Java进阶1 第5天

**【学习目标】理解、了解、应用、记忆**

通过今天的学习，参训学员能够：（解释的时候说出二级目标的掌握程度）

1. **【应用】双列集合Map**
2. 【理解】能够阐述Java中Map的概念
3. 【应用】能够独立使用Map集合的基本功能
4. 【应用】能够独立使用遍历方式1对Map集合进行遍历
5. 【应用】能够独立使用遍历方式2对Map集合进行遍历
6. 【理解】能够阐述LinkedHashMap的特点
7. **【应用】集合相关其他**
8. 【应用】能够独立使用可变参数
9. 【应用】能够独立使用Collections类的shuffle方法
10. 【应用】能够独立使用Collections类的sort方法
11. 【应用】能够独立使用Collections类的binarySearch方法
12. 【应用】能够独立使用Arrays类的toString方法
13. 【应用】能够独立完成数组转集合的操作
14. 【应用】能够独立完成集合转数组的操作
15. **【应用】斗地主洗牌发牌案例**
16. 【应用】能够独立斗地主的准备牌,洗牌,发牌,排序,看牌相关代码
17. **【应用】File类**
18. 【应用】能够独立写出File类的常用构造方法
19. 【理解】能够阐述绝对路径与相对路径的概念
20. 【应用】能够独立测试File类的获取功能相关方法
21. 【应用】能够独立测试File类的创建删除功能相关方法
22. 【应用】能够独立测试File类的判断功能相关方法
23. 【应用】能够独立测试File类的listFiles方法
24. **【应用】递归**
25. 【理解】能够阐述递归的概念
26. 【应用】能够独立写出递归求阶乘案例
27. 【应用】能够独立写出递归打印多级文件夹路径

# 双列集合Map

## Map概述

双列集合是每个元素都有键与值两部分组成的集合，记录的是键值对对应关系。即通过键可以找到值。

常用子类：

最常用的双列集合是Map下的子类HashMap。

Hashtable也是Map集合的一种已被HashMap取代。

Map集合的特点，如是否可重复，是否有序仅作用在键上，如HashMap集合的键不得重复，值可以重复。

### 案例代码一:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.HashMap;  /\*\*  \* **@ClassName**: MapDemo  \* **@Description**: Map集合  \* **@date** 2017年11月20日 上午8:47:48  \* Company www.igeekhome.com  \*  \* Map:双列集合,每个元素分为键与值两部分,是一个键值得对应关系.我们经常使用键找值  \*  \* 最常用的Map集合是HashMap:键是唯一的且无序  \*  \* 用法:  \* put(key, value);  \* get(key);  \*/  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  HashMap<String, String> map = **new** HashMap<String, String>();  //向集合中添加元素  map.put("姓名", "Jack");  map.put("性别", "男");  map.put("年龄", "18");  map.put("分数", "18");  //可以通过键获取值  String value = map.get("姓名");  System.***out***.println(value);  }  } |

## Map基本使用

A：Map(HashMap)的使用：创建对象时加入两个泛型。

Map<k,v>

key - 此映射所维护的键的类型

value - 映射值的类型

B：常用方法：

public V put(K key,V value) //加入元素，则新值会覆盖掉旧值

public V get(Object key) //根据键找值

public Set<K> keySet() //返回所有键的集合

public Collection<V> values() //返回所有值的集合

### 案例代码二:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.HashMap;  /\*\*  \* **@ClassName**: MapDemo  \* **@Description**: Map集合常用方法  \* **@date** 2017年11月20日 上午8:50:58  \* Company www.igeekhome.com  \*  \* Map:双列集合,每个元素分为键与值两部分,是一个键值得对应关系.我们经常使用键找值  \*  \* 最常用的Map集合是HashMap:键是唯一的且无序  \*  \* 用法:  \* 创建对象时,要分别制定键的泛型与值的泛型  \*  \* Map(HashMap)的使用：创建对象时加入两个泛型。  \* Map<k,v>  \* key - 此映射所维护的键的类型  \* value - 映射值的类型  \*  \* 常用方法：  \* public V put(K key,V value) //加入元素，则新值会覆盖掉旧值  \* public V get(Object key) //根据键找值  \* public Set<K> keySet() //返回所有键的集合  \* public Collection<V> values() //返回所有值的集合  \*  \*/  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  HashMap<String, String> map = **new** HashMap<String, String>();    //向集合中添加元素  map.put("及时雨", "宋江");  map.put("玉麒麟", "卢俊义");  map.put("智多星", "高俅");  map.put("智多星", "吴用");    System.***out***.println(map);  //可以通过键获取值  String name = map.get("及时雨");  System.***out***.println(name);    String name2 = map.get("母夜叉");  System.***out***.println(name2);  }  } |

## Map集合遍历方式1

第1种遍历方式是，使用Map集合的keySet()方法

A:思路：

\* 通过keySet()方法获取所有键的集合

\* 遍历键的集合，获取到每一个键

\* 根据键找值

### 案例代码三:

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.Collection;  **import** java.util.HashMap;  **import** java.util.Iterator;  **import** java.util.Set;  /\*\*  \* **@ClassName**: MapDemo  \* **@Description**: Map集合遍历方式  \* **@date** 2017年11月20日 上午8:50:58  \* Company www.igeekhome.com  \*  \* Map:双列集合,每个元素分为键与值两部分,是一个键值得对应关系.我们经常使用键找值  \*  \* 最常用的Map集合是HashMap:键是唯一的且无序  \*  \* 用法:  \* 创建对象时,要分别制定键的泛型与值的泛型  \*  \* Map(HashMap)的使用：创建对象时加入两个泛型。  \* Map<k,v>  \* key - 此映射所维护的键的类型  \* value - 映射值的类型  \*  \* 常用方法：  \* public V put(K key,V value) //加入元素，则新值会覆盖掉旧值  \* public V get(Object key) //根据键找值  \* public Set<K> keySet() //返回所有键的集合  \* public Collection<V> values() //返回所有值的集合  \*  \* Map没有迭代器方法,最常用的遍历方法:先获取所有键的集合,迭代该集合,依次获取每一个键.通过键找值.  \*/  **public** **class** MapDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  HashMap<String, String> map = **new** HashMap<String, String>();    //向集合中添加元素  map.put("及时雨", "宋江");  map.put("玉麒麟", "卢俊义");  map.put("智多星", "高俅");  map.put("智多星", "吴用");    //返回所有键的集合  Set<String> keySet = map.keySet();  System.***out***.println(keySet);    //返回所有值得集合  Collection<String> values = map.values();  System.***out***.println(values);    System.***out***.println("==========================");    //map集合的常用遍历  //迭代所有键的Set集合,依次获取每一个键  Iterator<String> iterator = keySet.iterator();    **while**(iterator.hasNext()) {  String thisKey = iterator.next();  //通过键找值,记住这里使用map集合通过键找值  String thisValue = map.get(thisKey);  System.***out***.println(thisKey+"="+thisValue);  }  }  } |

## Map集合遍历方式2

第2种遍历方式是，使用Map集合的entrySet()方法

A: 思路：

\* 通过entrySet()方法获取所有键值对对象的集合

\* 遍历键值对对象的集合，获取到每一个键值对对象

\* 根据键值对对象找键和值

B: entrySet()方法解释

Set<Map.Entry<K,V>> entrySet() 方法用于返回某个集合所有的键值对对象。

Map.Entry说明Entry是Map的内部接口,将键和值封装成了Entry对象,并存储在Set集合中。可以从一个Entry对象中获取一个键值对的键与值。

C: Entry中的方法如下：

K getKey() 获取键

V getValue() 获取值

### 案例代码四:

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.HashMap;  **import** java.util.Map.Entry;  **import** java.util.Set;  /\*\*  \* **@ClassName**: EntryDemo  \* **@Description**: Map集合遍历方式2  \* **@date** 2017年11月20日 上午8:59:42  \* Company www.igeekhome.com  \*  \* Map集合的第二种遍历方式:  \*  \* Set<Map.Entry<K,V>> entrySet() 方法用于返回某个集合所有的键值对对象。  \* Entry将键值对对应关系封装成了对象。可以从一个Entry对象中中获取一个键值对的键与值。  \*  \* Map.Entry是一个内部接口  \*  \* Entry中的方法如下：  \* K getKey() 获取键  \* V getValue() 获取值  \*/  **public** **class** EntryDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  HashMap<String, String> map = **new** HashMap<String, String>();    //向集合中添加元素  map.put("及时雨", "宋江");  map.put("玉麒麟", "卢俊义");  map.put("智多星", "吴用");    //获取集合中所有的键值对对象的Set集合  Set<Entry<String,String>> entrySet = map.entrySet();    //迭代集合,依次获取每一个键值对对象  **for**(Entry<String,String> thisEntry: entrySet) {  //通过键值对对象获取键  String key = thisEntry.getKey();  //通过键值对对象获取值  String value = thisEntry.getValue();  System.***out***.println(key+":"+value);  }  }  } |

## LinkedHashMap

A:LinkedHashMap:

\* Linked链表结构,保证元素有顺序

\* Hash结构保证元素唯一

\* 以上约束对键起作用

B:LinkedHashMap的特点

\* 底层是链表实现的可以保证怎么存就怎么取

### 案例代码五:

|  |
| --- |
| **package** com.igeek\_05;  **import** java.util.LinkedHashMap;  **import** java.util.Set;  /\*\*  \* **@ClassName**: LinkedHashMapDemo  \* **@Description**: LinkedHashMap的使用  \* **@date** 2017年11月20日 上午9:03:01  \* Company www.igeekhome.com  \*  \* LinkedHashMap:  \* Linked链表结构,保证元素有顺序  \* Hash结构保证元素唯一  \* 以上约束对键起作用  \*/  **public** **class** LinkedHashMapDemo {  **public** **static** **void** main(String[] args) {  //创建集合对象  LinkedHashMap<String, String> map = **new** LinkedHashMap<String, String>();  //向集合中添加元素  map.put("近平", "丽媛");  map.put("路人甲", "路人乙");  map.put("Jack", "Rose");  map.put("Jack", "Lily");    //观察集合元素  System.***out***.println(map);    //遍历集合  //获取所有键的集合  Set<String> keys = map.keySet();    //迭代所有键的集合  **for**(String thisKey: keys) {  //通过键找值  String thisValue = map.get(thisKey);  //打印信息  System.***out***.println(thisKey+":"+thisValue);  }  }  } |

# 集合相关其他

## 可变参数

Collections中有一个方法可以一次加入多个元素

public static <T> boolean addAll(Collection<? super T> c,T... elements)

该方法使用到了可变参数，即定义时并不知道要传入多少个实际参数。此时定义成...的方式，此时可以在调用该方法时，一次传入多个参数。传入的多个数将被自动组织成数组，我们只要操作生成的数组即可。

注：可变参数只能放在最后定义。可变参数方法的参数本质是数组，所以不可以与数组类型参数重载。

### 案例代码六:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.ArrayList;  **import** java.util.Collection;  **import** java.util.Collections;  /\*\*  \* **@ClassName**: ArgsDemo  \* **@Description**: 方法的可变参数  \* **@date** 2017年11月20日 上午9:07:16  \* Company www.igeekhome.com  \*  \* 可变参数:  \* 参数数量可变  \*  \* Collections:  \* public static <T>boolean addAll(Collection c<? super T>, T... elements) 将指定的元素添加到集合  \*  \* 数据类型...代表可变参数.即可以传入任何多个该类型的数据  \* 可变参数不能放在前边,只能放在最后  \* 可变参数的方法与数组的方法不能重载,因为可变参数实质上就是数组  \*/  **public** **class** ArgsDemo {  **public** **static** **void** main(String[] args) {    Collection<String> c = **new** ArrayList<String>();    //向c集合中添加N个元素  Collections.*addAll*(c, "Jack","Rose","Trump");    System.***out***.println(c);  System.***out***.println("=====================================");    **int** sum = *add*(10,20,30,40);  System.***out***.println(sum);  System.***out***.println("=====================================");    **int**[] arr={10,20,30,40};  //可变参数，本质上就是数组  sum = *add*(arr);  System.***out***.println(sum);  }    //用...定义可变参数,a表示了所有传进来的参数,a就是一个数组名称,存储了所有的参数  /\*\*  \* **@Title**: add  \* **@Description**: 求任意个数的和  \* **@param** a  \* **@return**  \*/  **public** **static** **int** add(**int**... a) {    //用来记录所有数的和  **int** sum = 0;  //遍历可变参数a代表的数组  **for**(**int** i=0; i<a.length; i++) {  **int** j = a[i];  System.***out***.println(j);  //每次遇到一个数,就将该数累加到和中  sum += j;  }    //返回和  **return** sum;  }    /\*  //可变参数方法不能与数组方法重载  //可变参数，本质上就是数组  public static void add(int[] x) {    }  \*/  } |

## Collections的shuffle方法

shuffle方法的作用：

打乱集合中元素顺序

### 案例代码七:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  /\*\*  \* **@ClassName**: CollectionsDemo  \* **@Description**: Collections的shuffle方法  \* **@date** 2017年11月20日 上午9:14:15  \* Company www.igeekhome.com  \*  \* Collections:集合工具类,包含了众多集合的方法  \*  \* public static void shuffle(List<?> list) //打乱元素顺序  \*  \* 有顺序(有序):第一个元素是多少,第二个元素是多少,第几个元素对应的是第几,顺序不变.  \* 排序:不管是第几个放的,只要到集合中(以Integer集合为例),就按照一定的顺序重新排列了.  \*/  **public** **class** CollectionsDemo {  **public** **static** **void** main(String[] args) {  //准备集合及元素  List<Integer> list = **new** ArrayList<Integer>();    list.add(2);  list.add(7);  list.add(6);  list.add(10);  list.add(9);    //打乱前的顺序，就是放入元素的先后顺序  System.***out***.println(list);    //打乱集合顺序  Collections.*shuffle*(list);  //打印集合  System.***out***.println(list);  }  } |

## Collections的sort方法

A：sort方法的作用：

对集合中元素排序

B：sort方法签名

public static <T> void sort(List<T> list)

有顺序(有序)：第一个元素是多少,第二个元素是多少,第几个元素对应的是第几,顺序不变.

排序：不管是第几个放的,只要到集合中(以Integer集合为例),就按照一定的顺序重新排列了.

### 案例代码八:

|  |
| --- |
| **package** com.igeek\_03;  /\*\*  \* **@ClassName**: Person  \* **@Description**: Person类  \* **@date** 2017年11月20日 上午9:21:16  \* Company www.igeekhome.com  \*  \*/  **public** **class** Person {  /\*\*  \* **@Fields** name : 姓名  \*/  **private** String name;  /\*\*  \* **@Fields** age : 年龄  \*/  **private** **int** age;  /\*\*  \* **@Title**: Person  \*/  **public** Person() {  **super**();  }  /\*\*  \* **@Title**: Person  \* **@param** name  \* **@param** age  \*/  **public** Person(String name, **int** age) {  **super**();  **this**.name = name;  **this**.age = age;  }  /\*\*  \* **@return** the name  \*/  **public** String getName() {  **return** name;  }  /\*\*  \* **@param** name the name to set  \*/  **public** **void** setName(String name) {  **this**.name = name;  }  /\*\*  \* **@return** the age  \*/  **public** **int** getAge() {  **return** age;  }  /\*\*  \* **@param** age the age to set  \*/  **public** **void** setAge(**int** age) {  **this**.age = age;  }    } |

|  |
| --- |
| **package** com.igeek\_03;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  /\*\*  \* **@ClassName**: CollectionsDemo  \* **@Description**: Collections的sort方法  \* **@date** 2017年11月20日 上午9:14:15  \* Company www.igeekhome.com  \*  \* Collections:集合工具类,包含了众多集合的方法  \*  \* public static <T> void sort(List<T> list) //排序  \* 排序:不管是第几个放的,只要到集合中(以Integer集合为例),就按照一定的顺序重新排列了.  \*  \* public static <T> int binarySearch(List<?>list,T key) //查找元素索引  \* 二分法查找:在一个集合当中,查找一个指定元素的索引是多少,如果不存在该元素,就返回负数索引  \* 二分法查询必须要求集合中的元素排好顺序  \*  \* public static void shuffle(List<?> list) //打乱元素顺序  \*  \* 有顺序(有序):第一个元素是多少,第二个元素是多少,第几个元素对应的是第几,顺序不变.  \*  \*/  **public** **class** CollectionsDemo {  **public** **static** **void** main(String[] args) {  //准备集合及元素  List<Integer>list = **new** ArrayList<Integer>();    list.add(2);  list.add(7);  list.add(6);  list.add(10);  list.add(9);    System.***out***.println(list);  //将集合排好顺序(Integer数字的自然顺序是从小到大排列)  Collections.*sort*(list);  System.***out***.println(list);    //如果放字符串(字符串的顺序是字母排序.)  List<String> list2 = **new** ArrayList<String>();    list2.add("a");  list2.add("d");  list2.add("c");  list2.add("b");    System.***out***.println(list2);  Collections.*sort*(list2);  System.***out***.println(list2);    //如果放Person对象(人这个自定义数据类型的对象没有比较顺序,不知道什么样的人叫大,什么样的人叫小,所以人不能比较.就不能排序)  Person p = **new** Person("Jack",18);  Person p2 = **new** Person("Rose",20);  Person p3 = **new** Person("Adda",24);    List<Person> list3 = **new** ArrayList<Person>();  list3.add(p);  list3.add(p2);  list3.add(p3);    //Collections.sort(list3);  }  } |

## Collections的二分法查找

A：binarySearch方法的作用

查找集合中指定元素的索引

B：binarySearch方法签名

public static <T>int binarySearch(List<?> list,T key)

### 案例代码九:

|  |
| --- |
| **package** com.igeek\_04;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.List;  /\*\*  \* **@ClassName**: CollectionsDemo  \* **@Description**: Collections的binarySearch方法  \* **@date** 2017年11月20日 上午9:14:15  \* Company www.igeekhome.com  \*  \* Collections:集合工具类,包含了众多集合的方法  \*  \* public static <T> void sort(List<T> list) //排序  \* 排序:不管是第几个放的,只要到集合中(以Integer集合为例),就按照一定的顺序重新排列了.  \*  \* public static <T> int binarySearch(List<?>list,T key) //查找元素索引  \* 二分法查找:在一个集合当中,查找一个指定元素的索引是多少,如果不存在该元素,就返回负数索引  \* 二分法查询必须要求集合中的元素排好顺序  \*  \* public static void shuffle(List<?> list) //打乱元素顺序  \*  \* 有顺序(有序):第一个元素是多少,第二个元素是多少,第几个元素对应的是第几,顺序不变.  \*  \*/  **public** **class** CollectionsDemo {  **public** **static** **void** main(String[] args) {    //验证二分法查找元素必须有序  List<Integer> list = **new** ArrayList<Integer>();    list.add(2);  list.add(7);  list.add(6);  list.add(10);  list.add(9);    **int** binarySearch = Collections.*binarySearch*(list, 9);  System.***out***.println(binarySearch);    Collections.*sort*(list);  System.***out***.println(list);    **int** binarySearch2 = Collections.*binarySearch*(list, 9);  System.***out***.println(binarySearch2);  }  } |

## Arrays的toString方法

A：toString方法的作用：

返回字符串内容

B：toString方法签名：

public static String toString(Xxx[] a)

### 案例代码十:

|  |
| --- |
| **package** com.igeek\_05;  **import** java.util.Arrays;  /\*\*  \* **@ClassName**: ArraysDemo  \* **@Description**: Arrays工具类的toString方法  \* **@date** 2017年11月20日 上午9:49:16  \* Company www.igeekhome.com  \*  \* Arrays:  \* 数组工具类,包含了许多数组工具方法  \*  \* public static int binarySearch(Xxx[] a,Xxx key) //查找元素索引  \* public static void sort(Xxx[] a) //排序  \* public static String toString(Xxx[] a) //返回字符串内容  \*/  **public** **class** ArraysDemo {  **public** **static** **void** main(String[] args) {  String[] arr = {"Jack","Rose","Trump"};  //Integer[] arr = {3,1,5,6,2,19};    System.***out***.println(arr);    //调用Arrays的toString方法  System.***out***.println(Arrays.*toString*(arr));  }  } |

## 数组转集合

数组转集合使用的是，Arrays类中的asList方法

方法签名如下：

public static <T> List<T> asList(T... a)

解释：

静态方法，直接类名.方式调用

方法的形参是可变参数类型，可变参数本质也是数组，传入实际数组，将该数组转成集合返回

注意：

数组转成集合之后，该集合不支持添加或者删除操作，否则会抛出UnsupportedOperationException异常

### 案例代码十一:

|  |
| --- |
| **package** com.igeek\_06;  **import** java.util.Arrays;  **import** java.util.List;  /\*\*  \* **@ClassName**: ArraysAsListDemo  \* **@Description**: Arrays类中的asList方法  \* **@date** 2017年11月20日 上午10:06:01  \* Company www.igeekhome.com  \*  \* 数组转集合  \*  \* Arrays:  \* public static <T> List<T> asList(T... a) //数组转集合  \*  \* UnsupportedOperationException:不支持的添加或者删除操作  \*/  **public** **class** ArraysAsListDemo {  **public** **static** **void** main(String[] args) {  List<String> newList = Arrays.*asList*("Jack","Rose","Trump");  System.***out***.println(newList);    //使用asList转成的集合,不能添加或者删除元素,该方法返回的集合长度不能改变!所以,下面代码运行报错.  newList.add("Obama");  System.***out***.println(newList);  }  } |

## 集合转数组

集合ArrayList转数组使用的是，ArrayList中的toArray()方法。

该方法是重载的方法：

public Object[] toArray()

public <T> T[] toArray(T[] a)

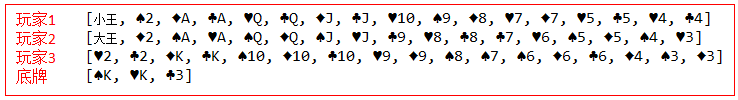
### 案例代码十二:

|  |
| --- |
| **package** com.igeek\_07;  **import** java.util.ArrayList;  **import** java.util.Arrays;  **import** java.util.List;  /\*\*  \* **@ClassName**: ArrayListToArrayDemo  \* **@Description**: ArrayList中的toArray()方法  \* **@date** 2017年11月20日 上午10:15:43  \* Company www.igeekhome.com  \*  \* 集合转数组:  \* 集合ArrayList转为数组的方法为ArrayList的以下两个方法：  \* public Object[] toArray() //集合转数组  \* public <T> T[] toArray(T[] a)  \*/  **public** **class** ArrayListToArrayDemo {  **public** **static** **void** main(String[] args) {  List<Integer>list = **new** ArrayList<Integer>();    list.add(2);  list.add(7);  list.add(6);  list.add(10);  list.add(9);    //调用方法,将集合转为Object类型数组  Object[] objArr = list.toArray();  System.***out***.println(Arrays.*toString*(objArr));    Integer i = (Integer) objArr[0];  //如果使用元素类型Integer的特有方法,则必须先强转成Integer,否则intValue方法不能使用,比较麻烦  System.***out***.println(i.intValue());  System.***out***.println("=======================");    //调用方法,将集合转为带类型的数组  Integer[] intArr = **new** Integer[2];  Integer[] returnArr = list.toArray(intArr);    //如果参数数组足够放下集合中所有元素,就放入参数数组中,返回参数数组  //如果参数数组无法放下集合中所有元素,则参数数组只起到确定类型作用,方法逻辑会自动创建新数组存储集合内容,并返回.  System.***out***.println(Arrays.*toString*(intArr));  System.***out***.println(Arrays.*toString*(returnArr));    //带泛型后,元素类型直接就是集合中的元素类型,避免了强转的问题  Integer integer = returnArr[0];  System.***out***.println(integer.intValue());  }  } |

# 斗地主洗牌发牌案例

## 斗地主排序版案例介绍

按照斗地主的规则，完成洗牌发牌的动作。



具体规则：

1. 组装54张扑克牌

2. 将54张牌顺序打乱

3. 三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌。

4. 查看三人各自手中的牌（按照牌的大小排序）、底牌

手中扑克牌从大到小的摆放顺序：大王,小王,2,A,K,Q,J,10,9,8,7,6,5,4,3

## 案例需求分析

A:准备牌：

完成数字与纸牌的映射关系：

使用双列Map(HashMap)集合，完成一个数字与字符串纸牌的对应关系(相当于一个字典)。

B:洗牌：

通过数字完成洗牌发牌

C:发牌：

将每个人以及底牌设计为ArrayList<Integer>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。

存放的过程中要求数字大小与斗地主规则的大小对应。

将代表不同纸牌的数字分配给不同的玩家与底牌。

D:看牌：

通过Map集合找到对应字符展示。

通过查询纸牌与数字的对应关系，由数字转成纸牌字符串再进行展示。

## 实现代码步骤:

### 案例代码十三:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.util.ArrayList;  **import** java.util.Collections;  **import** java.util.HashMap;  **import** java.util.Map;  /\*\*  \* **@ClassName**: CardDemo  \* **@Description**: 按照斗地主的规则，完成洗牌发牌的动作  \* **@date** 2017年11月20日 上午10:41:34  \* Company www.igeekhome.com  \*  \* ♣♦♠♥ 大☺小☺  \* 具体规则：  \* 使用54张牌打乱顺序  \* 三个玩家参与游戏，三人交替摸牌，每人17张牌，最后三张留作底牌。  \*  \* 逻辑分析:  \*  \* 每张牌由花色数字两部分组成，我们可以使用花色集合与数字集合嵌套迭代完成每张牌的组装。  \* 使用一个Map集合,<<将小的数字与小的牌对应起来>>,做成Map集合的键与值  \* 即:  \* 键:数字  \* 值:花色+数字的字符串(牌)  \*  \* 准备牌：  \* 牌可以设计为一个ArrayList<Integer>,每个数字为一张牌。  \* 牌由Collections类的shuffle方法进行随机排序。  \* 发牌：  \* 将每个人以及底牌设计为ArrayList<Integer>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。  \* 看牌：  \* 将所有集合排序Collections类的sort方法进行排序  \* 将每个集合的数字依次找到对应的纸牌字符串打印出来  \*/  **public** **class** CardDemo {  **public** **static** **void** main(String[] args) {  //确立纸牌与数字的对应关系  Map<Integer, String> map = **new** HashMap<Integer, String>();    //花色集合  ArrayList<String> colors = **new** ArrayList<String>();  colors.add("♣");  colors.add("♦");  colors.add("♠");  colors.add("♥");    //数字集合,加入数字时,直接按照从小到大的顺序加入  ArrayList<String> numbers = **new** ArrayList<String>();  **for**(**int** i = 3; i<= 10; i++) {  numbers.add(i+"");  }  Collections.*addAll*(numbers, "J","Q","K","A","2");    //定义数字,用于记录数字与字符串纸牌的对应关系  **int** cardNumber = 0;    //遍历数字集合,使用每个数字匹配每种花色  **for**(String thisNumber : numbers) {  //使用数字匹配所有花色  **for**(String thisColor : colors) {  String thisCard = thisColor + thisNumber;  //匹配数字与字符串纸牌,完成对应关系  map.put(cardNumber, thisCard);  //每加入一个对应关系,就让数字加1  cardNumber++;  }  }    //加入大小王对应关系  map.put(cardNumber++, "小☺");  map.put(cardNumber, "大☺");  System.***out***.println(map);  //准备牌：  ArrayList<Integer> poker = **new** ArrayList<Integer>();  **for**(**int** i = 0; i<54; i++) {  poker.add(i);  }  //打乱顺序  Collections.*shuffle*(poker);  //发牌：  //将每个人以及底牌设计为ArrayList<Integer>,将最后3张牌直接存放于底牌，剩余牌通过对3取模依次发牌。  ArrayList<Integer> player01 = **new** ArrayList<Integer>();  ArrayList<Integer> player02 = **new** ArrayList<Integer>();  ArrayList<Integer> player03 = **new** ArrayList<Integer>();  ArrayList<Integer> dipai = **new** ArrayList<Integer>();    **for**(**int** index = 0; index <poker.size()-3; index++) {  //通过索引,获取代表牌的数字  Integer integerCard = poker.get(index);    //对3取模判断发给哪个玩家  **if**(index%3==0) {  player01.add(integerCard);  }**else** **if**(index%3==1){  player02.add(integerCard);  }**else**{  player03.add(integerCard);  }  }    //最后3张是底牌  **for**(**int** index=poker.size()-3; index<poker.size(); index++){  //获取当前这张牌  Integer integerCard = poker.get(index);  dipai.add(integerCard);  }    //将所有集合排序Collections类的sort方法进行排序  Collections.*sort*(player01);  Collections.*sort*(player02);  Collections.*sort*(player03);  Collections.*sort*(dipai);  //看牌  **for**(**int** i=player01.size()-1;i>=0;i--) {  Integer integer = player01.get(i);  String realCard = map.get(integer);  System.***out***.print(realCard+" ");  }  System.***out***.println();    **for**(**int** i=player02.size()-1;i>=0;i--) {  Integer integer = player02.get(i);  String realCard = map.get(integer);  System.***out***.print(realCard+" ");  }  System.***out***.println();    **for**(**int** i=player03.size()-1;i>=0;i--) {  Integer integer = player03.get(i);  String realCard = map.get(integer);  System.***out***.print(realCard+" ");  }  System.***out***.println();    **for**(Integer integer : dipai) {  String realCard = map.get(integer);  System.***out***.print(realCard+" ");  }  }  } |

# File类

## File类的概念

打开API，搜索File类。

阅读其描述：File文件和目录路径名的抽象表示形式。

即，Java中把文件或者目录（文件夹）都封装成File对象。

也就是说如果我们要去操作硬盘上的文件，或者文件夹只要找到File这个类即可。

那么我们就要研究研究File这个类中都有哪些功能可以操作文件或者文件夹呢？

### 案例代码十四:

|  |
| --- |
| **package** com.igeek\_01;  **import** java.io.File;  /\*\*  \* **@ClassName**: FileDemo  \* **@Description**: File文件类  \* **@date** 2017年11月20日 上午11:07:47  \* Company www.igeekhome.com  \*  \* File:  \* 文件可以持久化地存储数据  \* File的一个对象就代表一个文件或文件夹(自己简单地定义,方便记忆的说法)  \* 文档上说明File类代表文件或文件夹路径,但是我们可以通过路径找到对应的文件或文件夹.  \* 可以认为File类就代表文件或文件夹(通过路径找到)  \*  \* 构造方法:  \* public File(String pathname) 通过路径找到对应的文件或文件夹  \*/  **public** **class** FileDemo {  **public** **static** **void** main(String[] args) {  //查找文件  File file = **new** File("D:\\other\\test.txt");    //查找文件夹  File dir = **new** File("D:\\other");  }  } |

## 相对路径与绝对路径

绝对路径：从盘符开始到文件的路径

相对路径：从某个参照路径开始到指定文件所经过的路径

### 案例代码十五:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.File;  /\*\*  \* **@ClassName**: AbsoluteRelativeDemo  \* **@Description**: 相对路径与绝对路径  \* **@date** 2017年11月20日 下午1:38:10  \* Company www.igeekhome.com  \*  \* 路径分为绝对路径与相对路径:  \*  \* 绝对路径:从盘符开始到文件的全路径  \* 相对路径:从某个参照路径开始到指定文件所经过的路径  \* 在Eclipse中,所有的文件如果想使用相对路径,则其参照路径均为项目根目录  \* 如test.txt相对于javaadvanced1\_d05\_c04项目根目录经过了a/b/test.txt,则a/b/test.txt就是该文件的相对路径  \*  \* 在java当中对于路径的分隔符可以使用/或者\\,是通用的  \*/  **public** **class** AbsoluteRelativeDemo {  **public** **static** **void** main(String[] args) {  //通过绝对路径在Eclipse中的某个项目当中创建一个文件对象  //"D:/javaadvanced1\_workspace/javaadvanced1\_d05\_c04/a/b/test.txt"  File file = **new** File("D:\\javaadvanced1\_workspace\\javaadvanced1\_d05\_c04\\a\\b\\test.txt");    //通过相对路径在Eclipse中的某个项目当中创建一个文件对象  //"a/b/test.txt"  File file2 = **new** File("a\\b\\test.txt");    //获取文件大小,判断文件是否为test.txt  System.***out***.println(file.length());  System.***out***.println(file2.length());  }  } |

## File类中的方法

### File类中的构造方法

public File(String parent, String child)：parent指的是父级目录，child指的是子级目录

public File(File parent, String child) ：parent为File类型，为了使用File类中的方法

#### 案例代码十六:

|  |
| --- |
| **package** com.igeek\_03\_01;  **import** java.io.File;  /\*\*  \* **@ClassName**: FileMethodDemo  \* **@Description**: File类中的构造方法  \* **@date** 2017年11月20日 下午1:51:03  \* Company www.igeekhome.com  \*  \* File类的方法:  \* 构造方法:  \* 1:直接给于路径(绝对路径,相对路径)  \* 2:路径分为父亲(该文件或文件夹所在的文件夹)与孩子(该文件或文件夹自己的名字)  \*  \*/  **public** **class** FileMethodDemo {  **public** **static** **void** main(String[] args) {    //通过父亲与孩子的构造方法创建test.txt对应的文件对象  File file = **new** File("a/b", "test.txt");  System.***out***.println(file.length());    //通过父亲与孩子的构造方法创建a文件夹下的b文件夹对应的文件对象  File dir = **new** File("a", "b");  }  } |

### File类的获取方法

public String getAbsolutePath() 获取绝对路径

public String getPath() 获取路径(用什么方式创建的对象,就返回什么方式的路径(绝对路径/相对路径))

public String getName() 获取文件/文件夹的名称

public String getParent() 返回所在文件夹路径(根据创建对象时是否为绝对路径/相对路径)

#### 案例代码十七:

|  |
| --- |
| **package** com.igeek\_03\_02;  **import** java.io.File;  /\*\*  \* **@ClassName**: FileMethodDemo  \* **@Description**:File类的获取方法  \* **@date** 2017年11月20日 下午2:00:42  \* Company www.igeekhome.com  \*  \* File类常见方法  \* 1:获取文件/文件夹信息方法  \* public String getAbsolutePath() 获取绝对路径  \* public String getPath() 获取路径(用什么方式创建的对象,就返回什么方式的路径(绝对路径/相对路径))  \* public String getName() 获取文件/文件夹的名称  \* public String getParent() 返回所在文件夹路径(根据创建对象时是否为绝对路径/相对路径)  \*/  **public** **class** FileMethodDemo {  **public** **static** **void** main(String[] args) {  //创建文件对象  File file = **new** File("a/b/test.txt");  File file2 = **new** File("D:/javaadvanced1\_workspace/javaadvanced1\_d05\_c04/a/b/test.txt");  File dir = **new** File("a/b");    //获取绝对路径  System.***out***.println(file.getAbsolutePath());  System.***out***.println(file2.getAbsolutePath());  System.***out***.println("===========================");    //获取路径  System.***out***.println(file.getPath());  System.***out***.println(file2.getPath());  System.***out***.println("===========================");    //获取文件/文件夹的名称  System.***out***.println(file.getName());  System.***out***.println(dir.getName());  System.***out***.println("===========================");    //返回所在文件夹路径  System.***out***.println(file.getParent());  System.***out***.println(file2.getParent());  System.***out***.println(dir.getParent());  }  } |

### File类的创建与删除方法

public boolean delete() 删除文件

public boolean createNewFile() throws IOException 创建文件

在创建文件时,如果文件所在的文件夹不存在,则报错系统找不到指定的路径.创建文件时,必须确保文件夹已经存在

public boolean mkdir() 使用mkdir方法创建文件夹时,必须保证其所在文件夹已经存在,否则创建失败(不会报错)

public boolean mkdirs() 一次性创建多级目录

public boolean delete() 删除文件夹不能为非空(有东西),否则删除失败

#### 案例代码十八:

|  |
| --- |
| **package** com.igeek\_03\_03;  **import** java.io.File;  **import** java.io.IOException;  /\*\*  \* **@ClassName**: FileMethodDemo  \* **@Description**:File类的创建与删除方法  \* **@date** 2017年11月20日 下午2:08:30  \* Company www.igeekhome.com  \*  \* 2:可以创建/删除该文件(Java删除时，不会使用windows的回收站)  \* public boolean delete() 删除文件  \* public boolean createNewFile() throws IOException 创建文件  \* 在创建文件时,如果文件所在的文件夹不存在,则报错系统找不到指定的路径.创建文件时,必须确保文件夹已经存在  \*  \* public boolean mkdir() 使用mkdir方法创建文件夹时,必须保证其所在文件夹已经存在,否则创建失败(不会报错)  \* public boolean mkdirs() 一次性创建多级目录  \*  \* public boolean delete() 删除文件夹不能为非空(有东西),否则删除失败  \*/  **public** **class** FileMethodDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  File file = **new** File("a/b/test.txt");  //删除文件  System.***out***.println(file.getPath()+"的删除结果:"+file.delete());    //创建文件  System.***out***.println(file.getPath()+"的创建结果:"+file.createNewFile());    //在创建文件时,如果文件所在的文件夹不存在,则报错系统找不到指定的路径.创建文件时,必须确保文件夹已经存在  //File file2 = new File("c/d/test.txt");  //System.out.println(file2.createNewFile());  System.***out***.println("==========================");    //创建文件夹  File dir = **new** File("c/d");    //必须保证其所在文件夹已经存在,否则创建失败(不会报错)  System.***out***.println(dir.mkdir());  //一次性创建多级目录  System.***out***.println(dir.mkdirs());    File dir2 = **new** File("e1/e2");  //必须保证其所在文件夹已经存在,否则创建失败(不会报错)  System.***out***.println(dir2.mkdir());  System.***out***.println("==========================");    //删除文件夹  System.***out***.println(dir2.delete());    //删除文件夹不能为非空(有东西),否则删除失败  File dir3 = **new** File("a/b");  System.***out***.println(dir3.delete());  }  } |

### File类的判断方法

public boolean exists() 文件或者文件夹是否存在

public boolean isDirectory() 判断文件对象是否为文件夹

public boolean isFile() 判断文件对象是否为文件

#### 案例代码十九:

|  |
| --- |
| **package** com.igeek\_03\_04;  **import** java.io.File;  /\*\*  \* **@ClassName**: FileMethodDemo  \* **@Description**: File类的判断方法  \* **@date** 2017年11月20日 下午2:18:30  \* Company www.igeekhome.com  \*  \* 3:判断该File对象是否存在或者判断该File对象代表一个文件还是代表一个文件夹  \* public boolean exists() 文件或者文件夹是否存在  \* public boolean isDirectory() 判断文件对象是否为文件夹  \* public boolean isFile() 判断文件对象是否为文件  \*/  **public** **class** FileMethodDemo {  **public** **static** **void** main(String[] args) {  File file = **new** File("a/b/test.txt");  File file2 = **new** File("a/b/c.txt");  File dir = **new** File("c/d");  File dir2 = **new** File("e1/e2");    //文件或者文件夹是否存在  System.***out***.println(file.exists());  System.***out***.println(file2.exists());  System.***out***.println(dir.exists());  System.***out***.println(dir2.exists());  System.***out***.println("================================");    // 判断文件对象是否为文件夹/文件  System.***out***.println(file.isDirectory());  System.***out***.println(file.isFile());  System.***out***.println(dir.isDirectory());  System.***out***.println(dir.isFile());  System.***out***.println("=================================");    //如果是不存在的文件或者文件夹,则判断是否为文件或文件夹时一律报false  System.***out***.println(file2.isFile());  System.***out***.println(dir2.isDirectory());  }  } |

### File类的listFiles方法

public File[] listFiles() 获取调用方法文件夹下的所有file对象(文件或文件夹)

#### 案例代码二十:

|  |
| --- |
| **package** com.igeek\_03\_05;  **import** java.io.File;  /\*\*  \* **@ClassName**: FileListFilesDemo  \* **@Description**: File类的listFiles方法  \* **@date** 2017年11月20日 下午2:24:32  \* Company www.igeekhome.com  \*  \* public File[] listFiles() 获取调用方法文件夹下的所有file对象(文件或文件夹)  \*/  **public** **class** FileListFilesDemo {  **public** **static** **void** main(String[] args) {  //创建文件对象  File dir = **new** File("./");    //获取该文件夹下所有文件对象  File[] filesDirs = dir.listFiles();    **for** (File fileDir : filesDirs) {  System.***out***.println(fileDir);  }  }  } |

# 递归

## 递归概念

递归，指在当前方法内调用自己的这种现象

public void method(){

System.out.println(“递归的演示”);

//在当前方法内调用自己

method();

}

递归分为两种，直接递归和间接递归。

直接递归称为方法自身调用自己。

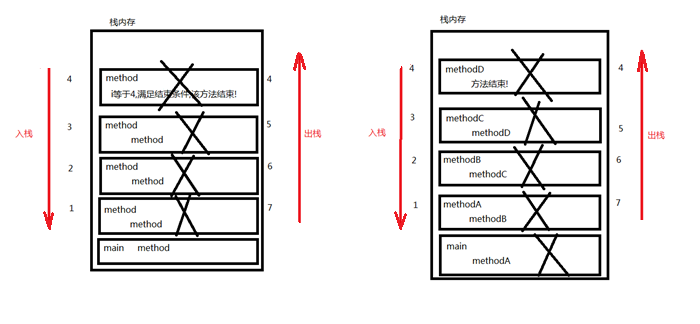
间接递归可以A方法调用B方法，B方法调用C方法，C方法调用A方法。

### 案例代码二十一:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: RecursionDemo  \* **@Description**: 方法递归基础  \* **@date** 2017年11月20日 下午2:36:44  \* Company www.igeekhome.com  \*  \* 不同的方法之间可以相互调用,即A方法可以将B方法调用,内存当中就多了一个B方法,B方法结束,A继续执行  \*  \*/  **public** **class** RecursionDemo {  /\*\*  \* **@Fields** i : 成员位置定义变量,供不同的方法访问  \*/  **static** **int** *i* = 1;  **public** **static** **void** main(String[] args) {  *methodA*();  }    /\*\*  \* **@Title**: methodA  \* **@Description**: 第一个方法：打印i,i++,再调用其他方法,i++,打印  \*/  **public** **static** **void** methodA() {  System.***out***.println("方法A:"+*i*);  *i*++;  *methodB*();  *i*++;  System.***out***.println("方法A:"+*i*);  }    /\*\*  \* **@Title**: methodB  \* **@Description**: 第二个方法  \*/  **public** **static** **void** methodB() {  System.***out***.println("方法B:"+*i*);  *i*++;  *methodC*();  *i*++;  System.***out***.println("方法B:"+*i*);  }  /\*\*  \* **@Title**: methodC  \* **@Description**: 第三个方法  \*/  **private** **static** **void** methodC() {  System.***out***.println("方法C:"+*i*);  *i*++;  *methodD*();  *i*++;  System.***out***.println("方法C:"+*i*);  }  /\*\*  \* **@Title**: methodD  \* **@Description**: 第四个方法  \*/  **private** **static** **void** methodD() {  System.***out***.println("方法D:"+*i*);  }  } |

### 案例代码二十二:

|  |
| --- |
| **package** com.igeek\_01;  /\*\*  \* **@ClassName**: RecursionDemo2  \* **@Description**: 方法递归  \* **@date** 2017年11月20日 下午2:41:52  \* Company www.igeekhome.com  \*  \* 递归算法: 方法自身调用方法自身,必须有方法出口(可以结束方法的条件),递归次数不宜过多,会有stackoverflow(栈内存溢出)  \*/  **public** **class** RecursionDemo2 {    /\*\*  \* **@Fields** i : 成员位置定义变量,供不同的方法访问  \*/  **static** **int** *i* = 1;    **public** **static** **void** main(String[] args) {  *method*();  }    /\*\*  \* **@Title**: method  \* **@Description**: 定义了一个自己调用自己的递归方法  \*/  **public** **static** **void** method() {  System.***out***.println("方法开始:"+*i*);    //如果i已经加到4了,就将method方法结束  **if**(*i*==4) {  **return**;  }    *i*++;  *method*();  *i*++;  System.***out***.println("方法结束:"+*i*);  }  } |



## 递归案例

### 递归求阶乘

求N的阶乘,以5的阶乘为例

5\*4\*3\*2\*1

5! = 5 \* 4!;

4! = 4 \* 3!;

3! = 3 \* 2!;

2!= 2 \* 1!;

1! = 1;

#### 案例代码二十三:

|  |
| --- |
| **package** com.igeek\_02;  /\*\*  \* **@ClassName**: RecursionTest  \* **@Description**: 递归求阶乘  \* **@date** 2017年11月20日 下午3:24:15  \* Company www.igeekhome.com  \*  \* 递归可以将复杂问题简单化  \*  \* 求N的阶乘,以5的阶乘为例  \* 5\*4\*3\*2\*1  \*  \* 5! = 5 \* 4!;  \* 4! = 4 \* 3!;  \* 3! = 3 \* 2!;  \* 2!= 2 \* 1!;  \* 1! = 1; 单独唯一的逻辑  \*  \* 递归逻辑:  \* 当前这个数 \* (比当前这个数-1)的阶乘  \* 递归出口:  \* 如果求到了1的阶乘,就直接返回1,不再求阶乘了  \*/  **public** **class** RecursionTest {  **public** **static** **void** main(String[] args) {  **int** result=*method*(5);  System.***out***.println("result:"+result);  }    /\*\*  \* **@Title**: method  \* **@Description**: 求某个数的阶乘  \* **@param** n  \* **@return**  \*/  **public** **static** **int** method(**int** n) {    //方法出口,如果是求1的阶乘,则直接返回1  **if**(n == 1) {  **return** 1;  }    **return** n \* *method*(n-1);  }  } |

### 递归打印多级文件夹路径

打印一个文件夹内所有的文件路径

#### 案例代码二十四:

|  |
| --- |
| **package** com.igeek\_02;  **import** java.io.File;  /\*\*  \* **@ClassName**: RecursionTest2  \* **@Description**: 递归打印多级文件夹路径  \* **@date** 2017年11月20日 下午3:28:24  \* Company www.igeekhome.com  \*  \* 打印一个文件夹内所有的文件夹和文件路径  \*  \* 1:定义方法,接受文件夹,打印该文件夹中所有的文件路径  \*  \* 2:获取该文件夹中的所有文件对象  \* 3:判断该文件对象是文件还是文件夹  \* 如果是文件,打印路径  \* 如果是文件夹,调用方法,打印该文件夹中所有的文件路径  \*/  **public** **class** RecursionTest2 {  **public** **static** **void** main(String[] args) {  File dir = **new** File("./");  *method*(dir);  }    /\*\*  \* **@Title**: method  \* **@Description**: 定义递归打印方法,打印该文件夹中所有的文件路径  \* **@param** dir  \*/  **public** **static** **void** method(File dir) {    //获取该文件夹中的所有文件对象  File[] filesDirs = dir.listFiles();  //依次获取所有的文件对象  **for** (File file : filesDirs) {  //判断该文件对象是文件还是文件夹  **if**(file.isFile()) {  //如果是文件,打印路径  System.***out***.println(file.getAbsolutePath());  }**else** {  //System.out.println(file.getAbsolutePath());  //如果是文件夹,调用方法,打印该文件夹中所有的文件路径  *method*(file);  }  }  }  } |

重点和总结

1、双列集合Map的使用及遍历方式

2、集合工具类的相关方法

3、数组和集合之间相互转换

4、使用Map类实现斗地主洗牌、发牌及显示的案例

5、File类的基本方法及使用

6、相对路径和绝对路径

7、方法递归算法的概率及案例