

/\*

控制流程语句之---for循环语句

for循环语句的格式:

for(初始化语句;判断语句;循环后的语句){

循环语句;

}

for循环语句 要注意的事项：

1. for(;;)这种写法 是一个死循环语句，相当于while(true);

2. for循环语句的初始化语句只会执行一次，只是在第一次循环的时候执行而已。

3. for循环语句的循环体语句只有一句的时候，可以省略大括号不写。但是不建议省略。

需求： 在控制上打印五句hello world.

\*/

class Demo11

{

public static void main(String[] args)

{

/\*

int count=0;

while(count<5);{

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

count++;

}

int count = 0 ;

for(System.out.println("初始化语句A");count<5 ;System.out.println("循环后的语句C")){

System.out.println("循环体语句B");

count++;

}

\*/

for(int count = 0 ; count<5; count++){

}

{

System.out.println("hello world");

}

}

}

/\*

需求： 在控制台上打印一个 五行五列矩形/.

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

\*\*\*\*\*

先打印一行

\*/

class Demo12 {

public static void main(String[] args)

{

for(int j = 0 ; j<5 ; j++){ // 控制行数

for(int i = 0 ; i<5 ; i++){ // 控制列数

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

} // \*\*\*\*\*

//换行

System.out.println();

}

}

}

/\*

需求： 在控制台上打印一个正立的直角三角形 。

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

多行多列的图形。

行数 5行

列数： 会发生变化 的.

分析列数:

i = 0 ; i<5; j=0 ; j<=i 1个星号

i = 1 ; i<5 ;j=0 ; j<=1 2个星号

i = 2 ; i<5; j=0 ; j<=2 3个星号

.....

\*/

class Demo13

{

public static void main(String[] args)

{

for(int i = 0 ; i< 5 ; i++){

for(int j = 0 ; j<=i ; j++){ //控制列数

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

}

//换行

System.out.println();

}

}

}

/\*

需求： 打印一个倒立的直角三角形。

\*\*\*\*\*

\*\*\*\*

\*\*\*

\*\*

\*

5行

列数会发生变化

j<(5-i)

i= 0 ; i<5; j=0 ; j<5 ; 五个星号

i = 1; i<5; j=0 ; j<4; 四个星号

i = 2; i<5; j=0 ; j<3; 三个星号

\*/

class Demo14

{

public static void main(String[] args)

{

for(int i = 0 ; i<5; i++){

for (int j = 0 ; j<(5-i) ;j++ ){

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

}

//换行

System.out.println();

}

}

}

/\*

需求： 打印一个九九乘法表.

\*/

class Demo15

{

public static void main(String[] args)

{

for(int i = 1 ; i<=9 ; i++){

for(int j = 1 ; j<=i ; j++){ //控制列数

System.out.print(i+"\*"+j+"="+i\*j+"\t");

}

//换行

System.out.println();

}

}

}

Demo05

/\*

转义字符：特殊字符使用”\”把其转化成字符的本身输出，那么使用”\”的字符称作为转移字符。

需求： 在控制台上打印一个 hello" world

常见的转义字符有：

\b Backspace （退格键）

\t Tab 制表符(制表符的作用就是为了让一列对齐) 一个tab一般等于四个空格。

\n 换行

\r 回车 把光标移动到一行的首位置上。

注意： 如果是在windows系统上操作文件的时候需要换行，是需要\r\n一起使用的。

如果是在其他的操作系统上需要换行，仅需要\n即可。

\*/

import java.io.\*;

class Demo16 {

public static void main(String[] args) throws Exception

{

//System.out.println("Hello哈哈\rworld!");

File file = new File("F:\\a.txt");

FileWriter out = new FileWriter(file);

out.write("大家好\r\n");

out.write("你们好");

out.close();

}

}

Demo06

/\*

break、

break适用范围：只能用于switch或者是循环语句中。

break作用：

1. break用于switch语句的作用是结束一个switch语句。

2. break用于循环语句中的作用是结束当前所在的循环语句。

笔试题目：break目前位于内层的for循环，如何才能让break作用于外层 的for循环。

可以标记解决

标记的命名只要符合标识符的命名规则即可。

\*/

class Demo17

{

public static void main(String[] args)

{

aaa:for(int j = 0 ; j<3 ; j++){ // j=0 外层for循环

bbb:for(int i = 0 ; i< 2 ; i++){ // i=0 内层for循环

System.out.println("hello world"); // 1

break aaa;

}

}

}

}

Demo07

/\*

continue关键字

continue的适用范围： continue只能用于循环语句。

continue的作用：continue的作用是跳过本次的循环体内容。继续下一次。

continue要注意的事项：

1. 在一种情况下，continue后面不能跟有其他语句，因为是永远都无法执行到。

2. continue 也可以配合标记使用的。

\*/

class Demo18

{

public static void main(String[] args)

{

/\*

for(int i = 0 ; i<5 ; i++){ // i=1 2

if(i==1){

continue;

}

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

}

outer:for(int i = 0 ; i<3; i++){ // i= 0; i =1 i=2 3

inner:for(int j = 0 ; j<2 ; j++){ //j=0

System.out.println("hello"); //1 2 3

continue outer;

}

}

需求： 计算1-100的偶数总和.

\*/

int sum = 0 ;

for(int num = 1 ; num<=100 ; num++){

if(num%2!=0){

continue; //如果是奇数就跳过本次循环。

}

sum = sum+num;

}

System.out.println("总和："+ sum);

}

}

Day05//数组,排序算法,函数重载//

Demo01

/\*

函数：

函数的作用： 为了提高功能代码的复用性。

函数的定义格式:

修饰符 返回值类型 变量名(形式参数) {

函数体

}

函数的特点：

1. 函数是把一个功能的代码 封装起来以达到提高功能 代码的复用性。

2. 函数定义好之后，需要调用才能执行的。

3. 如果一个函数没有返回值类型，那么就使用void关键字表示。

需求: 定义一个函数判断分数的等级，然后返回分数对应的等级。

"a等级"

\*/

class Demo1

{

public static void main(String[] args)

{

String result = getGrade(90); //调用函数

System.out.println("对应的等级是："+ result );

}

public static String getGrade(int score){ //未知的参数定义在形参中， 由函数的调用者确定。

if(score>=90&&score<=100){

return "A等级";

}else if(score>=80&&score<=89){

return "B等级";

}else if(score>=70&&score<=79){

return "C等级";

}else if(score>=60&&score<=69){

return "D等级";

}else if(score>=0&&score<=59){

return "E等级";

}

}

/\*

public static String getGrade(int score){ //未知的参数定义在形参中， 由函数的调用者确定。

String grade = ""; //用于存储对应的等级

if(score>=90&&score<=100){

grade = "A等级";

}else if(score>=80&&score<=89){

grade = "B等级";

}else if(score>=70&&score<=79){

grade = "C等级";

}else if(score>=60&&score<=69){

grade = "D等级";

}else if(score>=0&&score<=59){

grade = "E等级";

}

return grade; // 把分数的等级返回给调用者。

}

\*/

}

Demo02

/\*

注意： 如果一个函数的返回值类型是具体的数据类型，那么该函数就必须要保证在任意情况下都保证有返回值。（除了返回值类型是void以外）

return 关键字的作用：

1. 返回数据给函数的调用者。

2. 函数一旦执行到了return关键字，那么该函数马上结束。 (能结束一个函数)

注意：一个函数的返回值类型 是void，那么也可以出现return关键字，但是return关键字的后面不能有数据。

break关键字与return关键字的区别：

1.break关键字是结束一个循环。

2. return关键字是结束一个函数。

\*/

class Demo2 {

public static void main(String[] args)

{

// String result = getGrade(10); //调用函数

// System.out.println("对应的等级是："+ result );

//add(0,2);

print();

}

public static void print(){

for(int i = 0 ; i < 5; i++){

System.out.println("hello world");

///break; //结束了当前的循环

return ; //结束当前的函数

}

System.out.println("哈哈我能执行吗??");

}

//目前该函数的返回值类型是void，那么是否可以存在return关键字呢？

public static void add(int a , int b){

if(a==0){

return; //结束一个函数

}

System.out.println("总和："+(a+b));

}

public static String getGrade(int score){ //未知的参数定义在形参中由函数的调用者确定。

if(score>=90&&score<=100){

return "A等级";

}else if(score>=80&&score<=89){

return "B等级";

}else if(score>=70&&score<=79){

return "C等级";

}else if(score>=60&&score<=69){

return "D等级";

}else if(score>=0&&score<=59){

return "E等级";

}else{

return "补考等级";

}

}

}

Demo03

/\*

需求： 定义一个函数做加法功能。

函数的重载：在一个类中出现两个或者两个以上的同名函数，这个称作为函数的重载。

函数重载的作用： 同一个函数名可以出现了不同的函数，以应对不同个数或者不同数据类型的参数。

函数重载的要求：

1. 函数名一致。

2. 形参列表不一致。（形式参数的个数或者是对应的数据类型不一致）

3. 与函数的返回值类型是无关的。

\*/

class Demo3

{

public static void main(String[] args)

{

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

//add1(1,2);

add(1,2.0);

}

// 这些函数都是在做加法运算。

public static double add(int a, int b){

System.out.println("两个参数的总和: "+ (a+b));

return 3.14;

}

//重复定义

public static int add(int a, double b){

System.out.println("double参数的总和: "+ (a+b));

return 12;

}

/\*

public static void add(int a , int b , int c){

System.out.println("三个参数的总和: "+ (a+b+c));

}

public static void add(int a , int b , int c,int d){

System.out.println("四个参数的总和: "+ (a+b+c+d));

}

\*/

}

Demo04

/\*

数组：数组是存储同一种数据类型数据的集合容器。

数组的定义格式：

数据类型[] 变量名 = new 数据类型[长度];

分析数组：

左边： int[] arr 声明了一个int类型的的数组变量，变量名为arr。

int : 表示该数组容器只能存储int类型的数据。

[] ： 这是一个数组类型。

arr ： 变量名.

右边：new int[50]; 创建了一个长度为50的int类型数组对象。

new ： 创建数组对象的关键字。

int: 表示该数组对象只能存储int类型数据。

[]： 表示是数组类型。

50 : 该数组最多能存储50个数据。数组的容量。

数组的好处： 对分配到数组对象中每一个数据都分配一个编号（索引值、角标、下标）,索引值的范围是从0开始，最大是： 长度-1.

局部变量： 如果一个变量是在一个方法（函数）的内部声明的，那么该变量就是一个局部变量。

成员变量： 成员变量就是定义在方法之外，类之内的.

\*/

class Demo4

{

public static void main(String[] args)

{

//定义一个数组

int[] arr = new int[4];

arr[0] = 10;

arr[1] = 30;

arr[2] = 50;

arr[3] = 90;

//System.out.println("数组的容量："+ arr.length); //数组的有一个length 的属性，可以查看数组 的容量。

//System.out.println("arr[2] = "+ arr[2]);

//查看数组中的所有数据。

for(int index = 0 ; index<arr.length ; index++){

System.out.println(arr[index]);

}

}

}

Demo05

/\*

数组的内存分析:

\*/

class Demo5

{

public static void main(String[] args)

{

/\*

int[] arr1 = new int[2];

int[] arr2 = new int[2];

arr1[1] = 10;

arr2[1] = 20;

System.out.println(arr1[1]); // 吴： 10 李：10

int[] arr1 = new int[2];

arr1[1] = 100;

int[] arr2 = arr1;

arr1[1] = 10;

arr2[1] = 20;

System.out.println(arr1[1]); // 20 20 20

\*/

}

}

Demo06

/\*

数组中最常见的问题：

1. NullPointerException 空指针异常

原因： 引用类型变量没有指向任何对象，而访问了对象的属性或者是调用了对象的方法。\

2. ArrayIndexOutOfBoundsException 索引值越界。

原因：访问了不存在的索引值。

\*/

class Demo6

{

public static void main(String[] args)

{

/\*

int[] arr = new int[2];

arr = null ; //null 让该变量不要引用任何的对象。 不要记录任何 的内存地址。

arr[1] = 10;

System.out.println(arr[1]);

\*/

int[] arr = new int[4];

arr[0] = 10;

arr[1] = 30;

arr[2] =40;

arr[3] = 50;

//System.out.println(arr[4]); //访问索引值为4的内存空间存储的值。

for(int index = 0 ; index<=arr.length ; index++){

System.out.print(arr[index]+",");

}

}

}

Demo07

/\*

数组的初始化方式：

动态初始化:

数据类型[] 变量名 = new 数据类型[长度];

静态初始化：

数据类型[] 变量名 = {元素1，元素2.....};

如果程序一开始你就已经确定了数据，那么这时候建议使用静态初始化。如果

数据一开始还不太明确，这时候就建议使用动态初始化。

\*/

class Demo7

{

public static void main(String[] args)

{

//动态初始化

//int[] arr = new int[10];

//静态初始化

int[] arr = {10,20,30,40,50};

for(int index = 0 ; index<arr.length ; index++){

System.out.print(arr[index]+",");

}

/\*

int[] arr = new int[50];

Scanner scanner = new Scanner(System.in);

for(int i = 0 ; i< arr.length ; i++){

arr[i] = scanner.nextInt();

}

\*/

}

}

Demo08

/\*

需求： 定义一个函数接收一个int类型的数组对象，找出数组对象中 的最大元素返回给调用者。

\*/

class Demo8

{

public static void main(String[] args)

{

int[] arr = {-12,-14,-5,-26,-4};

int max = getMax(arr);

System.out.println("最大值："+ max);

}

public static int getMax(int[] arr){

int max = arr[0]; //用于记录最大值

for(int i = 1 ; i < arr.length ; i++){

if(arr[i]>max){ //如果发现有元素比max大，那么max变量就记录该元素。

max = arr[i];

}

}

return max;

}

}

Demo09

/\*

需求： 定义一个函数接收一个int类型的数组对象, 把数组中的最大值放在数组中的第一位。

选择排序(直接排序)：使用一个元素与其他 的元素挨个比较一次，符合条件交换位置。

\*/

class Demo9{

public static void main(String[] args)

{

int[] arr = {12,5,17,8,9}; //对于5元素的数组，只需要找出4个最大值就可以排序了。

selectSort(arr);

}

public static void selectSort(int[] arr){

//把最大值放在首位置。

for(int j = 0; j<arr.length-1; j++){ // 控制的是轮数。

for(int i = j+1 ; i<arr.length ; i++){ // 找出最大值

if(arr[i]>arr[j]){

//交换位置

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

/\*

//把老二放在第二个位置

for(int i = 2 ; i< arr.length ; i++){

if(arr[i]>arr[1]){

int temp = arr[i];

arr[i] = arr[1];

arr[1] = temp;

}

}

//把老三放在第三个位置

for(int i = 3 ; i< arr.length ; i++){

if(arr[i]>arr[2]){

int temp = arr[i];

arr[i] = arr[2];

arr[2] = temp;

}

}

//把老四放在第四个位置

for(int i = 4 ; i< arr.length ; i++){

if(arr[i]>arr[3]){

int temp = arr[i];

arr[i] = arr[3];

arr[3] = temp;

}

}

\*/

//遍历数组，查看效果

System.out.print("目前的元素：");

for (int i = 0 ; i<arr.length ;i++){

System.out.print(arr[i]+",");

}

}

}

Demo10

/\*

需求： 定义一个函数接收一个int类型的数组对象, 把数组中的最大值放在数组中的最后一位。

冒泡排序：冒泡排序的思想就是使用相邻的两个 元素挨个比较一次，符合条件交换位置。

\*/

class Demo10

{

public static void main(String[] args)

{

int[] arr = {12,8,17,5,9}; // 最大的索引值: 4 容量：5

bubbleSort(arr);

}

public static void bubbleSort(int[] arr){

// 把最大值放在最后一个位置

for(int j = 0 ; j<arr.length-1 ; j++){ //控制轮数

for(int i = 0 ; i<arr.length-1-j ; i++){ // 找出一个最大值

//相邻的元素比较

if(arr[i]>arr[i+1]){

int temp = arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

}

/\*

//把老二放在倒数第二个位置上。

for(int i = 0 ; i <arr.length-1-1 ; i++){

if(arr[i]>arr[i+1]){

int temp = arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

//把老三放在倒数第三个位置上。

for(int i = 0 ; i <arr.length-1-2 ; i++){

if(arr[i]>arr[i+1]){

int temp = arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

//把老四放在倒数第四个位置上。

for(int i = 0 ; i <arr.length-1-3 ; i++){

if(arr[i]>arr[i+1]){

int temp = arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

\*/

//遍历数组，查看效果

System.out.print("目前的元素：");

for (int i = 0 ; i<arr.length ;i++){

System.out.print(arr[i]+",");

}

}

}

Demo11

/\*

需求：定义一个函数接收一个数组对象和一个要查找的目标元素，函数要返回该目标元素在

数组中的索引值，如果目标元素不存在数组中，那么返回-1表示。

折半查找法(二分法): 使用前提必需是有序的数组。

\*/

class Demo12

{

public static void main(String[] args)

{

int[] arr = {12,16,19,23,54};

//int index = searchEle(arr,23);

int index = halfSearch(arr,54);

System.out.println("元素所在的索引值是："+ index);

}

public static int halfSearch(int[] arr, int target){

//定义三个初始变量索引值，分别记录最大、最小、中间的查找范围索引值

int max = arr.length-1;

int min = 0;

int mid = (max+min)/2;

while(true){

if(target>arr[mid]){

min = mid+1;

}else if(target<arr[mid]){

max = mid -1;

}else{

//找到了元素

return mid;

}

//没有找到的情况

if (max<min){

return -1;

}

//重新计算中间索引值

mid = (min+max)/2;

}

}

/\*public static int searchEle(int[] arr, int target){

for(int i = 0 ; i<arr.length ; i++){

if(arr[i]==target){

return i;

}

}

return -1;

}

\*/

}

Demo12

/\*

需求： 定义 一个函数接收一个char类型的数组对象,然后翻转数组中的元素。

char[] arr = {'a','b','c','d','e'};

\*/

class Demo13

{

public static void main(String[] args)

{

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

char[] arr = {'a','b','c','d','e'};

reverse(arr);

}

public static void reverse(char[] arr){

for(int startIndex = 0 ,endIndex = arr.length-1 ; startIndex<endIndex ; startIndex++,endIndex--){

char temp = arr[startIndex];

arr[startIndex] = arr[endIndex];

arr[endIndex] = temp;

}

//遍历数组，查看效果

System.out.print("目前的元素：");

for (int i = 0 ; i<arr.length ;i++){

System.out.print(arr[i]+",");

}

}

}

Demo13

/\*

toString();

排序（sort）

找元素在数组中的位置(二分查找法) binarySearch

数组的工具类（Arrays）：

作业1： 自定实现一个Arrays的toString方法。

\*/

import java.util.\*;

class Demo14

{

public static void main(String[] args)

{

int[] arr = {12,3,1,10,8};

//排序的方法

Arrays.sort(arr);

String info = Arrays.toString(arr);

System.out.println("数组的元素："+ info);

// 1, 3, 8, 10, 12

int index = Arrays.binarySearch(arr,9);// 二分法查找 : 如果能在数组中找到对应 的元素，那么就返回该数据的索引值，如果没有找到那么就返回一个负数表示。

System.out.println("找到的索引值："+ index);

}

}

Demo14

/\*

二维数组： 二维数组就是数组中的数组。

一根 变量

一盒 20根 一维数组

一条 10包 20根 二维数组

二维数组 的定义格式：

数据类型[][] 变量名 = new 数据类型[长度1][长度2];

长度1：一条烟有多少盒。

长度2： 一盒烟有多少根。

二维数组 的初始化方式：

动态初始化:

数据类型[ ][ ] 变量名 = new 数据类型[长度1][长度2];

静态初始化：

数据类型[ ][ ] 变量名 = {{元素1,元素2...},{元素1,元素2...},{元素1,元素2...} ..}

\*/

class Demo15{

public static void main(String[] args)

{

//定义了一个二维数组

int[][] arr = new int[3][4];

arr[1][1] = 100;

/\*

System.out.println("二维数组的长度："+ arr.length); // 3

System.out.println("二维数组的长度："+ arr[1].length); //

\*/

System.out.println("数组的元素："+ arr[1][1]);

}

}

Demo15

/\*

静态初始化：

数据类型[][] 变量名 = {{元素1,元素2...},{元素1,元素2...},{元素1,元素2...} ..}

\*/

class Demo16

{

public static void main(String[] args)

{

int[][] arr = {{10,11,9},{67,12},{33,35,39,40}};

//遍历二维数组

for(int i = 0; i <arr.length ; i++){

for(int j = 0 ; j<arr[i].length ; j++){

System.out.print(arr[i][j]+",");

}

//换行

System.out.println();

}

}

// 定义一个二维数组，然后计算二维数组中偶数元素的总和。

}

Demo16

//打印二维数组

package Test02;

public class Log01 {

//地图的宽

public static final int WIDTH=30;

//地图的高

public static final int HEIGHT=10;

//地图

private char[][] background=new char[HEIGHT][WIDTH];

//初始化地图

public void initBackground(){

for(int rows=0;rows<background.length;rows++){

for(int cols=0;cols<background[rows].length;cols++){

if(rows==0||rows==(HEIGHT-1)||cols==0||cols==(WIDTH-1)){

background[rows][cols]='\*';

}else{

background[rows][cols]=' ';

}

}

}

//打印

for(int rows=0;rows<background.length;rows++){

for(int cols=0;cols<background[rows].length;cols++){

System.out.print(background[rows][cols]);

}

System.out.println();//换行

}

}

public static void main(String[] args) {

Log01 log01=new Log01();

log01.initBackground();

}

}

/\*

数组的特点：

1. 数组只能存储同一种 数据类型的数据。

2. 数组是会给存储到数组中 的元素分配一个索引值的，索引值从0开始，最大的索引值是length-1；

3. 数组一旦初始化，长度固定。

4. 数组中的元素与元素之间的内存地址是连续的。

\*/

class Demo17

{

public static void main(String[] args)

{

int[] arr = new int[3];

arr = new int[4];

System.out.println(arr.length);

}

}

Day06//面向对象,变量,封装//

Demo01

/\*

java面向对象的语言

对象：真实存在唯一的事物。

类： 实际就是对某种类型事物的共性属性与行为的抽取。 抽象的概念.... 车

我们从小的时候就不断的接触很多的对象，我们的大脑就会把

这些具备相同属性与行为的事物进行分类。

在现实生活中就存在对象与类这两种事物。计算机是服务于人类，为了让计算机更加接近我们的

生活，这时候就出现了面向对象的计算机语言。

面向过程（C） 面向对象（java）

现实生活中 : 对象---------类

在java中： 类------------>对象

面向对象的计算机语言核心思想： 找适合的对象做适合的事情。

如何找适合的对象：

1. sun已经定义好了很多的类，我们只需要认识这些类，我们就可以通过这些类创建对象使用。

2. 自定义类，通过自定义类来创建对象使用。

自定义类创建对象的三步骤：

1. 自定义类。

格式：

class 类名{

事物的公共属性使用成员变量描述。

事物的公共行为使用函数描述。

}

2. 通过自定义类创建对象。

格式：

类名 变量名 = new 类名();

3. 访问(设置)对象的属性或者调用对象的功能。

1.访问对象属性的格式：

对象.属性名.

2.设置对象的属性：

对象.属性名 = 数据;

3. 调用对象的功能

对象.函数名();

\*/

//车类

class Car{

//事物的公共属性使用成员变量描述。

String name; //名字的属性

String color; //颜色属性

int wheel; //轮子数

//事物的公共行为使用函数描述。

public void run(){

System.out.println(name+"飞快的跑起来啦...");

}

}

class Demo1

{

public static void main(String[] args)

{

/\*

//使用了Car类声明了一c变量， c变量指向了一个车对象。

Car c = new Car();

//设置车对象的属性值。

c.name = "BMW";

c.color = "白色";

c.wheel = 4;

//访问车对象的属性值

System.out.println("名字："+ c.name+" 颜色:"+ c.color+" 轮子数："+c.wheel);

c.run();

\*/

Car c1 = new Car();

c1.name = "宝马";

Car c2 = new Car();

c2.name = "大众";

c1 = c2;

System.out.println("名字："+ c1.name); //大众 大众 \大众

}

}

Demo02

/\*

需求： 使用java描述一个车与修车厂两个事物， 车具备的公共属性:轮子数、 名字、 颜色 ，还

具备跑的功能行为。跑之前要检测轮子是否少于了4个，如果少于了4个，那么要送到修车厂修理，

修车厂修理之后，车的轮子数要补回来4个。 然后车就继续的跑起来。

修车厂： 具备公共属性： 名字、 地址、 电话。

公共的行为： 修车。

初学者的经典错误：

1. 变量在同一个作用域(大括号)上是可以直接访问的。

2. 如果一个类要访问另外一个类变量时，那么这时候就只能通过创建对象进行访问。（仅对于目前正确）

\*/

//车类

class Car{

//事物的公共属性使用成员变量描述

String name ; // 名字

String color; //名字

int wheel; //轮子数

//事物的公共行为使用函数描述

public void run(){

if(wheel>=4){

System.out.println(name+wheel+"个轮子飞快跑起来..");

}else{

System.out.println(name+"不够4个轮子了，赶快去修理");

}

}

}

//修车厂

class CarFactory{

String name;//名字

String address ; //地址

String tel; //电话

//修车公共行为 ------ 返回值类型、 未知的参数

public void repair(Car c){

if(c.wheel>=4){

System.out.println("告诉你，费了很大力气修好了，给钱");

}else{

c.wheel = 4;

System.out.println("修好了，给钱!!");

}

}

}

class Demo2

{

public static void main(String[] args)

{

/\*

//0x98

Car c = new Car();

//给车对象赋予属性值

c.name = "陆丰";

c.color = "黑色";

c.wheel = 4;

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

c.run();

}

c.wheel = 3;

c.run();

//创建修车厂对象

CarFactory f = new CarFactory();

//给修车厂赋予属性值

f.name = "集群宝修车厂";

f.address = "韵泰商业广场一楼";

f.tel = "020-1234567";

//调用修车的修车

f.repair(c);//0x98

//

c.run();

\*/

//需求： 调用一次repair方法。 前提： 该对象调用完一个方法之后，这个对象就不再使用了。

}

}

Demo03

/\*

成员变量与局部变量的区别:

定义的位置上区别：

1. 成员变量是定义在方法之外，类之内的。

2. 局部变量是定义在方法之内。

作用上的区别：

1. 成员变量的作用是用于描述一类事物的公共 属性的。

2. 局部变量的作用就是提供一个变量给方法内部使用而已。

生命周期区别：

1. 随着对象 的创建而存在，随着对象的消失而消失。

2. 局部变量在调用了对应的方法时执行到了创建该变量的语句时存在，局部变量一旦出了自己的作用域

那么马上从内存中消失。

初始值的区别：

1. 成员变量是有默认的初始值。

数据类型 默认的初始值

int 0

float 0.0f

double 0.0

boolean false

char ' '

String（引用数据类型） null

2. 局部变量是没有默认的初始值的，必须要先初始化才能使用。

\*/

//人类

class Person {

String name; //成员变量

public void eat(){

int age ;

age = 12;

for(int i = 0 ; i< 5 ; i++){

System.out.println("hello world"+ age);

}

}

}

class Demo3

{

public static void main(String[] args)

{

Person p1 = new Person();

p1.eat();

}

}

Demo04

/\*

匿名对象：没有引用类型变量指向的对象称作为匿名对象。

需求： 使用 java类描述一个学生类。

匿名对象要注意的事项：

1. 我们一般不会给匿名对象赋予属性值，因为永远无法获取到。

2. 两个匿名对象永远都不可能是同一个对象。

匿名对象好处：简化书写。

匿名对象的应用场景：

1. 如果一个对象需要调用一个方法一次的时候，而调用完这个方法之后，该对象就不再使用了，这时候可以使用

匿名对象。

2. 可以作为实参调用一个函数。

\*/

//学生类

class Student{

int num; //学号

String name; //名字

public void study(){

System.out.println("好好学习，为将来称为高帅富做准备!");

}

}

class Demo4{

public static void main(String[] args)

{

//创建一个学生对象

//Student s = new Student();

//new Student().name = "狗娃"; //匿名对象

//System.out.println(Student().name); //null

System.out.println(new Student() == new Student()) ; // "==" 用于引用类型变量时，比较的是内存地址。判断两个 对象是否为同一个对象

需求： 调用Student的study方法。

Student s = new Student();

s.study();

new Student().study();

}

}

Demo05

/\*

面向对象三大特征：

1. 封装

2. 继承

3 多态

需求： 使用java类描述百合网的会员。

问题： 性别有问题？？

根本原因： 由于其他人可以直接操作sex属性。可以对sex属性进行了直接的赋值。

封装:

权限修饰符：权限修饰符就是控制变量可见范围的。

public : 公共的。 public修饰的成员变量或者方法任何人都可以直接访问。

private ： 私有的， private修饰的成员变量或者方法只能在本类中进行直接访问。

封装的步骤：

1. 使用private修饰需要被封装的属性。

2. 提供一个公共的方法设置或者获取该私有的成员属性。

命名规范：

set属性名();

get属性名();

疑问： 封装一定要提供get或者set方法吗？

不一定， 根据需求而定的。

规范 ： 在现实开发中一般实体类entity的所有成员属性（成员变量）都要封装起来。

实体类：实体类就是用于描述一类 事物的就称作为实体类。

工具类(Arrays数组的工具类):

封装的好处：

1. 提高数据的安全性。

2. 操作简单。

3. 隐藏了实现。

\*/

class Member{

public String name; //名字

private String sex; //性别

public int salary; //薪水

//定义一个公共的方法设置sex属性

public void setSex(String s){

if (s.equals("男")||s.equals("女")){ //注意： 如果比较两个字符串的内容是否一致，不要使用==比较， 使用equals方法。

sex = s;

}else{

//默认是男

sex = "男";

}

}

//定义一个公共的方法获取sex属性

public String getSex(){

return sex;

}

//聊天

public void talk(){

System.out.println(name+"聊得非常开心");

}

}

class Demo5{

public static void main(String[] args)

{

Member m = new Member();

m.name="狗娃";

m.setSex("女");

m.salary = 800;

System.out.println("姓名："+ m.name+" 性别："+ m.getSex()+" 薪水："+ m.salary);

}

}

Demo06

/\*

需求： 使用java类描述一个计算器类，计算器具备操作数1， 操作数2 、操作符三个公共 的属性

，还具备计算的功能行为。

要求： 不能直接对操作数1，操作数2，运算符这些属性进行直接 的赋值，要封装起来。 (+ - \* / )

根据需求提供get或者set方法。

需要提供set方法

\*/

//计算器类

class Calculator{

private int num1; //操作数1

private int num2; //操作数2

private char option ; //运算符

//提供公共的方法设置属性值....

public void initCalculator(int n1 , int n2 , char o){

num1 = n1;

num2 = n2;

if(o=='+'||o=='-'||o=='\*'||o=='/'){

option = o;

}else{

option = '+';

}

}

//计算的功能

public void calculate(){

switch(option){

case '+':

System.out.println("做加法运算,结果是："+(num1+num2));

break;

case '-':

System.out.println("做减法运算,结果是："+(num1-num2));

break;

case '\*':

System.out.println("做乘法运算,结果是："+(num1\*num2));

break;

case '/':

System.out.println("做除法运算,结果是："+(num1/num2));

break;

}

}

}

class Demo6

{

public static void main(String[] args)

{

//创建了一个计算器对象

Calculator c = new Calculator();

//设置属性值

c.initCalculator(1,2,'a');

//调用计算器的计算功能

c.calculate();

}

}

Demo07

/\*

需求： 目前存在数组：int[] arr = {0,0,12,1,0,4,6,0} ，编写一个函数

接收该数组，然后把该数组的0清空，然后返回一个不存在0元素的数组。

步骤：

1. 计算机新数组的长度。 原来的数组长度-0的个数

\*/

import java.util.\*;

class Demo7

{

public static void main(String[] args)

{

int[] arr = {0,0,12,1,0,4,6,0};

arr = clearZero(arr);

System.out.println("数组的元素："+Arrays.toString(arr));

}

public static int[] clearZero(int[] arr){

//统计0的个数

int count = 0; //定义一个变量记录0的个数

for(int i = 0 ; i<arr.length ; i++){

if(arr[i]==0){

count++;

}

}

//创建一个新的数组

int[] newArr = new int[arr.length-count];

int index =0 ; //新数组使用的索引值

//把非0的数据存储到新数组中。

for(int i = 0; i<arr.length ; i++){

if(arr[i]!=0){

newArr[index] = arr[i];

index++;

}

}

return newArr;

}

}

Day07//构造函数,构造代码块,this,ststic//

Demo (Calculator计算器)

public class Calculator{

private int num1; //操作数1

private int num2; //操作数2

private char option ; //运算符

//提供公共的方法设置属性值....

public void initCalculator(int n1 , int n2 , char o){

num1 = n1;

num2 = n2;

if(o=='+'||o=='-'||o=='\*'||o=='/'){

option = o;

}else{

option = '+';

}

}

//计算的功能

public void calculate(){

switch(option){

case '+':

System.out.println("做加法运算,结果是："+(num1+num2));

break;

case '-':

System.out.println("做减法运算,结果是："+(num1-num2));

break;

case '\*':

System.out.println("做乘法运算,结果是："+(num1\*num2));

break;

case '/':

System.out.println("做除法运算,结果是："+(num1/num2));

break;

}

}

}

Demo01

/\*

对象：真实存在的唯一的事物。

类： 同一种类型的事物公共属性与公共行为的抽取。

java面向对象语言：

核心思想： 找适合的对象做适合的事情.

找对象的方式：

方式一： sun已经定义好了很多了类，我们只需要认识这些类即可创建对象使用。

方式二： 我们需要自定义类，通过自定义类创建对象。

在整理笔记的过程中对自己有疑问的概念使用代码验证或者推翻它。

自定义类三步骤：

1. 自定义一个类。

class 类{

事物 的公共属性使用成员 变量描述

事物的公共行为使用函数描述。

}

2. 可以通过类创建对象。

3. 访问(设置)对象的属性或者调用对象的方法。

成员变量与局部变量 的区别：

1. 自定义的位置区别：

1. 成员变量是定义在方法之外，类之内的变量。

2. 局部变量是声明在方法之内的变量。

2. 作用上的区别：

1. 成员变量的作用描述一类事物的属性。

2. 局部变量的作用是提供一个变量给方法内部使用的。

3. 生命周期的区别；

1. 成员变量随着对象的创建而存在，随着对象的消失而消失。

2. 局部变量是调用到了对应的方法执行 到了创建该变量的语句时存在，一旦出了自己的作用域马上从内存中消失。

4. 初始值的区别：

1. 成员变量是有默认的初始值的。

2. 局部变量是没有默认的初始值的，必须要先初始化才能使用。

匿名对象： 没有引用类型变量指向的对象就称作为匿名对象。

匿名对象要注意的细节：

1. 一般不会给匿名对象的属性赋值，因为永远都没法获取到。

2. 两个匿名对象永远都不可能是同一个对象。

匿名对象好处：简化书写。

匿名对象的应用场景：

1. 如果调用一个对象的方法一次时，然后这个对象就不再使用，那么这时候可以使用匿名对象。

2. 可以作为实参调用一个函数。

封装：

封装的步骤：

1. 使用private修饰需要被封装的属性.

2. 根据需要提供get或者set方法设置以及获取属性。

封装的好处：

1. 提高了数据的安全性。

2. 操作简单。

3. 隐藏了实现。

注意：一般实体类的属性我们都会封装起来。

\*/

class Demo1

{

public static void main(String[] args)

{

int i = 10;

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

}

}

Demo02

/\*

java是 面向对象的语言： "万物皆对象": 任何事物都可以使用类进行描述。

需求： 使用java类描述 一个婴儿.

在现实生活中有两种婴儿， 一种婴儿一出生就具备了名字(白户)，还有一种婴儿就是 出生之后才有名字 的(黑户)。

构造函数：

构造函数的作用： 给对应的对象进行初始化。

构造函数的定义的格式：

修饰符 函数名(形式参数){

函数体...

}

构造函数要注意的细节：

1. 构造函数 是 没有 返回值类型的。

2. 构造函数的函数名必须要与 类名一致。

3. 构造函数并不是由我们手动调用的，而是在创建对应的对象时，jvm就会主动调用到对应的构造函数。

4. 如果一个类没有显式的写上一个构造方法时，那么java编译器会为该类添加一个无参的构造函数的。

5. 如果一个类已经显式的写上一个构造方法时,那么java编译器则 不会再为该类添加 一个无参 的构造方法。

6. 构造函数是 可以在一个类中以函数重载 的形式存在多个的。

疑问：创建对象时，jvm就会调用到对应的构造方法，那么我们以前没有学构造方法，那么

以前创建对象时，jvm是否 也会调用构造方法呢？如果有？构造方法从何而来呢？

会调用， java编译器在编译的 时候给加上去的。

jdk提供了一个java开发工具(javap.exe)给我们进行反编译的。

javap 反编译工具的使用格式：

javap -c -l -private 类名

疑问： java编译器添加 的无参构造方法的权限修饰符是 什么？

----->与类的权限修饰是一致的。

构造函数与普通函数的区别：

1. 返回值类型的区别：

1. 构造函数是没有返回值类型 的，

2. 普通函数是有返回值类型的，即使函数没有返回值，返回值类型也要写上void。

2. 函数名的区别：

1. 构造函数的函数名必须要与类名一致，

2. 普通函数的函数名只要符合标识符的命名规则即可。

3. 调用方式的区别：

1. 构造函数是 在创建对象的时候由jvm调用的。

2. 普通函数是由我们使用对象调用的，一个对象可以对象多次普通 的函数，

4. 作用上的区别：

1. 构造函数 的作用用于初始化一个对象。

2. 普通函数是用于描述一类事物的公共行为的。

\*/

//婴儿类

class Baby{

int id; //身份证

String name; //名字

//构造函数

public Baby(int i , String n){

id = i;

name = n;

System.out.println("baby的属性初始化完毕！！");

}

//无参 的构造函数

public Baby(){

System.out.println("无参的构造函数被调用了..");

}

//哭

public void cry(){

System.out.println(name+"哇哇哭...");

}

}

class Demo2

{

public static void main(String[] args)

{

//创建一个baby对象

Baby b1 = new Baby(110,"狗娃"); //婴儿诞生 白户

System.out.println("编号："+ b1.id +" 姓名："+ b1.name);

b1.cry();

b1.cry();

/\*

//黑户

Baby b2 = new Baby();

new Baby();

b2.id = 112;

b2.name = "狗剩";

System.out.println("编号："+ b2.id +" 姓名："+ b2.name);

\*/

}

}

Demo03

/\*

描述一个员工类，员工具备 的属性：id\ name \ age 。

具备的公共行为： 工作。

要求：一旦创建一个员工对象 的时候，那么该员工对象就要对应 的属性值。

\*/

class Employee{

int id; //编号

String name; //名字

int age; //年龄

//构造函数

public Employee(int a, String b , int c){

id =a;

name = b;

age = c;

}

public void work(){

System.out.println(name+"好好工作，努力挣钱!!");

}

}

class Demo3

{

public static void main(String[] args)

{

//创建一个员工对象

Employee e = new Employee(110,"狗娃",20);

System.out.println("编号："+ e.id+" 名字："+e.name +" 年龄："+ e.age);

}

}

Demo04

/\*

构造代码块：

构造代码块的作用：给对象进行统一的初始化。

构造函数的作用： 给对应的对象进行初始化。

构造代码块的格式：

{

构造代码块

}

注意： 构造代码块的大括号必须位于成员 位置上。

代码块的类别：

1. 构造代码块。

2. 局部代码块. 大括号位于方法之内。 作用：缩短局部 变量 的生命周期，节省一点点内存。

3. 静态代码块 static

\*/

class Baby{

int id; //身份证

String name; //名字

//构造代码块...

{

//System.out.println("构造代码块的代码执行了......");

}

//带参构造函数

public Baby(int i , String n){

id = i;

name = n;

}

//无参构造方法

public Baby(){

}

public void cry(){

System.out.println(name+"哇哇哭...");

}

}

class Demo4

{

public static void main(String[] args)

{

Baby b1 = new Baby(110,"狗娃"); // 狗娃 狗剩 铁蛋

System.out.println("编号："+ b1.id + " 名字："+b1.name);

/\*

System.out.println("编号："+ b1.id + " 名字："+b1.name);

new Baby(112,"狗剩");

new Baby();

\*/

/\*

new Baby(110,"狗娃");

new Baby(112,"狗剩");

new Baby();

\*/

}

}

Demo05

/\*

构造 代码块要注意的事项：

1. java编译器编译一个java源文件的时候，会把成员变量的声明语句提前至一个类的最前端。

2. 成员变量的初始化工作其实都在在构造函数中执行的。

3. 一旦经过java编译器编译后，那么构造代码块的代码块就会被移动构造函数中执行，是在构造函数之前执行的，构造函数的中代码是最后执行 的。

4. 成员变量的显示初始化与构造代码块 的代码是按照当前代码的顺序执行的。

\*/

class Demo5

{

//构造函数

public Demo5(){ //构造函数

i = 300000000;

}

//构造代码块 //构造代码块的初始化

{

i = 200000000;

}

int i = 100000000; //成员变量的显初始化

public static void main(String[] args)

{

Demo5 d = new Demo5();

System.out.println("i = "+d.i); // 双、吴、刘 ：2000 ：

}

}

Demo06

/\*

需求： 使用java类描述一个动物。

问题：存在同名的成员变量与局部变量时，在方法的内部访问的是局部变量(java 采取的是就近原则的机制访问的。)

this关键字：

this关键字代表了所属函数的调用者对象。

this关键字作用：

1. 如果存在同名成员变量与局部变量时，在方法内部默认是访问局部变量的数据，可以通过this关键字指定访问成员变量的数据。

2. 在一个构造函数中可以调用另外一个构造函数初始化对象。

this关键字调用其他的构造函数要注意的事项：

1. this关键字调用其他的构造函数时，this关键字必须要位于构造函数中 的第一个语句。

2. this关键字在构造函数中不能出现相互调用 的情况，因为是一个死循环。

this关键字要注意事项：

1. 存在同名的成员变量与局部变量时，在方法的内部访问的是局部变量(java 采取的是“就近原则”的机制访问的。)

2. 如果在一个方法中访问了一个变量，该变量只存在成员变量的情况下，那么java编译器会在该变量的 前面添加this关键字。

\*/

class Animal{

String name ; //成员变量

String color;

public Animal(String n , String c){

name = n;

color = c;

}

//this关键字代表了所属函数的调用者对象

public void eat(){

//System.out.println("this:"+ this);

String name = "老鼠"; //局部变量

System.out.println(name+"在吃..."); //需求： 就要目前的name是成员变量的name.

}

}

class Demo6

{

public static void main(String[] args)

{

Animal dog = new Animal("狗","白色"); //现在在内存中存在两份name数据。

Animal cat = new Animal("猫","黑色");

cat.eat();

}

}

Demo07

/\*

this关键字调用其他的构造函数要注意的事项：

1. this关键字调用其他的构造函数时，this关键字必须要位于构造函数中 的第一个语句。

2. this关键字在构造函数中不能出现相互调用 的情况，因为是一个死循环。

\*/

class Student{

int id; //身份证

String name; //名字

//目前情况：存在同名 的成员 变量与局部变量，在方法内部默认是使用局部变量的。

public Student(int id,String name){ //一个函数的形式参数也是属于局部变量。

this(name); //调用了本类的一个参数的构造方法

//this(); //调用了本类无参的 构造方法。

this.id = id; // this.id = id 局部变量的id给成员变量的id赋值

System.out.println("两个参数的构造方法被调用了...");

}

public Student(){

System.out.println("无参的构造方法被调用了...");

}

public Student(String name){

this.name = name;

System.out.println("一个参数的构造方法被调用了...");

}

}

class Demo7

{

public static void main(String[] args)

{

Student s = new Student(110,"铁蛋");

System.out.println("编号："+ s.id +" 名字：" + s.name);

/\*

Student s2 = new Student("金胖子");

System.out.println("名字：" + s2.name);

\*/

}

}

Demo08

/\*

需求： 使用java定义一个人类，人具备 id、name 、 age三个属性， 还具备一个比较年龄的方法。

要求： 必须 要 写上构造函数，构造函数也必须要使用上this关键字。

\*/

class Person{

int id; //编号

String name; //姓名

int age; //年龄

//构造函数

public Person(int id,String name ,int age){

this.id = id;

this.name = name;

this.age = age;

}

//比较年龄的方法

public void compareAge(Person p2){

if(this.age>p2.age){

System.out.println(this.name+"大!");

}else if(this.age<p2.age){

System.out.println(p2.name+"大!");

}else{

System.out.println("同龄");

}

}

}

class Demo8{

public static void main(String[] args)

{

Person p1 = new Person(110,"狗娃",17);

Person p2 = new Person(119,"铁蛋",9);

p1.compareAge(12);

}

}

Demo09

/\*

static(静态方法)

需求：描述一下传智学生类。 都是中国人....

目前存在的问题： 所有的学生都是中国的，有n个学生就会有n份中国的 数据存内存中，这样子会浪费内存。

目前方案： 把“中国”这个数据移动 到数据共享区中，共享这个数据给所有的Student对象使用即可。

问题2： 如何才能把这个数据移动 到数据共享区中共享呢？

解决方案： 只需要使用static修饰该数据即可。

静态的成员变量只会在数据共享区中维护一份，而非静态成员变量的数据会在每个对象中都维护一份的。

\*/

class Student{

String name;

//使用了static修饰country，那么这时候country就是一个共享的数据。

Static String country = "中国"; //国籍

//构造函数（初始化对象）

public Student(String name){

this.name = name;

}

}

class Demo9 {

public static void main(String[] args)

{

Student s1 = new Student("张三");

Student s2 = new Student("陈七");

s1.country = "小日本";

System.out.println("姓名："+s1.name+" 国籍："+ s1.country); // 中国

System.out.println("姓名："+s2.name+" 国籍："+ s2.country); // 小日本

}

}

Demo10

static(静态\修饰符)

1. static修饰成员变量 ：如果有数据需要被共享给所有对象使用时，那么就可以使用static修饰。

静态成员变量的访问方式：

方式1： 可以使用对象进行访问。

格式： 对象.变量名。

方式2： 可以使用类名进行访问。

格式： 类名.变量名;

注意：

1. 非静态的成员变量只能使用对象进行访问，不能使用类名进行访问。

2. 千万不要为了方便访问数据而使用static修饰成员变量，只有成员变量的数据是真正需要被共享的时候才使用static修饰。

static修饰成员变量的应用场景： 如果一个数据需要被所有对象共享使用的时候，这时候即可好实用static修饰。

2. static修饰成员函数:

class Student{

static String name;

static String country = "中国"; //静态的成员变量

public Student(String name){

this.name = name;

}

}

class Demo10 {

public static void main(String[] args)

{

Student s1 = new Student("狗娃");

Student s2 = new Student("狗剩");

System.out.println("国籍："+ Student.country);//中国

System.out.println("名字："+ s1.name);//狗剩--->相当于Student.name

System.out.println("名字："+ Student.name);//狗剩--->就近原则

}

}

Demo11

/\*

需求： 统计一个类被使用了多少次创建对象，该类对外显示被创建的次数。

\*/

class Emp{

static int count = 0; //计数器

String name;

//构造代码块

{

count++;

}

public Emp(String name){

this.name = name;

}

public Emp(){ //每创建一个对象的时候都会执行这里 的代码

}

public void showCount(){

System.out.println("创建了"+ count+"个对象");

}

}

class Demo11

{

public static void main(String[] args)

{

Emp e1 = new Emp();

Emp e2 = new Emp();

Emp e3 =new Emp();

e3.showCount();

}

}

Day08//静态函数,main,单例设计模式,继承,super,instanceof

Demo01

构造函数：

构造函数的作用： 初始化对应的对象。

构造函数的格式：

修饰符 函数名(形式参数){

函数体

}

构造函数要注意的事项：

1. 构造函数是没有返回值类型的。

2. 构造函数的函数名是必须与类名一致。

3. 构造函数不是由我们手动调用的，是我们在创建了对应的对象时，jvm会根据你创建的对象传递的参数调用对应 的对象。

4. 如果一个类没有显示写上一个构造函数的时候，那么java编译器会为该类添加一个无参 的构造函数。

5. 如果一个类已经显示写上一个构造函数的时候,那么java编译器则不会再为该类添加一个无参 的构造函数。

6. 构造函数是可以以函数重载的形式存在多个。

构造代码块：

构造代码块的作用：给对象进行统一的初始化。

构造代码块 的格式：

{

代码；

}

构造代码块是在创建了代码块所属类的对象时调用一次。

this关键字：

this关键字代表了所属函数的调用者对象。

this关键字的作用：

1. 一个类中存在同名的成员变量与局部变量时，在方法的内部默认是访问局部 变量的，可以通过this关键字指定访问成员变量 的数据。

2. this关键字还可以在构造函数中调用另外一个构造函数初始化对象，

this关键字调用构造函数要注意的事项：

1. this关键字调用其他 的构造函数时，必须要是在第一个语句。

2. this关键字调用构造函数时不能出现相互调用，因为是一个死循环。

3. 如果在方法的内部访问一个变量时，该变量只存在于成员变量中，没有局部变量，这时候java编译器会为该变量的前面添加this关键字。

static（静态、修饰符）

static修饰成员变量时：static修饰成员变量时,那么该成员变量的数据就是一个共享的数据.

静态成员变量的访问方式：

方式一： 使用对象进行访问。

对象.属性名

方式二：可以使用类名进行访问。

类名.属性名

注意：

1. 非静态成员变量不能类名直接访问，只能使用对象进行访问。

2. 千万不要为了方便访问成员变量而使用static修饰，一定要是该数据是共享数据 时才使用static修饰。

Demo02

/\*

static（静态、修饰符）

static修饰成员变量时：static修饰成员变量时,那么该成员变量的数据就是一个共享的数据.

静态成员变量的访问方式：

方式一： 使用对象进行访问。

对象.属性名

方式二：可以使用类名进行访问。

类名.属性名

注意：

1. 非静态成员变量不能类名直接访问，只能使用对象进行访问。

2. 千万不要为了方便访问成员变量而使用static修饰，一定要是该数据是共享数据 时才使用static修饰。

static修饰方法（静态的成员方法）:

访问方式：

方式一：可以使用对象进行访问。

对象.静态的函数名();

方式二：可以使用类名进行访问。

类名.静态函数名字。

推荐使用是类名直接访问静态的成员。

静态的成员变量与非静态的成员变量的区别：

1. 作用上的区别：

1. 静态的成员变量的作用共享一个 数据给所有的对象使用。

2. 非 静态的成员变量的作用是描述一类事物的公共属性。

2. 数量与存储位置上的区别：

1. 静态成员变量是存储方法 区内存中，而且只会存在一份数据。

2. 非静态的成员变量是存储在堆内存中，有n个对象就有n份数据。

3. 生命周期的区别：

1. 静态的成员变量数据是随着类的加载而存在，随着类文件的消失而消失。

2.非静态的成员数据是随着对象的创建而存在，随着 对象被垃圾回收器回收而消失。

静态函数要注意的事项：

1. 静态函数是可以使用 类名或者对象 进行调用的，而非静态函数只能使用 对象 进行调用。

2. 静态的函数可以直接访问静态的成员，但是不能直接访问非静态的成员。

原因：静态函数是可以使用类名直接调用的，这时候可能还没有存在对象，

而非静态的 成员数据是随着对象 的存在而存在的。

3. 非静态的函数是可以直接访问静态与非静态的成员。

原因：非静态函数只能由对象调用，当对象存在的时候，静态数据老早就已经存在了，而非静态

数据也随着对象的创建而存在了。

4. 静态函数不能出现this或者super关键字。

原因：因为静态的函数是可以使用类名调用的，一旦使用类名调用这时候不存在对象，而this

关键字是代表了一个函数 的调用者对象，这时候产生了冲突。

静态的数据的生命周期：静态的成员变量数据是优先于对象存在的。

static什么时候修饰一个函数？

如果一个函数没有直接访问到非静态的成员时，那么就可以使用static修饰了。 一般用于工具类型的方法

静态函数不能访问非静态的成员？

静态函数只要存在有对象，那么也可以访问非 静态的数据。只是不能直接访问而已。

\*/

class Student{

String name; //名字

static String country = "中国"; //国籍

//静态代码块 ：静态代码块是在Student.class文件加载到内存的时候就马上执行的。

static{

System.out.println("静态代码块执行了...");

}

//构造函数

public Student(String name){

this.name = name;

}

//非静态的成员函数

public void study(){

System.out.println("好好学习"+this);

}

//静态函数

public static void sleep(){ //静态方法与非静态方法的字节码文件是同时存在内存中 的。 只是静态的成员变量数据是优先于对象存在而已。

Student s= new Student("铁蛋");

System.out.println(s.name+"呼呼大睡...");

}

}

class Demo2

{

public static void main(String[] args)

{

Student.sleep();

// Student s = new Student("狗娃");

}

}

Demo03

/\*

需求：编写一个数组的工具类。

Arrays.toString（） [1,2,3,4];

sort()

\*/

//数组工具类

class ArrayTool{

public static String toString(int[] arr){

String result = "";

for(int i = 0; i < arr.length ; i++){

if (i==0){

result+="["+arr[i]+",";

}else if(i==(arr.length-1)){

result+= arr[i]+"]";

}else{

result+=arr[i]+",";

}

}

return result;

}

public static String toString(int [] arr){

String result="";

for(int i=0;i<arr.length;i++){

if(i==0){

result="["+arr[i]+","+result;

}else if(i==(arr.length-1)){

result=arr[i]+"]";

}else{

result=arr[i]+","+result;

}

}

}

public static void sort(int[] arr){

for(int i = 0; i < arr.length-1 ; i++){

for(int j = i+1 ; j<arr.length ; j++){

if(arr[i]>arr[j]){

int temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

}

}

}

}

class Demo3

{

public static void main(String[] args)

{

int[] arr = {12,1,456,165};

//ArrayTool tool = new ArrayTool();

ArrayTool.sort(arr);

String info = ArrayTool.toString(arr);

System.out.println("数组的元素："+ info);

}

}

Demo04

/\*

main函数的详解：

public ： 公共的。 权限是最大，在任何情况下都可以访问。 private

原因： 为了保证让jvm在任何情况下都可以访问到main方法。

static: 静态。静态可以让jvm调用main函数的时候更加的方便。不需要通过对象调用。

void: 没有返回值。 因为返回的数据是 给 jvm，而jvm使用这个数据是没有意义的。所以就不要了。

main: 函数名。 注意： main并不是关键字,只不过是jvm能识别的一个特殊的函数名而已。

arguments ：担心某些程序在启动需要参数。

\*/

class Demo4 {

public static void main(String[] args)

{

System.out.println("数组的长度："+ args.length);

for(int i = 0 ; i <args.length ; i++){

System.out.print(args[i]+",");

}

Scanner scanner = new Scanner(System.in);

}

}

Demo05

/\*

单例设计模式：保证一个类在内存中只有一个对象。

模式：模式就是解决 一类 问题的固定步骤 。

模式的概念最早起源于建筑行业....

建房子的步骤都是一样子：

打地基-----> 浇柱子------->盖楼面--------->砌墙--------->封顶---->装修-----入住

软件行业中23种设计模式：

单例设计模式

模板设计模式

装饰者设计模式

观察者设计模式

工厂设计模式

单例设计模式的步骤：

饿汉单例设计模式

1. 私有化构造函数。

2. 声明本类的引用类型变量，并且使用该变量指向本类对象。

3. 提供一个公共静态的方法获取本类的对象。

懒汉单例设计模式：

1. 私有化构造函数。

2. 声明本类的引用类型变量，但是不要创建对象，

3. 提供公共静态 的方法获取本类 的对象，获取之前先判断是否已经创建了本类 对象

，如果已经创建了，那么直接返回对象即可，如果还没有创建，那么先创建本类的对象，

然后再返回。

推荐使用： 饿汉单例设计模式。 因为懒汉单例设计模式会存在线程安全问题，目前还不能保证一类在内存中只有一个对象。

\*/

//饿汉单例设计模式 ----> 保证Single在在内存中只有一个对象。

class Single{

//声明本类的引用类型变量，并且使用该变量指向本类对象

private static Single s = new Single();

//私有化构造函数

private Single(){}

//提供一个公共静态的方法获取本类的对象

public static Single getInstance(){

return s;

}

}

class Test{

public static Test t=new Test();

private Test{}

public static Test getInstance(){

return t;

}

}

class Test{

public static Test t;

private Test{}

public static Test getInstance(){

if(t==null){

t=new Test();

}

return t;

}

}

//懒汉单例设计模式 ----> 保证Single在在内存中只有一个对象。

class Single2{

//声明本类的引用类型变量，不创建本类的对象

private static Single2 s;

//私有化了构造函数

private Single2(){}

//

public static Single2 getInstance(){

if(s==null){

s = new Single2();

}

return s;

}

}

class Demo5

{

public static void main(String[] args)

{

Single2 s1 = Single2.getInstance();//静态的，可以直接类名调用

Single2 s2 = Single2.getInstance();

System.out.println("是同一个对象吗？"+ (s1==s2));

}

}

Demo06

/\*

在现实生活中事物与事物之间是存在关系.

球员---->球队 整体与部分关系 has a 关系

学生----->人 继承的关系 is a 关系

\*/

//球员类

class Player{

int num; //编码

String name;

public Player(int num , String name){

this.num = num;

this.name = name;

}

public void run(){

System.out.println(name+"开跑...");

}

}

//球队类

class Team{

String name; //球队的名字

Player p1; //球员1

Player p2; //球员2

Player p3; //球员3

public Team(String name,Player p1,Player p2,Player p3){

this.name = name;

this.p1 = p1;

this.p2 = p2;

this.p3 = p3;

}

//开始比赛

public void startGame(){

System.out.println(name+"开赛啦！！");

}

}

class Demo6

{

public static void main(String[] args)

{

Player p1 = new Player(12,"梅西");

Player p2 = new Player(7,"C罗");

Player p3 = new Player(11,"内马尔");

//球队

Team t = new Team("恒大",p1,p2,p3);

t.startGame();

System.out.println("名字："+ t.p2.name);

}

}

Demo07

/\*

目前存在的问题：

1. 无法描述清楚这两个类之间的继承关系。

2. 存在着重复代码。

面向对象的三大特征：

1. 封装

2. 继承

3. 多态.

继承：继承是通过关键字extends体现的。

继承的格式：

class 类名1 extends 类名2{

}

继承要注意的事项：

1. 千万不要为了减少重复代码而去继承，只有真正存在着继承关系的时候才去继承。

2. 父类私有的成员变量不能被继承。

3. 父类的构造函数不能被继承。(普通函数可以继承)

4. 创建子类对象时默认会先调用父类无参的构造函数。

\*/

//人类

class Person{

String name;

private int age;

public Person(String name){

this.name = name;

}

public Person(){

System.out.println("Person类的构造方法被调用了....");

}

public void eat(){

System.out.println(name+"在吃饭...");

}

}

//学生类

class Student extends Person { // Student 就称作为Person类的子类， Person类就称作为Student的父类(超类、基类)

int num; //学号

public Student(){

System.out.println("Student类的构造方法被调用了....");

}

public void study(){

System.out.println(name+"good good study , day day up");

}

}

class Demo7

{

public static void main(String[] args)

{

Student s = new Student();

/\*

s.name = "狗娃";

System.out.println("名字："+ s.name);

s.eat();

\*/

}

}

Demo08

/\*

疑问： 为什么要调用父类的构造方法啊？这样子做的意义在那？

------>调用父类的构造方法是可以初始化从父类继承下去的属性的。

\*/

class Fu{

int x = 10;

String name;

public Fu(String name){

this.name = name;

System.out.println("Fu类d带参的构造方法...");

}

public Fu(){

System.out.println("Fu类无参的构造方法...");

}

}

class Zi extends Fu{

int x = 20;

public Zi(String name){

super(name); //指定调用父类一个参数的构造函数。就像原来的this(name);调用另一个构造函数

}

public void print(){

System.out.println("x1 = "+ x);

}

}

class Demo8

{

public static void main(String[] args)

{

Zi z = new Zi("大头儿子");

System.out.println("name= "+z.name);

}

}

Demo09

/\*

super关键字：

super关键字代表了父类空间的引用。

super关键字的 作用：

1. 子父类存在着同名的成员时，在子类中默认是访问子类的成员，可以通过super关键字指定访问父类的成员。

2. 创建子类对象时，默认会先调用父类无参的构造方法，可以通过super关键字指定调用父类的构造方法。

super关键字调用父类构造方法要注意的事项：

1. 如果在子类的构造方法上没有指定调用父类的构造方法，那么java编译器会在子类的构造方法上面加上super()语句。

2. super关键字调用父类的构造函数时，该语句必须要是子类构造函数中的第一个语句。

3. super与this关键字不能同时出现在同一个构造函数中调用其他的构造函数。因为两个语句都需要第一个语句。

super关键字与this关键字的区别：

1. 代表的事物不一致。

1. super关键字代表的是父类空间的引用。

2. this关键字代表的是所属函数的调用者对象。

2. 使用前提不一致。

1. super关键字必须要有继承关系才能使用。

2. this关键字不需要存在继承关系也可使用。

3. 调用构造函数的区别：

1. super关键字是调用父类的构造函数。

2. this关键字是调用本类的构造函数。

\*/

class Fu{

int x = 10;

String name;

public Fu(){

System.out.println("Fu类无参的构造方法..");

}

public Fu(String name){

this.name = name;

System.out.println("Fu类带参的构造方法..");

}

public void eat(){

System.out.println("小头爸爸吃番薯..");

}

}

class Zi extends Fu{

int x = 20;

int num;

public Zi(String name,int num){

super(name); //指定调用了父类带参的 构造方法...

this(); // 调用本类无参构造方法..

//super(); //指定调用了父类无参构造方法。。。

System.out.println("Zi类带参的构造方法..");

}

public Zi(){

System.out.println("Zi类无参的构造方法..");

}

public void print(){

System.out.println("x = " +super.x);

}

public void eat(){

System.out.println("大头儿子吃龙虾..");

}

}

class Demo9 {

public static void main(String[] args)

{

Zi z = new Zi("狗娃");

}

}

Demo10

/\*

目前的问题：父类的功能无法满足子类的需求。

方法重写overriding的前提： 必须要存在继承的关系。

方法的重写: 子父类出了同名的函数，这个我们就称作为方法的重写。

什么是时候要使用方法的重写：父类的功能无法满足子类的需求时。

方法重写要注意的事项：

1.方法重写时， 方法名与形参列表必须和父类一致。

2.方法重写时，子类的权限修饰符必须要大于或者等于父类的权限修饰符。

3.方法重写时，子类的返回值类型必须要小于或者 等于父类的返回值类型。

4.方法重写时， 子类抛出的异常类型要小于或者等于父类抛出的异常类型。

Exception(最坏)

RuntimeException(小坏)

方法的重载overloading：在一个类中 存在两个或者两个 以上的同名函数,称作为方法重载。

方法重载的要求（如：带参与无参的构造方法）

1. 函数名要一致。

2. 形参列表不一致（形参的个数或形参 的类型不一致）

3. 与返回值类型无关。

\*/

重载(Overloading)：一个类中有一个方法A，你又在这个类中创建了一个方法B，方法B的名字和A一样，返回值类型也一样，但是参数的类型或个数不同，此时B重载了A。

例如：

public class TestClass{

public int test(int i){return 1;}

public int test(float f){return 1;}

}

重写(Overriding)：一个类M继承另一个类N，N中有一个方法A，这时你在M写了一个方法B，方法B的名字、返回值以及参数都和A一样，此时B重写(又称覆盖)了A。

例如：

public class TestClass1{

public int test(int i){return 1;}

}

public class TestClass2 extends TestClass1{

public int test(int i){return 2;}

}

<<<<<<---------------------------------------------------------------------------------------->>>>>>>

class Animal{ //大的数据 类型

}

class Fish extends Animal{ //Fish小 的数据类型。

}

class Fu{

String name;

public Fu(String name){

this.name = name;

}

public Animal eat() throws RuntimeException {

System.out.println(name+"吃番薯...");

return new Animal();

}

}

class Zi extends Fu{

String num;

public Zi(String name){

super(name);//指定调用 父类带参的构造方法

}

//重写父类的eat方法

public Animal eat() throws Exception{

System.out.println("吃点开胃菜..");

System.out.println("喝点汤....");

System.out.println("吃点龙虾....");

System.out.println("吃青菜....");

System.out.println("喝两杯....");

System.out.println("吃点甜品....");

return new Animal();

}

}

class Demo10{

public static void main(String[] args)

{

Zi z = new Zi("大头儿子");

z.eat();

}

}

Demo11

/\*

需求：使用java描述一下普通的学生、 java基础班的学生、 就业班的学生。

所有的学生都会学习。但是学习的内容不一样。

普通 的学生： 马克思列宁主义。

基础班的学生：学习的是 javase。

就业班学生： javaee+android.

\*/

//普通的学生类

class Student{

String name;

//构造函数

public Student(String name){

this.name = name;

}

public void study(){

System.out.println(name+"学习马克思列宁主义");

}

}

//基础班的学生是属于学生中一种

class BaseStudent extends Student{

public BaseStudent(String name){

super(name);//指定调用父类构造函数

}

//重写

public void study(){

System.out.println(name+"学习javase..");

}

}

//就业班学生 也是属于普通学生中一种

class WorkStudent extends Student{

//构造 函数

public WorkStudent(String name){

super(name);

}

//重写

public void study(){

System.out.println(name+"学习javaee+android..");

}

}

class Demo11

{

public static void main(String[] args)

{

BaseStudent s = new BaseStudent("居东东");

s.study();

//创建一个就业班的学生

WorkStudent w = new WorkStudent("张三");

w.study();

}

}

Demo12

/\*

instanceof 关键字

instanceof关键字的作用：判断一个对象是否属于指定的类别。

instanceof关键字的使用前提：判断的对象与指定的类别必须要存在继承或者实现的关系。

instanceof关键字的使用格式：

对象 instanceof 类别

instanceof关键字的作用： 目前没用。但是后天我们学习 到了多态之后就非常有用。

一般我们做强制类型转换之前都会使用该关键字先判断一把，然后在进行转换的。

\*/

class Animal{

String name;

String color;

//构造函数

public Animal(String name, String color){

this.name = name;

this.color = color;

}

}

//狗是属于动物中一种

class Dog extends Animal {

public Dog(String name,String color){

super(name,color); //指定调用父类两个 参数的构造函数。

}

public void bite(){

System.out.println(name+"咬人!!");

}

}

//老鼠 也是属于动物中一种

class Mouse extends Animal{

public Mouse(String name,String color){

super(name,color);

}

public void dig(){

System.out.println(name+"打洞..");

}

}

class Demo12{

public static void main(String[] args)

{

Dog d = new Dog("哈士奇","白色");

System.out.println("狗是狗类吗？"+ (d instanceof Dog));

System.out.println("狗是动物类吗？"+ (d instanceof Animal));

//System.out.println("狗是老鼠类吗？"+ (d instanceof Mouse)); // false true

Animal a = new Animal("狗娃","黄色"); //狗娃是人

System.out.println("动物都是狗吗？"+ (a instanceof Dog));

}

}

Day09//final,抽象类,值交换,接口,多态//

Demo01

/\*

static（静态、修饰符）

static修饰成员变量: 如果一个成员变量使用static修饰，那么该成员变量的数据就是一个共享 的数据。

静态成员变量的访问方式

方式一：可以使用对象访问。

对象.属性名

方式二：可以使用类名访问。

类名.属性名

注意：

1. 非静态成员变量不能使用类名直接访问，要使用对象访问.

2. 千万不要为了访问方便而使用static修饰。一定要数据真正是需要被共享的时候才使用static修饰。

static修饰成员函数:

静态函数的访问方式

方式一：可以使用对象访问。

对象.属性名

方式二：可以使用类名访问。

类名.属性名

推荐使用类名访问静态的成员。

静态函数要注意的细节：

1. 非静态函数只能由对象调用，静态函数可以由类名或者对象进行调用。

2. 静态函数中不能直接访问非静态成员，可以直接访问静态的成员。

3. 非静态函数可以直接访问非静态成员以及可以直接访问静态的成员。

4. 静态函数中不能出现this或者super关键字。

单例设计模式：保证一个类在内存中只有一个对象。

饿汉单例设计模式：

1. 私有化构造函数。

2. 声明本类的引用类型变量并且指向本类的对象，(private static)

3. 提供一个公共静态的方法获取本类对象。

懒汉单例设计模式:

1. 私有化构造函数。

2. 声明本类的引用类型变量，但是不要创建对象。

3. 提供一个公共静态的方法获取本类对象，获取之前先判断是否已经创建了本类的对象，

如果没有创建，创建再返回。如果已经创建了，那么直接访问即可。

继承：

继承的格式:

class 类名1 extends 类名2{

}

继承要注意的事项：

1. 父类私有的成员不能被继承。

2. 父类的构造函数不能被继承。

3. 创建子类对象时，默认会先调用父类的无参构造函数，然后再调用子类 的构造函数。

super关键字：super关键字代表的是父类的引用空间。

super关键字的作用：

1. 如果子父类存在同名的成员时，在子类中默认是访问子类的成员，可以通过super关键字指定访问 父类的成员。

2. 创建子类对象时，默认会先调用父类的无参构造函数，可以通过super关键字指定调用父类的构造函数，

super关键字要注意的事项：

1. 如果在子类 的构造函数中没有指定调用具体父类构造函数，那么java编译器会在子类的构造函数上添加super()语句。

2. super关键字调用构造函数时必须出现构造函数中第一个语句。

3. this与super调用构造函数的时候不能同时出现在一个构造函数中，因为都需要是第一个语句。

方法重写：子父类存在着同名的函数。

方法重写的需求： 如果父类的功能无法满足子类的需求，那么就进行重写。

方法重写的要素：

1. 函数名与形参列表必须一致。

2. 子类的权限修饰符必须大于或者等于父类 的权限 修饰符。

3. 子父类的返回值类型必须小于或者等于父类的返回值类型。

4. 子类抛出的异常类型必须要小于或者等于父类抛出的异常类型。

instanceof：判断一个对象是否属于某种类型。

instanceof关键字的使用格式：

对象 instanceof 类。

instanceof的使用前提：判断的对象与类型必须存在继承或者实现的关系。

\*/

Demo02

/\*

final(最终、修饰符)

final关键字的用法：

1. final关键字修饰一个基本类型的变量时，该变量不能重新赋值，第一次的值为最终的。

2. fianl关键字修饰一个引用类型变量时，该变量不能重新指向新的对象。

3. final关键字修饰一个函数的时候，该函数不能被重写。

4. final关键字修饰一个类的时候，该类不能被继承。

常量 的修饰符一般为： public static final

\*/

//圆形

class Circle{

double r; //半径

public static final double pi = 3.14; //固定不变的

public Circle(double r){

this.r = r;

}

//计算面积

public final void getArea(){

System.out.println("圆形的面积是："+r\*r\*pi);

}

}

class Demo2 extends Circle

{

public Demo2(double r){

super(r);

}

public static void main(String[] args)

{

/\*

final Circle c = new Circle(4.0);

test(c);

\*/

Demo2 c = new Demo2(4.0);

c.getArea();

}

public static void test(Circle c){

c = new Circle(5.0); //c变量又重新指向了新的对象。

c.getArea();

}

}

Demo03

/\*

抽象类：

目前存在的问题：

1. 动物类的run方法描述的不正确。

2. 没有强制要子类一定要重写run方法。

抽象类的应用场景：

我们在描述一类事物的时候，发现该种事物确实存在着某种行为，

但是这种行为目前是不具体的，那么我们可以抽取这种行为 的声明，但是

不去实现该种行为，这时候这种行为我们称作为抽象的行为，我们就需要使用抽象类。

抽象类的好处: 强制要求子类一定要实现指定的方法。

抽象类要注意的细节：

1. 如果一个函数没有方法体，那么该函数必须要使用abstract修饰,把该函数修饰成抽象 的函数。

2. 如果一个类出现了抽象的函数，那么该类也必须 使用abstract修饰。

3. 如果一个非抽象类继承了抽象类，那么必须要把抽象类的所有抽象方法全部实现。

4. 抽象类可以存在非抽象方法，也可以存在抽象的方法.

5. 抽象类可以不存在抽象方法的。

5. 抽象类是不能创建对象的。

疑问：为什么抽象类不能创建对象呢？

因为抽象类是存在抽象方法的，如果能让抽象类创建对象的话，那么使用抽象的对象调用抽象方法是没有任何意义的。因为抽象的方法没有定义函数体...

6. 抽象类如果存在构造函数，其构造函数的作用：是提供给子类，创建对象的时候初始化父类的成员属性的。（所以子类会使用super（）来指定调用父类的一个构造函数）

\*/

//动物类--->抽象类

abstract class Animal{

String name;

String color;

public Animal(String name,String color){

this.name = name;

this.color = color;

}

//非抽象的方法

public void eat(){

System.out.println(name+"吃粮食");

}

//移动...

public abstract void run();//没有方法体---->抽象函数

}

//狗 是属于动物中一种

class Dog extends Animal{

public Dog(String name,String color){

super(name,color);

}

public void run(){

System.out.println(name+"四条腿跑得很快...");

}

}

//鱼 是属于动物中一种

class Fish extends Animal{

public Fish(String name,String color){

super(name,color);

}

public void run(){

System.out.println(name+"摇摇尾巴游啊游！");

}

}

class Demo3 {

public static void main(String[] args)

{

/\*

Dog d = new Dog("牧羊犬","棕色");

d.run();

//创建一个鱼对象

Fish f = new Fish("锦鲤","金黄色");

f.run();

\*/

Animal a = new Animal();

}

}

Demo04

/\*

需求： 描述一个图形、圆形、 矩形三个类。不管哪种图形都会具备计算面积

与周长的行为，但是每种图形计算的方式不一致而已。

常量的命名规范：全部字母大写，单词与单词 之间 使用下划线分隔。

abstract不能与以下关键字共同修饰一个方法：

1. abstract不能与private共同修饰一个方法。

2. abstract 不能与static共同修饰一个方法。

3. abstract 不能与final共同修饰一个方法。

\*/

//abstract 抽象

//图形类

abstract class MyShape{

String name;

public MyShape(String name){

this.name = name;

}

public abstract void getArea();//抽象的方法

public abstract void getLength();//抽象的方法

}

//圆形 是属于图形类的一种

class Circle extends MyShape{

double r;

public static final double PI = 3.14;

public Circle(String name,double r){

super(name);

this.r =r;

}

public void getArea(){

System.out.println(name+"的面积是："+PI\*r\*r);

}

public void getLength(){

System.out.println(name+"的周长是："+2\*PI\*r);

}

}

class Circle extends MyShape(){

double r;

public static final double PI=3.14;

public Circle(String name,double r){

super(name);

this.r=r;

}

public void getArea(){

System.out.println(name+"的面积是："+PI\*r\*r);

}

public void getLength(){

System.out.println(name+"的周长是："+2\*PI\*r);

}

}

//矩形 属于图形中的 一种

class Rect extends MyShape{

int width;

int height;

public Rect(String name,int width, int height){

super(name);

this.width = width;

this.height = height;

}

public void getArea(){

System.out.println(name+"的面积是："+width\*height);

}

public void getLength(){

System.out.println(name+"的周长是："+2\*(width+height));

}

}

class Demo4

{

public static void main(String[] args)

{

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

Circle c = new Circle("圆形",4.0);

c.getArea();

c.getLength();

//矩形

Rect r = new Rect("矩形",3,4);

r.getArea();

r.getLength();

}

}

Demo05

/\*

值传递: 调用一个方法的时候，传递给方法的参数 ，实际上传递变量所存储的值。

\*/

import java.util.\*;

class Person{

int x =10;

}

class Demo5

{

public static void main(String[] args)

{

/\*

int a = 3;

int b = 5;

changeValue(a,b);

System.out.println("交换之后的值：a = "+a +" b="+b);

int[] arr = {23,10,9};

changeArr(arr,1,2);

System.out.println("数的元素："+ Arrays.toString(arr));// 9 ,10

\*/

Person p = new Person();

changeObj(p,20);

System.out.println("x = "+ p.x);

}

public static void changeObj(Person p ,int x){

p.x = x;

}

//需求2： 定义一个函数交换数组中两个 元素的位置。

public static void changeArr(int[] arr , int index1, int index2){

int temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

//需求1：定义一个函数交换两个基本类型变量的值。

public static void changeValue(int a , int b){

int temp = a;

a = b;

b = temp;

System.out.println("方法内部交换的值：a = "+a +" b="+b);

}

}

Demo06

/\*

一个类最多只能有一个直接的父类。但是有多个间接的父类。

java是单继承。

\*/

class Ye{

String name;

}

class Fu extends Ye{

}

class Zi extends Fu{

}

class Demo6

{

public static void main(String[] args)

{

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

Zi zi =new Zi();

zi.name = "狗娃";

}

}

Demo07

/\*

接口：

接口：拓展功能的。 usb接口.。。

接口的定义格式：

interface 接口名{

}

接口要注意的事项 ：

1. 接口是一个特殊的类。

2. 接口的成员变量默认的修饰符为： public static final 。那么也就是说接口中的成员变量都是常量。

3. 接口中的方法都是抽象的方法，默认的修饰符为： public abstract。

4. 接口不能创建对象（因为是抽象的类）。

5. 接口是没有构造方法的。

6. 接口是给类去实现使用的，非抽象类实现一个接口的时候，必须要把接口中所有方法全部实现。

实现接口的格式：

class 类名 implements 接口名{

}

\*/

interface A{

//成员变量

public static final int i=10;

//成员函数--->都是抽象的方法

~~public abstract~~  void print();

}

class Demo7 implements A{ // Demo7就实现了A接口

public static void main(String[] args)

{

Demo7 d = new Demo7();

d.print();

}

//实现接口中的方法

public void print(){

System.out.println("这个是接口中的print方法...");

}

}

Demo08

/\*

接口的作用：

1. 程序的解耦。 （低耦合）

2. 定义约束规范。

3. 拓展功能。

\*/

//普通的铅笔类

class Pencil{

String name;

public Pencil(String name){

this.name = name;

}

public void write(){

System.out.println(name+"沙沙的写...");

}

}

//橡皮接口

interface Eraser{

public void remove();

}

//带橡皮的铅笔

class PencilWithEraser extends Pencil implements Eraser {

public PencilWithEraser(String name){

super(name);

}

public void remove(){

System.out.println("涂改,涂改....");

}

}

class Demo8

{

public static void main(String[] args)

{

PencilWithEraser p = new PencilWithEraser("2B铅笔");

p.write();

p.remove();

}

}

Demo09

/\*

需求：在现实生活中有部分同学在学校期间只会学习，但是有部分学生除了学习外还会赚钱。

interface （定义接口）

implements (实现接口)

分析：

普通的学生类

接口

会挣钱的学生

\*/

//普通的学生类

class Student{

String name;

public Student(String name){

this.name = name;

}

public void study(){

System.out.println(name+"好好学习");

}

}

//挣钱是学生拓展的功能---定义在接口上

interface Money{

public void makeMoney();

}

//会挣钱的学生

class MoneyStudent extends Student implements Money{

public MoneyStudent(String name){

super(name);

}

public void makeMoney(){

System.out.println(name+"好好挣钱，然后交学费！");

}

}

class Demo9

{

public static void main(String[] args)

{

Student s = new Student("李金华");

s.study();

//s.makeMoney();//普通学生无法调用赚钱

MoneyStudent m = new MoneyStudent("孙双双");

m.study();

m.makeMoney();

}

}

Demo10

/\*

类与接口（interface）之间关系： 实现关系。

类与接口要注意的事项：

1. 非抽象类实现一个接口时，必须要把接口中所有方法全部实现。

2. 抽象类实现一个接口时，可以实现也可以不实现接口中的方法。

3. 一个类可以实现多个接口 。

疑问： java为什么不支持多继承，而支持了多实现---->单继承多实现

class A{

public void print(){

System.out.println("AAAAAA");

}

}

class B{

public void print(){

System.out.println("BBBBBB");

}

}

class C extends A ,B{

}

new C().print();

接口与接口之间关系： 继承关系。

接口与接口之间要注意事项：

1. 一个接口是可以继承多个接口的。

\*/

interface A{

public void print();

}

interface C{

public void getArea();

}

interface B extends A,C{ // B接口继承A接口

public void test();

}

class Demo10 implements B{

public static void main(String[] args)

{

Demo10 d = new Demo10();

d.print();

}

public void getArea(){}

public void test(){}

public void print(){

System.out.println("这个是A接口的print方法...");

}

}

Demo11

/\*

面向对象的三大特征：

1. 封装

2. 继承。

3. 多态

多态：一个对象具备多种形态。(父类的引用类型变量指向了子类的对象)

或者是接口 的引用类型变量指向了接口实现类的对象)

多态的前提：必须存在继承或者实现的关系。

动物 a = new 狗();

多态要注意的细节：

1. 多态情况下，子父类存在同名的成员变量时，访问的是父类的成员变量。

2. 多态情况下，子父类存在同名的非静态的成员函数时，访问的是子类的成员函数。

3. 多态情况下，子父类存在同名的静态的成员函数时，访问的是父类的成员函数。

4. 多态情况下，不能访问子类特有的成员。

总结：多态情况下，子父类存在同名的成员时，访问的是父类的成员，只有在同名非静态函数时才是访问子类的。

编译看左边，运行不一定看右边。

编译看左边：java编译器在编译的时候，会检查引用类型变量所属的类是否具备指定的成员，如果不具备马上编译报错。

\*/

//动物类

abstract class Animal{

String name;

String color = "动物色";

public Animal(String name){

this.name = name;

}

public abstract void run();

public static void eat(){

System.out.println("动物在吃..");

}

}

//老鼠

class Mouse extends Animal{

String color = "黑色";

public Mouse(String name){

super(name);

}

public void run(){

System.out.println(name+"四条腿慢慢的走!");

}

public static void eat(){

System.out.println("老鼠在偷吃..");

}

//老鼠特有方法---打洞

public void dig(){

System.out.println("老鼠在打洞..");

}

}

class Fish extends Animal {

public Fish(String name){

super(name);

}

public void run(){

System.out.println(name+"摇摇尾巴游..");

}

}

class Demo11

{

public static void main(String[] args)

{

/\*

Mouse m = new Mouse("老鼠");

System.out.println(m.color);

//多态： 父类的引用类型变量指向子类的对象

\*/

Animal a = new Mouse("老鼠");

a.dig();

//a.eat();---->eat是同名的static函数----->父类：动物在吃

}

}

Demo12

/\*

多态的应用：

1. 多态用于形参类型的时候，可以接收更多类型的数据 。

2. 多态用于返回值类型的时候，可以返回更多类型的数据。

多态的好处： 提高了代码的拓展性。

需求1： 定义一个函数可以接收任意类型的图形对象，并且打印图形面积与周长。

\*/

//图形类

abstract class MyShape{

public abstract void getArea();

public abstract void getLength();

}

class Circle extends MyShape{

public static final double PI = 3.14;

double r;

public Circle(double r){

this.r =r ;

}

public void getArea(){

System.out.println("圆形的面积："+ PI\*r\*r);

}

public void getLength(){

System.out.println("圆形的周长："+ 2\*PI\*r);

}

}

class Rect extends MyShape{

int width;

int height;

public Rect(int width , int height){

this.width = width;

this.height = height;

}

public void getArea(){

System.out.println("矩形的面积："+ width\*height);

}

public void getLength(){

System.out.println("矩形的周长："+ 2\*(width+height));

}

}

class Demo12 {

public static void main(String[] args)

{

/\*

//需求1

Circle c = new Circle(4.0);

print(c);

Rect r = new Rect(3,4);

print(r);

\*/

//需求2

MyShape m = getShape(0); //调用了使用多态的方法，定义的变量类型要与返回值类型一致。

m.getArea();

m.getLength();

}

//需求1： 定义一个函数可以接收任意类型的图形对象，并且打印图形面积与周长。

public static void print(MyShape s){ // MyShpe s = new Circle(4.0);

s.getArea();

s.getLength();

}

// 需求2： 定义一个函数可以返回任意类型的图形对象。

public static MyShape getShape(int i){

if (i==0){

return new Circle(4.0);

}else{

return new Rect(3,4);

}

}

}

Day10//强制类型转换,接口关系下的多态,内部类,异常//

Demo01

/\*

抽象类：

什么是时候使用抽象类：

描述一类事物的时候，发现该事物确实存在着某种 行为，但是 目前这种

是不具体的，这时候应该抽取这种行为的声明，而不去实现该种行为，这时候

这种行为我们把它称为抽象的行为，这时候应该使用抽象类。

抽象类要注意的细节：

1. 如果一个方法没有方法体，那么该方法必须使用abstract修饰。

2. 如果一个类函数抽象方法，那么这个类肯定是一个抽象类或者接口。

3. 抽象类不能创建对象。

4. 抽象类是含有构造方法的。

5. 抽象类可以存在非抽象方法与抽象方法。

6. 抽象类可以不存在抽象方法。

7. 非抽象类继承抽象类的时候，必须要把抽象类中所有抽象方法全部实现。

abstract不能与以下关键字一起使用

1.abstract 不能与static共同修饰一个方法。

2. abstract 不能与private共同修饰一个方法。

3. abstract不能以final关键字共同修饰一个方法。

接口：

接口的定义格式：

interface 接口名{

成员变量

成员函数..

}

接口的作用：

1. 程序解耦。

2. 定义约束规范

3. 拓展功能。

接口要注意的细节：

1. 接口中成员变量都是常量，默认 的修饰符是 public static fianl.

2. 接口中的方法全部都是抽象的方法，默认的修饰符： public abstract。

3. 接口不能创建对象。

4. 接口是没有构造函数的。

5. 非抽象类通过implements实现接口的时候，必须要把接口中所有方法全部实现。

接口与类之间 的关系： 实现关系。

注意： 一个类可以实现多个接口。

接口与接口之间的关系： 继续关系。

注意： 一个接口是可以继承多个接口的。

多态：父类的引用类型变量指向了子类的对象，或者是接口的引用类型变量指向了接口实现类的对象。

多态要注意细节：

1. 多态情况下，如果子父类存在同名的成员变量时，访问的是父类的成员变量.

2. 多态情况下，如果子父类存在同名的非静态函数时，访问的是子类 的成员函数。

3. 多态情况下，如果子父类存在同名的静态函数时，访问的是父类的成员函数。

4. 多态情况下， 不能访问子类特有的成员。

多态的应用：

1.多态用于形参类型的时候，可以接收更多类型的数据

2. 多态用于返回值类型的时候，可以返回更多类型的数据.

\*/

Demo02

/\*

目前多态情况下不能访问子类特有的成员。

如果需要访问子类特有的成员，那么需要进行类型强制转换.

基本数据类型的转换

小数据类型-------->大的数据类型 自动类型转换

大数据类型--------->小数据类型 强制类型转换 小数据类型 变量名 = （小数据类型）大数据类型;

引用数据类型的转换

小数据类型--------->大数据类型 自动类型转换。

大数据类型--------->小数据类型 强制类型转换

类型转换最场景的问题： java.lang.ClassCastException。 强制类型转换失败。

\*/

//动物类

abstract class Animal{

String name;

public Animal(String name){

this.name = name;

}

public abstract void run();

}

//老鼠

class Mouse extends Animal{

public Mouse(String name){

super(name);

}

public void run(){

System.out.println(name+"四条腿慢慢的走!");

}

//老鼠特有方法---打洞

public void dig(){

System.out.println("老鼠在打洞..");

}

}

//鱼

class Fish extends Animal{

public Fish(String name){

super(name);

}

public void run(){

System.out.println(name+"摇摇尾巴游啊游 !");

}

//吹泡泡

public void bubble(){

System.out.println(name+"吹泡泡...!");

}

}

class Demo2

{

public static void main(String[] args)

{

/\*

Animal a = new Mouse("老鼠"); //多态

//调用子类特有的方法

Mouse m = (Mouse)a; //强制类型转换

m.dig();

\*/

Mouse m = new Mouse("米老鼠");

Fish f = new Fish("草鱼");

print(f);

}

//需求：定义一个函数可以接收任意类型的动物对象，在函数内部要调用到动物特有的方法。

public static void print(Animal a){ // Animal a = new Mouse("米老鼠");

//先判断再强转！

if(a instanceof Fish){

Fish f = (Fish)a;

f.bubble();

}else if(a instanceof Mouse){

Mouse m = (Mouse)a;

m.dig();

}

}

}

Demo03

/\*

多态： 父类的引用类型变量指向了子类的对象或者是接口类型的引用类型变量指向了接口实现类 的对象。

实现关系下的多态：

接口 变量 = new 接口的实现类对象。

\*/

interface Dao{ //接口的方法全部都是非静态的方法。

public void add();

public void delete();

}

//接口的实现类

class UserDao implements Dao{

public void add(){

System.out.println("添加员工成功！！");

}

public void delete(){

System.out.println("删除员工成功！！");

}

}

class Demo3

{

public static void main(String[] args)

{

//实现关系下的多态

Dao d = new UserDao(); //接口的引用类型变量指向了接口实现类的对象。

d.add();

}

}

Demo04

/\*

内部类：一个类定义在另外一个类的内部，那么该类就称作为内部类。

内部类的class文件名： 外部类$内部类. 好处：便于区分该class文件是属于哪个外部类的。

内部类的类别：

1. 成员内部类:

成员内部类的访问方式：

方式一：在外部类提供一个方法创建内部类的对象进行访问。

方式2二：在其他类直接创建内部类的对象。 格式：外部类.内部类 变量名 = new 外部类().new 内部类();

注意： 如果是一个静态内部类，那么在其他类创建 的格式：

外部类.内部类 变量名 = new 外部类.内部类();

内部类的应用场景： 我们在描述A事物的时候，发现描述的A事物内部还存在另外一个比较复杂的事物B时候，而且这个比较复杂事物B还需要访问A事物的属性等数据，那么这时候我们就可以使用内部类描述B事物。

比如： 人--->心脏

class 人{

血

氧气

等....

class 心脏{

}

}

内部类的好处：内部类可以直接访问外部类的所有成员。

内部类要注意的细节：

1. 如果外部类与内部类存在同名的成员变量时，在内部类中默认情况下是访问内部类的成员变量。

可以通过 "外部类.this.成员变量名" 指定访问外部类的成员。

2. 私有的成员内部类只能在外部类提供一个方法创建(实例化)内部类的对象进行访问，不能在其他类创建对象了。

3. 成员内部类中 一旦出现了静态的成员，那么该类也必须使用static修饰。

2. 局部内部类：

\*/

//外部类

class Outer{

//成员变量

int x = 100; // Outer.class文件被加载到内存的时候存在内存中。 静态的成员数据是不需要对象存在才能访问。

//成员内部类

static class Inner{

Static int i = 10;

public void print(){

System.out.println("这个是成员内部类的print方法！"+i);

}

}

//在外部的方法中创建了内部类的对象，然后调用内部方法。

public void instance(){

Inner inner = new Inner();

inner.print();

}

}

//其他类：在其他类直接创建内部类的对象。

// 格式：外部类.内部类 变量名 = new 外部类().new 内部类();

class Demo4

{

public static void main(String[] args)

{

/\*

System.out.println(Outer.Inner.i);

Outer outer = new Outer();

outer.instance();

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

inner.print();

\*/

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

inner.print();

}

}

Demo05

/\*

局部内部类： 在一个类的方法内部，定义另外一个类，那么另外一个类就称作为局部内部类。

局部内部类要注意的细节：

1. 如果局部内部类访问了一个局部变量，那么该局部变量必须使用final修饰、

\*/

class Outer{

String name= "外部类的name";

public void test(){

//局部变量

final int y =100; // y 什么时候从内存中消失？ 方法执行完毕之后y消失。

//局部内部类

class Inner{

/\*

当test方法执行完毕之后，那么y马上从内存中消失，而Inner对象在方法执行完毕的时候还没有从内存中消失，而inner对象的print方法还在访问着y变量，这时候的y变量已经消失了，那么就给人感觉y的生命变量已经被延长了.

解决方案： 如果一个局部内部类访问一个局部变量的时候，那么就让该局部内部类访问这个局部变量的复制品。

\*/

int x = 10;

public void print(){

System.out.println("这个是局部内部类的print方法.."+y);

}

}

Inner inner = new Inner(); //这个inner对象什么时候消失？ Inner对象的生命周期比局部变量y的生命周期要长。

inner.print();

}

}

class Demo5

{

public static void main(String[] args)

{

Outer outer = new Outer();

outer.test();

}

}

Demo06

/\*

匿名内部类：没有类名的类就称作为匿名内部类。

匿名内部类的好处：简化书写。

匿名内部类的使用前提：必须存在继承或者实现关系才能使用。

匿名内部类一般是用于实参。

\*/

abstract class Animal{

public abstract Animal run();

public abstract void sleep();

}

class Outer{

public void print(){

//需求： 在方法内部定义一个类继承Animal类，然后调用run方法与sleep()。

/\*

//局部内部类

class Dog extends Animal{

public void run(){

System.out.println("狗在跑..");

}

public void sleep(){

System.out.println("狗趴在睁开眼睛睡..");

}

}

//创建对象

Dog d = new Dog();

d.run();

d.sleep();

\*/

//匿名内部类 ：匿名内部类只是没有类名，其他的一概成员都是具备的。

// 匿名内部类与Animal是继承 的关系。 目前是创建Animal子类的对象.

Animal a = new Animal(){ //多态

//匿名内部的成员

public Animal run(){

System.out.println("狗在跑..");

return this;

}

public void sleep(){

System.out.println("狗趴在睁开眼睛睡..");

}

//特有的方法

public void bite(){

System.out.println("狗在咬人..");

}

};

a.bite();

a.run();

a.sleep();

}

}

class Demo6

{

public static void main(String[] args)

{

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

Outer outer = new Outer();

outer.print();

}

}

Demo07

//实现关系下匿名内部类

/\*

interface Dao{

public void add();

}

class Outer{

public void print(){

//创建一个匿名内部类的对象

new Dao(){ //不是接口不能创建对象吗？怎么现在又可以了？

public void add(){

System.out.println("添加成功");

}

}.add();

}

}

\*/

class Demo7

{

public static void main(String[] args)

{

//匿名内部类

test ( new Dao(){

public void add(){

System.out.println("添加员工成功");

}

} ) ;

}

//调用这个方法...

public static void test(Dao d){ //形参类型是一个接口引用..

d.add();

}

}

Demo08

/\*

异常：

我们的java程序也是会存在某些不正常 的情况的，这些不正常的 情况我们就统称异常。

异常体系：

--------| Throwable

------------| Error

------------| Exception

Throwable常用的方法：

toString() 返回当前异常对象的完整类名+病态信息。

getMessage() 返回的是创建Throwable传入的字符串信息。

printStackTrace() 打印异常的栈信息。

\*/

class Demo8

{

public static void main(String[] args)

{

/\*

//创建了一个Throwable对象。

Throwable t = new Throwable("头晕，感冒..");

String info = t.toString();

String message = t.getMessage();

System.out.println("toString: "+ info); // java.lang.Throwable 包名+类名 = 完整类名

System.out.println("message: "+ message);

\*/

test();

}

public static void test(){

//

Throwable t = new Throwable();

t.printStackTrace();

}

}

Demo09

/\*

异常的体系：

----------| Throwable 所以异常或者错误类的超类

--------------|Error 错误 错误一般是用于jvm或者是硬件引发的问题，所以我们一般不会通过代码去处理错误的。

--------------|Exception 异常 是需要通过代码去处理的。

如何区分错误与异常呢：

如果程序出现了不正常的信息，如果不正常的信息的类名是以Error结尾的，那么肯定是一个错误。

如果是以Exception结尾的，那么肯定就是一个异常。

\*/

class Demo9

{

public static void main(String[] args)

{

//java虚拟机在默认的情况下只能管理64m内存。

byte[] buf = new byte[1024\*1024];

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

}

}

Demo10

/\*

疑问： 下面的信息是通过printStackTrace方法打印出来，那么异常对象从何而来呢？

Exception in thread "main" java.lang.ArithmeticException: / by zero

at Demo10.div(Demo10.java:10)

at Demo10.main(Demo10.java:5)

jvm运行到a/b这个语句的时候，发现b为0，除数为0在我们现实生活中是属于

不正常的情况，jvm一旦发现了这种不正常的情况时候，那么jvm就会马上创建

一个对应的异常对象，并且会调用这个异常对象 的printStackTrace的方法来处理。

异常的处理：

方式一：捕获处理

捕获处理的格式：

try{

可能发生异常的代码；

}catch(捕获的异常类型 变量名){

处理异常的代码....

}

捕获处理要注意的细节：

1. 如果try块中代码出了异常经过了处理之后，那么try-catch块外面的代码可以正常执行。

2. 如果try块中出了异常的代码，那么在try块中出现异常代码后面的代码是不会执行了。

3. 一个try块后面是可以跟有多个catch块的，也就是一个try块可以捕获多种异常的类型。

4. 一个try块可以捕获多种异常的类型,但是捕获的异常类型必须从小到大进行捕获，否则编译报错。

疑问一 : 异常的处理感觉没有多大作用，因为都是输出一个话而已?

异常处理非常有用，只不过是由于我们目前所接触的知识点太过于局限而已。

疑问二： 以后捕获处理 的时候是否就是捕获Exception即可？

错的，因为我们在现实开发中遇到不同的异常类型的时候，我往往会有不同 的处理方式。

所以要分开不同的异常类型处理。

\*/

class Demo10

{

public static void main(String[] args)

{

int[] arr = null;

div(4,0,arr);

}

public static void div(int a , int b,int[] arr){

int c = 0;

try{

c = a/b; //jvm在这句话的时候发现了不正常的情况，那么就会创建一个对应的异常对象。

System.out.println("数组的长度："+ arr.length);

}catch(ArithmeticException e){

//处理异常的代码....

System.out.println("异常处理了....");

System.out.println("toString:"+ e.toString());

}catch(NullPointerException e){

System.out.println("出现了空指针异常....");

}catch(Exception e){

System.out.println("我是急诊室，包治百病！");

}

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

}

}

Demo11

/\*

异常的处理方式----抛出处理

抛出处理（throw throws）

抛出处理要注意的细节：

1. 如果一个方法的内部抛出了一个异常 对象，那么必须要在方法上声明抛出。

2. 如果调用了一个声明抛出异常 的方法，那么调用者必须要处理异常。

3. 如果一个方法内部抛出了一个异常对象，那么throw语句后面的代码都不会再执行了（一个方法遇到了throw关键字，该方法也会马上停止执行的）。

4. 在一种情况下，只能抛出一种类型异常对象。

throw 与throws两个关键字：

1. throw关键字是用于方法内部的，throws是用于方法声声明上的。

2. throw关键字是用于方法内部抛出一个异常对象的，throws关键字是用于在方法声明上声明抛出异常类型的。

3. throw关键字后面只能有一个异常对象，throws后面一次可以声明抛出多种类型的 异常。

疑问：何时使用抛出处理？何时捕获处理？原则是如何？

如果你需要通知到调用者，你代码出了问题，那么这时候就使用抛出处理.

如果代码是直接与用户打交道遇到了异常千万不要再抛，再抛的话，就给了用户了。

这时候就应该使用捕获处理。

\*/

class Demo11

{

public static void main(String[] args)

{

try{

int[] arr = null;

div(4,0,arr); //调用了一个 声明抛出异常类型 的方法

}catch(Exception e){

System.out.println("出现异常了...");

e.printStackTrace();

}

}

public static void div(int a, int b,int[] arr) throws Exception,NullPointerException {

if(b==0){

throw new Exception(); //抛出一个异常对象...

}else if(arr==null){

throw new NullPointerException();

}

int c = a/b;

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

}

}

Day11//自定义异常//运行时异常//编译时异常//finally//导包语句//权限修饰符//模版模式//清除重复元素//

Demo01（复习）

/\*

内部类

内部类类别：

成员内部类: 在一个类的成员位置定义另外一个类，那么另外 一个 类就称作为成员内部类。

成员内部类的访问方式：

方式1： 在外部类内提供一个方法创建内部类的对象进行访问。

方式2： 在其他类创建内部类的对象进行访问。 创建的格式： 外部类.内部类 变量名 = new 外部类().new 内部类();

注意：如果是静态内部类在其他类的创建对象方式： 外部类.内部类 变量名= new 外部类.内部类();

成员内部类要注意的事项：

1. 成员内部类可以直接访问外部类成员。

2. 如果成员内部类与外部类存在同名的成员，在内部类中默认是访问内部类 的成员。

成员通过“外部类.this.成员”指定访问外部类的成员。

3. 如果成员内部类出现了静态的成员，那么该成员内部类也必须使用static修饰。

4. 如果成员内部类是私有的，那么创建内部类的对象就只能在外部类提供方法创建。

局部内部类 ： 在一个类的方法内部定义另外一个类, 另外一个类就称作为局部内部类。

局部内部类要注意： 如果局部内部类访问了局部变量，那么该变量需要使用fianl修饰。 （原因： 因为生命周期不一致，所以内部类访问局部变量其实是一个复制品。）

匿名内部类 : 没有类名的类。

匿名内部类的好处： 简化书写。

匿名内部类的使用前提： 必须存在继承或者实现关系。

匿名内部类的格式：

new 父类(父接口){

匿名内部类的成员;

};

异常：

异常的体系：

---------|Throwable

--------------| Error （错误） 错误一般是由于jvm或者是硬件引发的问题，所以我们一般都不会通过代码去处理。

--------------| Exception (异常) 异常我们一般都会通过代码去处理的。

异常的处理方式

方式一：捕获处理

捕获处理的格式

try{

可能发生异常的代码

}catch(捕获的异常类型 变量名){

处理异常的代码

}

捕获处理要注意的细节：

1. 如果一个try块中出现了异常的代码，经过处理之后，那么try-catch块外面的代码可以正常执行。

2. 如果一个try块中出现了异常的代码，那么在try块中出现异常的代码后面 的语句无法执行。

3. 一个try块后面可以跟多个catch块，也就是一个try块可以捕获多种异常的类型，但是捕获的

异常类型必须从小到大进行捕获。

方式二：抛出处理(throw throws)

抛出处理要注意的细节：

1. 如果一个方法内部抛出了一个编译时异常对象，那么该方法必须要声明抛出。

2. 如果调用了一个声明抛出编译时异常的方法，那么调用者必须要处理。

3. 如果一个方法抛出了一个异常对象，那么该方法也会马上停止（一个方法遇到了throw关键字，那么该方法就会马上停止）

4. 在一种情况下只能抛出一种异常对象。

throw 关键字是用于方法体之内抛出异常对象 的，throws是用于方法 声明上声明抛出异常类型的。

\*/

Demo02

/\*

sun提供了很多的异常类给我们用于描述程序中各种的不正常情况，但是sun 给我

提供异常类还不足以描述我们现实生活中所有不正常情况，那么这时候我们就需要

自定义异常类。

需求: 模拟feiQ上线的时候,如果没有插上网线，那么就抛出一个没有插上网线的异常，

如果已经插上了网上，那么就正常显示好友列表。

自定义异常类的步骤： 自定义一个类继承Exception即可。

\*/

//自定义了一个没有网线的异常类了。

class NoIpException extends Exception{

public NoIpException(String message){

super(message); //调用了Exception一个参数的构造函数。

}

}

class Demo2

{

public static void main(String[] args)

{

String ip = "192.168.10.100";

ip = null;

try{

feiQ(ip); // 如果调用了一个声明抛出异常类型的方法，那么调用者必须要处理。

}catch(NoIpException e){

e.printStackTrace();

System.out.println("马上插上网线！");

}

}

public static void feiQ(String ip) throws NoIpException{

if(ip==null){

throw new NoIpException("没有插网线啊，小白！");

}

System.out.println("正常显示好友列表..");

}

}

Demo03

/\*

需求：模拟你去吃木桶饭，如果带钱少于了10块，那么就抛出一个没有带够钱的异常对象，

如果带够了，那么就可以吃上香喷喷的地沟油木桶饭.

\*/

//定义没钱的异常

class NoMoneyException extends Exception {

public NoMoneyException(String message){

super(message);

}

}

class Demo3

{

public static void main(String[] args)

{

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

try{

eat(9);

}catch(NoMoneyException e){

e.printStackTrace();

System.out.println("跟我洗碗一个月！！");

}

}

public static void eat(int money) throws NoMoneyException{

if(money<10){

throw new NoMoneyException("吃霸王餐");

}

System.out.println("吃上了香喷喷的地沟油木桶饭!!");

}

}

Demo04

/\*

异常体系：

--------| Throwable 所有错误或者异常的父类

--------------| Error（错误）

--------------| Exception(异常) 异常一般都通过代码处理

------------------| 运行时异常: 如果一个方法内部抛出了一个运行时异常，那么方法上 可以声明也可以不 声明，调用者可以以处理也可以不处理。

------------------| 编译时异常(非运行时异常、受检异常): 如果一个方法内部抛出了一个编译时异常对象，那么方法上就必须要声明，而且调用者也必须要处理。

运行时异常： RuntimeException以及RuntimeException子类 都是属于运行时异常。

编译时异常： 除了运行时异常就是编译异常。

疑问： 为什么java编译器会如此严格要求编译时异常，对运行时异常如此宽松？

运行时异常都是可以通过程序员良好的编程习惯去避免，所以java编译器就没有严格要求处理运行时异常。

\*/

import java.security.acl.\*;

class Demo4

{

public static void main(String[] args) throws InterruptedException

{

int[] arr = null;

div(4,0,arr);

Object o = new Object();

o.wait();

}

public static void div(int a , int b ,int[] arr) {

if(b==0){

return;

}

int c = a/b;

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

if(arr!=null){

System.out.println("数组的长度： "+arr.length);

}

}

}

Demo05

/\*

finally 块:

finally块的 使用前提是必须要存在try块才能使用。

finally块的代码在任何情况下都会执行的，除了jvm退出的情况。

finally非常适合做资源释放的工作，这样子可以保证资源文件在任何情况下都 会被释放。

try块的三种组合方式：

第一种： 比较适用于有异常要处理，但是没有资源要释放的。

try{

可能发生异常的代码

}catch(捕获的异常类型 变量名){

处理异常的代码

}

第二种：比较适用于既有异常要处理又要释放资源的代码。

try{

可能发生异常的代码

}catch(捕获的异常类型 变量名){

处理异常的代码

}finally{

释放资源的代码;

}

第三种： 比较适用于内部抛出的是运行时异常，并且有资源要被释放。

try{

可能发生异常的代码

}finally{

释放资源的代码;

}

\*/

class Demo5

{

public static void main(String[] args)

{

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

div(4,0);

}

public static void div(int a, int b){

try{

if(b==0){

System.exit(0);//退出jvm

}

int c = a/b;

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

}catch(Exception e){

System.out.println("出了除数为0的异常...");

throw e;

}finally{

System.out.println("finall块的代码执行了..");

}

}

}

Demo06

/\*

fianlly释放资源的代码

\*/

import java.io.\*;

class Demo6

{

public static void main(String[] args)

{

FileReader fileReader = null;

try{

//找到目标文件

File file = new File("f:\\a.txt");

//建立程序与文件的数据通道

fileReader = new FileReader(file);

//读取文件

char[] buf = new char[1024];

int length = 0;

length = fileReader.read(buf);

System.out.println("读取到的内容："+ new String(buf,0,length));

}catch(IOException e){

System.out.println("读取资源文件失败....");

}finally{

try{

//关闭资源

fileReader.close();

System.out.println("释放资源文件成功....");

}catch(IOException e){

System.out.println("释放资源文件失败....");

}

}

}

}

Demo07

/\*

包:

java中的包就相当于windows文件夹。

包的作用：

1. 解决类名重复产生冲突的问题。

2. 便于软件版本的发布。

定义包的格式：

package 包名;

包名命名规范：包名全部小写。

包语句要注意的事项：

1. package语句必须位于java文件中中第一个语句。

2. 如果一个类加上了包语句，那么该类的完整类名就是: 包名.类名

3. 一个java文件只能有一个包语句。

问题： 每次编译的时候都需要自己创建一个文件夹，把对应 的class文件存储 到文件夹中。烦！！！

javac -d 指定类文件的存放路径 java源文件

\*/

package aa;

class Demo1

{

public static void main(String[] args)

{

System.out.println("这个是Demo1的main方法...");

}

}

Demo08

/\*

有了包之后类与类之间的访问：

问题： 有了包之后类与类之间的访问每次都必须 要写上包名！烦！！！！

解决方案： sum提供导包语句让我们解决该问题。

导包语句作用：简化书写。 (误区： 把一个类导入到内存中)

导包语句的格式：

import 包名.类名; （导入xxx包中某个类）

导包语句要注意的细节：

1. 一个java文件中可以出现多句导包语句。

2. "\*"是 导包语句的通配符。可以匹配任何 的类名。

3. import aa.\*; 是不会作用于aa包下面的子包的。

推荐使用：import 包名.类名; 因为使用\*通配符会导致结构不清晰。

什么时候使用import语句:

1. 相互访问的两个类不是在同一个包下面，这时候就需要使用到导包语句。

2. java.lang 是默认导入 的，不需要我们自己导入。

\*/

package aa;

public class Demo3

{

/\*

static{

System.out.println("这个是Dmeo3的静态代码块...");

}

\*/

public void print(){

System.out.println("这个是Dmeo3的print方法...");

}

}

Demo09

/\*

权限修饰符： 权限修饰符就是控制被修饰的成员的范围可见性。

public(公共) protected(受保护) default(缺省) private (大到小)

同一个类 true true true true

同一个包 true true true false

子父类 true true false false

不同包 true false false false

注意： 在不同包下面只有public 与 protected 可以访问，而且protected必须是在继承关系下才能够访问。

\*/

Demo10

/\*

打jar包: 需要使用到jdk的开发工具（jar.exe）.

jar的用法：

使用格式：

jar cvf jar文件的名字 class文件或者是文件夹

打jar包要注意的事项：

1. 一个程序打完了jar之后 必须要在清单文件上指定入口类： 格式 Main-Class: 包名.类名

2. jar包双击运行仅对于图形化界面的程序起作用，对控制台的程序不起作用。

jar文件的作用：

1. 方便用户快速运行一个项目。

2. 提供工具类以jar包的形式给别人使用。

如果使用jar包里面的类必须要先设置classpath路径。

jre = jvm+ 核心类库

\*/

package qq;

import javax.swing.\*;

class Demo9

{

public static void main(String[] args)

{

System.out.println("QQ程序..");

JFrame frame = new JFrame("QQ程序");

frame.setSize(400,500);

frame.setVisible(true); //设置窗口可见。

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

}

Demo11

/\*

模板模式 ：解决某类事情的步骤有些是固定的，有些是会发生变化的，那么这时候我们可以为这类事情提供一个模板代码，从而提高效率 。

需求；编写一个计算程序运行时间 的模板。

模板模式的步骤：

1. 先写出解决该类事情其中 的一件的解决方案。

2. 分析代码，把会发生变化的代码抽取出来独立成一个方法。把该方法描述成一个抽象的方法。

3. 使用final修饰模板方法，防止别人重写你的模板方法。

\*/

abstract class MyRuntime{

public final void getTime(){

long startTime = System.currentTimeMillis(); //记录开始的时间

code();

long endTime = System.currentTimeMillis(); //记录结束的时间.

System.out.println("运行时间 ："+ (endTime-startTime));

}

public abstract void code();

}

class Demo11 extends MyRuntime

{

public static void main(String[] args)

{

Demo11 d = new Demo11();

d.getTime();

}

//code方法内部就写要计算运行时间 的代码；

public void code(){

int i = 0;

while(i<100){

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

i++;

}

}

}

Demo12

/\*

11.目前有数组” int[] arr = {11,2, 4, 2, 10, 11}，

定义一个函数清除该数组的重复元素，返回的数组存储了那些非重复的元素而且数组不准浪费长度。

分析：

1. 确定新数组的长度。 原数组的长度-重复元素个数

\*/

import java.util.\*;

class Demo12 {

public static void main(String[] args)

{

int[] arr = {11,2, 4, 2, 10, 11};

arr = clearRepeat(arr);

System.out.println("清除重复元素的数组："+ Arrays.toString(arr));

}

public static int[] clearRepeat(int[] arr){

//先计算出重复元素的个数:

int count = 0; //记录重复元素的个数

for(int i = 0 ; i < arr.length-1 ; i++){

for(int j = i+1 ; j<arr.length ; j++){

if(arr[i]==arr[j]){

count++;

break;

}

}

}

//新数组 的长度

int newLength = arr.length - count;

//创建一个新的数组

int[] newArr = new int[newLength];

int index = 0 ; //新数组的索引值

//遍历旧数组

for(int i = 0 ; i< arr.length ; i++){

int temp = arr[i]; //旧数组中的元素

boolean flag = false; //默认不是重复元素

//拿着旧数组 的元素 与新数组的每个元素比较一次。

for(int j = 0 ; j< newArr.length ; j++){

if(temp==newArr[j]){

flag = true;

break;

}

}

if(flag == false){

newArr[index++] = temp;

}

}

return newArr;

}

}

Demo13

class User{

String userName;

String password;

}

class Demo13

{

public static void main(String[] args)

{

Scanner scanner = new Scanner(System.in);

User[] users = new User[10];

while(true){

System.out.println("请选择功能 : A(注册) B(查看用户)");

String option = scanner.next();

if(option.equals("A")){

System.out.println("请输入用户名:");

String userName = scanner.next();

System.out.println("请输入密码:");

String password = scanner.next();

}else if(option.equals("B")){

}

}

}

}

Demo14

package myutil;

public class MyArrays

{

public static String toString(int[] arr){

String result = "";

for(int i = 0 ; i < arr.length ; i++){

if(i==0){

result +="["+ arr[i]+",";

}else if(i==arr.length-1){

result += arr[i]+"]";

}else{

result += arr[i]+",";

}

}

return result;

}

}

Day12//eclipse的使用//Object类//字符串入门//StringBuffer//

Demo01

/\*

使用eclipse 好处：

1. ctrl + / 内部补全。

2. 不需要你再使用javac工具编译，你在Eclipse上面一旦保存，Eclipse自动编译。

3. 精准报错，自动修复(ctrl+ 1)。

\*/

Demo02

package cn.itcast.object;

/\*

java是面向对象的语言，核心思想：找适合 的对象做适合 的事情：

方式一：自定义类，然后通过自定义的类创建对象。

方式二：sun提供了很多的类给我使用，我们只需要认识这些类，我们就可以通过这些类创建对象了。

Object类是所有类的终极父类。 任何一个类都继承了Object类。

Object类：

Object类常用的方法：

toString(); 返回该对象的字符串表示。 返回一个字符串用于描述该对象的。

疑问： toString() 有何作用？ 重写toString之后，我们直接输出一个对象的时候，就会输出符合我们所需求的格式数据。

equals(Object obj) 用于比较两个对象的内存地址，判断两个对象是否为同一个对象。

hashCode() 返回该对象的哈希码值(大家可以把哈希码就 理解成是对象的内存地址)。

java中的规范：一般我们重写了一个类的equals方法，我们都会重写它的hashCode方法。

java是开源....源代码公开...

查看源代码的方式：

方式一： 按住ctrl键，单击你需要看的源代码.

方式二：把光标移动到你需要查看代码处按下F3.

为什么我们要查看源代码：

1，查看源代码可以了解别人是如何写出这个技术的，让我们了解的更加的深入。

2. 吸收大牛的思想。

看源代码最忌讳的一点： 不要每行代码都弄懂是什么意思，能看个大概 猜出他意思就足矣。

\*/

class Person{

int id;

String name;

public Person(int id, String name) {

this.id = id;

this.name = name;

}

public Person() {

}

//目前我需要直接输出一个对象的时候，输出 的格式是： 编号：110 姓名： 狗娃 这种格式。 目前Object的toString方法无法满足子类的需求，那么这时候我们就应该对Object类的toString进行重写。

@Override

public String toString() {

return "编号："+ this.id + " 姓名："+this.name;

}

//为什么要重写：Object的equals方法默认比较的是两个对象的内存地址，我目前需要比较的是两个对象的身份证，所以Object类的equals方法不符合我现在的需求。

@Override

public boolean equals(Object obj) {

Person p = (Person)obj;

return this.id== p.id;

}

@Override

public int hashCode() {

return this.id;

}

}

public class Demo1 {

public static void main(String[] args) {

/\*

Object o = new Object();

System.out.println(o.toString()); // java.lang.Object@18b3364 返回的字符串表示： 完整类名+@+ 对象的哈希码

System.out.println(o); // 通过查看源代码得知，直接输出一个对象 的时候，实际上在println方法内部会调用这个 调用的toString方法，把把toString方法返回的内容输出。

//疑问： 为什么直接输出一个对象的时候和输出对象的toString方法返回的字符串结果是一样的呢？

Person p1 = new Person(110,"狗娃");

System.out.println("p1:"+p1);

//如果我们能够输出一个p对象的时候，输出的格式： 编号：110 姓名： 狗娃..

Person p2 = new Person(112,"狗剩");

System.out.println("p2:"+p2);

\*/

Person p1 = new Person(110,"狗娃");

Person p2 = new Person(110,"陈富贵");

//需求：在现实生活中只要两个人的身份证一致，那么就是同一个人。

System.out.println("p1与p2是同一个对象吗？"+ p1.equals(p2));

System.out.println("p1哈希码:"+ p1.hashCode());

System.out.println("p2哈希码:"+ p2.hashCode());

}

}

Demo03

package cn.itcast.stringbuffer;

/\*

字符串特点：字符串是常量；它们的值在创建之后不能更改.

字符串的内容一旦发生了变化，那么马上会创建一个新的对象。

注意： 字符串的内容不适宜频繁修改，因为一旦修改马上就会创建一个新的对象。

如果需要频繁修改字符串 的内容，建议使用 字符串缓冲类（StringBuffer）。

StringBuffer 其实就是一个存储字符 的容器。

\*/

public class Demo1 {

public static void main(String[] args) {

String str1 = "hello";

String str2 = str1+" world";

System.out.println("str1与str2是同一个 对象吗？"+(str1==str2));//false--->创建了不同的对象嘛！

}

}

Demo04

package cn.itcast.stringbuffer;

/\*

如果需要频繁修改字符串 的内容，建议使用字符串缓冲 类（StringBuffer）。

StringBuffer 其实就是一个存储字符 的容器。

笔试题目：使用Stringbuffer无参的构造函数创建 一个对象时，默认的初始容量是多少？ 如果长度不够使用了，自动增长多少倍？

StringBuffer 底层是依赖了一个字符数组才能存储字符数据 的，该字符串数组默认 的初始容量是16， 如果字符数组的长度不够使用，自动增长1倍。

容器的具备 的行为

增加

删除

修改

查看

判断

\*/

public class Demo2 {

public static void main(String[] args) {

//先使用StringBuffer无参的构造函数创建一个字符串缓冲类。

StringBuffer sb = new StringBuffer();

sb.append("java");

sb.append("java");

sb.append("java");

sb.append("java");

sb.append("java");

System.out.println(sb);

}

}

Demo05

package cn.itcsat.string;

/\*

String 字符串类:

笔试题目：new String("abc")创建了几个对象？

------------->两个对象， 一个对象是位于字符串常量池中，一个对象是位于堆内存中。

\*/

public class Demo1 {

public static void main(String[] args) {

String str1 = "hello";

String str2 = "hello";

String str3 = new String("hello");

String str4 = new String("hello");

System.out.println("str1==str2?"+(str1==str2)); // true

System.out.println("str2==str3?"+(str2==str3)); //false

System.out.println("str3==str4?"+(str3==str4)); // false

System.out.println("str3.equals(str4)?"+(str3.equals(str4))); //true

//是String类重写了Object的equals方法，比较的是两个字符串对象 的内容 是否一致。

// "=="用于比较 引用数据类型数据的时候比较的是两个对象 的内存地址，equals方法默认情况下比较也是两个对象 的内存地址。

test(null);

}

public static void test(String str){

if("中国".equals(str))

{

System.out.println("回答正确");

}else{

System.out.println("回答错误");

}

}

}

Demo06

package cn.itcsat.string;

/\*

String 的构造方法：

String() 创建一个空内容 的字符串对象。

String(byte[] bytes) 使用一个字节数组构建一个字符串对象

String(byte[] bytes, int offset, int length)

bytes : 要解码的数组

offset： 指定从数组中那个索引值开始解码。

length：　要解码多个元素。

String(char[] value) 使用一个字符数组构建一个字符串。

String(char[] value, int offset, int count) 使用一个字符数组构建一个字符串， 指定开始的索引值，与使用字符个数。

String(int[] codePoints,int offset,int count)

String(String original)

记住： 使用字节数组或者字符数组都可以构建一个字符串对象。

\*/

public class Demo2 {

public static void main(String[] args) {

String str = new String();//创建一个空字符串对象

byte[] buf = {97,98,99};

str = new String(buf); //使用一个字节数组构建一个字符串对象

str = new String(buf,1,2); //使用一个字节数组构建一个字符串对象,指定开始解码 的索引值和解码的个数。

char[] arr = {'明','天','是','圣','诞'};

str = new String(arr); //使用字符数组构建一个字符串

str = new String(arr,3,2);

int[] buf2 = {65,66,67};

str = new String(buf2,0,3);

str = new String("abc");

System.out.println("字符串的内容："+str);

}

}

Demo07

package cn.itcsat.string;

/\*

2.2 获取方法

int length() 获取字符串的长度

char charAt(int index) 获取特定位置的字符 (角标越界)

int indexOf(String str) 查找子串第一次出现的索引值,如果子串没有出现 在字符串中，那么则返回-1表示。

int lastIndexOf(String str) 查找子串最后一次出现的索引值 , 如果子串没有出现 在字符串中，那么则返回-1表示

\*/

public class Demo3 {

public static void main(String[] args) {

String str = "abc中国ab中国";

System.out.println("字符串的字符 个数: " + str.length() );

System.out.println("根据索引值获取对应 的字符:"+ str.charAt(3));

System.out.println("查找子串第一次出现的索引值：" + str.indexOf("中国"));

System.out.println("查找子串最后一次出现的索引值：" + str.lastIndexOf("中国"));

}

}

Demo08

package cn.itcsat.string;

import java.util.Arrays;

/\*

2.3 判断方法

boolean endsWith(String str) 是否以指定字符结束

boolean isEmpty()是否长度为0 如：“” null V1.6

boolean contains(CharSequences) 是否包含指定序列 应用：搜索

boolean equals(Object anObject) 是否相等

boolean equalsIgnoreCase(String anotherString) 忽略大小写是否相等

2.4 转换方法

char[] toCharArray() 将字符串转换为字符数组

byte[] getBytes();

字节数组与字符数组、字符串他们三者之间是可以互相转换。

\*/

public class Demo4 {

public static void main(String[] args) {

String str = "Demo.java";

System.out.println("是否以指定的字符串结束:"+ str.endsWith("ja"));

//str = "";

System.out.println("判断字符串是否为空内容："+str.isEmpty());

System.out.println("判断字符串是否包含指定的内容："+ str.contains("Demo"));

System.out.println("判断两个字符串的内容是否一致："+ "DEMO.JAVA".equals(str));

System.out.println("判断两个字符串的内容是否一致(忽略大小写比较):"+ "DEMO.JAVA".equalsIgnoreCase(str));

//转换的方法

char[] buf = str.toCharArray(); //把字符串转换字符数组

System.out.println("字符数组："+ Arrays.toString(buf));

byte[] buf2 = str.getBytes(); //把字符串转字节数组

System.out.println("字节数组："+ Arrays.toString(buf2));

}

}

Demo09

package cn.itcsat.string;

import java.util.Arrays;

/\*

其他方法

String replace(String oldChar, String newChar) 替换

String[] split(String regex) 切割

String substring(int beginIndex) 指定开始的索引值截取子串

String substring(int beginIndex, int endIndex)指定开始 与结束的索引值截取子串

String toUpperCase() 转大写

String toLowerCase() 转小写

String trim() 去除字符串首尾的空格

\*/

public class Demo5 {

public static void main(String[] args) {

String str = "今天晚上不考试";

System.out.println("指定新内容替换旧 的内容:"+ str.replace("不", "要好好"));

str = "大家-下-午-好";

String[] arr = str.split("-"); //根据指定的字符进行切割 。

System.out.println("字符串数组的内容："+ Arrays.toString(arr));

str = "广州传智播客";

System.out.println("指定开始的索引值截取子串:"+ str.substring(2));

System.out.println("指定开始与结束的索引值截取子串:"+ str.substring(2,6)); //包头不包尾

注意：截取的内容是包括开始的索引值，不包括结束的索引值， 截取的位置是结束的索引值-1.

str = "abC中国";

System.out.println("转大写：" + str.toUpperCase());

str = "AbdfSDD";

System.out.println("转小写："+ str.toLowerCase());

str = " 大家最近 都非常努力 ";

System.out.println("去除字符串首尾的空格："+ str.trim());

}

}

Demo10

package cn.itcsat.string;

/\*

需求1：自己实现trim的方法。

需求2： 获取上传文件名 "D:\\20120512\\day12\\Demo1.java"。

需求3： 将字符串对象中存储的字符反序。 新中国好 -----> 好国中新

需求4： 求一个子串在整串中出现的次数。

\*/

public class Demo6 {

public static void main(String[] args) {

String str =" 传智 播客 ";

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

str = "D:\\20120512\\day12\\Demo1.java";

getFileName(str);

str = "新中国好";

System.out.println("翻转后的字符串："+ reverse(str));

str = "abcjavaabcjavaphpjava"; //java

getCount(str, "java");

}

//需求4：统计子串出现的次数

public static void getCount(String str,String target){

int count = 0 ; //用于记录出现的次数

int fromIndex = 0; // 记录从那个索引值开始寻找目标子串

while((fromIndex = str.indexOf(target, fromIndex))!=-1){

//如果indexof方法返回 的不是-1，那么就是已经找到了目标 元素。

count++;

fromIndex = fromIndex+target.length();

}

System.out.println("出现的次数："+ count);

}

//需求3：将字符串对象中存储的字符反序

public static String reverse(String str){

char[] arr = str.toCharArray();//把数组转成字符数组

for(int startIndex = 0 , endIndex=arr.length-1 ; startIndex<endIndex; startIndex++,endIndex--){

char temp = arr[startIndex];

arr[startIndex] = arr[endIndex];

arr[endIndex] = temp;

}

//使用字符数组构建一个字符串。

return new String(arr);

}

//需求2： 获取上传文件名 "D:\\20120512\\day12\\Demo1.java"。

public static void getFileName(String path){

int index = path.lastIndexOf("\\");

String fileName = path.substring(index+1);

System.out.println("文件名："+ fileName);

}

// 需求1：自己实现trim的方法。

public static String myTrim(String str){

//先转换成字符 数组

char[] arr = str.toCharArray();

//定义两个 变量记录开始与结束 的索引值

int startIndex = 0 ;

int endIndex = arr.length -1;

//确定开始 的索引值

while(true){

if(arr[startIndex]==' '){

startIndex++;

}else{

break;

}

}

//确定结束 的索引值：

while(true){

if(arr[endIndex]==' '){

endIndex--;

}else{

break;

}

}

//截取子串返回

return str.substring(startIndex,endIndex+1);

}

}

Day13//StringBuffer字符串缓冲类//System.RunTime类//Math类Date类//验证码//自定义线程//线程的生命周期//线程的安全问题//

Demo01

package cn.itcast.stringbuffer;

/\*

字符串特点：字符串是常量；它们的值在创建之后不能更改.

字符串的内容一旦发生了变化，那么马上会创建一个新 的对象。

注意： 字符串的内容不适宜频繁修改，因为一旦修改马上就会创建一个新的对象。

如果需要频繁修改字符串 的内容，建议使用字符串缓冲 类（StringBuffer）。

StringBuffer 其实就是一个存储字符 的容器。

\*/

public class Demo1 {

public static void main(String[] args) {

String str1 = "hello";

String str2 = str1+" world";

System.out.println("str1与str2是同一个 对象吗？"+(str1==str2));

}

}

Demo02

package cn.itcast.stringbuffer;

/\*

如果需要频繁修改字符串的内容，建议使用字符串缓冲 类（StringBuffer）。

StringBuffer 其实就是一个存储字符的容器。

笔试题目：使用Stringbuffer无参的构造函数创建 一个对象时，默认的初始容量是多少？ 如果长度不够使用了，自动增长多少倍？

StringBuffer 底层是依赖了一个字符数组才能存储字符数据 的，该字符串数组默认 的初始容量是16， 如果字符数组的长度不够使用，自动增长1倍。

StringBuffer 是一个存储字符的容器

容器的具备的行为:

增加

append(boolean b) 可以添加任意类型 的数据到容器中

insert(int offset, boolean b) 指定插入的索引值，插入对应的内容。

删除

delete(int start, int end) 根据指定的开始与结束的索引值删除对应的内容。

deleteCharAt(int index) 根据指定 的索引值删除一个字符。

修改

replace(int start, int end, String str) 根据指定 的开始与结束索引值替代成指定的内容。

reverse() 翻转字符串缓冲类的内容。 abc--->cba

setCharAt(int index, char ch) 把指定索引值的字符替换指定的字符。

substring(int start, int end) 根据指定的索引值截取子串。

ensureCapacity(int minimumCapacity) 指定StringBuffer内部的字符数组长度的。

查看

indexOf(String str, int fromIndex) 查找指定的字符串第一次出现的索引值,并且指定开始查找的位置。

lastIndexOf(String str)

capacity() 查看当前字符数组的长度。

length()

charAt(int index)

toString() 把字符串缓冲类的内容转成字符串返回。

StringBuffer 与 StringBuilder的相同处与不同处：

相同点：

1. 两个类都是字符串缓冲类。

2. 两个类的方法都是一致的。

不同点：

1. StringBuffer是线程安全的,操作效率低 ，StringBuilder是线程非安全的,操作效率高。

2. StringBuffer是jdk1.0出现 的，StringBuilder 是jdk1.5的时候出现的。

推荐使用： StringBuilder，因为操作效率高。

\*/

public class Demo2 {

public static void main(String[] args) {

//先使用StringBuffer无参的构造函数创建一个字符串缓冲类。

StringBuffer sb = new StringBuffer();

sb.append("abcjavaabc");

/\*

添加

sb.append(true);

sb.append(3.14f);

插入

sb.insert(2, "小明");

\*/

/\*

删除

sb.delete(2, 4); // 删除的时候也是包头不包尾

sb.deleteCharAt(3); //根据指定 的索引值删除一个字符

修改

sb.replace(2, 4, "陈小狗");

sb.reverse(); // 翻转字符串的内容

sb.setCharAt(3, '红');

String subString = sb.substring(2, 4);

System.out.println("子串的内容："+ subString);

查看

int index = sb.indexOf("abc", 3);

System.out.println("索引值为："+index);

sb.append("javajava");

System.out.println("查看字符数组的长度："+ sb.capacity());

\*/

System.out.println("存储的字符个数："+sb.length());

System.out.println("索引指定的索引值查找字符："+sb.charAt(2) );

System.out.println("字符串缓冲类的内容："+ sb);

String content = sb.toString();

test(content);

}

public static void test(String str){

}

}

Demo03

package cn.itcast.other;

/\*

Math 数学类， 主要是提供了很多的数学公式。

abs(double a) 获取绝对值

ceil(double a) 向上取整

floor(double a) 向下取整

round(float a) 四舍五入

random() 产生一个随机数. 大于等于 0.0 且小于 1.0 的伪随机 double 值

\*/

public class Demo4 {

public static void main(String[] args) {

System.out.println("绝对值:"+Math.abs(-3));

System.out.println("向上(大)取整："+Math.ceil(3.14));//4

System.out.println("向下(小)取整："+Math.floor(-3.14)); //-4

System.out.println("四舍五入:"+Math.round(3.54));

System.out.println("随机数："+Math.random());

}

}

Demo04

package cn.itcast.other;

import java.io.IOException;

import javax.management.RuntimeErrorException;

/\*

RunTime 该类类主要代表了应用程序运行的环境。

getRuntime() 返回当前应用程序的运行环境对象。

exec(String command) 根据指定的路径执行对应的可执行文件。

freeMemory() 返回 Java 虚拟机中的空闲内存量。。 以字节为单位

maxMemory() 返回 Java 虚拟机试图使用的最大内存量。

totalMemory() 返回 Java 虚拟机中的内存总量

\*/

public class Demo2 {

public static void main(String[] args) throws IOException, InterruptedException {

Runtime runtime = Runtime.getRuntime();

// Process process = runtime.exec("C:\\Windows\\notepad.exe");

// Thread.sleep(3000); //让当前程序停止3秒。

// process.destroy();

System.out.println(" Java虚拟机中的空闲内存量。"+runtime.freeMemory());

System.out.println("Java 虚拟机试图使用的最大内存量:"+ runtime.maxMemory());

System.out.println("返回 Java 虚拟机中的内存总量:"+ runtime.totalMemory());

}

}

Demo05

package cn.itcast.other;

import java.text.ParseException;

import java.text.SimpleDateFormat;

import java.util.Calendar;

import java.util.Date;

/\*

日期类 Date

Calendar

日期格式化类 SimpleDateFormat

\*/

public class Demo3 {

public static void main(String[] args) throws ParseException {

/\*Date date = new Date(); // 获取当前的系统时间

System.out.println("年份："+ date.getYear());\*/

/\*

Calendar calendar = Calendar.getInstance(); //获取当前的系统时间。

System.out.println("年："+ calendar.get(Calendar.YEAR));

System.out.println("月："+ (calendar.get(Calendar.MONTH)+1));

System.out.println("日："+ calendar.get(Calendar.DATE));

System.out.println("时："+ calendar.get(Calendar.HOUR\_OF\_DAY));

System.out.println("分："+ calendar.get(Calendar.MINUTE));

System.out.println("秒："+ calendar.get(Calendar.SECOND));

// 显示 当前系统时间: 2014年12月26日 xx时xx分xx秒

\* 日期格式化类 SimpleDateFormat

\* 作用1： 可以把日期转换转指定格式的字符串 format()--->格式化

\* 作用2： 可以把一个 字符转换成对应的日期。 parse()---> 解析 生日

\*

\*/

Date date = new Date(); //获取当前的系统时间。

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss") ; //使用了默认的格式创建了一个日期格式化对象。

String time = dateFormat.format(date); //调用格式化：把日期转换转指定格式的字符串

System.out.println("当前的系统时间："+ time);

String birthday = "2000年12月26日 11:29:08";

Date date2 = dateFormat.parse(birthday); //注意： 解析指定的字符串格式必须要与SimpleDateFormat的模式要一致。

System.out.println(date2);

Date date21 =new Date();

SimpleDateFormat dateFormat2 = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");

String time2 =dateFormat.format(date21);

String time21=dateFormat.format(date);

System.out.println("当前的系统时间："+time);

String birthday1= "2000年12月26日 11:28:08";

Date date22=dateFormat.parse(birthday1);

System.out.println(date22);

}

}

Demo06

package cn.itcast.other;

import java.util.Arrays;

import java.util.Properties;

/\*

System 系统类 主要用于获取系统的属性数据。

System类常用的方法：

arraycopy(Object src, int srcPos, Object dest, int destPos, int length) 一般

src - 源数组。

srcPos - 源数组中的起始位置。

dest - 目标数组。

destPos - 目标数据中的起始位置。

length - 要复制的数组元素的数量。

currentTimeMillis() 获取当前系统系统。 重点

exit(int status) 退出jvm 如果参数是0表示正常退出jvm，非0表示异常退出jvm。 一般

gc() 建议jvm赶快启动垃圾回收期回收垃圾。

getenv(String name) 根据环境变量的名字获取环境变量。

getProperty(key)

finalize() 如果一个对象被垃圾回收 器回收的时候，会先调用对象的finalize()方法。

\*/

class Person{

String name;

public Person(String name) {

this.name = name;

}

@Override

public void finalize() throws Throwable {

super.finalize();

System.out.println(this.name+"被回收了..");

}

}

public class Demo1 {

public static void main(String[] args) {

/\*

int[] srcArr = {10,12,14,16,19};

//把srcArr的数组元素拷贝到destArr数组中。

int[] destArr = new int[4];

System.arraycopy(srcArr, 1, destArr, 0,4);

//System.exit(0); //jvm退出.. 注意： 0或者非0的 数据都可以退出jvm。对于用户而言没有任何区别。

System.out.println("目标数组的元素："+ Arrays.toString(destArr)); // 0 14 16 0

System.out.println("当前的系统时间：" + System.currentTimeMillis());

System.out.println("环境变量："+System.getenv("JAVA\_HOME"));

for(int i = 0 ; i<4; i++){

new Person("狗娃"+i);

System.gc(); //建议马上启动垃圾回收期

}

Properties properties = System.getProperties(); //获取系统的所有属性。

properties.list(System.out);

\*/

String value = System.getProperty("os.name"); //根据系统的属性名获取对应的属性值

System.out.println("当前系统："+value);

}

}

Demo07

package cn.itcast.thread;

/\*

进程 ： 正在执行的程序称作为一个进程。 进程负责了内存空间的划分。

问题： windows号称是多任务的操作系统,那么windows是同时运行多个应用程序吗？

从宏观的角度： windows确实是在同时运行多个应用程序。

从微观角度： cpu是做了一个快速切换执行的动作，由于速度态度，所以我感觉不到在切换 而已。

线程： 线程在一个进程 中负责了代码的执行，就是进程中一个执行路径，

多线程： 在一个进程中有多个线程同时在执行不同的任务。

疑问 ：线程负责了代码的执行，我们之前没有学过线程，为什么代码可以执行呢？

运行任何一个java程序，jvm在运行的时候都会创建一个main线程执行main方法中所有代码。

一个java应用程序至少有几个线程？

至少有两个线程， 一个是主线程负责main方法代码的执行，一个是垃圾回收器线程,负责了回收垃圾。

多线程的好处：

1. 解决了一个进程能同时执行多个任务的问题。

2. 提高了资源的利用率。

多线程的弊端：

1. 增加cpu的负担。

2. 降低了一个进程中线程的执行概率。

3. 引发了线程安全 问题。

4. 出现了死锁现象。

如何创建多线程：

创建线程的方式：

方式一：

1. 自定义一个类继承Thread类。

2. 重写Thread类的run方法 , 把自定义线程的任务代码写在run方法中

疑问： 重写run方法的目的是什么？

每个线程都有自己的任务代码，jvm创建的主线程的任务代码就是main方法中的所有代码, 自定义线程的任务代码就写在run方法中，自定义线程负责了run方法中代码。

3. 创建继承了Thread的子类的对象，并且调用start方法开启线程。

注意： 一个线程一旦开启，那么线程就会执行run方法中的代码，run方法千万不能直接调用，直接调用run方法就相当调用了一个普通的方法而已

并没有开启新的线程。

\*/

public class Demo1 extends Thread {

@Override //把自定义线程的任务代码写在run方法中。

public void run() {

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

System.out.println("自定义线程："+i);

}

}

public static void main(String[] args) {

//创建了自定义的线程对象。

Demo1 d = new Demo1();

//调用start方法启动线程

d.start();

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

System.out.println("main线程："+i);

}

}

}

Demo08

package cn.itcast.thread;

/\*

需求： 模拟QQ视频与聊天同时在执行。

\*/

class TalkThread extends Thread{

@Override

public void run() {

while(true){

System.out.println("hi，你好！开视频呗...");

}

}

}

class VideoThread extends Thread{

@Override

public void run() {

while(true){

System.out.println("视频视频....");

}

}

}

public class Demo2 {

public static void main(String[] args) {

TalkThread talkThread = new TalkThread();

talkThread.start();

VideoThread videoThread = new VideoThread();

videoThread.start();

}

}

Demo09

package cn.itcast.thread;

/\*

线程常用的方法：

Thread(String name) 初始化线程的名字

setName(String name) 设置线程对象名

getName() 返回线程的名字-->(Thread.currentThread().getName())

sleep() 线程睡眠指定的毫秒数。 静态的方法， 那个线程执行了sleep方法代码那么就是那个线程睡眠。

currentThread() 返回当前的线程对象，该方法是一个静态的方法， 注意： 那个线程执行了currentThread()代码就返回那个线程 的对象。

getPriority() 返回当前线程对象的优先级 默认线程的优先级是5

setPriority(int newPriority) 设置线程的优先级 虽然设置了线程的优先级，但是具体的实现取决于底层的操作系统的实现（最大的优先级是10 ，最小的1 ， 默认是5）。

\*/

public class Demo3 extends Thread {

public Demo3(String name){

super(name); //调用了Thread类的一个 参数的构造方法。

}

@Override

public void run() {

/\*System.out.println("this:"+ this);

System.out.println("当前线程对象：" + Thread.currentThread()); \*/

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

System.out.println(Thread.currentThread().getName()+":"+i);

/\*try {

Thread.sleep(100); //为什么在这里不能抛出异常，只能捕获？？ Thread类的run方法没有抛出异常类型，所以子类不能抛出异常类型。

} catch (InterruptedException e) {

e.printStackTrace();

} \*/

}

}

public static void main(String[] args) throws InterruptedException {

//创建了一个线程对象

Demo3 d = new Demo3("狗娃");

d.setPriority(10); //设置线程 的优先级。 优先级的数字越大，优先级越高 ， 优先级的范围是1~10

d.start();

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

System.out.println(Thread.currentThread().getName()+":"+i);

}

/\*

System.out.println("自定义线程的优先级："+d.getPriority()); //线程的优先级默认是5

System.out.println("主线程的优先级："+Thread.currentThread().getPriority());

d.start();

d.setName("铁蛋"); //setName设置线程的名字

d.start(); //开启线程

Thread mainThread = Thread.currentThread();

System.out.println("主线程的名字："+ mainThread.getName());

\*/

}

}

Demo10

package cn.itcast.thread;

/\*

需求： 模拟3个窗口同时在售50张 票 。

问题1 ：为什么50张票被卖出了150次？

出现的原因： 因为num是非静态的，非静态的成员变量数据是在每个对象中都会维护一份数据,三个线程对象就会有三份。

解决方案：把num票数共享出来给三个线程对象使用。使用static修饰。

问题2： 出现了线程安全问题 ?

线程 安全问题的解决方案：sun提供了线程同步机制让我们解决这类问题的。

java线程同步机制的方式：

方式一：同步代码块

同步代码块的格式：

synchronized(锁对象){

需要被同步的代码...

}

同步代码块要注意事项：

1. 任意的一个对象都可以做为锁对象。

2. 在同步代码块中调用了sleep方法并不是释放锁对象的。

3. 只有真正存在线程安全问题的时候才使用同步代码块，否则会降低效率的。

4. 多线程操作的锁对象必须 是唯一共享的。否则无效。

作为锁对象：

1. 类名.class

2. 对象.getClass()

需求： 一个银行账户5000块，两夫妻一个拿着 存折，一个拿着卡，开始取钱比赛，每次只能取一千块，要求不准出现线程安全问题。

方式二：同步函数

出现线程安全问题的根本原因：

1. 存在两个或者两个以上 的线程对象,而且线程之间共享着一个资源。

2. 有多个语句操作了共享资源。

\*/

class SaleTicket extends Thread{

static int num = 50;//票数 非静态的成员变量,非静态的成员变量数据是在每个对象中都会维护一份数据的。

static Object o = new Object();//锁的”钥匙”

//子类的构造器：在创建SaleTicket对象时传入参数

public SaleTicket(String name) {

super(name);//指定调用父类的带参方法

}

@Override

public void run() {

while(true){

//同步代码块

synchronized (o) {

if(num>0){

System.out.println(Thread.currentThread().getName()+"售出了第"+num+"号票");

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

num--;

}else{

System.out.println("售罄了..");

break;

}

}

}

}

}

public class Demo4 {

public static void main(String[] args) {

//创建三个线程对象，模拟三个窗口

SaleTicket thread1 = new SaleTicket("窗口1");

SaleTicket thread2 = new SaleTicket("窗口2");

SaleTicket thread3 = new SaleTicket("窗口3");

//开启线程售票

thread1.start();

thread2.start();

thread3.start();

}

}

Demo11

//随机生成验证码：

import java.util.Random;

public class Demo5 {

public static void main(String[] args) {

char[] arr={'s','b','g','h','过','傻'};

//字符串容器：

StringBuilder sb=new StringBuilder();

//创建随机数对象

Random random=new Random();

//循环4次

for(int i=0;i<4;i++){

//随机取出数组的长度，赋值到index

int index=random.nextInt(arr.length);

sb.append(arr[index]);

}

System.out.println("验证码:"+sb);

}

}

Demo12

package cn.itcast.other;

import java.util.Random;

/\*

随机数类

Random

需求： 编写一个函数随机产生四位的验证码。

\*/

public class Demo5 {

public static void main(String[] args) {

/\*

Random random = new Random();

int randomNum = random.nextInt(10)+1; //产生 的 随机数就是0-10之间

System.out.println("随机数："+ randomNum);

\*/

char[] arr = {'中','国','传','a','Q','f','B'};

StringBuilder sb = new StringBuilder();

Random random = new Random();

//需要四个随机数，通过随机数获取字符数组中的字符，

for(int i = 0 ; i< 4 ; i++){

int index = random.nextInt(arr.length); //产生的 随机数必须是数组的索引值范围之内的。

sb.append(arr[index]);

}

System.out.println("验证码："+ sb);

}

}

Day14//同步函数//线程实现方式//线程通讯//wait方法//notify方法//后台进程//join方法//集合的引入//Collection方法

Demo01

package cn.itcast.thread;

/\*

进程: 进程就是正在运行的应用程序。 进程了负责了内存空间划分。

线程： 一个进程中的 代码是由线程去执行的，线程也就是进程中一个执行路径。

多线程： 一个进程中有多个线程可以同时执行任务。

多线程 的好处：

1. 解决一个进程中可以同时执行多个任务的问题。

2. 提高了资源利用率。

多线程的弊端：

1. 增加了cpu的负担。

2. 降低了一个进程中线程 的执行概率。

3. 出现了线程 安全问题。

4. 会引发死锁现象。

自定义线程 的实现方式：

方式一 ：

1. 自定义一个类继承Thread类。

2. 重写Thread类的run方法，把自定义线程的任务代码写在run方法上。

3. 创建Thread的子类对象，并且调用start方法启动一个线程。

注意：千万不要直接调用run方法，调用start方法的时候线程就会开启，线程一旦开启就会执行run方法中代码，如果直接调用

run方法，那么就 相当于调用了一个普通的方法而已。

线程安全问题：

线程安全出现 的根本原因：

1. 存在两个或者两个以上 的线程对象共享同一个资源。

2. 多线程操作共享资源的代码 有多句。

线程安全问题的解决方案：

方式一： 可以使用同步代码块去解决。

格式：

synchronized(类名.class){

需要被同步的代码

}

同步代码块要注意的事项：

1. 锁对象可以是任意的一个对象。

2. 一个线程在同步代码块中sleep了，并不会释放锁对象。

3. 如果不存在着线程安全问题，千万不要使用同步代码块，因为会降低效率。

4. 锁对象必须是多线程共享的一个资源，否则锁不住。

方式二：同步函数 ： 同步函数就是使用synchronized修饰一个函数。

同步函数要注意的事项 ：

1. 如果是一个非静态的同步函数的锁 对象是this对象，如果是静态的同步函数的锁 对象是当前函数所属的类的字节码文件（class对象）。

2. 同步函数的锁对象是固定的，不能由你来指定 的。

推荐使用： 同步代码块。

原因：

1. 同步代码块的锁对象可以由我们随意指定，方便控制。同步函数的锁对象是固定 的，不能由我们来指定。

2. 同步代码块可以很方便控制需要被同步代码的范围，同步函数必须是整个函数 的所有代码都被同步了。

需求： 一个银行账户5000块，两夫妻一个拿着存折，一个拿着卡，开始取钱比赛，每次只能取一千块，要求不准出现线程安全问题。

\*/

class BankThread extends Thread{

Static int count = 5000;//静态

public BankThread(String name){

super(name);

}

@Override //

public synchronized void run() {

while(true){

synchronized ("锁") {

if(count>0){

System.out.println(Thread.currentThread().getName()+"取走了1000块,还剩余"+(count-1000)+"元");

count= count - 1000;

}else{

System.out.println("取光了...");

break;

}

}

}

}

//静态的函数---->函数所属 的类的字节码文件对象--->BankThread.class 唯一的。

public static synchronized void getMoney(){

}

}

public class Demo1 {

public static void main(String[] args) {

//创建两个线程对象

BankThread thread1 = new BankThread("老公");

BankThread thread2 = new BankThread("老婆");

//调用start方法开启线程取钱

thread1.start();

thread2.start();

}

}

Demo02

package cn.itcast.thread;

/\*

java中同步机制解决了线程安全问题，但是也同时引发死锁现象。

死锁现象：

死锁现象出现 的根本原因：

1. 存在两个或者两个以上的线程。

2. 存在两个或者两个以上的共享资源。

死锁现象的解决方案： 没有方案。只能尽量避免发生而已。

\*/

class DeadLock extends Thread{

public DeadLock(String name){

super(name);

}

public void run() {

if("张三".equals(Thread.currentThread().getName())){

synchronized ("遥控器") {

System.out.println("张三拿到了遥控器，准备 去拿电池!!");

synchronized ("电池") {

System.out.println("张三拿到了遥控器与电池了，开着空调爽歪歪的吹着...");

}

}

}else if("狗娃".equals(Thread.currentThread().getName())){

synchronized ("电池") {

System.out.println("狗娃拿到了电池，准备去拿遥控器!!");

synchronized ("遥控器") {

System.out.println("狗娃拿到了遥控器与电池了，开着空调爽歪歪的吹着...");

}

}

}

}

}

public class Demo2 {

public static void main(String[] args) {

DeadLock thread1 = new DeadLock("张三");

DeadLock thread2 = new DeadLock("狗娃");

//开启线程

thread1.start();

thread2.start();

}

}

Demo03

package cn.itcast.thread;

/\*

自定义线程的创建方式:

方式一 ： (继承)

1. 自定义一个类继承Thread类。

2. 重写Thread类的run方法，把自定义线程的任务代码写在run方法上。

3. 创建Thread的子类对象，并且调用start方法启动一个线程。

注意：千万不要直接调用run方法，调用start方法的时候线程就会开启，线程一旦开启就会执行run方法中代码，如果直接调用run方法，那么就 相当于调用了一个普通的方法而已。

方式二：(接口)

1. 自定义一个类实现Runnable接口。

2. 实现Runnable接口 的run方法，把自定义线程的任务定义在run方法上。

3. 创建Runnable的实现类(Demo03)的对象。

4. 创建Thread类 的对象，并且把Runnable的实现类的对象，作为实参传递。

5. 调用Thread对象 的start方法开启一个线程。

问题1： 请问Runnable实现类的对象是线程对象吗？

Runnable实现类的对象并不是一个线程对象，只不过是实现了Runnable接口 的对象而已。

只有是Thread或者是Thread的子类才是线程对象。

问题2： 为什么要把Runnable实现类的对象作为实参传递给Thread对象呢？作用是什么？

作用就是把Runnable实现类的对象的run方法作为了线程的任务代码去执行了。

推荐使用： 第二种（接口）。 实现Runable接口的。

原因： 因为java单继承 ,多实现的。

\*/

public class Demo3 implements Runnable{

@Override

public void run() {

/\*System.out.println("this："+ this);

System.out.println("当前线程："+ Thread.currentThread());\*/

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

System.out.println(Thread.currentThread().getName()+":"+i);

}

}

public static void main(String[] args) {

//创建Runnable实现类的对象

Demo3 d = new Demo3();

//创建Thread类的对象， 把Runnable实现类对象作为实参传递。

Thread thread = new Thread(d,"狗娃"); //Thread类使用Target变量记录了d对象，

//调用thread对象的start方法开启线程。

thread.start();

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

System.out.println(Thread.currentThread().getName()+":"+i);

}

}

/\*

Thread类 的run方法

\* @Override

public void run() {

if (target != null) {

target.run(); //就相当于Runnable实现类的对象的run方法作为了Thread对象的任务代码了。

}

}

\*/

}

Demo04

package cn.itcast.thread;

class SaleTicket implements Runnable{

int num = 50; // 票数

@Override

public void run() {

while(true){

synchronized ("锁") {

if(num>0){

System.out.println(Thread.currentThread().getName()+"售出了第"+ num+"号票");

num--;

}else{

System.out.println("售罄了..");

break;

}

}

}

}

}

public class Demo4 {

public static void main(String[] args) {

//创建了一个Runnable实现类的对象

SaleTicket saleTicket = new SaleTicket();

//创建三个线程对象模拟三个窗口

Thread thread1 = new Thread(saleTicket,"窗口1");

Thread thread2 = new Thread(saleTicket,"窗口2");

Thread thread3 = new Thread(saleTicket,"窗口3");

//开启线程售票

thread1.start();

thread2.start();

thread3.start();

}

}

Demo05

package cn.itcast.thread;

/\*

线程通讯： 一个线程完成了自己的任务时，要通知另外一个线程去完成另外一个任务.

生产者与消费者

wait(): 等待 如果线程执行了wait方法，那么该线程会进入等待的状态，等待状态下的线程必须要被其他线程调用notify方法才能唤醒。

notify()： 唤醒 唤醒线程池等待线程其中的一个。

notifyAll() : 唤醒线程池所有等待 线程。

wait与notify方法要注意的事项：

1. wait方法与notify方法是属于Object对象的。

2. wait方法与notify方法必须要在同步代码块或者是同步函数中才能使用。

3. wait方法与notify方法必需要由锁对象调用。

问题一：出现了线程安全问题。 价格错乱了...

\*/

//产品类

class Product{

String name; //名字

double price; //价格

boolean flag = false; //产品是否生产完毕的标识，默认情况是没有生产完成。

}

//生产者

class Producer extends Thread{

Product p ; //产品

public Producer(Product p) {

this.p = p ;

}

@Override

public void run() {

int i = 0 ;

while(true){

synchronized (p) {

if(p.flag==false){

if(i%2==0){

p.name = "苹果";

p.price = 6.5;

}else{

p.name="香蕉";

p.price = 2.0;

}

System.out.println("生产者生产出了："+ p.name+" 价格是："+ p.price);

p.flag = true;

i++;

p.notifyAll(); //唤醒消费者去消费

}else{

//已经生产 完毕，等待消费者先去消费

try {

p.wait(); //生产者等待

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

}

//消费者

class Customer extends Thread{

Product p;

public Customer(Product p) {

this.p = p;

}

@Override

public void run() {

while(true){

synchronized (p) {

if(p.flag==true){ //产品已经生产完毕

System.out.println("消费者消费了"+p.name+" 价格："+ p.price);

p.flag = false;

p.notifyAll(); // 唤醒生产者去生产

}else{

//产品还没有生产,应该 等待生产者先生产。

try {

p.wait(); //消费者也等待了...

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}

}

}

public class Demo5 {

public static void main(String[] args) {

Product p = new Product(); //产品

//创建生产对象

Producer producer = new Producer(p);

//创建消费者

Customer customer = new Customer(p);

//调用start方法开启线程

producer.start();

customer.start();

}

}

Demo06

package cn.itcast.thread;

/\*

线程的停止：

1. 停止一个线程 我们一般都会通过一个变量去控制的。

2. 如果需要停止一个处于等待状态下的线程，那么我们需要通过变量配合notify方法或者interrupt()来使用。

\*/

public class Demo6 extends Thread {

boolean flag = true;

public Demo6(String name){

super(name);

}

@Override

public synchronized void run() {

int i = 0 ;

while(flag){

try {

this.wait(); //狗娃等待..

} catch (InterruptedException e) {

System.out.println("接收到了异常了....");

}

System.out.println(Thread.currentThread().getName()+":"+i);

i++;

}

}

public static void main(String[] args) {

Demo6 d = new Demo6("狗娃");

d.setPriority(10);

d.start();

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

System.out.println(Thread.currentThread().getName()+":"+i);

//当主线程的i是80的时候停止狗娃线程。

//d.interrupt(); // interrupt()根本就是无法停止一个线程。

if(i==80){

d.flag = false;

d.interrupt(); //把线程的等待状态强制清除，被清除状态的线程会接收到一个InterruptedException。

/\*synchronized (d) {

d.notify();

}\*/

}

}

}

}

Demo07

package cn.itcast.thread;

/\*

守护线程setDaemon() （后台线程）:在一个进程中如果只剩下了守护线程，那么守护线程也会死亡。

需求： 模拟QQ下载更新包。

一个线程默认都不是守护线程。

\*/

public class Demo7 extends Thread {

public Demo7(String name){

super(name);

}

@Override

public void run() {

for(int i = 1 ; i<=100 ; i++){

System.out.println("更新包目前下载"+i+"%");

if(i==100){

System.out.println("更新包下载完毕,准备安装..");

}

try {

Thread.sleep(100);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) {

Demo7 d = new Demo7("后台线程");

d.setDaemon(true); //setDaemon() 设置线程是否为守护线程，true为守护线程， false为非守护线程。

// System.out.println("是守护线程吗？"+ d.isDaemon()); //判断线程是否为守护线程。

d.start();

for(int i = 1 ; i<=100 ; i++){

System.out.println(Thread.currentThread().getName()+":"+i);

}

}

}

Demo08

package cn.itcast.thread;

/\*

join方法。 加入

\*/

//老妈

class Mon extends Thread{

public void run() {

System.out.println("妈妈洗菜");

System.out.println("妈妈切菜");

System.out.println("妈妈准备炒菜，发现没有酱油了..");

//叫儿子去打酱油

Son s= new Son();

s.start();

try {

s.join(); //加入。 一个线程如果执行join语句，那么就有新的线程加入，执行该语句的线程必须要让步给新加入的线程先完成任务，然后才能继续执行。

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("妈妈继续炒菜");

System.out.println("全家一起吃饭..");

}

}

class Son extends Thread{

@Override

public void run() {

System.out.println("儿子下楼..");

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println("儿子一直往前走");

System.out.println("儿子打完酱油了");

System.out.println("上楼，把酱油给老妈");

}

}

public class Demo8 {

public static void main(String[] args) {

Mon m = new Mon();

m.start();

}

}

Demo09

package cn.itcast.collection;

/\*

数组： 存储同一种数据类型的集合容器.

数组的特点：

1. 只能存储同一种数据类型的数据。

2. 一旦初始化，长度固定。

3. 数组中的元素与元素之间的内存地址是连续的。

注意： Object类型的数组可以存储任意类型的数据。

需求： 收集我们班同学的兴趣爱好。

数组存储兴趣爱好：

String[] arr= new String[1000];

\*/

public class Demo1 {

public static void main(String[] args) {

Object[] arr = new Object[10];

arr[1] = "abc";

arr[2] = 'a';

arr[3] = 12;

}

}

Demo10

package cn.itcast.collection;

import java.util.ArrayList;

import java.util.Collection;

/\*

集合：集合是存储对象数据的集合容器。

集合比数组的优势：

1. 集合可以存储任意类型的对象数据，数组只能存储同一种数据类型 的数据。

2. 集合的长度是会发生变化的，数组的长度是固定的。

-------| Collection 单例集合的跟接口。

----------| List 如果是实现了List接口的集合类，具备的特点： 有序，可重复。

----------| Set 如果是实现了Set接口的集合类，具备特点： 无序，不可重复。

Collection接口中的方法：

增加

add(E e) 添加成功返回true，添加 失败返回false.

addAll(Collection c) 把一个集合 的元素添加到另外一个集合中去。

删除

clear()

remove(Object o)

removeAll(Collection c)

retainAll(Collection c)

查看

size()

判断

isEmpty()

contains(Object o)

containsAll(Collection<?> c)

迭代

toArray()

iterator()

\*/

public class Demo2 {

public static void main(String[] args) {

Collection c = new ArrayList();

c.add("令计划");

c.add("徐才厚");

c.add("周永康");

System.out.println("添加成功吗？"+c.add("郭美美"));

//创建集合

Collection c2 = new ArrayList();

c2.add("徐才厚");

c2.add("郭美美");

c2.add("狗娃");

/\*

c.addAll(c2); // 把c2的元素的添加到c集合 中去。

\*/

/\*

删除方法:

c.clear(); //clear()清空集合中的元素

System.out.println("删除成功吗？"+c.remove("美美")); // remove 指定集合中的元素删除，删除成功返回true，删除失败返回false.

c.removeAll(c2); //删除c集合中与c2的交集元素。

c.retainAll(c2); //只保留c集合与c2的交集元素，其他的元素一并删除。

\*/

System.out.println("查看元素个数："+c.size());

System.out.println("集合的元素："+ c);

}

}

Demo11

package cn.itcast.collection;

import java.util.ArrayList;

import java.util.Collection;

/\*

判断

isEmpty()

contains(Object o)

containsAll(Collection<?> c)

\*/

class Person{

int id;

String name;

public Person(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "{编号："+this.id+" 姓名："+ this.name+"}";

}

@Override

public boolean equals(Object obj) {

Person p = (Person)obj;

return this.id == p.id ;

}

//java规范： 一般重写equlas方法我们都会重写hashCode方法的。

@Override

public int hashCode() {

return this.id;

}

}

public class Demo3 {

public static void main(String[] args) {

/\*Collection c = new ArrayList();

c.add("令计划");

c.add("徐才厚");

c.add("周永康");

System.out.println("判断集合是否为空元素："+ c.isEmpty());

System.out.println("判断集合中是否存在指定的元素："+ c.contains("薄熙来"));\*/

//集合添加自定义的元素

Collection c = new ArrayList();

c.add(new Person(110,"狗娃"));

c.add(new Person(119,"狗剩"));

c.add(new Person(120,"铁蛋"));

Collection c2 = new ArrayList();

c2.add(new Person(110,"狗娃"));

c2.add(new Person(119,"狗剩"));

c2.add(new Person(104,"陈狗剩"));

System.out.println("c集合有包含c2集合中的所有元素吗？"+ c.containsAll(c2));

//如果在现实生活中，只要身份证编号一致，那么就为同一个人。

System.out.println("存在该元素吗？"+c.contains(new Person(120,"陈铁蛋"))); // 其实contains方法内部是依赖于equals方法进行比较的。

System.out.println("集合的元素："+ c);

}

}

Demo12

package cn.itcast.collection;

import java.util.ArrayList;

import java.util.Arrays;

import java.util.Collection;

/\*

迭代

toArray()

\*/

public class Demo4 {

public static void main(String[] args) {

/\*Collection c = new ArrayList();

c.add("令计划");

c.add("徐才厚");

c.add("周永康");

Object[] arr = c.toArray(); //把集合中的元素全部 存储到一个Object的数组中，并返回。

System.out.println("数组的元素："+Arrays.toString(arr));\*/

Collection c = new ArrayList();

c.add(new Person(110,"狗娃"));

c.add(new Person(119,"狗剩"));

c.add(new Person(120,"铁蛋"));

Object[] arr = c.toArray();

//需求： 把编号是110的人信息 输出。

for(int i = 0 ; i<arr.length ; i++){

Person p = (Person) arr[i]; //从Object数组中取出的元素只能使用Object类型声明变量接收，如果需要其他 的类型需要进行强制类型转换。

if(p.id==110){

System.out.println(p);

}

}

}

}

Day15//迭代器//List接口//迭代器要注意的事项//ArrayList的原理//LinkedList的原理与方法//

Dome01

package cn.itcast.collelction;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

import java.util.Scanner;

/\*

作业2：使用集合实现注册登陆功能，

第一步： 提示用户选择功能， A（注册） B(登陆) 。 要求： 功能选择 的时候要忽略大小写。

注册：

1. 提示用户输入注册的账号(数字)与密码，如果输入账号已经存在集合中，提示用户重新输入。 注册完毕之后，把集合中的所有用户信息打印出来。(使用：toArrry()方法)

登陆：

提示用户输入登陆的账号与密码,如果账号与密码这个用户已经存在集合中，那么登陆成功，否则登陆失败。

\*/

//用户

class User{

int id; //账号

String password; //密码

//生成setget方法

public int getId() {

return id;

}

public void setId(int id) {

this.id = id;

}

public String getPassword() {

return password;

}

public void setPassword(String password) {

this.password = password;

}

//构造器

public User(int id, String password) {

this.id = id;

this.password = password;

}

//重载equals方法

@Override

public boolean equals(Object obj) {

User user = (User)obj;

return this.id==user.id;

}

//重载tostring方法

@Override

public String toString() {

return "{ 账号："+this.id+" 密码："+this.password+"}";

}

}

public class Demo1 {

static Scanner scanner = new Scanner(System.in);

static Collection users = new ArrayList(); //使用该集合保存所有的用户信息..

public static void main(String[] args) {

while(true){

System.out.println("请选择功能 A(注册 ) B(登陆)");

String option = scanner.next();

if("a".equalsIgnoreCase(option)){

reg();

}else if("b".equalsIgnoreCase(option)){

login();

}else{

System.out.println("你的选择有误,请重新输入");

}

}

}

//登录

public static void login() {

System.out.println("请输入账号：");

int id = scanner.nextInt();

System.out.println("请输入密码:");

String password = scanner.next();

//判断集合的用户是否存在该用户名与密码

//遍历集合的元素，查看是否存在该用户信息

boolean isLogin = false; //定义变量用于记录是否登陆成功的信息 , 默认是没有登陆成功的

Iterator it = users.iterator();

while(it.hasNext()){

User user = (User) it.next();

if(user.id==id&&user.password.equals(password)){

//存在该用户信息，登陆成功...

isLogin = true;

}

}

if(isLogin==true){

System.out.println("欢迎登陆...");

}else{

System.out.println("用户名或者密码错误，登陆失败...");

}

}

//注册

public static void reg() {

//110 , 220

User user = null;

while(true){

System.out.println("请输入账号:");

int id = scanner.nextInt(); //220

//创建User对象

user = new User(id,null);

if(users.contains(user)){ // contains底层依赖了equals方法。

//如果存在

System.out.println("该账号已经存在，请重新输入账号");

}else{

//不存在

break;

}

}

System.out.println("请输入密码：");

String password = scanner.next();

user.setPassword(password);

//把user对象保存到集合中

users.add(user);

System.out.println("注册成功!");

System.out.println("当前注册的人员："+users);

}

}

Demo02

package cn.itcast.collelction;

import java.util.ArrayList;

import java.util.Collection;

import java.util.Iterator;

/\*

------| Collection 单例集合的根接口

----------| List 如果是实现了 List接口的集合类，该集合类具备 的特点： 有序、可重复。

----------| Set 如果是实现了Set接口的集合类，该集合具备的特点： 无序，不可 重复。

Collection---迭代的方法：

toArray()

iterator()

迭代器的作用：就是用于抓取集合中的元素。

迭代器的方法：

hasNext() 问是否有元素可遍历。如果有元素可以遍历，返回true，否则返回false 。

next() 获取元素...

remove() 移除迭代器最后一次返回 的元素。

NoSuchElementException 没有元素的异常。

出现的原因： 没有元素可以被迭代了。。。

\*/

public class Demo2 {

public static void main(String[] args) {

Collection c = new ArrayList();

c.add("狗娃");

c.add("狗剩");

c.add("铁蛋");

c.add("美美");

/\*

//遍历集合的元素------>方式一： 可以使用toArray方法。

Object[] arr = c.toArray(); // toArray() 把集合 的元素存储到一个 Object的数组中 返回。

for(int i = 0 ; i<arr.length ; i++){

System.out.print(arr[i]+",");

}

//方式二：要求使用iterator迭代器遍历。

\*/

Iterator it = c.iterator(); //返回一个迭代器 疑问：iterator()方法返回的是一个接口类型，为什么接口又可以调用方法可以使用呢？ iterator 实际 上返回的是iterator接口的实现类对象。

/\*

while(it.hasNext()){ // hasNext() 判断是否有元素可以遍历

System.out.println("元素："+ it.next()); //获取元素

}

\*/

/\*

it.next();

it.next();

it.remove(); //删除迭代器最后一次返回的元素。

\*/

//清空集合 的元素

while(it.hasNext()){

it.next();

it.remove();

}

System.out.println("集合的元素："+ c);

}

}

Demo03

package cn.itcast.collelction;

interface Money{

public void makeMoney();

}

class Worker implements Money{

@Override

public void makeMoney() {

System.out.println("工人在赚钱...");

}

}

class Student implements Money{

@Override

public void makeMoney() {

// TODO Auto-generated method stub

}

}

public class Demo3 {

public static void main(String[] args) {

Money m = test();//--->等价于：Money m=new Worker();

m.makeMoney();//---->等价于：m.makeMoney;

}

public static Money test(){

return new Worker();

}

}

Demo04

package cn.itcast.list;

import java.util.ArrayList;

import java.util.List;

/\*

集合的体系：

----------| Collection 单列集合 的根接口

--------------| List 如果是实现了List接口的集合类，该集合类具备的特点：有序，可重复。

--------------| Set 如果是实现了Set接口的集合类，该集合类具备的特点： 无序，不可重复。

有序： 集合的有序不是指自然顺序，而是指添加进去的顺序与元素出来的顺序是一致的。

List接口中特有方法：

添加

add(int index, E element)

addAll(int index, Collection<? extends E> c)

获取：

get(int index)

indexOf(Object o)

lastIndexOf(Object o)

subList(int fromIndex, int toIndex)

修改：

set(int index, E element)

迭代

listIterator()

\*/

public class Demo1 {

public static void main(String[] args) {

List list = new ArrayList();

list.add("张三");

list.add("李四");

list.add("王五");

list.add("赵六");

list.add("赵六");

System.out.println("集合的元素："+ list);

}

}

Demo05

package cn.itcast.list;

import java.util.ArrayList;

import java.util.List;

/\*

List接口中特有方法：

添加

add(int index, E element)

addAll(int index, Collection<? extends E> c)

获取：

get(int index)

indexOf(Object o)

lastIndexOf(Object o)

subList(int fromIndex, int toIndex)

修改：

set(int index, E element)

List接口中特有的方法具备的特点： 操作的方法都存在索引值。

只有List接口下面的集合类才具备索引值。其他接口下面的集合类都没有索引值。

ctrl + shift + / 添加多行注释

ctrl + shift + \ 取消多行注释.

\*/

public class Demo2 {

public static void main(String[] args) {

List list= new ArrayList();

list.add("狗娃");

list.add("狗剩");

list.add("铁蛋"); //把元素添加到集合的末尾处。

list.add("狗娃");

/\*

//添加方法

list.add(1, "赵本山"); // 把元素添加到集合中的指定索引值位置上。

List list2 = new ArrayList();

list2.add("本山");

list2.add("小沈阳");

list.addAll(2,list2); //把list2的元素添加到list集合指定索引值的位置上。

\*/

/\*

//获取的方法

System.out.println("get方法获取元素："+list.get(1)); //根据索引值获取集合中的元素

使用get方法遍历集合的元素：

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

System.out.print(list.get(i)+",");

}

System.out.println("找出指定元素第一次出现在集合中 的索引值："+ list.indexOf("本山"));

System.out.println("找指定的元素最后一次出现在集合中的索引值："+list.lastIndexOf("狗娃"));

List subList = list.subList(1, 3); //指定开始与结束的索引值截取集合中的元素(包头不包尾)

System.out.println("子集合的元素是："+ subList);

\*/

list.set(3, "赵本山"); //使用指定的元素替换指定索引值位置的元素

System.out.println("集合的元素："+list);

}

}

Demo06

package cn.itcast.list;

import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;

/\*

迭代

listIterator()

ListIterator特有的方法：

添加：

hasPrevious() 判断是否存在上一个元素,Boolean类型

previous() 当前指针先向上移动一个单位，然后再取出当前指针指向的元素(it.previous();)

next(); 先取出当前指针指向的元素，然后指针向下移动一个单位(it.next();)

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

add(E e) 把当前有元素插入到当前指针指向的位置上

set(E e) 替换迭代器最后一次返回的元素

\*/

public class Demo3 {

public static void main(String[] args) {

List list = new ArrayList();

list.add("狗娃");

list.add("狗剩");

list.add("铁蛋");

list.add("美美");

ListIterator it = list.listIterator(); //返回的是一个List接口中特有的迭代器

/\*

System.out.println("有上一个元素吗？"+ it.hasPrevious());

System.out.println("获取上一个元素："+it.previous());

it.next();

System.out.println("获取上一个元素："+ it.previous());

while(it.hasNext()){

it.next();

}

while(it.hasPrevious()){

System.out.println("元素："+ it.previous());

}

it.next();

it.next();

it.add("张三");

\*/

it.next();

it.next();

it.set("张三");

System.out.println("集合的元素："+ list);

}

}

Demo07

package cn.itcast.list;

import java.util.List;

import java.util.ArrayList;

import java.util.ListIterator;

/\*

练习： 使用三种方式遍历集合的元素.

第一种： 使用get方法遍历。

第二种： 使用迭代器正序遍历。

第三种： 使用迭代器逆序遍历。

\*/

public class Demo4 {

public static void main(String[] args) {

List list = new ArrayList();

list.add("张三");

list.add("李四");

list.add("王五");

System.out.println("======get方法遍历=======");

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

System.out.print(list.get(i)+",");

}

System.out.println("\r\n======使用迭代器正序遍历==========");

ListIterator it = list.listIterator(); //获取到迭代器

while(it.hasNext()){

System.out.print(it.next()+",");

}

//只适合ArrayList

//由于上面的正序遍历，指针已经在（集合最后的位置+1）

System.out.println("\r\n======使用迭代器逆序遍历==========");

while(it.hasPrevious()){

System.out.print(it.previous()+",");

}

}

}

Demo08

package cn.itcast.list;

import java.util.ArrayList;

import java.util.List;

import java.util.ListIterator;

/\*

迭代器在变量元素的时候要注意事项： 在迭代器迭代（遍历）元素的过程中，不允许使用集合对象改变集合中的元素个数，如果需要添加或者删除只能使用迭代器的(add与remove)方法进行操作。

如果使用过了集合对象改变集合中元素个数那么就会出现ConcurrentModificationException异常。

迭代元素的过程中: 迭代器创建到使用结束的时间。

\*/

public class Demo5 {

public static void main(String[] args) {

List list = new ArrayList();

list.add("张三");

list.add("李四");

list.add("王五");

ListIterator it = list.listIterator(); //获取到迭代器

/\*

while(it.hasNext()){

System.out.print(it.next()+",");

it.add("aa"); // 把元素添加到当前指针指向位置

list.add("aa"); // add方法是把元素添加到集合的末尾处的。--->会报错。

list.remove("张三");//--->会报错，因为改变了元素个数

list.set(1,”aa”);//--->只是替换了李四，没有改变个数，不会报错

}

\*/

list.add("aa");

it.next();

System.out.println("\r\n集合的元素："+ list);

}

}

Demo09

package cn.itcast.list;

import java.util.ArrayList;

/\*

集合的体系：

----------| Collection 单列集合的根接口

--------------| List 如果实现了List接口的集合类，具备的特点： 有序，可重复

------------------------| ArrayList ArrayList 底层是维护了一个Object数组实现的;特点: 查询速度快，增删慢

什么时候使用ArrayList: 如果目前的数据是查询比较多，增删比较少的时候，那么就使用ArrayList存储这批数据 比如 ：高校的图书馆

------------------------| LinkedList

------------------------| Vector(了解即可)

--------------| Set 如果实现了Set接口的集合类， 具备的特点： 无序，不可重复。

ArrayList 特有的方法：

ensureCapacity(int minCapacity)

trimToSize()

笔试题目： 使用ArrayList无参的构造函数创建一个对象时， 默认的容量是多少? 如果长度不够使用时又自增增长多少？

ArrayList底层是维护了一个Object数组实现 的，使用无参构造函数时，Object数组默认的容量是10，当长度不够时，自动增长0.5倍。

\*/

public class Demo6 {

public static void main(String[] args) {

//创建ArrayList集合对象

ArrayList list = new ArrayList();

}

}

Demo10

package cn.itcast.list;

import java.util.ArrayList;

import java.util.Iterator;

class Book{

int id;

String name;// 名字

public Book(int id, String name) {

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "{ 书号："+ this.id+" 书名："+ this.name+" }";

}

@Override

public boolean equals(Object obj) {

Book book =(Book)obj;

return this.id==book.id;

}

}

// 需求： 编写一个函数清除集合中重复元素。 如果书号是一样就视为重复元素。 要求： 遍历集合元素的时候必须使用迭代器。 get 迭代器

public class Demo7 {

public static void main(String[] args) {

ArrayList list= new ArrayList();

list.add(new Book(110,"java编程思想"));

list.add(new Book(220,"java核心技术"));

list.add(new Book(330,"深入javaweb"));

list.add(new Book(110,"javas神书"));

ArrayList list2 = clearRepeat(list);

System.out.println("新集合的元素是："+ list2);

}

public static ArrayList clearRepeat(ArrayList list){

//创建一个新的集合

ArrayList newList = new ArrayList();

//获取迭代器

Iterator it = list.iterator();

while(it.hasNext()){

Book book = (Book) it.next(); //从旧集合中获取的元素

if(!newList.contains(book)){

//如果新集合没有包含该书籍，那么就存储到新集合中

newList.add(book);

}

}

return newList;

}

}

Demo11

package cn.itcast.list;

import java.util.Iterator;

import java.util.LinkedList;

/\*

集合的体系:

----------| Collection 单列集合的根接口

----------------| List 如果实现了List接口的集合类，具备的特点： 有序，可重复。

--------------------| ArrayList ArrayList 底层是维护了一个Object数组实现 的， 特点: 查询速度快，增删慢。

什么时候使用ArrayList: 如果目前的数据是查询比较多，增删比较少的时候，那么就使用ArrayList存储这批数据。 比如 ：高校的图书馆

--------------------| LinkedList LinkedList底层是使用了链表数据结构实现的，特点：查询速度慢，增删快。

--------------------| Vector(了解即可)

----------------| Set 如果实现了Set接口的集合类， 具备的特点： 无序，不可重复。

Linkedlist特有的方法：

1：方法介绍

addFirst(E e)

addLast(E e)

getFirst()

getLast()

removeFirst()

removeLast()

2：数据结构

1：栈 （1.6） : 主要是用于实现堆栈数据结构的存储方式。

先进后出

push()

pop()

2：队列（双端队列1.5）： 主要是为了让你们可以使用LinkedList模拟队列数据结构的存储方式。

先进先出

offer()

poll()

3：返回逆序的迭代器对象

descendingIterator() 返回逆序的迭代器对象

\*/

public class Demo8 {

public static void main(String[] args) {

LinkedList list= new LinkedList();

list.add("张三");

list.add("李四");

list.add("王五");

/\*

list.addFirst("狗娃"); //把元素添加到集合的首位置上。

list.addLast("狗剩"); //把元素添加到集合的末尾处。

System.out.println("获取集合中首位置的元素:"+list.getFirst());

System.out.println("获取集合中末尾的元素："+ list.getLast());

System.out.println("删除集合中的首位置元素并返回删除的元素："+ list.removeFirst());

System.out.println("删除集合中的末尾素并返回删除的元素："+ list.removeLast());

list.push("狗娃"); //将该元素插入此集合的开头处。

System.out.println("删除集合的首元素："+list.pop()); // 移除并返回集合中的第一个元素

list.offer("狗剩");

System.out.println("删除集合的首元素: "+list.poll());

System.out.println("集合中的元素："+ list);

\*/

Iterator it = list.descendingIterator();//逆序迭代器

while(it.hasNext()){

System.out.println(it.next());

}

}

}

Demo12

package cn.itcast.list;

import java.util.LinkedList;

/\*

1：栈 （1.6） : 主要是用于实现堆栈数据结构的存储方式。

先进后出

push()

pop()

2：队列（双端队列1.5）： 主要是为了让你们可以使用LinkedList模拟队列数据结构的存储方式。

先进先出

offer()

poll()

机试题目： 使用LinkedList实现堆栈数据结构的存储方式与队列的数据结构存储方式。

\*/

// 使用LinkedList模拟堆栈的数据结构存储方式

class StackList{

LinkedList list;

public StackList(){

list = new LinkedList();

}

//进栈

public void add(Object o){

list.push(o);

}

//弹栈 : 把元素删除并返回。

public Object pop(){

return list.pop();

}

//获取元素个数

public int size(){

return list.size();

}

}

//使用LinkedList模拟队列的存储方式

class TeamList{

LinkedList list;

public TeamList(){

list = new LinkedList();

}

public void add(Object o){

list.offer(o);

}

public Object remove(){

return list.poll();

}

//获取元素个数

public int size(){

return list.size();

}

}

public class Demo9 {

public static void main(String[] args) {

TeamList list= new TeamList();

list.add("李嘉诚");

list.add("马云");

list.add("王健林");

int size = list.size();//声明size变量，使size固定

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

System.out.println(list.remove());

}

}

}

Demo13

package cn.itcast.list;

import java.util.LinkedList;

/\*

需求： 使用LinkedList存储一副扑克牌，然后实现洗牌功能。

52张

一张扑克牌： 花色 点数。 梅花6

class Poker{

String color;

String num;

}

String[] colors = {"黑桃","红桃","梅花","方块"};

String[] nums = {"A","2"}

\*/

public class Demo10 {

public static void main(String[] args) {

LinkedList pokers = new LinkedList();

}

}

Day16//生成扑克牌//Vector//HashSet//TreeSet//泛型//

Demo01

package cn.itcast.list;

import java.util.LinkedList;

import java.util.Random;

/\*

需求： 使用LinkedList存储一副扑克牌，然后实现洗牌功能。

\*/

//扑克类

class Poker{

String color; //花色

String num; //点数

public Poker(String color, String num) {

this.color = color;

this.num = num;

}

@Override

public String toString() {

return "{"+color+num+"}";

}

}

public class Demo2 {

public static void main(String[] args) {

/\*

\* 这样创建扑克牌对象太过于繁琐！不采用！

\*LinkedList<Poker> pokers=new LinkedList<Poker>();

pokers.add(new Poker("红桃", "A"));

pokers.add(new Poker("红桃", "2"));\*/

//采取调用方法来创建对象

LinkedList pokers = createPoker();

shufflePoker(pokers);//调用洗牌

showPoker(pokers);//展示扑克牌

}

//洗牌的功能

public static void shufflePoker(LinkedList pokers){

//创建随机数对象

Random random = new Random();

for(int i = 0 ; i <100; i++){ //交换两张牌洗100次

//随机产生两个索引值

int index1 = random.nextInt(pokers.size());

int index2 = random.nextInt(pokers.size());

//根据索引值取出两张牌，然后交换两张牌的顺序

Poker poker1 = (Poker) pokers.get(index1);

Poker poker2 = (Poker) pokers.get(index2);

pokers.set(index1, poker2);

pokers.set(index2, poker1);

}

}

//显示扑克牌

public static void showPoker(LinkedList pokers){

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

System.out.print(pokers.get(i));

//换行

if(i%10==9){

System.out.println();

}

}

}

//生成扑克牌的方法

public static LinkedList createPoker(){

//该集合用于存储扑克对象

LinkedList list = new LinkedList();

//定义数组存储所有的花色与点数

String[] colors = {"黑桃","红桃","梅花","方块"};

String[] nums = {"A","2","3","4","5","6","7","8","9","10","J","Q","K"};

for(int i = 0 ; i < nums.length ; i++){

for(int j = 0 ; j<colors.length ; j++){

list.add(new Poker(colors[j], nums[i]));

}

}

return list;

}

}

Demo02

package cn.itcast.list;

import java.util.Enumeration;

import java.util.Vector;

/\*

集合 的体系：

------------| Collection 单例集合的根接口

----------------| List 如果是实现了List接口的集合类，具备的特点： 有序，可重复。

-------------------| ArrayList ArrayList 底层是维护了一个Object数组实现的。 特点： 查询速度快，增删慢。

-------------------| LinkedList LinkedList 底层是使用了链表数据结构实现的， 特点： 查询速度慢，增删快。

-------------------| Vector(了解即可) 底层也是维护了一个Object的数组实现的，实现与ArrayList是一样的，但是Vector是线程安全的，操作效率低。

----------------| Set 如果是实现了Set接口的集合类，具备的特点： 无序，不可重复。

笔试题: 说出ArrayLsit与Vector的区别?

相同点： ArrayList与Vector底层都是使用了Object数组实现的。

不同点：

1. ArrayList是线程不同步的，操作效率高。

Vector是线程同步的，操作效率低。

2. ArrayList是JDK1.2出现，Vector是jdk1.0的时候出现的。

\*/

public class Demo1 {

public static void main(String[] args) {

Vector v = new Vector();

//添加元素

v.addElement("张三");

v.addElement("李四");

v.addElement("王五");

//迭代该集合

Enumeration e = v.elements(); //获取迭代器

while(e.hasMoreElements()){

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

}

}

}

Demo03

//第一种写法

package cn.itcast.list;

import java.util.LinkedList;

class Person{

String name;

int age;

public Person(String name, int age) {

super();

this.name = name;

this.age = age;

}

@Override

public String toString() {

return "{ 名字："+ this.name+" 年龄："+ this.age+"}";

}

}

public class Demo3 {

public static void main(String[] args) {

LinkedList list = new LinkedList();

list.add(new Person("狗娃", 7));

list.add(new Person("狗剩", 17));

list.add(new Person("铁蛋", 3));

list.add(new Person("美美", 30));

//编写一个函数根据人的年龄及逆行排序存储。

for(int i= 0 ; i<list.size() -1 ; i++){

for(int j = i+1 ; j<list.size() ; j++){

//符合条件交换位置

Person p1 = (Person) list.get(i);

Person p2 = (Person) list.get(j);

if(p1.age>p2.age){

//交换位置

list.set(i, p2);

list.set(j, p1);

}

}

}

System.out.println(list);

}

}

//第二种写法

package cn.whstu02;

import java.util.LinkedList;

class Person{

String name;

int age;

//构造器

public Person(String name,int age){

this.name=name;

this.age=age;

}

@Override

public String toString() {

return "{名字："+name+"年龄："+age+"}";

}

}

public class Demo02 {

public static void main(String[] args) {

//创建persons放到linkedlist中

LinkedList<Person> persons=createPerson();

//根据年龄排序的方法

sortAge(persons);

}

private static void sortAge(LinkedList<Person> persons) {

for (int i = 0; i < persons.size()-1; i++) {

for (int j = i+1; j < persons.size(); j++) {

//根据索引值获得person

Person p1=persons.get(i);

Person p2=persons.get(j);

if(p1.age>p2.age){

//交换位置

persons.set(i, p2);

persons.set(j, p1);

}

}

}

System.out.println(persons);

}

private static LinkedList<Person> createPerson() {

LinkedList<Person> list=new LinkedList<Person>();

list.add(new Person("张三", 25));

list.add(new Person("李四", 23));

list.add(new Person("王五", 24));

return list;

}

}

Demo04（泛型）

package cn.itcast.genrictiry;

import java.util.ArrayList;

/\*

泛型是jdk1.5使用的新特性。

泛型的好处：

1. 将运行时的异常提前至了编译时。

2. 避免了无谓的强制类型转换。

泛型在集合中的常见应用：

ArrayList<String> list = new ArrayList<String>(); true 推荐使用。

ArrayList<Object> list = new ArrayList<String>(); false

ArrayList<String> list = new ArrayList<Object>(); false

// 以下两种写法主要是为了兼顾新老系统的兼用性问题。

\*

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

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

注意：泛型没有多态的概念，左右两边的数据类型必须要一致或者只是写一边的泛型类型。

推荐使用： 两边都写泛型。

需求： 把一个集合中元素全部转成大写。

\*/

public class Demo1 {

public static void main(String[] args) {

ArrayList<String> list = new ArrayList<String>(); //<String> 表示该容器只能存储字符串类型的数据。

list.add("aa");

list.add("bb");

list.add("cc");

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

String str = list.get(i);

System.out.println("大写："+ str.toUpperCase());

}

MyUtil.print(list);

ArrayList<String> list2 = MyUtil.getList();//接下面的MyUtil讲解泛型的兼容性

}

}

Demo05（泛型）

package cn.itcast.genrictiry;

/\*

需求： 定义一个方法可以接收任意类型的参数，而且返回值类型必须要与实参的类型一致。

自定义泛型：自定义泛型就是一个数据类型的占位符或者是一个数据类型的变量。

方法上自定义泛型：

修饰符 <声明自定义的泛型> 返回值类型 函数名(使用自定义泛型 ...) {

}

在泛型中不能使用基本数据类型，如果需要使用基本数据类型，那么就使用基本数据类型对应的包装类型。

byte----> Byte

short---> Short

int----> Integer

long----> Long

double ----> Double

float -----> Float

boolean----->Boolean

char-------> Character

方法泛型注意的事项：

1. 在方法上自定义泛型，这个自定义泛型的具体数据类型是在调用该方法的时候传入实参时确定具体的数据类型的。

2. 自定义泛型只要符合标识符的命名规则即可, 但是自定义泛型我们一般都习惯使用一个大写字母表示。 T Type E Element

\*/

public class Demo2 {

public static void main(String[] args) {

String str = getData("abc");//输出abc

Sysout(str);

Integer i = getData(123);//输出：123

Sysout(123);

}

//方法上自定义泛型

public static <abc>abc getData(abc o){

return o;

}

}

Demo06（泛型）

package cn.itcast.genrictiry;

import java.util.ArrayList;

/\*

需求： 编写一个数组的工具类

泛型类：

泛型类的定义格式：

class 类名<声明自定义泛型>{

}

泛型类要注意的事项：

1. 在类上自定义泛型的具体数据类型是在使用该类的时候创建对象时候确定的。

2. 如果一个类在类上已经声明了自定义泛型，如果使用该类创建对象 的时候没有指定 泛型的具体数据类型，默认为Object类型

3.在类上自定义泛型不能作用于静态的方法；如果静态的方法需要使用自定义泛型，那么需要在方法上自己声明使用。

\*/

class MyArrays<T>{

//使用泛型类

//元素翻转

public void reverse(T[] arr){

for(int startIndex = 0, endIndex = arr.length-1 ; startIndex<endIndex ; startIndex++,endIndex--){

T temp = arr[startIndex];

arr[startIndex] = arr[endIndex];

arr[endIndex] = temp;

}

}

public String toString(T[] arr){

StringBuilder sb = new StringBuilder();

for(int i = 0 ; i < arr.length ; i++){

if(i==0){

sb.append("["+arr[i]+",");

}else if(i==arr.length-1){

sb.append(arr[i]+"]");

}else{

sb.append(arr[i]+",");

}

}

return sb.toString();

}

public static <T>void print(T[] t){ //上述第3点，静态使用则再次自定义泛型<T>

}

}

public class Demo3 {

public static void main(String[] args) {

Integer[] arr = {10,12,14,19};//int类型

MyArrays<Integer> tool = new MyArrays<Integer>();

tool.reverse(arr);

System.out.println("数组的元素："+tool.toString(arr));

MyArrays<String> tool2 = new MyArrays<String>();//string

String[] arr2 = {"aaa","bbb","ccc"};

tool2.reverse(arr2);

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

}

}

Demo07（泛型）

package cn.itcast.genrictiry;

/\*

泛型接口

泛型接口的定义格式:

interface 接口名<声明自定义泛型>{

}

泛型接口要注意的事项：

1. 接口上自定义的泛型的具体数据类型是在实现一个接口的时候指定的。

2. 在接口上自定义的泛型如果在实现接口的时候没有指定具体的数据类型，那么默认为Object类型。

需求： 目前我实现一个接口的时候，我还不明确我目前要操作的数据类型，我要等待创建接口实现类 对象的时候我才能指定泛型的具体数据类型。

如果要延长接口自定义泛型的具体数据类型，那么格式如下：

格式：

public class Demo4<T> implements Dao<T>{

}

\*/

interface Dao<T>{

public void add(T t);

}

public class Demo4<T> implements Dao<T> {

public static void main(String[] args) {

Demo4<String> d = new Demo4<String>();

}

public void add(T t){

}

}

Demo08（泛型）

package cn.itcast.genrictiry;

import java.util.ArrayList;

import java.util.Collection;

import java.util.HashSet;

/\*

泛型的上下限：

需求1： 定义一个函数可以接收接收任意类型的集合对象， 要求接收的集合对象只能存储Integer或者是Integer的父类类型数据。

需求2： 定义一个函数可以接收接收任意类型的集合对象， 要求接收的集合对象只能存储Number或者是Number的子类类型数据。

泛型中通配符： ？

? super Integer : 只能存储Integer或者是Integer父类元素。 泛型下限

? extends Number ： 只能存储Number或者是Number类型的子类数据。 泛型上限

\*/

public class Demo5 {

public static void main(String[] args) {

ArrayList<Integer> list1 = new ArrayList<Integer>();

ArrayList<Number> list2 = new ArrayList<Number>();

HashSet<String> set = new HashSet<String>();

//getData(set);

}

//泛型的上限

public static void getData(Collection<? extends Number> c){

}

//泛型的下限

public static void print(Collection<? super Integer> c){

}

}

MyUtil（讲解泛型的兼容问题，接上面的demo04）

package cn.itcast.genrictiry;

import java.util.ArrayList;

//是老程序员写 的。 jdk1.4的时候写的。

public class MyUtil {

public static ArrayList getList(){

return new ArrayList();

}

public static void print(ArrayList list){

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

System.out.println(list.get(i));

}

}

}

Demo10

package cn.itcastset;

import java.util.HashSet;

import java.util.Set;

/\*

集合 的体系：

------------| Collection 单例集合的根接口

----------------| List 如果是实现了List接口的集合类，具备的特点： 有序，可重复。

-------------------| ArrayList ArrayList 底层是维护了一个Object数组实现的。 特点： 查询速度快，增删慢。

-------------------| LinkedList LinkedList 底层是使用了链表数据结构实现的， 特点： 查询速度慢，增删快。

-------------------| Vector(了解即可) 底层也是维护了一个Object的数组实现的，实现与ArrayList是一样的，但是Vector是线程安全的，操作效率低。

----------------| Set 如果是实现了Set接口的集合类，具备的特点： 无序，不可重复。

无序： 添加元素的顺序与元素出来的顺序是不一致的。

\*/

public class Demo1 {

public static void main(String[] args) {

Set set = new HashSet();

set.add("王五");

set.add("张三");

set.add("李四");

System.out.println("添加成功吗？"+set.add("李四")); //会显示false

System.out.println(set);

}

}

Demo11

package cn.itcastset;

import java.util.HashSet;

import javax.print.attribute.HashAttributeSet;

/\*

集合 的体系：

------------| Collection 单例集合的根接口

----------------| List 如果是实现了List接口的集合类，具备的特点： 有序，可重复。

---------------------| ArrayList ArrayList 底层是维护了一个Object数组实现的。 特点： 查询速度快，增删慢。

---------------------| LinkedList LinkedList 底层是使用了链表数据结构实现的， 特点： 查询速度慢，增删快。

---------------------| Vector(了解即可) 底层也是维护了一个Object的数组实现的，实现与ArrayList是一样的，但是Vector是线程安全的，操作效率低。

----------------| Set 如果是实现了Set接口的集合类，具备的特点： 无序，不可重复。

---------------------| HashSet 底层是使用了哈希表来支持的，特点： 存取速度快.

hashSet的实现原理：

往Hashset添加元素的时候，HashSet会先调用元素的hashCode方法得到元素的哈希值 ，

然后通过元素 的哈希值经过移位等运算，就可以算出该元素在哈希表中 的存储位置。

情况1： 如果算出元素存储的位置目前没有任何元素存储，那么该元素可以直接存储到该位置上。

情况2： 如果算出该元素的存储位置目前已经存在有其他的元素了，那么会调用该元素的equals方法与该位置的元素再比较一次，如果equals返回的是true，那么该元素与这个位置上的元素就视为重复元素，不允许添加；如果equals方法返回的是false，那么该元素可以添加。

---------------------| TreeSet

\*/

class Person{

int id;

String name;

public Person(int id, String name) {

super();

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "{ 编号:"+ this.id+" 姓名："+ this.name+"}";

}

//使得id相同的person添加不了！

@Override

public int hashCode() {

System.out.println("=======hashCode=====");

return this.id; //重写hashCode方法返回去比较id是否一样

}

@Override //并且也要重写equal方法，才能不添加重复id的对象

public boolean equals(Object obj) {

System.out.println("======equals======");

Person p = (Person)obj;

return this.id==p.id;

}

}

public class Demo2 {

public static void main(String[] args) {

/\*

HashSet set = new HashSet();

set.add("狗娃");

set.add("狗剩");

set.add("铁蛋");

System.out.println("集合的元素："+ set);

\*/

HashSet set = new HashSet();

set.add(new Person(110,"狗娃"));

set.add(new Person(220,"狗剩"));

set.add(new Person(330,"铁蛋"));

//在现实生活中只要编号一致就为同一个人.

System.out.println("添加成功吗？"+set.add(new Person(110,"狗娃"))); //显示false

System.out.println("集合的元素："+set);

}

}

Demo12

package cn.itcastset;

import java.util.HashSet;

import java.util.Scanner;

/\*

需求： 接受键盘录入用户名与密码，如果用户名与密码已经存在集合中，那么就是视为重复元素，不允许添加到HashSet中。

\*/

class User{

String userName;

String password;

public User(String userName, String password) {

super();

this.userName = userName;

this.password = password;

}

@Override

public String toString() {

return "{ 用户名："+this.userName+" 密码："+ this.password+"}";

}

@Override

public boolean equals(Object obj) {

User user = (User)obj;

//比较已经存在的user

return this.userName.equals(user.userName)&&this.password.equals(user.password);

}

@Override

public int hashCode() { //int类型,转为hashcode

return this.userName.hashCode()+this.password.hashCode();

}

}

public class Demo3 {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

HashSet set = new HashSet();

while(true){

System.out.println("请输入用户名:");

String userName = scanner.next();

System.out.println("请输入密码：");

String password = scanner.next();

//创建一个对象

User user = new User(userName, password);

if(set.add(user)){ //set方法有返回值（为true）执行下面代码

System.out.println("注册成功...");

System.out.println("当前的用户有："+ set);

}else{ //返回值为false

System.out.println("注册失败...");

}

}

}

}

Demo13

package cn.itcastset;

public class Demo4 {

public static void main(String[] args) {

String str1 = "hello";

String str2 = new String("hello");

System.out.println("两个是同一个对象吗？"+(str1==str2));//false

System.out.println("str1的hashCode："+ str1.hashCode());//99162322

System.out.println("str2的hashCode:"+ str2.hashCode());//同上

/\*

HashCode默认情况下表示的是内存地址，String 类已经重写了Object的hashCode方法了。

注意： 如果两个字符串的内容一致，那么返回的hashCode 码也一致。

\*/

}

}

Demo14（TreeSet）

package cn.itcastset;

import java.util.TreeSet;

/\*

集合 的体系：

------------| Collection 单例集合的根接口

----------------| List 如果是实现了List接口的集合类，具备的特点： 有序，可重复。

-------------------| ArrayList ArrayList 底层是维护了一个Object数组实现的。 特点： 查询速度快，增删慢。

-------------------| LinkedList LinkedList 底层是使用了链表数据结构实现的， 特点： 查询速度慢，增删快。

-------------------| Vector(了解即可) 底层也是维护了一个Object的数组实现的，实现与ArrayList是一样的，但是Vector是线程安全的，操作效率低。

----------------| Set 如果是实现了Set接口的集合类，具备的特点： 无序，不可重复。

-------------------| HashSet 底层是使用了哈希表来支持的，特点： 存取速度快.

hashSet的实现原理：

往Haset添加元素的时候，HashSet会先调用元素的hashCode方法得到元素的哈希值 ，

然后通过元素 的哈希值经过移位等运算，就可以算出该元素在哈希表中 的存储位置。

情况1： 如果算出元素存储的位置目前没有任何元素存储，那么该元素可以直接存储到该位置上。

情况2： 如果算出该元素的存储位置目前已经存在有其他的元素了，那么会调用该元素的equals方法与该位置的元素再比较一次

，如果equals返回的是true，那么该元素与这个位置上的元素就视为重复元素，不允许添加，如果equals方法返回的是false，那么该元素运行 添加。

-------------------| TreeSet 如果元素具备自然顺序的特性，那么就按照元素自然顺序的特性进行排序存储。

\*/

public class Demo5 {

public static void main(String[] args) {

TreeSet tree = new TreeSet();

/\* tree.add(1);

tree.add(10);

tree.add(7);

tree.add(19);

tree.add(9);\*/

tree.add('b');

tree.add('f');

tree.add('a');

tree.add('c');

System.out.println(tree);// 输出时可以自动排序

}

}

Demo15（TreeSet）

package cn.itcastset;

import java.util.Comparator;

import java.util.TreeSet;

/\*

treeSet添加自定义元素:

TreeSet要注意的事项：

1. 往TreeSet添加元素的时候，如果元素本身具备了自然顺序的特性，那么就按照元素自然顺序的特性进行排序存储。

2. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，那么该元素所属的类必须要实现Comparable接口，把元素的比较规则定义在compareTo(T o)方法上。返回正整数,负整数或零。

3. 如果比较元素的时候，compareTo方法返回的是0，那么该元素就被视为重复元素，不允许添加.(注意：TreeSet与HashCode、equals方法是没有任何关系！)

4. 往TreeSet添加元素的时候, 如果元素本身没有具备自然顺序的特性，而元素所属的类也没有实现Comparable接口，那么必须要在创建TreeSet的时候传入一个比较器。

5. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，而元素所属的类已经实现了Comparable接口，在创建TreeSet对象的时候也传入了比较器那么是以比较器的比较规则优先使用。

如何自定义定义比较器： 自定义一个类实现Comparator接口即可，把元素与元素之间的比较规则定义在compare方法内即可。

自定义比较器的格式 ：

class 类名 implements Comparator{

}

推荐使用：使用比较器(Comparator)。

\*/

class Emp implements Comparable<Emp>{

int id;

String name;

int salary;

public Emp(int id, String name, int salary) {

super();

this.id = id;

this.name = name;

this.salary = salary;

}

@Override

public String toString() {

return "{ 编号："+ this.id+" 姓名："+ this.name+" 薪水："+ this.salary+"}";

}

//@Override //元素与元素之间的比较规则。

// 返回值负整数、零或正整数，根据此对象是小于、等于还是大于指定对象。

public int compareTo(Emp o) {

//System.out.println(this.name+"compare"+ e.name);//查看如何比较

return this.salary- o.salary;

}

}

//自定义一个比较器

class MyComparator implements Comparator<Emp>{

@Override

public int compare(Emp o1, Emp o2) {

return o1.id-o2.id;

}

//根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数。

/\*@Override

public int compare(Object o1, Object o2) {

Emp e1 = (Emp) o1;

Emp e2 = (Emp) o2;

return e1.id - e2.id; //根据返回值判断

}\*/

}

public class Demo6 {

public static void main(String[] args) {

//创建一个比较器对象

MyComparator comparator = new MyComparator();

//创建TreeSet的时候传入比较器

TreeSet tree = new TreeSet(comparator);

tree.add(new Emp(110, "老陆", 100));

tree.add(new Emp(113, "老钟", 200));

tree.add(new Emp(220, "老汤", 300));

tree.add(new Emp(120, "老蔡", 500));

System.out.println("集合的元素："+tree);

}

}

Demo16（TreeSet）

package cn.itcastset;

import java.util.TreeSet;

/\*

TreeSet是可以对字符串进行排序的， 因为字符串已经实现了Comparable接口。

字符串的比较规则：

情况一： 对应位置有不同的字符出现， 就比较的就是对应位置不同的字符。

情况 二：对应位置上的字符都一样，比较的就是字符串的长度。

\*/

public class Demo7 {

public static void main(String[] args) {

/\*TreeSet tree = new TreeSet();

tree.add("abcccccccccccccccccc");

tree.add("abc");

System.out.println(tree);\*/

System.out.println("abw".compareTo("abcccccccccccc"));

}

}

Demo17（TreeSet）

package cn.itcastset;

import java.util.Iterator;

import java.util.TreeSet;

/\*

需求：将字符串中的数值进行排序。

例如String str="8 10 15 5 2 7"; ----> "2 5 7 8 10 15"

\*/

public class Demo8 {

public static void main(String[] args) {

String str="8 10 15 5 2 7";

String[] datas = str.split(" ");

TreeSet tree = new TreeSet();

for(int i = 0 ; i<datas.length ; i++){

tree.add(Integer.parseInt( datas[i])); //字符串转int类型要使用Integer.parseInt()方法

}

//遍历treeSet的元素拼接成对应的字符串，遍历Set集合对象都得要用迭代器

Iterator it = tree.iterator();

while(it.hasNext()){

System.out.print(it.next()+" ");

}

}

}

Day17//Map接口//HashMap,TreeMap//Collections与Arrays//正则表达式//Map接口的迭代//

Demo01

package cn.itcast.map;

/\*

单例集合 的体系：

---------| Collection 单例集合的根接口

------------| List 如果是实现了List接口的集合类， 具备的特点：有序，重复。

---------------| ArraryList 底层 是使用了Object数组实现的，特点： 查询速度快，增删慢。

---------------| LinkedList 底层是使用了链表数据结构实现 的， 特点： 查询速度慢，增删快。

---------------| Vector Vector的实现与ArrayList是一致，但是是线程安全 的，操作效率低。 jdk1.0的时候出现的

------------| Set 如果是实现了Set接口的集合类，具备的特点：无序，不可重复。

----------------| HashSet 底层是使用了一个哈希表支持的， 特点：存取速度快。

HashSet添加元素的原理：

往HashSet添加元素的时候，首先HashSet会调用元素的hashCOde方法得到元素的哈希码值，然后会经过一系列运算

就可以算出该元素在哈希表中的存储位置/

情况1：如果算出该元素的位置目前没有任何元素存储，那么该元素可以直接存储

情况2： 如果算出该元素的位置目前已经存有其他的元素，那么还会调用元素的equals方法与该位置的元素再比较一次。

如果equals方法返回的是false，那么该元素允许存储，如果euqlas方法返回的是true，那么该元素被视为重复元素，不允许存储。

------------------| TreeSet 底层是使用了红黑树（二叉树）数据结构实现的， 特点：会对元素进行排序存储。

TreeSet要注意的事项：

1. 往TreeSet添加元素的时候，如果元素本身具备自然顺序的特性，那么会根据元素自然顺序的特性进行排序存储。

2. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，那么元素所属的类必须要实现Comparable接口，把元素的比较规则定义在CompareTo方法上。

3. 往TreeSet添加元素的时候，如果元素本身不具备自然顺序的特性，而且元素所属的类没有实现COmparable接口，那么必须要在创建TreeSet对象的时候传入比较器。

4. 如果比较的方法(CompareTo 或者Compare )返回的是0的时候，那么该元素就被视为重复元素，不允许添加。

比较器的定义格式： 自定义一个类实现Comparator接口即可。

class 类名 implements Comparator{

}

泛型：泛型是jdk1.5出现的新特性。

泛型的好处：

1. 将运行时出现 的问题提前至了编译时。

2. 避免了无谓强制类型转换。

自定义泛型： 自定义泛型就是一个数据类型的占位符或者理解为一个数据类型的变量。

泛型方法:

修饰符 <声明自定义的泛型>返回值类型 函数名(自定义的泛型 变量名..)

泛型方法要注意的事项：

1. 泛型方法中 的自定义泛型的具体数据类型是在调用该函数的时候传入实参时确定的。

2. 自定义泛型所用 的标识符只要符合标识符 的命名规则即可。但是我们一般都习惯使用一个大写字母表示。

泛型类：

泛型类的定义格式

class 类名<声明自定义的泛型>{

}

泛型类要注意的事项：

1. 泛型类上的自定义泛型是在使用该类创建对象的时候指定具体的数据类型的。

2. 如果一个类已经自定义了泛型，使用该类创建对象的时候如果没有指定泛型的具体数据类型，那么默认为Object类型。

3. 静态的函数不能使用类上自定义的泛型，如果静态函数需要使用，必须要在函数上自定义泛型。

泛型接口：

泛型接口的定义格式：

interface 接口名<声明自定义的泛型>{

}

泛型接口要注意事项：

1. 泛型接口上的自定义泛型是在实现该接口的时候指定具体数据类型的。

2. 如果实现接口的时候没有指定接口上 的自定义泛型的具体数据类型，那么默认为Object数据类型。

3. 如果需要在创建接口实现类对象的时候才指定接口上自定义泛型，那么需要以下格式： class<T> 类名 implements 接口<T>

泛型上下限：

? super Integer 允许是Integer数据类型或者是Integer父类类型 泛型的下限

? extedns Number 允许是Number数据类型或者是Number子类的数据类型 泛型的上限。

\*/

Demo02

package cn.itcast.map;

import java.util.HashMap;

import java.util.Map;

/\*

在现实生活中有些数据是以映射关系存在的，也就是成对存在的，比如：

民政局 ：

键key 值velue

老公 老婆

身份证 人

一把要锁 锁

双列集合：

-----------| Map 如果是实现了Map接口的集合类，具备的特点： 存储的数据都是以键值对的形式存在的，键不可重复，值可以重复。

----------------| HashMap

----------------| TreeMap

----------------| Hashtable

Map接口的方法：

添加：

put(K key, V value)

putAll(Map<? extends K,? extends V> m)

删除

remove(Object key)

clear()

获取：

get(Object key)

size()

判断：

containsKey(Object key)

containsValue(Object value)

isEmpty()

\*/

public class Demo2 {

public static void main(String[] args) {

Map<String,String> map = new HashMap<String, String>();

//添加方法

map.put("汪峰", "章子怡");

map.put("文章", "马伊琍");

map.put("谢霆锋","张柏芝");

/\*

添加

System.out.println("返回值："+map.put("谢霆锋","黄菲")); // 返回值:张柏芝//如果之前没有存在该键，那么返回的是null，如果之前就已经存在该键了，那么就返回该键之前对应的值

Map<String,String> map2 = new HashMap<String, String>();

map2.put("杨振宁", "翁帆");

map2.put("习总", "彭丽媛");

map.putAll(map2); // 把map2的元素添加到map集合中。

\*/

/\*

删除

System.out.println("删除的数据是:"+map.remove("汪峰")) ; // 输出：章子怡//(只能)根据键删除一条map中的K，返回的是该键对应的V。

map.clear(); //清空集合中的所有数据。

\*/

/\* 获取

System.out.println("(只能)根据指定的键获取对应的值:"+ map.get("文章"));

System.out.println("获取map集合键值对个数："+map.size());

判断

System.out.println("判断map集合是否包含指定的键："+ map.containsKey("文章"));

System.out.println("判断map集合中是否包含指定的值："+ map.containsValue("张柏芝"));

map.clear();

System.out.println("判断map集合是否为空元素："+ map.isEmpty());

\*/

System.out.println("集合的元素："+ map);

}

}

Demo03

package cn.itcast.map;

import java.util.Collection;

import java.util.HashMap;

import java.util.Iterator;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Set;

/\*

迭代(遍历)：

keySet()

values()

entrySet()

\*/

public class Demo3 {

public static void main(String[] args) {

Map<String,String> map = new HashMap<String, String>();

//添加方法

map.put("汪峰", "章子怡");

map.put("文章", "马伊琍");

map.put("谢霆锋","张柏芝");

map.put("成龙", "林凤娇");

/\*

//map集合中遍历方式一： 使用keySet方法进行遍历//缺点： keySet方法只是返回了所有的键，没有值。

Set<String> keys = map.keySet(); //keySet() 把Map集合中的所有键都保存到一个Set类型 的集合对象中返回---->Set<String> keys=map.keySet();

Iterator<String> it = keys.iterator();

while(it.hasNext()){

String key = it.next();

//通过键获取值

System.out.println("键："+ key+" 值："+ map.get(key));

}

//map集合的遍历方式二： 使用values方法进行 遍历//缺点： values方法只能返回所有 的值，没有键。

Collection<String> c = map.values(); //values() 把所有的值存储到一个Collection集合中返回。

Iterator<String> it = c.iterator();

while(it.hasNext()){

System.out.println("值："+ it.next());

}

\*/

//map集合的遍历方式三： entrySet方法遍历。

Set<Map.Entry<String,String>> entrys = map.entrySet();

Iterator<Map.Entry<String,String>> it = entrys.iterator();//调用迭代器

while(it.hasNext()){

Map.Entry<String,String> en= it.next();

System.out.println("键："+ en.getKey()+" 值："+ en.getValue());

}

}

}

Demo04（解释上例为何使用Map.Entry<String,String>而不是Entry<String,String>）

package cn.itcast.map;

import java.util.ArrayList;

class Map{

//静态内部类

static class Entry<K ,V>{

K key;

V value;

}

}

public class Demo4 {

public static void main(String[] args) {

Map.Entry;

}

}

Demo05

package cn.itcast.map;

import java.util.HashMap;

/\*

双列集合：

-------------| Map 如果是实现了Map接口的集合类，具备的特点： 存储的数据都是以键值对的形式存在的，键不可重复，值可以重复。

----------------| HashMap 底层也是基于哈希表实现的。

HashMap的存储原理：

往HashMap添加元素的时候，首先会调用 键 的hashCode方法得到元素的哈希码值，然后经过运算就可以算出该元素在哈希表中的存储位置。

情况1：如果算出的位置目前没有任何元素存储，那么该元素可以直接添加到哈希表中。

情况2：如果算出 的位置目前已经存在其他的元素，那么还会调用该元素的equals方法与这个位置上的元素进行比较，如果equals方法返回 的是false，那么该元素允许被存储，如果equals方法返回的是true，那么该元素被视为重复元素，不允存储。

----------------| TreeMap

----------------| Hashtable

\*/

class Person{

int id;

String name;

public Person(int id, String name) {

super();

this.id = id;

this.name = name;

}

@Override

public String toString() {

return "[编号："+this.id+" 姓名："+ this.name+"]";

}

@Override

public int hashCode() {

return this.id;

}

@Override

public boolean equals(Object obj) {

Person p = (Person) obj;

return this.id== p.id;

}

}

public class Demo5 {

public static void main(String[] args) {

HashMap<Person, String> map = new HashMap<Person, String>();

map.put(new Person(110,"狗娃"), "001");

map.put(new Person(220,"狗剩"), "002");

map.put(new Person(330,"铁蛋"), "003");

map.put(new Person(110,"狗娃"), "007"); // （前提是已经重写了hashcode与equals方法）如果出现了相同的键，那么后添加的数据的值会取代之前的值。

System.out.println("集合的元素："+ map);

}

}

Demo06

package cn.itcast.map;

import java.util.Comparator;

import java.util.TreeMap;

/\*

双列集合：

-------------| Map 如果是实现了Map接口的集合类，具备的特点： 存储的数据都是以键值对的形式存在的，键不可重复，值可以重复。

----------------| HashMap 底层也是基于哈希表实现 的。

HashMap的存储原理：

往HashMap添加元素的时候，首先会调用键的hashCode方法得到元素 的哈希码值，然后经过运算就可以算出该元素在哈希表中的存储位置。

情况1： 如果算出的位置目前没有任何元素存储，那么该元素可以直接添加到哈希表中。

情况2：如果算出 的位置目前已经存在其他的元素，那么还会调用该元素的equals方法与这个位置上的元素进行比较，如果equals方法返回 的是false，那么该元素允许被存储，如果equals方法返回的是true，那么该元素被视为重复元素，不允存储。

----------------| TreeMap TreeMap也是基于红黑树（二叉树）数据结构实现 的， 特点：会对元素的键进行排序存储。

TreeMap 要注意的事项：

1. 往TreeMap添加元素的时候，如果元素的键具备自然顺序，那么就会按照键的自然顺序特性进行排序存储。

2. 往TreeMap添加元素的时候，如果元素的键不具备自然顺序特性， 那么键所属的类必须要实现Comparable接口，把键的比较规则定义在CompareTo方法上。

3. 往TreeMap添加元素的时候，如果元素的键不具备自然顺序特性，而且键所属的类也没有实现Comparable接口，那么就必须在创建TreeMap对象的时候传入比较器。

----------------| Hashtable (了解) jdk1.0 底层也是依赖哈希表实现的，实现方式与HashMap是一致的，但是HashTable是线程安全的，效率低。

\*/

class Emp {//implements Comparable<Emp>{

String name;

int salary;

public Emp(String name, int salary) {

super();

this.name = name;

this.salary = salary;

}

@Override

public String toString() {

return "[姓名："+this.name+" 薪水："+ this.salary+"]";

}

/\*

@Override

public int compareTo(Emp o) {

return this.salary - o.salary;

}\*/

}

//自定义一个比较器

class MyComparator implements Comparator<Emp>{

//按照salary排序

@Override

public int compare(Emp o1, Emp o2) {

return o1.salary - o2.salary;

}

}

public class Demo6 {

public static void main(String[] args) {

/\* TreeMap<Character, Integer> tree = new TreeMap<Character, Integer>();

tree.put('c',10);

tree.put('b',2);

tree.put('a',5);

tree.put('h',12);

System.out.println(tree);\*/

//创建一个自定义比较器对象

MyComparator comparator = new MyComparator();

//在创建TreeMap时传入比较器

TreeMap<Emp, String> tree = new TreeMap<Emp, String>(comparator);

tree.put(new Emp("冰冰", 2000),"001");

tree.put(new Emp("家宝", 1000),"002");

tree.put(new Emp("习总", 3000),"003");

tree.put(new Emp("克强", 5000),"005");

tree.put(new Emp("财厚", 5000),"008");

System.out.println(tree);// 财厚和克强salary一样，只是此时值008取代了005，输出的键emp还是上一个数据：克强,5000=008

}

}

Demo07

/\*

需求： 定义一个TreeMap，键存储的是书对象，值存储的是字符串。 根据书的出版出版日期排序。

\*/

import java.util.TreeMap;  
class Book implements Comparable<Book>{  
 String name;  
 String date;  
 public Book(String name,String date){  
 super();  
 this.name=name;  
 this.date=date;  
 }  
 @Override  
 public int compareTo(Book o) {  
 return this.date.compareTo(o.date);  
 }  
 @Override  
 public String toString() {  
 return "[书名："+this.name+"日期："+this.date+"]";  
 }  
}  
  
public class Demo05 {  
 public static void main(String[] args) {  
 TreeMap<Book,String> map=new TreeMap<Book,String>();  
 map.put(new Book("红楼梦","1965-1-2"),"01");  
 map.put(new Book("水浒传","1980-1-2"),"02");  
 map.put(new Book("西游记","1490-1-2"),"03");  
 map.put(new Book("三国","1198-1-2"),"04");  
 System.out.println(map);  
 }  
}

Demo08

集合的工具类(Collections)

Collection与Collections的区别：Collection是一个单例集合的根接口，Collections是操作集合对象的一个工具类。

package wh06;  
import java.util.ArrayList;  
import java.util.Collections;

/\*\*Collections:常见的方法  
 \*  
 \* 1，对list集合进行排序  
 \* sort(list);  
 \* sort(list,Comparator);  
 \*  
 \* 2，对list进行二分法查找  
 \* int binarySearch(list,key);  
 \* int binarySearch(list,key,Comparator);  
 \*  
 \* 3，对集合取最大值或者最小值  
 \* max(Collection)  
 \* max(Collection,Comparator)  
 \* min(Collection)  
 \* min(Collection,Comparator)  
 \* 4，对list集合的反转  
 \* reverse（list）;  
 \*  
 \* 5，可以将不同步的集合变成同步的集合  
 \* Set synchronizedSet(Set<T> s)  
 \* Map synchronizedMap(Map<K,V> m)  
 \* List synchronizedList(List<T> list)  
 \*  
 \*/

//具备自然顺序的元素  
public class Demo06 {  
 public static void main(String[] args) {  
 ArrayList<Integer> list=new ArrayList<Integer>();  
 list.add(12);  
 list.add(22);  
 list.add(34);  
 list.add(18);  
 //排序  
 Collections.sort(list);  
 //已经排序后的集合中  
 System.out.println("已经排序后集合元素34的索引值是："+Collections.binarySearch(list,34));  
  
 System.out.println("最大值："+Collections.max(list));  
 System.out.println("最小值："+Collections.min(list));  
 Collections.reverse(list);//反转  
 System.out.println("集合中的元素："+list);  
 list=(ArrayList<Integer>) Collections.synchronizedList(list);  
 }  
}

//不具备自然顺序的元素，需要比较器 Comparator

package wh06;  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.Comparator;  
import java.util.List;  
import java.util.concurrent.Callable;  
class Person{  
 String name;  
 int age;  
 public Person(String name,int age){  
 super();  
 this.name=name;  
 this.age=age;  
 }  
 @Override  
 public String toString() {  
 return "{姓名："+this.name+"年龄："+this.age+"}";  
 }  
}  
class AgeCompartor implements Comparator<Person>{  
 @Override  
 public int compare(Person o1, Person o2) {  
 return o1.age-o2.age;  
 }  
}  
public class Demo07 {  
 //如果集合存储的都是不具备自然顺序的元素，那么排序需要传入比较器  
 public static void main(String[] args) {  
 ArrayList<Person> list=new ArrayList<Person>();  
 list.add(new Person("狗娃",14));  
 list.add(new Person("狗蛋",16));  
 list.add(new Person("狗生",11));  
 Collections.sort(list,new AgeCompartor());  
 System.out.println("找到的索引值是"+Collections.binarySearch(list,new Person("狗蛋",16),new AgeCompartor()));  
  
 System.out.println("最大值是："+Collections.max(list,new AgeCompartor()));  
 System.out.println("最小值是："+Collections.min(list,new AgeCompartor()));  
  
 System.out.println("集合中的元素："+list);  
 }  
}

// 数组的工具类（Arrays）

package wh06;  
  
import java.util.Arrays;  
import java.util.List;  
  
/\*\*  
 \*1~3前面已经讲过  
 \*  
 \* 1,二分法查找，数组需要有序  
 \* binarySearch(int[]);  
 \* binarySearch(double[]);  
 \* 2,数组排序  
 \* sort(int[]);  
 \* sort(char[]);  
 \* 3,将数据变成字符串  
 \* toString(int[]);  
 \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
 \*  
 \*

\* 4,复制数组  
 \* copyOf(boolean[] original,int newLength);  
 \* original:源数组  
 \* newLength:新数组的长度  
 \* 5,复制部分数组  
 \* copyOfRange(boolean[] original,int from,int to);  
 \* original:源数组  
 \* from:开始拷贝的索引值  
 \* to:结束拷贝的索引值  
 \* 6,比较两个数组的元素是否相同  
 \* equals(int[],int[]);  
 \* 7,将数组变成集合  
 \* List asList();  
 \*  
 \*/

//4~7的讲解  
public class Demo08 {  
 public static void main(String[] args) {  
 Integer[] arr={10,8,6,9};  
 Integer[] arr0={8,10,6,9};  
 Integer[] arr2=Arrays.copyOf(arr,5);//返回的是一个数组  
 Integer[] arr3=Arrays.copyOfRange(arr,1,3);  
  
 System.out.println("arr2拷贝后的数组元素："+Arrays.toString(arr2));  
 System.out.println("arr3拷贝后的数组元素："+Arrays.toString(arr3));  
 System.out.println("判断两个数组元素是否完全一致：："+Arrays.equals(arr,arr2));//false  
 System.out.println("判断两个数组对应位置的元素是否完全一致："+Arrays.equals(arr,arr0 ));//false  
  
 List<Integer> list=Arrays.asList(arr);//把数组变集合  
 System.out.println("集合的元素："+list);  
 }  
}

Demo09（正则表达式）

正则表达式：用于操作字符串的一个规则，正则表达式的规则使用了特殊的符号表示

package wh07;  
 \* 预定义字符类  
 \* . 任何字符(与行结束符可能匹配也可能不匹配)  
 \* \d 数字[0-9]  
 \* \D 非数字[^0-9]  
 \* \s 空白字符[\t \n x0B /f /r]  
 \* \S 非空白字符[^\s]  
 \* \w 单词字符[a-zA-Z\_0-9]  
 \* \W 非单词字符[^\w]  
 \*  
 \* 注意：任何预定义字符没有加上数量词之前都只能匹配一个字符。

public class Demo10{  
 public static void main(String[] args){  
  
 System.out.println("任意字符："+("%".matches(".")));  
 System.out.println("数字字符："+("1".matches("\\d")));  
 System.out.println("非数字字符："+("#".matches("\\D")));  
 System.out.println("空白字符："+("\f".matches("\\s")));  
 System.out.println("非空白字符："+("7".matches("\\S")));  
 System.out.println("单词字符："+("a".matches("\\w")));  
 System.out.println("非单词字符："+("&".matches("\\W")));

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   
 Greedy 数量词（大跨号）  
 X? X 一次或者零次  
 X\* X 零次或者多次  
 X+ X 一次或者多次  
 X{n} X 恰好n次  
 X{n,} X 至少n次  
 X{n,m} X 至少n次，但不超过m次  
  
  
 System.out.println("?一次或者一次也没有："+"22".matches("\\d?"));  
 System.out.println("\*零次或多次："+"22".matches("\\d\*"));  
 System.out.println("+至少出现一次："+"22".matches("\\d+"));  
 System.out.println("｛次数｝恰好出现n次："+"22".matches("\\d{5}"));  
 System.out.println("｛次数，｝至少n次："+"22".matches("\\d{3,}"));  
 System.out.println("｛次数1，次数2｝制定出现次数范围："+"22344".matches("\\d{4,6}"));  
   
 //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
   
 范围词：(方跨号)  
 [abc] a,b或c(简单类)  
 [^abc] 除了a,b,c的任何字符（否定）  
 [a-zA-Z] a到z或A到Z，两头包括在内（范围）  
  
 [a-d[m-p]] a到d或m到p的并集 == [a-dm-p]  
 [a-z&&[def]] d,e或f(交集)  
 注意：范围词里面不管内容多长，没有数量词的匹配都只能匹配一个字符  
 System.out.println("d".matches("[abc]"));  
 System.out.println("@".matches("[^abc]"));  
 System.out.println("字符可以出现在a-z之间："+("#".matches("[a-z0-9]")));  
  
 }  
}  
   
 单词边界匹配器：  
 \b 单词边界匹配器只是代表了单词的开始或者是结束部分，不匹配任何字符  
   
class Test01{  
 public static void main(String[] args) {  
 System.out.println("hello world".matches("hello\\ world"));  
 }  
 }

Demo10

需求：校验一个 QQ号 不能以0开头,长度为5--11位,只能由数字组成

public class Demo09 {  
 public static void main(String[] args) {  
 String QQ="234545a6";  
 if(!QQ.startsWith("0")){  
 if(QQ.length()>=5&&QQ.length()<=11){  
 try {  
 Long.parseLong(QQ);  
 System.out.println("恭喜你，你的QQ浩合法");  
 }catch (Exception e){  
 System.out.println("非法QQ号，只能由数字组成");  
 }  
 }else {  
 System.out.println("非法QQ，长度有误");  
 }  
 }else {  
 //以0开头  
 System.out.println("非法QQ，不能以0开头");  
 }  
 System.out.println(QQ.matches("[1-9]\\d{4,10}")?"合法QQ":"非法QQ");}}

Demo11

package wh07;  
import java.util.Arrays;  
/\*\*  
 \* 正则表达式主要适用于操作字符串的规则，正则表达式对字符串的操作主要有以下几种应用：  
 \*  
 \* 匹配：  
 \* matches（）；  
 \* 切割：  
 \* split();  
 \* 替换：  
 \* (replaceAll(String regex,String replacement));  
 \* 查找：  
 \* find();  
 \* group();  
 \*/

public class Demo11 {  
 public static void main(String[] args) {  
 //matchesPhone("13879299691");  
 //matchesTel("0791-222131");  
 testPlit2();  
 replaceAll();  
 replaceTest2();  
 }  
 //需求：编写一个正则表达式匹配手机号 第一位：只能1开头；第二位：3 4 5 7 8 ；长度：11位  
 public static void matchesPhone(String phone){  
 System.out.println(phone.matches("1[34578]\\d{9}")?"合法手机号":"非法手机号");  
 }  
 //需求2：匹配固定电话 区号-主机号 区号：首尾是0 长度：3-4 主机号：7-8位  
 public static void matchesTel(String tel){  
 System.out.println(tel.matches("0\\d{2,3}-[1-9]\\d{6,7}")?"合法固话":"非法固话");  
 }  
  
 //按照空格切割  
 public static void testPlitl(){  
 String str="明 天 放 假";  
 String [] datas=str.split(" +");  
 System.out.println("数组的元素："+ Arrays.toString(datas));  
 }  
 //根据重叠词进行切割  
 public static void testPlit2(){  
 String str="大家家家家明天天天玩的的的的开心";  
 String[] datas=str.split("(.)\\1+");//如果正则的内容需要被复用，那么需要对正则的内容进行分组；分组的目的就是为了提高正则的复用性；组号不能指定，组号是从1开始的  
 System.out.println("数组的元素："+Arrays.toString(datas));  
 }  
  
 //替换  
 public static void replaceAll(){  
 String str="如有需求请联系我：13567872323；"+"如有需求请联系我：13567872323；"+"如有需求请联系我：13567872323；"+"如有需求请联系我：13567872323；"+"如有需求请联系我：13567872323；"+"如有需求请联系我：13567872323；"  
 +"如有需求请联系我：13567872323；";  
 String reg="1[345678]\\d{9}";  
 str=str.replaceAll(reg,"####");  
 System.out.println("被替换的内容是："+str);  
 }  
 public static void replaceTest2(){  
 String str="我我我要要做做做做项项项项项目目目";  
 //还原结巴的话----->我要做项目 把重叠词替换成单个单词  
 str=str.replaceAll("(.)\\1+","$1");//如果需要在replaceAll方法正则的外部引用组的内容，那么是使用"$组号"  
 System.out.println(str);  
 }  
}

package wh07;  
  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
/\*\*  
 \* 查找：  
 \* 指定为字符串的正则表达式必须首先被编译为此类的实例。然后，可以将得到的正则对象匹配任意的字符串用于创建Matcher对象，  
 \* 执行匹配所涉及的所有状态都驻留在匹配器中，所以多个匹配器可以共享同一模式。因此，典型的调用顺序是：  
 \*  
 \* Pattern p=Pattern.compile("正则对象");  
 \* Matcher m=p.matcher("aaaab");//信息留在匹配器里  
 \* boolean b=m.matches();  
  
 \* 查找需要使用的对象：  
 \* 1，Pattern (正则对象)  
 \* 2，Matcher (匹配器对象)  
 \*  
 \* 匹配器要使用的方法：  
 \* 1，find() 通知匹配器去匹配字符串，查找符合规则的字符串;如果可以查找到符合规则的字符串，则返回true 否则返回false  
 \* 2，group() 获取符合规则的子串  
 \*  
 \* 注意：使用group方法时一定要先调用find方法去让匹配器查找符合规则的字符串，否则会报错。  
 \*/

public class Demo12 {  
 public static void main(String[] args) {  
 //找出三个字母组成的单词  
 String content="one day after one days you are foolish !";  
 String reg="\\b[a-zA-Z]{3}\\b";// 使用\b边界符确定一个单词  
 //先把要字符串的正则编译成Pattern对象  
 Pattern p=Pattern.compile(reg);  
 //使用正则对象匹配字符串用产生一个Matcher对象  
 Matcher m=p.matcher(content);  
 //System.out.println("有符合规则的字符串吗"+m.find()+m.group());  
 while (m.find()){  
 System.out.println(m.group());  
 }  
 }  
}

Demo12

package wh07;  
  
import java.util.regex.Matcher;  
import java.util.regex.Pattern;  
  
/\*\*  
 \* 网络爬虫  
 \*/  
public class Demo13 {  
 public static void main(String[] args) {  
 String content="有事没事联系：1223122@163.com"+"有事没事联系：1223122@163.com.cn"+"有事没事联系：1223122@163.com"+"有事没事联系：1223122@163.net"+"有事没事联系：1223122@163.com";  
 String reg="[a-zA-Z1-9]\\w{5,17}@[a-zA-Z0-9]{2,}(\\.(com|cn|net)){1,2}";  
 //把字符串的正则编译成正则对象  
 Pattern p=Pattern.compile(reg);  
 //使用正则对象产生匹配器对象  
 Matcher m= p.matcher(content);  
 while(m.find()){  
 System.out.println("爬来的结果是："+m.group());  
 }  
 }  
}

Day19//静态导入,增强for循环//可变参数,自动装箱与自动拆箱//枚举类//File类//

//Jdk1.5新特性//

Demo01

package cn.itcast.jdk15;

import java.util.ArrayList;

import java.util.Collections;

/\*

jdk1.5新特性之-------静态导入

静态导入的作用： 简化书写。

静态导入可以作用一个类的所有静态成员。

静态导入的格式：

import static 包名.类名.静态的成员；

静态导入要注意的事项：

1. 如果静态导入的成员与本类的成员存在同名的情况下，那么默认使用本类的静态成员，如果需要指定使用静态导入的成员，那么需要在静态成员前面加上类名

\*/

import static java.util.Collections.sort;

import static java.util.Collections.binarySearch;

import static java.util.Collections.max;

import static java.lang.System.out;

public class Demo1 {

public static void main(String[] args) {

ArrayList<Integer> list = new ArrayList<Integer>();

list.add(13);

list.add(9);

list.add(10);

list.add(19);

//排序

Collections.sort(list);

out.println("集合的元素："+ list);

out.println("索引值："+ binarySearch(list,13));

out.println("最大值："+ max(list));

}

public static void sort(ArrayList<Integer> list){

System.out.println("本类 的sort方法.....");

}

}

Demo02

package cn.itcast.jdk15;

import java.util.HashMap;

import java.util.HashSet;

import java.util.Iterator;

import java.util.Map;

import java.util.Set;

/\*

jdk1.5出现的新特性---->增强for循环

增强for循环的作用： 简化迭代器的书写格式。(注意：增强for循环的底层还是使用了迭代器遍历。)

增强for循环的适用范围： 如果是实现了Iterable接口的对象或者是数组对象都可以使用增强for循环。

增强for循环的格式：

for(数据类型 变量名 : 遍历的目标){

}

增强for循环要注意的事项：

1. 增强for循环底层也是使用了迭代器获取的，只不过获取迭代器由jvm完成，不需要我们获取迭代器而已，所以在使用增强for循环变量元素的过程中，不准使用集合对象对集合的元素个数进行修改。

2. 迭代器遍历元素与增强for循环变量元素的区别：使用迭代器遍历集合的元素时可以删除集合的元素，而增强for循环变量集合的元素时，不能调用迭代器的remove方法删除元素。

3. 普通for循环与增强for循环的区别：普通for循环可以没有变量的目标，而增强for循环一定要有变量的目标。

\*/

public class Demo2 {

public static void main(String[] args) {

HashSet<String> set = new HashSet<String>();

//添加元素

set.add("狗娃");

set.add("狗剩");

set.add("铁蛋");

/\*

//使用迭代器遍历Set的集合.

Iterator<String> it = set.iterator();

while(it.hasNext()){

String temp = it.next();

System.out.println("元素："+ temp);

it.remove();

}

//使用增强for循环解决

for(String item : set){

System.out.println("元素："+ item);

}

int[] arr = {12,5,6,1};

普通for循环的遍历方式

for(int i = 0 ; i<arr.length ; i++){

System.out.println("元素："+ arr[i]);

}

//使用增强for循环实现

for(int item :arr){

System.out.println("元素："+ item);

}

//需求： 在控制台打印5句hello world.

for(int i = 0 ; i < 5; i++){

System.out.println("hello world");

}

\*/

//注意： Map集合没有实现Iterable接口，所以map集合不能直接使用增强for循环，如果需要使用增强for循环需要借助于Collection

// 的集合。

HashMap<String, String> map = new HashMap<String, String>();

map.put("001","张三");

map.put("002","李四");

map.put("003","王五");

map.put("004","赵六");

Set<Map.Entry<String, String>> entrys = map.entrySet();

for(Map.Entry<String, String> entry :entrys){

System.out.println("键："+ entry.getKey()+" 值："+ entry.getValue());

}

}

}

Demo03

package cn.itcast.jdk15;

import java.util.Iterator;

//自定一个类使用增强for循环

class MyList implements Iterable<String>{

Object[] arr = new Object[10];

int index = 0 ; //当前的指针

public void add(Object o){

arr[index++] = o; // 1

}

public int size(){

return index;

}

@Override

public Iterator<String> iterator() {

return new Iterator<String>() {

int cursor = 0;

@Override

public boolean hasNext() {

return cursor<index;

}

@Override

public String next() {

return (String) arr[cursor++];

}

@Override

public void remove() {

}

};

}

}

public class Demo3 {

public static void main(String[] args) {

MyList list = new MyList();

list.add("张三");

list.add("李四");

list.add("王五");

for(String item :list){

System.out.println(item);

}

}

}

Demo04

package cn.itcast.jdk15;

/\*

jdk1.5新特性之------->可变参数

需求： 定义一个函数做加法功能（函数做几个数据 的加法功能是不确定）。

可变参数的格式：

数据类型... 变量名

可变参数要注意的细节：

1. 如果一个函数的形参使用上了可变参数之后，那么调用该方法的时候可以传递参数,也可以不传递参数。

2. 可变参数实际上是一个数组对象。

3. 可变参数必须位于形参中的最后一个参数。

4. 一个函数最多只能有一个可变参数，因为可变参数要位于形参中最后一个位置上。

\*/

public class Demo4 {

public static void main(String[] args) {

int[] arr = {1,2,45,6,7};

/\*System.out.println(arr);

add(arr);\*/

add();

}

public static void add(int... arr){ //长度是0

int result = 0;

for(int item : arr){

result+=item;

}

System.out.println("总和："+ result);

}

}

Demo05

package cn.itcast.jdk15;

import java.util.ArrayList;

/\*

jdk1.5新特性之-----自动装箱与自动拆箱。

java是面向对象 的语言，任何事物都可以使用类进行描述，sun就使用了

一些类描述java中八种基本数据类型数据

基本数据类型 包装类型

byte Byte

short Short

int Integer

long Long

float Float

double Double

boolean Boolean

char Character

基本数据类型数据有了对应 的包装 类型的好处：

\*/

public class Demo5 {

public static void main(String[] args) {

String str = "12";

//字符串转换成int类型数据。 可以把字符串转换成对应的数字

int i = Integer.parseInt(str);

System.out.println(i+1);

//把数字转换成字符串

System.out.println("把整数转换成对应 的字符串："+Integer.toString(i));

//把整数转换成对应的进制形式

System.out.println("10的二进制："+ Integer.toBinaryString(10));

System.out.println("10的十六进制："+ Integer.toHexString(10));

//可以把字符串当进行对应进制数据转换

String data = "10";

int a = Integer.parseInt(data, 2);

System.out.println("a="+a);//输出：2进制10的整数形式：2

//集合： 集合是可以存储任意对象类型数据的容器。

ArrayList list = new ArrayList();

list.add(1);

list.add(2);

list.add(3);

//自动装箱： 自动把java的基本数据类型数据转换成对象类型（引用类型）数据。

int temp = 10; //基本数据类型

Integer b =temp; //把a存储的值赋予给b变量

//自动拆箱： 把引用类型的数据转换成基本类型的数据

Integer c = new Integer(13);

int d = c; //

System.out.println(d);

//引用的数据类型

Integer e = 128;

Integer f = 128;

System.out.println("同一个对象吗？"+(e==f)); //比较内存地址 Integer类内部维护 了缓冲数组，该缓冲数组存储的-128~127 这些数据在一个数组中。如果你获取的数据是落入到这个范围之内的，那么就直接从该缓冲区中获取对应的数据。

}

}

Demo06

package cn.itcast.jdk15;

/\*

jdk1.5新特性之-----枚举

问题：某些方法所接收的数据必须是在固定范围之内的，

解决方案： 这时候我们的解决方案就是自定义一个类,然后是私有化构造函数，在自定义类中创建本类的对象对外使用。

jdk1.5对以上问题提出了新的解决方案： 就是使用枚举类解决。

一些方法在运行时，它需要的数据不能是任意的，而必须是一定范围内的值，Java5以后可以直接使用枚举予以解决。

比如： 方向 , 性别 、 季节 、 星期......

\*/

/\*

//自定义一个性别类

class Gender{

String value;

public static final Gender man = new Gender("男");

public static final Gender woman = new Gender("女");

private Gender(String value) {

this.value = value;

}

}

\*/

enum Gender{//等同于上面的class Gender{}

man("男"),woman("女");//枚举值

String value;

private Gender(String value){

this.value = value;

}

}

class Person{

private String name;

private Gender sex;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public Gender getSex() {

return sex;

}

public void setSex(Gender sex) {

this.sex = sex;

}

}

public class Demo6 {

public static void main(String[] args) {

Person p = new Person();

p.setName("狗娃");

p.setSex(Gender.woman);

System.out.println("名字："+ p.getName()+" 性别："+ p.getSex().value);

}

}

Demo07

package cn.itcast.jdk15;

/\*

枚举：一些方法在运行时，它需要的数据不能是任意的，而必须是一定范围内的值，可以直接使用枚举予以解决。

枚举类的定义格式：

enum 类名{

//枚举值

}

枚举要注意的细节：

1. 枚举类也是一个特殊的类。

2. 枚举值默认的修饰符是public static final。

3. 枚举值就是是枚举值所属的类的类型， 然后枚举值是指向了本类的对象的。

4. 枚举类的构造方法默认的修饰符是private的。

5. 枚举类可以定义自己的成员变量与成员函数。

6. 枚举类可以自定义构造函数，但是构造函数的修饰符必须是privat;并且在枚举值里传入参数。

7. 枚举类可以存在抽象 的方法，但是枚举值必须要实现抽象 的方法。

8. 枚举值必须要位置枚举类 的第一个语句。

\*/

//自定义一个枚举类

enum Sex{

man("男"){

@Override

public void run() {

System.out.println("男人在跑...");

}

},woman("女"){

@Override

public void run() {

System.out.println("女人在跑...");

}

}; //枚举值

String value; //成员 变量

// public static final Sex man = new Sex();

//构造函数

private Sex(String value){

this.value = value;

}

//成员函数

public void getValue(){

System.out.println("value :"+ value);

}

public abstract void run();

}

public class Demo7 {

public static void main(String[] args) {

Sex sex = Sex.man; //获取到了枚举类的对象

sex.value = "男";

sex.getValue();

sex.run();

}

}

Demo08

package cn.itcast.jdk15;

/\*

switch适用的数据类型： byte \ char \short \ int \ String\枚举类型

注意：

case语句后面跟的枚举值，只需要单写枚举值即可，不需要再声明该 枚举值是属于哪个枚举类的。

\*/

//季节枚举类

enum Season{

spring,summer,autumn,winter;

}

enum Person2{

student,worker;

}

public eclass Demo8 {

public static void main(String[] args) {

Season season = Season.summer;//获取枚举值对象

switch(season){

case spring:

System.out.println("春天...");

break;

case summer:

System.out.println("夏天...");

break;

case autumn:

System.out.println("秋天...");

break;

case winter:

System.out.println("冬天...");

break;

}

}

}

Demo09

package cn.itcast.jdk15;

import java.util.ArrayList;

import java.util.Iterator;

import java.util.List;

public class Test {

public static void main(String[] args) {

List list = new ArrayList();

list.add("1");

list.add("4");

list.add("2");

list.add("7");

// 因为2是最后一个元素了，也就是删除后 没有在调用了迭代器的next方法。

Iterator<String> it = list.iterator();

while(it.hasNext()){

String s = it.next();

if("1".equals(s)){

list.remove("1");

break;

}

}

System.out.println(list);

}

}

//File类//

Demo01

package cn.itcast.file;

import java.io.File;

/\*

IO流(Input Output) ：

IO技术主要的作用是解决设备与设备之间 的数据传输问题。 比如： 硬盘--->内存 内存的数据---->硬盘上 把键盘的数据------->内存中

IO技术的应用场景：

导出报表 ， 上传大头照 、 下载 、 解释xml文件 ...

数据保存到硬盘上，该数据就可以做到永久性的保存。 数据一般是以文件的形式保存到硬盘上

sun使用了一个File类描述了文件或者文件夹的。

File类可以描述一个文件或者一个文件夹。

File类的构造方法：

File(String pathname) 指定文件或者文件夹的路径创建一个File文件。

File(File parent, String child) 根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

File(String parent, String child)

目录分隔符： 在windows机器上 的目录分隔符是 \ ,在linux机器上的目录分隔符是/ .

注意： 在windows上面\ 与 / 都可以使用作为目录分隔符。 而且，如果写/ 的时候只需要写一个即可。

\*/

public class Demo1 {

public static void main(String[] args) {

//File file = new File("F:"+File.separator+"a.txt"); // 在linux机器上是不是一个合法路径？？？

File file = new File("F:/a.txt");

/\*File parentFile = new File("F:\\");

File file = new File("F:\\","a.txt");\*/

System.out.println("存在吗？ "+ file.exists()); // exists 判断该文件是否存在，存在返回true，否则返回false。

// System.out.println("目录分隔符："+ File.separator);

}

}

Demo02

package cn.itcast.file;

import java.io.File;

/\*

路径问题：

绝对路径： 该文件在硬盘上 的完整路径。绝对路径一般都是以盘符开头的。

相对路径: 相对路径就是资源文件相对于当前程序所在的路径。

. 当前路径

.. 上一级路径

注意： 如果程序当前所在的路径与资源文件不是在同一个盘下面，是没法写相对路径 的。

\*/

public class Demo2 {

public static void main(String[] args) {

File file = new File(".");//先获得绝对路径

System.out.println("当前路径是："+ file.getAbsolutePath());

File file2 = new File("..\\..\\a.txt");

System.out.println("存在吗？"+ file2.exists());

}

}

Demo03

package cn.itcast.file;

import java.io.File;

import java.io.IOException;

/\*

创建：

createNewFile() 在指定位置创建一个空文件，成功就返回true，如果已存在就不创建然后返回false

mkdir() 在指定位置创建目录，这只会创建最后一级目录，如果上级目录不存在就抛异常。

mkdirs() 在指定位置创建目录，这会创建路径中所有不存在的目录。

renameTo(File dest) 重命名文件或文件夹，也可以操作非空的文件夹，文件不同时相当于文件的剪切,剪切时候不能操作非空的文件夹。移动/重命名成功则返回true，失败则返回false。

\*/

public class Demo3 {

public static void main(String[] args) throws IOException {

File file = new File("F:\\aa.txt");

System.out.println("创建成功了吗？"+file.createNewFile()); //createNewFile 创建一个指定的文件，如果该文件存在了，则不会再创建，如果还没有存在则创建。创建成功返回true，否则返回false。

File dir = new File("F:\\a.txt");

System.out.println("创建文件夹成功吗？"+dir.mkdir()); // mkdir 创建一个单级文件夹，

dir = new File("F:\\aa\\bb");

System.out.println("创建多级文件夹："+ dir.mkdirs());

//renameTo() 如果目标文件与源文件是在同一个路径下，那么renameTo的作用是重命名， 如果目标文件与源文件不是在同一个路径下，那么renameTo的作用就是剪切，而且还不能操作文件夹。

File destFile = new File("F:\\aaaaaaw");

System.out.println("重命名成功吗？"+file.renameTo(destFile)) ;

}

}

Demo04

package cn.itcast.file;

import java.io.File;

/\*

删除：

delete() 删除文件或一个空文件夹，如果是文件夹且不为空，则不能删除，成功返回true，失败返回false。

deleteOnExit() 在虚拟机终止时，请求删除此抽象路径名表示的文件或目录，保证程序异常时创建的临时文件也可以被删除

判断：

exists() 文件或文件夹是否存在。

isFile() 是否是一个文件，如果不存在，则始终为false。

isDirectory() 是否是一个目录，如果不存在，则始终为false。

isHidden() 是否是一个隐藏的文件或是否是隐藏的目录。

isAbsolute() 测试此抽象路径名是否为绝对路径名。

\*/

public class Demo4 {

public static void main(String[] args) {

/\*

删除的。

\* File file = new File("F:\\a.txt");

System.out.println("删除成功吗？ "+ file.delete()); //delete方法不能用于删除非空的文件夹。 delete方法会马上删除一个文件。

file.deleteOnExit(); //jvm退出的时候删除文件。 一般用于删除临时 文件。

判断

\*/

File file = new File("..\\..\\a.txt");

System.out.println("存在吗？"+ file.exists());

System.out.println("判断是否是一个文件："+file.isFile()); //如果是文件返回true，否则返回false.

System.out.println("判断是否是一个文件夹："+ file.isDirectory()); // 是文件夹返回ture，否则返回false.

System.out.println("是隐藏的 文件吗："+ file.isHidden());

System.out.println("是绝对路吗？"+ file.isAbsolute());

}

}

Demo05

package cn.itcast.file;

import java.io.File;

import java.sql.Date;

import java.text.SimpleDateFormat;

/\*

获取：

getName() 获取文件或文件夹的名称，不包含上级路径。

getPath() 返回绝对路径，可以是相对路径，但是目录要指定

getAbsolutePath() 获取文件的绝对路径，与文件是否存在没关系

length() 获取文件的大小（字节数），如果文件不存在则返回0L，如果是文件夹也返回0L。

getParent() 返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回null。

lastModified() 获取最后一次被修改的时间。

\*/

public class Demo5{

public static void main(String[] args) {

File file = new File("..\\..\\a.txt");

System.out.println("文件名："+ file.getName());

System.out.println("获取绝对路径："+ file.getPath());

System.out.println("getAbsolutePath获取绝对路径："+file.getAbsolutePath());

System.out.println("获取文件的的大小(字节为单位)："+ file.length());

System.out.println("获取文件的父路径："+ file.getParent());

//使用毫秒值转换成Date对象

long lastModified = file.lastModified();

Date date = new Date(lastModified);

SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy年MM月dd日 HH:mm:ss");

System.out.println("获取最后一次的修改时间(毫秒值)："+ dateFormat.format(date) );

}

}

Demo06

package cn.itcast.file;

import java.io.File;

/\*

文件夹相关：

static File[] listRoots() 列出所有的根目录（Window中就是所有系统的盘符）

list() 返回目录下的文件或者目录名，包含隐藏文件。对于文件这样操作会返回null。

listFiles() 返回目录下的文件或者目录对象（File类的实例），包含隐藏文件。对于文件这样操作会返回null。

list(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

listFiles(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

\*/

public class Demo6 {

public static void main(String[] args) {

/\*File[] roots = File.listRoots(); //列出所有的根目录

for(File file : roots){

System.out.println(file);

}\*/

//指定目录

File file = new File("F:\\1208\\day17");

String[] fileNames = file.list(); //把当前文件夹下面的所有子文件名与子文件夹名存储到一个String类型的数组中返回。

for(String fileName : fileNames){

System.out.println(fileName);

}

File[] files = file.listFiles(); // 把当前文件夹下面的所有子文件与子文件夹都使用了一个File对象描述，然后把这些File对象存储到一个File数组中返回

for(File fileItem : files){

System.out.println("文件名："+ fileItem.getName());

}

}

}

Demo07

package cn.itcast.file;

import java.io.File;

import java.io.FilenameFilter;

/\*

需求1 ： 指定一个文件夹，然后该文件夹下面所有java文件。

需求2： 指定一个文件夹，然后列出文件夹下面的所有子文件与文件夹，但是格式要如下:

文件：

文件名1

文件名2

文件名3

..

文件夹：

文件夹名1

文件夹名2

文件夹名3

....

listFiles(FilenameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

list(fileNameFilter filter) 返回指定当前目录中符合过滤条件的子文件或子目录。对于文件这样操作会返回null。

\*/

// 自定义一个文件名过滤器

class MyFilter implements FilenameFilter{

@Override

public boolean accept(File dir, String name) {

//System.out.println("文件夹:"+dir+" 文件名："+ name);

return name.endsWith(".java");

}

}

public class Demo7 {

public static void main(String[] args) {

File dir = new File("F:\\1208\\day06\\代码\\day06");

listJava2(dir);

}

public static void listJava2(File dir){

File[] files = dir.listFiles(new MyFilter()); //得到文件夹下面的所有子文件与文件夹。

//增强for循环{数据类型:File变量名:file遍历目标:files}

for(File file : files){

System.out.println(file.getName());

}

}

//列出所有的java文件

public static void listJava(File dir){

File[] files = dir.listFiles(); //获取到了所有的子文件

for(File file : files){

String fileName = file.getName();

/\*if(fileName.endsWith(".java")&&file.isFile()){//isFile()的作用是过滤掉同名的文件夹

System.out.println(fileName);

}\*/

if(fileName.matches(".+\\.java")&&file.isFile()){//使用正则表达式

System.out.println(fileName);

}

}

}

public static void listFile(File dir){

File[] files = dir.listFiles();//获取到所有的子文件

System.out.println("文件：");

for(File fileItem : files){

if(fileItem.isFile()){

System.out.println("\t"+fileItem.getName());

}

}

System.out.println("文件假：");

for(File fileItem : files){

if(fileItem.isDirectory()){

System.out.println("\t"+fileItem.getName());

}

}

}

}

Day20//FileInputStream//FileOutputStream//异常处理//缓冲输入与输出//

Demo01//Input

package cn.itcast.input;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

/\*

File类: 用于描述一个文件或者文件夹的。

通过File对象我们可以读取文件或者文件夹的属性数据，如果我们需要读取文件的内容数据，那么我们需要使用IO流技术。

IO流（Input Output）

IO流解决问题： 解决设备与设备之间的数据传输问题。 内存--->硬盘 硬盘--->内存

IO流技术：

IO流分类：

如果是按照数据的流向划分：

输入流

输出流

如果按照处理的单位划分：

字节流: 字节流读取得都是文件中二进制数据，读取到二进制数据不会经过任何的处理。

字符流： 字符流读取的数据是以字符为单位的 。 字符流也是读取文件中的二进制数据，不过会把这些二进制数据转换成我们能 识别的字符。

字符流 = 字节流 + 解码

输入字节流：

--------| InputStream 所有输入字节流的基类 抽象类

------------| FileInputStream 读取文件数据的输入字节流

使用FileInputStream读取文件数据的步骤：

1. 找到目标文件

2. 建立数据的输入通道。

3. 读取文件中的数据。

4. 关闭 资源.

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

readTest4();

}

//方式4：使用缓冲数组配合循环一起读取。28ms

public static void readTest4() throws IOException{

long startTime = System.currentTimeMillis();

//找到目标文件

File file = new File("F:\\美女\\1.jpg");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲数组配合循环读取文件的数据。

int length = 0; //保存每次读取到的字节个数。

byte[] buf = new byte[1024]; //存储读取到的数据缓冲数组的长度一般是1024的倍数，因为与计算机的处理单位吻合理论上缓冲数组越大，效率越高。

while((length = fileInputStream.read(buf))!=-1){ // read方法如果读取到了文件的末尾，那么会返回-1表示。

System.out.print(new String(buf,0,length));

}

//关闭资源

fileInputStream.close();

long endTime = System.currentTimeMillis();

System.out.println("读取的时间是："+ (endTime-startTime));

}

//方式3：使用缓冲数组读取。 缺点： 无法读取完整一个文件的数据.

public static void readTest3() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲字节数组，读取文件的数据。

byte[] buf = new byte[1024]; //相当于超市里面的购物车。

int length = fileInputStream.read(buf); // 如果使用read读取数据传入字节数组，那么数据是存储到字节数组中的，而这时候read方法的返回值是表示的是本次读取了几个字节数据到字节数组中。

System.out.println("length:"+ length);

//使用字节数组构建字符串

String content = new String(buf,0,length);

System.out.println("内容："+ content);

//关闭资源

fileInputStream.close();

}

//方式2 ： 使用循环读取文件的数据

public static void readTest2() throws IOException{

long startTime = System.currentTimeMillis();

//找到目标文件

File file = new File("F:\\美女\\1.jpg");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//读取文件的数据

int content = 0; //声明该变量用于存储读取到的数据

while((content = fileInputStream.read())!=-1){

System.out.print((char)content);

}

//关闭资源

fileInputStream.close();

long endTime = System.currentTimeMillis();

System.out.println("读取的时间是："+ (endTime-startTime));

}

//方式1：读取的方式一缺陷：无法读取完整一个文件的数据.

public static void readTest1() throws IOException{

//1. 找到目标文件

File file = new File("F:\\a.txt");

//2.建立数据的输入通道。

FileInputStream fileInputStream = new FileInputStream(file);

//3.读取文件中的数据

int content = fileInputStream.read(); // read() 读取一个字节的数据，把读取的数据返回。

System.out.println("读到的内容是："+ (char)content);

//4.关闭资源 实际上就是释放资源。

fileInputStream.close();

}

}

Demo02//Input

package cn.itcast.input;

import java.io.File;

import java.io.FileInputStream;

import java.io.IOException;

/\*

问题1： 读取完一个文件的数据的时候，我不关闭资源有什么影响?

答案： 资源文件一旦 使用完毕应该马上释放，否则其他的程序无法对该资源文件进行其他 的操作。

\*/

import java.util.Arrays;

public class Demo2 {

public static void main(String[] args) throws IOException {

//找到目标文件

File file = new File("F:\\day18-day21.IO.doc");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲字节数组读取文件

byte[] buf = new byte[8];

int length = 0 ;//设置初始值为0

while((length = fileInputStream.read(buf))!=-1){

System.out.print(new String(buf)); // aaaa bbba bbb' '

// System.out.println(Arrays.toString(buf));

}

//释放文件资源

fileInputStream.close();

}

}

Demo01//Output

package cn.itcast.output;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

输出字节流：

--------| OutputStream 是所有输出字节流 的父类。 抽象类

-----------| FileOutStream 向文件输出数据的输出字节流。

FileOutputStream如何使用呢？

1. 找到目标文件

2. 建立数据的输出通道。

3. 把数据转换成字节数组写出。

4. 关闭资源

FileOutputStream要注意的细节：

1. 使用FileOutputStream 的时候，如果目标文件不存在，那么会自动创建目标文件对象。

2. 使用FileOutputStream写数据的时候，如果目标文件已经存在，那么会先清空目标文件中的数据，然后再写入数据。

3.使用FileOutputStream写数据的时候, 如果目标文件已经存在，需要在原来数据基础上追加数据的时候应该使用new FileOutputStream(file,true)构造函数，第二参数为true。

4.使用FileOutputStream的write方法写数据的时候，虽然接收的是一个int类型的数据，但是真正写出的只是一个字节的数据，只是

把低八位的二进制数据写出，其他二十四位数据全部丢弃。

00000000-000000000-00000001-11111111 511

11111111---> -1

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

writeTest3();

}

//3.使用字节数组把数据写出。

public static void writeTest3() throws IOException{

//找到目标文件

File file = new File("F:\\b.txt");

//建立数据输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file);

//把数据写出。

String data = "时间的流逝";

byte[] buf = data.getBytes();

fileOutputStream.write(buf, 0, buf.length); // 0 从字节数组的指定索引值开始写， buf.length：写出整长度的个字节。

//关闭资源

fileOutputStream.close();

}

//2.使用字节数组把数据写出。

public static void writeTest2() throws IOException{

//找到目标文件

File file = new File("F:\\b.txt");

//建立数据输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file,true);

//把数据写出。

String data = "\r\nhello world";

fileOutputStream.write(data.getBytes());

//关闭资源

fileOutputStream.close();

}

//1.每次只能写一个字节的数据出去。

public static void writeTest1() throws IOException{

//找到目标文件

File file = new File("F:\\b.txt");

//建立数据的输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file);

//把数据写出

fileOutputStream.write('h');

fileOutputStream.write('e');

fileOutputStream.write('l');

fileOutputStream.write('l');

fileOutputStream.write('o');

//关闭资源

fileOutputStream.close();

}

}

Deno02//Output

package cn.itcast.output;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Arrays;

/\*

4.使用FileOutputStream的write方法写数据的时候，虽然接收的是一个int类型的数据，但是真正写出的只是一个字节的数据，只是

把低八位的二进制数据写出，其他二十四位数据全部丢弃。

00000000-000000000-00000001-11111111 511

\*/

public class Demo2 {

public static void main(String[] args) throws IOException {

readTest();

}

public static void readTest() throws IOException{

//找到目标文件

File file = new File("F:\\c.txt");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲输入读取文件数据

byte[] buf = new byte[4];

//读取文件数据

int length = fileInputStream.read(buf);

System.out.println("字节数组的内容："+ Arrays.toString(buf));

//关闭资源

fileInputStream.close();

}

public static void writeTest() throws FileNotFoundException, IOException {

//找到目标文件

File file = new File("F:\\c.txt");

//建立数据的输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file);

//把数据写出

fileOutputStream.write(511);

//关闭资源

fileOutputStream.close();

}

}

Demo03//Output

package cn.itcast.output;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

需求： 拷贝一张图片。

\*/

public class CopyImage {

public static void main(String[] args) throws IOException {

//找到目标文件

File inFile = new File("F:\\美女\\1.jpg");

File destFile = new File("E:\\1.jpg");

//建立数据的输入输出通道

FileInputStream fileInputStream = new FileInputStream(inFile);

FileOutputStream fileOutputStream = new FileOutputStream(destFile,true); //追加数据....

//每新创建一个FileOutputStream的时候，默认情况下FileOutputStream 的指针是指向了文件的开始的位置。 每写出一次，指向都会出现相应移动。

//建立缓冲数据，边读边写

byte[] buf = new byte[1024];

int length = 0 ;

while((length = fileInputStream.read(buf))!=-1){ //最后一次只剩下了824个字节

fileOutputStream.write(buf,0,length); //写出很多次数据，所以就必须要追加。

}

//关闭资源 原则： 先开后关，后开先关。

fileOutputStream.close();

fileInputStream.close();

}

}

Demo01//Exception

package cn.itcast.exception;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import javax.management.RuntimeErrorException;

/\*

IO异常 的处理

\*/

public class Demo1 {

public static void main(String[] args) {

// readTest();

copyImage();

}

// 拷贝图片

public static void copyImage() {

FileInputStream fileInputStream = null;

FileOutputStream fileOutputStream = null;

try {

// 找到目标文件

File inFile = new File("F:\\美女\\1.jpg");

File outFile = new File("E:\\1.jpg");

// 建立输入输出通道

fileInputStream = new FileInputStream(inFile);

fileOutputStream = new FileOutputStream(outFile);

// 建立缓冲数组，边读边写

byte[] buf = new byte[1024];

int length = 0;

while ((length = fileInputStream.read(buf)) != -1) {

fileOutputStream.write(buf, 0, length);

}

} catch (IOException e) {

System.out.println("拷贝图片出错...");

throw new RuntimeException(e);

} finally {

// 关闭资源

try {

if (fileOutputStream != null) {

fileOutputStream.close();

System.out.println("关闭输出流对象成功...");

}

} catch (IOException e) {

System.out.println("关闭输出流资源失败...");

throw new RuntimeException(e);

} finally {

if (fileInputStream != null) {

try {

fileInputStream.close();

System.out.println("关闭输入流对象成功...");

} catch (IOException e) {

System.out.println("关闭输入流对象失败...");

throw new RuntimeException(e);

}

}

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

public static void readTest() {

FileInputStream fileInputStream = null;

try {

// 找到目标文件

File file = new File("F:\\aaaaa.txt");

// 建立数据输入通道

fileInputStream = new FileInputStream(file);

// 建立缓冲数组读取数据

byte[] buf = new byte[1024];

int length = 0;

while ((length = fileInputStream.read(buf)) != -1) {

System.out.print(new String(buf, 0, length));

}

} catch (IOException e) {

/\*

\* //处理的代码... 首先你要阻止后面的代码执行,而且要需要通知调用者这里出错了... throw new

\* RuntimeException(e);

\* //把IOException传递给RuntimeException包装一层，然后再抛出，这样子做的目的是

\* 为了让调用者使用变得更加灵活。

\*/

System.out.println("读取文件资源出错....");

throw new RuntimeException(e);

} finally {

try {

if (fileInputStream != null) {

fileInputStream.close();

System.out.println("关闭资源成功...");

}

} catch (IOException e) {

System.out.println("关闭资源失败...");

throw new RuntimeException(e);

}

}

}

}

Demo01//BufferedInputStream

package cn.itcast.buffered;

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

我们清楚读取文件数据使用缓冲数组读取效率更高，sun也知道使用缓冲数组读取效率更高，那么这时候sun给我们提供了一个------缓冲输入字节流对象,让我们可以更高效率读取文件。

输入字节流体系：

----| InputStream 输入字节流的基类。 抽象

----------| FileInputStream 读取文件数据的输入字节流

----------| BufferedInputStream 缓冲输入字节流 缓冲输入字节流的出现主要是为了提高读取文件数据的效率。

其实该类内部只不过是维护了一个8kb的字节数组而已。

注意： 凡是缓冲流都不具备读写文件的能力。

使用BufferedInputStream的步骤 :

1. 找到目标文件。

2. 建立数据的输入通道

3. 建立缓冲输入字节流流

4. 读取文件

5. 关闭资源

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

readTest2();

}

public static void readTest2() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据输入通道

FileInputStream fileInputStream= new FileInputStream(file);

//建立缓冲输入字节流

//疑问一：为什么创建BuffereInputStream的时候需要传递 FileInputStream?

//BuffereInputStream 本身不具备读文件的能力，所以需要借助 FileInputStream来读文件

BufferedInputStream bufferedInputStream= new BufferedInputStream(fileInputStream);

//读取文件数据

int content = 0 ;

//疑问二：BufferedInputStream出现 的目的是了提高读取文件的效率，但是BufferedInputStream的read方法每次读取一个字节的数据

//而FileInputStreram每次也是读取一个字节的数据，那么BufferedInputStream效率高从何而来？

byte[] buf=new byte[1024];

while((content = bufferedInputStream.read(buf))!=-1){

System.out.print(new String(buf,0,content);

}

//关闭资源

bufferedInputStream.close();//调用BufferedInputStream的close方法实际上关闭的是FileinputStream.

}

//读取文件的时候我们都是使用缓冲数组读取。效率会更加高

public static void readTest() throws IOException{

File file = new File("F:\\a.txt");

//建立数组通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲数组读取数据

byte[] buf = new byte[1024\*8];

int length = 0;

while((length = fileInputStream.read(buf))!=-1){

System.out.print(new String(buf,0,length));

}

//关闭资源

fileInputStream.close();

}

}

/\*

bufferedInputStream.read();

FileOutputStream fileOutputStream= new FileOutputStream(file);

BufferedOutputStream bufferedOutputStream= new BufferedOutputStream(fileOutputStream);

fileOutputStream.write(null);

\*/

Demo02//BufferedOutputStream

package cn.itcast.buffered;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

输出字节流

--------| OutputStream 所有输出字节流的基类 抽象类

------------| FileOutputStream 向文件 输出数据 的输出字节流

------------| Bufferedoutputstream 缓冲输出字节流 BufferedOutputStream出现的目的是为了提高写数据的效率。 内部也是维护了一个8kb的字节数组而已。

使用BufferedOutputStream的步骤：

1. 找到目标文件

2. 建立数据的输出通道

3.建立缓冲输出字节流对象

4.把数据写出

5.把缓冲数组中内部的数据写到硬盘上面

BufferedOutputStream 要注意的细节

1. 使用BufferedOutStream写数据的时候，它的write方法是是先把数据写到它内部维护的字节数组中。

2. 使用BufferedOutStream写数据的时候，它的write方法是是先把数据写到它内部维护的字节数组中，如果需要把数据真正的写到硬盘上面，需要调用flush方法或者是close方法、或者是内部维护的字节数组 已经填满数据的时候。

\*/

public class Demo2 {

public static void main(String[] args) throws IOException {

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file);

//建立缓冲输出字节流对象

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(fileOutputStream);

//把数据写出

bufferedOutputStream.write("hello world".getBytes());

//把缓冲数组中内部的数据写到硬盘上面。

//方法1：bufferedOutputStream.flush();

//方法2：bufferedOutputStream.close();

}

}

Demo03//BufferedCopyImage

package cn.itcast.buffered;

import java.io.BufferedInputStream;

import java.io.BufferedOutputStream;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

/\*

练习： 使用缓冲输入输出字节流拷贝一个图片。

\*/

public class CopyImage {

public static void main(String[] args) throws IOException {

//找到目标文件

File inFile = new File("F:\\美女\\1.jpg");

File outFile = new File("E:\\1.jpg");

//建立数据输入输出通道

FileInputStream fileInputStream = new FileInputStream(inFile);

FileOutputStream fileOutputStream = new FileOutputStream(outFile);

//建立缓冲输入输出流

BufferedInputStream bufferedInputStream = new BufferedInputStream(fileInputStream);

BufferedOutputStream bufferedOutputStream = new BufferedOutputStream(fileOutputStream);

//边读边写

int content = 0;

// int length = bufferedInputStream.read(buf); 如果传入了缓冲数组，内容是存储到缓冲数组中，返回值是存储到缓冲数组中的字节个数。

while((content = bufferedInputStream.read())!=-1){ // 如果使用read方法没有传入缓冲数组，那么返回值是读取到的内容。

bufferedOutputStream.write(content);

// bufferedOutputStream.flush();不需要flush,会影响效率

}

//关闭资源

bufferedInputStream.close();

bufferedOutputStream.close();

}

}

//Day21//输入字符流//输出字符流//字符流拷贝文件//缓冲输入输出字符流//装饰者设计模式

Demo01(复习)

package day21;

/\*

字节流：

输入字节流

-----------| InputStream 输入字节流的基类 抽象类

----------------| FileInputStream 读取文件数据的输入字节流。

----------------| BufferedInputStream 缓冲输入字节流 缓冲输入字节流出现的目的： 为了提高读取文件数据的效率。 该类其实内部就是维护了一个8kb字节数组而已。

输出字节流：

---------| OutputStream 输出字节流的基类。 抽象类。

--------------| FileOutStream 向文件输出数据的输出字节流。

--------------| BufferedOutputStream 缓冲输出字节流。 该类出现的目的是为了提高写数据的效率 。 其实该类内部也是维护了一个8kb的数组而已，当调用其write方法的时候数据默认是向它内部的数组中存储 的，只有调用flush方法或者是close方法或者是8kb的字节数组存储满数据的时候才会真正的向硬盘输出。

\*/

Demo02(测试字节流的读写中文)

package cn.itcast.reader;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Arrays;

public class Demo1 {

public static void main(String[] args) throws IOException {

// writeTest();

readrTest();

}

//使用字节流读取中文

public static void readrTest() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//读取内容

//int content = 0;

/\*while((content = fileInputStream.read())!=-1){ //出现乱码的原因： 一个中文在gbk码表中默认是占两个字节，

// 目前你只读取了一个字节而已，所以不是一个完整的中文。

System.out.print((char)content);

}\*/

byte[] buf = new byte[2];

for(int i = 0 ; i < 3 ; i++){

fileInputStream.read(buf);

System.out.print(new String(buf));

}

//关闭资源

fileInputStream.close();

}

//使用字节流写中文。字节流之所以能够写中文是因为借助了字符串的getBytes方法对字符串进行了编码（字符---->数字）

public static void writeTest() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输出通道

FileOutputStream fileOutputStream = new FileOutputStream(file);

//准备数据，把数据写出。

String data = "大家好";

byte[] buf = data.getBytes(); //把字符串转换成字节数组

System.out.println("输出的内容："+ Arrays.toString(buf));

fileOutputStream.write(buf);

///关闭资源

fileOutputStream.close();

}

}

Demo01(输入字符流)

package cn.itcast.reader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

/\*

字节流：字节流读取的是文件中的二进制数据，读到的数据并不会帮你转换成你看得懂的字符。

字符流： 字符流会把读取到的二进制的数据进行对应的编码与解码工作；字符流 = 字节流 + 编码(解码)

输入字符流：

----------| Reader 输入字符流的基类 抽象类

-------------| FileReader 读取文件的输入字符流。

FileReader的用法：

1. 找到目标文件

2. 建立数据的输入通道

3. 读取数据

4. 关闭资源

\*/

public class Demo2 {

public static void main(String[] args) throws IOException {

readTest2();

}

//使用缓冲字符数组读取文件。

public static void readTest2() throws IOException{

//找到目标文件

File file = new File("F:\\1208project\\day21\\src\\day21\\Demo1.java");

// 建立数据的输入通道

FileReader fileReader = new FileReader(file);

//建立缓冲字符数组读取文件数据，效率高

char[] buf = new char[1024];

int length = 0 ;

while((length = fileReader.read(buf))!=-1){

System.out.print(new String(buf,0,length));

}

}

//未使用缓冲字符数组

public static void readTest1() throws IOException{

//找到目标文件

File file = new File("F:\\1208project\\day21\\src\\day21\\Demo1.java");

//建立数据的输入通道

FileReader fileReader = new FileReader(file);

int content = 0 ;

while((content = fileReader.read())!=-1){ //每次只会读取一个字符，效率低。

System.out.print((char)content);

}

//关闭资源

fileReader.close();

}

}

Demo01(输出字符流)

package cn.itcast.writer;

import java.io.File;

import java.io.FileOutputStream;

import java.io.FileWriter;

import java.io.IOException;

/\*

输出字符流:

------| Writer 输出字符流的基类。 抽象类

-----------| FileWriter 向文件数据数据的输出字符流

FileWriter的使用步骤：

1. 找到目标文件。

2. 建立数据输出通道

3. 写出数据。

4. 关闭资源

FileWriter要注意的事项：

1. 使用FileWriter写数据的时候，FileWriter内部是维护了一个1024个字符数组的，写数据的时候会先写入到它内部维护的字符数组中，如果需要把数据真正写到硬盘上，需要调用flush或者是close方法或者是填满了内部的字符数组。

2. 使用FileWriter的时候，如果目标文件不存在，那么会自动创建目标文件。

3.使用FileWriter的时候， 如果目标文件已经存在了，那么默认情况会先清空文件中的数据，然后再写入数据 ， 如果需要在原来的基础上追加数据，需要使用“new FileWriter(File , boolean)”的构造方法，第二参数为true。

练习： 使用字符流拷贝一个文本文件(java文件).

接着使用字符流拷贝一个图片（观察图片的大小变化，思考为什么会这样子??）

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

writeTest1();

}

public static void writeTest1() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据输出通道

FileWriter fileWriter = new FileWriter(file,true);//第二参数为true,则是追加内容，不会清空原来的内容

//准备数据，把数据写出

String data = "今天天气非常好！！";

fileWriter.write(data); //字符流 具备解码的功能

//刷新字符流

// fileWriter.flush();

//关闭资源

fileWriter.close();

}

}

Demo02

package cn.itcast.writer;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileReader;

import java.io.IOException;

//使用字节流读取中文(具有风险性)

public class Demo2 {

public static void main(String[] args) throws IOException {

File file = new File("F:\\a.txt");

FileInputStream fileInputStream = new FileInputStream(file);

byte[] buf = new byte[1024];//数组如果太小时，就会出现乱码

int length = 0;

while((length = fileInputStream.read(buf))!=-1){

System.out.println(new String(buf,0,length)); //借用字符串的解码功能。

}

fileInputStream.close();

}

}

Demo03(copy)

package cn.itcast.writer;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

/\*

何时使用字符流，何时使用字节流？依据是什么？

使用字符流的应用场景： 如果是读写字符数据的时候则使用字符流。

使用字节流的应用场景： 如果读写的数据都不需要转换成字符的时候，则使用字节流。

\*/

//使用字符流拷贝文件

public class Copy {

public static void main(String[] args) throws IOException {

BufferedReader bufferedReader = new BufferedReader(new FileReader("F:\\Test.txt"));

BufferedWriter bufferedWriter = new BufferedWriter(new FileWriter("E:\\Test.txt"));

String line=null;

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

bufferedWriter.write(line);

}

bufferedWriter.close();

bufferedReader.close();

}

}

Demo01(BufferedReader)

package cn.itcast.buffered;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.util.Arrays;

/\*

输入字符流:

-------| Reader 所有输入字符流的基类。 抽象类

----------| FileReader 读取文件字符串的输入字符流。

----------| BufferedReader 缓冲输入字符流 缓冲输入字符流出现的目的是为了提高读取文件的效率和拓展了FileReader的功能。其实该类内部也是维护了一个字符数组。

记住：缓冲流都不具备读写文件的能力。

BufferedReader的使用步骤：

1.找到目标文件

2 .建立数据的输入通道

3.建立缓冲输入字符流

4.读取数据

5.关闭资源

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

// readTest1();

File file = new File("F:\\1208project\\day21\\src\\cn\\itcast\\buffered\\Demo1.java");

//建立数据的输入通道

FileReader fileReader = new FileReader(file);

String line = null;

while((line = myReadLine(fileReader))!=null){

System.out.println(line);

}

}

//自己去实现readLine方法。

public static String myReadLine(FileReader fileReader) throws IOException{

//创建一个字符串缓冲类对象

StringBuilder sb = new StringBuilder(); //StringBuilder主要是用于存储读取到的数据

int content = 0 ;

while((content = fileReader.read())!=-1){

if(content=='\r'){

continue;

}else if(content=='\n'){

break;

}else{

//普通字符

sb.append((char)content);

}

}

//代表已经读取完毕了。

if(content ==-1){

return null;

}

return sb.toString();

}

public static void readTest1() throws IOException{

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输入通道。

FileReader fileReader = new FileReader(file);

//建立缓冲输入字符流

BufferedReader bufferedReader = new BufferedReader(fileReader);

//读取数据

/\*int content = bufferedReader.read(); //读到了一个字符。 读取到的字符肯定也是从Bufferedreader内部的字符数组中获取的到。所以效率高。

System.out.println((char)content);\*/

//使用BufferedReader拓展的功能，readLine() 一次读取一行文本，如果读到了文件的末尾返回null表示。

String line = null;

while((line = bufferedReader.readLine())!=null){ // 虽然readLine每次读取一行数据，但是但会的line是不包含\r\n的

System.out.println(Arrays.toString(Arrays.toString(line.getBytes()));

}

//关闭资源

bufferedReader.close();//其实关闭的也是FileReader

}

}

Demo02(BuffereWriter)

package cn.itcast.buffered;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileWriter;

import java.io.IOException;

/\*

输出字符流

----------| Writer 所有输出字符流的基类， 抽象类。

--------------- | FileWriter 向文件输出字符数据的输出字符流。

----------------| BufferedWriter 缓冲输出字符流 缓冲输出字符流作用:提高FileWriter的写数据效率与拓展FileWriter的功能。

BufferedWriter内部只不过是提供了一个8192长度的字符数组作为缓冲区而已，拓展了FileWriter的功能。

BufferedWriter如何使用？

1. 找到目标文件

2. 建立数据的输出通道

3.建立缓冲输出流对象

4.关闭

\*/

public class Demo2 {

public static void main(String[] args) throws IOException {

//找到目标文件

File file = new File("F:\\a.txt");

//建立数据的输出通道

FileWriter fileWriter = new FileWriter(file,true);

//建立缓冲输出流对象

BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

//写出数据

// bufferedWriter.newLine(); //newLine() 换行。 实际上就是向文件输出\r\n.

bufferedWriter.write("\r\n");

bufferedWriter.write("前两天李克强来萝岗！！");

//关闭资源

bufferedWriter.flush();

// bufferedWriter.close();

}

}

Demo03（Login）

package cn.itcast.buffered;

import java.io.BufferedReader;

import java.io.BufferedWriter;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Scanner;

/\*

练习： 缓冲输入输出字符流用户登陆注册...

\*/

public class Login {

static Scanner scanner = new Scanner(System.in);

public static void main(String[] args) throws IOException {

while(true){

System.out.println("请选择功能： A(注册) B(登陆)");

String option = scanner.next();

if("a".equalsIgnoreCase(option)){

//注册

reg();

}else if("b".equalsIgnoreCase(option)){

//登陆

login();

}else{

System.out.println("你的输入有误,请重新输入...");

}

}

}

//登陆

public static void login() throws IOException{

System.out.println("请输入用户名：");

String userName = scanner.next();

System.out.println("请输入密码：");

String password = scanner.next();

String info = userName+" "+ password;

//读取文件的信息，查看是否有该用户的信息存在，如果存在则登陆成功。

//建立数据的输入通道

//建立缓冲输入字符流

BufferedReader bufferedReader = new BufferedReader(new FileReader("F:\\users.txt"));

String line = null;

boolean isLogin = false; // 用于记录是否登陆成功的标识， 默认是登陆失败的。

//不断的读取文件的内容

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

if(info.equals(line)){

isLogin = true;

break;

}

}

if(isLogin){

System.out.println("欢迎"+userName+"登陆成功...");

}else{

System.out.println("不存在该用户信息，请注册!!");

}

bufferedReader.close();

}

//注册

public static void reg() throws IOException{

System.out.println("请输入用户名：");

String userName = scanner.next();

System.out.println("请输入密码：");

String password = scanner.next();

String info = userName+" "+ password;

//把用户的注册的信息写到文件上

File file = new File("F:\\users.txt");

FileWriter fileWriter = new FileWriter(file,true);

//建立缓冲输出字符流

BufferedWriter bufferedWriter = new BufferedWriter(fileWriter);

//把用户信息写出

bufferedWriter.write(info);

bufferedWriter.newLine();//分行写出

//关闭资源

bufferedWriter.close();

}

}

Demo01(装饰者设计模式)

package other;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.io.IOException;

import java.io.Reader;

/\*

装饰者设计模式：增强一个类的功能，而且还可以让这些装饰类互相装饰。

BufferedReader是不是拓展了FileReader的功能。

BuferedWriter 也是拓展了FileWriter的功能。

需求1： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有行号。（显示行数）

需求2：编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有分号。

需求3： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有双引号。

----| Reader

-----------| BufferedReader

---------------| BufferedLineNum 带行号

---------------| BufferedSemi 带分号

---------------| BufferedQuto 带双引

---------------| 子类..

---------------|

增强一个类的功能时候我们可以选择使用继承：

通过继承实现增强一个类的功能优点：代码结构清晰，通俗易懂。

缺点： 使用不灵活，会导致继承的体系过于庞大。

\*/

class BufferedLineNum extends BufferedReader{

//行号

int count = 1 ;

public BufferedLineNum(Reader in) {

super(in);

}

@Override

public String readLine() throws IOException {

String line = super.readLine();

if(line ==null){

return null;

}

line = count+" "+ line;

count++;

return line;

}

}

//带分号的缓冲输入字符流

class BufferedSemi extends BufferedReader{

public BufferedSemi(Reader in) {

super(in);

}

@Override

public String readLine() throws IOException {

String line = super.readLine();

if(line==null){

return null;

}

line = line+";";

return line;

}

}

//带双引号的缓冲输入字符流

class BufferedQuto extends BufferedReader{

public BufferedQuto(Reader in) {

super(in);

}

@Override

public String readLine() throws IOException {

String line = super.readLine();

if(line==null){

return null;

}

line = "\""+line+"\"";

return line;

}

}

public class Demo1 {

public static void main(String[] args) throws IOException {

File file = new File("F:\\Demo1.java");

//建立数据的输入通道

FileReader fileReader = new FileReader(file);

//建立带行号的缓冲输入字符流

BufferedLineNum bufferedLineNum = new BufferedLineNum(fileReader);

//带有分号的缓冲输入字符流

BufferedSemi bufferedSemi = new BufferedSemi(fileReader);

//带有双引号的缓冲输入字符流

BufferedQuto bufferedQuto = new BufferedQuto(fileReader);

String line = null;

while((line = bufferedQuto.readLine())!=null){

System.out.println(line);

}

bufferedQuto.close();

}

}

Demo02(装饰者设计模式)

package other;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileReader;

/\*

装饰者设计模式：增强一个类的功能，而且还可以让这些装饰类互相装饰。

装饰者设计模式的步骤：

1. 在 装饰类 的内部维护一个 被装饰类 的引用。

2. 让 装饰类 有一个共同的父类或者是父接口。

需求1： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有行号。

需求2：编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有分号。

需求3： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有双引号。

需求4： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有行号+ 分号。

需求5： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有分号+ 双引号。

需求6： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有双引号+ 行号。

需求7： 编写一个类拓展BufferedReader的功能， 增强readLine方法返回 的字符串带有行号+ 分号+双引号。

继承实现的增强类和修饰模式实现的增强类有何区别？

继承实现的增强类：

优点：代码结构清晰，而且实现简单.

缺点：对于每一个的需要增强的类都要创建具体的子类来帮助其增强，这样会导致继承体系过于庞大。

修饰模式实现的增强类：

优点：内部可以通过多态技术对多个需要增强的类进行增强， 可以是这些装饰类达到互相装饰的效果。使用比较灵活。

缺点：需要内部通过多态技术维护需要被增强的类的实例。进而使得代码稍微复杂。

\*/

import java.io.IOException;

//带行号的缓冲输入字符流

class BufferedLineNum2 extends BufferedReader{

//在内部维护一个被装饰类的引用。

BufferedReader bufferedReader;

int count = 1;

public BufferedLineNum2(BufferedReader bufferedReader){

super(bufferedReader);// 注意： 该语句没有任何的作用，只不过是为了让代码不报错。

this.bufferedReader = bufferedReader;

}

//实现readLine方法

public String readLine() throws IOException{

String line = bufferedReader.readLine();

if(line==null){

return null;

}

line = count+": "+line;

count++;

return line;

}

}

//带分号缓冲输入字符流

class BufferedSemi2 extends BufferedReader{ //为什么要继承? 是为了让这些装饰类的对象可以作为参数进行传递，达到互相装饰 的效果。

//在内部维护一个被装饰类的引用。

BufferedReader bufferedReader;

public BufferedSemi2(BufferedReader bufferedReader){ // new BuffereLineNum();

super(bufferedReader);// 注意： 该语句没有任何的作用，只不过是为了让代码不报错。

this.bufferedReader = bufferedReader;

}

public String readLine() throws IOException{

String line = bufferedReader.readLine(); //如果这里的ReadLine方法是调用了buffereLineNum的readLine方法，问题马上解决。

if(line==null){

return null;

}

line = line +";";

return line;

}

}

//缓冲类带双引号

class BufferedQuto2 extends BufferedReader{

//在内部维护一个被装饰的类

BufferedReader bufferedReader;

public BufferedQuto2(BufferedReader bufferedReader){ //new BufferedSemi2();

super(bufferedReader) ; //只是为了让代码不报错..

this.bufferedReader = bufferedReader;

}

public String readLine() throws IOException{

String line = bufferedReader.readLine(); //如果这里的ReadLine方法是调用了buffereLineNum的readLine方法，问题马上解决。

if(line==null){

return null;

}

line = "\""+line +"\"";

return line;

}

}

public class Demo2 {

public static void main(String[] args) throws IOException {

File file = new File("F:\\Demo1.java");

//建立通道

FileReader fileReader = new FileReader(file);

//建立缓冲输入字符流

BufferedReader bufferedReader = new BufferedReader(fileReader);

//建立带行号的缓冲输入字符流

BufferedLineNum2 bufferedLineNum = new BufferedLineNum2(bufferedReader);

//带分号的缓冲输入字符流

BufferedSemi2 bufferedSemi2 = new BufferedSemi2(bufferedLineNum);

//带双引号的缓冲输入字符流

BufferedQuto2 bufferedQuto2 = new BufferedQuto2(bufferedSemi2);

String line = null;

while((line = bufferedQuto2.readLine())!=null){

System.out.println(line);//输出的是需求7的结果

}

}

}

Demo03(装饰者设计模式)

package other;

/\*练习：

一家三口每个人都会工作，儿子的工作就是画画，母亲的工作就是在儿子的基础上做一个增强，不单止可以画画，还可以上涂料。

爸爸的工作就是在妈妈基础上做了增强，就是上画框。

\*/

装饰者设计模式的步骤：

1. 在 装饰类 的内部维护一个 被装饰类 的引用。

2. 让 装饰类 有一个共同的父类或者是父接口。

interface Work{

public void work();

}

class Son implements Work{

@Override

public void work() {

System.out.println("画画...");

}

}

class Mother implements Work{

//需要被装饰的类。

Work worker;

public Mother(Work worker){

this.worker = worker;

}

@Override

public void work() {

worker.work();

System.out.println("给画上颜色..");

}

}

class Father implements Work{

//需要被装饰的类的引用

Work worker;

public Father(Work worker){

this.worker = worker;

}

@Override

public void work() {

worker.work();

System.out.println("上画框...");

}

}

public class Demo3 {

public static void main(String[] args) {

Son s = new Son();

// s.work();

Mother m = new Mother(s);

// m.work();

Father f = new Father(s);

f.work();

}

Day22//SequenceInputStream//合并切割//对象的输入输出流对象//Properties配置文件//printStream//编码与解码//转换流//递归//

Demo01(复习)

package day22;

/\*

字节流

输入字节流：

-----------| InputStream 所有输入字节流的基类 抽象类

-----------------| FileInputStream 读取文件数据的输入字节流

-----------------| BufferedInputStream 缓冲输入字符流 该类出现的目的是为了提高读取文件 数据的效率。 这个类其实只不过是在内部维护了一个8kb的字节数组而已。

输出字节流：

-----------| OutputStream 所有输出字节流的基类。 抽象类。

----------------| FileOutputStream 向文件输出数据的输出字节流

----------------| BufferedOutputStream 缓冲输出字节流 该类出现的目的也是为了提高向文件写数据的效率。 这个类的也只不过是在内部维护了一个8kb的字节数组而已。

字符流 : 字符流 = 字节流 + 编码（解码）

输入字符流:

---------| Reader 所有输入字符流的基类。 抽象类。

----------------| FileReader 读取文件数据的输入字符流。

----------------| BufferedReader 缓冲输入字符流 该类出现的目的是为了提高读取文件数据的效率与拓展FileReader的(readLine)功能。 这个类的也只不过是在内部维护了一个8kb的字符数组而已。

输出字符流:

---------| Writer 所有输出字符流的基类。 抽象类

----------------| FileWriter 向文件输出数据的输出字符流

----------------| BufferedWriter 缓冲输出字符流 该类出现的目的是为了提高写文件数据的效率与拓展FileWriter的(newLine)功能.

\*/

Demo01(SequenceInputStream)

package cn.itcast.other;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.SequenceInputStream;

import java.util.ArrayList;

import java.util.Enumeration;

import java.util.Vector;

/\*

SequenceInputStream(序列流)

需求：把a.txt与b.txt 文件的内容合并。

\*/

public class Demo1 {

public static void main(String[] args) throws IOException {

merge3();

}

//3.把三个文件合并成一个文件

public static void merge3() throws IOException{

//找到目标文件

File file1 = new File("F:\\a.txt");

File file2 = new File("F:\\b.txt");

File file3 = new File("F:\\c.txt");

File file4 = new File("F:\\d.txt");

//建立对应 的输入输出流对象

FileOutputStream fileOutputStream = newFileOutputStream(file4);

FileInputStream fileInputStream1 = new FileInputStream(file1);

FileInputStream fileInputStream2 = new FileInputStream(file2);

FileInputStream fileInputStream3 = new FileInputStream(file3);

//创建序列流对象

Vector<FileInputStream> vector = new Vector<FileInputStream>();

vector.add(fileInputStream1);

vector.add(fileInputStream2);

vector.add(fileInputStream3);

//迭代器

Enumeration<FileInputStream> e = vector.elements();

SequenceInputStream sequenceInputStream = new SequenceInputStream(e);

//读取文件数据

byte[] buf = new byte[1024];

int length = 0;

while((length = sequenceInputStream.read(buf))!=-1){

fileOutputStream.write(buf,0,length);

}

//关闭资源

sequenceInputStream.close();

fileOutputStream.close();

}

// 2.使用SequenceInputStream合并文件

public static void merge2() throws IOException{

//找到目标文件

File inFile1 = new File("F:\\a.txt");

File inFile2 = new File("F:\\b.txt");

File outFile = new File("F:\\c.txt");

//建立数据的输入输出通道

FileOutputStream fileOutputStream = new FileOutputStream(outFile);

FileInputStream fileInputStream1 = new FileInputStream(inFile1);

FileInputStream fileInputStream2 = new FileInputStream(inFile2);

//建立序列流对象

SequenceInputStream inputStream = new SequenceInputStream(fileInputStream1,fileInputStream2);

byte[] buf = new byte[1024];

int length = 0 ;

while((length = inputStream.read(buf))!=-1){

fileOutputStream.write(buf,0,length);

}

//关闭资源

inputStream.close();

fileOutputStream.close();

}

//1.需求：把a.txt与b.txt 文件的内容合并

public static void merge1() throws IOException{

//找到目标文件

File inFile1 = new File("F:\\a.txt");

File inFile2 = new File("F:\\b.txt");

File outFile = new File("F:\\c.txt");

//建立数据的输入输出通道

FileOutputStream fileOutputStream = new FileOutputStream(outFile);

FileInputStream fileInputStream1 = new FileInputStream(inFile1);

FileInputStream fileInputStream2 = new FileInputStream(inFile2);

//把输入流存储到集合中，然后再从集合中读取

ArrayList<FileInputStream> list = new ArrayList<FileInputStream>();

list.add(fileInputStream1);

list.add(fileInputStream2);

//准备一个缓冲数组

byte[] buf = new byte[1024];

int length = 0 ;

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

FileInputStream fileInputStream = list.get(i);

while((length = fileInputStream.read(buf))!=-1){

fileOutputStream.write(buf,0,length);

}

//关闭资源

fileInputStream.close();

}

fileOutputStream.close();

}

}

Demo02(切割与合并Mp3)

package cn.itcast.other;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.SequenceInputStream;

import java.util.Enumeration;

import java.util.Vector;

/\*

需求： 把一首mp3先切割成n份，然后再把这些文件合并起来。

（视频是切割无法播放的）

\*/

public class Demo2 {

public static void main(String[] args) throws IOException {

// cutFile();

mergeFlile();

}

//合并

public static void mergeFlile() throws IOException{

//找到目标文件

File dir = new File("F:\\music");

//通过目标文件夹找到所有的MP3文件，然后把所有的MP3文件添加到vector中。

Vector<FileInputStream> vector = new Vector<FileInputStream>();

File[] files = dir.listFiles();

for(File file : files){

if(file.getName().endsWith(".mp3")){

vector.add(new FileInputStream(file));

}

}

//通过Vector获取迭代器

Enumeration<FileInputStream> e = vector.elements();

//创建序列流

SequenceInputStream inputStream = new SequenceInputStream(e);

//建立文件的输出通道

FileOutputStream fileOutputStream = new FileOutputStream("F:\\合并.mp3");

//建立缓冲数组读取文件

byte[] buf = new byte[1024];

int length = 0 ;

while((length = inputStream.read(buf))!=-1){

fileOutputStream.write(buf,0,length);

}

//关闭资源

fileOutputStream.close();

inputStream.close();

}

//切割MP3

public static void cutFile() throws IOException{

File file = new File("F:\\美女\\1.mp3");

//目标文件夹

File dir = new File("F:\\music");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立缓冲数组读取

byte[] buf = new byte[1024\*1024];

int length = 0;

for(int i = 0 ; (length = fileInputStream.read(buf))!=-1 ; i++){

FileOutputStream fileOutputStream = new FileOutputStream(new File(dir,"part"+i+".mp3"));//dir:目录名

fileOutputStream.write(buf,0,length);

fileOutputStream.close();

}

//关闭资源

fileInputStream.close();

}

}

Demo03（对象的输入输出流）

package cn.itcast.other;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

import java.io.Serializable;

/\*

对象的输入输出流 : 对象的输入输出流 主要的作用是用于写对象的信息与读取对象的信息。 对象信息一旦写到文件上那么对象的信息就可以做到持久化了

对象的输出流： ObjectOutputStream

对象的输入流: ObjectInputStream

ObjectOutputStream的使用步骤：

对象输入输出流要注意的细节：

1. 如果对象需要被写出到文件上，那么对象所属的类必须要实现Serializable接口。 Serializable接口没有任何的方法，是一个标识接口而已。

2. 对象的反序列化创建对象的时候并不会调用到构造方法的。

3. serialVersionUID 是用于记录class文件的版本信息的，serialVersionUID这个数字是通过一个类的类名、成员、包名、工程名算出的一个数字。

4. 使用ObjectInputStream反序列化的时候，ObjeectInputStream会先读取文件中的serialVersionUID，然后与本地的class文件的serialVersionUID进行对比，如果这两个id不一致，那么反序列化就失败了。

5. 如果序列化与反序列化的时候可能会修改类的成员，那么最好一开始就给这个类指定一个serialVersionUID，如果一类已经指定的serialVersionUID，然后在序列化与反序列化的时候，jvm都不会再自己算这个 class的serialVersionUID了。

6. 如果一个对象某个数据不想被序列化到硬盘上，可以使用关键字transient修饰。

7. 如果一个类维护了另外一个类的引用，那么另外一个类也需要实Serializable接口。

\*/

class Address implements Serializable{

String country;

String city;

public Address(String country,String city){

this.country = country;

this.city = city;

}

}

class User implements Serializable{

//添加默认的UID就可以修改成员变量

private static final long serialVersionUID = 1L;

String userName ;

String password;

transient int age; // transient 透明,瞬间,不可见

//维护了Adress类的引用

Address address ;

public User(String userName , String passwrod) {

this.userName = userName;

this.password = passwrod;

}

public User(String userName , String passwrod,int age,Address address) {

this.userName = userName;

this.password = passwrod;

this.age = age;

this.address = address;

}

@Override

public String toString() {

return "用户名："+this.userName+ " 密码："+ this.password+" 年龄："+this.age+" 地址："+this.address.city;

}

}

public class Demo3 {

public static void main(String[] args) throws IOException, Exception {

writeObj();

// readObj();

}

//把文件中的对象信息读取出来-------->对象的反序列化

public static void readObj() throws IOException, ClassNotFoundException{

//找到目标文件

File file = new File("F:\\obj.txt");

//建立数据的输入通道

FileInputStream fileInputStream = new FileInputStream(file);

//建立对象的输入流对象

ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

//读取对象信息

User user = (User) objectInputStream.readObject(); //创建对象肯定要依赖对象所属 的class文件

System.out.println("对象的信息："+ user);

}

//定义方法把对象的信息写到硬盘上------>对象的序列化。

public static void writeObj() throws IOException{

//把user对象的信息持久化存储。

Address address = new Address("中国","广州");

User user = new User("admin","123",15,address);

//找到目标文件

File file = new File("F:\\obj.txt");

//建立数据输出流对象

FileOutputStream fileOutputStream = new FileOutputStream(file);

//建立对象的输出流对象

ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);

//把对象写出

objectOutputStream.writeObject(user);

//关闭资源

objectOutputStream.close();

}

}

Demo04（Properties配置文件类）

package cn.itcast.other;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.util.Map;

import java.util.Map.Entry;

import java.util.Properties;

import java.util.Set;

/\*

Properties（配置文件类）: 主要用于生产配置文件与读取配置文件的信息。

Properties要注意的细节：

1. 如果配置文件的信息一旦使用了中文，那么在使用store方法生成配置文件的时候只能使用字符流解决，如果使用字节流生成配置文件的话，默认使用的是iso8859-1码表进行编码存储，这时候会出现乱码。

2. 如果Properties中的内容发生了变化，一定要重新使用properties.store()生成配置文件，否则配置文件信息不会发生变化。

\*/

public class Demo4 {

public static void main(String[] args) throws IOException {

creatProperties();

// readProperties();

}

//读取配置文件的信息

public static void readProperties() throws IOException{

//创建Properties对象

Properties properties = new Properties();

//加载配置文件信息到Properties中

properties.load(new FileReader("F:\\persons.properties"));

//遍历

Set<Entry<Object, Object>> entrys = properties.entrySet();

for(Entry<Object, Object> entry :entrys){

System.out.println("键："+ entry.getKey() +" 值："+ entry.getValue());

}

//修改狗娃的密码

//把修改后的Properties再生成一个配置文件

properties.setProperty("狗娃", "007");

properties.store(new FileWriter("F:\\persons.properties"), "hehe");

}

//保存一个配置文件文件的信息

public static void creatProperties() throws IOException{

//创建Properties

Properties properties = new Properties();

properties.setProperty("狗娃", "123");

properties.setProperty("狗剩","234");

properties.setProperty("铁蛋","345");

// 遍历Properties

/\*Set<Entry<Object, Object>> entrys = properties.entrySet();

for(Entry<Object, Object> entry :entrys){

System.out.println("键："+ entry.getKey() +" 值："+ entry.getValue());

}\*/

//使用Properties生产配置文件

//properties.store(new FileOutputStream("F:\\persons.properties"), "haha"); //第一个参数是一个输出流对象，第二参数是使用一个字符串描述这个配置文件的信息

properties.store(new FileWriter("F:\\persons.properties"), "hehe");//配置文件含有中文，就必须使用字符流生成文件

}

}

Demo05(Propterties练习)

package cn.itcast.other;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.util.Properties;

/\*

需求： 使用properties实现本软件只能 运行三次，超过了三次之后就提示购买正版，退jvm.

\*/

public class Demo5 {

public static void main(String[] args) throws IOException {

//目标文件

File file = new File("F:\\count.properties");

if(!file.exists()){

//如果配置文件不存在，则创建该配置文件

file.createNewFile();

}

//创建Properties对象。

Properties properties = new Properties();

//把配置文件的信息加载到properties中

properties.load(new FileInputStream(file));

FileOutputStream fileOutputStream = new FileOutputStream(file);//该语句有清空的作用，应放在load后;否则植入true但也会造成数据积累，不好

int count = 0; //定义该变量是用于保存软件的运行次数的

//读取配置文件的运行次数

String value = properties.getProperty("count");

if(value!=null){

count = Integer.parseInt(value);

}

//判断使用的次数是否已经达到了三次，

if(count==3){

System.out.println("你已经超出了试用次数，请购买正版软件！！");

System.exit(0);

}

count++;

System.out.println("你已经使用了本软件第"+count+"次");

properties.setProperty("count",count+"");

//使用Properties生成一个配置文件

properties.store(fileOutputStream,"runtime");

}

}

Demo06(PrintStream)

package cn.itcast.other;

import java.io.File;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.PrintStream;

/\*

打印流（PrintStream） 打印流可以打印任意类型的数据(包括对象)，而且打印数据之前都会先把数据转换成字符串再进行打印。

\*/

class Animal{

String name;

String color;

public Animal(String name,String color){

this.name = name;

this.color = color;

}

@Override

public String toString() {

return "名字："+this.name+ " 颜色："+ this.color;

}

}

public class Demo6 {

public static void main(String[] args) throws IOException {

/\*FileOutputStream fileOutputStream = new FileOutputStream("F:\\a.txt");

fileOutputStream.write("97".getBytes());

fileOutputStream.close();\*/

//作用1：打印流可以打印任何类型的数据，而且打印数据之前都会先把数据转换成字符串再进行打印。

File file = new File("F:\\a.txt");

//创建一个打印流

PrintStream printStream = new PrintStream(file);

/\*

printStream.println(97);

printStream.println(3.14);

printStream.println('a');

printStream.println(true);

Animal a = new Animal("老鼠", "黑色");

printStream.println(a);//打印对象a

//默认标准的输出流就是向控制台输出的，

System.setOut(printStream); //重新设置了标准的输出流对象

System.out.println("哈哈，猜猜我在哪里！！");

\*/

//作用2：收集异常的日志信息。

File logFile = new File("F:\\2015年1月8日.log");

PrintStream logPrintStream = new PrintStream( new FileOutputStream(logFile,true) );//使用OutputStream追加保存

try{

int c = 4/0;

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

int[] arr = null;

System.out.println(arr.length);

}catch(Exception e){

e.printStackTrace(logPrintStream);

}

}

}

Demo07（编码与解码）

package cn.itcast.other;

import java.io.UnsupportedEncodingException;

import java.util.Arrays;

/\*

编码与解码

编码： 把看得懂的字符变成看不懂码值这个过程我们称作为编码。

解码： 把码值查找对应的字符，我们把这个过程称作为解码。

注意： 以后编码与解码一般我们都使用统一的码表。否则非常容易出乱码。

\*/

public class Demo7 {

public static void main(String[] args) throws Exception {

/\*

String str = "中国";

// 平台默认的编码表是gbk编码表

//编码

byte[] buf = str.getBytes("utf-8");//更改指定为utf-8

System.out.println("数组的元素："+Arrays.toString(buf));

//默认使用了gbk码表去解码

str = new String(buf,"utf-8"); //根据编码的utf-8，使用相同的

System.out.println("解码后的字符串："+ str);

\*/

/\*String str = "a中国"; // -1,-2,0,97,78,45,86,-3

byte[] buf = str.getBytes("unicode"); //编码与解码的时候指定 的码表是unicode实际上就是用了utf-16；//-1，-2只是一个标识

System.out.println("数组的内容："+ Arrays.toString(buf));

\*/

String str = "大家好";

byte[] buf = str.getBytes(); //默认使用gbk进行编码

System.out.println("字节数组："+ Arrays.toString(buf)); // -76, -13, -68, -46, -70, -61

str = new String(buf,"iso8859-1");

// 出现乱码之后都可以被还原吗？

byte[] buf2 = str.getBytes("iso8859-1");

str = new String(buf2,"gbk");

System.out.println(str);

}

}

Demo08(转换流)

package cn.itcast.other;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.OutputStream;

import java.io.OutputStreamWriter;

import java.net.InetSocketAddress;

import java.net.Socket;

/\*

转换流:

输入字节流的转换流：

InputStreamReader 是字节流通向字符流的桥

输出字节流的转换流：

OutputStreamWriter 可以把输出字节流转换成输出字符流

转换流的作用：

1. 如果目前所 获取到的是一个字节流需要转换字符流使用，这时候就可以使用转换流。 字节流----> 字符流

2. 使用转换流可以指定编码表进行读写文件。

\*/

public class Demo8 {

public static void main(String[] args) throws IOException {

// readTest();

// writeTest();

// writeTest2();

readTest2();

}

//4.使用输入字节流的转换流指定码表进行读取文件数据

public static void readTest2() throws IOException{

File file = new File("F:\\a.txt");

//字节流对象

FileInputStream fileInputStream = new FileInputStream(file);

//创建字节流的转换流并且指定码表进行读取

InputStreamReader inputStreamReader = new InputStreamReader(fileInputStream,"utf-8");

char[] buf = new char[1024];

int length = 0;

while((length = inputStreamReader.read(buf))!=-1){

System.out.println(new String(buf,0,length));

}

}

//3.使用输出字节流的转换流指定码表写出数据

public static void writeTest2() throws IOException{

File file = new File("F:\\a.txt");

//建立数据的输出通道(输出字节流)

FileOutputStream fileOutputStream = new FileOutputStream(file);

//把输出字节流转换成字符流并且指定编码表

OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream, "utf-8");

outputStreamWriter.write("战争与和平");

//关闭资源

outputStreamWriter.close();

}

//2.需要把输出字节流转换成输出字符流

public static void writeTest() throws IOException{

File file = new File("F:\\a.txt");

//输出字节流

FileOutputStream fileOutputStream = new FileOutputStream(file);

//转换流

OutputStreamWriter outputStreamWriter = new OutputStreamWriter(fileOutputStream);

outputStreamWriter.write("大家好");

//关闭资源

outputStreamWriter.close();

}

//1.需要把输入字节流转换成输入字符流

public static void readTest() throws IOException{

InputStream in = System.in; //（字节流）获取了标准的输入流。

// System.out.println("读到 的字符："+ (char)in.read()); //read()一次只能读取一个字节

InputStreamReader inputStreamReader = new InputStreamReader(in);

//使用字符流的缓冲类

BufferedReader bufferedReader = new BufferedReader(inputStreamReader);//传入字符流inputStreamReader

String line = null;

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

System.out.println("内容："+ line);

}

}

}

Demo09（递归）

package cn.itcast.other;

/\*

递归：函数的自身调用函数的自身。

递归的使用前提： 必须要有条件的情况下调用。

需求： 算出5的阶乘。 5! = 5\*4 \*3 \* 2\*1

\*/

public class Demo9 {

public static void main(String[] args) {

int result = print(5);

System.out.println("结果："+ result);

}

//递归算法

public static int print(int num){//首先print传入 num=5

if(num==1){

return 1;

}else{

return num\*print(num-1);

}

}

/\*传统方法

public static int test(int num){

int result = 1;

while(num>0){

result = result\*num;

num--;

}

return result;

}

\*/

}

Demo10(递归)

package cn.itcast.other;

import java.io.File;

/\*

需求

1：列出一个文件夹的子孙文件与目录。

2：列出指定目录中所有的子孙文件与子孙目录名，要求名称前面要有相应数量的空格：

第一级前面有0个，第二级前面有1个，第三级前面有2个...，以此类推。

3：列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构，效果如下所示：

|--src

| |--cn

| | |--itcast

| | | |--a\_helloworld

| | | | |--HelloWorld.java

| | | |--b\_for

| | | | |--ForTest.java

| | | |--c\_api

| | | | |--Student.java

|--bin

| |--cn

| | |--itcast

| | | |--i\_exception

| | | | |--ExceptionTest.class

| | | |--h\_linecount

| | | | |--LineCounter3.class

| | | | |--LineCounter2.class

| | | | |--LineCounter.class

|--lib

| |--commons-io.jar

需求4: 删除一个非空的文件夹。

作业： 剪切一个非空目录。

\*/

public class Demo10 {

public static void main(String[] args) {

/\* File dir = new File("F:\\1208project\\day22");

listFiles3(dir,"|--");//第一级目录

File dir = new File("F:\\aa");

listFiles2(dir,"");//第一级目录

\*/

File dir=new File("F:\\aa");

deleteDir(dir);

}

//4.删除一个非空的目录

public static void deleteDir(File dir){

File[] files = dir.listFiles(); //列出了所有的子文件

for(File file : files){

if(file.isFile()){

file.delete();

}else if(file.isDirectory()){

deleteDir(file);//调用自身方法

}

}

dir.delete();//删除bb

}

//3.列出指定目录中所有的子孙文件与子孙目录名，要求要是树状结构

public static void listFiles3(File dir,String space){ //space存储的是空格

File[] files = dir.listFiles(); //列出所有的子文件

for(File file : files){

if(file.isFile()){

System.out.println(space+file.getName());

}else if(file.isDirectory()){

System.out.println(space+file.getName());

listFiles3(file,"| "+space);

}

}

}

//2.列出一个文件夹的子孙文件与目录，有空格分开

public static void listFiles2(File dir,String space){ //space存储的是空格

File[] files = dir.listFiles(); //列出所有的子文件

for(File file : files){

if(file.isFile()){

System.out.println(space+file.getName());

}else if(file.isDirectory()){

System.out.println(space+file.getName());

listFiles2(file," "+space);

}

}

}

//1.列出一个文件夹的子孙文件与目录

public static void listFiles1(File dir){

File[] files = dir.listFiles(); //列出所有的子文件

for(File file : files){

if(file.isFile()){

System.out.println("文件名："+file.getName());

}else if(file.isDirectory()){

System.out.println("文件夹："+file.getName());

listFiles1(file);//再次调用listFiles1方法

}

}

}

}

Day23//网络编程的三要素//udp的第一个例子//feiQ发送消息//群聊//tcp的引用//tcp的群聊与Tomcat//

Demo01(复习)

/\*

字节流

输入字节流：

---------| InputStream 所有输入字节流的基类。 抽象类

------------| FileInputStream 读取文件的输入字节流

------------| BufferedInputStream 缓冲输入字节流流，其实该类内部只不过是维护了8kb的字节数组而已。 出现的目的主要是为了提高读取文件的效率。

输出字节流：

---------| OutputStream 所有输出字节流的基类。 抽象类

--------------| FileOutputStream 向文件输出数据的输出字节流。

--------------| BufferedOutputStream 向文件输出数据的输出字节流。

字符流

输入字符流：

----------| Reader 所有输入字符流的基类。 抽象类

--------------| FileReader 读取文件字符的输入字符流 。

--------------| BufferedReader 缓冲输入字符流， 该类出现的目的主要是为了提高读取文件的效率与拓展功能(readLine)。

输出字符流

---------| Writer 所有输出字符流的基类。 抽象类。

-------------| FileWriter 向文件输出数据的输出字符流。

-------------| BufferedWriter 缓冲输出字符流， 该类出现 的目的是为了提高写文件数据的效率与拓展功能。

转换流

输入字节流的转换流 InputStreamReader

InputStream--------------------> Reader

输出字节流的转换流 OutputStream

OutputStream --------------------> Writer

转换流的作用：

1. 可以把字节流转换成字符流使用。

2. FileReader与FileWriter都是固定是gbk码表进行读写数据的，而转换流可以指定码表进行读写文件的数据。

Properties(配置文件类)

体系：

-------| Map

------------| HashTable

----------------| Properties 配置文件类、

store() 用于生成一个配置文件

load() 加载一个配置文件

注意：

1. 如果配置文件存在着中文，那么生成配置文件的时候要使用字符流，否则会出现乱码。

2. 如果需要修改配置文件的内容，应该先加载原本配置文件，然后再生成一个配置文件。

\*/

Demo01(网络通讯三要素)

package cn.itcast.ip;

import java.net.InetAddress;

import java.net.UnknownHostException;

/\*

网络编程： 网络编程主要用于解决计算机与计算机（手机、平板..）之间的数据传输问题。不需要基于html页面就可以达到数据之间的传输。 比如： feiQ , QQ , 微信....

网页编程： 就是要基于html页面的基础上进行数据的交互的。 比如： 珍爱网、 oa(办公自动化)、 高考的报告系统...

计算机网络： 分布在不同地域的计算机通过外部设备链接起来达到了消息互通、资源共享的效果就称作为一个计算机网络。

网络通讯的三要素：

1. IP

2. 端口号

3. 协议

192.168.10.1

IP地址： IP地址的本质就是一个由32位的二进制数据组成的数据。 后来别人为了方便我们记忆IP地址，就把IP地址切成了4份，每份8bit. 2^8 = 0~255

00000000-00000000-00000000-00000000

IP地址 = 网络号+ 主机号。

IP地址的分类：

A类地址 = 一个网络号 + 三个主机号 2^24 政府单位

B类地址 = 两个网络号+ 两个主机号 2^16 事业单位（学校、银行...）

C类地址= 三个网络号+ 一个主机号 2^8 私人使用

InetAddress(IP类)

常用的方法：

getLocalHost(); 获取本机的IP地址

getByName("IP或者主机名") 根据一个IP地址的字符串形式或者是一个主机名生成一个IP地址对象(用于获取别人的IP地址对象)

getHostAddress() 返回接受一个IP地址的字符串表示形式。

getHostName() 返回接受计算机的主机名。

端口号是没有类描述的。

端口号的范围： 0-65535 2^16个

0-1023 系统紧密绑定于一些服务

1024-49151系统松散的绑定了一些服务（也可以使用）

49152-65535 可以使用....

网络通讯的协议：

udp通讯协议

tcp通讯协议

\*/

public class Demo1 {

public static void main(String[] args) throws UnknownHostException {

//getLocalHost 获取本机的IP地址对象

/\*InetAddress address = InetAddress.getLocalHost();

System.out.println("IP地址："+address.getHostAddress());

System.out.println("主机名："+address.getHostName());\*/

//获取别人机器的IP地址对象

//可以根据一个IP地址的字符串形式或者是一个主机名，生成一个IP地址对象。

InetAddress address = InetAddress.getByName("Jolly-pc140116");

System.out.println("IP地址："+address.getHostAddress());

System.out.println("主机名："+address.getHostName());

//getAllByName返回一个数组

InetAddress[] arr = InetAddress.getAllByName("www.baidu.com");//域名

}

}

Demo(feiQ案例)

package test;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.UnknownHostException;

import java.util.Date;

public class FeiQDemo {

public static void main(String[] args) throws IOException {

//第一步：建立udp的服务

DatagramSocket datagramSocket = new DatagramSocket();

//第二步：准备数据，然后把数据封装 到数据包中，

String data = getData("大家早上好！");

DatagramPacket datagramPacket = new DatagramPacket(data.getBytes(), data.getBytes().length, InetAddress.getByName("192.168.15.255"), 2425);

//第三步： 调用udp的服务发送数据

datagramSocket.send(datagramPacket);

//关闭资源

datagramSocket.close();

}

//把数据组装成feiQ处理的格式数据

public static String getData(String data){

StringBuilder sb = new StringBuilder();

sb.append("1.0:");

sb.append(System.currentTimeMillis()+":");

sb.append("孔凡江:");

sb.append("192.168.15.31:");

sb.append("32:");

sb.append(data);

return sb.toString();

}

}

Demo01UDP(Sender)发送端

package cn.itcast.udp;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

/\*

在java中网络通讯业称作为Socket(插座)通讯，要求通讯的两台器都必须要安装Socket。

不同的协议就有不同的插座（Socket）

UDP通讯协议的特点：

1. 将数据极封装为数据包，面向无连接。

2. 每个数据包大小限制在64K中

3. 因为无连接，所以不可靠(可能数据丢失)

4. 因为不需要建立连接，所以速度快

5. udp 通讯是不分服务端与客户端的，只分发送端与接收端。

比如： 物管的对讲机, 飞Q聊天、 游戏...

udp协议下的Socket:

DatagramSocket(udp插座服务)发送与接收

DatagramPacket(数据包类)

DatagramPacket(buf, length, address, port)

buf: 发送的数据内容

length : 发送数据内容的大小。

address : 发送的目的IP地址对象

port : 端口号。

发送端的使用步骤：

1. 建立udp的服务。

2. 准备数据，把数据封装到数据包中发送。 发送端的数据包要带上ip地址与端口号。

3. 调用udp的服务，发送数据。

4. 关闭资源。

\*/

//发送端

public class Demo1Sender {

public static void main(String[] args) throws IOException {

//建立udp的服务

DatagramSocket datagramSocket = new DatagramSocket();

//准备数据，把数据封装到数据包中

String data = "这个是我第一个udp的例子..";

//创建了一个数据包

DatagramPacket packet = new DatagramPacket(data.getBytes(), data.getBytes().length,InetAddress.getLocalHost() , 9090);//发送给了本机

//调用udp的服务发送数据包

datagramSocket.send(packet);

//关闭资源 ---实际上就是释放占用的端口号

datagramSocket.close();

}

}

Demo01UDP(Receive)接收端

package cn.itcast.udp;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

//接收端

/\*

接收端的使用步骤

1. 建立udp的服务。

2. 准备空的数据包接收数据。

3. 调用udp的服务接收数据。

4. 关闭资源。

\*/

public class Demo1Receive {

public static void main(String[] args) throws IOException {

//建立udp的服务 ，并且要监听一个端口。

DatagramSocket socket = new DatagramSocket(9090);

//准备空的数据包用于存放数据。

byte[] buf = new byte[1024];//建立空数组

DatagramPacket datagramPacket = new DatagramPacket(buf, buf.length); // 1024

//调用udp的服务接收数据

socket.receive(datagramPacket); //receive是一个阻塞型的方法，没有接收到数据包之前会一直等待，不会执行下面的代码。 数据实际上就是存储到了byte的字节数组中了。

System.out.println("接收端接收到的数据："+ new String(buf,0,datagramPacket.getLength())); // getLength() 获取数据包存储了几个字节。

//关闭资源

socket.close();

}

}

Demo02（使用udp服务给FeiQ发消息）

package cn.itcast.udp;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

/\*

每个网络程序都有自己所处理的特定格式数据,如果接收到的数据不符合指定的格式，那么就会被当成垃圾数据丢弃。(加密..)

飞Q接收的数据格式：

version: time : sender : ip: flag: content

版本号： 时间 ：发送人 :IP：发送的标识符(32): 真正的内容

在udp协议中，有一个IP地址称作为广播地址，广播地址就是主机号为255地址。

给广播IP地址发送消息的时候，在同一个网络段的机器都可以接收到信息。

192.168.15.255

\*/

//使用udp协议给飞Q发送消息。

public class FeiQDemo {

public static void main(String[] args) throws IOException {

// 建立udp的服务

DatagramSocket socket = new DatagramSocket();

// 准备数据，把数据封装到数据包中

String data = getData("feiQ你好！");

DatagramPacket packet = new DatagramPacket(data.getBytes(),

data.getBytes().length,

InetAddress.getByName("192.168.15.255"), 2425);//使用广播

// 发送数据

socket.send(packet);

// 关闭资源

socket.close();

}

// 把数据拼接成指定格式的数据。注意冒号不能少。

public static String getData(String content) {

StringBuilder sb = new StringBuilder();

sb.append("1.0:");

sb.append(System.currentTimeMillis() + ":");

sb.append("习大大:");

sb.append("192.168.10.1:");

sb.append("32:");

sb.append(content);//此处没冒号

return sb.toString();

}

}

Demo02 UDP (ChatSender)群聊

package cn.itcast.udp;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.SocketException;

import java.net.UnknownHostException;

//群聊发送端

public class ChatSender extends Thread {

@Override

public void run() {

try {

//建立udp的服务

DatagramSocket socket = new DatagramSocket();

//准备数据，把数据封装到数据包中发送

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

String line = null;

DatagramPacket packet = null;

while((line = keyReader.readLine())!=null){

//把数据封装 到数据数据包中，然后发送数据。

packet = new DatagramPacket(line.getBytes(), line.getBytes().length, InetAddress.getByName("192.168.15.255"), 9090);

//把数据发送出去

socket.send(packet);

}

//关闭 资源

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

}

Demo02 UDP (ChatReceive)群聊

package cn.itcast.udp;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.SocketException;

//群聊接收端

public class ChatReceive extends Thread {

@Override

public void run() {

try {

//建立udp的服务,要监听一个端口

DatagramSocket socket = new DatagramSocket(9090);

//准备空的数据包存储数据

byte[] buf = new byte[1024];

DatagramPacket packet = new DatagramPacket(buf, buf.length);

boolean flag = true;

while(flag){

socket.receive(packet);

// packet.getAddress() 获取对方数据包的IP地址对象

System.out.println(packet.getAddress().getHostAddress()+"说:"+new String(buf,0,packet.getLength()));

}

//关闭资源

socket.close();

}catch (IOException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

}

Demo02 UDP (ChatMain)群聊

package cn.itcast.udp;

public class ChatMain {

public static void main(String[] args) {

ChatReceive chatReceive = new ChatReceive();

chatReceive.start();

ChatSender chatSender = new ChatSender();

chatSender.start();

}

}

Demo03(SafeSend)udp通讯协议的不可靠问题

package cn.itcast.udp;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

import java.net.InetAddress;

import java.net.UnknownHostException;

/\*

udp是一个不可靠（数据包可能会丢失）的协议

什么情况下数据包会出现丢失呢？

1.带宽不足 。

2.cpu的处理能力不足。

\*/

public class SafeSender {

public static void main(String[] args) throws Exception {

//建立udp的服务

DatagramSocket socket = new DatagramSocket();

//准备数据，数据封装到数据中发送

DatagramPacket packet = null;

for(int i = 0 ; i< 10; i++){ //连续发送10个数据包，a是为了大于1kb的寄存器

String data =i +"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa";

packet = new DatagramPacket(data.getBytes(), data.getBytes().length, InetAddress.getLocalHost(), 9090);

//发送数据包

socket.send(packet);

}

//关闭资源

socket.close();

}

}

Demo03(SafeReceive)udp通讯协议的不可靠问题

package cn.itcast.udp;

import java.io.IOException;

import java.net.DatagramPacket;

import java.net.DatagramSocket;

public class SafeReceive {

public static void main(String[] args) throws IOException, Exception {

//建立udp的服务

DatagramSocket socket = new DatagramSocket(9090);

//建立空的数据包存储数据

byte[] buf = new byte[1024];

DatagramPacket packet = new DatagramPacket(buf, buf.length);

//不断接收数据包

while(true){

socket.receive(packet);

System.out.println(new String(buf,0,packet.getLength()));

Thread.sleep(10);//模拟cup不足

}

}

}

Demo1Clinet客户端 (tcp通讯协议)

package cn.itcast.tcp;

import java.io.IOException;

import java.io.InputStream;

import java.io.OutputStream;

import java.net.InetAddress;

import java.net.Socket;

import java.net.UnknownHostException;

/\*

UDP通讯协议的特点：

1. 将数据极封装为数据包，面向无连接。

2. 每个数据包大小限制在64K中

3.因为无连接，所以不可靠

4. 因为不需要建立连接，所以速度快

5.udp 通讯是不分服务端与客户端的，只分发送端与接收端。

TCP通讯协议特点：

1. tcp是基于IO流进行数据 的传输 的，面向连接。

2. tcp进行数据传输的时候是没有大小限制的。

3. tcp是面向连接，通过三次握手的机制保证数据的完整性。 可靠协议。

4. tcp是面向连接的，所以速度慢。

5. tcp是区分客户端与服务端 的。

比如： 打电话、 QQ\feiQ的文件传输、 迅雷下载....

tcp协议下的Socket：

Socket(客户端) , tcp的客户端一旦启动马上要与服务端进行连接。

ServerSocket(服务端类)

tcp的客户端使用步骤：

1. 建立tcp的客户端服务。

2. 获取到对应的流对象。

3. 写出或读取数据

4. 关闭资源。

\*/

//tcp客户端

public class Demo1Clinet {

public static void main(String[] args) throws IOException{

//建立tcp的服务

Socket socket = new Socket(InetAddress.getLocalHost(),9090);

//获取到Socket的输出流对象

OutputStream outputStream = socket.getOutputStream();

//利用输出流对象把数据写出即可。

outputStream.write("服务端你好".getBytes());

//获取到输入流对象，读取服务端回送的数据。

InputStream inputStream = socket.getInputStream();

byte[] buf = new byte[1024];

int length = inputStream.read(buf);

System.out.println("客户端接收到的数据："+ new String(buf,0,length));

//关闭资源

socket.close();

}

}

Demo1Server服务端(tcp通讯协议)

package cn.itcast.tcp;

import java.io.IOException;

import java.io.InputStream;

import java.io.InputStreamReader;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

//tcp的服务端

/\*

异常 java.net.BindException: 指端口被占用。

ServerSocket的使用 步骤

1. 建立tcp服务端 的服务。

2. 接受客户端的连接产生一个Socket.

3. 获取对应的流对象读取或者写出数据。

4. 关闭资源。

为什么ServerSocket不设计一个getInputStream与getOutputStream 呢？

\*/

//服务端

public class Demo1Server {

public static void main(String[] args) throws Exception {

//建立Tcp的服务端,并且监听一个端口。

ServerSocket serverSocket = new ServerSocket(9090);

//接受客户端的连接

Socket socket = serverSocket.accept(); //accept() 接受客户端的连接 该方法也是一个阻塞型的方法，没有客户端与其连接时，会一直等待下去。

//获取输入流对象，读取客户端发送的内容。

InputStream inputStream = socket.getInputStream();

byte[] buf = new byte[1024];

int length = 0;

length = inputStream.read(buf);

System.out.println("服务端接收："+ new String(buf,0,length));

//获取socket输出流对象，想客户端发送数据

OutputStream outputStream = socket.getOutputStream();

outputStream.write("客户端你好啊！".getBytes());

//关闭资源

serverSocket.close();

}

}

Demo02(ChatClient)群聊

package cn.itcast.tcp;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.InetAddress;

import java.net.Socket;

/\*

需求： 客户端与服务端一问一答聊天。

1.如果使用BuffrerdReader的readline方法一定要加上\r\n才把数据写出。

2.使用输出字符流一定要调用flush方法数据才会写出。

\*/

//聊天的客户端

public class ChatClient {

public static void main(String[] args) throws IOException {

//建立tcp的客户端服务

Socket socket = new Socket(InetAddress.getLocalHost(),9090);

//获取socket的输出流对象，转为字符流

OutputStreamWriter socketOut = new OutputStreamWriter(socket.getOutputStream());

//获取socket的输入流对象

BufferedReader socketReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

//获取键盘的输入流对象，读取数据

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

String line = null;

//不断的读取键盘录入的数据，然后把数据写出

while((line = keyReader.readLine())!=null){

socketOut.write(line+"\r\n");//不可缺少/r/n，才能使服务端的readLine()辨别到分行才可以读到数据

//输出流的刷新

socketOut.flush();

//读取服务端回送的数据

line = socketReader.readLine();

System.out.println("服务端回送的数据是："+line);

}

//关闭资源

socket.close();

}

}

Demo02(chatServer)群聊

package cn.itcast.tcp;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.ServerSocket;

import java.net.Socket;

/\*

聊天的服务端

\*/

public class ChatServer {

public static void main(String[] args) throws IOException {

//建立tcp的服务端

ServerSocket serverSocket = new ServerSocket(9090);

//接受客户端的连接，产生一个SOcket

Socket socket = serverSocket.accept();

//获取到Socket的输入流对象

BufferedReader socketReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

//获取到Socket输出流对象

OutputStreamWriter socketOut = new OutputStreamWriter(socket.getOutputStream());

//获取键盘的输入流对象

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

//读取客户端的数据

String line = null;

while((line = socketReader.readLine())!=null){

System.out.println("服务端接收到的数据："+ line);

System.out.println("请输入回送给客户端的数据：");

line = keyReader.readLine();

socketOut.write(line+"\r\n");

socketOut.flush();

}

//关闭资源

serverSocket.close();

}

}

Demo03(模拟Tomcat服务器)

package cn.itcast.tcp;

import java.io.IOException;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

//模拟Tomcat服务器

public class TomcatDemo extends Thread {

Socket socket;

public TomcatDemo(Socket socket){

this.socket = socket;

}

public void run() {

try {

//获取socket的输出流对象

OutputStream outputStream = socket.getOutputStream();

//把数据写到浏览器上

outputStream.write("<html><head><title>aaa</title></head><body>你好啊浏览器</body></html>".getBytes());

socket.close();

} catch (IOException e) {

e.printStackTrace();

}

}

public static void main(String[] args) throws IOException {

//建立tcp的服务端

ServerSocket serverSocket = new ServerSocket(9090);

//不断的接受客户端的连接

while(true){

Socket socket = serverSocket.accept();

new TomcatDemo(socket).start();

}

}

}

Day24//复习前面所学//

Demo01(TCP通讯协议)ImageServer

package cn.itcast.tcp;

import java.io.FileInputStream;

import java.io.FileNotFoundException;

import java.io.IOException;

import java.io.OutputStream;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.HashSet;

/\*

1. 编写一个服务端可以给多个客户端发送图片。 （多线程）

\*/

//下载图片服务端

public class ImageServer extends Thread {

Socket socket ;

//使用该集合是用于存储ip地址的，静态共享此集合,hashset不接受重复数据

static HashSet<String> ips = new HashSet<String>();

public ImageServer(Socket socket) {

this.socket = socket;

}

@Override

public void run() {

try {

//获取到socket输出流对象

OutputStream outputStream = socket.getOutputStream();

//获取图片的输入流对象

FileInputStream fileInputStream = new FileInputStream("F:\\美女\\3.jpg");

//读取图片数据，把数据写出

byte[] buf = new byte[1024];

int length = 0 ;

while((length = fileInputStream.read(buf))!=-1){

outputStream.write(buf,0,length);

}

String ip = socket.getInetAddress().getHostAddress(); // socket.getInetAddress() 获取对方的IP地址

if(ips.add(ip)){

System.out.println("恭喜"+ip+"同学成功下载，当前下载的人数是："+ ips.size());//使用hashset，不会接收重复ip

}

//关闭资源

fileInputStream.close();

socket.close();

}catch (IOException e) {

}

}

public static void main(String[] args) throws IOException {

//建立tcp的服务 ,并且要监听一个端口

ServerSocket serverSocket = new ServerSocket(9090);

while(true){

//接受用户端的链接数据。

Socket socket = serverSocket.accept();

new ImageServer(socket).start();

}

}

}

Demo01(TCP通讯协议)ImageClient

package cn.itcast.tcp;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.InputStream;

import java.net.InetAddress;

import java.net.Socket;

import java.net.UnknownHostException;

//下载图片的客户端

public class ImageClient {

public static void main(String[] args) throws Exception{

//建立tcp的服务

Socket socket = new Socket(InetAddress.getLocalHost(),9090);

//获取socket的输入流对象

InputStream inputStream = socket.getInputStream();

//获取文件的输出流对象

FileOutputStream fileOutputStream = new FileOutputStream("F:\\3.jpg");

//边读边写

byte[] buf = new byte[1024];

int length = 0 ;

while((length = inputStream.read(buf))!=-1){

fileOutputStream.write(buf,0,length);

}

//关闭资源

fileOutputStream.close();

socket.close();

}

}

Demo02(TCP通讯协议)LoginClient

package cn.itcast.tcp;

import java.io.BufferedReader;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.InetAddress;

import java.net.Socket;

/\*

2. 实现登陆与注册 功能。

客户端与服务端连接的时候，就要提示客户端请选择功能。

客户端注册的时候，用户名与密码都是发送给服务端 的，服务端需要把数据保存到服务端的文件上。

登陆： 登陆的时候客户端输入用户名与密码发送给服务端，服务端需要校验，返回结果给客户端。

\*/

//客户端

public class LoginClinet {

public static void main(String[] args) throws IOException {

Socket socket = new Socket(InetAddress.getLocalHost(),9090);

//获取socket的输出流对象

OutputStreamWriter socketOut = new OutputStreamWriter(socket.getOutputStream());

//获取到socket的输入流对象

BufferedReader socketReader = new BufferedReader(new InputStreamReader(socket.getInputStream()));

//获取到键盘的输入流对象

BufferedReader keyReader = new BufferedReader(new InputStreamReader(System.in));

while(true){

System.out.println("请选择功能： A(登陆) B(注册)");

String option = keyReader.readLine();

if("a".equalsIgnoreCase(option)){

getInfo(socketOut, keyReader, option);

//读取服务器反馈的信息

String line = socketReader.readLine();

System.out.println(line);

}else if("b".equalsIgnoreCase(option)){

getInfo(socketOut, keyReader, option);

//读取服务器反馈的信息

String line = socketReader.readLine();

System.out.println(line);

}

}

}

public static void getInfo(OutputStreamWriter socketOut,BufferedReader keyReader, String option)

throws IOException {

System.out.println("请输入用户名:");

String userName = keyReader.readLine();

System.out.println("请输入密码：");

String password = keyReader.readLine();

String info = option +" "+userName+" "+password+"\r\n";//加上option一起传到服务端

socketOut.write(info);

socketOut.flush();

}

}

Demo02(TCP通讯协议)LoginServe

//服务端

package cn.itcast.tcp;

import java.io.BufferedReader;

import java.io.File;

import java.io.FileReader;

import java.io.FileWriter;

import java.io.IOException;

import java.io.InputStreamReader;

import java.io.OutputStreamWriter;

import java.net.ServerSocket;

import java.net.Socket;

import java.util.Properties;

public class LoginServer extends Thread {

Socket socket;

static File file = new File("F:\\users.properties");

//构造器

public LoginServer(Socket socket) {

this.socket = socket;

}

static {

try {

if (!file.exists()) {

file.createNewFile();

}

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void run() {

while(true){

try {

// 获取socket的输入流对象

BufferedReader bufferedReader = new BufferedReader(

new InputStreamReader(socket.getInputStream()));

// 获取socket的输出流对象

OutputStreamWriter socketOut = new OutputStreamWriter(

socket.getOutputStream());

// 读取客户端输入的信息

String info = bufferedReader.readLine();

String[] datas = info.split(" ");

// 获取到用户 的选择功能

String option = datas[0];

// 注册

String userName = datas[1];

String password = datas[2];

if ("a".equalsIgnoreCase(option)) {

// 登陆

Properties properties = new Properties();

// 加载配置文件

properties.load(new FileReader(file));

if (properties.containsKey(userName)) {

String tempPass = properties.getProperty(userName);

if (password.equals(tempPass)) {

socketOut.write("欢迎" + userName + "登陆成功\r\n");

} else {

socketOut.write("密码错误\r\n");

}

} else {

socketOut.write("用户名不存在，请重新输入...\r\n");

}

socketOut.flush();

} else if ("b".equalsIgnoreCase(option)) {

// 创建一个配置文件类

Properties properties = new Properties();

//加载原来的配置文件

properties.load(new FileReader(file));

if (!properties.containsKey(userName)) {

// 不存在该用户名

properties.setProperty(userName, password);

// 生成一个配置文件

properties.store(new FileWriter(file), "users");

socketOut.write("注册成功..\r\n");

} else {

// 存在用户名

socketOut.write("用户名已经被注册，请重新输入\r\n");

}

socketOut.flush();

}

} catch (Exception e) {

e.printStackTrace();

}

}

}

public static void main(String[] args) throws IOException {

ServerSocket serverSocket = new ServerSocket(9090);

while (true) {

Socket socket = serverSocket.accept();

new LoginServer(socket).start();

}

}

}

Damo01(三元运算符)

package cn.itcast.review;

/\*

三元运算符:

三元运算符的格式：

布尔表达式?值1:值2

教务系统

学生 缴费

1

0

\*/

public class Demo1 {

public static void main(String[] args) {

String str = "hello";

System.out.println(str.matches("\\w{6}")?"正确":"错误");

int a =3;

System.out.println(3>a?"第一个":"第二个");

int temp = 1;

System.out.println(temp==1?"缴费":"未缴费");

}

}

Demo02(continue,break,return)

package cn.itcast.review;

/\*

continue: 跳过本次循环语句，继续下一次循环。

continue的作用范围： 只能适用于循环语句。

一旦执行了continue语句，那么在循环体内continue之后的循环语句跳过执行。

break: 用于结束当前的一个循环语句或者是一个switch语句.

break作用范围： 只能用于循环语句或者是switch语句。

return

\*/

public class Demo2 {

public static void main(String[] args) {

/\*

\* continue

\* for(int i = 0 ; i<5 ; i++){

if(i==1){

continue;//跳过1，之后的依然显示

}

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

}

\*/

//break//后面的代码不执行了

/\*outer:for(int j = 0 ; j<2; j++){ // j=0 j=1

inner:for(int i = 0 ; i<3 ; i++){ //i=0

System.out.println("hello world");

break outer; //默认是结束当前所在的循环。 如果配合标识的使用，可以作用于外层的for循环。

}

}\*/

for(int i = 0 ; i<3 ; i++){

System.out.println("hello world");

return;

}

System.out.println("哈哈");

}

}

Demo03(函数重载)

package cn.itcast.review;

/\*

函数的重载: 在一个类中出现了两个 或两个以上 的同名函数称作为函数的重载。

函数重载的作用： 一个函数名可以应对各种不同类型的参数。

函数重载的要求

1. 函数名一致。

2. 形参列表不一致（形参的个数不一致或者是形参对应 的类型不一致）

3. 与返回值类型无关。

函数重写：子父类出现了同名的函数称作为函数的重写。

函数重写：必须要是在继承的关系才存在的。

函数重写的需求： 父类的功能无法满足子类的需求。

函数重写的要求

1. 子父类的函数名eat与形参列表(int num)必须一致。

2. 子类的权限修饰符必须 要大于或者等于父类的权限修饰符。

3. 子类的返回值类型必须要小于或者等于父类的返回类型。

4. 子类抛出的异常类型必须要小于或者等于父类抛出的异常类型。

\*/

class Animal{

}

class Cat extends Animal{}

class Fu{

public Animal eat(int num){

System.out.println("父类的eat方法");

return new Animal();

}

}

class Zi extends Fu{

public Animal eat(int num){

System.out.println("子类的eat方法..");

return new Animal();

}

}

public class Demo3 {

public static void main(String[] args) {

/\*int[] arr = {1,2,4};

double[] arr2 = {2.14,3.14,5.0};

sort(arr);

sort(arr2);\*/

Zi z = new Zi();

z.eat(11);

}

public static void sort(int[] arr){

for(int i = 0 ; i< arr.length-1; i++){

for(int j = i+1 ; j<arr.length ; j++){

if(arr[i]>arr[j]){

int temp = arr[i];

arr[i] = arr[j];

arr[j]= temp;

}

}

}

}

public static void sort(double[] arr){

for(int i = 0 ; i< arr.length-1; i++){

for(int j = i+1 ; j<arr.length ; j++){

if(arr[i]>arr[j]){

double temp = arr[i];

arr[i] = arr[j];

arr[j]= temp;

}

}

}

}

}

Demo04（冒泡排序）

package cn.itcast.review;

import java.util.Arrays;

/\*

冒泡排序 ： 相邻的两个元素比较，符合条件交换位置。

\*/

public class Demo4 {

public static void main(String[] args) {

int[] arr = {2,5,10,8,1}; //所谓的排序就是将一个个的最大元素放在对应 的位置上。

for(int j = 0 ; j<arr.length -1; j++){ // 控制的轮数，每一轮都可以找出一个最大值。

//把最大值放在最后一个位置

for(int i = 0 ; i<arr.length-1-j ; i++){ //控制的是相邻的两个元素比较。

//相邻的两个元素比较

if(arr[i] >arr[i+1]){

int temp =arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

}

/\* //把老二放在倒数第二个位置上

for(int i = 0 ; i<arr.length-1-1 ; i++){

//相邻的两个元素比较

if(arr[i] >arr[i+1]){

int temp =arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

//把老三放在倒数第三个位置上

for(int i = 0 ; i<arr.length-1-2 ; i++){

//相邻的两个元素比较

if(arr[i] >arr[i+1]){

int temp =arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

//把老四放在倒数第四个位置上

for(int i = 0 ; i<arr.length-1-3 ; i++){

//相邻的两个元素比较

if(arr[i] >arr[i+1]){

int temp =arr[i];

arr[i] = arr[i+1];

arr[i+1] = temp;

}

}

\*/

System.out.println("数组的元素： "+Arrays.toString(arr));

}

}

Demo05(折半查找法)

package cn.itcast.review;

/\*

折半查找法(二分法) ：折半查找法使用的前提是数据必须 是有序的。

\*/

public class Demo5 {

public static void main(String[] args) {

int[] arr = {12,15,17,19,30};

int index = halfSearch(arr,0);

System.out.println("idnex:"+ index);

}

public static int halfSearch(int[] arr,int target){

int max = arr.length - 1;

int min = 0;

int mid = (max+min)/2;

while(true){

if(target>arr[mid]){

min = mid+1;

}else if(target<arr[mid]){

max = mid-1;

}else{

return mid;

}

//重新计算中间值

mid = (min+max)/2;

//找不到

if(max<min){

return -1;

}

}

}

}

Demo06(面向对象)

package cn.itcast.review;

class Person{

int id;

String name;

@Override

public String toString() {

return "编号："+this.id+" 姓名："+this.name;

}

}

public class Demo6 {

public static void main(String[] args) {

Person p = new Person();

p.id= 110;

p.name = "狗娃";

System.out.println(p);

}

}

Demo07(匿名对象)

package cn.itcast.review;

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

/\*

匿名对象： 没有引用类型变量指向的对象就称作为匿名对象。

匿名对象的主要作用： 简化书写。

匿名对象主要用于两种应用场景：

1. 如果一个对象的方法只会调用一次的时候，然后该对象就不再使用了，这时候就就可以使用匿名对象。

2. 作为参数传递。

\*/

public class Demo7 {

public static void main(String[] args) throws IOException {

// Runtime runtime = Runtime.getRuntime();

// Runtime.getRuntime().exec("C:\\Windows\\notepad.exe");

// FileReader fileReader = new FileReader("F:\\a.txt");

BufferedReader bufferedReader = new BufferedReader(new FileReader("F:\\a.txt"));

}

}

Demo08(封装)

package cn.itcast.review;

/\*

封装：

封装的步骤：

1. 私有化要封装的属性。

2. 根据需求提供对应的get或者是set方法。

封装的好处：

1. 提高数据的安全性。

2. 操作简单。

3. 隐藏实现。

封装一定会用的场景：如果是实体类的成员属性，我们在现实开发中全部都封装起来。

\*/

class Member{

private String name;

private String sex;

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public void setSex(String sex){

if("男".equals(sex)||"女".equals(sex)){

this.sex = sex;

}

}

public String getSex(){

return sex;

}

}

public class Demo8 {

public static void main(String[] args) {

Member m = new Member();

m.setName("狗娃");

m.setSex("男");

}

}

Demo09(构造代码块)

package cn.itcast.review;

/\*

构造代码块：给对象进行统一的初始化工作。

应用场景： 如何创建任意对象的时候都需要调用某个方法为该对象进行初始化时，这时候就可以使用构造代码块。

静态代码块:

静态代码块是静态代码块所属的类被加载到内存的时候执行的。

静态代码块的应用场景： 以后主要用于准备一个项目的初始化工作。

比如： 从配置配置文件中读取数据库用户名与密码。

\*/

class Baby{

int id;

String name;

//构造代码块的代码其实是在构造函数中执行的。

{

cry();

}

static{

System.out.println("静态代码块执行了...");

}

public Baby(int id, String name) {

this.id = id;

this.name = name;

}

public Baby(){

}

public void cry(){

System.out.println("哭...");

}

@Override

public String toString() {

return " 编号："+this.id+" 姓名："+ this.name;

}

}

public class Demo9 {

public static void main(String[] args) {

Baby b1 = new Baby();

Baby b2 = new Baby(110, "狗娃");

}

}

Demo10(继承instanceof)

package cn.itcast.review;

/\*

instanceof 判断指定的对象是否属于某种类别。

instanceof 使用前提： 判断的对象与类别必须存在继承或者实现的关系。

instanceof 使用格式：

对象 instanceof 类别。

有多态才可能使用到instanceof关键字，

主要的应用场景： 数据类型强转之前的判断。

\*/

class Fish extends Animal{

}

class Mouse extends Animal{}

public class Demo10 {

public static void main(String[] args) {

/\*Fish f = new Fish();

Animal a = new Animal();

System.out.println("鱼是属于动物类吗？"+ (f instanceof Animal));

System.out.println("鱼是属于老鼠吗?"+ (a instanceof Fish));\*/

Animal a = getAnimal(1);

if(a instanceof Fish){

Fish f =(Fish) a;

}else if(a instanceof Mouse){

Mouse m = (Mouse)a;

}

}

public static Animal getAnimal(int i){

if(i==0){

return new Fish();

}else{

return new Mouse();

}

}

}

Demo11(抽象类)

package cn.itcast.review;

import java.util.ArrayList;

import java.util.LinkedList;

/\*

抽象类的应用场景：描述一类事物的时候，如果该类确实存在某种行为，但是目前这种行为的是不具体的

这时候就可以把这种行为描述成抽象 的行为，这时候使用抽象。

抽象类在现实开发中的应用场景：

父类私有的成员不能被继承。

继承的问题： 隐式继承。

\*/

//工作

class Worker{

private String name;

public void setInfo(String name){

this.name = name;

System.out.println("姓名："+ this.name);

}

}

class Teacher extends Worker{

}

public class Demo11 {

public static void main(String[] args) {

Teacher t = new Teacher();

//t.name ="狗娃";

t.setInfo("老钟");

}

}

Demo12(多态)

package cn.itcast.review;

import java.util.ArrayList;

import java.util.Collections;

import java.util.LinkedList;

import java.util.List;

/\*

多态： 父类的引用类型变量指向了子类的对象，或者是接口的引用类型变量指向接口实现类的对象。

应用：

1. 多态应用于形参类型的时候，可以接收更多类型的参数，

sort(List list)

sort(ArrayList list)

sort(LinkedList list)

2. 多态用于返回值类型的时候可以返回更多类型的参数。

迭代器的作用： 用于获取集合中的元素。

内部类：

内部类的好处： 内部类可以直接访问外部类的成员。

\*/

interface MyList{

List subList(int fromIndex, int toIndex);

}

public class Demo12 {

public static void main(String[] args) {

/\*Collections.sort(new ArrayList());

Collections.sort(new LinkedList());\*/

ArrayList<Integer> list = new ArrayList<Integer>();

list.add(10);

list.add(19);

list.add(12);

list.add(6);

Iterable<Integer> it = (Iterable<Integer>) list.iterator();

List<Integer> list2 = list.subList(1, 3);

}

}

Demo13(匿名内部类)

package cn.itcast.review;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

/\*

匿名对象：没有引用类型变量指向的对象就称作为匿名对象。

匿名内部类：没有类名的类就称作为匿名内部类。

匿名内部类使用前提： 必须存在继承或者实现关系。

因为匿名内部类没有类名，所以创建匿名内部类对象的时候就必须要借助与它父类的名字或者它父接口的名字来创建。

但是匿名内部类只不过是没有类名，其他的一概成员都是具备的。

匿名内部类的应用场景： 主要是作为参数传递使用。

\*/

interface Dao{

public void add();

}

class Outer{

public void print(){

new Dao(){ //这里不是创建接口 的对象，是创建了Dao实现类的对象，但是这个类是一个匿名内部类而已，没有类名借用了父接口 的名字而已。

//大括号中就写匿名内部类的成员。

int num =10;

@Override

public void add() {

System.out.println("添加学生 ..");

}

}.add();

}

}

public class Demo13 {

public static void main(String[] args) {

/\*Outer outer = new Outer();

outer.print();\*/

JButton button = new JButton("aa");

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

System.out.println("aaa");

}

});

}

}

Demo14(多线程)

package cn.itcast.review;

/\*

线程：

多线程的存在的意义： 解决了一个进程允许多个任务可以同时执行。

多线程的创建方式：

方式一: 继承Thread。

1. 自定义一个类继承Thread.

2. 重写Thread的run方法，把自定义线程的任务代码放在run方法上。

3. 创建Thread类的子类对象，并且调用start方法开启线程。

方式二： 实现Runnable接口。

1. 自定义一个类实现Runnable接口。

2. 实现Runnable的run方法,把自定义线程的任务代码放在run方法上。

3. 创建Runnable实现类的对象。

4. 创建Thread的对象，然后把Runnable实现类的对象作为参数传递。

5. 调用Thread对象的start方法开启线程。

java中的同步机制：

出现线程安全问题的根本原因：

1. 存在两个或者两个以上的线程共享着资源。

2. 操作资源的代码块必须有语句。

1. 同步代码块

同步代码块的格式：

synchronized(锁对象){

需要被同步的代码块...

}

同步代码块要注意的细节：

1.锁对象可以是任意的对象。

2. 锁对象必须 是多线程共享的资源。否则锁不住。

3. 没有线程安全问题的时候不要使用锁，因为会导致效率降低。

4. 调用sleep方法并不会释放锁对象，但是调用wait方法的线程就会释放锁对象。

2. 同步函数

修饰符 synchronized 返回值类型 函数名(形参列表){

}

注意：

1. 同步函数的锁对象是不能任意的，非静态同步函数的锁对象是this对象，静态函数的锁对象是当前字节码对象。

2. 同步函数的锁不能由你指定，是固定的。

\*/

class MyThread2 implements Runnable{

@Override

public void run() {

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

System.out.println(Thread.currentThread().getName()+":"+i);

}

}

}

class MyThread extends Thread{

@Override

public void run() {

//把自定义线程的任务代码代码写在这里。。

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

System.out.println(Thread.currentThread().getName()+":"+i);

}

}

}

public class Demo14 {

public static void main(String[] args) {

MyThread thread1 = new MyThread();

thread1.start(); //开启了自定义的线程。线程一旦开启就会执行run方法中的代码块。

MyThread2 thread2 = new MyThread2();

//(runable方法)创建Thread的对象

Thread t = new Thread(thread2);

//调用Thread对象的start方法

t.start();

}

}

Demo15(集合)

package cn.itcast.review;

import java.util.HashSet;

import java.util.Iterator;

/\*

集合

单例集合

----------| Collection 单列集合的根接口

----------------| List 如果是实现了List接口集合类具备的特点： 有序，可重复。

-------------------| ArrayList 底层使用Object数组实现的。 特点：查询速度快，增删慢。

-------------------| LinkedList 底层是使用了链表数据数据结构实现的。 特点： 查询慢，增删快。

-------------------| Vector(了解) 底层使用Object数组实现的， 实现与ArrayList是一样，只不过是线程安全的，操作效率低。

----------------| Set 如果是实现了Set接口集合类具备的特点： 无序，不可重复。

------------------| HashSet 底层使用的是哈希表实现的。

------------------| TreeSet 底层使用二叉数实现。

双列集合：

--------| Map (只需要把Map接口的方法全部练习一次即可。)

-----------| HashMap 底层使用的是哈希表实现的。

-----------| TreeMap 底层使用二叉数实现

-----------| HashTable（了解）

\*/

class Book{

String name;

double price;

public Book(String name, double price) {

super();

this.name = name;

this.price = price;

}

@Override

public int hashCode() {

return this.name.hashCode();

}

@Override

public boolean equals(Object obj) {

Book b = (Book)obj;

return this.name.equals(b.name);

}

@Override

public String toString() {

return "[书名："+ this.name+" 价格："+this.price+"]";

}

}

public class Demo15 {

public static void main(String[] args) {

//不允许重复的书名存在。

HashSet<Book> books = new HashSet<Book>();

books.add(new Book("深入javaweb",34));

books.add(new Book("java神书",78));

//修改书名

Iterator<Book> it = books.iterator();

while(it.hasNext()){

Book b = it.next();

if(b.name.equals("java神书")){

b.name = "java编程思想";

}

}

//为什么修改名字之后不能删除了呢?

books.remove(new Book("java神书",78));

System.out.println("集合的元素："+ books);

}

}

Demo16(TreeMap)

package cn.itcast.review;

import java.util.Scanner;

import java.util.TreeMap;

/\*

从键盘输入一个字母组成字符串，分别统计每个字母出现的次数(10分)

要求输出的效果按照字母的顺序输出 a(7)b(5)...

\*/

public class Demo16 {

public static void main(String[] args) {

System.out.println("请输入一段字符串：");

Scanner scanner = new Scanner(System.in);

String line = scanner.next();

char[] arr = line.toCharArray(); //先把字符串转换成字符数组。

TreeMap<Character, Integer> map = new TreeMap<Character, Integer>();

for(char c : arr){

if(map.containsKey(c)){ //map集合已经包含了该字符

int count = map.get(c);

map.put(c, count+1);

}else{ //没有包含

map.put(c, 1);

}

}

System.out.println(map);

}

}

Day25//JFrame,JOptionPane,FileDialog,JPanel//布局管理器//非容器组件//菜单组件//事件//

Demo01(JFrame)

package cn.itcast.frame;

import java.awt.Dimension;

import java.awt.Toolkit;

import javax.swing.JFrame;

/\*

软件的交互方式：

1. 控制台的交互方式、

2. 图形化界面的交互方式 。

java使用到的图形类主要在java.awt 与javax.swing包中。

java.awt 与 javax.swing包的区别：

java.awt中使用的图形类都是依赖于系统 的图形库的。

javax.swing包使用到的图形类都是sun自己实现，不需要依赖系统的图形库。

疑问： 既然swing包中的图形类已经取代awt包的图形类，为什么不删除awt呢？

在java中所有的图形类都被称作组件类。

组件的类别：

----------| 容器组件

----------| 非容器组件

\*/

public class Demo1 {

public static void main(String[] args) {

JFrame frame = new JFrame("这个是我第一个图形化界面的例子");

//设置窗体的大小

// frame.setSize(300,400);

//设置窗体(左上角)出现的位置

//frame.setBounds((1366-300)/2, (768-400)/2, 300,400); // 第一个参数是左上角的x轴坐标， 第二参数是左上角y的坐标。 第三个窗体宽， 第四窗体的高。

initFrame(frame, 300,400);

frame.setVisible(true); //setVisible 设置窗体的可见性。

//设置窗体关闭事件

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

}

//获取屏幕的分辨率 设置窗体在屏幕的居中位置。

public static void initFrame(JFrame frame,int width , int height){

Toolkit toolkit = Toolkit.getDefaultToolkit(); //获取一个与系统相关工具类对象

//获取屏幕的分辨率

Dimension d = toolkit.getScreenSize();

int x = (int) d.getWidth();

int y = (int) d.getHeight();

frame.setBounds((x-width)/2, (y-height)/2, width, height);

}

}

Demo02(JDialog)

package cn.itcast.frame;

import java.awt.Dialog;

import javax.swing.JDialog;

import javax.swing.JFrame;

import javax.swing.JOptionPane;

import cn.itcast.util.FrameUtil;

/\*

对话框类(Dialog):

JDialog(Dialog owner, String title, boolean modal)

owner: 所有者

title : 标题

modal : modal

JOptionPane(对话框)

消息对话框

警告对话框

错误对话框

输入对话框

确认对话框

\*/

public class Demo2 {

public static void main(String[] args) {

/\*JFrame frame = new JFrame("窗体");

//创建对话框

JDialog dialog = new JDialog(frame, "对话框",true);

//使用我自定义的 窗体工具类

FrameUtil.initFrame(frame, 300, 400);

dialog.setBounds(500,300, 100, 200);

dialog.setVisible(true); //设置对话框的可见性

\*/

JFrame frame = new JFrame("窗体");

//显示一个对话框

FrameUtil.initFrame(frame, 300, 400);

//消息对话框

/\*JOptionPane.showMessageDialog(frame, "明天不用上课", "通知",JOptionPane.INFORMATION\_MESSAGE);

//警告对话框

JOptionPane.showMessageDialog(frame,"警告某位同学晚上晚自习不要看动漫","警告",JOptionPane.WARNING\_MESSAGE);\*/

//错误对话框

// JOptionPane.showMessageDialog(frame,"扣6分","错误",JOptionPane.ERROR\_MESSAGE);

//输入框

/\* String moeny = JOptionPane.showInputDialog("请输入你要给我的金额($)");

System.out.println("money:"+ moeny);\*/

//确认框

int num = JOptionPane.showConfirmDialog(frame, "你确认要卸载吗？");

System.out.println(num);

}

}

Demo03(FileDialog)

package cn.itcast.frame;

import java.awt.FileDialog;

import javax.swing.JFrame;

import cn.itcast.util.FrameUtil;

/\*

文件对话框(FileDialog)：

FileDialog(Dialog parent, String title, int mode)

parent： 对话框的所有者

tiltle : 对话框的标题

mode: load(打开) 、 save(保存)

\*/

public class Demo3 {

public static void main(String[] args) {

JFrame frame = new JFrame("窗体");

//创建一个文件对话框(初始也是不可见)

//FileDialog fileDialog = new FileDialog(frame, "请选择打开的文件", FileDialog.LOAD);

FileDialog fileDialog = new FileDialog(frame,"请选择保存的路径",FileDialog.SAVE);

FrameUtil.initFrame(frame, 300,400);

fileDialog.setVisible(true);//设置可见

System.out.println("文件所在的目录："+ fileDialog.getDirectory());

System.out.println("文件名："+ fileDialog.getFile());

}

}

Demo04(JPanel)

package cn.itcast.frame;

import java.awt.Color;

import javax.swing.JFrame;

import javax.swing.JPanel;

import cn.itcast.util.FrameUtil;

/\*

面板(JPanel)

\*/

public class Demo4 {

public static void main(String[] args) {

JFrame frame = new JFrame("窗体");

//创建一个面板

JPanel panel = new JPanel();

//设置面板的背景颜色

panel.setBackground(Color.RED);

//把面板添加到窗体

frame.add(panel);

FrameUtil.initFrame(frame, 400, 300);

}

}

Demo01(非容器组件)

package cn.itcast.compnent;

import javax.swing.ButtonGroup;

import javax.swing.JCheckBox;

import javax.swing.JComboBox;

import javax.swing.JFrame;

import javax.swing.JLabel;

import javax.swing.JPanel;

import javax.swing.JPasswordField;

import javax.swing.JRadioButton;

import javax.swing.JTextArea;

import javax.swing.JTextField;

import cn.itcast.util.FrameUtil;

/\*

非容器组件:

\*/

public class Demo1 {

public static void main(String[] args) {

JFrame frame= new JFrame("注册");

//创建一个面板

JPanel panel = new JPanel();

frame.add(panel);

//用户名

JLabel nameLabel = new JLabel("用户名");

//用户名的输入框

JTextField nameField = new JTextField(12);

//把用户名的组件添加到面板上

panel.add(nameLabel);

panel.add(nameField);

//密码

JLabel passLabel= new JLabel("密码");

//密码框

JPasswordField passField = new JPasswordField(12);

//把密码的组件添加到面板

panel.add(passLabel);

panel.add(passField);

//性别--单选框

JLabel sexLabel = new JLabel("性别");

JRadioButton man = new JRadioButton("男",true);//默认选中

JRadioButton woman = new JRadioButton("女");

//如果是单选框必须要进行分组，同一个组的单选框只能选择其中的一个

ButtonGroup group = new ButtonGroup();

group.add(woman);

group.add(man);

//把性别组件添加到面板上

panel.add(sexLabel);

panel.add(man);

panel.add(woman);

//来自城市--->下拉框

JLabel cityLabel = new JLabel("来自的城市");

Object[] arr = {"北京","上海","广州","深圳","湛江"};

JComboBox citys = new JComboBox(arr);

panel.add(cityLabel);

panel.add(citys);

//兴趣爱好---->复选框

JLabel hobitLabel = new JLabel("兴趣爱好:");

JCheckBox checkBox1 = new JCheckBox("篮球",true);

JCheckBox checkBox2 = new JCheckBox("java",true);

JCheckBox checkBox3 = new JCheckBox("javascript");

JCheckBox checkBox4 = new JCheckBox("android");

panel.add(hobitLabel);

panel.add(checkBox1);

panel.add(checkBox2);

panel.add(checkBox3);

panel.add(checkBox4);

//个人简介

JLabel jLabel = new JLabel("个人简介");

JTextArea area = new JTextArea(20, 15);

area.setLineWrap(true); //设置自动换行

panel.add(jLabel);

panel.add(area);

FrameUtil.initFrame(frame, 500, 400);

}

}

Demo02(菜单组件)

package cn.itcast.compnent;

import java.awt.BorderLayout;

import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

/\*

菜单组件

菜单条(MenuBar) 、 菜单（Menu） 、 菜单项(MenuItem)

菜单条可以添加菜单

菜单可以添加菜单项

复选菜单：

首先菜单添加菜单 ， 菜单添加菜单项。

\*/

import javax.swing.JTextArea;

import cn.itcast.util.FrameUtil;

public class Demo2 {

JFrame frame = new JFrame("记事本");

//菜单条

JMenuBar bar = new JMenuBar();

//文件菜单

JMenu fileMenu = new JMenu("文件");

JMenu editMenu = new JMenu("编辑");

JMenu switchMenu = new JMenu("切换工作目录");

//菜单项

JMenuItem openMenu = new JMenuItem("打开");

JMenuItem saveMenu = new JMenuItem("保存");

JMenuItem aboutMenu = new JMenuItem("关于");

JMenuItem closeMenu = new JMenuItem("关闭");

JMenuItem workMenu1 = new JMenuItem("0910project");

JMenuItem workMenu2 = new JMenuItem("1208project");

JMenuItem workMenu3 = new JMenuItem("1110project");

//编辑区

JTextArea area = new JTextArea(20,30);

public void initNotepad(){

//菜单添加菜单项目

fileMenu.add(openMenu);

fileMenu.add(saveMenu);

editMenu.add(aboutMenu);

editMenu.add(closeMenu);

//复选菜单

switchMenu.add(workMenu1);

switchMenu.add(workMenu2);

switchMenu.add(workMenu3);

//菜单添加菜单就是复选菜单

fileMenu.add(switchMenu);

//菜单条添加菜单

bar.add(fileMenu);

bar.add(editMenu);

//添加菜单条

frame.add(bar,BorderLayout.NORTH);

frame.add(area);

FrameUtil.initFrame(frame, 500, 600);

}

public static void main(String[] args) {

new Demo2().initNotepad();

}

}

Demo01(BorderLayout)

package cn.itcast.layout;

import java.awt.BorderLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import cn.itcast.util.FrameUtil;

/\*

布局管理器:布局管理就是用于指定组件的 摆放位置的。

每种布局管理器都有自己的摆放风格

BorderLayout(边框布局管理器)

摆放的风格： 上北 、 下南 、 左西、 右东 ， 中

Borderlayout 要注意的事项：

1. 使用Borderlayout添加组件的时候，如果没有指定组件的方位，那么默认添加到中间的位置上。

2. 使用BorderLayout的时候，如果东南西北那个方向没有对应 的组件，那么中间位置的组件就会占据其空缺的位置。

3. 窗体默认的布局管理器就是Borderlayout.

\*/

public class Demo1 {

public static void main(String[] args) {

JFrame frame = new JFrame("边框局部管理器");

//创建一个边框布局管理器

BorderLayout borderLayout = new BorderLayout();

//让borderlayout管理frame窗体。

frame.setLayout(borderLayout);

frame.add(new JButton("北"),BorderLayout.NORTH);

frame.add(new JButton("南"),BorderLayout.SOUTH);

frame.add(new JButton("西"),BorderLayout.WEST);

frame.add(new JButton("东"),BorderLayout.EAST);

frame.add(new JButton("中"),BorderLayout.CENTER);

//初始化窗体

FrameUtil.initFrame(frame, 300, 300);

}

}

Demo02(FlowLayout)

package cn.itcast.layout;

import java.awt.FlowLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import cn.itcast.util.FrameUtil;

/\*

流式布局管理器(FlowLayout)

流式布局管理器要注意的事项：

1. 流式布局管理器默认情况是居中对齐的。

2. panel默认的局部管理器就是FlowLayout.

\*/

public class Demo2 {

public static void main(String[] args) {

JFrame frame = new JFrame("窗体");

//创建面板

JPanel panel = new JPanel();

frame.add(panel);

//创建一个流式布局管理器

FlowLayout flowLayout = new FlowLayout(FlowLayout.LEFT, 0, 30);// FlowLayout.LEFT 指定对齐的方式。

//让流式布局管理器管理frame窗体

panel.setLayout(flowLayout);

panel.add(new JButton("按钮1"));

panel.add(new JButton("按钮2"));

panel.add(new JButton("按钮3"));

panel.add(new JButton("按钮4"));

//初始化窗体

FrameUtil.initFrame(frame, 300, 300);

}

}

Demo03(GridLayout)

package cn.itcast.layout;

import java.awt.GridLayout;

import javax.swing.JButton;

import javax.swing.JFrame;

import cn.itcast.util.FrameUtil;

/\*

表格布局管理器(GridLayout)

注意的事项： 如果表格数量不够使用时，默认会多加一列。

\*/

public class Demo3 {

public static void main(String[] args) {

JFrame frame = new JFrame("计算器");

//创建表格布局管理器

GridLayout gridLayout = new GridLayout(4, 4, 1, 2);

//让窗体交给表格布局管理器管理

frame.setLayout(gridLayout);

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

frame.add(new JButton(i+""));

}

frame.add(new JButton("+"));

frame.add(new JButton("-"));

frame.add(new JButton("\*"));

frame.add(new JButton("/"));

frame.add(new JButton("="));

frame.add(new JButton("."));

// frame.add(new JButton("aa"));

//初始化窗体

FrameUtil.initFrame(frame, 300, 300);

}

}

Demo04(CardLayout)

package cn.itcast.layout;

import java.awt.CardLayout;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import cn.itcast.util.FrameUtil;

/\*

卡片布局管理器(CardLayout)

\*/

public class Demo4 {

public static void main(String[] args) {

JFrame frame = new JFrame("卡片布局管理器");

final JPanel panel = new JPanel();

frame.add(panel);

//创建一个卡片布局管理器

final CardLayout cardLayout = new CardLayout();

panel.setLayout(cardLayout);

//往面板添加数据

JButton button = new JButton("黑桃A");

panel.add(button);

panel.add(new JButton("红桃K"));

panel.add(new JButton("梅花6"));

panel.add(new JButton("方块2"));

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

cardLayout.next(panel); //下一张

// cardLayout.previous(parent); 上一张

}

});

//初始化窗体

FrameUtil.initFrame(frame,300, 300);

}

}

Demo01(动作事件监听器)

package cn.itcast.event;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import javax.swing.JButton;

import javax.swing.JFrame;

/\*

事件： 当发生了某个事件的时候，就会有相应处理方案。

事件源 监听器 事件 处理方案

\*/

import cn.itcast.util.FrameUtil;

public class Demo1 {

public static void main(String[] args) {

JFrame frame = new JFrame("窗体");

JButton button = new JButton("点我啊");

frame.add(button);

//给按钮添加动作监听器 动作时间监听器对于鼠标点击以及空格键都会起作用的。

button.addActionListener(new ActionListener() {//匿名内部类

//当按钮被点击的时候，就会调用actionPerformed的方法。

@Override

public void actionPerformed(ActionEvent e) { // ActionEvent 当前按钮被点击的时候，jvm就会把对应的事件传递ActionEvent e中，并且调用actionPerformed方法。

//System.out.println("哎呀，被点了...");

JButton button =(JButton) e.getSource(); //getSource() 获取到事件源

if(button.getText().equals("点我啊")){

button.setText("点他吧");//改变按键文字

}else{

button.setText("点我啊");

}

}

});

FrameUtil.initFrame(frame, 200, 200);

}

}

Demo02(鼠标事件监听器)

package cn.itcast.event;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.awt.event.MouseListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import cn.itcast.util.FrameUtil;

/\*

鼠标事件监听器

\*/

public class Demo2 {

public static void main(String[] args) {

JFrame frame = new JFrame("鼠标事件监听器");

JButton button = new JButton("按钮");

frame.add(button);

//给按钮添加鼠标事件监听器

/\*button.addMouseListener(new MouseListener() {

@Override

public void mouseReleased(MouseEvent e) {

System.out.println("鼠标松开...");

}

@Override

public void mousePressed(MouseEvent e) {

System.out.println("鼠标按下..");

}

@Override

public void mouseExited(MouseEvent e) {

System.out.println("鼠标移出...");

}

@Override

public void mouseEntered(MouseEvent e) {

System.out.println("鼠标进入...");

}

@Override

public void mouseClicked(MouseEvent e) {

System.out.println("鼠标单击..");

}

});

添加鼠标监听器的时候我只使用到单击事件，但是目前要我实现所有的方法？？

解决方案： 适配器 适配器是实现了MouseListener方法的所有方法，但是实现的方法全部都是空实现。

\*/

button.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

// System.out.println("鼠标单击了..");

if(e.getClickCount()==2){

System.out.println("双击了..");

}

}

});

FrameUtil.initFrame(frame, 300, 300);

}

}

Demo03(键盘事件监听器)

package cn.itcast.event;

import java.awt.event.KeyAdapter;

import java.awt.event.KeyEvent;

import java.awt.event.KeyListener;

import javax.swing.JButton;

import javax.swing.JFrame;

import cn.itcast.util.FrameUtil;

/\*

键盘事件监听器

\*/

public class Demo3 {

public static void main(String[] args) {

JFrame frame = new JFrame("键盘事件监听器");

JButton button = new JButton("按钮");

frame.add(button);

//给按钮添加键盘事件监听器

/\*button.addKeyListener(new KeyListener() {

@Override

public void keyTyped(KeyEvent e) {

System.out.println("键入某个键");

}

@Override

public void keyReleased(KeyEvent e) {

// System.out.println("释放某个键");

}

@Override

public void keyPressed(KeyEvent e) {

System.out.println("按下某个键..");

}

});\*/

button.addKeyListener(new KeyAdapter() {

@Override

public void keyPressed(KeyEvent e) {

// System.out.println("按下的字符："+e.getKeyChar());//获取按下的字符

//System.out.println("获取键对应的数值："+ e.getKeyCode());

int code = e.getKeyCode();

switch (code) {

case 38:

System.out.println("上");

break;

case 40:

System.out.println("下");

break;

case 37:

System.out.println("左");

break;

case 39:

System.out.println("右");

break;

default:

break;

}

}

});

FrameUtil.initFrame(frame,300, 300);

}

}

Demo04（练习FileSearch）

package cn.itcast.event;

import java.awt.BorderLayout;

import java.awt.ScrollPane;

import java.awt.Scrollbar;

import java.awt.TextField;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.awt.event.MouseAdapter;

import java.awt.event.MouseEvent;

import java.io.File;

import javax.swing.JButton;

import javax.swing.JFrame;

import javax.swing.JPanel;

import javax.swing.JScrollBar;

import javax.swing.JTextArea;

import javax.swing.JTextField;

import cn.itcast.util.FrameUtil;

public class FileSearch {

JFrame frame = new JFrame("文件搜索器");

JPanel panel = new JPanel();

JTextField field = new JTextField("请输入目录名...",15);

JButton button = new JButton("搜索");

JTextArea area = new JTextArea(15,15);

//滚动条

ScrollPane bar = new ScrollPane();

public void init(){

//先把area添加 到滚动条上。

bar.add(area);

//先把组件添加到panel上

panel.add(field);

panel.add(button);

//给输入框添加事件

field.addMouseListener(new MouseAdapter() {

@Override

public void mouseClicked(MouseEvent e) {

JTextField field = (JTextField) e.getSource();

if(field.getText().equals("请输入目录名...")){

field.setText("");

}

}

});

//给按钮添加事件监听器

button.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

//获取输入框输入的路径

String path = field.getText();

//使用输入的路径构建一个FIle对象

File dir = new File(path);

//找到目录下的所有子 文件

File[] files = dir.listFiles();

for(File file : files){ // 1208Project 资料

area.setText(area.getText()+ file.getName()+"\r\n");

}

}

});

//把面板添加到frame上

frame.add(panel,BorderLayout.NORTH);

frame.add(bar);

FrameUtil.initFrame(frame, 300, 400);

}

public static void main(String[] args) {

new FileSearch().init();

}

}

Demo05(练习Notepad)

package cn.itcast.event;

import java.awt.BorderLayout;

import java.awt.FileDialog;

import java.awt.TextArea;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.io.File;

import java.io.FileNotFoundException;

import java.io.FileOutputStream;

import java.io.IOException;

import javax.swing.JFrame;

import javax.swing.JMenu;

import javax.swing.JMenuBar;

import javax.swing.JMenuItem;

/\*

菜单组件

菜单条(MenuBar) 、 菜单（Menu） 、 菜单项(MenuItem)

菜单条可以添加菜单

菜单可以添加菜单项

复选菜单：

首先菜单添加菜单 ， 菜单添加菜单项。

\*/

import javax.swing.JTextArea;

import cn.itcast.util.FrameUtil;

public class Notepad {

JFrame frame = new JFrame("记事本");

//菜单条

JMenuBar bar = new JMenuBar();

//文件菜单

JMenu fileMenu = new JMenu("文件");

JMenu editMenu = new JMenu("编辑");

JMenu switchMenu = new JMenu("切换工作目录");

//菜单项

JMenuItem openMenu = new JMenuItem("打开");

JMenuItem saveMenu = new JMenuItem("保存");

JMenuItem aboutMenu = new JMenuItem("关于");

JMenuItem closeMenu = new JMenuItem("关闭");

JMenuItem workMenu1 = new JMenuItem("0910project");

JMenuItem workMenu2 = new JMenuItem("1208project");

JMenuItem workMenu3 = new JMenuItem("1110project");

TextArea area = new TextArea(20,30);

public void initNotepad(){

//菜单添加菜单项目

fileMenu.add(openMenu);

fileMenu.add(saveMenu);

//给保存添加事件

saveMenu.addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

try {

FileDialog fileDialog = new FileDialog(frame, "请选择保存的路径",FileDialog.SAVE);

fileDialog.setVisible(true);

//获取用户选择的路径与文件名

String path = fileDialog.getDirectory();

String fileName = fileDialog.getFile();

//创建一个输入对象

FileOutputStream fileOutputStream = new FileOutputStream(new File(path,fileName));

//获取文本域的内容，把内容写出

String content = area.getText();

// content = content.replaceAll("\n", "\r\n");//取代法解决JTextArea的换行问题。或者在前面使用TextArea

fileOutputStream.write(content.getBytes());

//关闭资源

fileOutputStream.close();

} catch (IOException e1) {

// TODO Auto-generated catch block

e1.printStackTrace();

}

}

});

editMenu.add(aboutMenu);

editMenu.add(closeMenu);

//复选菜单

switchMenu.add(workMenu1);

switchMenu.add(workMenu2);

switchMenu.add(workMenu3);

//菜单添加菜单就是复选菜单

fileMenu.add(switchMenu);

//菜单条添加菜单

bar.add(fileMenu);

bar.add(editMenu);

//添加菜单条

frame.add(bar,BorderLayout.NORTH);

frame.add(area);

FrameUtil.initFrame(frame, 500, 600);

}

public static void main(String[] args) {

new Notepad().initNotepad();

}

}

Day26//bat处理文件//对象的克隆//单例设计模式//观察者设计模式//反射//工厂模式//

Demo01(bat处理文件)

package cn.itcast.bat;

/\*

bat处理文件: bat处理文件就是可以一次性执行多个命令的文件。

为什么要学bat处理文件， 快速运行一个软件我一般都会把软件打包一个jar包。

jar双击可以运行仅对于图形化界面的软件起作用，对于控制台的程序是不起作用的。

对于控制台的程序我们可以使用bat处理文件快速启动一个项目。

如何编写bat处理文件呢？

步骤： 编写一个自定义的文本文件，然后把后缀名改成bat即可，然后把你所要执行的命令写在bat处理文件中即可。

bat处理文件常用的命令：

echo 向控制台输出指定的内容。

echo off 隐藏echo off后面执行过的命令。

@ 隐藏当前行执行的命令。

title 改变当前控制台窗口的标题

color 指定控制台的背景颜色与前景颜色

%注释的内容%

pause: 让当前控制台停留。

%1~%9: 给bat处理文件传入参数。

\*/

public class Demo1 {

public static void main(String[] args) {

System.out.println("哈哈...");

}

}

Demo02(copy克隆)

package cn.itcast.copy;

/\*

对象的克隆

对象的浅克隆:

对象浅克隆要注意的细节：

1. 如果一个对象需要调用clone的方法克隆，那么该对象所属的类必须要实现Cloneable接口。

2. Cloneable接口只不过是一个标识接口而已，没有任何方法。

3. 对象的浅克隆就是克隆一个对象的时候，如果被克隆的对象中维护了另外一个类的对象，这时候只是克隆另外一个对象的地址，而没有把

另外一个对象也克隆一份。

4. 对象的浅克隆也不会调用到构造方法的。

对象的深克隆：

\*/

public class Demo1 {

//浅克隆

public static void main(String[] args) throws Exception {

Address address = new Address("广州");

Person p1 = new Person(110,"狗娃",address);

Person p2 = (Person) p1.clone(); //clone() 克隆了一个对象。

p2.name = "狗剩";

p2.address.city ="长沙";

System.out.println("p1:"+p1);//显示 长沙

System.out.println("p2:"+ p2);//显示 长沙

}

}

package cn.itcast.copy;

import java.io.FileInputStream;

import java.io.FileOutputStream;

import java.io.IOException;

import java.io.ObjectInput;

import java.io.ObjectInputStream;

import java.io.ObjectOutputStream;

/\*

对象的深克隆： 对象的深克隆就是利用对象的输入输出流把对象先写到文件上，然后再读取对象的信息这个过程就称作为对象的深克隆。

ObjectInputStream

ObjectOutputStream

\*/

public class Demo2 {

//深克隆

public static void main(String[] args) throws IOException, ClassNotFoundException {

Address address = new Address("广州");

Person p1 = new Person(110,"狗娃",address);

writeObj(p1);

Person p2 =readObj();

p2.address.city = "长沙";

System.out.println("p1:"+ p1);//p1显示 广州

System.out.println("p2:"+ p2);//p2显示长沙

}

//2.再从文件中读取对象的信息

public static Person readObj() throws ClassNotFoundException, IOException{

FileInputStream fileInputStream = new FileInputStream("F:\\obj.txt");

//创建对象的输入流对象

ObjectInputStream objectInputStream = new ObjectInputStream(fileInputStream);

return (Person) objectInputStream.readObject();

}

//1.先要把对象写到文件上

public static void writeObj(Person p) throws IOException{

//建立一个文件 的输出流对象

FileOutputStream fileOutputStream = new FileOutputStream("F:\\obj.txt");

//建立对象的输出流

ObjectOutputStream objectOutputStream = new ObjectOutputStream(fileOutputStream);

//把对象写出

objectOutputStream.writeObject(p);

//关闭资源

objectOutputStream.close();

}

}

package cn.itcast.copy;

import java.io.Serializable;

class Address implements Serializable{ //Serializable文件读写的接口

String city;

public Address(String city){

this.city = city;

}

}

public class Person implements Cloneable,Serializable {//必须实现Cloneable接口

int id;

String name;

Address address;

public Person(int id, String name) {

this.id = id;

this.name = name;

}

public Person(int id, String name, Address address) {

this.id = id;

this.name = name;

this.address = address;

System.out.println("=======构造方法调用了===");

}

@Override

public String toString() {

return "编号："+ this.id+" 姓名："+ this.name+" 地址："+ address.city;

}

@Override

public Object clone() throws CloneNotSupportedException {

return super.clone();

}

}

Demo03(内存泄漏)

package cn.itcast.list;

import java.util.Arrays;

/\*

内存泄露

需求：编写一个类使用数组模拟堆栈的存储方式。

堆栈存储特点： 先进后出，后进先出。

注意： 不再使用的对象，应该不要让变量指向该对象，要让该对象尽快的被垃圾回收期回收。

\*/

class StackList{

Object[] elements;

int index = 0 ; //当前的索引值

public StackList(){

this.elements = new Object[3];

}

//添加内容

public void add(Object o){

//添加元素之前应该要先检查是否容量够用。

ensureCapcity();

elements[index++] = o;

}

//出栈: 删除集合的元素，并且返回。

public Object pop(){

int tempIndex = --index;

Object o = elements[tempIndex];//取出里面的元素

elements[tempIndex] = null; //让该位置不再引用着指定的对象,让垃圾回收期赶快回收该垃圾。

return o;

}

//检查当前的数组使用够用。

public void ensureCapcity(){

if(index==elements.length){

//计算一个新的长度

int newLength = elements.length\*2;

elements = Arrays.copyOf(elements, newLength);

}

}

//获取当前的元素 个数

public int size(){

return index;

}

}

public class Demo1 {

public static void main(String[] args) {

StackList list = new StackList();

list.add("狗娃");

list.add("狗剩");

list.add("铁蛋");

list.add("美美");

int size = list.size();//固定size

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

System.out.println(list.pop());

}

}

}

Demo04(单例设计模式的效率问题)

package cn.itcast.single;

/\*

单例设计模式：

懒汉单例设计模式(线程安全问题 的解决方案):

步骤：

1. 私有化构造函数。

2. 声明本类引用类型变量，但是不要创建对象。

3. 声明一个公共静态的方法获取本类的对象，获取之前先判断是否已经创建了本类的对象了，如果已经创建了

就不要在创建直接返回即可，如果还没有创建，那么就先创建本类的引用类型变量，然后再返回。

\*/

//懒汉单例设计模式

class Single{

// 声明本类引用类型变量，但是不要创建对象。

private static Single s;

//私有化构造函数

private Single(){}

// 声明一个公共静态的方法获取本类的对象

public static Single getInstance(){

if(s==null){ //提高以后的执行效率

synchronized ("锁") {

if(s==null){

s = new Single();

}

}

}

return s;

}

}

public class Demo1 {

}

Demo01(观察者设计模式)

package cn.itcast.observer;

import java.util.ArrayList;

import java.util.Random;

/\*

观察者设计模式：观察者设计模式解决的问题时当一个对象发生指定的动作时，要通过另外一个对象做出相应的处理。

需求： 编写一个气象站、一个工人两个类，当气象站更新天气 的时候，要通知人做出相应的处理。

问题1： 气象站更新了多次天气，然后人才做一次的处理。

问题2： 目前气象站只能通知一个人而已。

问题3： 在现实生活中出了工人群体要关注天气，其他 的群体也需要关注天气

观察者设计模式的步骤：

1. 当前目前对象发生指定的动作是，要通知另外一个对象做出相应的处理，这时候应该把对方的相应处理方法定义在接口上。

2. 在当前对象维护接口的引用，当当前对象发生指定的动作这时候即可调用接口中的方法了。

\*/

//气象站

public class WeatherStation {

String[] weathers = {"晴天","雾霾","刮风","冰雹","下雪"};

//当前天气

String weather ;

//该集合中存储的都是需要收听天气预报的人

ArrayList<Weather> list = new ArrayList<Weather>(); //程序设计讲究低耦合---->尽量不要让一个类过分依赖于另外一个类。

public void addListener(Weather e){

list.add(e);

}

//开始工作

public void startWork() {

final Random random = new Random();

new Thread(){

@Override

public void run() {

while(true){

updateWeather(); // 每1~1.5秒更新一次天气 1000~1500

for(Weather e : list){

e.notifyWeather(weather);

}

int s = random.nextInt(501)+1000; // 500

try {

Thread.sleep(s);

} catch (InterruptedException e) {

e.printStackTrace();

}

}

}

}.start();

}

//更新天气的 方法

public void updateWeather(){

Random random = new Random();

int index = random.nextInt(weathers.length);

weather = weathers[index];

System.out.println("当前的天气是： " + weather);

}

}

//工人类

package cn.itcast.observer;

public class Emp implements Weather{

String name;

public Emp(String name) {

this.name = name;

}

//人是要根据天气做出相应的处理的。 "晴天","雾霾","刮风","冰雹","下雪"

public void notifyWeather(String weather){

if("晴天".equals(weather)){

System.out.println(name+"高高兴兴的去上班!!");

}else if("雾霾".equals(weather)){

System.out.println(name+"戴着消毒面具去上班!");

}else if("刮风".equals(weather)){

System.out.println(name+"拖着大石头过来上班!");

}else if("冰雹".equals(weather)){

System.out.println(name+"戴着头盔过来上班!");

}else if("下雪".equals(weather)){

System.out.println(name+"戴着被子过来上班!");

}

}

}

package cn.itcast.observer;

//学生类

public class Student implements Weather{

String name;

public Student(String name) {

super();

this.name = name;

}

public void notifyWeather(String weather){

if("晴天".equals(weather)){

System.out.println(name+"高高兴兴的去开学!!");

}else if("雾霾".equals(weather)){

System.out.println(name+"吸多两口去上学!");

}else if("刮风".equals(weather)){

System.out.println(name+"在家睡觉!");

}else if("冰雹".equals(weather)){

System.out.println(name+"在家睡觉!");

}else if("下雪".equals(weather)){

System.out.println(name+"等下完再去上学!");

}

}

}

//主函数代码

package cn.itcast.observer;

import java.util.Random;

public class WeatherMain {

public static void main(String[] args) throws Exception {

//工人

Emp e = new Emp("小明");

Emp e2 = new Emp("如花");

//学生

Student s1 = new Student("狗娃");

Student s2 = new Student("狗剩");

WeatherStation station = new WeatherStation();

station.addListener(e);

station.addListener(e2);

station.addListener(s1);

station.addListener(s2);

station.startWork();

}

}

//接口类代码

package cn.itcast.observer;

//订阅天气预报的接口

public interface Weather {

public void notifyWeather(String weather);

}

//反射技术

Demo01(reflect技术获取Method类)

package cn.itcast.reflect;

/\*

反射： 当一个字节码文件加载到内存的时候，jvm会对该字节码进行解剖，然后会创建一个对象的Class对象，把字节码文件的信息全部都存储到该Class对象中，我们只要获取到Class对象，我们就可以使用字节码对象设置对象的属性或者调用对象的方法等操作....

注意： 在反射技术中一个类的任何成员都有对应 的类进行描述。 比如： 成员变量（Field） 方法----> Method类

\*/

public class Demo1 {

Person p;

public static void main(String[] args) throws ClassNotFoundException {

// Person p = new Person(110,"狗娃");

//推荐使用： 使用完整类名，获取Class对象的方式一

Class clazz1 = Class.forName("cn.itcast.reflect.Person");

System.out.println("clazz1:"+ clazz1);

//获取Class对象的方式二： 通过类名获取

Class clazz2 = Person.class;

System.out.println("clazz1==clazz2?"+ (clazz1==clazz2));

//获取Class对象的方式三 ：通过对象获取

Class clazz3 = new Person(110,"狗娃").getClass();

System.out.println("clazz2==clazz3?"+ (clazz2==clazz3));

}

}

Demo02(reflect技术获取Method类)

package cn.itcast.reflect;

import java.lang.reflect.Constructor;

/\*

如何通过Class对象获取构造方法。

\*/

public class Demo2 {

public static void main(String[] args) throws Exception {

//获取到对应的Class对象

Class clazz = Class.forName("cn.itcast.reflect.Person");

//通过Class对象获取对应的构造方法

/\*Constructor[] constructors = clazz.getConstructors(); // getConstructors()获取一个类的所有公共的构造方法

for(Constructor constructor : constructors){

System.out.println(constructor);

}

Constructor[] constructors = clazz.getDeclaredConstructors(); //获取到一个类的所有构造方法，包括私有的在内 。

for(Constructor constructor : constructors){

System.out.println(constructor);

}

\*/

/\*Constructor constructor = clazz.getConstructor(int.class,String.class); // getConstructor 获取单个指定的构造方法。

Person p = (Person) constructor.newInstance(999,"小城"); // newInstance()创建一个对象

System.out.println(p);\*/

//获取私有的构造函数

Constructor constructor = clazz.getDeclaredConstructor(null);

//暴力反射

constructor.setAccessible(true);//设置权限暴力创建对象

Person p =(Person) constructor.newInstance(null);

System.out.println(p);

}

}

Demo03(reflect技术获取Method类)

package cn.itcast.reflect;

import java.lang.reflect.Method;

/\*

通过Class对象获取到对应的方法。

在反射技术中使用了Method类描述了方法的。

\*/

public class Demo3 {

public static void main(String[] args) throws Exception {

//获取到对应的Class对象

Class clazz = Class.forName("cn.itcast.reflect.Person");

//获取到所有公共的方法

/\*Method[] methods = clazz.getMethods(); // getMethods() 获取所有的公共方法而已。

Method[] methods = clazz.getDeclaredMethods(); //获取到所有的方法，但是不包含父类的方法

//遍历

for(Method method : methods){

System.out.println(method);

}\*/

Person p = new Person(110,"狗娃");

Method m = clazz.getMethod("eat", int.class);

m.invoke(p, 3); //invoke 执行一个方法。 第一个参数：方法的调用对象。 第二参数： 方法所需要的参数。

//执行私有的方法

Method m =clazz.getDeclaredMethod("sleep",int.class);

//设置访问权限允许访问

m.setAccessible(true);

m.invoke(null, 6);//静态方法不需对象调用所以也可以是null

//获取数组类型

Method m = clazz.getMethod("sum", int[].class);

m.invoke(p,new int[]{12,5,9});

}

}

Demo04(reflect技术获取Filed类)

package cn.itcast.reflect;

import java.lang.reflect.Field;

/\*

通过反射获取对应的成员变量

在反射技术中使用了Field类描述了成员变量的。

\*/

public class Demo4 {

public static void main(String[] args) throws Exception {

//获取到对应的Class对象

Class clazz = Class.forName("cn.itcast.reflect.Person");

//获取到所有的成员变量

/\*Field[] fields = clazz.getDeclaredFields();

for(Field field : fields){

System.out.println(field);

}\*/

Person p = new Person();

Field field = clazz.getDeclaredField("id");//获取单个成员变量

//设置访问权限可以访问

field.setAccessible(true);

field.set(p, 110); //第一个参数： 设置该数据的成员变量， 第二个参数：属性值。

System.out.println(p);

}

}

//"cn.itcast.reflect.Person"(reflect技术person类)

package cn.itcast.reflect;

public class Person {

private int id;

String name;

public Person(int id,String name){

this.id = id;

this.name = name;

}

public Person(){

}

public void eat(int num){

System.out.println(name+"吃很"+num+"斤饭");

}

private static void sleep(int num){

System.out.println("明天睡上"+num+"小时");

}

public static void sum(int[] arr){

System.out.println("长度是："+ arr.length);

}

@Override

public String toString() {

return " 编号："+ this.id +" 姓名："+ this.name;

}

}

Demo01(工厂设计模式:反射的应用)

package cn.itcast.factory;

import java.io.BufferedReader;

import java.io.FileNotFoundException;

import java.io.FileReader;

import java.lang.reflect.Constructor;

import java.lang.reflect.Field;

import cn.itcast.reflect.Person;

/\*

工厂设计模式：就是用于产生对象的。

\*/

class Car{}

class BMW extends Car{}

class BSJ extends Car{}

public class Demo1 {

public static void main(String[] args) throws Exception {

Person p = (Person) getInstance();

System.out.println(p);

}

//需求： 编写一个工厂方法根据配置文件返回对应的对象

public static Object getInstance() throws Exception{

//读取配置文件

BufferedReader bufferedReader = new BufferedReader(new FileReader("info.txt"));

//读取第一行 : 读取类文件的信息

String className = bufferedReader.readLine();

//通过完整类名获取对应 的Class对象

Class clazz = Class.forName(className);

//获取到对应的构造方法

Constructor constructor = clazz.getDeclaredConstructor(null);

constructor.setAccessible(true);

Object o = constructor.newInstance(null);

//给对象设置对应的属性值

String line = null;

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

String[] datas = line.split("=");//按照=切割

Field field =clazz.getDeclaredField(datas[0]);

//设置可以访问

field.setAccessible(true);

if(field.getType()==int.class){//判断获取的字段的类型

field.set(o, Integer.parseInt(datas[1]));//如果是int类型

}else{

field.set(o, datas[1]);

}

}

return o;

}

}

>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

info.txt

cn.itcast.reflect.Person

id=110

name=旺财

<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*基础增强\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# 基础回顾

## 集合

### 集合的类型与各自的特性

|  |
| --- |
| ---|**Collection**: 单列集合  ---|**List**: 有存储顺序, 可重复  ---|**ArrayList**: 数组实现, 查找快, 增删慢  由于是数组实现, 在增和删的时候会牵扯到数组  增容, 以及拷贝元素. 所以慢。数组是可以直接按索引查找, 所以查找时较快  ---|**LinkedList**: 链表实现, 增删快, 查找慢由于链表实现, 增加时只要让前一个元素记住自己就可以, 删除时让前一个元素记住后一个元素, 后一个元素记住前一个元素. 这样的增删效率较高但查询时需要一个一个的遍历, 所以效率较低  ---|**Vector**: 和ArrayList原理相同, 但线程安全, 效率略低  和ArrayList实现方式相同, 但考虑了线程安全问题, 所以效率略低  ---|**Set**: 无存储顺序, 不可重复  ---|**HashSet** 线程不安全，存取速度快。底层是以哈希表实现的。  ---|**TreeSet** 红-黑树的数据结构，默认对元素进行自然排  序（String）。如果在比较的时候两个对象  返回值为0，那么元素重复。  ---| Map: 键值对 键不可重复，键可以重复  ---|**HashMap** 线程不安全，存取速度快。底层是以哈希表实现的.  ---|**TreeMap** 红-黑树的数据结构，默认对元素进行自然排  序（String）。如果在比较的时候两个对象  返回值为0，那么元素重复  ---|**HashTable** 底层也是使用了哈希表 维护的，存取的读取快，存储元素是  无序的。 |

### 遍历集合

#### 遍历集合的几种方式

1. 使用迭代器Iterator的方式。
2. **使用增强for循环的方式。**
3. **如果有下标，则可以使用下标的方式。**

#### 遍历数组

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

// 遍历数组：

String[] arr = **new** String[] { "xx", "yy", "zz" };

// 1，增强的for循环

**for** (String elt : arr) {

System.*out*.println(elt);

}

// 2，下标的方式

**for** (**int** i = 0; i < arr.length; i++) {

System.*out*.println(arr[i]);

}

}

#### 遍历List

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

// 遍历List：

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

list.add("aa");

list.add("bb");

list.add("cc");

// 1，增强的for循环

**for** (String elt : list) {

System.*out*.println(elt);

}

// 2，下标

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

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

}

// 3，迭代器

**for** (Iterator<String> iter = list.iterator(); iter.hasNext();) {

String elt = iter.next();

System.*out*.println(elt);

}

}

#### 遍历Set

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

// 遍历Set：

Set<String> set = **new** HashSet<String>();

set.add("dd");

set.add("ee");

set.add("ff");

// 1，增强的for循环

**for** (String elt : set) {

System.*out*.println(elt);

}

// 2，迭代器

**for**(Iterator<String> iter = set.iterator(); iter.hasNext() ; ){

String elt = iter.next();

System.*out*.println(elt);

}

}

#### 遍历Map

## 泛型(Generic)

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

// 遍历Map：

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

map.put("aa", "xx");

map.put("bb", "yy");

map.put("cc", "zz");

// 1，增强的for循环（Entry集合）

**for** (Entry<String, String> entry : map.entrySet()) {

System.*out*.println(entry);

}

// 2，增强的for循环（Key集合）

**for**(String key : map.keySet()){

System.*out*.println(key + " = " + map.get(key));

}

// 3，遍历值的集合

**for**(String value : map.values()){

System.*out*.println(value);

}

}

当集合中存储的对象类型不同时，那么会导致程序在运行的时候的转型异常

|  |
| --- |
| **import** java.util.ArrayList;  **import** java.util.Iterator;  **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  ArrayList arr = **new** ArrayList();  arr.add(**new** Tiger("华南虎"));  arr.add(**new** Tiger("东北虎"));  arr.add(**new** Sheep("喜羊羊"));  System.*out*.println(arr);  Iterator it = arr.iterator();  **while** (it.hasNext()) {  Object next = it.next();  Tiger t = (Tiger) next;  t.eat();  }  }  }  **class** Tiger {  String name;  **public** Tiger() {  }  **public** Tiger(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Tiger@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃羊");  }  }  **class** Sheep {  String name;  **public** Sheep() {  }  **public** Sheep(String name) {  **this**.name = name;  }  @Override  **public** String toString() {  **return** "Sheep@name:" + **this**.name;  }  **public** **void** eat() {  System.*out*.println(**this**.name + "吃青草");  }  } |

原因 :发现虽然集合可以存储任意对象,但是如果需要使用对象的特有方法,那么就需要类型转换,如果集合中存入的对象不同,可能引发类型转换异常.

|  |
| --- |
| [Tiger@name:华南虎, Tiger@name:东北虎, Sheep@name:喜羊羊]  华南虎吃羊  东北虎吃羊  Exception in thread "main" java.lang.ClassCastException: cn.itcast.gz.map.Sheep cannot be cast to cn.itcast.gz.map.Tiger  at cn.itcast.gz.map.Demo5.main(Demo5.java:17) |

出现问题:

存入的是特定的对象,取出的时候是Object对象,需要强制类型转换,可能诱发类型转换异常.

无法控制存入的是什么类型的对象,取出对象的时候进行强转时可能诱发异常.而且在编译时期无法发现问题.

虽然可以再类型转换的时候通过if语句进行类型检查(instanceof),但是效率较低.(例如吃饭的时候,还需要判断米饭里有没有沙子,吃饭效率低).可以通过给容器加限定的形式规定容器只能存储一种类型的对象.

就像给容器贴标签说明该容器中只能存储什么样类型的对象。

所以在jdk5.0后出现了泛型

泛型应用:

格式

1. 集合类<类类型> 变量名 = new 集合类<类类型>();

|  |
| --- |
| **public** **class** Demo5 {  **public** **static** **void** main(String[] args) {  // 使用泛型后,规定该集合只能放羊,老虎就进不来了.  ArrayList<Sheep> arr = **new** ArrayList<Sheep>();  arr.add(**new** Sheep("美羊羊"));  arr.add(**new** Sheep("懒洋洋"));  arr.add(**new** Sheep("喜羊羊"));  // 编译失败  // arr.add(new Tiger("东北虎"));  System.*out*.println(arr);  Iterator<Sheep> it = arr.iterator();  **while** (it.hasNext()) {  // 使用泛型后,不需要强制类型转换了  Sheep next = it.next();  next.eat();  }  }  } |

1. 将运行时的异常提前至编译时发生。

2. 获取元素的时候无需强转类型，就避免了类型转换的异常问题

格式 通过<> 来指定容器中元素的类型.

什么时候使用泛型:当类中操作的引用数据类型不确定的时候,就可以使用泛型类.

JDK5.0之前的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable {  **public** **int** compareTo(Object o);  } |

JDK5.0之后的Comparable

|  |
| --- |
| **package java.lang;**  **public** **interface** Comparable<T> {  **public** **int** compareTo(T o);  } |

这里的<T>表示泛型类型,随后可以传入具体的类型来替换它.

细节一

声明好泛型类型之后,集合中只能存放特定类型元素

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  //创建一个存储字符串的list  ArrayList<String> arr=**new** ArrayList<String>();  arr.add("gz");  arr.add("itcast");  //存储非字符串编译报错.  arr.add(1);  }  } |

细节二:

泛型类型必须是引用类型

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  // 泛型类型必须是引用类型,也就是说集合不能存储基本数据类型  // ArrayList<int> arr2=new ArrayList<int>();  // 使用基本数据类型的包装类  ArrayList<Integer> arr2 = **new** ArrayList<Integer>();      }  } |

细节三: 使用泛型后取出元素不需要类型转换.

|  |
| --- |
| **public** **class** Demo6 {  **public** **static** **void** main(String[] args) {  ArrayList<String> arr = **new** ArrayList<String>();  arr.add("gzitcast");  arr.add("cditcast");  arr.add("bjitcast");  //使用泛型后取出元素不需要类型转换.  String str=arr.get(0);  System.*out*.println();  }  } |

## 泛型方法

需求：写一个函数，调用者传递什么类型的变量，该函数就返回什么类型的变量？

实现一:

由于无法确定具体传递什么类型的数据.那么方法的形参就定义为Object类型.返回值也就是Object类型.但是使用该函数时需要强制类型转换.

|  |
| --- |
| **private** Object getDate(Object obj) {  **return** obj;  } |

当不进行强制类型转换能否写出该功能.?

目前所学的知识无法解决该问题

就需要使用泛型类解决

使用的泛型的自定义来解决以上问题。

泛型： 就是将类型当作变量处理。规范泛型的定义一般是一个大写的任意字母。

|  |
| --- |
| 1. 函数上的泛型定义  当函数中使用了一个不明确的数据类型，那么在函数上就可以进行泛型的定义。  public <泛型的声明> 返回值类型 函数名( 泛型 变量名 ){      } |

|  |
| --- |
| **public** **static** **void** main(String[] args) {  **int**[] arr = { 1, 2, 3, 4, 5 };    new Demo6().*getData*(5);  }  **public** <T> T getData(T data) {  **return** data;  } |

细节：

使用泛型方法前需要进行泛型声明，使用一对尖括号 <泛型>，声明的位置在static后返回值类型前。

当一个类中有多个函数声明了泛型，那么该泛型的声明可以声明在类上。

## 泛型类

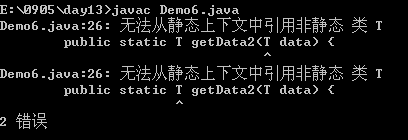
格式

|  |
| --- |
| 2. 类上的泛型声明  修饰符 class 类名<泛型>{    } |

|  |
| --- |
| **import** java.util.Arrays;  **public** **class** Demo6<T> {  **public** **static** **void** main(String[] args) {  // 使用泛型类，创建对象的时候需要指定具体的类型  **new** Demo6<Integer>().getData(5);  }  **public** T getData(T data) {  **return** data;  }  // 反序任意类型数组  **public** **void** reverse(T[] arr) {  **int** start = 0;  **int** end = arr.length - 1;  **for** (**int** i = 0; i < arr.length; i++) {  **if** (start < end) {  T temp = arr[start];  arr[start] = arr[end];  arr[end] = temp;  }  }    } |

在泛型类中定义一个静态方法

|  |
| --- |
| public class Demo6<T> {  public static void main(String[] args) {  System.out.println(getData2(100));  }  public T getData(T data) {  return data;  }  //静态方法  public static T getData2(T data) {  return data;  }  } |



注意：静态方法不可以使用类中定义的泛型

因为类中的泛型需要在对象初始化时指定具体的类型，而静态优先于对象存在。那么类中的静态方法就需要单独进行泛型声明，声明泛型一定要写在static后，返回值类型之前

泛型类细节：

|  |
| --- |
| 1、创建对象的时候要指定泛型的具体类型  2、创建对象时可以不指定泛型的具体类型(和创建集合对象一眼)。默认是Object，例如我们使用集合存储元素的时候没有使用泛型就是那么参数的类型就是Object  3、类上面声明的泛型只能应用于非静态成员函数，如果静态函数需要使用泛型，那么  需要在函数上独立声明。  4、如果建立对象后指定了泛型的具体类型，那么该对象操作方法时，这些方法只能操作一种数据类型。  5、所以既可以在类上的泛型声明，也可以在同时在该类的方法中声明泛型。 |

泛型练习：

定义泛型成员

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  } |

如果Father类有子类，子类该如何实现

|  |
| --- |
| **public** **class** Demo7 {  **public** **static** **void** main(String[] args) {  Father<String> f = **new** Father<String>("jack");  System.*out*.println(f.getT());  Father<Integer> f2 = **new** Father<Integer>(20);  System.*out*.println(f2.getT());  }  }  **class** Father<T> {  **private** T t;  **public** Father() {  }  **public** Father(T t) {  **super**();  **this**.t = t;  }  **public** T getT() {  **return** t;  }  **public** **void** setT(T t) {  **this**.t = t;  }  }  //子类指定了具体的类型  **class** Son **extends** Father<String>{    }  //子类也需要使用泛型  **class** Son3<T> **extends** Father<T>{    }  //错误写法，父类上定义有泛型需要进行处理  **class** Son2 **extends** Father<T>{    } |

## 泛型接口

|  |
| --- |
| **public** **class** Demo8 {  **public** **static** **void** main(String[] args) {  MyInter<String> my = **new** MyInter<String>();  my.print("泛型");  MyInter2 my2 = **new** MyInter2();  my.print("只能传字符串");  }  }  **interface** Inter<T> {  **void** print(T t);  }  // 实现不知为何类型时可以这样定义  **class** MyInter<T> **implements** Inter<T> {  **public** **void** print(T t) {  System.*out*.println("myprint:" + t);  }  }  //使用接口时明确具体类型。  **class** MyInter2 **implements** Inter<String> {  @Override  **public** **void** print(String t) {  System.*out*.println("myprint:" + t);  }  } |

## IO流

### IO流的分类

|  |  |  |  |
| --- | --- | --- | --- |
|  | **输入流** | **输出流** | **说明** |
| **字节流** | InputStream | OutputStream | 字节流是处理字节的（二进制） |
| **字符流** | Reader | Writer | 字符流是处理字符的 |

注：这几个类都是抽象类。

### 读文件的代码

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

String path = "c:/a.txt";

FileInputStream in = **null**;

**try** {

// 打开流

in = **new** FileInputStream(path);

// 使用流读文件内容

**int** b = in.read();

**while** (b != -1) {

System.*out*.print((**char**) b);

b = in.read();

}

} **catch** (Exception e) {

**throw** **new** RuntimeException(e);

} **finally** {

// 释放资源

**if** (in != **null**) {

**try** {

in.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

}

}

### 拷贝文件的代码

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

String srcPath = "c:/a.txt";

String destPath = "c:/b.txt";

// 一定要使用字节流

InputStream in = **null**;

OutputStream out = **null**;

**try** {

// 打开流

in = **new** FileInputStream(srcPath);

out = **new** FileOutputStream(destPath);

// 使用流

**byte**[] buf = **new** **byte**[1024 \* 8];

**for** (**int** len = -1; (len = in.read(buf)) != -1;) {

out.write(buf, 0, len);

}

} **catch** (Exception e) {

e.printStackTrace();

} **finally** {

// 释放资源

**try** {

**if** (in != **null**) {

in.close();

}

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

} **finally** {

**if** (out != **null**) {

**try** {

out.close();

} **catch** (IOException e) {

**throw** **new** RuntimeException(e);

}

}

}

}

}

## 多线程

### 启动线程方式

1. 自定义的类继承Thread类。

使用代码为new MyThread().start()

2，自定义的类实现Runnable接口。

使用代码为new Thread(new MyRunnable()).start

### 代码

**以下代码是分别用两种方式启动线程（还是用到了匿名内部类）**

**private** **static** **int** *count* = 100;

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

// 用继承Thread类的方式启动一个线程

**new** Thread() {

**public** **void** run() {

**synchronized** (StartThreadTest.**class**) {

**while** (*count* > 0) {

*count*--;

System.*out*.println(Thread.*currentThread*() + "卖了一张票，还剩" + *count*);

}

}

}

}.start();

// 用实现Runnable接口的方式启动一个线程

**new** Thread(**new** Runnable() {

**public** **void** run() {

**synchronized** (StartThreadTest.**class**) {

**while** (*count* > 0) {

*count*--;

System.*out*.println(Thread.*currentThread*() + "卖了一张票，还剩" + *count*);

}

}

}

}).start();

}

# Junit单元测试

### Junit单元测试框架的基本使用

一、搭建环境：

导入junit.jar包（junit4）

二、写测试类：

0，一般一个类对应一个测试类。

1，测试类与被测试类最好是放到同一个包中（可以是不同的源文件夹）

2，测试类的名字为被测试类的名字加Test后缀。

三：写测试方法：

0，一般一个方法对应一个单元测试方法。

1，测试方法的名字为test前缀加被测试方法的名字，如testAddPerson()。

2，单元测试方法上面要加上@Test注解（org.junit.Test）！

3，单元测试方法不能有参数，也不能有返回值（返回void）！测试的方法不能是静态的方法。

四、测试方法的基本使用：

1，可以单独执行一个测试方法，也可以一次执行所有的、一个包的、一个类中所有的测试方法。

2，执行完后，显示绿色表示测试成功；显示红色表示测试失败（抛异常后会测试失败）。

### Assert断言工具类

其中有一些静态的工具方法（不符合期望就抛异常）：

assertTrue(...) 参数的值应是true

assertFalse(...) 参数的值应是false

assertNull(...) 应是null值

assertNotNull(...) 应是非null的值

assertSame(...) 使用==比较的结果为true（表示同一个对象）

AssertNotSame(...) 使用==比较的结果为false

assertEquals(...) 两个对象equals()方法比较结果为true

### 用于准备环境、清理环境的方法

@Test

表示单元测试方法。

@Before

所修饰的方法应是非static的（且没有参数，返回值为void）。

表示这个方法会在本类中的每个单元测试方法之前都执行一次。

@After

所修饰的方法应是非static的（且没有参数，返回值为void）。

表示这个方法会在本类中的每个单元测试方法之后都执行一次。

@BeforeClass

所修饰的方法应是static的（且没有参数，返回值为void）。

表示这个方法会在本类中的所有单元测试方法之前执行，只执行一次。

@AfterClass

所修饰的方法应是static的（且没有参数，返回值为void）。

表示这个方法会在本类中的所有单元测试方法之后执行，只执行一次。

# 内省(Introspector)

# 为什么要学内省？

开发框架时，经常需要使用java对象的属性来封装程序的数据，每次都使用反射技术完成此类操作过于麻烦，所以sun公司开发了一套API，专门用于操作java对象的属性。

内省是用于操作java对象的属性的，那么以下问题我们必须要清楚。

**问题一：** 什么是Java对象的属性和属性的读写方法？

问题二: 如何通过内省访问到javaBean的属性 ?

1. 通过PropertyDescriptor类操作Bean的属性.

|  |
| --- |
| **public** **static** **void** testPropertyDescriptor() **throws** Exception{  Person p = **new** Person();  PropertyDescriptor propertyDescriptor = **new** PropertyDescriptor("id",Person.**class**);  //获取属性的写的方法。  Method writeMethod = propertyDescriptor.getWriteMethod();  Method readMethod = propertyDescriptor.getReadMethod();  propertyDescriptor.getReadMethod();  writeMethod.invoke(p, 12);  System.*out*.println(readMethod.invoke(p, **null**));  } |

2. 通过Introspector类获得Bean对象的 BeanInfo，然后通过 BeanInfo 来获取属性的描述器（ PropertyDescriptor ），通过这个属性描述器就可以获取某个属性对应的 getter/setter 方法，然后通过反射机制来调用这些方法。

|  |
| --- |
| **public** **static** **void** testIntrospector() **throws** Exception{  BeanInfo beanInfo = Introspector.*getBeanInfo*(Person.**class**);  PropertyDescriptor[] descriptor = beanInfo.getPropertyDescriptors();  **for**(PropertyDescriptor itemProperty : descriptor){  System.*out*.println(itemProperty.getReadMethod().getName());  }  } |

存在的问题： sun公司的内省API过于繁琐，所以Apache组织结合很多实际开发中的应用场景开发了一套简单、易用的API操作Bean的属性——BeanUtils。

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** Exception {  Person p = **new** Person();  ConvertUtils.*register*(**new** Converter() {    @Override  **public** Object convert(Class type, Object value) {  **try** {  **if**(value!=**null**){    SimpleDateFormat dateFormat = **new** SimpleDateFormat("yyyy MM dd");  Date d = dateFormat.parse((String) value);  **return** d;  }  } **catch** (ParseException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }    **return** **null**;  }  }, Date.**class**);    BeanUtils.*setProperty*(p,"id","110");  BeanUtils.*setProperty*(p,"name","狗娃");  BeanUtils.*setProperty*(p, "birthDay","1992 12 12");  System.*out*.println(p.getId() +"=="+ p.getName()+"======"+p.getBirthDay());  } |

# Properties类与配置文件

### Properties配置文件说明

Properties类对应.properties文件。文件内容是键值对，键值对之间使用"="或空格隔开。开头是"#"的表示注释

Properties类在加载.properties文件时使用的iso8859-1的编码。所以这个文件中的中文要特殊处理：如果这个配置文件中有中文就必须要进行转义，使用native2ascii.exe命令操作:

**native2ascii d:/my.properties d:/my2.properties**

使用Properties类中的load(InputStream) 方法可以加载配置文件，使用其中的store(OutputStream) 方法可以保存配置到指定文件。

更多的信息可以看Properties类的API文档。

### 加载配置文件

**public** **static** **void** testLoadProperties() **throws** Exception {

Properties properties = **new** Properties();

InputStream in = **new** FileInputStream("E:/itcast/config.properties");

properties.load(in); // 加载

in.close();

System.*out*.println(properties);

}

### 写配置文件

**public** **static** **void** testStoreProperties() **throws** Exception {

// 准备配置信息

Properties properties = **new** Properties();

properties.setProperty("name", "李四");

properties.setProperty("age", "20");

// 准备

OutputStream out = **new** FileOutputStream("d:/my.properties");

String comments = "这是我的配置文件";

// 写出去

properties.store(out, comments);

out.close();

}

作业：使用properties读取配置文件，读取数据库的用户名、密码。并且打包成jar包。

### 使用Properties类

|  |
| --- |
| **public** **class** DBUtil {    **static** Properties *properties* = **new** Properties();    **static**{  **try** {  Class clazz = DBUtil.**class**;  InputStreamReader fileReader =  **new** InputStreamReader(clazz.getResourceAsStream("/db.properties"));  *properties*.load(fileReader);  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **static** String getUserName(){  String userName =*properties*.getProperty("userName");  **return** userName;  }    **public** **static** String getPassword(){  **return** *properties*.getProperty("password");  }  **public** **static** **void** main(String[] args) {  System.*out*.println("用户名："+ *getUserName*());  System.*out*.println("密码: "+ *getPassword*());  }  } |

# 文件路径

### 绝对路径

以根目录或某盘符开头的路径（或者说完整的路径）

例如：

* c:/a.txt （Windows操作系统中）
* c:/xxx/a.txt （Windows操作系统中）
* /var/xx/aa.txt （Linux操作系统中）

绝对路径的问题: 比如C:\abc\a.properties文件路径，该路径在windows上执行没有 问题，但是如果把该项目移动到linux上面执行 ，该路径就会出现问题了，因为在linux上面没有c盘的，只有根目录\。

### 相对路径

相对于当前路径的一个路径。例如当前文件夹为c:/abc时：相对路径a.txt表示c:/abc/a.txt，相对路径xx/a.txt = c:/abc/xx/a.txt

* . 表示当前文件夹
* .. 表示上级文件夹

相对路径存在的问题:相对路径是相对于目前执行class文件的时候，控制台所在的路径，这样子也会导致出现问题。

### Java程序中的相对路径

**在Java程序中使用File时写相对路径，是指相对于执行java命令时当前所在的文件夹。**

**测试代码：**

**public** **class** PathTest {

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

System.*out*.println(**new** File("a.txt").getAbsolutePath());

}

}

在命令行中使用cd命令切换到不同的路径下试试，可以看到以上所说的效果。

**在Eclipse中，当前路径是工程的根目录。**

### classpath路径

#### classpath路径说明

在Java程序中，一般情况下使用绝对路径还是相对路径都不太合适，因为Java程序的jar包所放的位置不确定，执行java程序时当前的路径也不确定，所以不合适。一般在Java程序中我们会把资源放到classpath中，然后使用classpath路径查找资源。

Classpath路径：就是使用classpath目前的路径。

#### 获取classpath中的资源（InputStream）

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

Class clazz = **new** ClassPathTest().getClass();

// 开头的'/'表示classpath的根目录，这个是表示从classpath的根目录中开始查找资源

InputStream in = clazz.getResourceAsStream("/cn/itcast/my.properties");

// 如果开头没有'/'，表示从当前这个class所在的包中开始查找

InputStream in2 = clazz.getResourceAsStream("my.properties");

}