



主讲教师 张 智  
计算机学院软件工程系  
课程群: 421694618

## 8 集合和泛型

### 8.1 集合

### 8.2 泛型

## 8.1 集合

### ■ 数组存在的不足：

- 数组初始化后大小不可变，只能存放类型一样的数据
- length只告诉了数组的容量，无法判断其中实际存有多少元素
- 数组只能按索引顺序存取

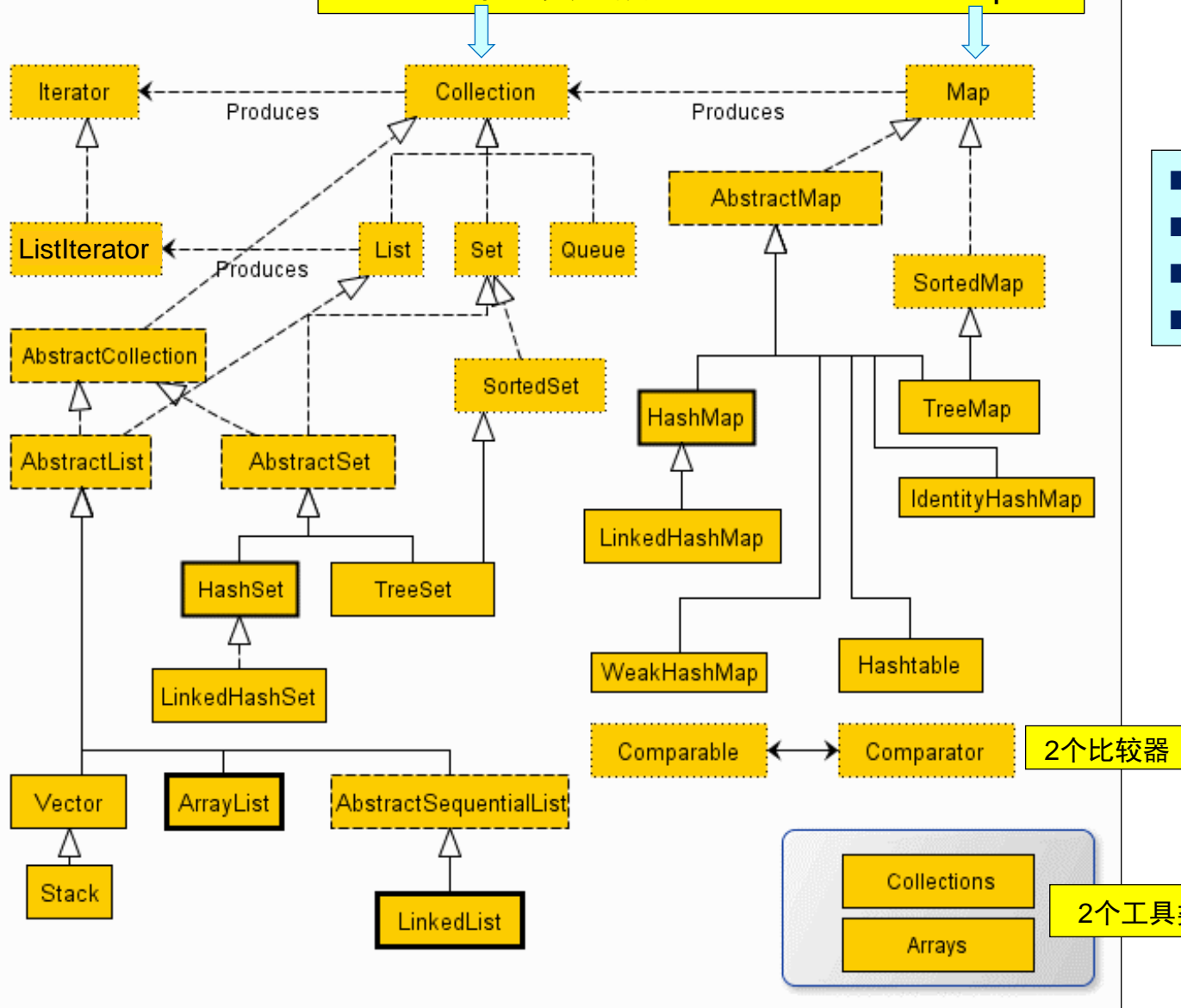
### ■ 集合基本特点：(也称容器)

Java集合类均在 `java.util` 包中

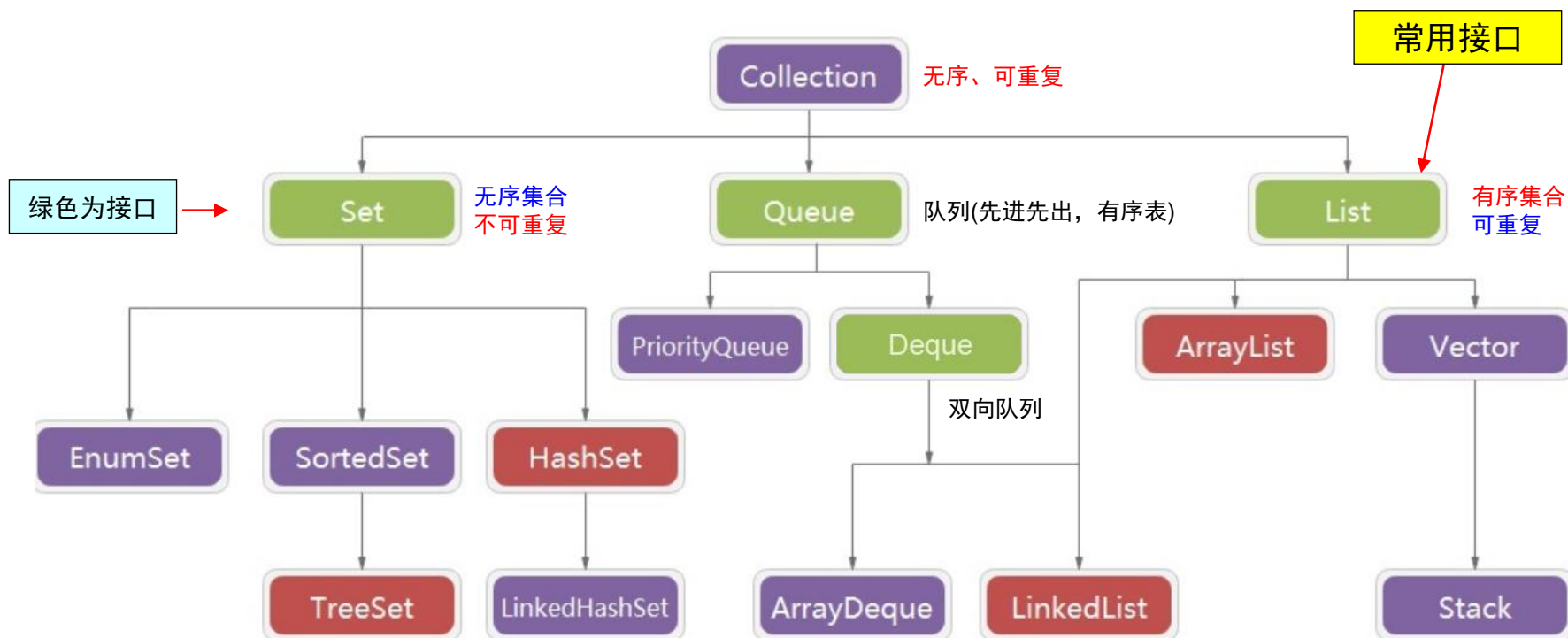
- 长度可变，可以存放不同的对象，可以确切知道元素的个数
- 集合以类的形式存在（具有封装、继承、多态等类的特性），通过简单的方法和属性即可实现各种复杂操作，提高开发效率

说明：集合里只能存放对象（实际上是对象的引用变量，但通常习惯上认为集合里保存的是对象）

## ★ Java集合类根接口：Collection 和 Map ★



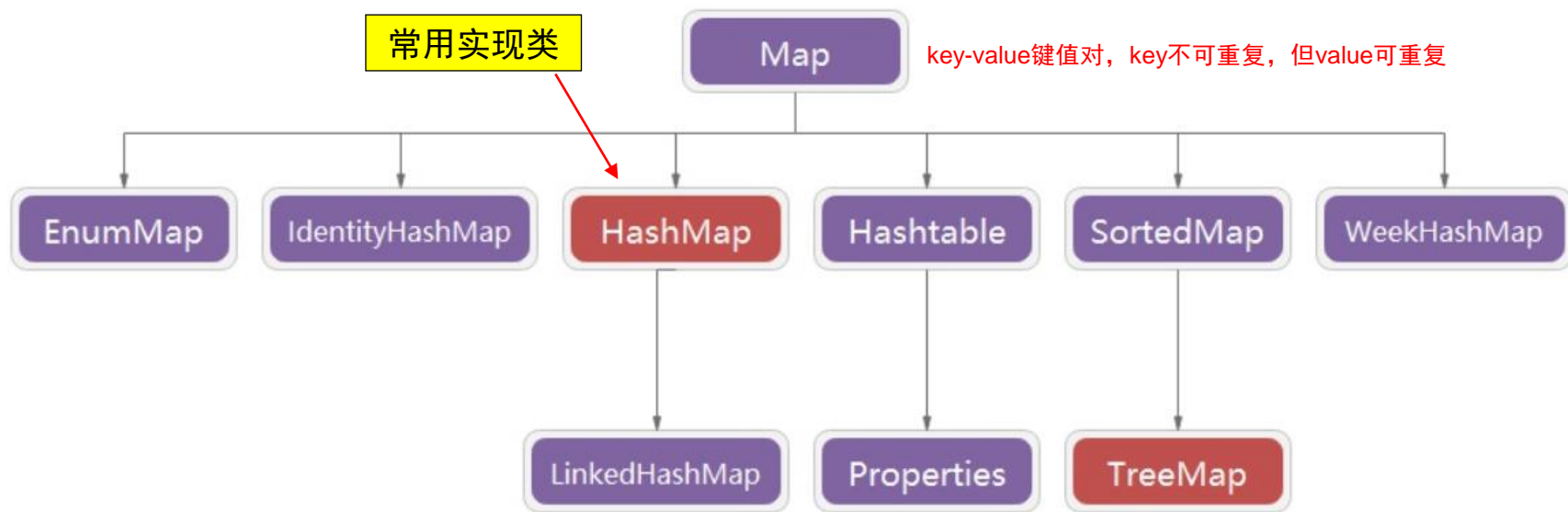
# Collection集合



## Collection主要实现类特点

实现类名称	主要特点
ArrayList	线性表，底层用Object数组实现，特点：访问快、增删慢
LinkedList	链表，底层用双向链表结构实现的，特点：访问慢、增删快
Vector	实现与ArrayList是一样，不过是线程安全的，操作效率低
Stack	堆栈：后进先出
HashSet	无序，唯一，基于 HashMap 实现的，至多有一个null元素
LinkedHashSet	底层哈希表 + 链表实现，元素不重复，与HashSet相比访问更快，增删稍慢
TreeSet	底层是红黑树(自平衡的排序二叉树)，有序，唯一
ArrayDeque	双向队列的数组实现，可在头部和尾部添加或者删除元素
PriorityQueue	优先队列，本质上是一个最小(大)堆，元素不允许null

## Map集合



## Map主要实现类特点

实现类名称	主要特点
HashMap	底层哈希表实现，元素存取顺序不能保证一致，最多允许一个键值为null，不支持线程同步
LinkedHashMap	底层用哈希表 + 双向链表实现
TreeMap	底层使用二叉树实现，可对集合中的键进行排序
Hashtable	几乎等价于HashMap，但不接受null键，支持线程同步



## 集合主要框架

- Collection

- List ✓

- Set

- Queue

- Map ✓

- 工具类：Collections和Arrays

- 集合排序问题 ✓

【[返回](#)】

## 1. Collection

最基本的集合接口，允许重复的对象，对象之间没有指定的顺序

### ■ 元素添加/删除操作：

E 表示元素的数据类型

- `boolean add( E e )`：向集合中添加一个元素e，成功则返回true
- `boolean remove(Object o)`：删除元素(多个时只删除第一个)，成功返回true

### ■ 查询操作：

- `int size()`：返回当前集合中元素的数量
- `boolean isEmpty()`：当前集合是否为空
- `boolean contains(Object e)`：当前集合是否包含e对象
- `Iterator iterator()`：返回一个迭代器，用于遍历集合各个元素

## ■ 组操作：

- `boolean containsAll(Collection c)`：集合中是否包含集合c所有元素
- `boolean addAll(Collection c)`：将集合c中的元素添加给该集合
- `void clear()`：删除集合中所有元素
- `void removeAll(Collection c)`：从集合中删除集合c中的所有元素
- `void retainAll(Collection c)`：从集合中删除集合c中不包含的元素

## ■ Collection转换为Object数组：

- `Object[ ] toArray()`：返回一个内含集合所有元素的数组

## Collection示例

注：Collection仅仅是一个接口，而真正使用的时候，是该接口的一个实现类(如ArrayList)

```
Collection list = new ArrayList(); //创建一个集合对象
```

```
list.add("000");
```

添加任意类型对象

```
list.add('a');
```

获取本地日期串

```
list.add( new SimpleDateFormat("yyyy-MM-dd").format(Calendar.getInstance().getTime()) );
```

```
list.add(111);
```

```
list.add(222.0);
```

实际元素个数

```
System.out.println("集合list的大小: "+ list.size() ); //5
```

```
System.out.println("集合list: "+ list ); // [000, a, 2021-09-10, 111, 222.0]
```

集合可以直接输出

```
list.remove("000"); //移除 "000" 这个对象
```

```
System.out.println("集合list移除000后: "+ list ); // [a, 2021-09-10, 111, 222.0]
```

```
System.out.println("集合list中是否包含000 : "+ list.contains("000") ); // false
```

## 代码续前

```
Collection list2 = new ArrayList();  
list2.addAll(list);    //将list元素全部都加到list2中（一般不用list2=list）  
System.out.println("集合list2的内容： "+ list2 );    // [a, 2021-09-10, 111, 222.0]  
list2.clear();    //清空集合元素  
System.out.println("集合list2是否为空： "+ list2.isEmpty() );    // true  
Object[ ] s= list.toArray(); ———→ 将集合转化为数组  
System.out.print( Arrays.toString(s) );    // [a, 2021-09-10, 111, 222.0]
```

## 集合Iterator：迭代器

- Iterator功能：遍历集合对象（程序员不必知道该集合底层结构）
- 创建迭代器的代价是轻量级的

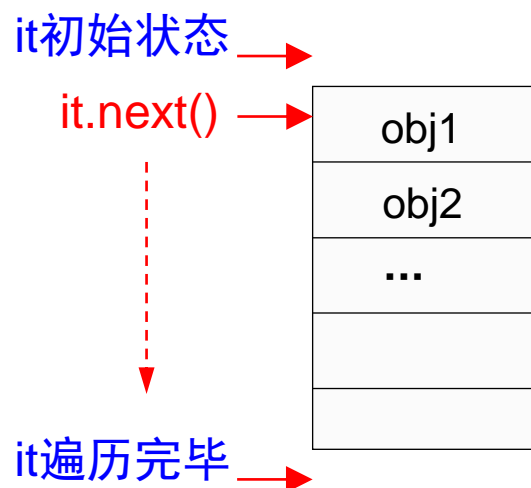


### 常用方法：

- `boolean hasNext()`：是否存在另一个可访问的元素
- `Object next()`：返回要访问的下一个元素（通常需要强转）
- `void remove()`：将迭代器新返回的元素删除

## Iterator遍历集合

```
Iterator it = Collection对象.iterator();    // 获得迭代器  
while( it.hasNext() ) {  
    Object obj = it.next();    // 得到下一个元素（通常需要强转）  
    //处理obj  
}
```



## Iterator遍历集合示例

调用forEach() + Lambda表达式遍历集合：  
list.forEach(obj -> System.out.println("集合元素: " + obj));

```
Collection list = new ArrayList();  
list.add("s1"); list.add("s2"); list.add("s3");  
Iterator it = list.iterator();    // 获得迭代器  
while ( it.hasNext() ) {          // 遍历  
    String element = (String) it.next(); //强转一下  
    System.out.println(element);  
}
```

**it = list.iterator();** // 再获得获取一次迭代器

```
while ( it.hasNext() ) {  
    Object element = it.next(); //没有强转  
    System.out.println("remove: " + element);  
    it.remove(); // 使用迭代器的移除元素操作，不要使用 list.remove(element);  
}
```

System.out.print( list );

运行结果

```
s1  
s2  
s3  
remove: s1  
remove: s2  
remove: s3  
[]
```

使用Iterator迭代过程中，不可在外部修改集合

一次性删除所有直接用：list.clear();

**【返回】**



## 2. List

- List 接口继承了 Collection 接口，定义一个允许重复项的有序集合
- List是按对象进入顺序进行保存对象
- List除了实现 Collection 所有方法外，还添加了面向位置的操作

List集合像一个数组，有序，长度可变



0	1	2	3	4	5	6
ele1	ele2	ele3	ele4	ele5	ele6	ele7

## List的两个常用实现类

接口	简述	实现类	操作特性	成员要求
List	基于索引的 对成员随机访问	<b>ArrayList</b> 线性表	提供快速的基于索引的成员访问，对尾部成员的增加和删除支持较好。	任意Object子类对象
		<b>LinkedList</b> 链表	对列表中任何位置的成员的增加和删除支持较好，但对基于索引的成员访问支持性能较差。	任意Object子类对象

## List面向位置的操作（部分）

 E 表示元素的数据类型

- `E get(int index)` : 取出index位置的元素（通常需要强转）
- `E set(int index, E e)` : 将index位置上的对象替换为e并返回老的元素
- `void add(int index, Object o)` : 添加对象o到index位置上
- `Object remove(int index)` : 删除index位置上的元素
- `boolean addAll(int index, Collection c)` : 在index位置后添加容器c中所有的元素
- `int indexOf(Object o)` : 查找对象o在List中第一次出现的位置(没找到则返回-1)
- `int lastIndexOf(Object o)` : 查找对象o在List中最后出现的位置(没找到则返回-1)
- `List<E> subList(int fromIndex, int toIndex)` : 返回一个子列表List，范围是 [ fromIndex, toIndex)元素(不含toIndex)

## List迭代器：ListIterator


### ■ ListIterator listIterator()

- 返回一个ListIterator 迭代器，默认开始位置为0

### ■ ListIterator listIterator(int startIndex)

- 返回一个ListIterator 迭代器，开始位置为startIndex

- ListIterator 接口继承了 Iterator 接口
- 支持添加或更改底层集合中的元素
- 支持双向访问

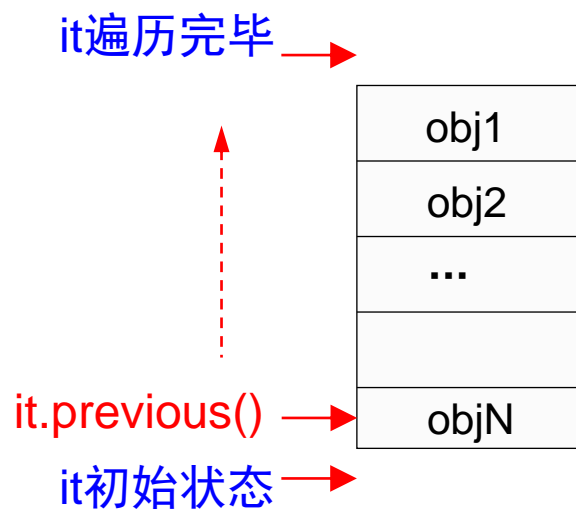


```
add(E) : void
hasNext() : boolean
hasPrevious() : boolean
next() : E
nextIndex() : int
previous() : E
previousIndex() : int
remove() : void
set(E) : void
```

## ListIterator逆向遍历

```
ListIterator it = list.listIterator( list.size() ); // list为一个List对象
while ( it.hasPrevious() ) {
    Object element = it.previous();
    // 处理元素
}
```

关键点



List  
示  
例

```
List list = new ArrayList();    // List接口比Collection接口更常用 ←
list.add("aaa");    list.add(0,100);    // 添加100到第0位置（插入）
list.add("ccc");    list.add("ddd");
list.set(3, 200);    // 修改index=3对象
System.out.println( list );    // [100, aaa, ccc, 200]
System.out.println( (String) list.get(2) );    // ccc
System.out.println( list.listIterator(1).next() );    // aaa（想一想）
System.out.println( list.subList(1, 3) );    // [aaa, ccc]
ListIterator it = list.listIterator();
while ( it.hasNext() ) {    → 顺序遍历List集合
    System.out.println( it.next() );
}
it = list.listIterator( list.size() );
while ( it.hasPrevious() ) {    → 逆向遍历List集合
    System.out.println( it.previous() );
}
```

## 遍历List集合的其他方法

```
List list = new ArrayList();  
list.add("s1");  
list.add("s2");  
list.add("s3");  
// for-each循环遍历集合  
for(Object o:list) {  
    System.out.println(o);  
}  
// for循环遍历集合  
for( int i=0; i < list.size(); i++ ) {  
    Object o = list.get(i);  
    System.out.println(o);  
}  
// 使用集合的forEach() + Lambda表达式遍历集合  
list.forEach( obj -> System.out.println(obj) );
```

## LinkedList的方法：部分

- `LinkedList()`：创建一个空的双向链接列表
- `LinkedList(Collection c)`：创建一个双向链接列表，并添加集合c所有元素
- `LinkedList`添加了一些处理列表两端元素的方法：
  - `void addFirst(E e)`：将对象e添加到列表的开头
  - `void addLast(E e)`：将对象e添加到列表的结尾
  - `E getFirst()`：返回列表开头的元素
  - `E getLast()`：返回列表结尾的元素
  - `E removeFirst()`：删除并返回列表开头的元素
  - `E removeLastt()`：删除并返回列表结尾的元素

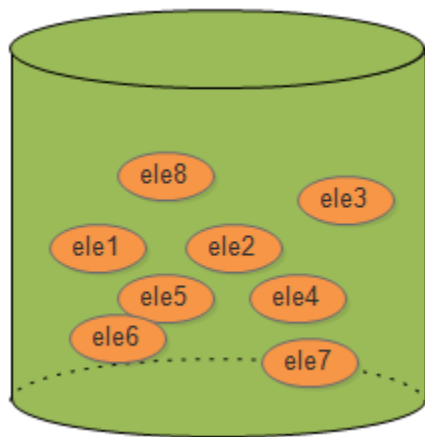
[【返回】](#)



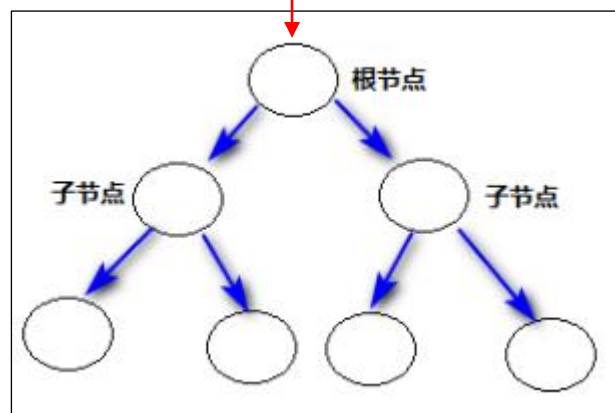
### 3. Set



- Set 接口继承了 Collection 接口，集合元素**无序**且**无重复**
  - 无重复原理：每个Set实现类依赖添加的对象的 equals() 方法来检查独特性，即任意两个元素e1和e2，都有e1.equals(e2)=false
- Set接口没有引入新方法

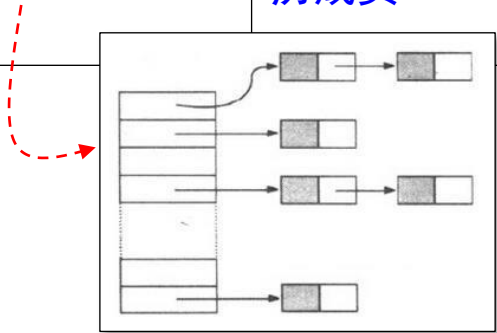


TreeSet 有序的，底层是二叉树结构(红黑树)



# Set接口的常用实现类

接口	简述	实现类	操作特性	成员要求
Set	成员不能重复	HashSet 无序，唯一	外部无序地遍历成员 (查找效率高)	成员可为任意Object子类的对象，但如果覆盖了equals方法，同时注意修改hashCode方法。
		TreeSet 有序，唯一 底层是自平衡二叉树结构(红黑树)	外部有序地遍历成员； 特殊：附加实现了SortedSet, 支持子集等要求顺序的操作	成员要求实现Comparable接口，或者使用Comparator构造TreeSet。 成员一般为同一类型(不能为null)。
		LinkedHashSet	外部按成员的插入顺序遍历成员	成员与HashSet成员类似



## Set接口示例

```
Set hashset = new HashSet();
Set linkset = new LinkedHashSet();
for( int i=0; i<5; i++ ){
    //产生一个随机数
    int s=(int) (Math.random()*100);
    hashset.add( Integer.valueOf(s) );
    linkset.add( Integer.valueOf(s) );
}
System.out.println("HashSet: "+ hashset);
System.out.println("LinkedHashSet: "+ linkset);
// 使用TreeSet进行重构和排序（厉害）
Set sortedset = new TreeSet( hashset );
System.out.println("排序后 TreeSet : "+sortedset);
```

程序的一次执行可能结果为：

HashSet: [64, 96, 95, 57, 14]

无序且无重复

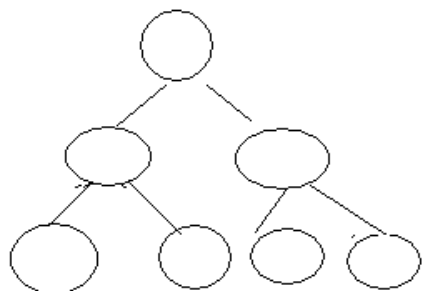
LinkedHashSet: [96, 64, 14, 95, 57]

排序后 TreeSet : [14, 57, 64, 95, 96]

TreeSet是有序的

## TreeSet图例（红黑树）

TreeSet：底层是二叉树结构。（红黑树是一种自平衡的二叉树）



```
TreeSet<Integer> ts = new TreeSet<Integer>();
```

```
// 创建元素并添加
```

```
// 20, 18, 23, 22, 17, 24, 19, 18, 24
```

```
ts.add(20);
```

```
ts.add(18);
```

```
ts.add(23);
```

```
ts.add(22);
```

```
ts.add(17);
```

```
ts.add(24);
```

```
ts.add(19);
```

```
ts.add(18);
```

```
ts.add(24);
```

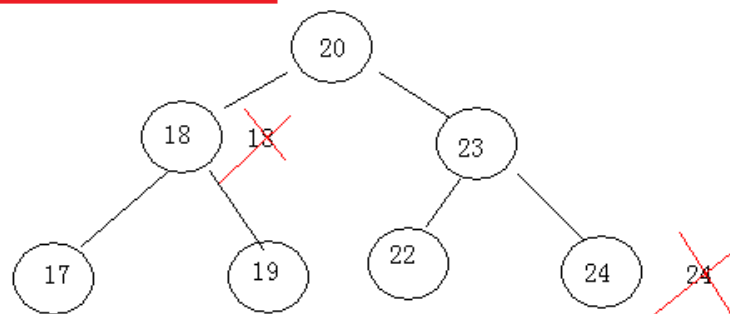
```
// 遍历
```

```
for (Integer i : ts) {  
    System.out.println(i);  
}
```

元素是如何存储进去的呢？

第一个元素存储的时候，直接作为根节点存储。  
从第二个元素开始，每个元素从根节点开始比较

大 就作为右孩子  
小 就作为左孩子  
相等 就不搭理它



元素是如何取出来的呢？（前序遍历，中序遍历，后序遍历）

从根节点开始，按照左，中，右的原则依次取出元素即可。

17    18    19    20    22    23    24

先序遍历

## TreeSet遍历：

### ■ Iterator顺序遍历：

```
Iterator it = treeset.iterator();  
while( it.hasNext() ) {  
    Object obj = it.next();  
}
```

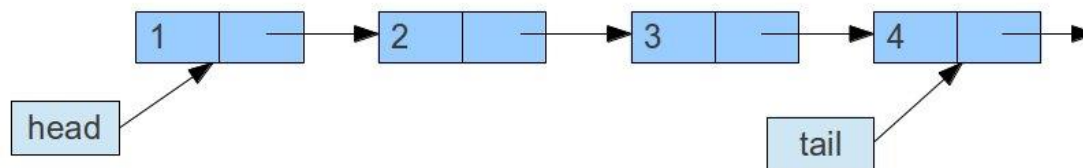
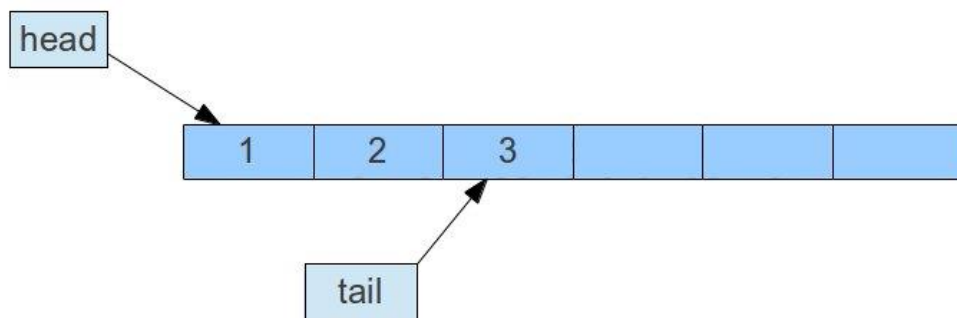
### ■ Iterator逆序遍历（TreeSet独有）：

```
Iterator it = treeset.descendingIterator();  
while( it.hasNext() ) {  
    Object obj = it.next();  
}
```

[【返回】](#)

## 4. Queue

- Queue 接口继承了 Collection 接口
- 是一种先入先出的模型 (FIFO)，有数组和链表两种实现形式



# Queue实现类

接口	简述	实现类	操作特性	成员要求
Queue	队列 (先进先出)	ArrayDeque	双向队列的数组实现(循环数组), 可在头部和尾部添加或者删除元素, 自动扩容, 支持迭代器遍历	任意Object子类对象, 但不允许null
		PriorityQueue	优先队列, 本质上是一个最小(大)堆, 队头元素指排序规则最小(大)那个元素, 自动扩容	元素不允许null, 也不允许插入不可比较的对象(没有实现Comparable接口的对象)

## Queue常用操作

E 表示元素的数据类型

- `boolean offer(E e);` // 添加元素到队尾，失败返回false
- `E poll();` // 获取队首元素，并从队列中移除，失败返回false或null
- `E peek();` // 获取队首元素，但不从队列中移除，失败返回false或null
- `add(E e);` // 添加元素到队尾，失败抛出异常
- `E remove();` // 获取队首元素，并从队列中移除，失败抛出异常
- `E element();` // 获取队首元素，但不从队列中移除，失败抛出异常



## Queue接口示例

```
Queue queue = new ArrayDeque();
```

```
//添加元素
```

```
queue.offer("a");
```

```
queue.offer("b");
```

```
queue.offer("c");
```

```
queue.offer("d");
```

```
queue.offer("e");
```

```
System.out.println("初始队列: " + queue );
```

```
System.out.println("poll=" + queue.poll() );
```

//返回第一个元素，并在队列中删除

```
System.out.println("peek=" + queue.peek() );
```

//返回第一个元素

```
System.out.println("element=" + queue.element() );
```

//返回第一个元素

```
System.out.println("最终队列: " + queue);
```

运行结果

初始队列: [a, b, c, d, e]

poll=a

peek=b

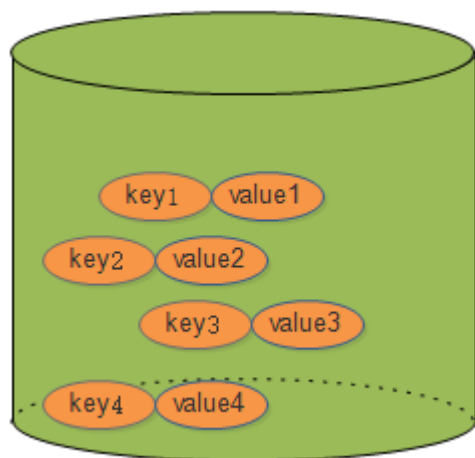
element=b

最终队列: [b, c, d, e]

[【返回】](#)

## 5. Map

- Map接口含有两个部分(两列): **关键字和值 (key-value, 简称键值对)**
- 一个key映射一个value (一对一的映射):
  - 添加数据时, 如果关键字重复, 则用新值替换原有的值



# Map实现类

接口	简述	实现类	操作特性	成员要求
Map	键值对成员，基于键找值操作	HashMap 常用	底层采用哈希表结构，元素的存取顺序不能保证一致。由于要保证键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。	键成员可为任意Object对象
		TreeMap 有序的	支持对键有序地遍历，附加实现了SortedMap接口，支持要求顺序的操作	键成员要求实现Comparable或使用Comparator接口，TreeMap键成员一般为同一类型。
		LinkedHashMap 保证键值对顺序	底层采用的哈希表+双向链表结构。通过链表结构保证键值对的插入顺序；通过哈希表结构可以保证的键的唯一、不重复，需要重写键的hashCode()方法、equals()方法。	成员与HashMap成员类似

## Map基本操作

### ■ 添加、删除操作：

V 表示元素的数据类型

- **V put(K key, V value)**: 将一个键值对存放到Map中。如果关键字已存在，则用新值替换映射中原有的值
- **V remove(Object key)**: 根据key，移除一个键值对，并将值返回
- **void putAll(Map map)**: 将另外一个Map中的元素存入当前的Map中
- **void clear()** : 清空当前Map中的元素

说明：关于put()方法返回值

如果关键字原先就不存在，则返回null，否则返回关键字的旧值

## Map基本操作

### ■ 查询操作：

- **V get(Object key)**：根据key取得对应的值（通常需要强转）
- **boolean containsKey(Object key)**：判断Map中是否存在某key
- **boolean containsValue(Object value)**：判断Map中是否存在某值value
- **int size()**：返回Map中键值对的个数
- **boolean isEmpty()**：判断当前Map是否为空

## Map基本操作

- 允许把键或值的组作为集合来处理：
  - **Set keySet()** : 返回所有的键, 并使用Set存放 (无序唯一)
  - **Set entrySet()** : 返回Map集合中所有的键值对对象(Map.Entry) ,以Set存放
  - **Collection values()** : 返回所有的值, 用Collection存放(可重复)

## HashMap常用

Map  
示例

```
Map map = new HashMap();
map.put("id", "001");           //key="id" value="001"
map.put("name", "小明");       //key="name" value="小明"
map.put("age", 20);
map.put("age", 19);           //覆盖老值

System.out.println( map.size() );           //3
System.out.println( map.get("age") );       //19
System.out.println( map.remove("age") );    //19
System.out.println( map.containsKey("address") ); // false
System.out.println( map.containsValue("小明") ); // true
System.out.println("keySet=" + map.keySet() ); // keySet=[name,id]
System.out.println("values=" + map.values() ); // values=[小明,001]
System.out.println("entrySet=" + map.entrySet() ); //entrySet=[name=小明, id=001]
map.clear();           // 清空map
System.out.println( map.isEmpty() );       // true
```

key值	value值
id	"001"
name	"小明"
age	19

## Map的遍历：使用迭代器遍历

### ■ 使用 entrySet() 遍历：常用

```
Set entrySet = map.entrySet();    // ① 获取键值对对象(Map.Entry)集合(Set集合)
Iterator it = entrySet.iterator(); // ② 使用Iterator遍历键值对集合
while ( it.hasNext() ) {
    Map.Entry me = (Map.Entry) it.next();    // ③ 获得键值对对象
    Object key = me.getKey();                // ④ 从键值对对象中获取key值
    Object value = me.getValue();           // ⑤ 根据key获得对应的value值
    System.out.println("key=" + key + " value=" + value);
}
```

使用for-each遍历entrySet()详见泛型



## Map的遍历：使用迭代器遍历（续）

### ■ 使用 keySet() 遍历：效率较低

```
Set keySet = map.keySet();    // ① 获取key集合(Set集合)
Iterator it = keySet.iterator(); // ② 使用Iterator遍历key集合
while ( it.hasNext() ) {
    Object key = it.next();      // ③ 获得当前key值
    Object value = map.get(key); // ④ 根据key获得对应的value值
    System.out.println("key=" + key + " value=" + value);
}
```

使用for-each遍历keySet ()详见泛型

## Map的遍历：

- 使用 values() 遍历：value值

```
// 遍历值集合（Iterator用法略）  
for ( Object value : map.values() ) {  
    System.out.println(value);  
}
```

**【[返回](#)】**

## 6. 工具类：Collections

- Collections是针对集合的工具类，提供了**排序、反转、求最值、二分查找**等功能，大大提高了开发人员工作效率。

Collections常用方法	说明
<code>sort(List&lt; T&gt; list)</code>	根据自然顺序(升序)对指定list集合排序
<code>sort(List&lt; T&gt; list,Comparator&lt; ? super T&gt; c)</code>	list-集合;c比较器：按指定比较器c为list排序
<code>max(Collection&lt; ? extends T&gt; collection)</code>	根据自然顺序(升序)排序，返回collection的最大元素
<code>min(Collection&lt; ? extends T&gt; collection)</code>	根据自然顺序(升序)排序，返回collection的最小元素
<code>binarySearch(List&lt;&gt;list,T key)</code>	list-集合，key-指定对象：对List二分查找key（必须先自然升序排序）
<code>fill(List list,T obj)</code>	list-集合，obj-指定元素：使用指定元素替换List中所有元素（填充）
<code>replaceAll(List list,T oldVal,T newVal)</code>	list-集合，oldVal-老值，newVal-新值：使用一个新值替换List中的旧值
<code>reverse(List list)</code>	反转List元素的顺序
<code>shuffle(List list)</code>	对List中的元素进行随机排序（洗牌）

## Collections工具类示例

```
List list = new ArrayList();  
list.add("dog"); list.add("cat"); list.add("sheep"); list.add("horse"); list.add("cock");  
System.out.println("排序前: "+list);  
  
Collections.sort(list);    //默认自然序  
System.out.println("排序后: "+list);  
System.out.println("list最大元素: " + Collections.max(list) );  
  
int index = Collections.binarySearch(list,"cat");    //二分查找必须先自然升序排序  
System.out.println("cat序号值: "+index);  
int index2 = Collections.binarySearch(list,"cow");  
System.out.println("cow序号值: "+index2);  
  
Collections.reverse(list);    //反转  
System.out.println("反转后: "+list);
```

### 运行结果

```
排序前: [dog, cat, sheep, horse, cock]  
排序后: [cat, cock, dog, horse, sheep]  
list最大元素: sheep  
cat序号值: 0  
cow序号值: -3  
反转后: [sheep, horse, dog, cock, cat]
```

如果找到, 则返回索引值  
否则返回 负的插入点值

## 工具类：Arrays

- Arrays是针对数组的工具类，提供了**排序，查找，二分查找**等功能。

Arrays常用方法	说明
<code>sort(array)</code>	对指定的 <b>基本数据类型数组array</b> 按升序排列
<code>binarySearch(array, val)</code>	对 <b>基本数据类型数组array</b> 进行二分查找val
<code>equals(array1, array2)</code>	如果两个指定的 <b>基本数据类型数组</b> 相等返回true
<code>fill(array, val)</code>	将指定的 <b>基本数据类型值数组array</b> 的所有元素都赋值为val (填充)
<code>copyof(array, length)</code>	把 <b>基本数据类型数组array</b> 复制成一个长度为length的新数组
<code>toString(array)</code>	把 <b>基本数据类型数组array</b> 内容转换为字符串
<code>asList(T... a)</code>	把数组a转换成List<T>集合（注：返回的List集合比较特殊，如不支持增删操作等）

## Arrays工具类示例

```
String[] str1 = {"1", "2", "3"};  
String[] str2 = {"1", "2", new String("3")};  
System.out.println(Arrays.equals(str1, str2)); // true
```

```
int[] score = {79, 65, 93, 64, 88};
```

```
Arrays.sort(score);
```

```
String str = Arrays.toString(score);
```

```
System.out.println("排序后: " + str);
```

```
int index = Arrays.binarySearch(score, 88);
```

```
System.out.println("88的序号值: " + index);
```

```
int[] a = Arrays.copyOf(score, 3);
```

```
System.out.println("复制后的数组: " + Arrays.toString(a));
```

```
Integer[] aa = new Integer[]{1, 2, 3};
```

```
List list = Arrays.asList(aa);
```

```
System.out.println(list); // 如果是int基本类型数组则无法打印结果
```

### 运行结果

true

排序后: [64, 65, 79, 88, 93]

88的序号值: 3

复制后的数组: [64, 65, 79]

[1, 2, 3]

//Java 8之后数组转为集合: 能将基本类型数组封装为对象数组  
List list = Arrays.stream(int数组).boxed().collect(Collectors.toList());

**【返回】**

## 7. 集合排序问题

- `Collections.sort(List<T> list)`可以实现元素的**自然排序**。
- 自然排序定义：  $\{ (x, y) \mid x.compareTo(y) \leq 0 \}$  **从小到大排序**

类	自然排序
数值类型，如Integer、Double等	按数字从小到大排序
Character	按 Unicode 值的从小到大排序
Date	按年代从小到大排序
String	按字符串中字符 Unicode 值从小到大排序

## 排序示例：自然序 -- 从小到大排序

```
public class Test {  
    public static void main(String[] args) {  
        List<String> list = new ArrayList<String>();  
        list.add("100");  
        list.add("50");  
        list.add("120");  
        Collections.sort( list );  
        System.out.println( list );    // [100, 120, 50]  
    }  
}
```



## 自定义排序：方法1--使用Comparator接口

### ■ 先自定义一个比较器类实现 `java.util.Comparator` 接口：

#### ■ 实现 `int compare(Object o1, Object o2)` 方法 (两个参数)：

- 返回负数(任意负值, 如-1): 表示o1位于o2之前
- 返回正数(任意正值, 如1): 表示o1位于o2之后
- 返回0: 表示o1和o2相等 (表示两个对象排在同一位置)

### ■ 然后调用 `Collections.sort(List list, Comparator c)` 完成自定义排序

记住比较器返回值：（后面示例总结的）

x,y分别对应第1个、第2个参数

- 从小到大排序: `return x-y;` (第1个减第2个)
- 从大到小排序: `return y-x;` (第2个减第1个)

## 自定义的比较器类：从大到小排序

```
class myComparator implements Comparator{
```

```
    @Override
```

```
    public int compare(Object o1, Object o2) { // 实现compare()方法
```

```
        int x = (int) o1;
```

```
        int y = (int) o2;
```

```
        if( x > y )
```

```
            return -1;
```

```
        else
```

```
            return 1;
```

```
    }
```

```
}
```

替换掉

普通写法

变个形

```
if( y-x<0 )
```

```
    return y-x;
```

```
else
```

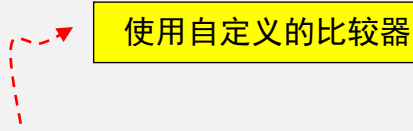
```
    return y-x;
```

简写

```
return y-x;
```

## Comparator比较器排序示例：从大到小排序

```
public class Test {  
    public static void main(String[] args) {  
        List<Integer> list = new ArrayList<Integer>();  
        list.add(100);  
        list.add(50);  
        list.add(120);  
        Collections.sort( list, new myComparator() );  
        System.out.println( list );    // [120,100,50]  
    }  
}
```



使用自定义的比较器

## 重要问题：对象集合的排序

### ■ 准备工作：Point类

```
class Point {  
    int x;  
    int y;  
    public Point( ) {  
    }  
    public Point(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    @Override  
    public String toString() {  
        return "(x = " + x + ", y = " + y + ")";  
    }  
}
```

假设排序规则要求：

1. x值小的对象排在前面
2. 如果x值相同，则y值小的对象排在前面

## 自定义的比较器类：

记住比较器返回值：

x,y分别对应第1个、第2个参数（x, y换成对象）

- 从小到大排序：return x.属性-y.属性；（第1个减第2个）
- 从大到小排序：return y.属性-x.属性；（第2个减第1个）

```
class myComparator implements Comparator{  
    public int compare(Object o1, Object o2) { // 实现compare()方法  
        Point p1 = (Point) o1;  
        Point p2= (Point)o2;  
        if ( p1.x != p2.x )  
            return p1.x - p2.x;  
        else  
            return p1.y - p2.y;  
    }  
}
```

假设排序规则要求：

1. x值小的对象排在前面
2. 如果x值相同，则y值小的对象排在前面

← 转化 →

练习：尝试修改一下排序规则

## 对象集合排序：

```
List<Point> list = new ArrayList<Point>();
```

```
list.add(new Point(3,2));
```

```
list.add(new Point(1,3));
```

```
list.add(new Point(1,2));
```

```
System.out.println("排序前：");
```

```
System.out.println(list); // 直接输出
```

使用自定义的比较器

```
Collections.sort( list, new myComparator() );
```

```
System.out.println("排序后：");
```

```
Iterator it = list.iterator();
```

```
while( it.hasNext() ) { // 遍历一下
```

```
    Point p = (Point) it.next();
```

```
    System.out.println(p);
```

```
}
```

运行结果

排序前：

[(x = 3, y = 2), (x = 1, y = 3), (x = 1, y = 2)]

排序后：

(x = 1, y = 2)

(x = 1, y = 3)

(x = 3, y = 2)

## 自定义排序：方法2--使用Comparable接口



- 在定义对象类时实现比较接口 `java.lang.Comparable`:

- 实现接口的 `int compareTo(Object o)` 比较方法 (1个参数):

- 返回值为负：表示当前对象this应该排在对象O的前面
- 返回值为正：表示当前对象this应该排在对象O的后面
- 返回值=0，表示两个对象并列

- 然后调用 `Collections.sort( List list )` 完成自定义排序

x, y分别对应第1个、第2个参数 (x换成this)

- 从小到大排序：return this.属性-y.属性; (第1个减第2个)
- 从大到小排序：return y.属性-this.属性; (第2个减第1个)

```
class Point implements Comparable {  
    int x;    int y;  
    public Point() { }  
    public Point(int x, int y) {    this.x = x;    this.y = y;    }  
    @Override  
    public String toString() {  
        return "[x = " + x + ", y = " + y + "];"  
    }  
    @Override  
    public int compareTo(Object o) {    //实现compareTo方法  
        Point p = (Point) o;  
        if ( this.x != p.x )  
            return this.x - p.x;  
        else  
            return this.y - p.y;  
    }  
}
```

对象类直接实现比较接口

假设排序规则要求:

1. x值小的对象排在前面
2. 如果x值相同, 则y值小的对象排在前面



## 对象集合排序：

```
List<Point> list = new ArrayList<Point>();
```

```
list.add(new Point(3,2));
```

```
list.add(new Point(1,3));
```

```
list.add(new Point(1,2));
```

```
System.out.println("排序前：");
```

```
System.out.println(list); // 直接输出
```

```
Collections.sort( list ); //自动按对象类自定义排序规则排序
```

```
System.out.println("排序后：");
```

```
Iterator it = list.iterator();
```

```
while( it.hasNext() ) { // 遍历一下
```

```
    Point p = (Point) it.next();
```

```
    System.out.println(p);
```

```
}
```

### 运行结果

排序前：

[(x = 3, y = 2), (x = 1, y = 3), (x = 1, y = 2)]

排序后：

(x = 1, y = 2)

(x = 1, y = 3)

(x = 3, y = 2)

## Comparator与Comparable接口比较

- Comparator是在集合外部实现的排序
- Comparable是在集合内部实现的排序
  - 一个类实现了Comparable接口则表明该类对象之间是可相互比较的，该类对象组成的集合就可以直接使用Collections.sort()排序
- Comparator一种比较器，能将算法和数据分离，通过Comparator来实现排序而不必改变对象本身
- 可定义多种Comparator为同一个集合对象使用

## 编程练习

- Person类：id, name, age;
  - (1) 只按age升序排序
  - (2) 只按id降序排序
  - (3) 按age升序排序，age相同时按id降序排序
- 要求使用：Comparator接口实现

```
class Person {  
    private int id;  
    private String name;  
    private int age;  
    public Person() {  
    }  
    public Person(int id,String name,int age) {  
        this.id = id;    this.name = name;    this.age = age;  
    }  
    public int getId() {    return id;    }  
    public void setName(String name) {    this.name = name;    }  
    public String getName() {    return name;    }  
    public void setAge(int age) {    this.age = age;    }  
    public int getAge() {    return age;    }  
    @Override  
    public String toString(){  
        return "Id: " + id + " Name: " + name + " Age: " + age;  
    }  
}
```

定义3个比较器

```
class cmpAge implements Comparator {    // 按age降序排序
    @Override
    public int compare(Object obj1, Object obj2) {
        Person p1 = (Person) obj1;
        Person p2 = (Person) obj2;
        return p1.getAge() - p2.getAge();
    }
}

class cmpId implements Comparator {    // 按id升序排序
    @Override
    public int compare(Object obj1, Object obj2) {
        Person p1 = (Person) obj1;
        Person p2 = (Person) obj2;
        return p2.getId() - p1.getId();
    }
}
```

定义3个比较器

```
// 按age升序，age相同时按id降序排序
class cmpAgeAndId implements Comparator {
    @Override
    public int compare(Object obj1, Object obj2) {
        Person p1 = (Person) obj1;
        Person p2 = (Person) obj2;
        if( p1.getAge() != p2.getAge() ) {
            return p1.getAge() - p2.getAge();
        }
        else {
            return p2.getId() - p1.getId();
        }
    }
}
```

```

List<Person> list = new ArrayList<Person>();
list.add(new Person(1003, "张三", 18));
list.add(new Person(1008, "李四", 21));
list.add(new Person(1015, "王五", 18));
list.add(new Person(1001, "赵六", 20));
Collections.sort(list, new cmpAge());
System.out.println("按age升序排序: ");
Iterator it1 = list.iterator();
while (it1.hasNext()) {
    System.out.println(it1.next());
}
Collections.sort(list, new cmpId());
System.out.println("按id降序排序: ");
Iterator it2 = list.iterator();
while (it2.hasNext()) {
    System.out.println(it2.next());
}
Collections.sort(list, new cmpAgeAndId());
System.out.println("按age升序, age相同时按id降序: ");
Iterator it3 = list.iterator();
while (it3.hasNext()) {
    System.out.println(it3.next());
}

```

## 运行结果

按age升序排序:

```

Id: 1003  Name: 张三  Age: 18
Id: 1015  Name: 王五  Age: 18
Id: 1001  Name: 赵六  Age: 20
Id: 1008  Name: 李四  Age: 21

```

按id降序排序:

```

Id: 1015  Name: 王五  Age: 18
Id: 1008  Name: 李四  Age: 21
Id: 1003  Name: 张三  Age: 18
Id: 1001  Name: 赵六  Age: 20

```

按age升序, age相同时按id降序:

```

Id: 1015  Name: 王五  Age: 18
Id: 1003  Name: 张三  Age: 18
Id: 1001  Name: 赵六  Age: 20
Id: 1008  Name: 李四  Age: 21

```

## 8.2 泛型

- Java集合有个缺点（为了更好的通用性）：当把一个对象"丢进"集合之后，集合就会"忘记"该对象的数据类型，当再次取出该对象时，该对象的编译类型就变成了Object类型
- 带来的问题：
  - 取出集合元素通常要进行强制转换，既增加了编程的复杂度，也可能引发 ClassCastException 异常



泛型可以在编译的时候检查类型安全，并且所有的强制转换都是自动和隐式的



## 泛型用法

泛型就是定义一种模板

- 泛型集合
- 泛型类
- 泛型方法
- 通配符
- 泛型接口

**[返回](#)**

## 1. 泛型集合

引例：

```
List list = new ArrayList(); //普通集合
```

```
list.add(100);
```

```
list.add(200);
```

```
list.add("300");
```

```
int sum=0;
```

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

```
    sum += (int) list.get(i); // 编译通过，运行报错
```

```
}
```

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

java.lang.ClassCastException: String cannot be cast to Integer



## 使用泛型集合：

添加<类型参数>

```
List<Integer> list = new ArrayList<Integer>();  
list.add(100);  
list.add(200);  
// list.add("300"); // 编译器自动对类型进行检查：此时编译不能通过  
int sum=0;  
for (int i = 0; i < list.size(); i++) {  
    sum += list.get(i);  
}  
System.out.println("sum=" + sum);
```

不必进行强转

## 泛型Map示例1：

key和value确保都是String类型

```
Map<String, String> map = new HashMap<String, String>();  
map.put("武汉大学", "https://www.wust.edu.cn");  
map.put("华中科技大学", "https://www.hust.edu.cn");  
  
for ( Map.Entry<String, String> entry : map.entrySet() ) {  
    String mapKey = entry.getKey();  
    String mapValue = entry.getValue();  
    System.out.println(mapKey + ": " + mapValue);  
}
```

使用for-each循环遍历entrySet()

## 泛型Map示例2:

```
class Book {  
    private int Id;           // 图书编号  
    private String Name;     // 图书名称  
    private int Price;       // 图书价格  
  
    public Book(int id, String name, int price) {  
        this.Id = id;  
        this.Name = name;  
        this.Price = price;  
    }  
  
    @Override  
    public String toString() {  
        return this.Id + "-" + this.Name + "-" + this.Price;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Book book1 = new Book(1001, "三国演义", 35);  
        Book book2 = new Book(1002, "水浒传", 28);  
  
        Map<Integer, Book> books = new HashMap<Integer, Book>();  
  
        books.put(1, book1);  
        books.put(2, book2);  
  
        System.out.println("图书信息如下: ");  
        for ( Integer id : books.keySet() ) {  
            System.out.print(id + ": ");  
            System.out.println( books.get(id) ); // 不需要类型转换  
        }  
    }  
}
```

■ key是 Integer 类型  
■ value是 Book 类型

运行结果

图书信息如下:

1: 1001-三国演义-35

2: 1002-水浒传-28

[【返回】](#)

## 2. 泛型类

添加<类型参数>，参数如有多个用逗号分隔

泛型类

```
class A<T> {  
    private T data;    //泛型参数作为属性类型  
    public A() {  
    }  
    public A( T data ) {  
        this.data = data;  
    }  
    public T getData() {    // 返回泛型参数  
        return data;  
    }  
}
```

泛型类一般用于类中的属性类型不确定的情况下

## 测试泛型类

在实例化泛型类时，需要指明泛型类中的类型参数

```
A<String> a1 = new A<String>("hello");
```

```
System.out.println( a1.getData() ); //hello
```

```
A<Integer> a2 = new A<Integer>(100);
```

```
System.out.println( a2.getData() ); // 100
```

```
System.out.println( a1.getClass() == a2.getClass() ); // true, 都是A泛型类
```

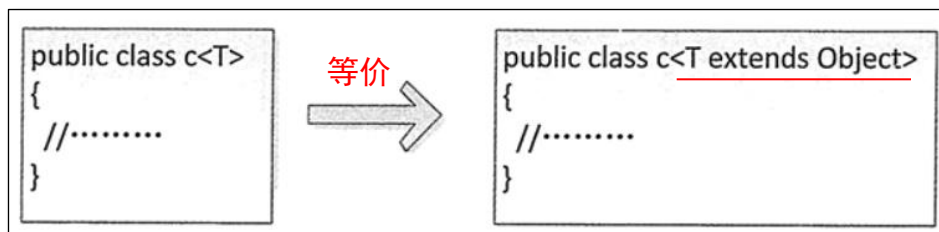
getClass(): 返回当前对象的类的信息（Java反射中的方法）

## 限制泛型可用类型

- 在 Java 中默认可以使用任何类型来实例化一个泛型类对象。当然也可以对泛型类实例的类型进行限制。
- 语法格式如下：

`class 类名称<T extends anyClass> { ... }`

- 使用泛型限制后，泛型类的类型必须实现或继承 anyClass 这个接口或类
- 无论anyClass是接口还是类，在进行泛型限制时必须使用extends关键字





## 示例

```
class ListClass<T extends List> {  
}  
  
class Test {  
    public static void main(String[] args) {  
        // 泛型参数使用ArrayList, OK  
        ListClass<ArrayList> lc1 = new ListClass<ArrayList>();  
  
        // 泛型参数使用LinkedList, OK  
        ListClass<LinkedList> lc2 = new ListClass<LinkedList>();  
  
        // 泛型参数使用HashMap, 报错: HasMap没有实现List接口  
        // ListClass<HashMap> lc3=new ListClass<HashMap>();  
    }  
}
```

## 多参数泛型类

泛型类

```
class Student<N, A, S> {  
    private N name; // 姓名  
    private A age; // 年龄  
    private S sex; // 性别  
  
    public Student(N name, A age, S sex) {  
        this.name = name;  
        this.age = age;  
        this.sex = sex;  
    }  
  
    public N getName() {  
        return name;  
    }  
  
    public void setName(N name) {  
        this.name = name;  
    }  
  
    public A getAge() {  
        return age;  
    }  
  
    public void setAge(A age) {  
        this.age = age;  
    }  
  
    public S getSex() {  
        return sex;  
    }  
  
    public void setSex(S sex) {  
        this.sex = sex;  
    }  
}
```

## 测试泛型类

```
Student<String, Integer, Character> stu =  
    new Student<String, Integer, Character>("小明", 19, '男');  
String name = stu.getName();  
Integer age = stu.getAge();  
Character sex = stu.getSex();  
System.out.println("姓名: " + name + ", 年龄: " + age + ", 性别: " + sex);
```


## 继承泛型类

```
class A<T> {  
}  
  
// 通常将父类的泛型类型保留 (T)  
// S是子类自己的类型参数  
class B<S,T> extends A<T>{  
}  
  
// 测试代码:  
A<String> b1 = new B<Integer, String>(); // OK  
A<String> b2 = new B<Integer, Integer>(); // 报错, 第二个必须是String类型
```

[【返回】](#)

### 3. 泛型方法

是否拥有泛型方法，与其所在的类是不是泛型没有关系



```
public static <T> void printArr( T[] arr ) {  
    for ( T e : arr ) {  
        System.out.printf("%s ", e);  
    }  
    System.out.println();  
}
```

1. 泛型方法声明都有一个参数声明部分，位于方法返回类型之前
2. 参数声明部分可以包含一个或多个类型参数，参数间用逗号隔开(如<T,E>)
3. 类型参数能被用来声明返回值类型，能作为泛型方法得到的实际参数类型的占位符
4. 注意类型参数只能代表引用型类型，不能是原始类型（如int,double,char的等）
5. 泛型方法体的声明和其他方法一样

## 泛型方法示例

```
public class Test {  
    public static <T> void printArr( T[] arr ) { // 泛型方法  
        for ( T e : arr ) {  
            System.out.printf("%s ", e);  
        }  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Integer[] a = { 1, 2, 3, 4, 5 }; // 不能使用int[]  
        String[] s = { "aaa","bbb" };  
        printArr(a);  
        printArr(s);  
    }  
}
```

运行结果

1	2	3	4	5
aaa	bbb			

[【返回】](#)

## 4. 通配符：使用?代替任意类型参数

```
class A<T> { // 代码同前，略 }  
  
public class Test {  
    public static void printData( A<?> data ) {  
        System.out.println("data :" + data.getData());  
    }  
  
    public static void main(String[] args) {  
        A<String> a1 = new A<String>("hello");  
        A<Integer> a2 = new A<Integer>(123);  
        printData(a1);    // hello  
        printData(a2);    // 123  
    }  
}
```

使用通配符代替任意类型参数

以支持多种  
不同类型的参数

## 给通配符设置限制

```
public static void printData( A<? extends 任意类> data ) {  
    ...  
}
```

限制形参只能是该类及其子类

```
public static void printData( A<? super 任意类> data ) {  
    ...  
}
```

限制形参只能是该类及其父类

[【返回】](#)



## 5. 泛型接口

泛型接口

```
interface IMy<T> {  
    T test();  
}
```

实现类

```
class B implements IMy< A<String> >{    // 传入 A<String>  
    @Override  
    public A<String> test() {  
        A<String> x = new A<String>("hello");  
        return x;  
    }  
}
```

测试:

```
IMy iMy = new B();  
System.out.println(((A<String>)iMy.test()).getData()); // hello
```

【完】