



EE基础与应用

翁秀木

第三回 Collections框架

- Java Collections框架简介
- Java Collections框架的使用

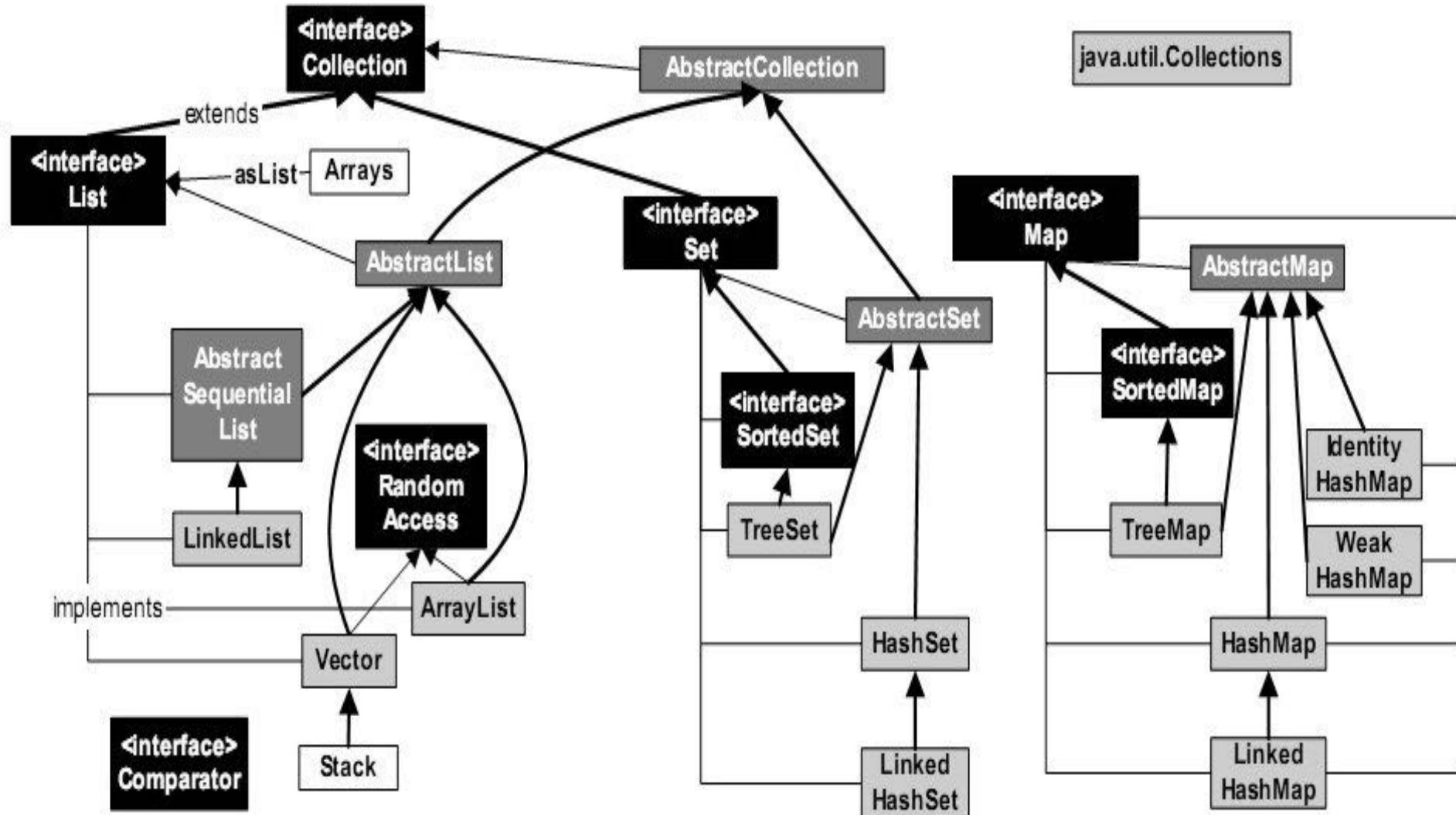
学习目标

- 掌握Java Collections框架的基本结构。
- 掌握Collections框架的基本类的使用：List（ArrayList、LinkedList）、Set（HashSet、TreeSet）、Map（HashMap、TreeMap）、Iterator和ListIterator、Collections和Arrays。

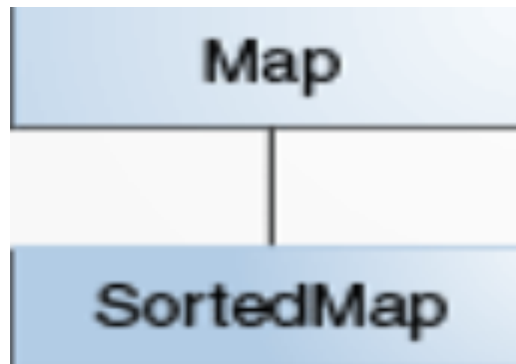
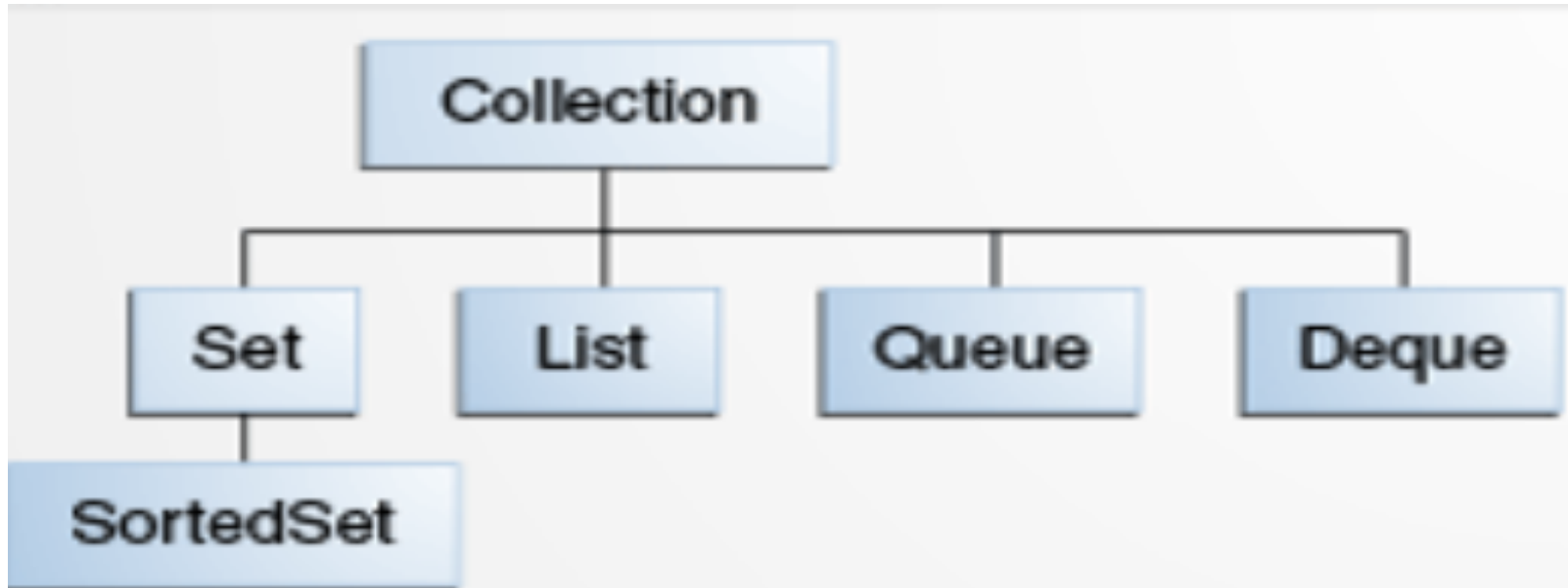
Java Collections (集合) 概念

集合是什么呢？很难给集合下一个精确的定义，通常情况下，把具有相同性质的一类东西，汇聚成一个整体，就可以称为集合。比如某个学校的全体班级、某个公司的全体员工等都可以称为集合。

Java collections框架 (1.4)



Java collections框架 (7.0)



Collection接口

Collection是最基本的集合接口，一个Collection代表一组Object，即Collection的元素（Elements）。一些Collection允许相同的元素而另一些不行。一些能排序而另一些不行。

所有实现Collection接口的类通常有两个标准的构造函数：**无参数的构造函数**用于创建一个空的Collection，**有一个Collection参数的构造函数**用于创建一个新的Collection，这个新的Collection与传入的Collection有相同的元素。后一个构造函数允许用户复制一个Collection。

Set接口

- Set是一种不包含重复的元素的Collection，即任意的两个元素e1和e2都有`e1.equals(e2)==false`，Set最多有一个null元素。

很明显，Set的构造函数有一个约束条件，传入的Collection参数不能包含重复的元素。

请注意：必须小心操作可变对象（Mutable Object）。如果一个Set中的可变元素改变了自身状态导致`Object.equals(Object)=true`将导致一些问题。

- 为不包含相同值的元素，放入Set集合中的对象必须重写`equals ()`方法和`hashCode ()`方法。

List接口

- List是有序的Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在List中的位置，类似于数组下标）来访问List中的元素，这类似于Java的数组。

和下面要提到的Set不同，List允许有相同的元素。

除了具有Collection接口必备的iterator()方法外，List还提供一个listIterator()方法，返回一个ListIterator接口，和标准的Iterator接口相比，ListIterator多了一些add()之类的方法，允许添加，删除，设定元素，还能向前或向后遍历。

实现List接口的常用类有LinkedList，ArrayList，Vector和Stack。

Map接口

- 请注意，Map没有继承Collection接口，Map提供key到value的映射。一个Map中不能包含相同的key，每个key只能映射一个value。Map接口提供3种集合的视图，Map的内容可以被当作一组key集合，一组value集合，或者一组key-value映射。
- 放入Map中的自定义类的对象，需要重写equals()和hashCode () 方法。

ArrayList(1)

接口List次序是List最重要的特点；它确保维护元素特定的顺序。List从Collection接口继承过来并添加了一些抽象方法，例如添加了插入与移除元素的抽象函数。

ArrayList类实现List接口。我们可以理解ArrayList是一个动态的数组。既然它是一个数组，所以它可以按照下标对其中元素进行操作。动态是指ArrayList能够自动分配或释放空间。它允许对元素进行快速随机访问，但是向List中间（非尾部）插入与移除元素的速度很慢。但当元素的增加或移除发生在List中央位置（非尾部）时，效率很差。

ArrayList(2)

```
public static void main(String[] argv) {  
    List list=new ArrayList();  
    Student s1=new Student("1001","zhou",67);  
    Student s2=new Student("1002","lou",87);  
    Student s3=new Student("1003","zhang",87);  
    Student s4=new Student("1004","zhao",76);  
    list.add(s1);list.add(s2);list.add(s3);list.add(s4);  
    for(int i=0;i<list.size();i++){  
        Student s=(Student)list.get(i);  
        System.out.println(s.getSno()+" "+s.getSname()+" "+s.getScore());  
    }  
}
```

LinkedList的使用(1)

- LinkedList: 与ArrayList相反, 适合用来进行**非尾部的**增加和移除元素, 但随机访问的速度较慢。此外, 可以通过LinkedList来实现栈stack与队列queue, 也可以用ArrayList来实现栈stack
-
- LinkedList中的addFirst()、addLast()、getFirst()、getLast()、removeFirst()、removeLast()等函数从前部、末尾或中间位置插入或删除元素。

LinkedList的使用(2)

```
LinkedList list=new LinkedList();  
list.add("a");//在尾部追加  
list.addFirst("b");//在头部增加元素  
list.addLast("c");//在尾部追加  
list.add(1, "d");//在第二个位置插入元素  
System.out.print(list.getFirst()+" ");//读取头部元素  
System.out.print(list.getLast()+" ");//读取尾部元素  
System.out.print(list.get(1)+" ");//读第二个位置元素  
list.removeFirst();//删除头部元素  
list.removeLast();//删除尾部元素  
  
System.out.print(list.getFirst()+" ");//读取头部元素  
System.out.print(list.getLast()+" ");//读取尾部元素
```

Java中的顺序表结构

- **ArrayList**
- **array**

Java中的顺序表结构 - ArrayList

Object[] elementData;

int size;

使用C结构体描述顺序表

```
#define ListSize 100
```

```
typedef int DataType;
```

```
typedef struct
```

```
{
```

```
    DataType data[ListSize];
```

```
    int length;
```

```
} SeqList;
```


Iterator

Outline

- hasNext() : boolean
- next() : Object
- remove() : void

```
void test() {
    List aList = new ArrayList(3);
    aList.add("3");
    aList.add("6");
    aList.add("9");
    testIterator(aList);
}
```

```
void testIterator(ArrayList aList) {
    Iterator i = aList.iterator();
    while( i.hasNext()) {
        System.out.println(i.next());
    }
}
```

Iterator

```
List aList = new ArrayList(3);  
aList.add("3");  
aList.add("6");  
aList.add("9");  
Iterator i = aList.iterator();  
i.next();  
i.next();  
i.remove();
```

ListIterator

Outline

- add(Object) : void
- ▲ hasNext() : boolean
- hasPrevious() : boolean
- ▲ next() : Object
- nextIndex() : int
- previous() : Object
- previousIndex() : int
- ▲ remove() : void
- set(Object) : void

Iterator

Outline

- hasNext() : boolean
- next() : Object
- remove() : void

public interface ListIterator extends
Iterator

ListIterator

```
void test() {  
    List aList = new ArrayList(3);  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator();  
    i.next();  
    i.set("5");  
}
```

ListIterator

```
void test() {  
    List aList = new ArrayList(3);  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator();  
    i.next();  
    i.remove();  
}
```

ListIterator

```
void test() {  
    List aList = new ArrayList(3);  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(aList.size());  
    i.previous();  
}
```

ListIterator

```
void test() {  
    List aList = new ArrayList(3);  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(1);  
    i.add("8");  
}
```

ListIterator

```
void test() {  
    List aList = new ArrayList(3);  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(2);  
    i.nextIndex();  
    i.previousIndex();  
}
```


Java array vs. ArrayList

- 若预先知道处理的是一组**元素个数固定**（不会增删）的线性表，用**array**，“**根据index查询/更新元素**”操作速度最快。
- 若线性表**元素的个数不固定**，且增删操作在表尾，用ArrayList，可自动扩容，且“**根据index查询/更新元素**”操作速度也快，但通常不如array快。

Java中的链表结构

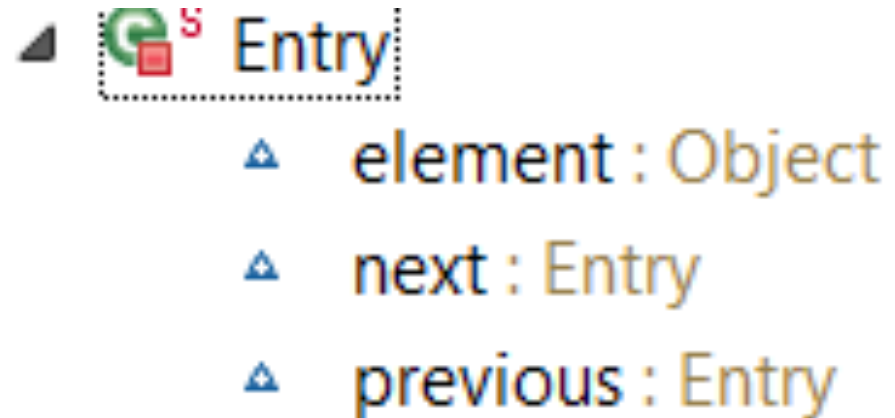
- **LinkedList**

Java中的链表结构 - LinkedList

带头结点的双向循环链表

Entry header;

int size;



双向链表的C描述

```
typedef char DataType
```

```
typedef struct dListNode {
```

```
    DataType data; // 结点的数据域
```

```
    struct dListNode *prior, *next; // 结点的指针域
```

```
}DListNode; // 结构体类型标识符
```

```
typedef DListNode *DLinkedList; // 定义指向结构体的指针类型
```

```
DListNode *p; // 定义指向某结点指针
```

```
DLinkedList head; // 定义头指针
```

ListIterator

```
void test() {  
    List aList = new LinkedList();  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator();  
    i.next();  
    i.set("5");  
}
```

ListIterator

```
void test() {  
    List aList = new LinkedList;  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(aList.size());  
    i.previous();  
}
```

ListIterator

```
void test() {  
    List aList = new LinkedList();  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(1);  
    i.add("8");  
}
```

ListIterator

```
void test() {  
    List aList = new LinkedList;  
    aList.add("3");  
    aList.add("6");  
    aList.add("9");  
    testListIterator(aList);  
}  
  
void testListIterator(List aList) {  
    ListIterator i = aList.listIterator(2);  
    i.nextIndex();  
    i.previousIndex();  
}
```

Java LinkedList vs. ArrayList vs. array

- 若线性表增删很多，且增删不在表尾，且是查询定位后的多次增删，用LinkedList。
- 若有很多“根据index查询/更新元素”的操作，且增删不多或者增删操作在表尾，用ArrayList。
- 若有很多“根据index查询/更新元素”的操作，且无增删（即元素个数固定），用array。
- 若有很多“根据index查询/更新元素”的操作，且无增删（即元素个数固定），但必须用ArrayList（如其中某些方法），则创建时用new ArrayList(n)，n为固定的元素个数。

一、哈希表 (Hash table)

1. 哈希表、哈希函数与哈希码

哈希函数 (Hash function、散列函数) 的自变量为元素的关键字 (key)，函数值称为**哈希码** (Hash code、Hash value或Hash)，根据哈希码即可得出该key对应的元素的地址。

查找时，由函数H对给定值关键字 k_x 计算得出地址，将 k_x 与地址单元中元素关键字进行比较，确定查找是否成功，这种查找方法称为**哈希查找方法**。

按这个算法构造而成的元素的存储结构称为**哈希表** (Hash table或Hash map)，该表中的元素是**键-值对** (key/value pair)，且表中元素之间**不一定遵循逻辑结构中的先后关系**，即

【示例】 设有11个记录的关键字的值分别是：18，27，1，20，22，6，10，13，41，15，25。选取关键字与元素位置间的函数为： $H(\text{key}) = \text{key} \bmod 11$ ，则这11个记录存放在数组a中的下标分别为：7，5，1，9，0，6，10，2，8，4，3。于是得到哈希表如图7-3

哈希地址	0	1	2	3	4	5	6	7	8	9	10
关键字	22	1	13	25	15	27	6	18	41	20	10

图7-3 哈希表

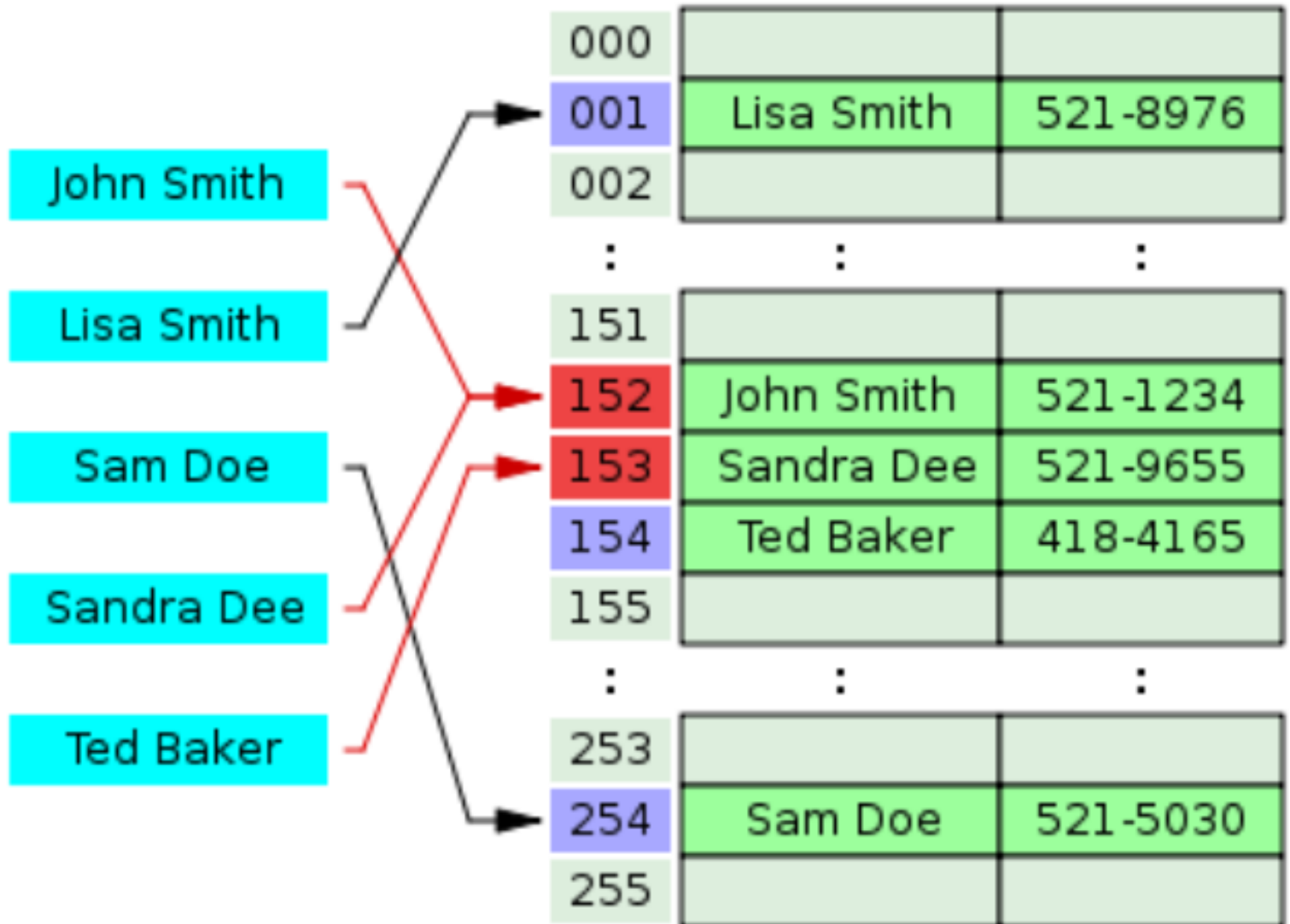
2. 冲突和同义词

对于某个哈希函数 H 和两个关键字 k_1, k_2 ，如果 $k_1 \neq k_2$ ，而 $H(k_1) = H(k_2)$ ，即经过哈希函数变换后，将不同的关键字映射到同一个哈希地址上，这种现象称为冲突， k_1 和 k_2 称为同义词。

【示例】 设哈希函数为： $H(\text{key}) = \text{key} \bmod 11$ ，两个记录的关键字的值分别为，2和13，则 $H(2) = H(13) = 2$ ，即以2和13为关键字的记录发生了冲突，2和13是同义词。

keys

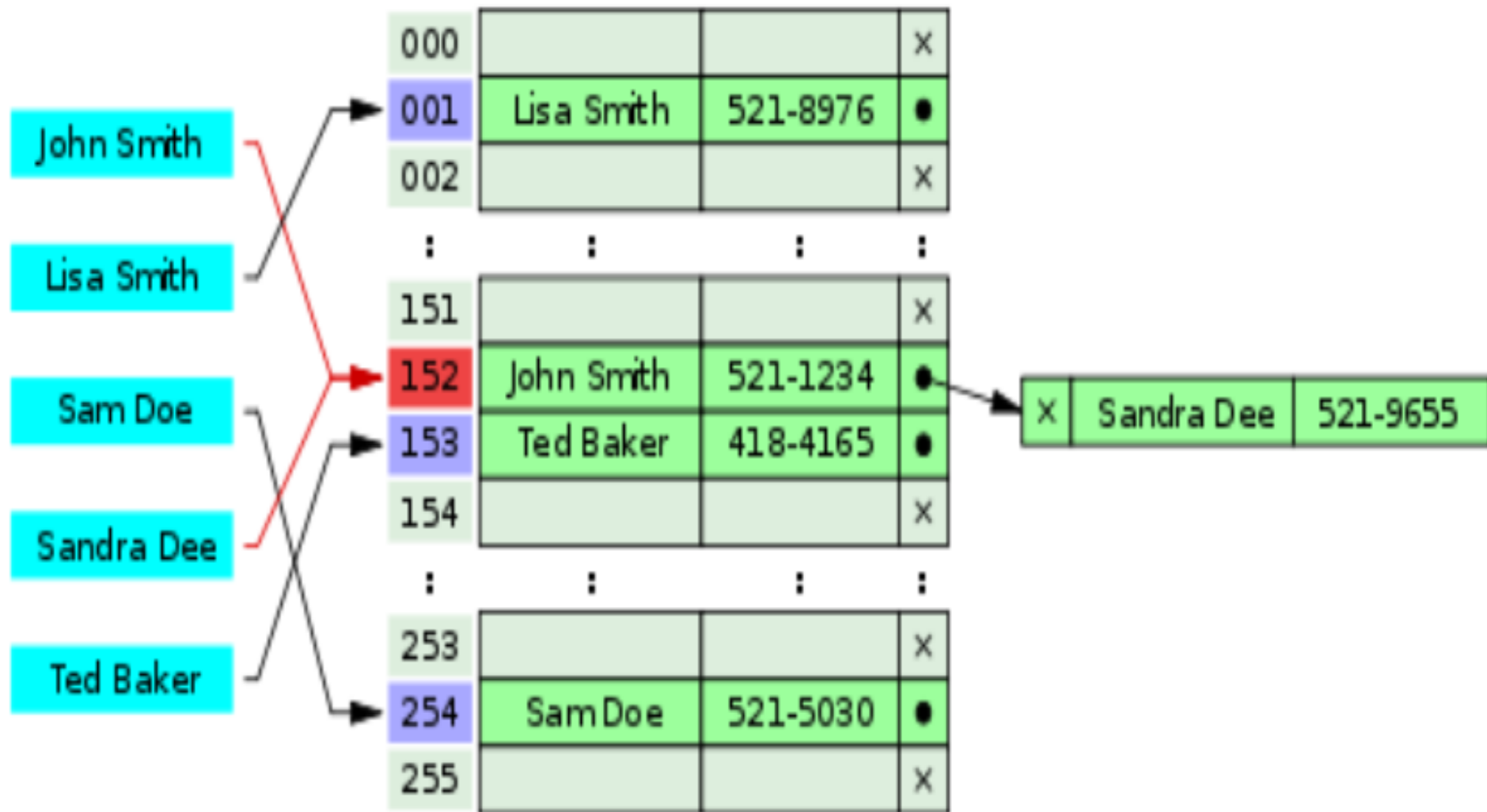
buckets



keys

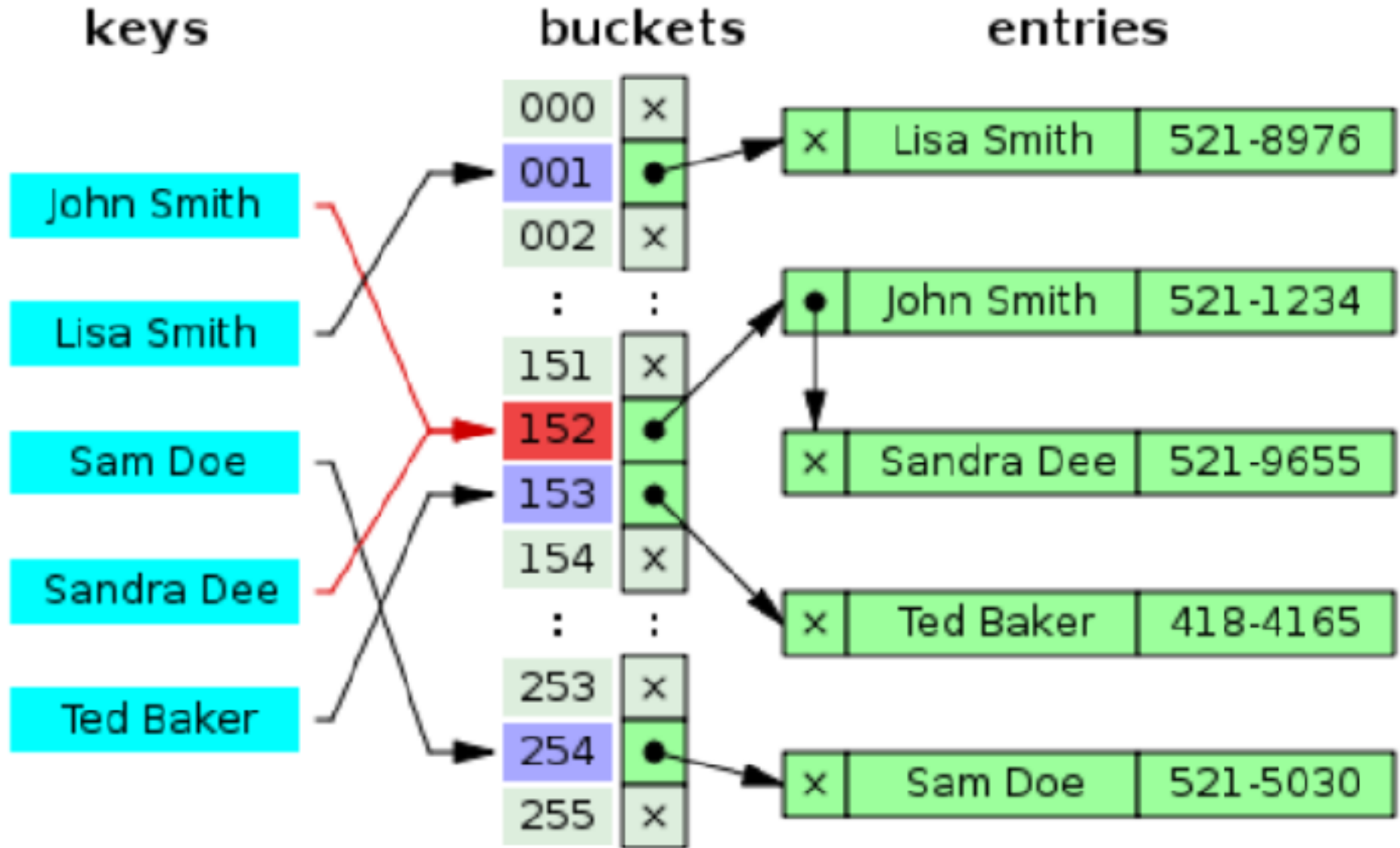
buckets

overflow
entries



Hash collision by separate chaining with head records in the bucket array.

java的HashMap相似，不同在于，当冲突发生时，这图是尾插法，java的HashMap是头插法



HashMap使用(1)

HashMap是Map接口下的实现类, 它每个元素由关键字Key与值Value构成, 根据元素Key以及相应的散列算法计算元素存储地址。

1、构造HashMap以及向集合中添加元素。

```
HashMap map=new HashMap();
```

```
map.put(key, value);
```

```
map.put("1001", "zhou");//向map中添加元素, 第一个参数为key,第二个参数为value
```

```
map.put("1002", "zhang");
```

```
map.put("1003", "zhou");
```

HashMap使用(2)

2、在Map接口中可以根据关键字查找对应元素值

String s=map.get("1002").toString();//"1002"为key,该函数返回关键字对应的值

HashMap使用(3)

3、如何遍历集合中所有元素。

有两种思路：第1种是读出集合中所有的关键字，根据关键字集合依次查找各个元素的值。第2种是能不能把Map看成是Set一样，只是Map集合中元素由两个对象组成，可以把这两个对象看成一个对象的两个属性。然后可以遍历了。

HashMap使用(4)

```
HashMap map = new HashMap();  
map.put("1001", "张军");  
map.put("1002", "李元");  
map.put("1003", "王钧");  
Set keys = map.keySet(); // 读取所有关键字集合  
Iterator it = keys.iterator(); // 遍历关键字集合  
while (it.hasNext()) {  
    String s = map.get(it.next()).toString(); // 通过关键字查找元素值  
    System.out.println(s);  
}
```

HashMap使用(5)

```
HashMap map = new HashMap();  
map.put("1001", "张军");  
map.put("1002", "李元");  
map.put("1003", "王钧");  
Set keys = map.entrySet(); //读取 Map 集合  
Iterator it = keys.iterator();  
//遍历 Map 中的元素，注意元素是 Key+Value 构成  
while (it.hasNext()) {  
    Map.Entry e = (Map.Entry) it.next(); //相当于将 Key, Value 变成一个对象两属性  
    System.out.println("key=" + e.getKey() + " value=" + e.getValue());  
}
```

TreeMap类使用(1)

- 其中的元素可以按照Key的自然顺序排列
- 作为**Key**的类需要实现**Comparable**接口
- 非同步的。可以变成同步的：

```
Map treeMap = new TreeMap();  
Map m =  
Collections.synchronizedMap(treeMap );
```

TreeMap类使用(2)

```

TreeMap map = new TreeMap();↵
map.put("1002", "张军");↵
map.put("1001", "李元");↵
map.put("1010", "王钧");↵
Set keys = map.entrySet();//读取 Map 集合↵
Iterator it = keys.iterator();//遍历 Map 中的元素，注意元素是 Key+Value 构成↵
while (it.hasNext()) {↵
    Map.Entry e=(Map.Entry)it.next();↵
    System.out.println("key="+e.getKey()+" value="+e.getValue());↵
}↵

```

TreeMap类使用(3)

- 按照学号倒序排列示例

第1步：创建一个比较算法类。

```
package chapter3.treecmp;
import java.util.Comparator;
public class MyCmp implements Comparator {
    public int compare(Object obj1, Object obj2) {
        int x = obj2.toString().compareTo(obj1.toString());
        return x;
    }
}
```

TreeMap类使用(4)

第2步：定义一个测试类。创建Map对象，测试该自定义算法。注意输出结果。

```
package chapter3.treecmp;↵
import java.util.*;↵
public class Test {↵
    public static void main(String[] args) {↵
        TreeMap map = new TreeMap(new MyCmp());↵
        map.put("1002", "周平");↵
        map.put("1001", "张军");↵
        map.put("1010", "张力");↵
        Set keys = map.entrySet();//读取 Map 集合↵
        Iterator it = keys.iterator();↵
        while (it.hasNext()) {//遍历 Map 中的元素，注意元素是 Key+Value 构成↵
            Map.Entry e=(Map.Entry)it.next();↵
            System.out.println("key="+e.getKey()+" value="+e.getValue());↵
        }↵
    }↵
}↵
```

HashSet使用(1)

- 主要的一个无重复的集合类。
- 元素没有顺序
- 为基本操作提供常数时间性能， $O(1)$
- 没有提供同步机制
- 后台使用HashMap实现

HashSet使用(2)

集合中元素是依据元素值和相应的哈希算法计算其地址，元素值相同地址就相同，值不同地址就不会相同。所以在HashSet集合中不存在元素值重复的元素。

HashSet使用(3)

```
HashSet Set set=new HashSet();  
set.add("a");//向集合中添加一个String  
set.add(new Integer(1)); //向集合中添加一个  
Integer  
int x[]={1,2,3,5};  
set.add(x); //向集合中添加数组  
Student s=new Student("1001","zhou");  
set.add(s); //向集合中添加一个自定义类的对象
```

HashSet使用(4)

遍历HashSet集合中的元素

集合中元素是依据元素值和相应的哈希算法计算其地址, 所以如何读取集合中的元素, 需要遍历算法即使用迭代器。所谓遍历是指按照某种顺序, 对于集合中每个元素仅访问一次。不重复也不遗漏。Iterator(迭代)是指获取集合中元素的过程, 实际上帮助获取集合中的元素

HashSet使用(5)

遍历HashSet

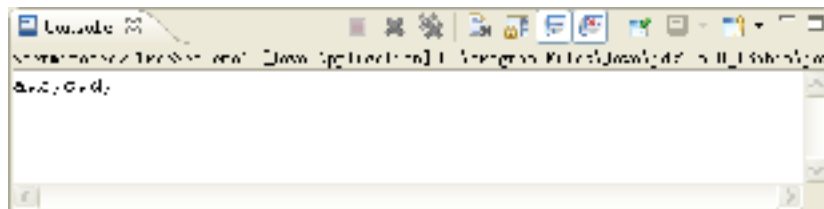
```
HashSet hs=new HashSet();  
hs.add(new String("Java"));  
hs.add(new String("c++"));  
hs.add(new Integer(100));  
hs.add(new Double(100.2));  
Iterator it=hs.iterator();  
while(it.hasNext()){//判断是否还有下一个元素  
    Object obj=it.next();//取迭代器中下一个值。  
    if(obj instanceof String)//判断是否是 String 类的实例  
        System.out.println("String:"+obj);  
    if(obj instanceof Integer)//判断是否是 Integer 类的实例  
        System.out.println("Integer:"+obj);  
    if(obj instanceof Double)//判断是否是 Double 类的实例  
        System.out.println("Double:"+obj);  
}
```

TreeSet 使用(1)

- 内部用TreeMap实现
- 是一个有序集合，可以按照一定的规则指定元素的顺序

TreeSet使用(2)

```
TreeSet tree = new TreeSet(); // 使用默认排序算法
tree.add("d"); // 通过 add() 方法向书中增加元素
tree.add("c");
tree.add("a");
tree.add("b");
Iterator it = tree.iterator();
while (it.hasNext()) {
    System.out.print(it.next() + ",");
}
```



TreeSet使用(3)

自定义比较函数实现

第 1 步：定义学生类↵

```
package chapter3.compare;↵  
public class Student {↵  
    private String sno, sname;↵  
    private int score;↵  
    public Student(String sno, String sname, int score) {↵  
        super();↵  
        this.sno = sno;↵  
        this.sname = sname;↵  
        this.score = score;↵  
    }↵  
    public String getSno() {↵  
        return sno;↵  
    }↵  
    public void setSno(String sno) {↵  
        this.sno = sno;↵  
    }↵  
    public String getSname() {↵  
        return sname;↵  
    }↵
```

TreeSet使用(4)

第 2 步：定义一个用于比较算法类。

```
package chapter3.compare;
import java.util.Comparator;
public class MyCmp implements Comparator {
    //向集合中添加元素时，自动调用 compare(Object obj1, Object obj2)
    //并将新增加的元素与原有元素比较，根据返回值决定新元素插入位置。
    //示例中：根据学生成绩排序，当成绩相同时按学号进行排序。
    public int compare(Object obj1, Object obj2) {
        int x = 0;
        Student s1 = (Student)obj1;
        Student s2 = (Student)obj2;
        if(s1.getScore() > s2.getScore()) {
            x = -1;
        } else if(s1.getScore() < s2.getScore()) {
            x = 1;
        } else { //当成绩相等时比较学号字典顺序。
            x = s1.getSno().compareTo(s2.getSno());
        }
        return x;
    }
}
```


TreeSet使用(5)

第3步：定义一个测试类

```
package chapter3.compare;

import java.util.*;

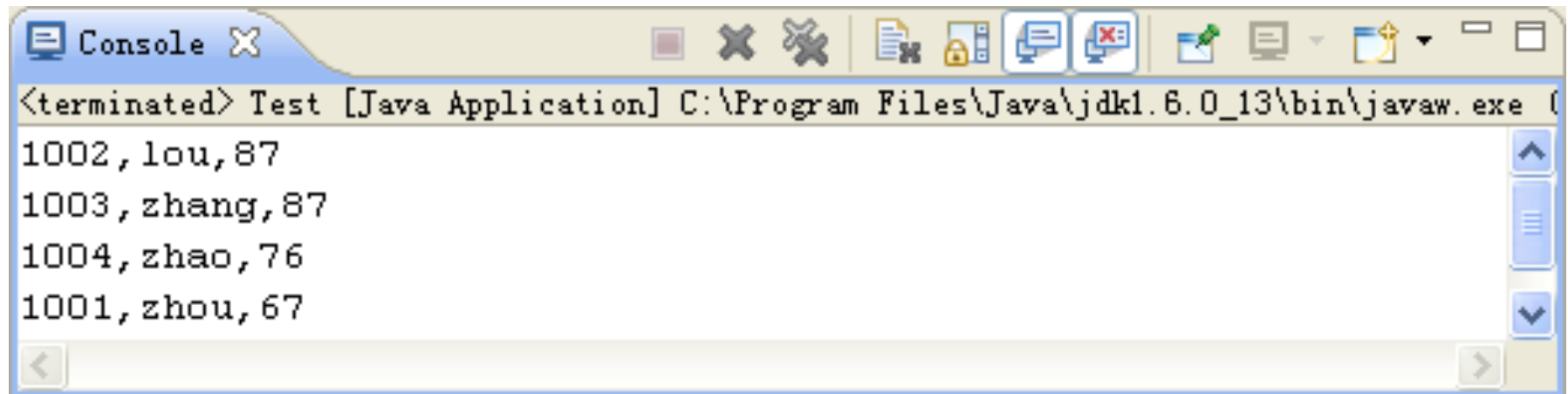
public class Test {

    public static void main(String[] args) {

        TreeSet tree=new TreeSet(new MyCmp());
        Student s1=new Student("1001","zhou",67);
        Student s2=new Student("1002","lou",87);
        Student s3=new Student("1003","zhang",87);
        Student s4=new Student("1004","zhao",76);
        tree.add(s1);tree.add(s2);
        tree.add(s3);tree.add(s4);
        Iterator it=tree.iterator();
        while(it.hasNext()){
            Student s=(Student)it.next();

            System.out.println(s.getSno()+" "+s.getSname()+" "+s.getScore());
        }
    }
}
```

TreeSet使用(6)

A screenshot of a Java console window titled "Console". The window shows the output of a Java application. The first line is "<terminated> Test [Java Application] C:\Program Files\Java\jdk1.6.0_13\bin\javaw.exe (". The subsequent lines are "1002, lou, 87", "1003, zhang, 87", "1004, zhao, 76", and "1001, zhou, 67". The console has a standard Windows-style toolbar with icons for file operations and a scroll bar on the right.

```
<terminated> Test [Java Application] C:\Program Files\Java\jdk1.6.0_13\bin\javaw.exe (  
1002, lou, 87  
1003, zhang, 87  
1004, zhao, 76  
1001, zhou, 67
```

Apache Commons项目collections包

- 提供JDK中的Collection、Set、List和Map接口的新实现类。
- 提供了格外的类，从而可以更灵活地操作Collection中的元素和Map中的键值对的。
- 下载地址：http://commons.apache.org/proper/commons-collections/download_collections.cgi

未完待续，谢谢！