请大家阅读文档时，在视图里勾选导航窗格，在左边显示章节目录方便浏览。

一、 编程第 2 题中的对象克隆问题

在第 11 章 PPT 里介绍了一种克隆的方法，即在 clone 方法里 new 一个新的对象，再对这个对象的每个成员进行 clone。但是这种方法在有继承关系的情况下，不利于复用父类的 clone 方法。这里介绍另外一种方法，就是调用 super.clone()首先得到子类对象里，父类那部分数据成员的克隆，再去克隆子类对象新加的数据成员。看下面的示例代码 (注意看注释)：

```
//首先必须实现 Cloneable 接口
class A implements Cloneable{

    protected int[] values = {1,2,3,4};

    public int[] getValues(){
        return values;
    }

    //覆盖 clone 方法，提升为 public，实现为深拷贝
    @Override
    public Object clone() throws CloneNotSupportedException {

        //不需要 new，调用 super.clone 拿到对象，再对成员逐个 clone

        //注意 super.clone 返回 Object 在运行时就是 A 类型 ( Object.clone 是本地方法，看不到实
现,估计是编译器保证了这一点 )
        A newObj = (A)super.clone(); //注意调用 Object 的 clone，是浅拷贝

        newObj.values = this.values.clone(); //数组的 clone 是深拷贝

        return newObj;
    }

}
```

```
class B extends A  {
    protected double[] doubleValues = {1.0, 2.0,3.0,4.0};

    public double[] getDoubleValues() {
        return doubleValues;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {

        //不需要 new，调用 super.clone 拿到对象，再对成员逐个 clone

        //这个时候对象的父类部分数据成员已经克隆好(由父类保证是深拷贝)

        B newObj = (B)super.clone(); //调用 A 的 clone，是深拷贝

        //再来克隆子类新的数据成员

        newObj.doubleValues = this.doubleValues.clone();

        return newObj;
    }
}
```

类似地，我们在实现子类的 equals，toString 方法时，应该去调用父类的相应实现，这样才能代码重用。面向对象的继承机制不就是为了代码重用，提高系统的可靠性么？如果都是重复造轮子，首先引入 bug 的风险大大增加，其次开发效率也低。

二．编程第 3 题中的迭代器

2.1 什么是迭代器

- 意图
  - 迭代器模式的目的是设计一个迭代器，提供一种可顺序访问聚合对象中各个元素的方法，但不暴露该对象内部表示
- 适用场合
  - 访问一个聚合对象的内容而无需暴露其内部表示
  - 支持对聚合对象的多种遍历
  - 为遍历不同的聚合结构提供一个统一接口 (支持多态迭代)

假设我们针对一个问题的内部数据结构有多种，有数组、有 ArrayList，那么针对数组和 ArrayList，我们遍历每个元素的方法是不同的。我们希望不管内部数据结构是怎么样的，能够通过一致的方式去顺序访问每个元素，而不暴露内部的数据结构。

JDK 给我们定义了迭代器接口 java.util.Iterator，该接口包括三个方法:其中 hasNext()方法当还有元素没有遍历完则返回 true；next 方法返回下一个要遍历的元素；remove 方法删除最后一个被遍历的元素。下面看一个使用例子：

```
public static void main(String[] args){
    String[] strings = {"aaa","bbb","ccc"};
    //注意数组转 List 的方法
    List<String> list = Arrays.asList(strings);

    //我们可以采用 for 循环来便利每个元素，但这种方式是基于特定数据结构的
    //如果集合类型换了，for 循环代码就不能用
    //因此我们采用迭代器来遍历
    Iterator<String> it = list.iterator(); //iterator 方法返回迭代器
    while (it.hasNext()){
        String s = it.next();
        //对 s 作进一步处理
    }
}
```

那么针对数组我们是否可以自己实现一个迭代器呢，当然可以。看下面示例代码：

```java
//一个数组迭代器的简单 Demo，这里没有采用泛型，注意是实现了 Java 的 Iterator 接口
class ArrayIterator implements java.util.Iterator{
    private int pos = 0; //保留迭代的当前位置
    private Object[] a = null; //要迭代的数组

    /**
     * 构造函数需要传进来一个数组
     * @param array
     */
    public ArrayIterator(Object[] array){
        a = array;
    }

    @Override
    public boolean hasNext() {
        if(pos >= a.length)
            return false;
        else
            return true;
    }


    @Override
    public Object next() {
        if(hasNext()){
            Object c = a[pos];
            pos ++;
            return c;
        }
        else
            return null;
    }

    //remove 方法略去


    public static void main(String[] args){
        String[] strings = {"aaa","bbb","ccc"};

        //现在我们用迭代器来遍历数组元素
        Iterator it = new ArrayIterator(strings);
        while (it.hasNext()){
            String s = (String)it.next(); //instanOf 检查略去 ce
            System.out.println(s);
        }
    }
}
```

现在我们可以用一致的方式来遍历数组了。比如我们需要实现一个通用的函数来处理不同集合类型的数据，我们可以这样定义方法，**注意参数类型是一个迭代器**：

```java
public static  void processDatas(Iterator it){
    while (it.hasNext()){
        Object o = it.next();

        //进一步处理

    }
}
```

现在来使用 processDatas 方法，不管是什么样的集合类型数据结构，都可以在 processDatas 方法以一致的方式迭代处理：

```
String[] strings = {"aaa","bbb","ccc"};
List<String> list = new ArrayList<>();
//向 list 里添加元素
//现在不管是数组，还是 list，都可以在方法 processDatas 里以一致的方法处理
processDatas(list.iterator());
processDatas(new ArrayIterator(strings));
```

2.2 复合迭代器的实现思路

复合迭代器用于迭代复合组件的子组件，由于复合组件的子组件还可能是复合组件，因此需要在复合迭代器里维护一个子组件的迭代器的集合 list（或者堆栈）：如果当前遍历到的子元素也是复合组件，需要将这个子元素的迭代器加入到 list。

由于遍历是从树的根节点开始，因此复合迭代器的构造函数需要传入根节点的迭代器并放入 list。复合迭代器每次从 list 首部取出迭代器 it，如果 it 已经遍历完所有元素，则应该将 it 从 list 里删除；如果 list 为空了，表示所有元素遍历完毕，hasNetx 返回 false。如果当前遍历到的子元素也是复合组件，需要将这个子元素的迭代器加入到 list 尾部。

这里要注意一个有趣的问题：当我们采用 list 或者 stack 来维护子组件的迭代器的集合时，会导致二种不同的遍历策略：广度优先或深度优先。这个问题大家如果有时间、有兴趣可以试试。