1、课程名称：Java 类集框架

2、知识点

2.1、上次课程的主要知识点

关于类的设计并没有一个严格的准则，包括设计模式也并不是 100%都是通用的，因为越复杂的设计，操作的复杂度就越高，类的设计的原则：主方法（客户端）的操作代码越简单越好，而且每一个类只完成每一个类的具体功能，如果多个类之间要进行互操作的话，可以通过接口进行解耦合的操作。

1、 StringBuffer 类与 String 类的区别在于 StringBuffer 的内容可以改变，而 String 类的内容不可改变。

2、 Date 表示的是一个日期，如果要想进行完整的格式化显示依靠 SimpleDateFomat，而且一定要记住的是，使用此类可以完成字符串和日期类型的相互转换。

3、 大数字操作使用 BigInteger、BigDecimal、如果要进行准确的四舍五入操作的话，可以通过 BigDecimal 完成，或者是通过 String 类的 format()方法。

4、 两种比较器：Comparable、Comparator

5、 File 类可以直接操作文件本身：crateNewFile()、delete()、exists()、listFiles()

6、 字节流：OutputStream、InputStream；字符流：Writer、Reader

7、 如果需要文件操作则使用 FileXxx 的类完成，如果是内存操作：字节内存：ByteArrayXxx

8、 如果要想输出最方便的做法是使用 PrintStream

9、 如果要输入内容最方便的做法是使用 Scanner。

10、 对象序列化：将一个对象变为二进制的 byte 流，但是对象所在的类必须实现 Serializable 接口。

2.2、本次预计讲解的知识点

1、 掌握类集设置的主要目的及主要操作接口。

2、 掌握各个操作子类的区别。

3、 掌握类集的主要使用。

3、具体内容

3.1、类集概述（了解）

最早的时候可以通过对象数组保存一组数据，但是慢慢的发现如果程序中都使用对象数组开发的话，本身会存在大小的限制问题，即：所有的数组的大小是不可改变的，但是从实际的开发来看，有很多的时候是根本就无法知道到底要开辟多少的数组空间，后来通过链表解决类此类问题，但是如果每一次的开发中都使用链表的话，肯定很麻烦，所以在 Java 中专门提供了一套动态对象数组的操作类 —— 类集框架，在 Java 中类集框架实际上也就是对数据结构的 Java 实现。

在 Java 中类集框架里，为了操作方便提供了一系列的类集的操作接口，主要的操作接口有以下三个：

- Collection：存放单值的最大父接口

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

- Map：是存放一对的内容

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

- Iterator：输出作用

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

在 JDK 1.5 之后这些接口中都增加了泛型的定义，最早的时候这三个接口中的内容都使用 Object 进行操作，但是很明显这样是会存在安全问题的，那么在 JDK 1.5 之后使用了泛型，那么这种安全性的问题就解决了，此时的类集真正的可以达到了以相同的类型或高度进行操作。

在以上的三个接口中 Collection 接口并不会被直接使用，而都使用它的两个子接口：List、Set。

之所以会有这样的定义，主要的原因就是在于 EJB 2.x 产生之后所提出来的问题，在 EJB 2.x 的版本中主要使用的都是 Collection，但是后来 SUN 的一个开源项目 PetShop（宠物商店）中代码的开发里面已经严格的使用了 List 或 Set 不再使用 Collection 了，所以以后的开发中基本上就已经严格规定出来了。

宠物商店是一个所有爱好者自己开发的一套项目，感觉上与 Linux 非常的类似。在 Collection 接口中定义了如下的几个核心操作：

No.	方法名称	类型	描述
1	public boolean add(E e)	普通	向集合中增加元素
2	public void clear()	普通	删除集合中的全部内容

3	public boolean contains(Object o)	普通	判断指定内容是否存在
4	**public Iterator<E> iterator()**	**普通**	**为 Iterator 接口实例化**
5	**public boolean remove(Object o)**	**普通**	**从集合中删除元素**
6	public int size()	普通	取得集合的大小
7	public Object[] toArray()	普通	将集合变为对象数组输出
8	public <T> T[] toArray(T[] a)	普通	将集合变为对象数组输出

3.2、允许重复的子接口：List（核心重点）

List 接口本身属于 Collection 的子接口，但是 List 子接口本身大量的扩充了 Collection 接口，主要的扩充方法如下：

No.	方法名称	类型	描述
1	public void add(int index,E element)	普通	在指定的位置上增加内容
2	**public E get(int index)**	**普通**	**取得指定位置上的内容**
3	public E set(int index,E element)	普通	修改指定位置的内容
4	public ListIterator<E> listIterator()	普通	为 ListIterator 接口实例化
5	public E remove(int index)	普通	删除指定位置上的内容

既然要使用接口，那么就一定要依靠子类进行父接口的实例化。

3.2.1、新的子类：ArrayList

ArrayList 子类是在进行 List 接口操作中使用最多的一个子类，那么此类定义如下：

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

那么下面通过代码来观察基本的使用。

范例：设置内容

```
package org.lxh.listdemo;
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接口
        all.add("hello"); // 向集合中增加内容
        all.add("world"); // 向集合中增加内容
        all.add("!!!"); // 向集合中增加内容
        for (int x = 0; x < all.size(); x++) {
            System.out.println(all.get(x)) ;
        }
    }
}
```

本程序的操作代码的形式与之前的链表操作非常的类似，所以，类集的主要功能就是增加和取出数据。

以上的操作功能由于 get()方法只是 List 接口才有的，那么以后这种操作只能适合于 List 接口，如果现在接收对象的

不是 List 了，而是 Collection 呢？那么如果要想输出，则必须将所有的集合变成对象数组完成。

```
package org.lxh.listdemo;
import java.util.ArrayList;
import java.util.Collection;
public class ArrayListDemo02 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>(); // 实例化List接口
        all.add("hello"); // 向集合中增加内容
        all.add("world"); // 向集合中增加内容
        all.add("!!!"); // 向集合中增加内容
        all.remove("!!!"); // 从集合中删除指定对象
        Object obj[] = all.toArray(); // 将所有的内容变为对象数组
        for (int x = 0; x < obj.length; x++) {
            String str = (String) obj[x];
            System.out.print(str + "、");
        }
    }
}
```

可是由于以上的操作都是将内容变成了对象数组（Object），所以肯定会有安全隐患，在 Collection 接口中又提供了另外一种转换的方法：

```
package org.lxh.listdemo;
import java.util.ArrayList;
import java.util.Collection;
public class ArrayListDemo02 {
    public static void main(String[] args) {
        Collection<String> all = new ArrayList<String>(); // 实例化List接口
        all.add("hello"); // 向集合中增加内容
        all.add("world"); // 向集合中增加内容
        all.add("!!!"); // 向集合中增加内容
        all.remove("!!!"); // 从集合中删除指定对象
        String obj[] = all.toArray(new String[]{}); // 将所有的内容变为对象数组
        for (int x = 0; x < obj.length; x++) {
            System.out.print(obj[x] + "、");
        }
    }
}
```

3.2.2、旧的子类：Vector

Vector 子类实际上是最早的数据结构的实现类，也称为向量，但是在 JDK 1.0 的时候并没有所谓现在的类集框架的概念，那么只是单单的提供了一个这样的操作类而已，后来到了 JDK 1.2 之后提出了类集的框架，而且也提供了大量的类集的操作接口，所以为了保留此类的使用，在 JDK 1.2 之后让其多实现了一个 List 的接口，这样才被保留下来继续使用。

但是，不管是使用何种子类的，最终的代码的操作形式永远是一样的，因为所有的子类最终都必须发生向上转型的关系为父接口进行对象的实例化。

```
package org.lxh.listdemo;
import java.util.List;
import java.util.Vector;
public class VectorDemo {
    public static void main(String[] args) {
        List<String> all = new Vector<String>(); // 实例化List接口
        all.add("hello"); // 向集合中增加内容
        all.add("world"); // 向集合中增加内容
        all.add("!!!"); // 向集合中增加内容
        for (int x = 0; x < all.size(); x++) {
            System.out.println(all.get(x)) ;
        }
    }
}
```

3.2.3、ArrayList 和 Vector 的区别

从代码的最终的操作形式上可以发现，代码的输出结果与之前是一样的，而且没有区别，但是两者的区别还在于其内部的组成上。

No.	区别点	ArrayList	Vector
1	推出时间	JDK 1.2 之后	JDK 1.0 的时候推出
2	线程处理	ArrayList 采用异步处理	采用同步处理
3	性能	速度较快	速度相对较慢
4	安全性	非线程安全性的操作	属于线程安全的操作
5	输出	由于都是 List 接口的子类，所以都可以依靠 size()和 get()两个方法完成循环输出	
		for、Iterator、ListIterator	for、Iterator、ListIterator、Enumeration

3.3、不允许重复的子接口：Set（核心重点）

List 接口中的内容是允许重复的，但是如果现在要求集合中的内容不允许重复的话，则就可以使用 Set 子接口完成，Set 接口并不像 List 接口那样对 Collection 接口进行了大量的扩充，而与 Collection 接口的定义是完全一样的。

与 List 接口一样，如果要想使用 Set 接口则一定也要通过子类进行对象的实例化，常用的两个子类：HashSet、TreeSet。

3.3.1、散列存放的子类：HashSet

HashSet 本身是 Set 的子类，此类的定义如下：

```
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

与 ArrayList 类的定义结构是非常类似的，也是继承了一个抽象类，而且实现了接口。

```
package org.lxh.setdemo;
import java.util.HashSet;
```

```
import java.util.Set;
public class HashSetDemo {
    public static void main(String[] args) {
        Set<String> all = new HashSet<String>();
        all.add("hello");
        all.add("hello"); // 重复设置
        all.add("world");
        all.add("!!!");
        System.out.println(all) ;
    }
}
```

在 Set 接口中不允许有重复的元素出现，而且发现与 List 接口不同的是，List 采用的是顺序的方式加入的元素，而 Set 中的内容并没有任何的顺序，属于散列存放的。

3.3.2、排序存放的子类：TreeSet

TreeSet 操作的子类是使用顺序的方式保存里面的元素，下面通过代码观察：

```
package org.lxh.setdemo;
import java.util.Set;
import java.util.TreeSet;
public class TreeSetDemo {
    public static void main(String[] args) {
        Set<String> all = new TreeSet<String>();
        all.add("B");
        all.add("B");
        all.add("X");
        all.add("C");
        all.add("A");
        System.out.println(all);
    }
}
```

此时，没有重复的内容，而且可以发现虽然加入的时候没有任何的顺序，但是输出的时候却是按照顺序的方式输出。

3.3.3、关于排序的说明

TreeSet 子类的内容是允许进行排序的，那么下面就使用这个子类完成一个任意类型的排序操作。

如果多个对象要想进行排序，则无论在何种情况下都必须使用 Comparable 接口完成，用于指定排序的规则。但是在进行排序的时候实际上每一个类中的属性最好都进行判断。

```
package org.lxh.setdemo.sort;
import java.util.Set;
import java.util.TreeSet;
class Person implements Comparable<Person> {
    private String name;
```

```
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
    @Override
    public int compareTo(Person o) {
        if (this.age < o.age) {
            return 1;
        } else if (this.age > o.age) {
            return -1;
        } else {
            return this.name.compareTo(o.name);
        }
    }
}
public class SortDemo {
    public static void main(String[] args) {
        Set<Person> all = new TreeSet<Person>();
        all.add(new Person("张三", 20));
        all.add(new Person("李四", 20));
        all.add(new Person("李四", 20));
        all.add(new Person("王五", 19));
        System.out.println(all);
    }
}
```

此时加入了验证之后，可以发现，如果加入的对象重复的话，是不会正常加入的，可以去掉重复的内容。

3.3.4、关于重复元素的说明

Comparable 可以完成 TreeSet 类的重复元素的判断，如果真的可以通用的话，那么在 HashSet 中也一样可以。

但是，从实际的结果来看，现在并没有完成此类内容，因为现在在 HashSet 中 Comparable 接口并不能使用。也就是说真正要去掉重复元素的操作并不是靠 Comparable 完成的，而是靠两个方法的作用，在 Object 类中定义：

No.	方法名称	类型	描述
1	public boolean equals(Object obj)	普通	判断是否相等
2	public int hashCode()	普通	对象编码

hashCode()就相当于一个对象的唯一的编号，而如果要想进行具体的内容验证，就需要使用 equals()方法完成。

hashCode()方法返回的是一个数字，那么很明显，就必须通过一种算法，可以完成一个唯一的编号。如果现在使用的是 eclipse 进行开发的话，则此工具将会自动生成编号和比较。

```
package org.lxh.setdemo.repeat;
```

```
import java.util.HashSet;
import java.util.Set;
class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
            return false;
        return true;
    }
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
public class RepeatDemo {
    public static void main(String[] args) {
        Set<Person> all = new HashSet<Person>();
        all.add(new Person("张三", 20));
        all.add(new Person("李四", 20));
```

```
        all.add(new Person("李四", 20));
        all.add(new Person("王五", 19));
        System.out.println(all);
    }
}
```

3.4、集合输出（重点）

在正常情况中，只要是集合的输出基本上都不会采用将其变为对象数组的方式，而是采用以下的四种形式完成的：

- Iterator（95%）
- ListIterator（1%）
- Enumeration（4%）
- foreach（0%）

3.4.1、迭代输出：Iterator（核心重点）

只要是碰见集合的输出问题，不要思考，直接使用 Iterator 输出。

Iterator 本身是一个专门用于输出的操作接口，其接口定义了三种方法：

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

在 Collection 接口中已经定义了 iterator()方法，可以为 Iterator 接口进行实例化操作。

范例：集合输出的标准操作

```
package org.lxh.printdemo;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class IteratorDemo {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        all.add("hello");
        all.add("world");
        Iterator<String> iter = all.iterator();
        while (iter.hasNext()) { // 指针向下移动，判断是否有内容
            String str = iter.next();
            System.out.print(str + "、");
        }
    }
}
```

以上的代码为集合输出的标准格式。

3.4.2、双向迭代输出：ListIterator

Iterator 接口的主要的功能只能完成从前向后的输出，而如果现在要想完成双向（由前向后、由后向前）则可以通过 ListIterator 接口完成功能，此接口定义如下：

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

ListIterator 是 Iterator 的子接口，除了本身继承的方法外，此接口又有如下两个重要方法：

No.	方法名称	类型	描述
1	public boolean hasPrevious()	普通	判断是否有前一个元素
2	public E previous()	普通	取出当前的元素

但是需要注意的是，如果要想进行由后向前的输出，必须先由前向后。但是在 Collection 接口中并没有为 ListIterator 接口实例化的操作，但是在 List 接口中存在此方法：public ListIterator<E> listIterator()

```
package org.lxh.printdemo;
import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;
public class ListIteratorDemo {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        all.add("hello");
        all.add("world");
        ListIterator<String> iter = all.listIterator();
        System.out.println("=========== 由前向后输出 ============");
        while (iter.hasNext()) {
            System.out.print(iter.next() + "、");
        }
        System.out.println("\n=========== 由后向前输出 ============");
        while (iter.hasPrevious()) {
            System.out.print(iter.previous() + "、");
        }
    }
}
```

3.4.3、几乎废弃的接口：Enumeration

Enumeration 接口算是最早完成这种迭代输出的操作接口了，但是 Enumeration 接口发展到今天几乎已经不再使用了，只在一些很少的关键部分上依然会看见此方法的应用，在此接口中定义了两个方法：

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

从方法名称上可以发现，要比 Iterator 接口中的方法难记很多，所以这个接口几乎废弃了。

而且 Collection 接口本身并不支持这种输出，而只有与之同时期出现的 Vector 支持此种输出方式。

在 Vector 类中定义了一个方法：public Enumeration<E> elements()

范例：使用 Enumeration 输出

```
package org.lxh.printdemo;
import java.util.Enumeration;
import java.util.Vector;
public class EnumerationDemo {
    public static void main(String[] args) {
        Vector<String> all = new Vector<String>();
        all.add("hello");
        all.add("world");
        Enumeration<String> enu = all.elements();
        while (enu.hasMoreElements()) {
            String str = enu.nextElement();
            System.out.print(str + "、");
        }
    }
}
```

3.4.4、新的支持：foreach

在 JDK 1.5 之后增加的 foreach 输出本身也可以用于集合的输出上，但是从一般的开发角度来看，使用此种输出方式的人员并不多。

```
package org.lxh.printdemo;
import java.util.ArrayList;
import java.util.List;
public class ForEachDemo {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        all.add("hello");
        all.add("world");
        for (String str : all) {
            System.out.println(str);
        }
    }
}
```

3.5、Map 接口

Collection 接口操作的时候每次都会向集合中增加一个元素，但是如果现在增加的元素是一对的话，则就可以使用 Map 接口完成功能，Map 接口的定义如下：

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

里面需要同时指定两个泛型，主要的原因，Map 中的所有保存数据都是按照“key à value”的形式存放的，例如：以电话号码本为例：

- 张三：123456
- 李四：234567
- 王五：345678

以上的数据每次保存的时候都是按照一对的形式存放的，如果现在要找到张三的电话，很明显张三是一个 key，而他的电话就是一个 value。

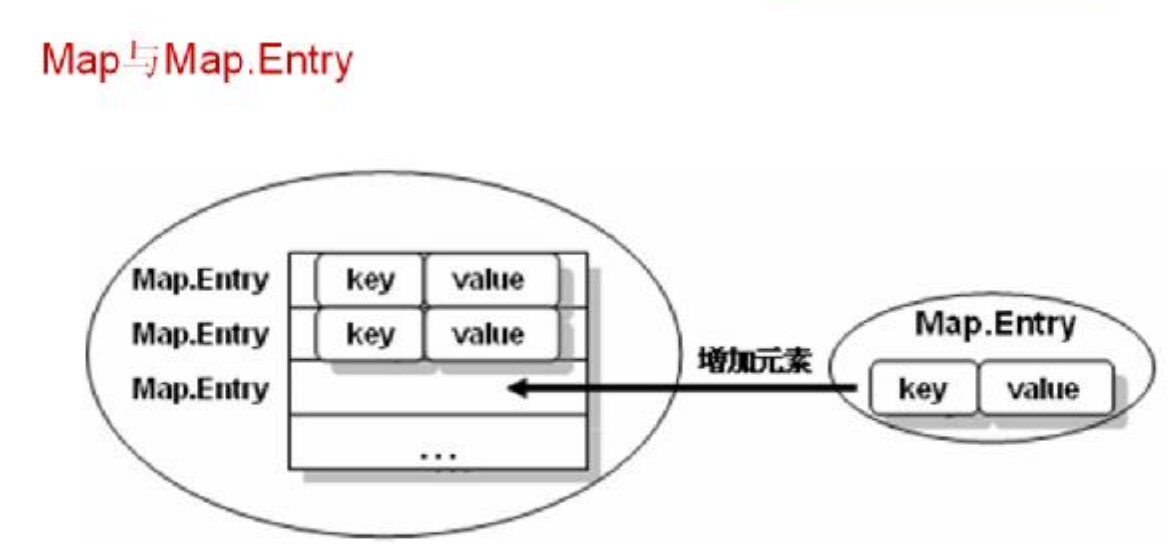
在 Map 接口中有以下几个常用方法：

No.	方法名称	类型	描述
1	public V put(K key,V value)	普通	向集合中增加元素
2	public V get(Object key)	普通	根据 key 取得 value
3	public Set<K> keySet()	普通	取出所有的 key
4	public Collection<V> values()	普通	取出所有的 value
5	public V remove(Object key)	普通	删除一个指定的 key
6	public Set<Map.Entry<K,V>> entrySet()	普通	将所有的集合变为 Set 集合

需要说明的是，在 Map 接口中还定义了一个内部接口 —— Map.Entry。

```
public static interface Map.Entry<K,V>
```

Entry 是在 Map 接口中使用的 static 定义的内部接口，所以就是一个外部接口。



3.5.1、新的子类：HashMap

如果要使用 Map 接口的话，可以使用 HashMap 子类为接口进行实例化操作。

范例：验证增加和查询

```
package org.lxh.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo01 {
    public static void main(String[] args) {
        Map<String, String> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        String value = all.get("BJ"); // 根据key查询出value
        System.out.println(value);
        System.out.println(all.get("TJ"));
    }
}
```

在 Map 的操作中，可以发现，是根据 key 找到其对应的 value，如果找不到，则内容为 null。

而且现在由于使用的是 HashMap 子类，所以里面的 key 允许一个为 null。

```
package org.lxh.mapdemo;
import java.util.HashMap;
import java.util.Map;
public class HashMapDemo02 {
    public static void main(String[] args) {
        Map<String, String> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        all.put(null, "NULL");
        System.out.println(all.get(null));
    }
}
```

现在所有的内容都有了，下面可以通过 keySet()方法取出所有的 key 集合。

```
package org.lxh.mapdemo;
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> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        all.put(null, "NULL");
        Set<String> set = all.keySet();
        Iterator<String> iter = set.iterator();
        while (iter.hasNext()) {
            String key = iter.next() ;
```

```
            System.out.println(key + " --> " + all.get(key)) ;
        }
    }
}
```

3.5.2、Map 集合的输出

按照最正统的做法，所有的 Map 集合的内容都要依靠 Iterator 输出，以上虽然是完成了输出，但是完成的不标准，Map 集合本身并不能直接为 Iterator 实例化，如果此时非要使用 Iterator 输出 Map 集合中内容的话，则要采用如下的步骤：

1、 将所有的 Map 集合通过 entrySet()方法变成 Set 集合，里面的每一个元素都是 Map.Entry 的实例；

2、 利用 Set 接口中提供的 iterator()方法为 Iterator 接口实例化；

3、 通过迭代，并且利用 Map.Entry 接口完成 key 与 value 的分离。

```
package org.lxh.mapdemo;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class MapPrint {
    public static void main(String[] args) {
        Map<String, String> all = new HashMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        all.put(null, "NULL");
        Set<Map.Entry<String, String>> set = all.entrySet();
        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());
        }
    }
}
```

以上的输出操作是 Map 接口输出的标准语句形式，以后包括各种框架的学习的输出原理与之是一样的。

3.5.3、有序的存放：TreeMap

HashMap 子类中的 key 都属于无序存放的，如果现在希望有序（按 key 排序）则可以使用 TreeMap 类完成，但是需要注意的是，由于此类需要按照 key 进行排序，而且 key 本身也是对象，那么对象所在的类就必须实现 Comparable 接口。

```
package org.lxh.mapdemo;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;
public class TreeMapDemo {
```

```
    public static void main(String[] args) {
        Map<String, String> all = new TreeMap<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        Set<Map.Entry<String, String>> set = all.entrySet();
        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());
        }
    }
}
```

3.5.4、旧的子类：Hashtable

Hashtable 与 Vector 都是属于 JDK 1.0 的时候推出的操作类，在 JDK 1.2 之后让其实现了 Map 接口，那么不管使用何种子类，肯定最终还是依靠 Map 接口完成功能。

```
package org.lxh.mapdemo;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Map;
import java.util.Set;
public class HashtableDemo {
    public static void main(String[] args) {
        Map<String, String> all = new Hashtable<String, String>();
        all.put("BJ", "BeiJing");
        all.put("NJ", "NanJing");
        Set<Map.Entry<String, String>> set = all.entrySet();
        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());
        }
    }
}
```

3.5.5、HashMap 与 Hashtable 的区别

HashMap 和 Hastable 在使用上相似，那么两者的区别如下：

No.	区别点	HashMap	Hashtable
1	推出时间	JDK 1.2 之后	JDK 1.0 的时候推出
2	线程处理	采用异步处理	采用同步处理
3	性能	速度较快	速度相对较慢

4	安全性	非线程安全性的操作	属于线程安全的操作
5	保存 null	允许 key 设置成 null	不允许设置，否则出现 NullPointerException

3.5.6、关于 key 的说明

在 Map 中如果一个任意类的对象需要作为 key 保存的话，则对象所在的类必须实现 Object 类中的 equals()和 hashCode()两个方法。

```
package org.lxh.mapdemo;
import java.util.HashMap;
import java.util.Map;
class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + age;
        result = prime * result + ((name == null) ? 0 : name.hashCode());
        return result;
    }
    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Person other = (Person) obj;
        if (age != other.age)
            return false;
        if (name == null) {
            if (other.name != null)
                return false;
        } else if (!name.equals(other.name))
```

```
            return false;
        return true;
    }
}
public class MapKeyDemo {
    public static void main(String[] args) {
        Map<Person, String> all = new HashMap<Person, String>();
        all.put(new Person("张三", 20), "ZS");
        System.out.println(all.get(new Person("张三", 20)));
    }
}
```

不过从一半的开发来看，使用 String 作为 key 的情况是最多的。

3.6、Stack 类（理解）

Stack 表示的是栈的操作类，栈是一种典型的先进后出的程序，此类的定义如下：

```
public class Stack<E> extends Vector<E>
```

Stack 是 Vector 类的子类，栈的主要操作方法：

- 入栈：public E push(E item)
- 出栈：public E pop()

范例：观察入栈和出栈的操作

```
package org.lxh.otherdemo;
import java.util.Stack;
public class StackDemo {
    public static void main(String[] args) {
        Stack<String> s = new Stack<String>();
        s.push("A");
        s.push("B");
        s.push("C");
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.pop());
        System.out.println(s.pop());
    }
}
```

3.7、Properties 类（核心重点）

Properties 类的主要功能是用于操作属性，在各个语言（包括操作系统）都会存在着许多的配置文件。所有的属性文件中的属性都是按照“key=value”的形式保存的，而且保存的内容都是 String（字符串），此类定义如下：

```
public class Properties extends Hashtable<Object,Object>
```

此类是 Hashtable 的子类，而且已经默认指定好了泛型是 Object，但是所有的属性操作中的类型肯定都是字符串，那么操作的时候主要使用的是 Properties 类完成。

Properties 类中定义的主要操作方法：

No.	方法名称	类型	描述
1	public Object setProperty(String key,String value)	普通	设置属性
2	public String getProperty(String key)	普通	根据属性的名字取得属性的内容，如果没有返回 null 结果
3	public String getProperty(String key,String defaultValue)	普通	根据属性的名字取得属性内容，如果没有则返回默认值（defaultValue）
4	public void list(PrintStream out)	普通	从一个输出流中显示所有的属性内容
5	public void store(OutputStream out,String comments) throws IOException	普通	向输出流中输出属性
6	public void load(InputStream inStream) throws IOException	普通	从输入流中读取属性内容
7	public void storeToXML(OutputStream os,String comment) throws IOException	普通	以 XML 文件格式输出属性内容
8	public void loadFromXML(InputStream in) throws IOException,InvalidPropertiesFormatException	普通	以 XML 文件格式输入属性内容

3.7.1、设置和取得属性

```
package org.lxh.otherdemo;
import java.util.Properties;
public class PropertiesDemo01 {
    public static void main(String[] args) {
        Properties pro = new Properties();
        pro.setProperty("BJ", "BeiJing");
        pro.setProperty("NJ", "NanJing");
        System.out.println(pro.getProperty("BJ"));
        System.out.println(pro.getProperty("TJ"));
        System.out.println(pro.getProperty("TJ", "没有此地区"));
    }
}
```

3.7.2、保存和读取属性

所有的属性的内容都是可以通过输出和输入流进行保存和读取的，下面先通过代码观察如何保存在普通的文件之中。

```
package org.lxh.otherdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.util.Properties;
public class PropertiesDemo02 {
    public static void main(String[] args) throws Exception {
        Properties pro = new Properties();
```

```
        pro.setProperty("BJ", "BeiJing");
        pro.setProperty("NJ", "NanJing");
        pro.store(new FileOutputStream(new File("D:" + File.separator
                + "area.properties")), "AREA INFO");
    }
}
```

此时通过一个普通的文件流将所有的属性保存在了文件之中，而且一定要记住，所有的属性文件一定要使用“*.properties”作为后缀。

```
package org.lxh.otherdemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
public class PropertiesDemo03 {
    public static void main(String[] args) throws Exception {
        Properties pro = new Properties();
        pro.load(new FileInputStream(new File("D:" + File.separator
                + "area.properties")));
        System.out.println(pro.getProperty("BJ"));
        System.out.println(pro.getProperty("TJ"));
        System.out.println(pro.getProperty("TJ", "没有此地区"));
    }
}
```

以上是将属性保存在了普通文件之中，也可以将其保存在 XML 文件之中，代码如下：

```
package org.lxh.otherdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.util.Properties;
public class PropertiesDemo04 {
    public static void main(String[] args) throws Exception {
        Properties pro = new Properties();
        pro.setProperty("BJ", "BeiJing");
        pro.setProperty("NJ", "NanJing");
        pro.storeToXML(new FileOutputStream(new File("D:" + File.separator
                + "area.xml")), "AREA INFO");
    }
}
```

以后肯定也只能从 XML 文件格式中读取属性了。

```
package org.lxh.otherdemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
public class PropertiesDemo05 {
    public static void main(String[] args) throws Exception {
        Properties pro = new Properties();
```

```
        pro.loadFromXML(new FileInputStream(new File("D:" + File.separator
                + "area.xml")));
        System.out.println(pro.getProperty("BJ"));
        System.out.println(pro.getProperty("TJ"));
        System.out.println(pro.getProperty("TJ", "没有此地区"));
    }
}
```

3.7.3、列出属性

在实际的开发中，如果使用了 IO 流进行操作的话，有可能由于编码的不同而造成乱码的问题，因为程序的编码和本地环境的编码不统一，所以会造成乱码的显示，那么该如何查询本地编码呢，就需要通过 Properties 类的 list()方法完成。

```
package org.lxh.otherdemo;
public class PropertiesDemo06 {
    public static void main(String[] args) throws Exception {
        System.getProperties().list(System.out) ;
    }
}
```

关于文件的编码，在程序中主要有以下几种：

- GB2312、GBK：表示的是国标编码，2312 表示的是简体中文，而 GBK 包含了简体和繁体中文
- ISO 8859-1：主要用于英文字母的编码方式
- UNICODE：Java 中使用十六进制进行编码，能够表示出世界上所有的编码
- UTF 编码：中文采用十六进制，而普通的字母依然采用和 ISO 8859-1 一样的编码

在以后的程序中经常会出现乱码的问题，这种问题造成的根本原因就是在于编码不统一。

```
package org.lxh.otherdemo;
import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStream;
public class EncodingDemo {
    public static void main(String[] args) throws Exception {
        OutputStream output = new FileOutputStream(new File("D:"
                + File.separator + "hello.txt"));
        String info = "你好啊，中国！" ;
        output.write(info.getBytes("ISO8859-1")) ;
        output.close() ;
    }
}
```

3.7.4、工厂设计终极版（理解）

工厂设计经过发展已经有两种模型：

1、 简单工厂，所有的工厂类随着子类的增加而要修改

2、 反射工厂，只要传递进去完整的包.类，就可以完成实例化操作

但是，第 2 种实现本身也有问题，因为每次都要传递一个完整的“包.类”实在是太麻烦了。

```
package org.lxh.factorydemo;
import java.io.File;
import java.io.FileInputStream;
import java.util.Properties;
interface Area {
    public void getInfo();
}
class BeiJing implements Area {
    public void getInfo() {
        System.out.println("你在北京，欢迎您！");
    }
}
class NanJing implements Area {
    public void getInfo() {
        System.out.println("你在南京，欢迎您！");
    }
}
class Factory {
    public static Area getInstance(String className) {
        Area a = null;
        try {
            a = (Area) Class.forName(className).newInstance();
        } catch (Exception e) {
        }
        return a;
    }
}
public class FactoryDemo {
    public static void main(String[] args) throws Exception {
        Properties pro = new Properties();
        pro.load(new FileInputStream(new File("D:" + File.separator
                + "area.properties")));
        Area a = Factory.getInstance(pro.getProperty("area"));
        a.getInfo();
    }
}
```

配置文件：area.properties

```
#AREA INFO
#Mon Dec 28 14:55:28 CST 2009
area=org.lxh.factorydemo.NanJing
```

现在的程序中可以发现，都是通过配置文件控制的，而且只要配置文件改变了，程序可以立刻发生改变。达到了配置文件与程序相分离的目的。

3.8、两种问题（重点）

3.8.1、一对多

一个人有多本书，要求通过程序描述。

```
package org.lxh.demo01;
import java.util.ArrayList;
import java.util.List;
public class Person {
    private String name;
    private int age;
    private List<Book> books;
    public Person() {
        super();
        this.books = new ArrayList<Book>();
    }
    public Person(String name, int age) {
        this();
        this.name = name;
        this.age = age;
    }
    public List<Book> getBooks() {
        return this.books;
    }
    @Override
    public String toString() {
        return "姓名：" + this.name + "，年龄：" + this.age;
    }
}
```

因为现在根本就无法明确的知道一个人有多少本书。

```
package org.lxh.demo01;
public class Book {
    private String title;
    private Person person;
    public Book(String title) {
        this.title = title;
    }
    public Person getPerson() {
        return person;
    }
    public void setPerson(Person person) {
        this.person = person;
```

```
    }
    @Override
    public String toString() {
        return "书名：" + this.title;
    }
}
```

在主方法中设置两者的关系。

```
package org.lxh.demo01;
import java.util.Iterator;
public class Test {
    public static void main(String[] args) {
        Person per = new Person("张三", 20);
        Book b1 = new Book("Java");
        Book b2 = new Book("WEB");
        per.getBooks().add(b1);
        per.getBooks().add(b2);
        b1.setPerson(per);
        b2.setPerson(per);
        System.out.println(per);
        Iterator<Book> iter = per.getBooks().iterator();
        while (iter.hasNext()) {
            System.out.println("\t|- " + iter.next()) ;
        }
    }
}
```

3.8.2、多对多

一个学生可以参加多门课程，一门课程有多个学生参加，现在要求通过一个学生可以找到他所参加的全部课程，也可以通过一门课程找到参加本课程的所有学生。

```
package org.lxh.demo02;
import java.util.List;
public class Course {
    private List<Student> students ;
}
```

```
package org.lxh.demo02;
import java.util.List;
public class Student {
    private List<Course> courses ;
}
```

类集，当不知道有多少个元素的时候就使用类集。

3.9、集合的工具类：Collections（了解）

从定义格式上看与 Collection 非常的类似，但是两者并没有任何的关系，此类定义如下：

```
public class Collections extends Object
```

是一个 Object 类的子类，并没有跟 Collection 有任何关系。

```
package org.lxh.demo;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
public class CollectionsDemo {
    public static void main(String[] args) {
        List<String> all = new ArrayList<String>();
        Collections.addAll(all, "A", "B", "C");
        System.out.println(all);
    }
}
```

4、总结

1、 类集就是动态对象数组
2、 类集中的主要接口：Collection、List、Set、Map、Iterator、ListIterator、Enumeration
3、 类集的主要目的就是为了保存数据和输出数据
4、 所有的属性都依靠 Iterator 完成
5、 就是各个子类的不同