# 集合

## Collection集合问答

### List集合特点

LinkedList Vector ArrayList的区别？

**答：**

**ArrayList：内部使用数组实现，动态调整大小。读取快，插入删除慢**

**LinkedList:** **内部使用双向链表实现，插入删除快，读取快**

**Vector:** **线程安全的动态数组实现，操作方法同步，多线程环境安全**

### List、Set集合对⽐

简述list集合和set集合的特点。

**答：**

**List**

List 是一个有序的集合，它可以包含重复的元素。List 接口中定义了一些基本的操作，比如添加元素、删除元素、获取指定位置的元素等。List 的实现类包括但不限于 ArrayList、LinkedList 和 Vector。

**有序**：元素按插入顺序保存，可以通过索引来访问特定的元素。

**可重复元素**：可以向 List 中添加相同的元素多次。

**索引支持**：支持通过索引访问元素，可以使用 get(int index) 方法获取某个位置的元素。

**常见实现**：ArrayList（基于数组）、LinkedList（基于链表）以及 Vector（线程安全的 ArrayList）。

**Set**

Set 是一个不包含重复元素的集合。这意味着每个元素都必须是唯一的。Set 接口没有提供索引支持，所以不能通过索引访问元素。Set 的实现类主要有 HashSet、TreeSet 和 LinkedHashSet。

**无重复元素**：试图添加重复元素到 Set 中会导致失败（不会改变 Set 的状态）。

**无序**：默认情况下，Set 不保证元素的顺序。但是，某些实现类（如 LinkedHashSet）可以按照插入顺序保存元素。

**不支持引**：Set 没有索引的概念，因此不能通过索引访问元素。

**常见实现**：

HashSet：基于哈希表，提供高效的添加和查找操作，但不保证元素的顺序。

TreeSet：基于红黑树，可以按照自然顺序或自定义比较器排序元素，但插入和查找操作较慢。

LinkedHashSet：结合了 HashSet 和 LinkedHashMap 的特性，维护元素的插入顺序。

## Collection集合遍历

假设顺序列表ArrayList中存储的元素是整型数字1~5，遍历每个元素，将每个元素顺序输出。（你能想到的所有

⽅式）

答：

代码：

import java.util.ArrayList;

import java.util.*List*;

public class Test2\_1 {

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

        ArrayList<Integer> list = **new** ArrayList<>();

        for (*int* i = 1; i <= 5; i++) {

            list.add(i);

        }

        System.out.println("集合原始数据");

        System.out.println(list);

**// 使用for循环**

        System.out.println("使用for循环遍历");

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

            System.out.print(list.get(i) + "\t");

        }

        System.out.println(" ");

        System.out.println("使用for-each循环遍历");

**// 使用foreach循环**

        for (Integer integer : list) {

            System.out.print(integer + "\t");

        }

        System.out.println(" ");

**// 使用stream流**

        System.out.println("使用stream流遍历");

        list.stream().forEach(System.out::print);

        System.out.println("\t");

**// 集合转数组遍历**

        System.out.println("集合转数组进行遍历");

        Object[] array = list.toArray();

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

            System.out.print(array[i] + "\t");

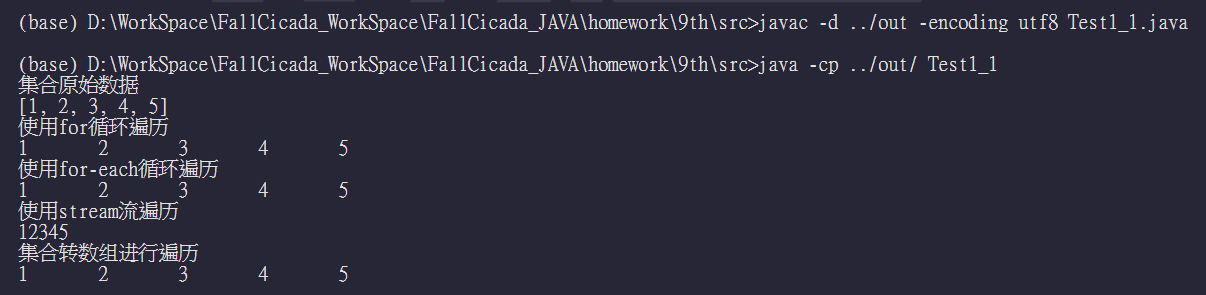
        }

        System.out.println(" ");

    }

}

**运行结果：**

****

## ⽅法应⽤

### contains()

ArrayList去除集合中字符串的重复值(字符串的内容相同则代表重复)

**答：**

**代码：**

import java.util.ArrayList;

**/\*\***

**\* ArrayList去除集合中字符串的重复值(字符串的内容相同则代表重复)**

**\*/**

public class Test3\_1 {

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

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

        list.add("hello");

        list.add("world");

        list.add("hello");

        list.add("java");

        list.add("hello");

        System.out.println("集合原始数据");

        System.out.println(list);

**//删除重复元素**

**// 使用for循环**

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

**// 从i+1开始**

**// 避免重复删除**

            for (*int* j = i + 1; j < list.size(); j++) {

**// 判断是否相等**

                if (list.get(i).equals(list.get(j))) {

**// 删除**

                    System.out.println("删除的元素" + list.remove(j));

                    j--;

                }

            }

        }

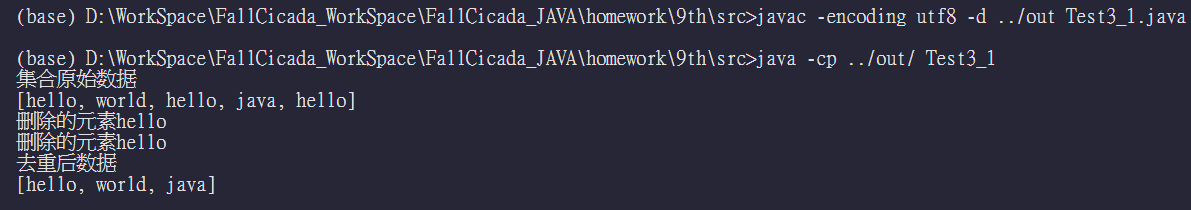
        System.out.println("去重后数据");

        System.out.println(list);

    }

}

**运行结果：**

****

### retainAll()、removeAll()

有两个集合list1和list2，list1中存放了1，2，3，4，5五个元素，list2中存放了4，5，6三个元素，按要求完成下列操作

1. 打印list1和list2的交集（list1和list2中都有的元素）
2. 打印list1和list2的差集（list1中有，list2中没有的元素）

**答：**

**代码：**

import java.util.ArrayList;

import java.util.*List*;

public class Test3\_2 {

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

*List*<Integer> list1 = **new** ArrayList<Integer>();

        list1.add(1);

        list1.add(2);

        list1.add(3);

        list1.add(4);

        list1.add(5);

*List*<Integer> list2 = **new** ArrayList<Integer>();

        list2.add(4);

        list2.add(5);

        list2.add(6);

**// 交集**

*List*<Integer> intersection = **new** ArrayList<Integer>();

        for (Integer i : list1) {

**// 判断list2中是否包含i**

            if (list2.contains(i)) {

**// 添加到交集中**

                intersection.add(i);

            }

        }

        System.out.println("交集：" + intersection);

**// 差集**

*List*<Integer> difference = **new** ArrayList<Integer>();

        for (Integer i : list1) {

**// 判断list2中是否不包含i**

            if (!list2.contains(i)) {

**// 添加到差集中**

                difference.add(i);

            }

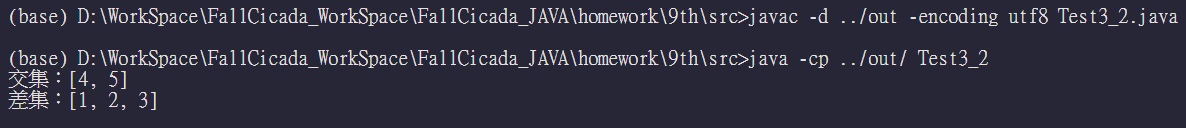
        }

        System.out.println("差集：" + difference);

    }

}

**运行结果：**



## 集合使⽤

### List集合的使⽤

定义两个集合，完成List集合的定义和遍历。

【要求】

定义⼀个集合list1存⼊20个随机整数筛选list1中的偶数元素

定义⼀个集合list2⽤来接收筛选出的偶数元素遍历list2

**答：**

**代码：**

import java.util.ArrayList;

import java.util.*List*;

public class Test4\_1 {

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

*List*<Integer> list1 = **new** ArrayList<Integer>();

*List*<Integer> list2 = **new** ArrayList<Integer>();

        for (*int* i = 0; i < 20; i++) {

            list1.add((*int*)(Math.random() \* 100));

        }

        for (*int* i = 0; i < list1.size(); i++) {

            if (list1.get(i) % 2 == 0) {

                list2.add(list1.get(i));

            }

        }

        System.out.println(list1);

        System.out.println(list2);

        for (*int* i = 0; i < list2.size(); i++) {

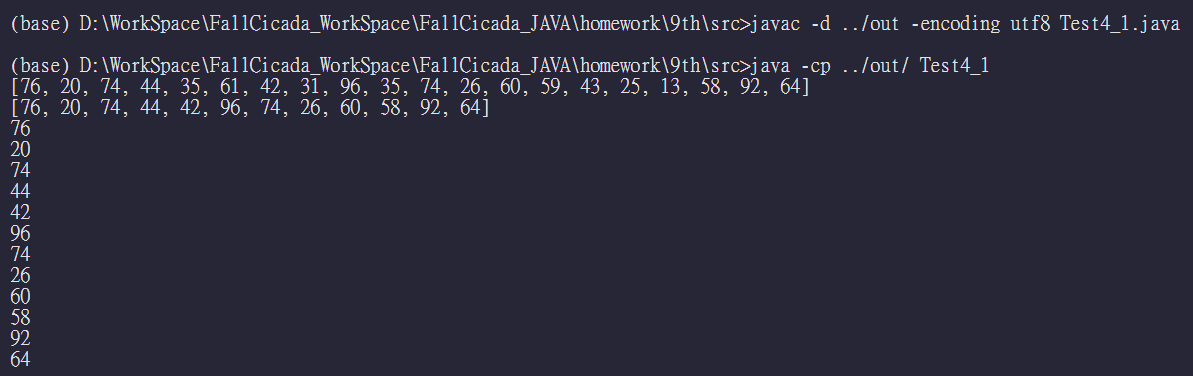
            System.out.println(list2.get(i));

        }

    }

}

**运行结果：**

****

### Set集合的使⽤

定义⼀个集合，完成Set集合的定义和遍历。

【要求】

Set集合中的元素为10个整数10个整数元素为1-20的随机数

10个整数元素互不重复

**答：**

**代码：**

import java.util.HashSet;

import java.util.Random;

import java.util.*Set*;

public class Test4\_2 {

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

**// 定义一个集合**

        System.out.println("集合的元素为10个整数10个整数元素为1-20的随机数");

*Set*<Integer> set = **new** HashSet<>();

        System.out.println("集合的元素个数为：" + set.size());

        Random r = **new** Random();

**// 循环添加**

        System.out.println("开始添加元素");

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

**// 添加**

            System.out.println("添加元素：" + (r.nextInt(20) + 1));

            set.add(r.nextInt(20) + 1);

        }

**// 遍历**

        System.out.println("遍历集合");

        for (Integer i : set) {

            System.out.println(i);

        }

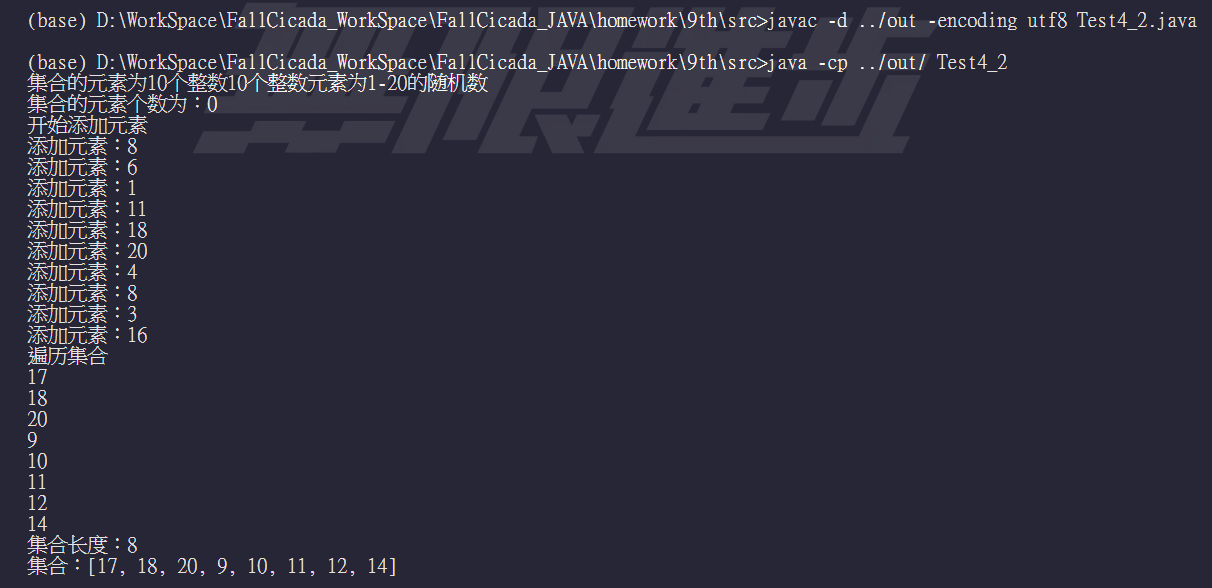
        System.out.println("集合长度：" + set.size());

        System.out.println("集合：" + set);

    }

}

运行结果：



## TreeSet排序问题

### 奇偶排序

将1-10按照奇数在前偶数在后，奇数正序，偶数倒序的

⽅式保存到Set集合中（排序）

【示例】

输出结果为：

1 [1，3，5，7，9，10，8，6，4，2]

**答：**

**代码：**

import java.util.LinkedHashSet;

import java.util.*Set*;

**/\*\***

**\* 将1-10按照奇数在前偶数在后，奇数正序，偶数倒序的**

**\* ⽅式保存到Set集合中（排序）**

**\*/**

public class Test5\_1 {

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

*Set*<Integer> set = **new** LinkedHashSet<>();

        for (*int* i = 1; i <= 10; i++) {

            if (i % 2 != 0) {

                set.add(i);

            }

        }

        System.out.println(set);

        for (*int* i = 10; i >= 1; i--) {

            if (i % 2 == 0) {

                set.add(i);

            }

        }

        System.out.println(set);

    }

}

**答：**

**代码：**

import java.util.*Comparator*;

import java.util.TreeSet;

**/\*\***

**\* 将1-10按照奇数在前偶数在后，奇数正序，偶数倒序的方式**

**\* 保存到Set集合中（排序）**

**\*/**

public class Test5\_1 {

**/\*\***

**\* 主函数入口**

**\* 本函数演示了如何使用TreeSet以及自定义比较器**

**\* @param** *args* **命令行参数**

**\*/**

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

**// 使用自定义比较器创建TreeSet实例**

        TreeSet<Integer> set = **new** TreeSet<>(**new** CustomComparator());

**// 添加1-10的所有数字**

        for (*int* i = 1; i <= 10; i++) {

            set.add(i);

        }

**// 打印结果**

        System.out.println(set);

    }

**// 自定义比较器类**

    static class CustomComparator implements *Comparator*<Integer> {

        @Override

        public *int* compare(Integer *o1*, Integer *o2*) {

**// 判断是否为奇数**

*boolean* isO1Odd = *o1* % 2 != 0; **// 将o1与2取模后判断是否不等于0，从而确定o1是否为奇数**

*boolean* isO2Odd = *o2* % 2 != 0; **// 将o2与2取模后判断是否不等于0，从而确定o2是否为奇数**

**// 如果都是奇数或都是偶数，则根据数值大小比较**

            if (isO1Odd == isO2Odd) { **// 比较两个布尔值是否相等，即判断o1和o2是否同为奇数或同为偶数**

                if (isO1Odd) { **// 当o1为奇数时**

**// 奇数正序**

                    return *o1*.compareTo(*o2*); **// 使用Integer对象的compareTo方法按自然顺序比较o1和o2**

                } else {

**// 偶数倒序**

                    return *o2*.compareTo(*o1*); **// 使用Integer对象的compareTo方法按自然顺序比较o2和o1，实现倒序排列**

                }

            } else {

**// 奇数排在偶数前面**

                return isO1Odd ? -1 : 1; **// 当o1为奇数时返回-1（表示o1小于o2），否则返回1（表示o1大于o2）**

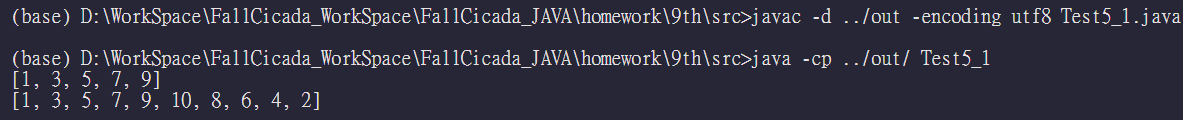
            }

        }

    }

}

**运行结果：**

****

### 成绩单应⽤

将学⽣按照成绩保存到集合中，并且名字叫tom的学⽣不管考多少分都位于班级的第⼀位。

代码结构：

public class Test5\_2\_ReportCards  {

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

**// 定义Set集合并传⼊⼀个⾃定义⽐较规则的Comparator**

**// 编写代码**

**// 测试**

        Student student1 = **new** Student("tom", 80);

        Student student2 = **new** Student("mike", 90);

        Student student3 = **new**

        Student("lily", 20);

        Student student4 = **new**

        Student("chris", 23);

        set.add(student3);

        set.add(student2);

        set.add(student1);

        set.add(student4);

        for (Student student : set) {

            System.out.println(student);

        }

    }

}

class Student {

    private String name;

    private *int* score;

    public Student(String *name*, *int* *score*) {

*super*();

*this*.name = *name*;

*this*.score = *score*;

    }

    public String getName() {

        return name;

    }

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

*this*.name = *name*;

    }

    public *int* getScore() {

        return score;

    }

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

*this*.score = *score*;

    }

    @Override

    public String toString() {

        return "Student [name=" + name + ", score=" + score + "]";

    }

}

**答：**

**代码：**

import java.util.*Comparator*;

import java.util.*Set*;

import java.util.TreeSet;

**/\***

**\* 将学⽣按照成绩保存到集合中，并且名字叫tom的学⽣不管考多少分都位于班级的第⼀位。**

**\* 代码结构：**

**\*/**

public class Test5\_2\_ReportCards {

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

**// 定义Set集合并传入一个自定义比较规则的Comparator**

*Set*<Student> set = **new** TreeSet<>(**new** StudentComparator());

**// 测试数据**

        Student student1 = **new** Student("tom", 80);

        Student student2 = **new** Student("mike", 90);

        Student student3 = **new** Student("lily", 20);

        Student student4 = **new** Student("chris", 23);

        set.add(student3);

        set.add(student2);

        set.add(student1);

        set.add(student4);

**// 输出结果**

        for (Student student : set) {

            System.out.println(student);

        }

    }

}

class Student {

    private String name;

    private *int* score;

    public Student(String *name*, *int* *score*) {

*this*.name = *name*;

*this*.score = *score*;

    }

    public String getName() {

        return name;

    }

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

*this*.name = *name*;

    }

    public *int* getScore() {

        return score;

    }

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

*this*.score = *score*;

    }

    @Override

    public String toString() {

        return "Student [name=" + name + ", score=" + score + "]";

    }

}

class StudentComparator implements *Comparator*<Student> {

**/\*\***

**\* 比较两个学生对象的顺序**

**\* 本方法主要用于排序学生，使得名字为"tom"的学生优先，其他学生则按照成绩降序排序**

**\***

**\* @param** *s1* **第一个学生对象，用于比较**

**\* @param** *s2* **第二个学生对象，用于比较**

**\* @return 如果s1的名字是"tom"，则s1优先，返回-1；如果s2的名字是"tom"，则s2优先，返回1；**

**\*         否则，根据成绩降序排序，成绩高的学生优先**

**\*/**

    @Override

    public *int* compare(Student *s1*, Student *s2*) {

**// 名字为 "tom" 的学生优先**

        if ("tom".equals(*s1*.getName())) {

            return -1;

        }

        if ("tom".equals(*s2*.getName())) {

            return 1;

        }

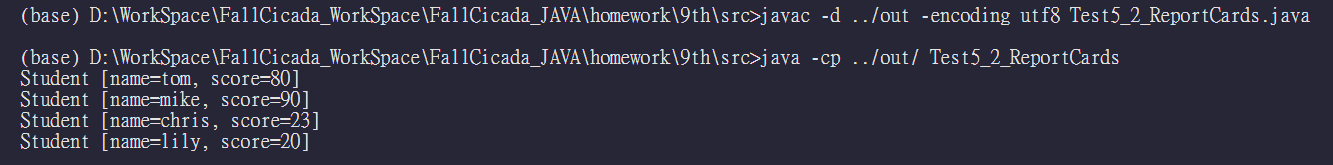
**// 其他学生按照成绩降序排序**

        return Integer.compare(*s2*.getScore(), *s1*.getScore());

    }

}

**运行结果：**



### 对⽐应⽤

通过两种⽅式实现：Comparable接⼝和Comparator接⼝

【要求】

设计Teacher类，包含属性id，name,age，创建

Teacher对象。

将对象保存在TreeSet集合中并且排序。

排序规则为：先通过名字排序，名字相同的时候通过年龄进⾏排序，年龄相同时通过id排序。

答：

代码：

import java.util.*Comparator*;

import java.util.TreeSet;

**/\*\***

**\* 通过两种方式实现：Comparable接口和Comparator接口**

**\*/**

public class Test5\_3 {

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

**// 使用自定义的Comparator进行排序**

        TreeSet<Teacher> treeSet = **new** TreeSet<>(**new** TeacherComparator());

        Teacher t1 = **new** Teacher("tom", 23, 1);

        Teacher t2 = **new** Teacher("lily", 24, 2);

        Teacher t3 = **new** Teacher("mike", 23, 3);

        Teacher t4 = **new** Teacher("lily", 24, 4);

        treeSet.add(t4);

        treeSet.add(t1);

        treeSet.add(t2);

        treeSet.add(t3);

        System.out.println(treeSet);

    }

}

class Teacher implements *Comparable*<Teacher>{

    private String name;

    private *int* age;

    private *int* id;

    public Teacher(String *name*, *int* *age*, *int* *id*) {

*this*.name = *name*;

*this*.age = *age*;

*this*.id = *id*;

    }

    public String getName() {

        return name;

    }

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

*this*.name = *name*;

    }

    public *int* getAge() {

        return age;

    }

    public *void* setAge(*int* *age*) {

*this*.age = *age*;

    }

    public *int* getId() {

        return id;

    }

    public *void* setId(*int* *id*) {

*this*.id = *id*;

    }

    @Override

    public String toString() {

        return "Teacher [name=" + name + ", age=" + age + ", id=" + id + "]";

    }

    @Override

    public *int* compareTo(Teacher *o*) {

*int* result = *this*.name.compareTo(*o*.getName());

        if (result == 0) {

            result = Integer.compare(*this*.age, *o*.getAge());

            if (result == 0) {

                result = Integer.compare(*this*.id, *o*.getId());

            }

        }

        return result;

    }

}

class TeacherComparator implements *Comparator*<Teacher>{

    @Override

    public *int* compare(Teacher *t1*, Teacher *t2*) {

*int* result = *t1*.getName().compareTo(*t2*.getName());

        if (result == 0) {

            result = Integer.compare(*t1*.getAge(), *t2*.getAge());

            if (result == 0) {

                result = Integer.compare(*t1*.getId(), *t2*.getId());

            }

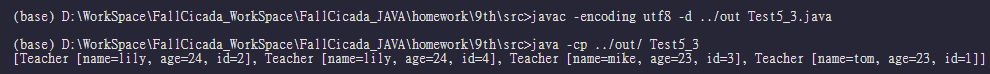
        }

        return result;

    }

}

运行结果：



## List集合⾃定义实现（选做题）

### ⾃定义实现ArrayList

实现MyList代码，模拟ArrayList的部分功能。代码结构：

public interface *MyList*{

    public *void* add(*int* *index*,Object *obj*);

**//在指定位置添加对象**

    public *void* add(Object *obj*);

**//在最后位置添加对象**

    public Object remove(*int* *index*);

**//删除指定位置上的对象**

    public *void* set(*int* *index*,Object *obj*);

**//修改指定位置上的数据**

    public Object get(*int* *index*);

**//获取指定位置上的数据**

    public *int* size();

**//获取当前数据结构当前的长度**

    public *void* clear();

**//清空所有的数据**

}

public class MyArrayList{

**//定义数组容器**

    private *Object*[] elementData;  **//默认容量为10**

    private static final *int* DEFAULT\_CAPACITY= 10;

**//实际ArrayList的大小**

    private *int* size;

**//提供有参构造器自定义初始容量**

    public MyArrayList(*int* *initialCapacity*) {

        if(initialCapacity< 0){

            throw **new** IllegalArgumentException("Illegal Capacity:" + initialCapacity);

        }

*this*.elementData= **new** *Object*[initialCapacity];

    }

**//无参构造初始化将默认10传递给有参构造器**

    public MyArrayList(){

*this*(DEFAULT\_CAPACITY);

    }

**//在尾部添加元素**

    public *void* add(*Object* *object*){

    }

**//在指定位置添加元素**

    public *void* add(*int* *index*, *Object* *object*){

        rangeCheckForAdd(index);

    }

**//获取指定索引元素**

    public *Object* get(*int* *index*){

        rangeCheckForAdd(index);

    }

**//检查数组是否越界**

    private *void* rangeCheckForAdd(*int* *index*) {

        if(index< 0|| index> size){

            throw **new** ArrayIndexOutOfBoundsException("数组越界异常");

        }

    }

**//删除指定索引元素**

    public *Object* remove(*int* *index*){

        rangeCheckForAdd(index);

    }

**//删除对象**

    public *boolean* remove(*Object* *o*){

    }

**//清空数据**

    public *void* clear(){

    }

**//扩容**

    private *void* grow(*int* *minCapacity*){

    }

**//测试**

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

*MyArrayList* my= **new** MyArrayList(1);

        my.add(1);

        my.add(2);

        my.add(3);

        System.out.println("获取第一个位置"+ my.get(1));

        my.remove(1);

        my.clear();

        System.out.println(my.size);

    }

}

答：

代码：

import java.util.Arrays;

public class MyArrayList {

**// 定义数组容器**

    private Object[] elementData;**//默认容量为10**

    private static final *int* DEFAULT\_CAPACITY = 10;

**// 实际ArrayList的大小**

    private *int* size;

**// 提供有参构造器自定义初始容量**

    public MyArrayList(*int* *initialCapacity*) {

**// 如果初始容量小于0，则抛出非法参数异常**

        if (*initialCapacity* < 0) {

            throw **new** IllegalArgumentException("Illegal Capacity:" + *initialCapacity*);

        }

**// 初始化数组容器**

*this*.elementData = **new** Object[*initialCapacity*];

    }

**// 无参构造初始化将默认10传递给有参构造器**

    public MyArrayList() {

**// 调用有参构造器，传入默认容量**

*this*(DEFAULT\_CAPACITY);

    }

**// 在尾部添加元素**

    public *void* add(Object *object*) {

**// 检查是否需要扩容**

        ensureCapacity(size + 1);

**// 将元素添加到数组末尾**

        elementData[size++] = *object*;

    }

**// 在指定位置添加元素**

    public *void* add(*int* *index*, Object *object*) {

**// 检查索引是否越界**

        rangeCheckForAdd(*index*);

**// 检查是否需要扩容**

        ensureCapacity(size + 1);

**// 将指定位置后的元素向后移动一位**

        System.arraycopy(elementData, *index*, elementData, *index* + 1, size - *index*);

**// 在指定位置插入新元素**

        elementData[*index*] = *object*;

**// 更新实际大小**

        size++;

    }

**// 获取指定索引元素**

    public Object get(*int* *index*) {

**// 检查索引是否越界**

        rangeCheckForAdd(*index*);

**// 返回指定索引处的元素**

        return elementData[*index*];

    }

**// 检查数组是否越界**

    private *void* rangeCheckForAdd(*int* *index*) {

**// 如果索引小于0或大于实际大小，则抛出数组越界异常**

        if (*index* < 0 || *index* > size) {

            throw **new** ArrayIndexOutOfBoundsException("数组越界异常");

        }

    }

**// 删除指定索引元素**

    public Object remove(*int* *index*) {

**// 检查索引是否越界**

        rangeCheckForAdd(*index*);

**// 获取要删除的元素**

        Object oldValue = elementData[*index*];

**// 计算要移动的元素数量**

*int* numMoved = size - *index* - 1;

**// 如果要移动的元素数量大于0，则进行数组复制**

        if (numMoved > 0) {

            System.arraycopy(elementData, *index* + 1, elementData, *index*, numMoved);

        }

**// 将最后一个元素置为null，帮助GC**

        elementData[--size] = null;

**// 返回被删除的元素**

        return oldValue;

    }

**// 删除对象**

    public *boolean* remove(Object *o*) {

**// 如果对象为null，则遍历数组找到第一个null元素并删除**

        if (*o* == null) {

            for (*int* index = 0; index < size; index++) {

                if (elementData[index] == null) {

                    remove(index);

                    return true;

                }

            }

        } else {

**// 如果对象不为null，则遍历数组找到第一个相等的元素并删除**

            for (*int* index = 0; index < size; index++) {

                if (*o*.equals(elementData[index])) {

                    remove(index);

                    return true;

                }

            }

        }

**// 如果没有找到要删除的元素，则返回false**

        return false;

    }

**// 清空数据**

    public *void* clear() {

**// 遍历数组，将每个元素置为null，帮助GC**

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

            elementData[i] = null;

        }

**// 重置实际大小为0**

        size = 0;

    }

**// 扩容**

    private *void* grow(*int* *minCapacity*) {

**// 获取当前数组容量**

*int* oldCapacity = elementData.length;

**// 新容量为旧容量的1.5倍**

*int* newCapacity = oldCapacity + (oldCapacity >> 1);

**// 如果新容量小于最小需要的容量，则将新容量设为最小需要的容量**

        if (newCapacity < *minCapacity*) {

            newCapacity = *minCapacity*;

        }

**// 将数组复制到新容量的数组中**

        elementData = Arrays.copyOf(elementData, newCapacity);

    }

**// 确保容量足够**

    private *void* ensureCapacity(*int* *minCapacity*) {

**// 如果最小需要的容量大于当前数组容量，则进行扩容**

        if (*minCapacity* - elementData.length > 0) {

            grow(*minCapacity*);

        }

    }

    @Override

    public String toString() {

    StringBuilder sb = **new** StringBuilder();

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

        sb.append(elementData[i]).append(",");

    }

**// 返回拼接后的字符串，并去除最后一个逗号**

    if (sb.length() == 0) {

        return "";

    }

    String result = "[" +sb.toString().substring(0, sb.length() - 1) + "]";

    return result;

}

**// 测试**

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

**// 创建一个初始容量为1的MyArrayList实例**

        MyArrayList my = **new** MyArrayList(1);

**// 添加元素1**

        my.add(1);

**// 添加元素2**

        my.add(2);

**// 添加元素3**

        my.add(3);

**// 打印**

        System.out.println(my.toString());

**// 获取第一个位置的元素并打印**

        System.out.println("获取第一个位置" + my.get(1));

        System.out.println(my.size);

**// 删除第一个位置的元素**

        my.remove(1);

**// 打印剩余元素**

        System.out.println(my.toString());

**// 清空所有元素**

        my.clear();

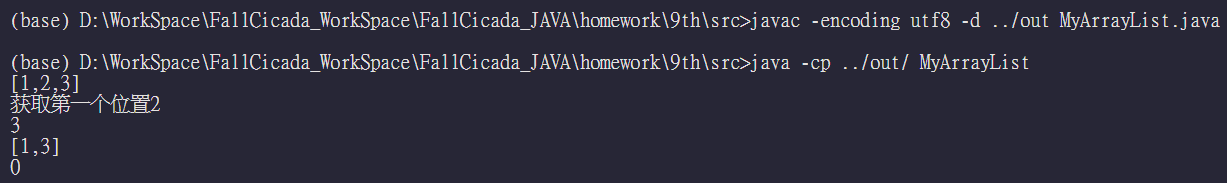
**// 打印实际大小**

        System.out.println(my.size);

    }

}

运行结果：



### ⾃定义实现LinkedList

完成MyLinkedList代码，模拟LinkedList的部分功能代码结构：

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

public class MyLinkedList{ private class Node{

//存放元素的值

private Object data;

//存放上⼀个节点private Node pre;

//存放下⼀个节点

private Node next;

}

//补全以下四种⽅法

public void add(Object o){}

public void add(int index,Object o){} public void remove(int index){} public Object get(int index){}

}

泛型类版本：

1

1. public class MyLinkedList<T> {
2. private Node<T> first;// 定义头节点
3. private Node<T> last;// 定义尾节点
4. private int size;// 链表的⻓度

6

1. private class Node<T> {
2. private T data;// 存放元素的值
3. private Node<T> pre; // 存放上⼀个节点
4. private Node<T> next;// 存放下⼀个节点

11 }

12

1. // 补全以下四种⽅法
2. // 添加元素⽅法
3. public void add(T t) { 16

17 }

18

1. // 将元素添加到指定索引⽅法
2. public void add(int index, T t) throws Exception {
3. // 校验参数
4. checkIndex(index); 23

24 }

25

1. // 根据索引删除元素⽅法
2. public void remove(int index) throws Exception {
3. // 校验参数

|  |  |  |
| --- | --- | --- |
| 29 | checkIndex(index); |  |
| 30 |  |  |
| 31 | } |  |
| 32 |  |  |
| 33 | // 根据索引获取元素⽅法 |  |
| 34 | private Node<T> get(int index) | throws |

Exception {

35

36 }

37

1. // 获取MyLinkedList⻓度⽅法
2. public int getSize() {
3. return this.size; 41 }

42

1. // 展示数据⽅法
2. public void show() throws Exception {
3. for (int i = 0; i < getSize(); i++)

{

1. if (i == getSize() - 1) { 47

System.out.println(get(i).data);

1. }else {
2. System.out.print(get(i).data

+ " -> ");

50 }

51 }

52 }

53

1. // 校验参数⽅法
2. private void checkIndex(int index) throws Exception {
3. if (index < 0 || index >= getSize())

{

1. throw new Exception("参数index不合法");

58 }

59 }

60

1. // 测试
2. public static void main(String[] args) throws Exception {

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 63 |  | MyLinkedList<Integer> | linkedList | = |
|  | new | MyLinkedList<>(); |  |  |
| 64 |  | linkedList.add(1); |  |  |
| 65 |  | linkedList.add(2); |  |  |
| 66 |  | linkedList.add(3); |  |  |
| 67 |  | linkedList.add(4); |  |  |
| 68 |  | linkedList.add(5); |  |  |
| 69 |  | linkedList.show(); |  |  |
| 70 |  | linkedList.add(2, 1); |  |  |
| 71 |  | linkedList.show(); |  |  |
| 72 |  | linkedList.remove(2); |  |  |
| 73 |  | linkedList.show(); |  |  |

74

75

}

}

答：

代码：

public class MyLinkedList<*T*> {

    private Node<*T*> first; **// 定义头节点**

    private Node<*T*> last; **// 定义尾节点**

    private *int* size; **// 链表的长度**

**// 定义节点类**

    private class Node<*T*> {

        private *T* data; **// 存放元素的值**

        private Node<*T*> pre; **// 存放上一个节点**

        private Node<*T*> next; **// 存放下一个节点**

**// 节点构造方法**

        public Node(*T* *data*) {

*this*.data = *data*;

        }

    }

**// 添加元素方法**

    public *void* add(*T* *t*) {

        Node<*T*> newNode = **new** Node<>(*t*); **// 创建新节点**

        if (first == null) { **// 如果链表为空**

            first = newNode; **// 新节点为头节点**

            last = newNode; **// 新节点为尾节点**

        } else { **// 如果链表不为空**

            last.next = newNode; **// 将新节点添加到链表末尾**

            newNode.pre = last; **// 设置新节点的前驱节点**

            last = newNode; **// 更新尾节点**

**//交换节点位置(个人理解)**

        }

        size++; **// 链表长度加1**

    }

**// 将元素添加到指定索引方法**

    public *void* add(*int* *index*, *T* *t*) throws Exception {

        checkIndex(*index*); **// 校验参数**

        Node<*T*> newNode = **new** Node<>(*t*); **// 创建新节点**

        if (*index* == 0) { **// 如果索引为0**

            newNode.next = first; **// 新节点的下一个节点为头节点**

            first.pre = newNode; **// 头节点的前驱节点为新节点**

            first = newNode; **// 更新头节点**

        } else { **// 如果索引不为0**

            Node<*T*> current = get(*index*); **// 获取指定索引的节点**

            Node<*T*> previous = current.pre; **// 获取指定索引节点的前驱节点**

            previous.next = newNode; **// 前驱节点的下一个节点为新节点**

            newNode.pre = previous; **// 新节点的前驱节点为前驱节点**

            newNode.next = current; **// 新节点的下一个节点为指定索引节点**

            current.pre = newNode; **// 指定索引节点的前驱节点为新节点**

        }

        size++; **// 链表长度加1**

    }

**// 根据索引删除元素方法**

    public *void* remove(*int* *index*) throws Exception {

        checkIndex(*index*); **// 校验参数**

        Node<*T*> current = get(*index*); **// 获取指定索引的节点**

        if (current == first) { **// 如果删除的是头节点**

            first = current.next; **// 更新头节点**

            if (first != null) { **// 如果新的头节点不为空**

                first.pre = null; **// 新的头节点的前驱节点为空**

            }

        } else if (current == last) { **// 如果删除的是尾节点**

            last = current.pre; **// 更新尾节点**

            if (last != null) { **// 如果新的尾节点不为空**

                last.next = null; **// 新的尾节点的下一个节点为空**

            }

        } else { **// 如果删除的是中间节点**

            Node<*T*> previous = current.pre; **// 获取前驱节点**

            Node<*T*> next = current.next; **// 获取后继节点**

            previous.next = next; **// 前驱节点的下一个节点为后继节点**

            next.pre = previous; **// 后继节点的前驱节点为前驱节点**

        }

        size--; **// 链表长度减1**

    }

**// 根据索引获取元素方法**

    private Node<*T*> get(*int* *index*) throws Exception {

        checkIndex(*index*); **// 校验参数**

        Node<*T*> current = first; **// 从头节点开始遍历**

        for (*int* i = 0; i < *index*; i++) { **// 遍历到指定索引**

            current = current.next; **// 移动到下一个节点**

        }

        return current; **// 返回指定索引的节点**

    }

**// 获取MyLinkedList长度方法**

    public *int* getSize() {

        return *this*.size; **// 返回链表长度**

    }

**// 展示数据方法**

    public *void* show() throws Exception {

        for (*int* i = 0; i < getSize(); i++) { **// 遍历链表**

            if (i == getSize() - 1) { **// 如果是最后一个元素**

                System.out.println(get(i).data); **// 打印元素值并换行**

            } else { **// 如果不是最后一个元素**

                System.out.print(get(i).data + "->"); **// 打印元素值和箭头**

            }

        }

    }

**// 校验参数方法**

    private *void* checkIndex(*int* *index*) throws Exception {

        if (*index* < 0 || *index* >= getSize()) { **// 如果索引不合法**

            throw **new** Exception("参数index不合法"); **// 抛出异常**

        }

    }

**// 测试**

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

        MyLinkedList<Integer> linkedList = **new** MyLinkedList<>(); **// 创建链表实例**

        linkedList.add(1); **// 添加元素1**

        linkedList.add(2); **// 添加元素2**

        linkedList.add(3); **// 添加元素3**

        linkedList.add(4); **// 添加元素4**

        linkedList.add(5); **// ........**

        linkedList.show(); **// 展示链表**

        linkedList.add(2, 1); **// 在索引2处添加元素1**

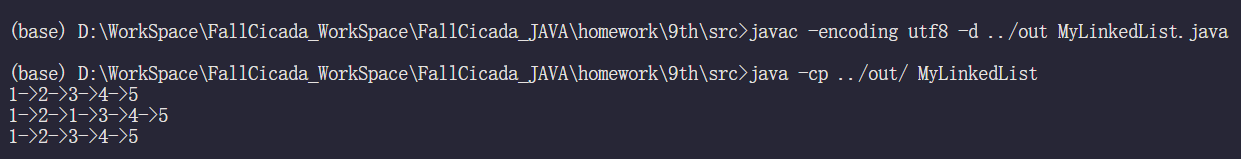
        linkedList.show(); **// 展示链表**

        linkedList.remove(2); **// 删除索引2处的元素**

        linkedList.show(); **// 展示链表**

    }

}

运行结果：  


## Map集合问答

1. map可以使⽤迭代器遍历吗？为什么？

答：可以使用迭代器来遍历Map容器中的元素。但是，因为Map接口本身并不直接提供迭代器，所以需要通过获取Map的键值对集合，然后在这个集合上调用迭代器来进行遍历。

1. map的键和值能为空吗？为什么？

答：

1. **键（Key）为null**：
   * 对于HashMap，允许单个null键。如果有多个键设置为null，只有第一个null键会被记录，后续的null键会覆盖之前的值。
   * 对于TreeMap，虽然理论上可以插入null键，但实际上TreeMap需要比较键的大小以保持排序顺序，而null不能与其他对象进行比较，因此尝试插入null键会抛出NullPointerException。
   * 其他一些实现类可能会有不同的处理方式，有些可能完全不允许null键。
2. **值（Value）为null**：
   * 对于大多数Map实现来说，允许将null作为值存储。这意味着可以将一个键与null值相关联。
   * 当然，这也意味着当用户查询这个键对应的值时，需要检查是否可能是null。

## Map集合使⽤和遍历

1）创建⼀个Map集合，⾥⾯有如下元素{⾹蕉 = 5.6 ，樱桃 = 25 ，桃⼦ = 5.6 ， 苹果 = 2.3}，按要求完成：

遍历该map集合，并且统计有多少种⽔果（key）⽤两种⽅式进⾏遍历

将“⾹蕉”的价格修改为10.9，并删除桃⼦这组数据， 将修改后的map中所有key和value都输出到控制台

答：

代码：

import java.util.HashMap;

import java.util.*Map*;

public class Test8\_1 {

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

**// 创建Map集合**

*Map*<String, Double> fruitPrices = **new** HashMap<>();

        fruitPrices.put("香蕉", 5.6);

        fruitPrices.put("樱桃", 25.0);

        fruitPrices.put("桃子", 5.6);

        fruitPrices.put("苹果", 2.3);

**// 遍历Map集合并统计水果种类**

        System.out.println("使用entrySet遍历:");

*int* count = 0;

        for (*Map*.*Entry*<String, Double> entry : fruitPrices.entrySet()) {

            System.out.println(entry.getKey() + " = " + entry.getValue());

            count++;

        }

        System.out.println("水果种类数: " + count);

        System.out.println("使用keySet遍历:");

        count = 0;

        for (String key : fruitPrices.keySet()) {

            System.out.println(key + " = " + fruitPrices.get(key));

            count++;

        }

        System.out.println("水果种类数: " + count);

**// 修改“香蕉”的价格为10.9，并删除“桃子”这组数据**

        fruitPrices.put("香蕉", 10.9);

        fruitPrices.remove("桃子");

        System.out.println("修改香蕉单价:5.6 => 10.9");

        System.out.println("删除桃子");

**// 输出修改后的Map集合**

        System.out.println("修改后的Map集合:");

        count = 0;

        for (*Map*.*Entry*<String, Double> entry : fruitPrices.entrySet()) {

            System.out.println(entry.getKey() + " = " + entry.getValue());

            count++;

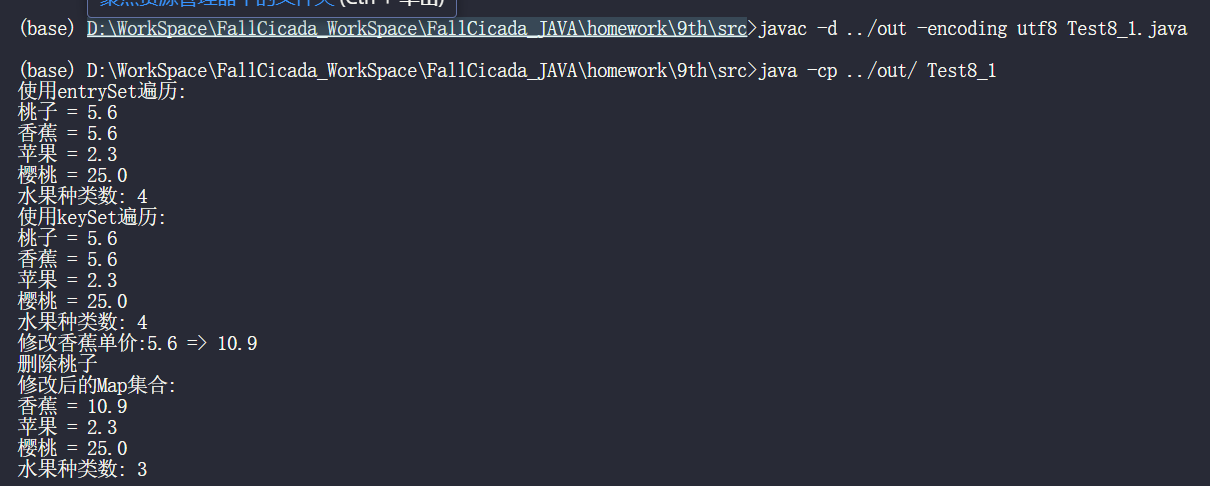
        }

        System.out.println("水果种类数: " + count);

    }

}

运行结果：



2）统计字符串中每个字符出现的次数，String s = "aabbddccaefdd";

【提示】

可使⽤Map<Character, Integer>集合，键存放字符，值存放出现的次数

答：

代码：

import java.util.HashMap;

import java.util.*Map*;

public class Test8\_2 {

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

        String s = "aabbddccaefdd";

*Map*<Character, Integer> charCountMap = **new** HashMap<>();

**// 遍历字符串中的每个字符，并统计其出现的次数**

        for (*char* c : s.toCharArray()) {

**// 如果字符c已经在charCountMap中，则获取其当前的计数值，否则返回0，然后加1**

            charCountMap.put(c, charCountMap.getOrDefault(c, 0) + 1);

        }

**// 输出每个字符及其出现的次数**

        for (*Map*.*Entry*<Character, Integer> entry : charCountMap.entrySet()) {

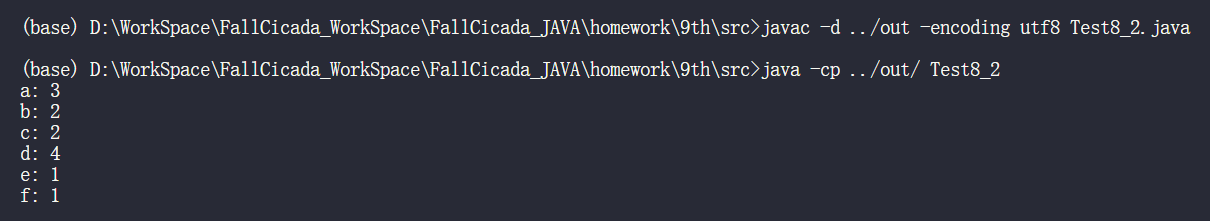
            System.out.println(entry.getKey() + ": " + entry.getValue());

        }

    }

}

运行结果：



## Map集合常⽤⽅法

按要求完成以下步骤：

1. 键盘录⼊整⾏字符串(sc.nextLine())，遇到quit结束录

⼊，录⼊字符串格式为：学号.姓名.年龄.分数

【示例】

001.zs.20.68

003.tom.19.78

quit

1. 拆解上述整⾏的字符串得到属性值，然后实例化学

⽣对象，按照"."分割字符串

1） split()⽅法的应⽤2）String --> Integer 的转换

1. 将学⽣对象添加到Map<学⽣,学号>集合中要求：添加时 按照成绩逆序排列
2. 遍历集合
3. 判断 003号学⽣是否存在如果存在，删除该学⽣
4. 再次遍历集合(采⽤ 另⼀种⽅式 遍历)

Student实体类代码结构：

1

2

3

4

5

6

7

8

9

10

public class Student { private String id; private String name; private int age;

private double score;

public Student() {

}

11

12

13

14

15

16

17

18

public Student(String id, String name, int age, double score) {

this.id = id; this.name = name; this.age = age; this.score = score;

}

public String getId() {

return id;

|  |  |  |  |
| --- | --- | --- | --- |
| 19 | } |  | |
| 20 |  |
| 21 | public void setId(String id) { |
| 22 | this.id = id; |
| 23 | } |
| 24 |  |
| 25 | public String getName() { |
| 26 | return name; |
| 27 | } |
| 28 |  |
| 29 | public void setName(String name) | { | |
| 30 | this.name = name; |  | |
| 31 | } |  | |
| 32 |  |  | |
| 33 | public int getAge() { |  | |
| 34 | return age; |  | |
| 35 | } |  | |
| 36 |  |  | |
| 37 | public void setAge(int age) { |  | |
| 38 | this.age = age; |  | |
| 39 | } |  | |
| 40 |  |  | |
| 41 | public double getScore() { | |  |
| 42 | return score; | |  |
| 43 | } | |  |
| 44 |  | |  |
| 45 | public void setScore(double score) | | { |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 46 | this.score = score; | | | | | |  |
| 47 | } | | | | | |
| 48 |  | | | | | |
| 49 | @Override | | | | | |
| 50 | public String toString() { | | | | | |
| 51 | return "Student{" + | | | | | |
| 52 | "id='" + id + '\'' + | | | | | |
| 53 | ", name='" + name + '\'' | | | | | | + |
| 54 | ", age=" + age + | | | | | |  |
| 55 |  |  | ", score=" | + | score | + | |
| 56 |  |  | '}'; |  |  |  | |
| 57 |  | } |  |  |  |  | |
| 58 | } |  |  |  |  |  | |

答:

代码：

Student类：

package student;

public class Student {

    private String id;

    private String name;

    private *int* age;

    private *double* score;

    public Student() {

    }

    public Student(String *id*, String *name*, *int* *age*, *double* *score*) {

*this*.id = *id*;

*this*.name = *name*;

*this*.age = *age*;

*this*.score = *score*;

    }

    public String getId() {

        return id;

    }

    public *void* setId(String *id*) {

*this*.id = *id*;

    }

    public String getName() {

        return name;

    }

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

*this*.name = *name*;

    }

    public *int* getAge() {

        return age;

    }

    public *void* setAge(*int* *age*) {

*this*.age = *age*;

    }

    public *double* getScore() {

        return score;

    }

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

*this*.score = *score*;

    }

    @Override

    public String toString() {

        return "Student{" +

        "id='" + id + '\'' +

        ", name='" + name + '\'' +

        ", age=" + age +

        ", score=" + score +

        '}';

    }

}

测试类：

package student;

import java.util.*Comparator*;

import java.util.HashMap;

import java.util.*Map*;

import java.util.Scanner;

import java.util.TreeMap;

public class Test9\_1 {

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

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

*Map*<Student, String> studentMap = **new** TreeMap<>(**new** Comparator<Student>() {

            @Override

            public *int* compare(Student *s1*, Student *s2*) {

                return Double.compare(*s2*.getScore(), *s1*.getScore());

            }

        });

        while (true) {

            String input = sc.nextLine();

            if (input.equals("quit")) {

                break;

            }

            String[] parts = input.split("\\.");

            String id = parts[0];

            String name = parts[1];

*int* age = Integer.parseInt(parts[2]);

*double* score = Double.parseDouble(parts[3]);

            Student student = **new** Student(id, name, age, score);

            studentMap.put(student, id);

        }

        System.out.println("学生信息如下:");

**// 遍历集合**

        for (*Map*.*Entry*<Student, String> entry : studentMap.entrySet()) {

            System.out.println(entry.getKey());

        }

**// 判断 003 号学生是否存在并删除**

        Student studentToRemove = null;

        for (Student student : studentMap.keySet()) {

            if (student.getId().equals("003")) {

                studentToRemove = student;

                break;

            }

        }

        System.out.println("是否删除003号学生: " + (studentToRemove != null));

        if (studentToRemove != null) {

            studentMap.remove(studentToRemove);

        }

**// 再次遍历集合，采用另一种方式遍历**

        System.out.println("删除003号后，剩余学生信息如下:");

        studentMap.forEach((*student*, *id*) *->* {

            System.out.println(*student*);

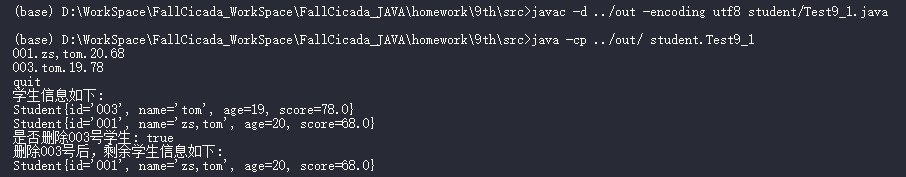
        });

        sc.close();

    }

}

运行结果：



# 泛型

## 泛型概念

使⽤泛型的好处是，什么是泛型擦除

答：  
在Java中，泛型（Generics）是一种允许在类、接口和方法中使用类型参数的功能。使用泛型的好处包括：

1. **类型安全**：编译器可以进行类型检查，确保只有指定类型的对象被传递给泛型类或方法，从而避免了运行时的ClassCastException异常。
2. **代码重用**：通过使用泛型类或接口，可以编写能够操作不同类型对象的代码，而不需要为每一种类型都编写新的类或方法。
3. **清晰性**：泛型增强了代码的可读性和意图表达，使得其他开发者更容易理解代码。

然而，在Java中，泛型的信息是在编译期被“擦除”的，这意味着在运行时，泛型信息不会保留。这种现象被称为**泛型擦除**（Generic Erasure）。具体来说：

* 在编译期间，Java编译器会检查泛型相关的代码，以确保它们符合类型安全规则。
* 当编译后的字节码被加载到JVM时，所有的类型参数会被替换为具体的类型（如Object），并且所有的泛型信息都会丢失。
* 运行时，你无法获取到泛型的实际类型，只能知道它是Object或者是哪个基本类型。

由于泛型擦除的存在，你不能在运行时获取到泛型的实际类型，也不能创建泛型类型的数组。此外，当你使用instanceof操作符时，它也不会考虑泛型类型。因此，在设计泛型类或方法时，应该遵循良好的编程实践，避免依赖于运行时的类型信息。

## 泛型使⽤

定义⼀个通⽤的(带泛型)的⽅法，该⽅法可以返回

Collection集合中的最⼤值

答：

代码：  
import java.util.Arrays;

import java.util.*Collection*;

import java.util.*List*;

**/\*\***

**\* Test11**

**\*/**

public class Test11 {

**/\*\***

**\* 返回 Collection 中的最大值。**

**\***

**\* @param** *<T>* **集合中元素的类型，必须实现了 Comparable 接口。**

**\* @param** *collection* **要查找最大值的 Collection。**

**\* @return 最大值，如果 Collection 为空，则返回 null。**

**\*/**

    public static <*T* extends *Comparable*<*T*>> *T* max(*Collection*<*T*> *collection*) {

        if (*collection* == null || *collection*.isEmpty()) {

            return null;

        }

*T* maxElement = null;

        for (*T* element : *collection*) {

            if (maxElement == null || element.compareTo(maxElement) > 0) {

                maxElement = element;

            }

        }

        return maxElement;

    }

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

**// 示例：使用 Integer 类型的 List**

*List*<Integer> numbers = Arrays.asList(5, 2, 9, 1, 5, 6);

        Integer maxValue = Test11.max(numbers);

        System.out.println("The maximum int value is: " + maxValue);

**//示例使用Double类型的List**

*List*<Double> doubleNumders = Arrays.asList(5.5, 2.2, 9.9, 1.1, 5.5, 6.6);

        Double maxDouble = Test11.max(doubleNumders);

        System.out.println("The maximum double value is: " + maxDouble);

**// 示例：使用 String 类型的 List**

*List*<String> strings = Arrays.asList("apple", "banana", "cherry");

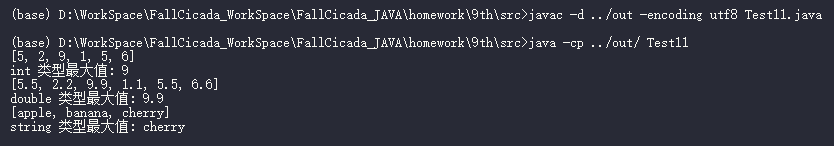
        String maxString = Test11.max(strings);

        System.out.println("The maximum string is: " + maxString);

    }

}

运行结果：



# 注解

## 注解概念

简述以下四个元注解的作⽤

@Target @Retention @Documented @Inherited

答：

1. **@Target**
   * @Target 元注解用于指定一个注解能应用于哪些程序元素上。它可以用来限制注解的使用范围。
   * @Target 可以指定注解适用于类、方法、字段、构造函数、局部变量等。
   * 常见的ElementType有 TYPE（类、接口、枚举）、METHOD（方法）、FIELD（字段）、PARAMETER（参数）等。
   * 例如，如果想要创建一个仅能在方法上使用的注解，可以在该注解的定义中使用 @Target(ElementType.METHOD)。
2. **@Retention**
   * @Retention 元注解用于指定注解的生命周期。
   * 注解的生命周期可以是 SOURCE（仅存在于源码阶段）、CLASS（存在于class文件中，但是虚拟机加载类时不关心）、RUNTIME（在运行时可以通过反射访问）。
   * 通过设置不同的保留策略，可以控制注解在不同阶段的行为，比如仅用于编译时检查的注解可以设置为 SOURCE 或 CLASS，而那些需要在运行时访问的注解则应该设置为 RUNTIME。
3. **@Documented**
   * @Documented 元注解用于标记某个注解是否会被文档工具（如 Javadoc）记录。
   * 如果一个注解被标记为 @Documented，那么当使用如 Javadoc 这样的工具生成文档时，这个注解将会出现在生成的文档中。
   * 通常用于那些开发者希望在API文档中显示出来的注解。
4. **@Inherited**
   * @Inherited 元注解用于指定一个注解是否可以由子类继承。
   * 如果一个注解类被声明为 @Inherited，那么当此注解应用于类时，它的子类将自动拥有这个注解。
   * 这对于那些希望在类层次结构中保持一致性的注解特别有用。例如，如果你有一个注解表示安全性敏感的操作，那么你可能希望所有继承了该类的子类也能具有相同的注解。

# 斗地主

答：

代码：

import java.util.ArrayList;

import java.util.Collections;

import java.util.*List*;

import java.util.*Comparator*;

enum *Suit* {

    HEARTS("红心"), DIAMONDS("方片"), CLUBS("梅花"), SPADES("黑桃"), JOKER("JOKER");

    private final String symbol;

*Suit*(String *symbol*) {

*this*.symbol = *symbol*;

    }

    @Override

    public String toString() {

        return *this*.symbol;

    }

}

enum *Rank* {

    THREE("3"), FOUR("4"), FIVE("5"), SIX("6"), SEVEN("7"), EIGHT("8"), NINE("9"), TEN("10"),

    JACK("J"), QUEEN("Q"), KING("K"), ACE("A"), TWO("2"), SMALL\_JOKER("小王"), BIG\_JOKER("大王");

    private final String symbol;

*Rank*(String *symbol*) {

*this*.symbol = *symbol*;

    }

    @Override

    public String toString() {

        return *this*.symbol;

    }

}

class Card {

*Suit* suit;

*Rank* rank;

    Card(*Suit* *suit*, *Rank* *rank*) {

*this*.suit = *suit*;

*this*.rank = *rank*;

    }

    @Override

    public String toString() {

        if (suit == *Suit*.JOKER) {

            return rank.toString();

        }

        return suit.toString() + rank.toString();

    }

}

public class PokerGame {

    private static *List*<Card> generateDeck() {

*List*<Card> deck = **new** ArrayList<>();

        for (*Suit* suit : *Suit*.values()) {

            if (suit == *Suit*.JOKER) continue;

            for (*Rank* rank : *Rank*.values()) {

                if (rank == *Rank*.SMALL\_JOKER || rank == *Rank*.BIG\_JOKER) continue;

                deck.add(**new** Card(suit, rank));

            }

        }

        deck.add(**new** Card(*Suit*.JOKER, *Rank*.SMALL\_JOKER));

        deck.add(**new** Card(*Suit*.JOKER, *Rank*.BIG\_JOKER));

        return deck;

    }

    private static *void* shuffleDeck(*List*<Card> *deck*) {

        Collections.shuffle(*deck*);

    }

    private static *void* dealCards(*List*<Card> *deck*, *List*<Card> *player1*, *List*<Card> *player2*, *List*<Card> *player3*, *List*<Card> *bottom*) {

        for (*int* i = 0; i < *deck*.size() - 3; i++) {

            if (i % 3 == 0) {

*player1*.add(*deck*.get(i));

            } else if (i % 3 == 1) {

*player2*.add(*deck*.get(i));

            } else {

*player3*.add(*deck*.get(i));

            }

        }

*bottom*.add(*deck*.get(*deck*.size() - 3));

*bottom*.add(*deck*.get(*deck*.size() - 2));

*bottom*.add(*deck*.get(*deck*.size() - 1));

    }

    private static *void* sortCards(*List*<Card> *cards*) {

*cards*.sort(*Comparator*.comparing((Card *card*) *->* *card*.rank.ordinal()).thenComparing(*card* *->* *card*.suit.ordinal()));

    }

    private static *void* printPlayerCards(String *player*, *List*<Card> *cards*) {

        System.out.print(*player* + ": [");

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

            System.out.print(*cards*.get(i));

            if (i < *cards*.size() - 1) {

                System.out.print(", ");

            }

        }

        System.out.println("]");

    }

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

*List*<Card> deck = generateDeck();

        shuffleDeck(deck);

*List*<Card> player1 = **new** ArrayList<>();

*List*<Card> player2 = **new** ArrayList<>();

*List*<Card> player3 = **new** ArrayList<>();

*List*<Card> bottom = **new** ArrayList<>();

        dealCards(deck, player1, player2, player3, bottom);

        sortCards(player1);

        sortCards(player2);

        sortCards(player3);

        sortCards(bottom);

        printPlayerCards("dipai", bottom);

        printPlayerCards("player1", player1);

        printPlayerCards("player2", player2);

        printPlayerCards("player3", player3);

    }

}

运行结果：

