第三次作业（50 points）

第一题：（5）

```
public class Fred {
        public int x = 0;
        public Fred(int x)
        {
            this.x=x;
        }
        pubic int getX()
         {
            return x;
        }
    }
```

上述代码是否正确使用封装概念，说明理由。

否，在类 Fred 中需要将变量 x 声明为 private。
封装概念中类中的变量对外完全不可见，对变量的操作需要通过方法来实现。
如：要改变 x 的值 ，需要增加一个 setX(int x)方法。
在改变 x 的值时通过调用 objectName.setX(value) 方法，而不是通过 objectName.x=value

第二题：（5）

简要说明 interface，abstract class，class 之间的关系。
意思正确即可，从不同的方面看具有不同的意义。Google 之

第三题：（20）

```
public interface Person
{
    String getName();//返回name
    String getDescription();//返回description
    int changeSomething();
     }
```

1.创建类 Man ,该类实现 interface Person：

（1）该类具有三个私有成员变量分别为 name，description，count。这两个变量均在创建对象过程中初始化。name 保存实例的名称，description 保存描述信息。这两个成员变量可以填充任意你想填写的内容。count 初始值为 0，在调用 changeSomething 方法时返回 count=count-1。

（2）在类中增加一个方法 move，没有参数和返回型，调用该方法时输出“I am moving...”

**注意一个文件仅包含一个接口或者类，文件名和类名一致，养成良好的编码习惯

```
public class Man implements Person
{
    private String name;
    private String description;
    private int count=0;
    public Man(String name,String description)
    {
        this.name=name;
        this.description=description;
    }
    @Override
    public String getName()
    {
        return name;
    }
    @Override
    public String getDescription()
    {
        return description;
    }
    @Override
    public int changeSomething()
    {
        return --count;
    }
    public void move()
    {
        System.out.println("Im moving");
    }
}
```

2.创建类 SuperMan，该类继承 Man

（1）在调用 move 方法时输出“I am flying...”。

（2）在该类中添加一个方法 fly，没有参数和返回值。调用该方法时输出“fly，fly，fly，I am a SuperMan”。

（3）调用 changeSomething 方法时返回 count=count+1；（count 初始值为 0）

（4）具有两个构造方法，一个构造方法与 Man 的构造方法的参数类型，数量相同。另一个构造方法没有任何构造参数，使用该构造方法实例化后，调用 getName 和 getDescription 分别返回“superMan”和“I can fly”。

*使用类的继承中，最重要的一个观点认为，使用继承是为了尽量不改变或尽量小的改变原有代码来实现行为的改变或新行为的增加。因为在 Man 中的变量 count 在 SuperMan 中是不可见的，因此在 SuperMan 中重新声明了一个变量 count，以符合行为的改变，同时又没有改变父类的行为。在 Man 中增加额外的方法来满足 SuperMan 的需求是不可取的。

```
public class SuperMan extends Man
{
    private int count=0;

    public SuperMan()
    {
        super("superMan","I can fly");
    }

    public SuperMan(String name,String description)
    {
        super(name,description);
    }

    public int changeSomething()
    {
        return ++count;
    }

    public void move()
    {
        System.out.println("Im flying");
    }

    public void fly()
    {
        System.out.println("fly,fly,fly");
```

```
    }

}
```

3.利用 1，2 中定义的类

```
Man man=new Man("man","nothing");
SuperMan superman=new SuperMan("superman","nothing");
Person pman=new Man("pman","nothing");
Person psman=new SuperMan("psman","nothing");
Man msMan=new SuperMan("msMan","nothing");
```

将实例 man，superman，pman，psman，msMan 所能够调用方法和调用结果列表。对所有的调用方法和对应的调用结果进行分析。

4.使用 1，2 分析下列代码是否可行，不可行说明原因

先做出如下定义：

子类向父类转型定义为 向上转型（自动）

父类向子类转型定义为 向下转型（强制转型）

(1)

```
Man man=new Man("man","nothing");
SuperMan sman=(SuperMan)man;
```

不可性，

该题第二行为向下转型，在编译时无法通过，而实际上在构造 man 时使用的构造函数 Man(String name,String description) 因此 man 实际指向一个类 Man 的实例。在向下转型后成为 SuperMan，但是其本质没有变，与实际的 SuperMan 相比没有 fly 方法，这会对程序的运行产生巨大问题，因此编译器会检查这种不靠普的向下转型。

(2)

```
Man man=new SuperMan("superman","nothing");   //向上转型（自动）
SuperMan sman=(SuperMan)man;//向下转型
Man man2=(Man)sman;//向上转型（自动，实际可写为 Man man2=sman）
```

完全可行，

第一行：变量 man 实际指向的 SuperMan 的实例， 因为 SuperMan 具有全部的 Man 的行为，所以转型是自动的。

第二行：由于 man 本身实际指向 SuperMan 实例，所以向下转型（强制转型）一定会成功。

第三行：解答同第一行。

(3)

```
Person man=new Man("man","nothing");
SuperMan sman=(SuperMan)man;
```

不可性，

第一行，向上转型，因为 Man 具有 Person 的所有行为，所以转型会成功。

第二行，向下转型，因为变量 man 实际所指为 Man 的实例，而不是 SuperMan，所以不会成

功。

第四题：（5）

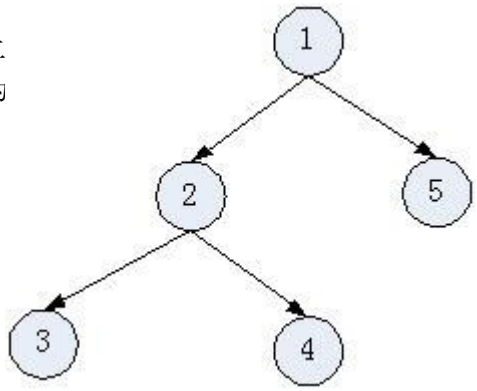
请你设计的一个类，类名为 Singleton，要求该类只能创建一个实例。请写出类的框架代码。

*此题为单例设计模式，无论如何仅有一个该类的实例。

```
public class Singleton
{
    private static Singleton s=null;
    private Singleton() //可以使用protected修饰
    {
    }
    public static Singleton getInstance()
    {
        if(s==null)
        {
            s=new Singleton();
        }
        return s;
    }
    }
```

第五题：（15）

二叉树示例：



（1）对类 Node 进行填充代码，构造二叉树结点,结点中保存数据的类型为 int。

```
class Node
{
    //your code here
    必须符合基本树节点定义

}
```

（2）利用（1）中二叉树结点构造二叉树，并实现二叉树的遍历。

a.该类包含5个方法，
（1）输出树中的总节点数.
（2）输出前序、中序、后序遍历结果。
（3）按数据大小降序排列输出.
b.二叉树可以任意构造.作爲Operation的構造參數
c.请按照实现功能的需要适当增加Operation的其他部分。
d.所有方法均将结果输出到控制台
e. Operation 类中已经预先给定的方法名和参数形式不可改变。

以下为最后一题的一种实现，学习 JAVA 最重要的一点就是学习它的类库，知道它的类库做什么。

```
import java.util.Collections;
class Node implements Comparable<Node>
{
    // your code here
    public Node left;
    public Node right;
    public int data;

    public Node(int data)
    {
        this.data = data;
    }

    @Override
    public int compareTo(Node o2)
    {
        return o2.data-this.data;
    }
   }
```

```
import java.util.ArrayList;

class Operation
{
    private Node root = null;
    private ArrayList<Node> list = null;

    public Operation(Node root)
    {
        this.root = root;
```

```
        list = new ArrayList<Node>();
        preProcess(root);
        Collections.sort(list);
    }

    private void preProcess(Node root)
    {
        if (root == null)
        {
            return;
        }
        list.add(root);
        preProcess(root.left);
        preProcess(root.right);
    }

    // 前序
    public void preOrder()
    {
        System.out.print("pre:");
        prePrint(root);
        System.out.println();

    }

    private void prePrint(Node root)
    {

        if (root == null)
        {
            return;
        }
        System.out.print(root.data+"\t");

        prePrint(root.left);
        prePrint(root.right);


    }

    // 后序
    public void postOrder()
    {
        System.out.print("post:");
```

```
        postPrint(root);
        System.out.println();

    }

    private void postPrint(Node root)
    {
        if (root == null)
        {
            return;
        }
        postPrint(root.left);
        postPrint(root.right);
        System.out.print(root.data+"\t");
    }

    // 中序
    public void inOrder()
    {
        System.out.print("in:");
        inPrint(root);
        System.out.println();
    }

    private void inPrint(Node root)
    {
        if (root == null)
        {
            return;
        }
        inPrint(root.left);
        System.out.print(root.data+"\t");
        inPrint(root.right);

    }

    // 统计数量
    public void size()
    {
        System.out.println("size:"+list.size());
    }

    // 按降序打印出所有值
    public void printAll()
```

```
    {
        System.out.print("all:");
        for(Node i:list)
        {
            System.out.print(i.data+",");
        }
        System.out.println();
    }

    public static void main(String[] args)
    {
        Node root = new Node(9);

        root.left = new Node(2);
        root.right = new Node(3);
        root.left.right = new Node(8);
        root.left.left = new Node(-1);

        Operation op = new Operation(root);

        op.size();
        op.preOrder();
        op.inOrder();
        op.postOrder();
        op.printAll();
    }
}
```