1、课程名称：Collections

2、知识点

2.1、本节预计讲解的知识点

1、 类集的主要作用
2、 Collection、List、Set 接口的作用及相关的子类
3、 Map 的作用及相关的子类
4、 Object 类的作用
5、 掌握泛型在整个 Java 类集的作用
6、 Collections 工具类
7、 equals、hashCode 与内存泄露分析

3、具体内容

3.1、类集设置的目的（重点）

类集：类集就是一系列的动态对象数组。

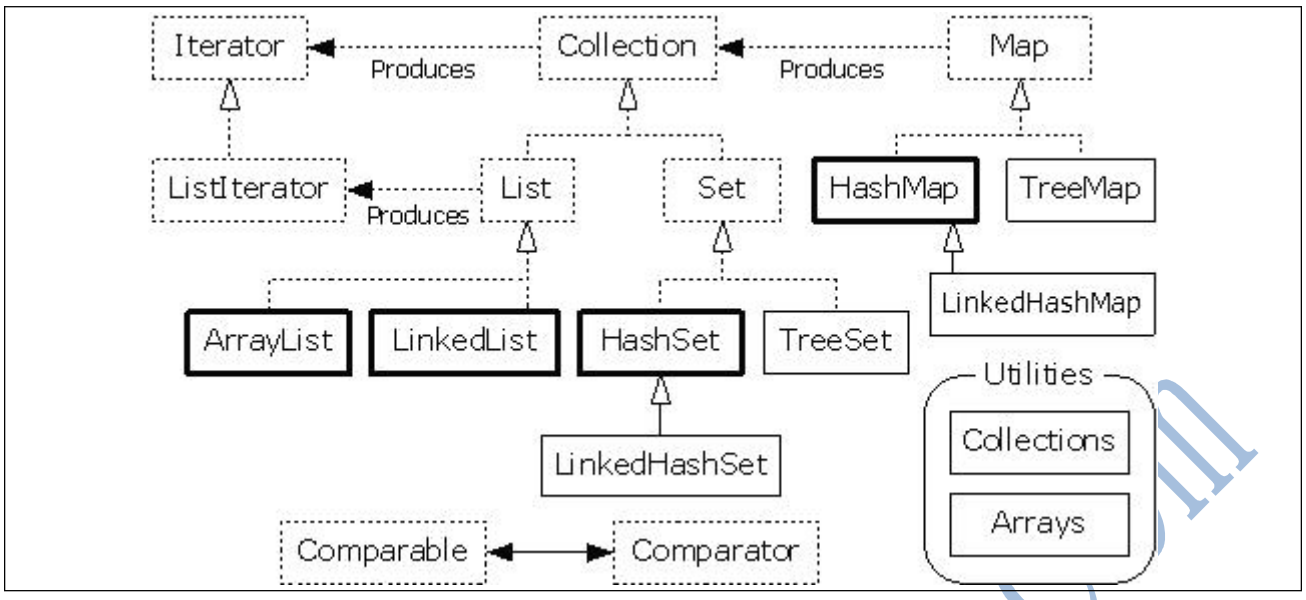
对象数组有那些问题？普通的对象数组的最大问题在于数组中的元素个数是固定的，不能动态的扩充大小，所以最早的时候可以通过链表实现一个动态对象数组。但是这样做毕竟太复杂了，所以在 Java 中为了方便用户操作各个数据结构，所以引入了类集的概念，有时候就可以把类集称为 java 对数据结构的实现。

在整个类集中的，这个概念是从 JDK 1.2（Java 2）之后才正式引入的，最早也提供了很多的操作类，但是并没有完整的提出类集的完整概念。

类集中最大的几个操作接口：Collection、Map、Iterator，这三个接口为以后要使用的最重点的接口。

所有的类集操作的接口或类都在 java.util 包中。

Java 类集结构图：



3.2、Collection 接口（重点）

Collection 接口是在整个 Java 类集中保存单值的最大操作父接口，里面每次操作的时候都只能保存一个对象的数据。此接口定义在 java.util 包中。

此接口定义如下：

```
public interface Collection<E> extends Iterable<E>
```

此接口使用了泛型技术，在 JDK 1.5 之后为了使类集操作的更加安全，所以引入了泛型。

此接口的常用方法如下所示。

No.	方法名称	类型	描述
1	**public boolean add(E e)**	**普通**	**向集合中插入一个元素**
2	public boolean addAll(Collection<? extends E> c)	普通	向集合中插入一组元素
3	public void clear()	普通	清空集合中的元素
4	public boolean contains(Object o)	普通	查找一个元素是否存在
5	public boolean containsAll(Collection<?> c)	普通	查找一组元素是否存在
6	public boolean isEmpty()	普通	判断集合是否为空
7	**public Iterator<E> iterator()**	**普通**	**为 Iterator 接口实例化**
8	public boolean remove(Object o)	普通	从集合中删除一个对象
9	boolean removeAll(Collection<?> c)	普通	从集合中删除一组对象
10	boolean retainAll(Collection<?> c)	普通	判断是否没有指定的集合
11	**public int size()**	**普通**	**求出集合中元素的个数**
12	public Object[] toArray()	普通	以对象数组的形式返回集合中的全部内容
13	<T> T[] toArray(T[] a)	普通	指定操作的泛型类型，并把内容返回
14	public boolean equals(Object o)	普通	从 Object 类中覆写而来
15	public int hashCode()	普通	从 Object 类中覆写而来

本接口中一共定义了 15 个方法，那么此接口的全部子类或子接口就将全部继承以上接口中的方法。

但是，在开发中不会直接使用 Collection 接口。而使用其操作的子接口：List、Set。

之所以有这样的明文规定，也是在 JDK 1.2 之后才有的。一开始在 EJB 中的最早模型中全部都是使用 Collection 操作

的，所以很早之前开发代码都是以 Collection 为准，但是后来为了更加清楚的区分，集合中是否允许有重复元素所以 SUN 在其开源项目 —— PetShop（宠物商店）中就开始推广 List 和 Set 的使用。

3.3、List 接口（重点）

在整个集合中 List 是 Collection 的子接口，里面的所有内容都是允许重复的。

List 子接口的定义：

```
public interface List<E> extends Collection<E>
```

此接口上依然使用了泛型技术。此接口对于 Collection 接口来讲有如下的扩充方法：

No.	方法名称	类型	描述
1	public void add(int index,E element)	普通	在指定位置处增加元素
2	boolean addAll(int index,Collection<? extends E> c)	普通	在指定位置处增加一组元素
3	**public E get(int index)**	**普通**	**根据索引位置取出每一个元素**
4	public int indexOf(Object o)	普通	根据对象查找指定的位置，找不到返回-1
5	public int lastIndexOf(Object o)	普通	从后面向前查找位置，找不到返回-1
6	**public ListIterator<E> listIterator()**	**普通**	**返回 ListIterator 接口的实例**
7	public ListIterator<E> listIterator(int index)	普通	返回从指定位置的 ListIterator 接口的实例
8	public E remove(int index)	普通	删除指定位置的内容
9	public E set(int index,E element)	普通	修改指定位置的内容
10	List<E> subList(int fromIndex,int toIndex)	普通	返回子集合

在 List 接口中有以上 10 个方法是对已有的 Collection 接口进行的扩充。

所以，证明，List 接口拥有比 Collection 接口更多的操作方法。

了解了 List 接口之后，那么该如何使用该接口呢？需要找到此接口的实现类，常用的实现类有如下几个：

- ArrayList（95%）、Vector（4%）、LinkedList（1%）

3.3.1、新的子类：ArrayList（重点）

ArrayList 是 List 接口的子类，此类的定义如下：

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

此类继承了 AbstractList 类。AbstractList 是 List 接口的子类。AbstractList 是个抽象类，适配器设计模式。

范例：增加及取得元素

```
package org.listdemo.arraylistdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>(); // 实例化List对象，并指定泛型类型
        all.add("hello "); // 增加内容，此方法从Collection接口继承而来
        all.add(0, "VINCE ");// 增加内容，此方法是List接口单独定义的
        all.add("world"); // 增加内容，此方法从Collection接口继承而来
        System.out.println(all); // 打印all对象调用toString()方法
    }
}
```

以上的操作向集合中增加了三个元素，其中在指定位置增加的操作是 List 接口单独定义的。随后进行输出的时候，实际上调用的是 toString()方法完成输出的。

可以发现，此时的对象数组并没有长度的限制，长度可以任意长，只要是内存够大就行。

范例：进一步操作

- 使用 remove()方法删除若干个元素，并且使用循环的方式输出。
- 根据指定位置取的内容的方法，只有 List 接口才有定义，其他的任何接口都没有任何的定义。

```
package org.listdemo.arraylistdemo;
import java.util.ArrayList;
import java.util.List;
public class ArrayListDemo02 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>(); // 实例化List对象，并指定泛型类型
        all.add("hello "); // 增加内容，此方法从Collection接口继承而来
        all.add(0, "VINCE ");// 增加内容，此方法是List接口单独定义的
        all.add("world"); // 增加内容，此方法从Collection接口继承而来
        all.remove(1); // 根据索引删除内容，此方法是List接口单独定义的
        all.remove("world");// 删除指定的对象
        System.out.print("集合中的内容是：");
        for (int x = 0; x < all.size(); x++) { // size()方法从Collection接口继承而来
            System.out.print(all.get(x) + "、"); // 此方法是List接口单独定义的
        }
    }
}
```

但是，这里需要注意的是，对于删除元素的操作，后面还会有更加清楚的讲解，此处只是简单的理解一下元素删除的基本操作即可。具体的原理可以暂时不进行深入掌握。

3.3.2、旧的子类：Vector（重点）

与 ArrayList 一样，Vector 本身也属于 List 接口的子类，此类的定义如下：

```
public class Vector<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

此类与 ArrayList 类一样，都是 AbstractList 的子类。所以，此时的操作只要是 List 接口的子类就都按照 List 进行操作。

```
package org.listdemo.vectordemo;
import java.util.List;
import java.util.Vector;
public class VectorDemo01 {
    public static void main(String[] args) {
        List<String> all = new Vector<String>(); // 实例化List对象，并指定泛型类型
        all.add("hello "); // 增加内容，此方法从Collection接口继承而来
        all.add(0, "VINCE ");// 增加内容，此方法是List接口单独定义的
        all.add("world"); // 增加内容，此方法从Collection接口继承而来
        all.remove(1); // 根据索引删除内容，此方法是List接口单独定义的
        all.remove("world");// 删除指定的对象
        System.out.print("集合中的内容是：");
        for (int x = 0; x < all.size(); x++) { // size()方法从Collection接口继承而来
```

```
            System.out.print(all.get(x) + "、"); // 此方法是List接口单独定义的
        }
    }
}
```

以上的操作结果与使用 ArrayList 本身并没有任何的区别。因为操作的时候是以接口为操作的标准。

但是 Vector 属于 Java 元老级的操作类，是最早的提供了动态对象数组的操作类，在 JDK 1.0 的时候就已经推出了此类的使用，只是后来在 JDK 1.2 之后引入了 Java 类集合框架。但是为了照顾很多已经习惯于使用 Vector 的用户，所以在 JDK 1.2 之后将 Vector 类进行了升级了，让其多实现了一个 List 接口，这样才将这个类继续保留了下来。

3.3.3、Vector 类和 ArrayList 类的区别（重点）

这两个类虽然都是 List 接口的子类，但是使用起来有如下的区别，为了方便大家笔试，列出此内容：

No.	区别点	ArrayList	Vector
1	时间	是新的类，是在 JDK 1.2 之后推出的	是旧的类是在 JDK 1.0 的时候就定义的
2	性能	性能较高，是采用了异步处理	性能较低，是采用了同步处理
3	输出	支持 Iterator、ListIterator 输出	除了支持 Iterator、ListIterator 输出，还支持 Enumeration 输出

3.3.4、链表操作类：LinkedList（理解）

此类的使用几率是非常低的，但是此类的定义如下：

```
public class LinkedList<E> extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, Serializable
```

此类继承了 AbstractList，所以是 List 的子类。但是此类也是 Queue 接口的子类，Queue 接口定义了如下的方法：

No.	方法名称	类型	描述
1	public boolean add(E e)	普通	增加元素，如果有容量限制，并且已满，则抛出异常
2	public E element()	普通	取得，但是不删除当前元素，如果对列为空则抛出异常
3	boolean offer(E e)	普通	添加，如果有容量限制，并且已满，只是无法添加，但不抛出异常，返回 false
4	E peek()	普通	取得头元素，但是不删除，如果队列为空，则返回 null
5	E poll()	普通	取得头元素，但是删除，如果队列为空，则返回 null
6	E remove()	普通	删除当前元素，如果对列为空则抛出异常

范例：验证 LinkedList 子类

```
import java.util.LinkedList;
import java.util.Queue;

public class TestDemo {

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

```
        Queue<String> queue = new LinkedList<String>();
        queue.add("A");
        queue.add("B");
        queue.add("C");
        int len=queue.size();//把queue的大小先取出来，否则每循环一次，移除一个元素，就少一个元素，那么queue.size()在变小，就不能循环queue.size()次了。
        for (int x = 0; x <len; x++) {
            System.out.println(queue.poll());
        }
        System.out.println(queue);
    }
}
```

3.4、Set 接口（重点）

Set 接口也是 Collection 的子接口，与 List 接口最大的不同在于，Set 接口里面的内容是不允许重复的。

Set 接口并没有对 Collection 接口进行扩充，基本上还是与 Collection 接口保持一致。因为此接口没有 List 接口中定义的 get(int index)方法，所以无法使用循环进行输出。

那么在此接口中有两个常用的子类：HashSet、TreeSet

3.4.1、散列存放：HashSet（重点）

既然 Set 接口并没有扩充任何的 Collection 接口中的内容，所以使用的方法全部都是 Collection 接口定义而来的。

HashSet 属于散列的存放类集，里面的内容是无序存放的。

范例：观察 HashSet

```
package org.listdemo.hashsetdemo;
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo01 {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>(); // 实例化Set接口对象
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        System.out.println(all);
    }
}
```

使用 HashSet 实例化的 Set 接口实例，本身属于无序的存放。

那么，现在思考一下？能不能通过循环的方式将 Set 接口中的内容输出呢？

是可以实现的，因为在 Collection 接口中定义了将集合变为对象数组进行输出。

```
package org.listdemo.hashsetdemo;
```

```
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo02 {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>(); // 实例化Set接口对象
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        Object obj[] = all.toArray(); // 将集合变为对象数组
        for (int x = 0; x < obj.length; x++) {
            System.out.print(obj[x] + "、");
        }
    }
}
```

但是，以上的操作不好，因为在操作的时候已经指定了操作的泛型类型，那么现在最好的做法是由泛型所指定的类型变为指定的数组。

所以只能使用以下的方法：<T> T[] toArray(T[] a)

```
package org.listdemo.hashsetdemo;
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo03 {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>(); // 实例化Set接口对象
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        String[] str = all.toArray(new String[] {});// 变为指定的泛型类型数组
        for (int x = 0; x < str.length; x++) {
            System.out.print(str[x] + "、");
        }
    }
}
```

下面再进一步验证 Set 接口中是不能有重复的内容的。

```
package org.listdemo.hashsetdemo;
import java.util.HashSet;
import java.util.Set;
public class HashSetDemo04 {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>(); // 实例化Set接口对象
        all.add("A");
        all.add("A"); // 重复元素
        all.add("A"); // 重复元素
```

```
        all.add("A"); // 重复元素
        all.add("A"); // 重复元素
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        System.out.println(all);
    }
}
```

以上字符串“A”设置了很多次，因为 Set 接口中是不能有任何的重复元素的，所以其最终结果只能有一个“A”。

3.4.2、排序的子类：TreeSet（重点）

与 HashSet 不同的是，TreeSet 本身属于排序的子类，此类的定义如下：

```
public class TreeSet<E> extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, Serializable
```

下面通过代码来观察其是如何进行排序的。

```
package org.listdemo.treesetdemo01;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetDemo01 {
    public static void main(String[] args) {
        Set<String> all = new TreeSet<String>(); // 实例化Set接口对象\
        all.add("D");
        all.add("X");
        all.add("A");
        System.out.println(all);
    }
}
```

虽然在增加元素的时候属于无序的操作，但是增加之后却可以为用户进行排序功能的实现。

3.4.3、排序的说明（重点）

既然 Set 接口的 TreeSet 类本身是允许排序，那么现在自定义一个类是否可以进行对象的排序呢？

范例：定义 Person 类

```
package org.listdemo.treesetdemo02;
public class Person {
    private String name;
    private int age;
    public Person() {
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
```

```
    }
    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 String toString() {
        return "姓名: " + this.name + ", 年龄: " + this.age;
    }
}
```

下面定义一个 TreeSet 集合，向里面增加若干个 Person 对象。

```
package org.listdemo.treesetdemo02;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetPersonDemo01 {
    public static void main(String[] args) {
        Set<Person> all = new TreeSet<Person>();
        all.add(new Person("张三", 10));
        all.add(new Person("李四", 10));
        all.add(new Person("王五", 11));
        all.add(new Person("赵六", 12));
        all.add(new Person("孙七", 13));
        System.out.println(all);
    }
}
```

执行以上的操作代码之后，发现出现了如下的错误提示：

```
Exception in thread "main" java.lang.ClassCastException:
org.vince.listdemo.treesetdemo02.Person cannot be cast to java.lang.Comparable
    at java.util.TreeMap.put(Unknown Source)
    at java.util.TreeSet.add(Unknown Source)
    at
org.vince.listdemo.treesetdemo02.TreeSetPersonDemo01.main(TreeSetPersonDemo01.java:11)
```

此时的提示是：Person 类不能向 Comparable 接口转型的问题？

所以，证明，如果现在要是想进行排序的话，则必须在 Person 类中实现 Comparable 接口。

```
package org.listdemo.treesetdemo03;
public class Person implements Comparable<Person> {
    private String name;
    private int age;
```

```
    public int compareTo(Person per) {
        if (this.age > per.age) {
            return 1;
        } else if (this.age < per.age) {
            return -1;
        } else {
            return 0;
        }
    }
    public Person() {
    }
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    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 String toString() {
        return "姓名: " + this.name + ", 年龄: " + this.age;
    }
}
```

那么此时再次使用之前的代码运行程序。程序的执行结果如下：

```
[姓名：张三，年龄：10，姓名：王五，年龄：11，姓名：赵六，年龄：12，姓名：孙七，年龄：13]
```

从以上的结果中可以发现，李四没有了。因为李四的年龄和张三的年龄是一样的，所以会被认为是同一个对象。则此时必须修改 Person 类，如果假设年龄相等的话，按字符串进行排序。

```
    public int compareTo(Person per) {
        if (this.age > per.age) {
            return 1;
        } else if (this.age < per.age) {
            return -1;
        } else {
            return this.name.compareTo(per.name);
        }
    }
```

此时，可以发现李四出现了，如果加入了同一个人的信息的话，则会认为是重复元素，所以无法继续加入。

3.4.4、关于重复元素的说明（重点）

之前使用 Comparable 完成的对于重复元素的判断，那么 Set 接口定义的时候本身就是不允许重复元素的，那么证明如果现在真的是有重复元素的话，使用 HashSet 也同样可以进行区分。

```
package org.listdemo.treesetdemo04;
import java.util.HashSet;
import java.util.Set;
public class HashSetPersonDemo01 {
    public static void main(String[] args) {
        Set<Person> all = new HashSet<Person>();
        all.add(new Person("张三", 10));
        all.add(new Person("李四", 10));
        all.add(new Person("李四", 10));
        all.add(new Person("王五", 11));
        all.add(new Person("赵六", 12));
        all.add(new Person("孙七", 13));
        System.out.println(all);
    }
}
```

此时发现，并没有去掉所谓的重复元素，也就是说之前的操作并不是真正的重复元素的判断，而是通过 Comparable 接口间接完成的。

如果要想判断两个对象是否相等，则必须使用 Object 类中的 equals()方法。

从最正规的来讲，如果要想判断两个对象是否相等，则有两种方法可以完成：

- 第一种判断两个对象的编码是否一致，这个方法需要通过 hashCode()完成，即：每个对象有唯一的编码
- 还需要进一步验证对象中的每个属性是否相等，需要通过 equals()完成。

所以此时需要覆写 Object 类中的 hashCode()方法，此方法表示一个唯一的编码，一般是通过公式计算出来的。

```
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (!(obj instanceof Person)) {
            return false;
        }
        Person per = (Person) obj;
        if (per.name.equals(this.name) && per.age == this.age) {
            return true;
        } else {
            return false;
        }
    }
    public int hashCode() {
        return this.name.hashCode() * this.age;
    }
```

发现，此时已经不存在重复元素了，所以如果要想去掉重复元素需要依靠 hashCode()和 equals()方法共同完成。

小结：

关于 TreeSet 的排序实现，如果是集合中对象是自定义的或者说其他系统定义的类没有实现 Comparable 接口，则不能实现 TreeSet 的排序，会报类型转换（转向 Comparable 接口）错误。换句话说要添加到 TreeSet 集合中的对象的类型必须实现了 Comparable 接口。

不过 TreeSet 的集合因为借用了 Comparable 接口，同时可以去除重复值，而 HashSet 虽然是 Set 接口子类，但是对于没有复写 Object 的 equals 和 hashCode 方法的对象，加入了 HashSet 集合中也是不能去掉重复值的。

3.5、集合输出（重点）

已经学习过了基本的集合操作，那么对于集合的输出本身也是有多种形式的，可以使用如下的几种方式：

- Iterator 迭代输出（90%）、ListIterator（5%）、Enumeration（1%）、foreach（4%）

但是在讲解输出的时候一定要记住以下的原则：“**只要是碰到了集合，则输出的时候想都不想就使用 Iterator 进行输出。**”

3.5.1、Iterator（绝对重点）

Iterator 属于迭代输出，基本的操作原理：是不断的判断是否有下一个元素，有的话，则直接输出。

此接口定义如下：

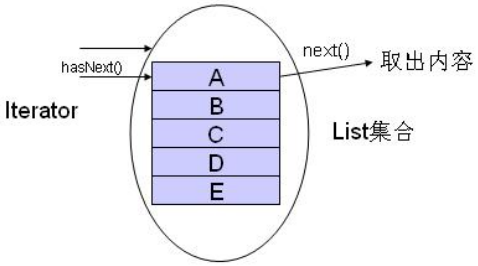
```
public interface Iterator<E>
```

要想使用此接口，则必须使用 Collection 接口，在 Collection 接口中规定了一个 iterator()方法，可以用于为 Iterator 接口进行实例化操作。

此接口规定了以下的三个方法：

No.	方法名称	类型	描述
1	boolean hasNext()	普通	是否有下一个元素
2	E next()	普通	取出内容
3	void remove()	普通	删除当前内容

通过 Collection 接口为其进行实例化之后，一定要记住，Iterator 中的操作指针是在第一条元素之上，当调用 hasNext()方法的时候，指针才向下继续移动。



范例：观察 Iterator 输出

```
package org.listdemo.iteratordemo;
import java.util.ArrayList;
import java.util.Collection;
```

```
import java.util.Iterator;
public class IteratorDemo01 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>();
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        Iterator<String> iter = all.iterator();
        while (iter.hasNext()) {// 判断是否有下一个元素
            String str = iter.next(); // 取出当前元素
            System.out.print(str + "、");
        }
    }
}
```

以上的操作是 Iterator 接口使用最多的形式，也是一个标准的输出形式。

但是在使用 Iterator 输出的时候有一点必须注意，在进行迭代输出的时候如果要想删除当前元素，则只能使用 Iterator 接口中的 remove()方法，而不能使用集合中的 remove()方法。否则将出现未知的错误。

```
package org.listdemo.iteratordemo;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
public class IteratorDemo02 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>();
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        Iterator<String> iter = all.iterator();
        while (iter.hasNext()) {// 判断是否有下一个元素
            String str = iter.next(); // 取出当前元素
            if (str.equals("C")) {
                all.remove(str); // 错误的，调用了集合中的删除
            } else {
                System.out.print(str + "、");
            }
        }
    }
}
```

此时出现了错误，因为原本的要输出的集合的内容被破坏掉了。

```
        Iterator<String> iter = all.iterator();
        while (iter.hasNext()) {// 判断是否有下一个元素
            String str = iter.next(); // 取出当前元素
```

```
            if (str.equals("C")) {
                iter.remove(); // 错误的，调用了集合中的删除
            } else {
                System.out.print(str + "、");
            }
        }
```

但是，从实际的开发角度看，元素的删除操作出现的几率是很小的，基本上可以忽略，即：集合中很少有删除元素的操作。

Iterator 接口本身可以完成输出的功能，但是此接口只能进行由前向后的单向输出。如果要想进行双向输出，则必须使用其子接口 —— ListIterator。

3.5.2、ListIterator（理解）

ListIterator 是可以进行双向输出的迭代接口，此接口定义如下：

```
public interface ListIterator<E>
extends Iterator<E>
```

此接口是 Iterator 的子接口，此接口中定义了以下的操作方法：

No.	方法名称	类型	描述
1	void add(E e)	普通	增加元素
2	boolean hasPrevious()	普通	判断是否有前一个元素
3	E previous()	普通	取出前一个元素
4	void set(E e)	普通	修改元素的内容
5	int previousIndex()	普通	前一个索引位置
6	int nextIndex()	普通	下一个索引位置

但是如果要想使用 ListIterator 接口，则必须依靠 List 接口进行实例化。

List 接口中定义了以下的方法：ListIterator<E> listIterator()

范例：使用 ListIterator 输出

```
package org.listdemo.listiteratordemo;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorDemo01 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        ListIterator<String> iter = all.listIterator();
        System.out.print("从前向后输出：");
        while (iter.hasNext()) {
            System.out.print(iter.next() + "、");
        }
        System.out.print("\n从后向前输出：");
```

```
        while (iter.hasPrevious()) {
            System.out.print(iter.previous() + "、");
        }
    }
}
```

但是，此处有一点需要注意的是，如果要想进行由后向前的输出，则首先必须先进行由前向后的输出。

但是，此接口一般使用较少。

3.5.3、废弃的接口：Enumeration（了解）

Enumeration 是一个非常古老的输出接口，其也是一个元老级的输出接口，最早的动态数组使用 Vector 完成，那么只要是使用了 Vector 则就必须使用 Enumeration 进行输出。

此接口定义如下：

```
public interface Enumeration<E>
```

在 JDK 1.5 之后，此接口实际上也已经加入了泛型操作。此接口常用方法如下：

No.	方法名称	类型	描述
1	boolean hasMoreElements()	普通	判断是否有下一个元素
2	E nextElement()	普通	取出当前元素

但是，与 Iterator 不同的是，如果要想使用 Enumeration 输出的话，则还必须使用 Vector 类完成，在类中定义了如下方法：public Enumeration<E> elements()

范例：验证 Enumeration 接口

```
package org.listdemo.enumerationdemo;
import java.util.Enumeration;
import java.util.Vector;
public class EnumerationDemo01 {
    public static void main(String[] args) {
        Vector<String> v = new Vector<String>();
        v.add("A");
        v.add("B");
        v.add("C");
        Enumeration<String> enu = v.elements();
        while (enu.hasMoreElements()) {
            System.out.println(enu.nextElement());
        }
    }
}
```

需要注意的是，在大部分的情况下，此接口都不再使用了，但是对于一些古老的类库，本身依然要使用此接口进行操作，所以此接口一定要掌握。

所有的代码最好可以用记事本独立写出。

3.5.4、新的支持：foreach（理解）

foreach 可以用来输出数组的内容，那么也可以输出集合中的内容。

```
package org.vince.listdemo.foreachdemo;
import java.util.ArrayList;
import java.util.Collection;
public class ForeachDemo01 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>();
        all.add("A");
        all.add("B");
        all.add("C");
        all.add("D");
        all.add("E");
        for (String str : all) {
            System.out.println(str) ;
        }
    }
}
```

在使用 foreach 输出的时候一定要注意的是，里面的操作泛型要指定具体的类型，这样在输出的时候才会更加有针对性。

3.6、Map 接口（重点）

以上的 Collection 中，每次操作的都是一个对象，如果现在假设要操作一对对象，则就必须使用 Map 了，类似于以下一种情况：

- 张三　　123456
- 李四　　234567

那么保存以上信息的时候使用 Collection 就不那么方便，所以要使用 Map 接口。里面的所有内容都按照 key→value 的形式保存，也称为二元偶对象。

此接口定义如下：

```
public interface Map<K,V>
```

此接口与 Collection 接口没有任何的关系，是第二大的集合操作接口。此接口常用方法如下：

No.	方法名称	类型	描述
1	void clear()	普通	清空 Map 集合中的内容
2	boolean containsKey(Object key)	普通	判断集合中是否存在指定的 key
3	boolean containsValue(Object value)	普通	判断集合中是否存在指定的 value
4	**Set<Map.Entry<K,V>> entrySet()**	**普通**	**将 Map 接口变为 Set 集合**
5	**V get(Object key)**	**普通**	**根据 key 找到其对应的 value**
6	boolean isEmpty()	普通	判断是否为空
7	**Set<K> keySet()**	**普通**	**将全部的 key 变为 Set 集合**
8	**Collection<V> values()**	**普通**	**将全部的 value 变为 Collection 集合**
9	**V put(K key,V value)**	**普通**	**向集合中增加内容**
10	void putAll(Map<? extends K,? extends V> m)	普通	增加一组集合
11	V remove(Object key)	普通	根据 key 删除内容

Map 本身是一个接口，所以一般会使用以下的几个子类：HashMap、TreeMap、Hashtable

3.6.1、新的子类：HashMap（重点）

HashMap 是 Map 的子类，此类的定义如下：

```
public class HashMap<K,V> extends AbstractMap<K,V>
implements Map<K,V>, Cloneable, Serializable
```

此类继承了 AbstractMap 类，同时可以被克隆，可以被序列化下来。

范例：向集合中增加内容

```
package org.listdemo.hashmapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo01 {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(1, "张三A");
        map.put(1, "张三B"); // 新的内容替换掉旧的内容
        map.put(2, "李四");
        map.put(3, "王五");
        String val = map.get(6);
        System.out.println(val);
    }
}
```

以上的操作是 Map 接口在开发中最基本的操作过程，根据指定的 key 找到内容，如果没有找到，则返回 null，找到了则返回具体的内容。

范例：得到全部的 key 或 value

```
package org.listdemo.hashmapdemo;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapDemo02 {
    public static void main(String[] args) {
        Map<Integer, String> map = new HashMap<Integer, String>();
        map.put(1, "张三A");
        map.put(2, "李四");
        map.put(3, "王五");
        Set<Integer> set = map.keySet(); // 得到全部的key
        Collection<String> value = map.values(); // 得到全部的value
        Iterator<Integer> iter1 = set.iterator();
        Iterator<String> iter2 = value.iterator();
        System.out.print("全部的key: ");
        while (iter1.hasNext()) {
            System.out.print(iter1.next() + "、");
        }
        System.out.print("\n全部的value: ");
```

```
        while (iter2.hasNext()) {
            System.out.print(iter2.next() + "、");
        }
    }
}
```

既然可以取得全部的 key，那么下面就可以对以上的操作进行扩充，循环输出 Map 中的全部内容。

```
package org.listdemo.hashmapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashMapDemo03 {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<String, String>();
        map.put("ZS", "张三");
        map.put("LS", "李四");
        map.put("WW", "王五");
        map.put("ZL", "赵六");
        map.put("SQ", "孙七");
        Set<String> set = map.keySet(); // 得到全部的key
        Iterator<String> iter = set.iterator();
        while (iter.hasNext()) {
            String i = iter.next(); // 得到key
            System.out.println(i + " --:> " + map.get(i));
        }
    }
}
```

HashMap 本身是属于无序存放的。

3.6.2、旧的子类：Hashtable（重点）

Hashtable 是一个最早的 key→value 的操作类，本身是在 JDK 1.0 的时候推出的。其基本操作与 HashMap 是类似的。

```
package org.listdemo.hashtabledemo;
import java.util.Hashtable;
import java.util.Map;
public class HashtableDemo01 {
    public static void main(String[] args) {
        Map<String, Integer> numbers = new Hashtable<String, Integer>();
        numbers.put("one", 1);
        numbers.put("two", 2);
        numbers.put("three", 3);
        Integer n = numbers.get("two");
        if (n != null) {
            System.out.println("two = " + n);
        }
```

```
    }
}
```

操作的时候，可以发现与 HashMap 基本上没有什么区别，而且本身都是以 Map 为操作标准的，所以操作的结果形式都一样。但是 Hashtable 中是不能向集合中插入 null 值的。

3.6.3、HashMap 与 Hashtable 的区别（重点）

在整个集合中除了 ArrayList 和 Vector 的区别之外，另外一个最重要的区别就是 HashMap 与 Hashtable 的区别。

No.	区别点	HashMap	Hastable
1	推出时间	JDK 1.2 之后推出的，新的操作类	JDK 1.0 时推出的，旧的操作类
2	性能	异步处理，性能较高	同步处理，性能较低
3	null	允许设置为 null	不允许设置，否则将出现空指向异常

3.6.4、排序的子类：TreeMap（理解）

TreeMap 子类是允许 key 进行排序的操作子类，其本身在操作的时候将按照 key 进行排序，另外，key 中的内容可以为任意的对象，但是要求对象所在的类必须实现 Comparable 接口。

```
package org.listdemo.treemapdemo;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
public class TreeMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = new TreeMap<String, String>();
        map.put("ZS", "张三");
        map.put("LS", "李四");
        map.put("WW", "王五");
        map.put("ZL", "赵六");
        map.put("SQ", "孙七");
        Set<String> set = map.keySet(); // 得到全部的key
        Iterator<String> iter = set.iterator();
        while (iter.hasNext()) {
            String i = iter.next(); // 得到key
            System.out.println(i + " --:> " + map.get(i));
        }
    }
}
```

此时的结果已经排序成功了，但是从一般的开发角度来看，在使用 Map 接口的时候并不关心其是否排序，所以此类只需要知道其特点即可。

3.6.5、关于 Map 集合的输出

在 Collection 接口中，可以使用 iterator()方法为 Iterator 接口实例化，并进行输出操作，但是在 Map 接口中并没有此方法的定义，所以 Map 接口本身是不能直接使用 Iterator 进行输出的。

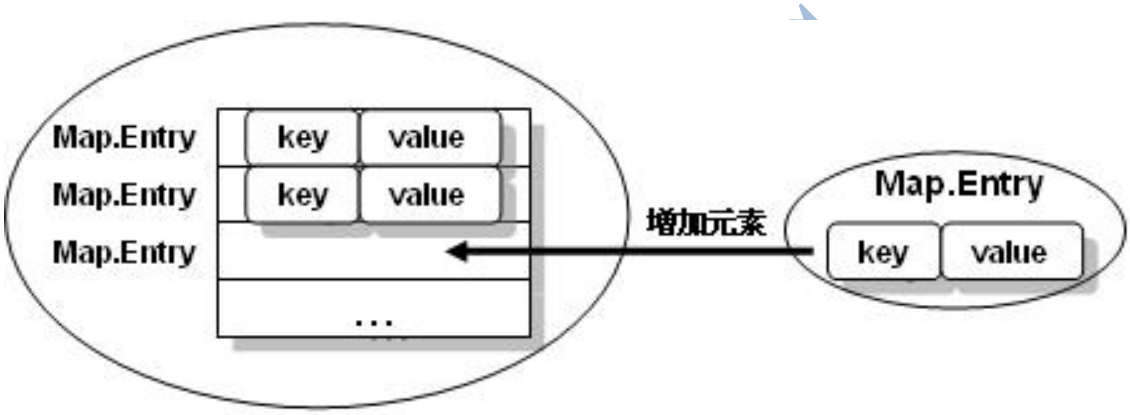
因为 Map 接口中存放的每一个内容都是一对值，而使用 Iterator 接口输出的时候，每次取出的都实际上是一个完整的对象。如果此时非要使用 Iterator 进行输出的话，则可以按照如下的步骤进行：

1、 使用 Map 接口中的 entrySet()方法将 Map 接口的全部内容变为 Set 集合
2、 可以使用 Set 接口中定义的 iterator()方法为 Iterator 接口进行实例化
3、 之后使用 Iterator 接口进行迭代输出，每一次的迭代都可以取得一个 Map.Entry 的实例
4、 通过 Map.Entry 进行 key 和 value 的分离

那么，到底什么是 Map.Entry 呢？

Map.Entry 本身是一个接口。此接口是定义在 Map 接口内部的，是 Map 的内部接口。此内部接口使用 static 进行定义，所以此接口将成为外部接口。

实际上来讲，对于每一个存放到 Map 集合中的 key 和 value 都是将其变为了 Map.Entry 并且将 Map.Entry 保存在了 Map 集合之中。



在 Map.Entry 接口中以下的方法最为常用：

No.	方法名称	类型	描述
1	K getKey()	普通	得到 key
2	V getValue()	普通	得到 value

范例：使用 Iterator 输出 Map 接口

```
package org.listdemo.mapoutdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class MapOutDemo01 {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<String, String>();
        map.put("ZS", "张三");
        map.put("LS", "李四");
        map.put("WW", "王五");
        map.put("ZL", "赵六");
        map.put("SQ", "孙七");
        Set<Map.Entry<String, String>> set = map.entrySet();// 变为Set实例
        Iterator<Map.Entry<String, String>> iter = set.iterator();
```

```
        while (iter.hasNext()) {
            Map.Entry<String, String> me = iter.next();
            System.out.println(me.getKey() + " --> " + me.getValue());
        }
    }
}
```

以上的代码一定要记住，Map 集合中每一个元素都是 Map.Entry 的实例，只有通过 Map.Entry 才能进行 key 和 value 的分离操作。

除了以上的做法之外，在 JDK 1.5 之后也可以使用 foreach 完成同样的输出，只是这样的操作基本上不使用。

```
package org.listdemo.mapoutdemo;
import java.util.HashMap;
import java.util.Map;
public class MapOutDemo02 {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<String, String>();
        map.put("ZS", "张三");
        map.put("LS", "李四");
        map.put("WW", "王五");
        map.put("ZL", "赵六");
        map.put("SQ", "孙七");
        for (Map.Entry<String, String> me : map.entrySet()) {
            System.out.println(me.getKey() + " --> " + me.getValue());
        }
    }
}
```

3.8、两种关系（理解）

使用类集，除了可以清楚的表示出动态数组的概念及各个数据结构的操作之外，也可以表示出以下的两种关系。

3.8.1、第一种关系：一对多关系

一个学校有多个学生，是一个典型的一对多的关系。

范例：定义学生类，一个学生属于一个学校

```
package org.listdemo.casedemo01;
public class Student {
    private String name;
    private int age;
    private School school;
    public Student(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public School getSchool() {
```

```
        return school;
    }
    public void setSchool(School school) {
        this.school = school;
    }
    public String toString() {
        return "学生信息" + "\n" + "\t|- 姓名: " + this.name + "\n" + "\t|- 年龄: "
                + this.age;
    }
}
```

范例：定义学校类，一个学校有多个学生

```
package org.listdemo.casedemo01;
import java.util.ArrayList;
import java.util.List;
public class School {
    private String name;
    private List<Student> allStudents = null;
    public School() {
        this.allStudents = new ArrayList<Student>();
    }
    public School(String name) {
        this();
        this.name = name;
    }
    public List<Student> getAllStudents() {
        return allStudents;
    }
    public String toString() {
        return "学校信息: " + "\n" + "\t|- 学校名称: " + this.name;
    }
}
```

之后在主方法处建立以上两者的关系。

```
package org.listdemo.casedemo01;
import java.util.Iterator;
public class TestDemo01 {
    public static void main(String[] args) {
        Student stu1 = new Student("张三", 10);
        Student stu2 = new Student("李四", 11);
        Student stu3 = new Student("王五", 12);
        School sch = new School("VINCE JAVA");
        sch.getAllStudents().add(stu1); // 一个学校有多个学生
        stu1.setSchool(sch);// 一个学生属于一个学校
        sch.getAllStudents().add(stu2); // 一个学校有多个学生
        stu2.setSchool(sch);// 一个学生属于一个学校
        sch.getAllStudents().add(stu3); // 一个学校有多个学生
        stu3.setSchool(sch);// 一个学生属于一个学校
```

```
        System.out.println(sch);
        Iterator<Student> iter = sch.getAllStudents().iterator();
        while (iter.hasNext()) {
            System.out.println(iter.next());
        }
        System.out.println(stu1.getSchool());
    }
}
```

此时，就已经完成了一个一对多的关系。

了解之后，下面继续研究。

3.8.2、第二种关系：多对多关系

一个学生可以选择多门课程，一门课程允许有多个学生参加。

范例：定义学生类，一个学生可以选择多门课程

```
package org.listdemo.casedemo02;
import java.util.ArrayList;
import java.util.List;
public class Student {
    private String name;
    private int age;
    private List<Course> allCourses;
    public Student() {
        this.allCourses = new ArrayList<Course>();
    }
    public Student(String name, int age) {
        this();
        this.name = name;
        this.age = age;
    }
    public List<Course> getAllCourses() {
        return allCourses;
    }
    public String toString() {
        return "学生信息" + "\n" + "\t|- 姓名: " + this.name + "\n" + "\t|- 年龄: "
                + this.age;
    }
}
```

范例：定义课程类，一门课程可以有多个学生参加

```
package org.listdemo.casedemo02;
import java.util.ArrayList;
import java.util.List;
public class Course {
    private String name;
    private int credit;
```

```
    private List<Student> allStudents = null;
    public Course() {
        this.allStudents = new ArrayList<Student>();
    }
    public Course(String name, int credit) {
        this();
        this.credit = credit;
        this.name = name;
    }
    public List<Student> getAllStudents() {
        return allStudents;
    }
    public String toString() {
        return "课程信息: " + "\n" + "\t|- 课程名称: " + this.name + "\n" + "\t|- 课程学分:
" + this.credit;
    }
}
```

下面同样在主方法处设置关系，但是必须注意的是，这个时候设置的关系也应该是双向操作完成的。

```
package org.listdemo.casedemo02;
import java.util.Iterator;
public class TestDemo02 {
    public static void main(String[] args) {
        Student stu1 = new Student("张三", 10);
        Student stu2 = new Student("李四", 11);
        Student stu3 = new Student("王五", 12);
        Student stu4 = new Student("赵六", 15);
        Student stu5 = new Student("孙七", 13);
        Course c1 = new Course("Oracle", 5);
        Course c2 = new Course("Java SE基础课程", 10);
        c1.getAllStudents().add(stu1); // 参加第一门课程
        c1.getAllStudents().add(stu2); // 参加第一门课程
        stu1.getAllCourses().add(c1); // 学生选择课程
        stu2.getAllCourses().add(c1); // 学生选择课程
        c2.getAllStudents().add(stu1); // 参加第二门课程
        c2.getAllStudents().add(stu2); // 参加第二门课程
        c2.getAllStudents().add(stu3); // 参加第二门课程
        c2.getAllStudents().add(stu4); // 参加第二门课程
        c2.getAllStudents().add(stu5); // 参加第二门课程
        stu1.getAllCourses().add(c2); // 学生选择课程
        stu2.getAllCourses().add(c2); // 学生选择课程
        stu3.getAllCourses().add(c2); // 学生选择课程
        stu4.getAllCourses().add(c2); // 学生选择课程
        stu5.getAllCourses().add(c2); // 学生选择课程
        System.out.println(c2);
        Iterator<Student> iter = c2.getAllStudents().iterator();
        while (iter.hasNext()) {
```

```
            System.out.println(iter.next());
        }
        System.out.println("----------------------------");
        System.out.println(stu1);
        Iterator<Course> iters = stu1.getAllCourses().iterator() ;
        while (iters.hasNext()) {
            System.out.println(iters.next());
        }
    }
}
```

3.9、Collections 类（理解）

Collections 实际上是一个集合的操作类，此类的定义如下：

```
public class Collections extends Object
```

这个类与 Collection 接口没有任何的关系。是一个单独存在的类。

范例：返回空的 List 集合

```
package org.listdemo.collectionsdemo;
import java.util.Collections;
import java.util.List;
public class CollectionsDemo01 {
    public static void main(String[] args) {
        List<String> all = Collections.emptyList() ;// 空的集合
        all.add("A") ;
    }
}
```

使用 Collections 类返回的空的集合对象，本身是不支持任何的修改操作的，因为所有的方法都没实现。

范例：使用 Collections 进行增加元素的操作

```
package org.listdemo.collectionsdemo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class CollectionsDemo02 {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        Collections.addAll(all, "A", "B", "C");// 向集合增加元素
        System.out.println(all);
    }
}
```

但是，从实际考虑，使用此类操作并不是很方便，最好的做法就是使用各个接口的直接操作的方法完成。此类只是一个集合的操作类。

3.10、分析 equals、hashCode 与内存泄露（理解）

equals 的作用：比较两个对象的地址值是否相等

equals()方法在 object 类中定义如下：

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

但是我们必需清楚，当 String 、Math、还有 Integer、Double。。。。等这些封装类在使用 equals()方法时，已经覆盖了 object 类的 equals（）方法，不再是地址的比较而是内容的比较。

我们还应该注意，Java 语言对 equals()的要求如下，这些要求是必须遵循的：

1. 对称性：如果 x.equals(y)返回是“true”，那么 y.equals(x)也应该返回是“true”。
2. 反射性：x.equals(x)必须返回是“true”。
3. 类推性：如果 x.equals(y)返回是“true”，而且 y.equals(z)返回是“true”，那么 z.equals(x)也应该返回是“true”。
4. 还有一致性：如果 x.equals(y)返回是“true”，只要 x 和 y 内容一直不变，不管你重复 x.equals(y)多少次，返回都是“true”。
5. 任何情况下，x.equals(null)，永远返回是“false”；x.equals(和 x 不同类型的对象)永远返回是“false”。

以上这五点是重写 equals()方法时，必须遵守的准则，如果违反会出现意想不到的结果，请大家一定要遵守。

hashcode() 方法，在 object 类中定义如下：

```
public native int hashCode();
```

说明是一个本地方法，它的实现是根据本地机器相关的。当然我们可以在自己写的类中覆盖 hashcode()方法，比如 String、Integer、Double。。。。等等这些类都是覆盖了 hashcode()方法的。

java.lnag.Object 中对 hashCode 的约定（很重要）：

1. 在一个应用程序执行期间，如果一个对象的 equals 方法做比较所用到的信息没有被修改的话，则对该对象调用 hashCode 方法多次，它必须始终如一地返回同一个整数。
2. 如果两个对象根据 equals(Object o)方法是相等的，则调用这两个对象中任一对象的 hashCode 方法必须产生相同的整数结果。
3. 如果两个对象根据 equals(Object o)方法是不相等的，则调用这两个对象中任一个对象的 hashCode 方法，不要求产生不同的整数结果。但如果能不同，则可能提高散列表的性能。

在 java 的集合中，判断两个对象是否相等的规则是：

（1）判断两个对象的 hashCode 是否相等

如果不相等，认为两个对象也不相等，完毕

如果相等，转入 2

（这一点只是为了提高存储效率而要求的，其实理论上没有也可以，但如果没有，实际使用时效率会大大降低，所以我们这里将其做为必需的。后面会重点讲到这个问题。）

（2）判断两个对象用 equals 运算是否相等

如果不相等，认为两个对象也不相等

如果相等，认为两个对象相等（equals()是判断两个对象是否相等的关键）

提示贴：

当一个对象被存进 HashSet 集合后，就不能修改这个对象中的那些参与计算的哈希值的字段了，否则，对象被修改后的哈希值与最初存储进 HashSet 集合中时的哈希值就不同了，在这种情况下，即使在 contains 方法使用该对象的当前引用作为的参数去 HashSet 集合中检索对象，也将返回找不到对象的结果，这也会导致无法从 HashSet 集合中删除当前对象，从而

造成内存泄露。

4、总结

1、 类集就是一个动态的对象数组，可以向集合中加入任意多的内容。
2、 List 接口中是允许有重复元素的，Set 接口中是不允许有重复元素。
3、 所有的重复元素依靠 hashCode()和 equals 进行区分
4、 List 接口的常用子类：ArrayList、Vector
5、 Set 接口的常用子类：HashSet、TreeSet
6、 TreeSet 是可以排序，一个类的对象依靠 Comparable 接口排序
7、 Map 接口中允许存放一对内容，key → value
8、 Map 接口的子类：HashMap、Hashtable、TreeMap
9、 Map 使用 Iterator 输出的详细步骤

5、作业

1、 用容器模拟数据库，进行增删改查操作
用 ArrayList 实现对用户（User.java）的增、删、改、查
详细设计：
1) User.java（用户类）

```
/** 编号 */
private int id;

/** 姓名 */
private String name;

/** 年龄 */
private int age;
```

2) UserManage.java

```
/** 用户管理类 */
public class UserManage {

    /** 用于保存用户信息的容器 */
    private ArrayList userList = new ArrayList();

    /** 添加用户 */
    public void addUser(User user) {

    /** 删除用户 */
    public void delUser(User user) {

    /** 显示用户 */
    public void showUsers() {

    /** 修改用户 */
    public void updUser(User user) {

    /**
     * 在main(String[])方法中实现如下步骤:
     * 创建3个User的实例对象
     * 调用addUser(User)方法将3个实例对象添加到容器中
     * 调用updUser(User)方法修改容器中1个对象
     * 调用delUser(User)方法删除容器中1个对象
     */
    public static void main(String[] args) {

}
```