4.2.1 将用户输入的数组进行翻转
1.程序问用户想要输入多少个数字。
2.程序将用户输入的数字全部以 Double 数据类型的数组存储起来。
3.程序将这个数组用一个名为 reverseInPlace 方法翻转后呈现给用户。

参考代码如下。

```
import java.util.Scanner;

public class Exercise_1_1 {
	public static void reverseInPlace(double[] arr) {
		for (int i = 0; i < arr.length / 2; i++) {
			double temp = arr[i];
			arr[i] = arr[arr.length - 1 - i];
			arr[arr.length - 1 - i] = temp;
		}
	}
	public static void main(String[] args) {
		Scanner sc = new Scanner(System.in);
		System.out.println("Enter the size of array: ");
		int size = Integer.parseInt(sc.nextLine());
		double[] arr = new double[size];

		System.out.println("Enter the elements of array: ");
		for (int i = 0; i < size; i++) {
			arr[i] = sc.nextDouble();
		}

		reverseInPlace(arr);

		System.out.print("The reversed array is: [");
		for (int i = 0; i < size; i++) {
			System.out.print(arr[i] + " ");
		}
		System.out.println("]");
	}
}
```

4.2.2 判断数组是否含有连续四个相同的值
1.程序使用 public static boolean isConsecutiveFour(int[] values)方法以判断数组是否含有连续四个相同的值。
参考代码如下。

```
import java.util.Scanner;
```

```
public class Exercise_1_2 {
    public static boolean isConsecutiveFour(int[] values){
        int len = values.length;
        if(len < 4){
            return false;
        }
        else{
            for (int i = 0; i < len - 3; i++){
                if (values[i] == values[i + 1] && values[i] == values[i + 2] && values[i] ==
values[i + 3]){
                    return true;
                }
            }
            return false;
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter the number of elements in the array: ");
        int size = Integer.parseInt(sc.nextLine());
        int[] arr = new int[size];
        System.out.println("Enter the elements in the array: ");
        for (int i = 0; i < size; i++) {
            arr[i] = sc.nextInt();
        }
        boolean isConsecutive = isConsecutiveFour(arr);
        if(isConsecutive){
            System.out.println("Consecutive");
        }
        else {
            System.out.println("Not Consecutive");
        }
    }
}
```

4.2.3 用于计算运行时间的类

设计一个名为 StopWatch 的类。

该类包含：

1.带有 getter 方法的私有数据字段 startTime 和 endTime。

2.一个空构造函数，用当前时间初始化 startTime 和 endTime。

3.一个名为 start（）的方法，它将 startTime 重置为当前时间。

4.一个名为 stop（）的方法，将 endTime 设置为当前时间。

5.一个名为 getElapsedTime（）的方法，它返回经过的时间（秒表的停止和开始时间（以毫

秒为单位）。
可以使用 System.currentTimeMillis（）来获取当前时间。
最后编写一个测试程序，测量使用讲座中讨论的选择排序对 100000 个数字进行排序的执行时间。
参考代码如下。

```
public class Exercise_1_3 {
    public static class StopWatch{
        private long startTime;
        private long endTime;

        public StopWatch(){
            startTime = System.currentTimeMillis();
            endTime = startTime;
        }

        public void start(){
            startTime = System.currentTimeMillis();
        }
        public void stop(){
            endTime = System.currentTimeMillis();
        }
        public double getElapsedTime(){
            return (endTime - startTime) / 1000.0;
        }
    }


    public static void selectionSort(int[] list) {
        for (int i = 0; i < list.length; i++) {
            // Find the minimum in the list[i..list.length-1]
            int currentMin = list[i];
            int currentMinIndex = i;
            for (int j = i + 1; j < list.length; j++) {
                if (currentMin > list[j]) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }
            // Swap list[i] with list[currentMinIndex] if necessary;
            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
    }

    public static void main(String[] args) {
        int[] numbers = new int[100000];
        for(int i = 0; i < numbers.length; i++){
            numbers[i] = (int)(Math.random() * 100000);
        }
        StopWatch stopWatch = new StopWatch();
        stopWatch.start();

        selectionSort(numbers);
        stopWatch.stop();
        System.out.println(stopWatch.getElapsedTime());
    }
}
```

4.2.4 学生记录类

创建一个学生记录类，代表学生参加高级 OOP 课程。

1.每个学生对象应代表名字、姓氏、电子邮件地址和组号。

2.包含一个 toString（）方法，该方法返回一个学生的字符串表示形式，以及一个 less（）方法通过两个学生的组号进行比较。

最后编写一个测试程序，打印和比较学生对象。

参考代码如下。

```
public class Exercise_1_4 {
    public static class Student {
        private float firstName;
        private float lastName;
        private float emailAddress;
        private int groupID;

        public Student(float firstName, float lastName, float emailAddress, int groupID) {
            this.firstName = firstName;
            this.lastName = lastName;
            this.emailAddress = emailAddress;
            this.groupID = groupID;
        }

        public String toString(){
            return "Student: " + firstName + " " + lastName + " " + emailAddress + " " +
groupID;
        }
        public void less(Student other){
```

```
                    if (this.groupID > other.groupID){
                        System.out.println("This student is bigger");
                    }
                    else if (this.groupID < other.groupID){
                        System.out.println("This student is smaller");
                    }
                    else {
                        System.out.println("They are equal");
                    }
                }

        }
        public static void main(String[] args) {
                Student student1 = new Student(1,2,3,4);
                Student student2 = new Student(2,3,4,5);
                Student student3 = new Student(3,4,5,4);
                student1.less(student2);
                student2.less(student3);
                student1.less(student3);
        }
}
```

二！！！！！

MyPoint 类
类中包含：
表示坐标的实例变量（数据字段）x 和 y，以及它们的获取器（getter）方法。
一个空构造函数，用于创建一个坐标为 (0.0, 0.0) 的点。
一个构造函数，用于根据指定的坐标创建一个点。
一个名为 distance 的实例方法，用于返回此点与另一个 MyPoint 类型点之间的距离。
一个同样名为 distance 的静态方法，用于返回两个 MyPoint 对象之间的距离。
并且编写一个测试程序，创建三个点：(0.0, 0.0)，(10.25, 20.8) 和 (13.25, 24.8)，并使用两种 distance 实现来显示它们之间的距离。
注意这里实例方法和静态方法之间的区别。
示例代码如下。

```
public class MyPoint {
        // 实例变量
        double x;
        double y;
```

```
// 空构造函数，创建一个坐标为 (0.0, 0.0) 的点
public MyPoint() {
    this.x = 0.0;
    this.y = 0.0;
}

// 根据指定的坐标创建一个点
public MyPoint(double x, double y) {
    this.x = x;
    this.y = y;
}

// 获取器方法
public double getX() {
    return x;
}

public double getY() {
    return y;
}

// 实例方法，返回此点与另一个 MyPoint 类型点之间的距离
public double distance(MyPoint other) {
    return Math.sqrt(Math.pow(this.x - other.x, 2) + Math.pow(this.y - other.y, 2));
}

// 静态方法，返回两个 MyPoint 对象之间的距离
public static double distance(MyPoint p1, MyPoint p2) {
    return Math.sqrt(Math.pow(p1.x - p2.x, 2) + Math.pow(p1.y - p2.y, 2));
}

// 测试程序
public static void main(String[] args) {
    // 创建三个点
    MyPoint p1 = new MyPoint(0.0, 0.0);
    MyPoint p2 = new MyPoint(10.25, 20.8);
    MyPoint p3 = new MyPoint(13.25, 24.8);

    // 使用实例方法计算距离
    double distance1 = p1.distance(p2);
    System.out.println("Distance between p1 and p2 (instance method): " + distance1);

    // 使用静态方法计算距离
    double distance2 = MyPoint.distance(p1, p2);
```

```
        System.out.println("Distance between p1 and p2 (static method): " + distance2);
    }
}
```

大数类

编写一个函数，输出前 10 个具有 50 位十进制数字且能被 2 或 3 整除的数。

示例代码如下。

```
import java.math.BigInteger;

public class BigDivisibleNumbers {
    public static void main(String[] args) {
        printNumbers();
    }

    public static void printNumbers() {
        // 定义 50 位数的最小值（10^49）
        BigInteger current = BigInteger.TEN.pow(49);
        // 定义常量大数 2 和 3，用于取模运算
        BigInteger TWO = BigInteger.valueOf(2);
        BigInteger THREE = BigInteger.valueOf(3);

        int count = 0;
        while (count < 10) {
            // 检查是否能被 2 或 3 整除
            if (current.mod(TWO).equals(BigInteger.ZERO) ||
                    current.mod(THREE).equals(BigInteger.ZERO)) {
                // 转换为字符串并补零（确保输出为 50 位）
                String numStr = current.toString();
                System.out.println(numStr);
                count++;
            }
            current = current.add(BigInteger.ONE); // 递增
        }
    }
}
```

Person 类

要求如下：

Person 类及其两个子类 Student 和 Employee。

Employee 类的两个子类 Faculty 和 Staff。

Person 有 name（姓名）、address（地址）、phone（电话号码）和 email（电子邮件）。

Student 有 classStatus（年级状态，如新生、大二、大三或大四），定义为常量。

Employee 有 office（办公室）、salary（薪水）和 dateHired（雇用日期）。
Faculty 有 officeHours（办公时间）和 rank（职称）。
Staff 有 title（职称）。
每个类添加一个只接受 name 参数的构造函数。
重写每个类的 toString 方法，以显示类名和人名。
确定 Person 类中 name 的访问修饰符。
创建一个数组，包含 Person、Student、Employee、Faculty 和 Staff 对象。
使用多态性调用它们的 toString() 方法。
示例代码如下。

```
import java.time.LocalDate;

// Person class
class Person {
	protected String name;
	protected String address;
	protected String phone;
	protected String email;

	public Person(String name) {
		this.name = name;
	}

	@Override
	public String toString() {
		return "Person: " + name;
	}
}

// Student class
class Student extends Person {
	public static final String[] STATUS = {"freshman", "sophomore", "junior", "senior"};
	private int classStatus;

	public Student(String name, int status) {
		super(name);
		this.classStatus = status;
	}

	@Override
	public String toString() {
		return "Student: " + name + ", Status: " + STATUS[classStatus];
	}
}
```

```
// Employee class
class Employee extends Person {
        protected String office;
        protected double salary;
        protected LocalDate dateHired;

        public Employee(String name, String office, double salary, LocalDate dateHired) {
                super(name);
                this.office = office;
                this.salary = salary;
                this.dateHired = dateHired;
        }

        @Override
        public String toString() {
                return "Employee: " + name + ", Office: " + office;
        }
}

// Faculty class
class Faculty extends Employee {
        private String officeHours;
        private String rank;

        public Faculty(String name, String office, double salary, LocalDate dateHired, String
officeHours, String rank) {
                super(name, office, salary, dateHired);
                this.officeHours = officeHours;
                this.rank = rank;
        }

        @Override
        public String toString() {
                return "Faculty: " + name + ", Rank: " + rank;
        }
}

// Staff class
class Staff extends Employee {
        private String title;

        public Staff(String name, String office, double salary, LocalDate dateHired, String title) {
                super(name, office, salary, dateHired);
        this.title = title;
    }

    @Override
    public String toString() {
        return "Staff: " + name + ", Title: " + title;
    }
}

// 测试程序
public class TestPerson {
    public static void main(String[] args) {
        Person[] people = new Person[5];
        people[0] = new Person("John Doe");
        people[1] = new Student("Jane Smith", 1);
        people[2] = new Employee("Alice Johnson", "123 Main St", 50000, LocalDate.of(2020, 6, 
1));
        people[3] = new Faculty("Bob Brown", "456 Elm St", 60000, LocalDate.of(2018, 8, 15), 
"8am-5pm", "Associate Professor");
        people[4] = new Staff("Charlie Davis", "789 Maple St", 45000, LocalDate.of(2019, 3, 20), 
"Manager");

        for (Person p : people) {
            System.out.println(p);
        }
    }
}
```

继承版 MyStack 类
将这次学习的 MyStack 类改为继承自 ArrayList<Object>类。此外，还需要编写一个测试程序，该程序提示用户输入五个字符串，并将它们以相反的顺序显示出来。
示例代码如下。

```
import java.util.ArrayList;
public class MyStack extends ArrayList<Object> {
    // 检查栈是否为空
    public boolean isEmpty() {
        return super.isEmpty();
    }

    // 获取栈的大小
    public int getSize() {
        return super.size();
    }
```

```
        // 返回栈顶元素
        public Object peek() {
                if (isEmpty()) return null;
                return super.get(getSize() - 1);
        }

        // 移除并返回栈顶元素
        public Object pop() {
                if (isEmpty()) return null;
                return super.remove(getSize() - 1);
        }

        // 向栈中添加元素
        public void push(Object o) {
                super.add(o);
        }

        // 查找元素在栈中的位置
        public int search(Object o) {
                return super.lastIndexOf(o);
        }

        // 返回栈的字符串表示
        public String toString() {
                return "stack: " + super.toString();
        }
}
import java.util.Scanner;

public class TestMyStack {
        public static void main(String[] args) {
                MyStack stack = new MyStack();
                Scanner scanner = new Scanner(System.in);

                System.out.println("Enter five strings:");
                for (int i = 0; i < 5; i++) {
                        stack.push(scanner.nextLine());
                }

                scanner.close();

                System.out.println("Strings in reverse order:");
                while (!stack.isEmpty()) {
```

```
                System.out.println(stack.pop());
            }
        }
}
```

三！！！！！
3.下面哪一个选项中的代码可以替换下面的代码？
A. return this.id + ((Student)o).id;
B. return this.id - ((Student)o).id;
C. return this.id * ((Student)o).id;
D. return this.id / ((Student)o).id;

```
public class Student implements Comparable {
        int id;

        public int compareTo(Object o) {
                if (this.id > ((Student)o).id)
                        return 1;
                if (this.id == ((Student)o).id)
                        return 0;
                return -1;
        }
}
```

在 compareTo 方法中，只要返回值满足以下条件即可：
如果当前对象小于参数对象，返回负整数；
如果当前对象等于参数对象，返回 0；
如果当前对象大于参数对象，返回正整数。
所以选 B。

下列代码会报什么错？
A. 第三行编译错误。
B. 第三行运行错误。
C 第四行编译错误。
D. 第四行运行错误。

```
4.public class AutomaticUnboxing {
        public static void main(String[] args) {
             Integer num = null;
             int x = num; // 这里尝试将 Integer 类型的对象自动拆箱为 int 类型
             System.out.println(x);
      }
}
```
运行时抛出异常 NullPointerException，自动拆箱（unboxing）要求包装类的对象引用必须不

为 null，因此选 D。

6.3.1 ArrayList 的洗牌算法
用下列的代码写一个洗牌算法出来。

```
public static void shuffle(ArrayList<Number> list)
```

提示如下。
我们可以使用 Fisher-Yates 洗牌算法来打乱 ArrayList<Number> 中数字。
Fisher-Yates 洗牌算法的步骤如下：

初始化：从列表的最后一个元素开始，向前遍历列表。
随机选择：对于列表中的每个元素，在它和它之前的所有元素（包括自己）中随机选择一个元素。
交换：将当前元素与随机选中的元素进行交换。
重复：继续向前遍历列表，重复步骤 2 和 3，直到到达列表的开始。
它的特点如下：

该算法的时间复杂度为 O(n)，其中 n 是列表的长度。这是因为每个元素只被访问一次。
每个元素都有相同的机会出现在列表的任何位置，确保了洗牌的随机性和公平性。
算法不需要额外的存储空间，直接在原列表上进行操作，空间复杂度为 O(1)。
最后的代码如下。

```
import java.util.ArrayList;
import java.util.Random;

public class ShuffleArrayList {
    public static void main(String[] args) {
        // 创建一个包含数字的 ArrayList
        ArrayList<Number> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        list.add(8);
        list.add(9);
        list.add(10);

        // 打印原始列表
        System.out.println("Original list: " + list);

        // 调用 shuffle 方法打乱列表
        shuffle(list);
        // 打印打乱后的列表
        System.out.println("Shuffled list: " + list);
    }

    public static void shuffle(ArrayList<Number> list) {
        Random random = new Random(); // 创建一个随机数生成器

        // Fisher-Yates 洗牌算法
        for (int i = list.size() - 1; i > 0; i--) {
            // 生成一个从 0 到 i（包含 i）的随机索引
            int j = random.nextInt(i + 1);

            // 交换索引 i 和 j 处的元素
            Number temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
}
```

6.3.2 ComparableCircle 类

创建一个名为 ComparableCircle 的类，该类继承自 Circle 类并实现了 Comparable 接口。然后，你需要实现 compareTo 方法，以便根据圆的面积来比较两个 ComparableCircle 对象的大小。最后，你需要编写一个测试类来验证 ComparableCircle 类的功能，通过比较两个 ComparableCircle 对象来找出较大的一个。

比较简单的示例如下。

```
// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        double thisArea = this.getArea();
        double otherArea = other.getArea();
        return Double.compare(thisArea, otherArea);
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        if (circle1.compareTo(circle2) > 0) {
            System.out.println("Circle1 is larger");
        } else if (circle1.compareTo(circle2) < 0) {
            System.out.println("Circle2 is larger");
        } else {
            System.out.println("Both circles are equal");
        }
    }
}
```
当然由于圆的特性，所以其实我们可以使用半径去比较。
相关的代码如下。

```
// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        return Double.compare(this.getRadius(), other.getRadius());
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        if (circle1.compareTo(circle2) > 0) {
            System.out.println("Circle1 is larger");
        } else if (circle1.compareTo(circle2) < 0) {
            System.out.println("Circle2 is larger");
        } else {
            System.out.println("Both circles are equal");
        }
    }
}
```
当然我们还可以使用前面的 max 方法去比较这里的面积。
示例如下。

```
import java.util.Date;

// 基类 GeometricObject
public abstract class GeometricObject {
    private String color = "white";
    private boolean filled;
    private java.util.Date dateCreated;

    protected GeometricObject() {
        this.dateCreated = new Date();
    }

    protected GeometricObject(String color, boolean filled) {
        this.dateCreated = new Date();
        this.color = color;
        this.filled = filled;
    }

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public boolean isFilled() {
        return filled;
    }

    public void setFilled(boolean filled) {
        this.filled = filled;
    }
    public java.util.Date getDateCreated() {
        return dateCreated;
    }

    public String toString() {
        return "created on " + dateCreated + "\ncolor: " + color +
               " and filled: " + filled;
    }

    // Abstract method getArea
    public abstract double getArea();

    // Abstract method getPerimeter
    public abstract double getPerimeter();
}

// 圆形类 Circle
public class Circle extends GeometricObject {
    private double radius;

    public Circle() {
        super();
    }

    public Circle(double radius) {
        super();
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public void setRadius(double radius) {
        this.radius = radius;
    }

    @Override
    public double getArea() {
        return radius * radius * Math.PI;
    }

    @Override
    public double getPerimeter() {
        return 2 * radius * Math.PI;
    }

    public double getDiameter() {
        return 2 * radius;
    }
}

// 可比较的圆形类 ComparableCircle
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {
    public ComparableCircle() {
        super();
    }

    public ComparableCircle(double radius) {
        super(radius);
    }

    @Override
    public int compareTo(ComparableCircle other) {
        return Double.compare(this.getRadius(), other.getRadius());
    }
}

// 比较类 Max
public class Max {
    public static Comparable max(Comparable o1, Comparable o2) {
        if (o1.compareTo(o2) > 0)
            return o1;
        else
            return o2;
    }
}

// 测试类 TestComparableCircle
public class TestComparableCircle {
    public static void main(String[] args) {
        ComparableCircle circle1 = new ComparableCircle(5);
        ComparableCircle circle2 = new ComparableCircle(3);

        ComparableCircle maxCircle = (ComparableCircle) Max.max(circle1, circle2);

        System.out.println("The larger circle has radius: " + maxCircle.getRadius());
    }
```

```
}
```

书本示例如下。

```
import java.util.Comparator;

public class GeometricObjectComparator implements Comparator<GeometricObject>, java.io.Serializable {
    public int compare(GeometricObject o1, GeometricObject o2) {
        double area1 = o1.getArea();
        double area2 = o2.getArea();
        if (area1 < area2)
            return -1;
        else if (area1 == area2)
            return 0;
        else
            return 1;
    }
}

import java.util.Comparator;

public class TestComparator {
    public static void main(String[] args) {
        GeometricObject g1 = new Rectangle(5, 5);
        GeometricObject g2 = new Circle(5);

        GeometricObject g = max(g1, g2, new GeometricObjectComparator());
        System.out.println("The area of the larger object is " + g.getArea());
    }

    public static GeometricObject max(GeometricObject g1, GeometricObject g2, Comparator<GeometricObject> c) {
        if (c.compare(g1, g2) > 0)
            return g1;
        else
            return g2;
    }
}
```

6.3.3 MyStack 类的深拷贝

重写 MyStack 类以实现深拷贝。

```
public class MyStack {
    private java.util.ArrayList list = new java.util.ArrayList();
```

```
	public void push(Object o) {
		list.add(o);
	}

	public Object pop() {
		Object o = list.get(getSize() - 1);
		list.remove(getSize() - 1);
		return o;
	}

	public Object peek() {
		return list.get(getSize() - 1);
	}

	public int search(Object o) {
		return list.lastIndexOf(o);
	}

	public boolean isEmpty() {
		return list.isEmpty();
	}

	public int getSize() {
		return list.size();
	}

	public String toString() {
		return "stack: " + list.toString();
	}
}
```

示例代码如下。

```
import java.util.ArrayList;

public class MyStack implements Cloneable {
	private ArrayList<Object> list = new ArrayList<>();

	public void push(Object o) {
		list.add(o);
	}

	public Object pop() {
		if (isEmpty()) {
			return null;
```

```
        }
        Object o = list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        if (isEmpty()) {
            return null;
        }
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }

    @Override
    public MyStack clone() {
        try {
            MyStack cloned = (MyStack) super.clone();
            cloned.list = new ArrayList<>(list);
            return cloned;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError(); // Can't happen
        }
    }
}
```

我们这里重写了 MyStack 类下的 clone() 方法，先调用 super.clone() 方法创建当前对象的一个浅拷贝。这会复制对象的所有字段，但对于对象引用类型的字段，只复制引用，不复制引用的对象。再通过调用 new ArrayList<>(list) 来创建了一个新的 ArrayList 对象，并将原 list 中的所有元素添加到这个新对象中。

6.3.4 判断代码对错

判断下列代码是否会成功编译？如果不能，原因是什么？

6.3.4.1 Animal 类

```
public interface Animal {
    String name; // Data field representing the name of the animal
    void makeSound(); // Abstract method to make the animal sound
}
```

这段代码不会成功编译。原因是在 Java 接口中，你不能直接声明实例字段。

6.3.4.2 MyInterface2 接口

```
public interface MyInterface2 {
    void method1(); // Abstract method without implementation
    void method2(); // Abstract method without implementation
    void method3() {
        // Concrete method with implementation
        System.out.println("Method 3 implementation");
    }
}
```

这段代码不会成功编译。原因是在 Java 接口中的方法默认是抽象的，不能有方法体。

6.3.4.3 MyClass 类

```
public abstract class MyClass {
    public MyClass() {
        System.out.println("Abstract class constructor");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass(); // Error here
    }
}
```

四！！！！！！

2.1 洗牌算法（泛型版）

修改上次的洗牌算法，使其能够处理泛型对象的 ArrayList。

示例代码如下。

```
import java.util.ArrayList;
import java.util.Random;

public class ShuffleArrayList {
    public static void main(String[] args) {
        // 创建一个包含数字的 ArrayList
        ArrayList<Integer> list = new ArrayList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.add(6);
        list.add(7);
        list.add(8);
        list.add(9);
        list.add(10);

        // 打印原始列表
        System.out.println("Original list: " + list);

        // 调用 shuffle 方法打乱列表
        shuffle(list);

        // 打印打乱后的列表
        System.out.println("Shuffled list: " + list);
    }

    public static <E> void shuffle(ArrayList<E> list) {
        Random random = new Random(); // 创建一个随机数生成器

        // Fisher-Yates 洗牌算法
        for (int i = list.size() - 1; i > 0; i--) {
            // 生成一个从 0 到 i（包含 i）的随机索引
            int j = random.nextInt(i + 1);

            // 交换索引 i 和 j 处的元素
            E temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }
}
```

2.2 ArrayList 中的最大元素
编写一个泛型方法，该方法能够找出 ArrayList 中的最大元素。
方法签名如下。

```
public static <E extends Comparable<E>> E max(ArrayList<E> list)
1
```

示例代码如下。

```java
import java.util.ArrayList;

public class MaxElementFinder {
    public static void main(String[] args) {
        // 创建一个包含 Integer 对象的 ArrayList
        ArrayList<Integer> list = new ArrayList<>();
        list.add(10);
        list.add(20);
        list.add(5);
        list.add(30);
        list.add(15);

        // 调用 max 方法找出最大元素
        Integer maxElement = max(list);

        // 打印最大元素
        System.out.println("The maximum element is: " + maxElement);
    }

    public static <E extends Comparable<E>> E max(ArrayList<E> list) {
        if (list == null || list.isEmpty()) {
            throw new IllegalArgumentException("List must not be null or empty");
        }

        // 假设第一个元素是最大的
        E maxElement = list.get(0);

        // 遍历列表中的所有元素
        for (E element : list) {
            // 如果找到更大的元素，更新 maxElement
            if (element.compareTo(maxElement) > 0) {
                maxElement = element;
            }
        }

        // 返回最大元素
        return maxElement;
    }
}
```

2.3 Pair 类

实现一个 Pair 类，它包含两个泛型元素 first 和 second，这两个元素必须是相同的类型 E。同时，你需要开发一个 print() 方法，用于在一行中输出这两个元素。

示例代码如下。

```
public class Pair<E> {
    public E first; // 第一个元素
    public E second; // 第二个元素

    // 构造函数：初始化一个包含两个元素的 Pair
    public Pair(E e, E f) {
        first = e;
        second = f;
    }

    public static void main(String[] args) {
        Pair<Integer> p1 = new Pair<>(1, 85);
        Pair<Integer> p2 = new Pair<>(2, 63);
        print(p1);
        print(p2);
    }

    public static void print(Pair p) {
        System.out.println(p.first + " " + p.second);
    }
}
```

现在要修改这里的代码，使其能够存储不同类型的对象。例如，一个 Pair 可以同时存储一个 Integer 和一个 Double，或者一个 Integer 和一个 String。`

```
public class Pair<E, F> {
    public E first; // 第一个元素
    public F second; // 第二个元素

    // 构造函数：初始化一个包含两个不同类型元素的 Pair
    public Pair(E e, F f) {
        first = e;
        second = f;
    }

    public static void main(String[] args) {
        Pair<Integer, Double> p1 = new Pair<>(1, 85.5);
        Pair<Integer, String> p2 = new Pair<>(2, "good");
        print(p1);
        print(p2);
    }

    public static void print(Pair p) {
        System.out.println(p.first + " " + p.second);
    }
}
```

测试代码如下。

```
public static void main(String[] args) {
    Pair<Integer, Double> p1 = new Pair<>(1, 85.5);
    Pair<Integer, String> p2 = new Pair<>(2, "good");
    print(p1);
    print(p2);
}
```

2.4 类型通配符

下面的代码出示了一个 print 方法，设计用来打印 ArrayList 中的所有元素。这里的代码有问题，需要解释代码不能编译的原因，以及修正方案。

```
import java.util.*;

public class week4test1{
    public static void main(String[] args) {
        ArrayList<Integer> c = new ArrayList<>();
        c.add(3);
        c.add(4);
        c.add(12);
        print(c);
    }
    public static void print(ArrayList<Object> o){
        for (Object e : o)
            System.out.println(e);
    }
}
```

尽管 Integer 是 Object 的子类，但是 ArrayList<Integer> 不是 ArrayList<Object> 的子类型。因此，直接将 ArrayList<Integer> 传递给期望 ArrayList<Object> 的方法会导致编译错误。
为了解决这个问题，可以使用通配符 ? 来表示 print 方法可以接受任何类型的 ArrayList。修正方案如下。

```
import java.util.*;

public class week4test1{
    public static void main(String[] args) {
        ArrayList<Integer> c = new ArrayList<>();
        c.add(3);
        c.add(4);
        c.add(12);
        print(c);
    }
    public static void print(ArrayList<? extends Object> o){
        for (Object e : o)
            System.out.println(e);
    }
}
```

五！！！！！！！！

用 Stack 类实现中缀表达式。
示例代码如下。

```
import java.util.Stack;

public class EvaluateExpression {
    public static void main(String[] args) {
        // Check number of arguments passed
        if (args.length != 1) {
            System.out.println("Usage: java EvaluateExpression \"expression\"");
            System.exit(1);
        }
        try {
            System.out.println(evaluateExpression(args[0]));
        } catch (Exception ex) {
            System.out.println("Wrong expression: " + args[0]);
        }
    }

    /** Evaluate an expression */
    public static int evaluateExpression(String expression) {
        // Create operandStack to store operands
        Stack<Integer> operandStack = new Stack<>();
        // Create operatorStack to store operators
        Stack<Character> operatorStack = new Stack<>();
        // Insert blanks around (, ), +, -, /, and *
        expression = insertBlanks(expression);
        // Extract operands and operators
        String[] tokens = expression.split(" ");
        // Phase 1: Scan tokens
        for (String token : tokens) {
            if (token.length() == 0) // Blank space
```

```
                continue; // Back to the while loop to extract the next token
            else if (token.charAt(0) == '+' || token.charAt(0) == '-') {
                // Process all +, -, *, / in the top of the operator stack
                while (!operatorStack.isEmpty() &&
                        (operatorStack.peek() == '+' ||
                        operatorStack.peek() == '-' ||
                        operatorStack.peek() == '*' ||
                        operatorStack.peek() == '/')) {
                    processAnOperator(operandStack, operatorStack);
                }
                // Push the + or - operator into the operator stack
                operatorStack.push(token.charAt(0));
            } else if (token.charAt(0) == '*' || token.charAt(0) == '/') {
                // Process all *, / in the top of the operator stack
                while (!operatorStack.isEmpty() &&
                        (operatorStack.peek() == '*' ||
                         operatorStack.peek() == '/')) {
                    processAnOperator(operandStack, operatorStack);
                }
                // Push the * or / operator into the operator stack
                operatorStack.push(token.charAt(0));
            } else if (token.trim().charAt(0) == '(') {
                operatorStack.push('('); // Push '(' to stack
            } else if (token.trim().charAt(0) == ')') {
                // Process all the operators in the stack until seeing '('
                while (operatorStack.peek() != '(') {
                    processAnOperator(operandStack, operatorStack);
                }
                operatorStack.pop(); // Pop the '(' symbol from the stack
            } else { // An operand scanned
                // Push an operand to the stack
                operandStack.push(new Integer(token)));
            }
        }
        // Phase 2: process all the remaining operators in the stack
        while (!operatorStack.isEmpty()) {
            processAnOperator(operandStack, operatorStack);
        }
        // Return the result
        return operandStack.pop();
    }

    /** Process one operator: Take an operator from operatorStack and
     * apply it on the operands in the operandStack */
    public static void processAnOperator(Stack<Integer> operandStack, Stack<Character> operatorStack) {
        char op = operatorStack.pop();
        int op1 = operandStack.pop();
        int op2 = operandStack.pop();
        if (op == '+')
            operandStack.push(op2 + op1);
        else if (op == '-')
            operandStack.push(op2 - op1);
        else if (op == '*')
            operandStack.push(op2 * op1);
        else if (op == '/')
            operandStack.push(op2 / op1);
    }

    public static String insertBlanks(String s) {
        String result = "";
        for (int i = 0; i < s.length(); i++) {
            if (s.charAt(i) == '(' || s.charAt(i) == ')' ||
                    s.charAt(i) == '+' || s.charAt(i) == '-' ||
                    s.charAt(i) == '*' || s.charAt(i) == '/') {
                result += " " + s.charAt(i) + " ";
            } else
                result += s.charAt(i);
        }
        return result;
    }
}
```

5.2.1 比较迭代器（Iterator）和使用 get() 方法

创建一个包含 100,000 个 Integer 对象的 LinkedList。
使用 System.currentTimeMillis() 创建一个计时器.
使用 get() 方法遍历 LinkedList。
使用迭代器遍历 LinkedList。
输出上述两个时间值。
示例代码如下。

```
import java.util.LinkedList;
import java.util.Iterator;
import java.util.List;

public class ListIteratorTest {
    public static void main(String[] args) {
        // 创建一个包含 100,000 个 Integer 对象的 LinkedList
        List<Integer> list = new LinkedList<>();
        for (int i = 0; i < 100000; i++) {
            list.add(i);
        }

        // 创建一个计时器
        long startTimeGet = System.currentTimeMillis();

        // 使用 get()方法遍历 LinkedList
        for (int i = 0; i < list.size(); i++) {
            list.get(i);
        }
        long endTimeGet = System.currentTimeMillis();
        System.out.println("Time consumed by get(): " + (endTimeGet - startTimeGet) + " ms");

        // 创建一个计时器
        long startTimeIterator = System.currentTimeMillis();

        // 使用迭代器遍历 LinkedList
        Iterator<Integer> iterator = list.iterator();
        while (iterator.hasNext()) {
            iterator.next();
        }
        long endTimeIterator = System.currentTimeMillis();
        System.out.println("Time consumed by iterator(): " + (endTimeIterator - startTimeGet)
+ " ms");
    }
}
```

5.2.2 优先队列

创建两个优先队列（Priority Queues），并展示它们的并集（union）、差集（difference）和交集（intersection）。

首先，创建两个 PriorityQueue 实例，并向它们添加指定的元素。
使用 PriorityQueue 的 addAll() 方法将第二个优先队列中的所有元素添加到第一个优先队列中，从而得到两个队列的并集。
使用 PriorityQueue 的 removeAll() 方法从第一个优先队列中移除第二个优先队列中存在的所有元素，从而得到两个队列的差集。
使用 PriorityQueue 的 retainAll() 方法从第一个优先队列中移除第二个优先队列中不存在的所有元素，从而得到两个队列的交集。
示例代码如下。

```
import java.util.PriorityQueue;
```

```
public class PriorityQueueDemo {
        public static void main(String[] args) {
                // 创建两个优先队列并添加元素
                PriorityQueue<String> queue1 = new PriorityQueue<>();
                PriorityQueue<String> queue2 = new PriorityQueue<>();

                String[] elements1 = {"George", "Jim", "John", "Blake", "Kevin", "Michael"};
                String[] elements2 = {"George", "Katie", "Kevin", "Michelle", "Ryan"};

                for (String element : elements1) {
                        queue1.add(element);
                }
                for (String element : elements2) {
                        queue2.add(element);
                }

                // 获取并集
                PriorityQueue<String> union = new PriorityQueue<>(queue1);
                union.addAll(queue2);
                System.out.println("Union: " + union);

                // 获取差集
                queue1.removeAll(queue2);
                System.out.println("Difference (queue1): " + queue1);

                // 获取交集
                queue2.retainAll(queue1);
                System.out.println("Intersection (queue2): " + queue2);
        }
}
```

5.2.3 Stack 类

使用 Stack 识别并验证圆括号 ()、花括号 {} 和方括号 [] 的正确配对，确保它们不会重叠。例如 (a{b)} 是非法的。

示例代码如下。

```
import java.util.Stack;

public class BracketChecker {
        public static void main(String[] args) {
                // 从键盘读取一行程序源代码
                java.util.Scanner scanner = new java.util.Scanner(System.in);
                System.out.println("Enter a program source-code line:");
                String input = scanner.nextLine();
        // 检查括号是否配对
        System.out.println(isPaired(input) ? "Paired" : "Unpaired");
    }

    public static boolean isPaired(String expression) {
        Stack<Character> stack = new Stack<>();
        for (int i = 0; i < expression.length(); i++) {
            char ch = expression.charAt(i);
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else if (ch == ')' || ch == '}' || ch == ']') {
                if (stack.isEmpty() || stack.pop() != getMatching(ch)) {
                    return false;
                }
            }
        }
        return stack.isEmpty();
    }

    public static char getMatching(char ch) {
        switch (ch) {
            case '(': return ')';
            case '{': return '}';
            case '[': return ']';
            case ')': return '(';
            case '}': return '{';
            case ']': return '[';
            default: return 0;
        }
    }
}
```

六！！！！！！！！！

3.1 基础练习

假设我们有两个集合 s1 = [1, 2, 5], s2 = [2, 3, 6]。
运行 s1.addAll(s2) 之后，s1 变为[1, 2, 5, 3, 6]，而 s2 不变，仍为[2, 3, 6]。
运行 s1.removeAll(s2) 之后，s1 变为[1, 5]。
运行 s1.retainAll(s2) 之后，s1 变为[]。

3.2 进阶练习

3.2.1 获取 TreeSet 的第一个元素和最后一个元素

示例代码如下。

```
import java.util.*;

public class Test {
        public static void main(String[] args) throws Exception {
                TreeSet<String> set = new TreeSet<>();
                set.add("Red");
                set.add("Yellow");
                set.add("Green");
                set.add("Blue");
                System.out.println(set.first());
                System.out.println(set.last());
        }
}
```

3.2.2 深拷贝版 MyStack 类

MyStack 类代码如下。

```
import java.util.ArrayList;

public class MyStack {
        private ArrayList list = new ArrayList();

        public void push(Object o) {
              list.add(o);
      }

      public Object pop() {
              Object o = list.get(getSize() - 1);
              list.remove(getSize() - 1);
              return o;
      }

      public Object peek() {
              return list.get(getSize() - 1);
      }

      public int search(Object o) {
              return list.lastIndexOf(o);
      }

      public boolean isEmpty() {
              return list.isEmpty();
      }

      public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
}
```
修改后的示例如下。

```
import java.util.ArrayList;
import java.util.List;

public class MyStack implements Cloneable {
    private List<Object> list = new ArrayList<>();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
    // 实现 Cloneable 接口的 clone 方法，进行深拷贝
    @Override
    public MyStack clone() throws CloneNotSupportedException {
        MyStack clonedStack = (MyStack) super.clone();
        clonedStack.list = new ArrayList<>(list); // 创建 list 的一个新副本
        return clonedStack;
    }
}
```

测试代码如下。

```
public class DeepCopyTest {
    public static void main(String[] args) {
        MyStack originalStack = new MyStack();
        originalStack.push("Apple");
        originalStack.push("Banana");
        originalStack.push("Cherry");

        try {
            MyStack clonedStack = originalStack.clone();
            System.out.println("Original stack: " + originalStack);
            System.out.println("Cloned stack: " + clonedStack);

            // 修改原始栈
            originalStack.push("Date");

            // 显示修改后的结果
            System.out.println("Modified original stack: " + originalStack);
            System.out.println("Cloned stack remains unchanged: " + clonedStack);
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
    }
}
```

3.2.3 深拷贝版 MyStack 类

原来的 MyStack 代码如下。

在这里插入代码片

深拷贝版如下。

```
import java.util.ArrayList;

public class MyStack {
    private ArrayList list = new ArrayList();

    public void push(Object o) {
        list.add(o);
    }

    public Object pop() {
        Object o = list.get(getSize() - 1);
        list.remove(getSize() - 1);
        return o;
    }

    public Object peek() {
        return list.get(getSize() - 1);
    }

    public int search(Object o) {
        return list.lastIndexOf(o);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    public String toString() {
        return "stack: " + list.toString();
    }
}
```

3.2.4 找出出现次数最多的整数

编写一个程序来读取一系列整数，并找出出现次数最多的整数。输入以 0 结束。如果多个整数出现次数最多，则应该报告所有这些整数。
这个代码与之前大二的 Java 练习类似，使用一个 Map 来跟踪每个整数的出现次数。
示例代码如下。

```
import java.util.*;

public class MostFrequentNumber {
    public static void main(String[] args) {
        Map<Integer, Integer> countMap = new HashMap<>();
        Scanner scanner = new Scanner(System.in);

        while (true) {
            int num = scanner.nextInt();
            if (num == 0) {
                break;
            }
            if (countMap.containsKey(num)) {
                countMap.put(num, countMap.get(num) + 1);
            } else {
                countMap.put(num, 1);
            }
        }

        int maxCount = 0;
        for (int value : countMap.values()) {
            if (value > maxCount) {
                maxCount = value;
            }
        }

        System.out.println("Numbers with most occurrences:");
        for (Map.Entry<Integer, Integer> entry : countMap.entrySet()) {
            if (entry.getValue() == maxCount) {
                System.out.println(entry.getKey());
            }
        }
    }
}
```

七！！！！！！！

2.1 基础练习

1.大 O 符号的主要目的是什么?

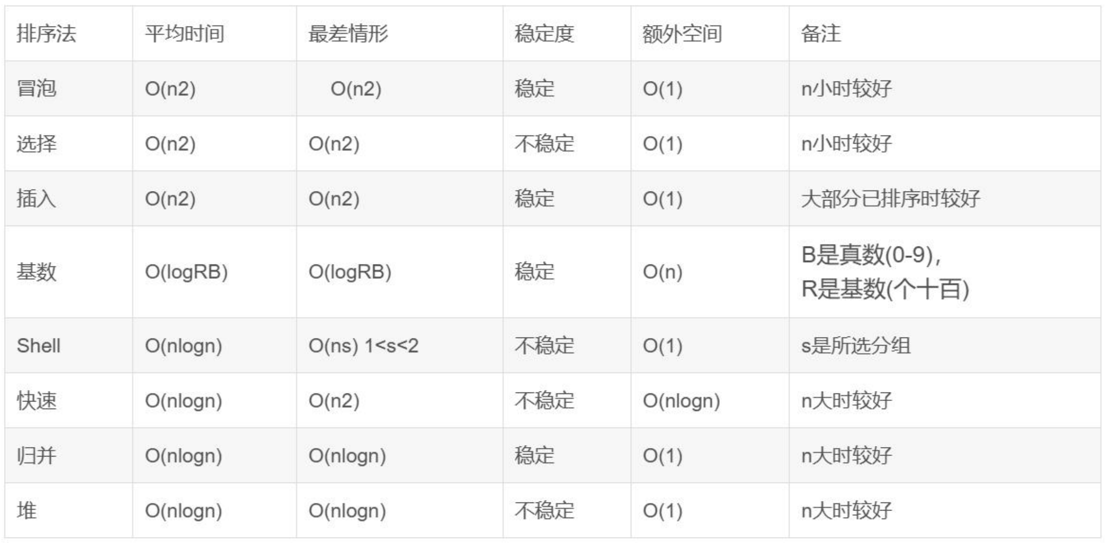
A.测量实际执行时间。

B.估计最坏情况下的内存使用量。

C.描述算法随着输入规模增加的增长速率。

D.计算在一台机器上的平均执行时间。

答案是 C。



排序法	平均时间	最差情形	稳定度	额外空间	备注
冒泡	O(n2)	O(n2)	稳定	O(1)	n小时较好
选择	O(n2)	O(n2)	不稳定	O(1)	n小时较好
插入	O(n2)	O(n2)	稳定	O(1)	大部分已排序时较好
基数	O(logRB)	O(logRB)	稳定	O(n)	B是真数(0-9), R是基数(个十百)
Shell	O(nlogn)	O(ns) 1<s<2	不稳定	O(1)	s是所选分组
快速	O(nlogn)	O(n2)	不稳定	O(nlogn)	n大时较好
归并	O(nlogn)	O(nlogn)	稳定	O(1)	n大时较好
堆	O(nlogn)	O(nlogn)	不稳定	O(1)	n大时较好

2.在使用大 O 符号时，哪些因素可以被忽略？
A.输入规模。
B.主导项。
C.递归调用次数。
D.常数乘数和低阶项。

答案是 D。

3.哪种算法具有对数时间复杂度？
A.Linear search（线性搜索）。
B.Binary search（二分搜索）。
C.Insertion sort（插入排序）。
D.Selection sort（选择排序）。

答案是 B。

4.递归斐波那契算法与其动态规划版本的性能对比？
A.它们具有相同的效率。
B.递归归版本更快。
C.动态规划避免重复计算并且更快。
D.递归归版本使用更少的内存并且更好。

答案是 C。

5.下列代码的时间复杂度是多少？

```
for (int i = 0; i < n; i++) {
        System.out.print('*');
}
```
答案是 O (n) O(n)O(n)。

```
for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
                System.out.print('*');
        }
}
```

答案是 O (n 2) O(n^2)O(n

2

）。

```
for (int k = 0; k < n; k++) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        System.out.print('*');
                }
        }
}
```

答案是 O (n 3) O(n^3)O(n

3

）。

```
for (int k = 0; k < 10; k++) {
        for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                        System.out.print('*');
                }
        }
}
```

答案是 O (n 2) O(n^2)O(n

2

）。

5.下列代码最糟糕情况下的时间复杂度是多少？

```
for (int i = 1; i < list.length; i++) {
        if (list[i] > list[i + 1]) {
                // 交换 list[i] 和 list[i + 1]
                int temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
        }
}
```

答案是 O (n 2) O(n^2)O(n

2

），这是一个冒泡排序。

2.2 进阶练习
如何使用分治法（Divide-and-Conquer）来找出列表中的最大数？
示例代码如下。

```
public class DivideAndConquerMax {
    public static int findMax(int[] list) {
        return findMaxRecursive(list, 0, list.length - 1);
    }

    private static int findMaxRecursive(int[] list, int left, int right) {
        if (left == right) {
            return list[left]; // 基本情况：只有一个元素
        } else if (left + 1 == right) {
            return Math.max(list[left], list[right]); // 两个元素，直接比较
        } else {
            int mid = (left + right) / 2;
            int maxLeft = findMaxRecursive(list, left, mid);
            int maxRight = findMaxRecursive(list, mid + 1, right);
            return Math.max(maxLeft, maxRight); // 比较左右两部分的最大值
        }
    }

    public static void main(String[] args) {
        int[] list = {178, 33, 4, 2, -3, 5};
        System.out.println("The largest number is: " + findMax(list));
    }
}
```

八！！！！！！！！

3.1 基础练习
1.给定列表：{2, 9, 5, 4, 8, 1}
冒泡排序算法第一步后，结果是什么？

答案是{2, 5, 4, 8, 1, 9}。因为 2 < 9 2 < 92<9，所以不需要交换。

2.对于堆来问号处可以是多少？

答案是 28/29。

3.2 进阶练习

3.2.1 编写两个泛型方法来实现归并排序（Merge Sort）

第一个方法使用 Comparable 接口进行排序，第二个方法使用 Comparator 接口进行排序。示例代码如下。

```
import java.util.Comparator;

public class MergeSortExample {

    // 使用 Comparable 接口进行排序
    public static <E extends Comparable<E>> void mergeSort(E[] list) {
        if (list.length > 1) {
            int middle = list.length / 2;
            @SuppressWarnings("unchecked")
            E[] firstHalf = (E[]) new Comparable[middle];
            @SuppressWarnings("unchecked")
            E[] secondHalf = (E[]) new Comparable[list.length - middle];

            System.arraycopy(list, 0, firstHalf, 0, middle);
            System.arraycopy(list, middle, secondHalf, 0, list.length - middle);

            mergeSort(firstHalf);
            mergeSort(secondHalf);

            merge(list, firstHalf, secondHalf);
        }
    }

    // 使用 Comparator 接口进行排序
    public static <E> void mergeSort(E[] list, Comparator<? super E> comparator) {
        if (list.length > 1) {
            int middle = list.length / 2;
            @SuppressWarnings("unchecked")
            E[] firstHalf = (E[]) new Comparable[middle];
            @SuppressWarnings("unchecked")
            E[] secondHalf = (E[]) new Comparable[list.length - middle];

            System.arraycopy(list, 0, firstHalf, 0, middle);
            System.arraycopy(list, middle, secondHalf, 0, list.length - middle);

            mergeSort(firstHalf, comparator);
            mergeSort(secondHalf, comparator);

            merge(list, firstHalf, secondHalf, comparator);
        }
    }

    // 合并两个已排序的子数组
    private static <E extends Comparable<E>> void merge(E[] list, E[] firstHalf, E[] secondHalf) {
        int i = 0, j = 0, k = 0;
        while (i < firstHalf.length && j < secondHalf.length) {
            if (firstHalf[i].compareTo(secondHalf[j]) <= 0) {
                list[k++] = firstHalf[i++];
            } else {
                list[k++] = secondHalf[j++];
            }
        }
        while (i < firstHalf.length) {
            list[k++] = firstHalf[i++];
        }
        while (j < secondHalf.length) {
            list[k++] = secondHalf[j++];
        }
    }

    // 合并两个已排序的子数组，使用 Comparator
    private static <E> void merge(E[] list, E[] firstHalf, E[] secondHalf, Comparator<? super E>
comparator) {
        int i = 0, j = 0, k = 0;
        while (i < firstHalf.length && j < secondHalf.length) {
            if (comparator.compare(firstHalf[i], secondHalf[j]) <= 0) {
                list[k++] = firstHalf[i++];
            } else {
                list[k++] = secondHalf[j++];
            }
        }
        while (i < firstHalf.length) {
            list[k++] = firstHalf[i++];
        }
        while (j < secondHalf.length) {
            list[k++] = secondHalf[j++];
        }
    }

    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        System.out.println("Original list: " + java.util.Arrays.toString(list));
        mergeSort(list);
        System.out.println("Sorted list using Comparable: " + java.util.Arrays.toString(list));
        String[] stringList = {"banana", "apple", "cherry", "date"};
        System.out.println("Original string list: " + java.util.Arrays.toString(stringList));
        mergeSort(stringList, (a, b) -> b.compareTo(a));
        System.out.println("Sorted string list using Comparator: " +
java.util.Arrays.toString(stringList));
    }
}
```

3.2.2 编写两个泛型方法来实现插入排序（Insertion Sort）

第一个方法使用 Comparable 接口进行排序，第二个方法使用 Comparator 接口进行排序。示例代码如下。

```
import java.util.Comparator;

public class InsertionSortExample {

    // 使用 Comparable 接口进行排序
    public static <E extends Comparable<E>> void insertionSort(E[] list) {
        for (int i = 1; i < list.length; i++) {
            E current = list[i];
            int j = i - 1;
            while (j >= 0 && list[j].compareTo(current) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = current;
        }
    }

    // 使用 Comparator 接口进行排序
    public static <E> void insertionSort(E[] list, Comparator<? super E> comparator) {
        for (int i = 1; i < list.length; i++) {
            E current = list[i];
            int j = i - 1;
            while (j >= 0 && comparator.compare(list[j], current) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = current;
        }
    }

    public static void main(String[] args) {
        Integer[] list = {2, 3, 2, 5, 6, 1, -2, 3, 14, 12};
        System.out.println("Original list: " + java.util.Arrays.toString(list));
        insertionSort(list);
        System.out.println("Sorted list using Comparable: " + java.util.Arrays.toString(list));

        String[] stringList = {"banana", "apple", "cherry", "date"};
        System.out.println("Original string list: " + java.util.Arrays.toString(stringList));
        insertionSort(stringList, (a, b) -> b.compareTo(a));
        System.out.println("Sorted    string    list    using    Comparator:    "    +
java.util.Arrays.toString(stringList));
    }
}
```

九！！！！！！！！！！！！！

2.1 二分图（Bipartite Graph）
二分图是指图的顶点可以被分成两个互不相交的集合，使得同一集合内的任意两个顶点之间没有边相连。
我们现在使用广度优先搜索（BFS）来判断一个图是否是二分图（Bipartite Graph）。
示例代码如下。

```
import java.util.*;

public class Graph {
    private int V; // 图的顶点数
    private List<List<Integer>> adj; // 邻接表表示图

    // 构造函数
    public Graph(int V) {
        this.V = V;
        adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }

    // 添加边
    public void addEdge(int v, int w) {
        adj.get(v).add(w);
        adj.get(w).add(v); // 无向图，添加双向边
    }

    // 判断图是否是二分图
    public boolean isBipartite() {
        // 颜色数组，-1 表示未访问，0 和 1 表示两种颜色
        int[] color = new int[V];
        Arrays.fill(color, -1);

        // 遍历所有顶点，处理未访问的顶点
        for (int start = 0; start < V; start++) {
            if (color[start] == -1) { // 如果当前顶点未访问
                if (!bfs(start, color)) {
                    return false; // 如果从某个顶点开始的 BFS 发现不是二分图，直接返
回 false
                }
            }
        }
        return true; // 所有连通分量都是二分图
    }

    // BFS 辅助方法
    private boolean bfs(int start, int[] color) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        color[start] = 0; // 给起始顶点染色为 0

        while (!queue.isEmpty()) {
            int u = queue.poll();

            // 遍历 u 的所有邻接顶点
            for (int v : adj.get(u)) {
                // 如果邻接顶点 v 未访问，染与 u 不同的颜色，并加入队列
                if (color[v] == -1) {
                    color[v] = 1 - color[u];
                    queue.add(v);
                } else if (color[v] == color[u]) {
                    // 如果邻接顶点 v 已访问且与 u 颜色相同，说明不是二分图
                    return false;
                }
            }
        }
        return true; // BFS 完成且未发现冲突
    }

    public static void main(String[] args) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(1, 2);
        g.addEdge(2, 3);
        g.addEdge(3, 0);

        System.out.println("Is the graph bipartite? " + g.isBipartite());
    }
}
```

2.2 加权图的邻接矩阵表示

示例代码如下。

```
import java.util.*;

public class WeightedGraph {
    private int V; // 图的顶点数
    private List<List<Edge>> adj; // 邻接表表示图

    // 边的类
    static class Edge {
        int dest; // 目的顶点
        double weight; // 边的权重

        public Edge(int dest, double weight) {
            this.dest = dest;
            this.weight = weight;
        }
    }

    // 构造函数
    public WeightedGraph(int V) {
        this.V = V;
        adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }
    }

    // 添加边
    public void addEdge(int src, int dest, double weight) {
        adj.get(src).add(new Edge(dest, weight));
        adj.get(dest).add(new Edge(src, weight)); // 无向图，添加双向边
    }

    // 生成邻接矩阵
    public double[][] getAdjacentMatrix() {
        double[][] matrix = new double[V][V];
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                matrix[i][j] = Double.POSITIVE_INFINITY; // 初始化为正无穷
            }
        }
        for (int i = 0; i < V; i++) {
            for (Edge edge : adj.get(i)) {
                matrix[i][edge.dest] = edge.weight; // 设置权重
            }
        }
        return matrix;
    }

    public static void main(String[] args) {
        WeightedGraph g = new WeightedGraph(4);
        g.addEdge(0, 1, 1.0);
        g.addEdge(1, 2, 2.0);
        g.addEdge(2, 3, 3.0);
        g.addEdge(3, 0, 4.0);

        double[][] matrix = g.getAdjacentMatrix();
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

2.3 使用邻接矩阵实现 Prim 算法

与之前使用邻接表的实现不同，这次需要使用邻接矩阵来表示图。
示例代码如下。

```
import java.util.*;

public class WeightedGraph {
    private int V; // 图的顶点数
    private double[][] adjMatrix; // 邻接矩阵

    // 构造函数
    public WeightedGraph(int V) {
        this.V = V;
        this.adjMatrix = new double[V][V];
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (i != j) {
                    adjMatrix[i][j] = Double.POSITIVE_INFINITY; // 初始化为正无穷
                }
            }
        }
    }

    // 添加边
    public void addEdge(int src, int dest, double weight) {
        adjMatrix[src][dest] = weight;
        adjMatrix[dest][src] = weight; // 无向图，添加双向边
    }

    // 普里姆算法
    public List<Edge> primMST() {
        boolean[] inMST = new boolean[V]; // 标记顶点是否在 MST 中
        double[] key = new double[V]; // 最小权重边的权重
        int[] parent = new int[V]; // 父节点数组，用于记录 MST 的边

        // 初始化
        Arrays.fill(key, Double.POSITIVE_INFINITY);
        key[0] = 0; // 从顶点 0 开始
        parent[0] = -1; // 顶点 0 没有父节点

        for (int count = 0; count < V - 1; count++) {
            // 找到不在 MST 中且 key 值最小的顶点
            int u = minKey(key, inMST);
            inMST[u] = true; // 将该顶点加入 MST

            // 更新相邻顶点的 key 值
            for (int v = 0; v < V; v++) {
                if (!inMST[v] && adjMatrix[u][v] < Double.POSITIVE_INFINITY &&
                    adjMatrix[u][v] < key[v]) {
                    parent[v] = u;
                    key[v] = adjMatrix[u][v];
                }
            }
        }

        // 构建 MST 的边列表
        List<Edge> mstEdges = new ArrayList<>();
        for (int i = 1; i < V; i++) {
            mstEdges.add(new Edge(parent[i], i, adjMatrix[parent[i]][i]));
        }
        return mstEdges;
    }

    // 辅助方法：找到不在 MST 中且 key 值最小的顶点
    private int minKey(double[] key, boolean[] inMST) {
        double min = Double.POSITIVE_INFINITY;
        int minIndex = -1;
        for (int v = 0; v < V; v++) {
            if (!inMST[v] && key[v] < min) {
                min = key[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    // 边的类
    public static class Edge {
        int src;
        int dest;
        double weight;

        public Edge(int src, int dest, double weight) {
            this.src = src;
            this.dest = dest;
            this.weight = weight;
        }

        @Override
        public String toString() {
            return "(" + src + " - " + dest + ", weight = " + weight + ")";
        }
    }

    public static void main(String[] args) {
        WeightedGraph g = new WeightedGraph(5);
        g.addEdge(0, 1, 2.0);
        g.addEdge(0, 3, 6.0);
        g.addEdge(1, 2, 3.0);
        g.addEdge(1, 3, 8.0);
        g.addEdge(1, 4, 5.0);
        g.addEdge(2, 4, 7.0);
        g.addEdge(3, 4, 9.0);
        List<Edge> mstEdges = g.primMST();
        System.out.println("Edges in the Minimum Spanning Tree:");
        for (Edge edge : mstEdges) {
            System.out.println(edge);
        }
    }
}
```

十！！！！！！！

2.1 非递归的中序遍历方法
实现一个非递归的中序遍历方法，并且编写一个测试程序来演示这个方法。
示例代码如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;
    private int size;

    public BST() {
        root = null;
        size = 0;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            size++;
            return true;
        } else {
            TreeNode<E> current = root;
            TreeNode<E> parent = null;
            while (current != null) {
                parent = current;
                if (element.compareTo(current.element) < 0) {
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element);
            } else {
                parent.right = new TreeNode<>(element);
            }
            size++;
            return true;
        }
    }

    public void nonRecursiveInorder() {
        Stack<TreeNode<E>> stack = new Stack<>();
        TreeNode<E> current = root;
        while (current != null || !stack.isEmpty()) {
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            current = stack.pop();
            System.out.print(current.element + " ");
            current = current.right;
        }
    }
}

public class TestBST {
    public static void main(String[] args) {
        BST<Integer> bst = new BST<>();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter 10 integers:");
        for (int i = 0; i < 10; i++) {
            System.out.print("Enter integer " + (i + 1) + ": ");
            int number = scanner.nextInt();
            bst.insert(number);
        }

        System.out.println("\nInorder traversal (non-recursive):");
        bst.nonRecursiveInorder();

        scanner.close();
    }
}
```

2.2 非递归的前序遍历方法

实现一个非递归的前序遍历方法，并且编写一个测试程序来演示这个方法。
示例代码如下。

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
```

```
                return true;
            } else {
                TreeNode<E> current = root;
                while (true) {
                    if (element.compareTo(current.element) < 0) {
                        if (current.left == null) {
                            current.left = new TreeNode<>(element);
                            return true;
                        }
                        current = current.left;
                    } else if (element.compareTo(current.element) > 0) {
                        if (current.right == null) {
                            current.right = new TreeNode<>(element);
                            return true;
                        }
                        current = current.right;
                    } else {
                        return false; // Duplicate element
                    }
                }
            }
        }

        public void nonRecursivePreorder() {
            Stack<TreeNode<E>> stack = new Stack<>();
            TreeNode<E> current = root;

            while (current != null || !stack.isEmpty()) {
                while (current != null) {
                    System.out.print(current.element + " ");
                    stack.push(current);
                    current = current.left;
                }
                current = stack.pop();
                current = current.right;
            }
        }
    }

    public class TestBST {
        public static void main(String[] args) {
            BST<Integer> bst = new BST<>();
            Scanner scanner = new Scanner(System.in);
            System.out.println("Enter 10 integers:");
            for (int i = 0; i < 10; i++) {
                System.out.print("Enter integer " + (i + 1) + ": ");
                int number = scanner.nextInt();
                bst.insert(number);
            }

            System.out.println("\nPreorder traversal (non-recursive):");
            bst.nonRecursivePreorder();

            scanner.close();
        }
    }
```

2.3 返回二叉树中叶子节点的数量

定义一个新的类 BSTWithNumberOfLeaves，它继承自 BST 类，并添加一个新的方法 getNumberOfLeaves() 来返回二叉树中叶子节点的数量。

示例代码如下。

```
class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
        this.left = null;
        this.right = null;
    }
}

class BST<E extends Comparable<E>> {
    protected TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            return true;
        } else {
            TreeNode<E> current = root;
            TreeNode<E> parent = null;
            while (current != null) {
                parent = current;
                if (element.compareTo(current.element) < 0) {
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
            if (element.compareTo(parent.element) < 0) {
                parent.left = new TreeNode<>(element);
            } else {
                parent.right = new TreeNode<>(element);
            }
            return true;
        }
    }

    // 其他必要的 BST 方法...
}

class BSTWithNumberOfLeaves<E extends Comparable<E>> extends BST<E> {
    public int getNumberOfLeaves() {
        return countLeaves(root);
    }

    private int countLeaves(TreeNode<E> node) {
        if (node == null) {
            return 0;
        }
        if (node.left == null && node.right == null) {
            return 1;
        }
        return countLeaves(node.left) + countLeaves(node.right);
    }
}

public class TestBST {
    public static void main(String[] args) {
        BSTWithNumberOfLeaves<Integer> bst = new BSTWithNumberOfLeaves<>();
        // 假设这里插入了一些元素
        // ...
        System.out.println("Number of leaves: " + bst.getNumberOfLeaves());
    }
}
```

2.4 前序遍历的迭代器

添加一个名为 preorderIterator 的方法，该方法返回一个迭代器，用于以前序遍历的方式遍历二叉搜索树（BST）中的元素。

示例代码如下。

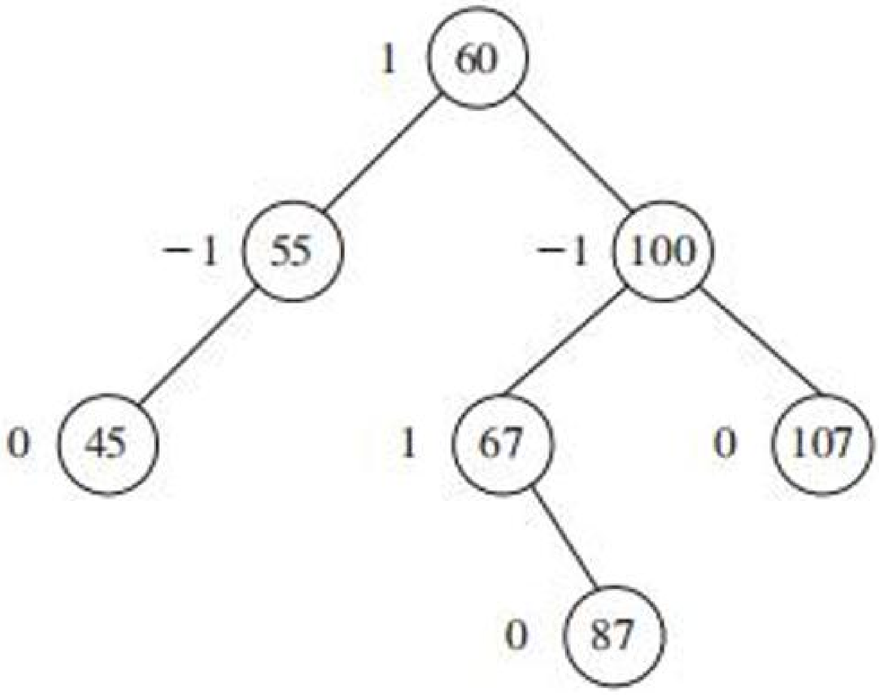
```
import java.util.Iterator;
import java.util.Stack;

class TreeNode<E extends Comparable<E>> {
    E element;
    TreeNode<E> left;
    TreeNode<E> right;

    public TreeNode(E element) {
        this.element = element;
    }
}

class BST<E extends Comparable<E>> {
    private TreeNode<E> root;

    public BST() {
        root = null;
    }

    public boolean insert(E element) {
        if (root == null) {
            root = new TreeNode<>(element);
            return true;
        } else {
            TreeNode<E> current = root;
            while (true) {
                if (element.compareTo(current.element) < 0) {
                    if (current.left == null) {
                        current.left = new TreeNode<>(element);
                        return true;
                    }
                    current = current.left;
                } else if (element.compareTo(current.element) > 0) {
                    if (current.right == null) {
                        current.right = new TreeNode<>(element);
                        return true;
                    }
                    current = current.right;
                } else {
                    return false; // Duplicate element
                }
            }
        }
    }

    // 非递归前序遍历的辅助方法
    private void preorderHelper(TreeNode<E> node, Stack<TreeNode<E>> stack) {
        if (node != null) {
            stack.push(node);
        }
    }

    // 返回前序遍历的迭代器
    public java.util.Iterator<E> preorderIterator() {
        Stack<TreeNode<E>> stack = new Stack<>();
        preorderHelper(root, stack);
        return new PreorderIterator(stack);
    }

    // 实现前序遍历的迭代器
    private class PreorderIterator implements Iterator<E> {
        private Stack<TreeNode<E>> stack;

        public PreorderIterator(Stack<TreeNode<E>> stack) {
            this.stack = stack;
        }

        @Override
        public boolean hasNext() {
            return !stack.isEmpty();
        }

        @Override
        public E next() {
            if (!hasNext()) {
                throw new java.util.NoSuchElementException();
            }
            TreeNode<E> node = stack.pop();
            preorderHelper(node.right, stack); // 先右后左，保证前序
```

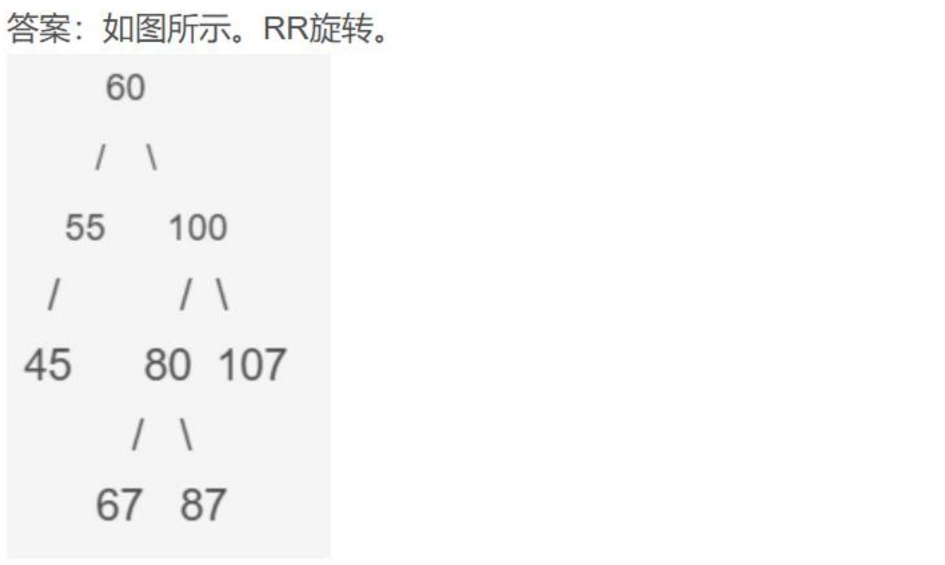
```
                        preorderHelper(node.left, stack);
                        return node.element;
                }
        }
}
```

十一！！！！！！

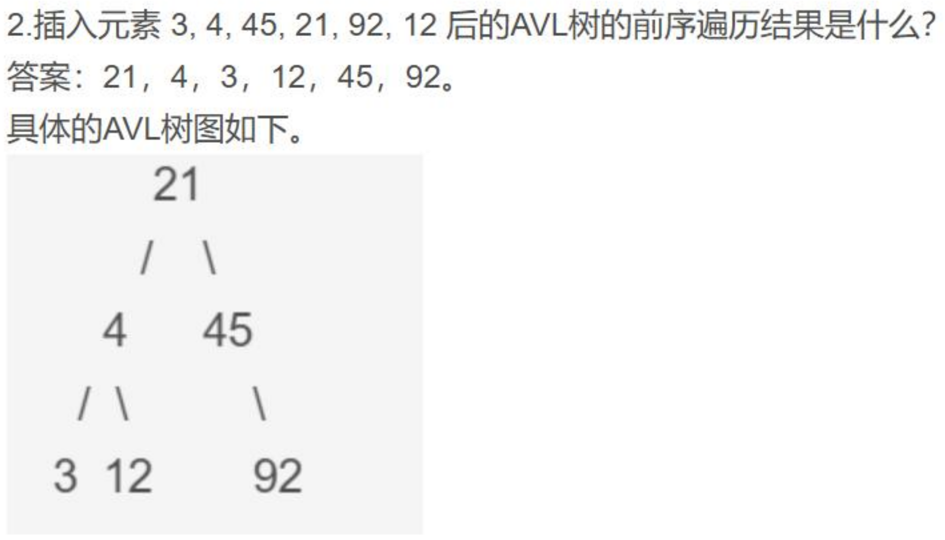
3.1 基础练习

1.这个 AVL 树添加 80 后的结果是什么？进行什么旋转实现的平衡？




答案：如图所示。RR旋转。

```
       60
      /  \
    55    100
   /      / \
45     80  107
      /  \
    67   87
```


2.插入元素 3, 4, 45, 21, 92, 12 后的AVL树的前序遍历结果是什么?

答案：21，4，3，12，45，92。

具体的AVL树图如下。

```
        21
       /  \
     4     45
    / \      \
   3  12      92
```

3.2.1 开放寻址的哈希映射

创建一个名为 MyOpenAddressingMap 的新类，该类使用开放寻址和二次探测来实现 MyMap 接口。哈希函数应使用 f(key) = key % size，其中 size 是哈希表的大小，初始哈希表大小为 4。当负载因子超过阈值（0.5）时，哈希表的大小翻倍。

示例代码如下。

```
import java.util.ArrayList;

import java.util.Arrays;


interface MyMap<K, V> {
    void put(K key, V value);
    V get(K key);
    boolean containsKey(K key);
    void remove(K key);
    int size();
    boolean isEmpty();
}

class MyOpenAddressingMap<K, V> implements MyMap<K, V> {
    private static final float LOAD_FACTOR_THRESHOLD = 0.5f;
    private ArrayList<Entry<K, V>> table;
    private int size;
    private int capacity;

    static class Entry<K, V> {
        K key;
        V value;
        boolean isDeleted;

        public Entry(K key, V value) {
            this.key = key;
            this.value = value;
            this.isDeleted = false;
        }
    }

    public MyOpenAddressingMap(int initialCapacity) {
        this.capacity = initialCapacity;
        this.table = new ArrayList<>(capacity);
        for (int i = 0; i < capacity; i++) {
            table.add(null);
        }
        this.size = 0;
    }

    @Override
    public void put(K key, V value) {
        if (key == null) {
            throw new IllegalArgumentException("Key cannot be null");
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) == null || table.get(index).isDeleted) {
            table.set(index, new Entry<>(key, value));
            size++;
            if ((float) size / capacity >= LOAD_FACTOR_THRESHOLD) {
                rehash();
            }
            return;
        }

        int j = 0;
        while (true) {
            index = (hashCode + j * j) % capacity;
            if (table.get(index) == null || table.get(index).isDeleted) {
                table.set(index, new Entry<>(key, value));
                size++;
                if ((float) size / capacity >= LOAD_FACTOR_THRESHOLD) {
                    rehash();
                }
                return;
            }
            j++;
        }
    }

    @Override
    public V get(K key) {
        if (key == null) {
            return null;
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) != null && table.get(index).key.equals(key)) {
            return table.get(index).value;
        }

        int j = 0;
        while (true) {
            index = (hashCode + j * j) % capacity;
            if (table.get(index) == null) {
                return null;
            }
            if (table.get(index).key.equals(key)) {
                return table.get(index).value;
            }
            j++;
        }
    }

    @Override
    public boolean containsKey(K key) {
        return get(key) != null;
    }

    @Override
    public void remove(K key) {
        if (key == null) {
            return;
        }

        int hashCode = key.hashCode();
        int index = hashCode % capacity;

        if (table.get(index) != null && table.get(index).key.equals(key)) {
            table.get(index).isDeleted = true;
            size--;
            return;
        }

        int j = 0;
        while (true) {
                index = (hashCode + j * j) % capacity;
                if (table.get(index) == null) {
                    return;
                }
                if (table.get(index).key.equals(key)) {
                    table.get(index).isDeleted = true;
                    size--;
                    return;
                }
                j++;
            }
        }

        @Override
        public int size() {
            return size;
        }

        @Override
        public boolean isEmpty() {
            return size == 0;
        }

        private void rehash() {
            ArrayList<Entry<K, V>> oldTable = table;
            capacity *= 2;
            table = new ArrayList<>(capacity);
            for (int i = 0; i < capacity; i++) {
                table.add(null);
            }
            size = 0;

            for (Entry<K, V> entry : oldTable) {
                if (entry != null && !entry.isDeleted) {
                    put(entry.key, entry.value);
                }
            }
        }

        public static void main(String[] args) {
            MyOpenAddressingMap<Integer, String> map = new
    MyOpenAddressingMap<>(4);
            map.put(3, "3");
            map.put(4, "4");
            map.put(45, "45");
            map.put(21, "21");
            map.put(92, "92");
            map.put(12, "12");

            System.out.println("Map size: " + map.size());
            System.out.println("Contains key 3: " + map.containsKey(3));
            System.out.println("Value for key 45: " + map.get(45));
            map.remove(45);
            System.out.println("After removal, contains key 45: " +
    map.containsKey(45));
        }
    }
```

23-24FINAL

✅ Q1. 查找泛型 ArrayList 中的最大值

题目解析：

实现一个泛型方法 max，用于从 ArrayList<E> 中找出最大值：

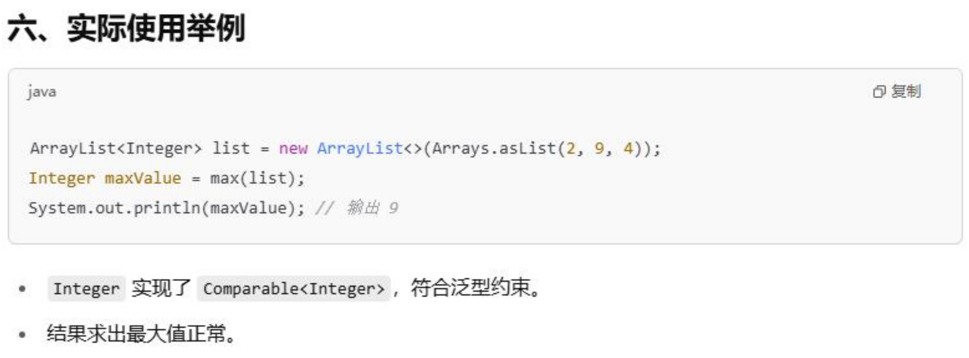
如果列表为空，返回 null

否则返回最大值

✅ 正确代码实现：

```
import java.util.ArrayList;
import java.util.Arrays;

public class RealEstateAnalyzer {
    public static void main(String[] args) {
        ArrayList<Double> prices2 = new ArrayList<>();
        System.out.println(max(prices2)); // -> null

        ArrayList<Double> prices1 = new ArrayList<>(Arrays.asList(1250000.1, 1750000.3,
2100000.6, 1900000.8, 2300000.2));
        System.out.println(max(prices1)); // -> 2300000.2
    }

    public static <E extends Comparable<E>> E max(ArrayList<E> list) {
        if (list == null || list.isEmpty()) {
            return null;
        }

        E maxElement = list.get(0);
        for (int i = 1; i < list.size(); i++) {
            if (list.get(i).compareTo(maxElement) > 0) {
                maxElement = list.get(i);
            }
        }



        //for (E element : list) {
            if (element.compareTo(maxElement) > 0) {
                maxElement = element;
            }
        }
        return maxElement;
    }
}
```

[修饰符] <类型参数声明> 返回类型 方法名(参数列表)



六、实际使用举例

```java
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(2, 9, 4));
Integer maxValue = max(list);
System.out.println(maxValue); // 输出 9
```

- `Integer` 实现了 `Comparable<Integer>`，符合泛型约束。
- 结果求出最大值正常。

✅ Q2. 找出专属 First Class 的乘客

题目解析：

有三个乘客队列（优先队列），找出只出现在 First Class 中、不出现在其他两个队列（Business 和 Economy）中的乘客。

✅ 正确代码实现：

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.PriorityQueue;
import java.util.Set;

public class FlightBoardingSystem {
    public static void main(String[] args) {
        // Task 1: 创建 PriorityQueue（优先队列）
        PriorityQueue<String> firstClassQueue = new PriorityQueue<>(Arrays.asList("Alice", "Bob", "Charlie", "Diana", "Ethan", "Fiona"));
        PriorityQueue<String> businessClassQueue = new PriorityQueue<>(Arrays.asList("Charlie", "Diana", "Grace", "Ian"));
        PriorityQueue<String> economyClassQueue = new PriorityQueue<>(Arrays.asList("Alice", "Grace", "Ethan", "Hannah", "Ian"));

        // Task 3: 调用方法并打印结果
        Set<String> result = findExclusiveFirstClassPassengers(firstClassQueue, businessClassQueue, economyClassQueue);
        System.out.println("Exclusive First Class Passengers: " + result);
    }

    public static Set<String> findExclusiveFirstClassPassengers(PriorityQueue<String> firstClassQueue,
PriorityQueue<String> businessClassQueue,

PriorityQueue<String> economyClassQueue) {
        // Task 2:  实现逻辑
        Set<String> first = new HashSet<>(firstClassQueue);
        Set<String> business = new HashSet<>(businessClassQueue);
        Set<String> economy = new HashSet<>(economyClassQueue);

        Set<String> exclusiveFirstClass = new HashSet<>(first);
        exclusiveFirstClass.removeAll(business);   //  移除在  business  中的
        exclusiveFirstClass.removeAll(economy);    //  移除在  economy  中的

        return exclusiveFirstClass;
    }
}
```
✅ 输出结果：

```
Exclusive First Class Passengers: [Bob, Fiona]
```
1. 数据结构说明

使用了**PriorityQueue<String>** 来存储各个舱位的乘客名单。

PriorityQueue 是一种按优先级排序的队列，内部元素有排序规则（字符串自然排序），但是在此题中，只需要存储名字，不关心优先级顺序。

为了方便对元素去重和合并，程序中将各优先队列转换为 HashSet（无序且不重复的集合）方便做集合运算。

2. 主程序 main 部分

```
PriorityQueue<String> firstClassQueue = new PriorityQueue<>(Arrays.asList("Alice", "Bob", "Charlie", "Diana", "Ethan", "Fiona"));
PriorityQueue<String> businessClassQueue = new PriorityQueue<>(Arrays.asList("Charlie", "Diana", "Grace", "Ian"));
PriorityQueue<String> economyClassQueue = new PriorityQueue<>(Arrays.asList("Alice", "Grace", "Ethan", "Hannah", "Ian"));
```
利用 Arrays.asList()将字符串数组转成列表，再传给 PriorityQueue 构造函数初始化 3 个乘客队列。

每个队列内的字符串即是乘客姓名。

之后调用：

```
Set<String> result = findExclusiveFirstClassPassengers(firstClassQueue, businessClassQueue, economyClassQueue);
System.out.println("Exclusive First Class Passengers: " + result);
```

调用了用于查找专属乘客的方法，并打印结果。

3. 关键方法 findExclusiveFirstClassPassengers

```
public static Set<String> findExclusiveFirstClassPassengers(
            PriorityQueue<String> firstClassQueue,
            PriorityQueue<String> businessClassQueue,
            PriorityQueue<String> economyClassQueue) {

    Set<String> first = new HashSet<>(firstClassQueue);
    Set<String> business = new HashSet<>(businessClassQueue);
    Set<String> economy = new HashSet<>(economyClassQueue);

    Set<String> exclusiveFirstClass = new HashSet<>(first);
    exclusiveFirstClass.removeAll(business);
    exclusiveFirstClass.removeAll(economy);

    return exclusiveFirstClass;
}
```

步骤详解：
转成 HashSet：

先把三个 PriorityQueue 都转成 HashSet。

HashSet 具有快速查找和不重复的特性，非常适合做集合的交集、差集运算。

构造 exclusiveFirstClass 集合：

新建一个 HashSet，初始元素为 firstClassQueue 中的所有元素。

移除在其他两个集合中出现的乘客：

exclusiveFirstClass.removeAll(business)：移除所有在 Business Class 中出现的乘客。

类似地，exclusiveFirstClass.removeAll(economy)：移除 Economy Class 的乘客。

剩下的就是只存在于 First Class 的乘客了。

4. 运行结果举例
在主函数中定义的：

First Class：Alice, Bob, Charlie, Diana, Ethan, Fiona

Business：Charlie, Diana, Grace, Ian

Economy：Alice, Grace, Ethan, Hannah, Ian

执行后，移除在 Business 和 Economy 同时出现的乘客后：

移除 Charlie, Diana（在 Business 中）

移除 Alice, Ethan（在 Economy 中）

剩余：Bob, Fiona

所以输出是：
Exclusive First Class Passengers: [Bob, Fiona]
5. 代码优点和技巧
利用 HashSet 实现集合差集操作：
通过 removeAll 能方便快捷地剔除不满足条件的元素，无需写繁琐循环。

参数类型使用 PriorityQueue<String>，而实际操作是把它转成 HashSet，兼顾传入参数和操作需求。

代码结构清晰，主程序负责数据和调用，方法负责核心逻辑。

23-24 R
// ✅ Q1: 图书馆管理系统

 题目 Q1：Library Management System（图书馆管理系统）
  问题描述：
你需要管理三个不同类型的图书分类队列：

Fiction（小说类）

Historical（历史类）

Reference（参考类）

每类图书使用 PriorityQueue 存储。你的任务是找出所有三类中都存在的书籍（即“热门图书”）。

✅ 解题思路：
将每个队列转成 HashSet，方便进行集合操作；

使用 retainAll() 方法执行集合“交集”，即找出同时存在于三个集合中的书名。

✅ 核心代码：

```
Set<String> fictionSet = new HashSet<>(fictionQueue);
Set<String> historicalSet = new HashSet<>(historicalQueue);
Set<String> referenceSet = new HashSet<>(referenceQueue);

fictionSet.retainAll(historicalSet);
fictionSet.retainAll(referenceSet);
```

输出结果解释：

例如，"The Great Gatsby" 出现在小说和参考类中，但没在历史类中，所以不是热门书。
而 "Pride and Prejudice" 出现在小说和历史类，但没在参考类中，也不算。

最终输出：

```
Popular Books in all 3 sections: [书名列表]
```

```
import java.util.*;

public class LibraryManagementSystem {
    public static void main(String[] args) {
        // Task 1: 创建每个书籍类别的优先队列
        PriorityQueue<String> fictionQueue = new PriorityQueue<>(Arrays.asList(
                "The Great Gatsby", "To Kill a Mockingbird", "1984",
                "Pride and Prejudice", "Harry Potter", "The Hobbit"));

        PriorityQueue<String> historicalQueue = new PriorityQueue<>(Arrays.asList(
                "Sapiens", "Pride and Prejudice", "A Brief History of Time", "Educated"));

        PriorityQueue<String> referenceQueue = new PriorityQueue<>(Arrays.asList(
                "The Great Gatsby", "A Brief History of Time", "Harry Potter",
                "Encyclopedia Britannica", "The Hobbit"));

        // Task 3: 调用 findPopularBooks 方法并打印出热门图书
        Set<String> popularBooks = findPopularBooks(fictionQueue, historicalQueue,
referenceQueue);
        System.out.println("Popular Books in all 3 sections: " + popularBooks);
    }

    // Task 2: 实现查找同时出现在三类中的书籍的方法
    public static Set<String> findPopularBooks(PriorityQueue<String> fictionQueue,
                                               PriorityQueue<String> historicalQueue,
                                               PriorityQueue<String> referenceQueue)
{
        // 将队列转换为集合便于求交集
        Set<String> fictionSet = new HashSet<>(fictionQueue);
        Set<String> historicalSet = new HashSet<>(historicalQueue);
        Set<String> referenceSet = new HashSet<>(referenceQueue);

        // 先和历史类求交集
        fictionSet.retainAll(historicalSet);
        // 再和参考类求交集，得到三类共有的书籍
        fictionSet.retainAll(referenceSet);

        return fictionSet;
    }
}
```

// ✅ Q2: 员工记录排序

✅ 题目 Q2：Employee Record Sorting（员工记录排序）
  问题描述：
你有两个数据：

员工编号（整型数组）

员工姓名（字符串数组）

你要完成：

用 Comparable 接口对员工编号进行升序插入排序；

用 Comparator 接口对员工姓名进行大小写无关的插入排序。

✅ 插入排序回顾：
插入排序是将一个元素一个一个插入已排序子数组中，时间复杂度为 $O(n^2)$，适合小规模排序。

✅ 核心点解析：
✔ Task 1: Comparable 排序 ID
public static <E extends Comparable<E>> void insertionSort(E[] list)
直接使用 compareTo() 方法判断顺序。

✔ Task 2: Comparator 排序姓名
public static <E> void insertionSort(E[] list, Comparator<? super E> comparator)
用外部传入的比较器规则排序，这里使用：
String.CASE_INSENSITIVE_ORDER
  输出结果解释：
```
Sorted Employee IDs:
[101, 102, 103, 104, 105]
Sorted Employee Names (Case-Insensitive):
```

```
[Alice, bob, Charlie, Diana, John]
```

注意 bob 和 Charlie 被排序在 Alice 后面，是因为忽略了大小写。

```
import java.util.*;

public class EmployeeRecordSorting {
    // Task 1: 使用 Comparable 接口实现插入排序（对员工 ID 升序排序）
    public static <E extends Comparable<E>> void insertionSort(E[] list) {
        for (int i = 1; i < list.length; i++) {
            E key = list[i];
            int j = i - 1;
            // 找到正确插入位置
            while (j >= 0 && list[j].compareTo(key) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = key;
        }
    }

    // Task 2: 使用 Comparator 接口实现插入排序（对姓名不区分大小写排序）
    public static <E> void insertionSort(E[] list, Comparator<? super E> comparator) {
        for (int i = 1; i < list.length; i++) {
            E key = list[i];
            int j = i - 1;
            // 按照比较器规则排序
            while (j >= 0 && comparator.compare(list[j], key) > 0) {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = key;
        }
    }

    public static void main(String[] args) {
        // 定义员工 ID 和姓名数组
        Integer[] employeeIds = {102, 101, 104, 103, 105};
        String[] employeeNames = {"John", "Alice", "bob", "Diana", "Charlie"};

        // 对 ID 排序
        insertionSort(employeeIds);
        System.out.println("Sorted Employee IDs:");
        System.out.println(Arrays.toString(employeeIds));
        // 对姓名排序（不区分大小写）
        insertionSort(employeeNames, String.CASE_INSENSITIVE_ORDER);
        System.out.println("Sorted Employee Names (Case-Insensitive):");
        System.out.println(Arrays.toString(employeeNames));
    }
}
```

编程题

Question 1:

Implement a generic method maxInList that accepts an ArrayList<E> where E extends Comparable<E>. It should return the maximum element or null if the list is empty.

```
public static <E extends Comparable<E>> E maxInList(ArrayList<E> list) {
    if (list == null || list.isEmpty()) return null;
    E max = list.get(0);
    for (E item : list) {
        if (item.compareTo(max) > 0) {
            max = item;
        }
    }
    return max;
}
```

该泛型方法遍历列表，查找最大元素，空列表返回 null。

This generic method iterates through the list to find the maximum element, returns null if empty.

Question 2:

Given three PriorityQueue<String> for Fiction, Historical, and Reference books, implement a method to find books that appear in all three queues.

```
public static Set<String> findPopularBooks(
        PriorityQueue<String> fictionQueue,
        PriorityQueue<String> historicalQueue,
        PriorityQueue<String> referenceQueue) {

        Set<String> fictionSet = new HashSet<>(fictionQueue);
        Set<String> historicalSet = new HashSet<>(historicalQueue);
        Set<String> referenceSet = new HashSet<>(referenceQueue);

        fictionSet.retainAll(historicalSet);
        fictionSet.retainAll(referenceSet);

        return fictionSet;
}
```

该方法通过集合交集筛选在三队列中均存在的书名。
This method uses set intersections to find books common to all three queues.

Part 2: Coding Questions (编程题)

Level 1 (Basic)

Q1. Find Minimum Element in ArrayList Complete the following generic method that finds the minimum element in an `ArrayList` of generic elements. Return `null` if the `ArrayList` is empty.

```
Java
import java.util.ArrayList;import java.util.Arrays;
public class ArrayMinFinder {
   public static <E extends Comparable<E>> E findMin(ArrayList<E> list)
{
       // Task 1: Handle empty list case
       // Task 2: Find and return the minimum element
       return null; // Placeholder
   }

   public static void main(String[] args) {
       ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2,
8, 1, 9));
       System.out.println("Min number: " + findMin(numbers)); //
Expected: 1

       ArrayList<String> words = new ArrayList<>(Arrays.asList("apple",
"orange", "banana", "grape"));
       System.out.println("Min word: " + findMin(words)); // Expected:
apple

       ArrayList<Double> emptyList = new ArrayList<>();
       System.out.println("Min from empty list: " + findMin(emptyList));
// Expected: null
   }
}
```

Level 2 (Intermediate)

Q2. Union of Two Sets You are given two sets of integers. Your task is to find the union of these two sets. The union of two sets A and B is the set containing all elements that are in A, or in B, or in both.

Complete the `findUnion` method.

```
Java
import java.util.HashSet;import java.util.Set;import java.util.Arrays;
public class SetOperations {
    /**
     * Finds the union of two sets of integers.
     * @param set1 The first set.
     * @param set2 The second set.
     * @return A new set containing the union of set1 and set2.
     */
    public static Set<Integer> findUnion(Set<Integer> set1, Set<Integer>
set2) {
        // Task 1: Create a new HashSet to store the union.
        // Task 2: Add all elements from set1 to the new set.
        // Task 3: Add all elements from set2 to the new set.
        // Task 4: Return the union set.
        return null; // Placeholder
    }

    public static void main(String[] args) {
         Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4));
         Set<Integer> s2 = new HashSet<>(Arrays.asList(3, 4, 5, 6));
        Set<Integer> unionSet = findUnion(s1, s2);
        System.out.println("Union: " + unionSet); // Expected: [1, 2, 3,
4, 5, 6] (order may vary)

        Set<Integer> s3 = new HashSet<>(Arrays.asList(10, 20));
        Set<Integer> s4 = new HashSet<>(Arrays.asList(30, 40));
        Set<Integer> unionSet2 = findUnion(s3, s4);
        System.out.println("Union 2: " + unionSet2); // Expected: [10, 20,
30, 40] (order may vary)
    }
}
```

Level 3 (Challenging)

Q3. Custom Insertion Sort with Comparator You need to sort an array of `Product` objects. Each `Product` has a `name` (String) and a `price` (double). Implement an insertion sort method that can sort these `Product` objects based on their price in ascending order using a `Comparator`.

```
Java
import java.util.Arrays;import java.util.Comparator;
public class ProductSorter {

    static class Product {
        String name;
        double price;

        public Product(String name, double price) {
            this.name = name;
            this.price = price;
        }

        @Override
        public String toString() {
            return name + " ($" + price + ")";
        }
    }

    /**
     * Sorts an array of elements using the Insertion Sort algorithm with
a custom Comparator.
     * @param list The array of elements to be sorted.
     * @param comparator The comparator to determine the order of
elements.
     */
    public static <E> void insertionSort(E[] list, Comparator<? super E>
comparator) {
        // Task 1: Implement the outer loop for insertion sort (from the
second element).
        // Task 2: Store the current element to be inserted.
        // Task 3: Implement the inner loop to shift elements greater than
the current element to the right.
        // Task 4: Place the current element in its correct sorted position.
    }

    public static void main(String[] args) {
        Product[] products = {
            new Product("Laptop", 1200.0),
            new Product("Mouse", 25.0),
            new Product("Keyboard", 75.0),
            new Product("Monitor", 300.0),
            new Product("Webcam", 50.0)
        };

        System.out.println("Original Products: " +
Arrays.toString(products));

        // Sort products by price in ascending order
        insertionSort(products, new Comparator<Product>() {
            @Override
            public int compare(Product p1, Product p2) {
                return Double.compare(p1.price, p2.price);
            }
        });

        System.out.println("Sorted by Price (Ascending): " +
Arrays.toString(products));
        // Expected: [Mouse ($25.0), Webcam ($50.0), Keyboard ($75.0),
Monitor ($300.0), Laptop ($1200.0)]
    }
}
```

Part 2: Coding Questions (编程题) Answers

Q1. Find Minimum Element in ArrayList

```java
Java
import java.util.ArrayList;import java.util.Arrays;import
java.util.Collections; // For convenience, if allowed
public class ArrayMinFinder {
   public static <E extends Comparable<E>> E findMin(ArrayList<E> list)
{
       // Task 1: Handle empty list case
       if (list == null || list.isEmpty()) {
          return null;
       }

       // Task 2: Find and return the minimum element
       // Option 1: Using Collections.min()
       return Collections.min(list);

       /*
        // Option 2: Manual iteration
        E minElement = list.get(0);
        for (int i = 1; i < list.size(); i++) {
            // compareTo returns a negative integer, zero, or a positive
integer
            // as this object is less than, equal to, or greater than the
specified object.
            if (list.get(i).compareTo(minElement) < 0) {
                minElement = list.get(i);
            }
        }
        return minElement;
        */
   }

   public static void main(String[] args) {
       ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(5, 2,
8, 1, 9));
       System.out.println("Min number: " + findMin(numbers));

       ArrayList<String> words = new ArrayList<>(Arrays.asList("apple",
"orange", "banana", "grape"));
```

```
        System.out.println("Min word: " + findMin(words));

        ArrayList<Double> emptyList = new ArrayList<>();
        System.out.println("Min from empty list: " + findMin(emptyList));
    }
}
```

Q2. Union of Two Sets

```
Java
import java.util.HashSet;import java.util.Set;import java.util.Arrays;
public class SetOperations {
    /**
     * Finds the union of two sets of integers.
     * @param set1 The first set.
     * @param set2 The second set.
     * @return A new set containing the union of set1 and set2.
     */
    public static Set<Integer> findUnion(Set<Integer> set1, Set<Integer>
set2) {
        // Task 1: Create a new HashSet to store the union.
        Set<Integer> unionSet = new HashSet<>();

        // Task 2: Add all elements from set1 to the new set.
        if (set1 != null) {
            unionSet.addAll(set1);
        }

        // Task 3: Add all elements from set2 to the new set.
        // HashSet automatically handles duplicates.
        if (set2 != null) {
            unionSet.addAll(set2);
        }

        // Task 4: Return the union set.
        return unionSet;
    }

    public static void main(String[] args) {
         Set<Integer> s1 = new HashSet<>(Arrays.asList(1, 2, 3, 4));
         Set<Integer> s2 = new HashSet<>(Arrays.asList(3, 4, 5, 6));
        Set<Integer> unionSet = findUnion(s1, s2);
        System.out.println("Union: " + unionSet);
        Set<Integer> s3 = new HashSet<>(Arrays.asList(10, 20));
        Set<Integer> s4 = new HashSet<>(Arrays.asList(30, 40));
        Set<Integer> unionSet2 = findUnion(s3, s4);
        System.out.println("Union 2: " + unionSet2);
    }
}
```

Q3. Custom Insertion Sort with Comparator

```
Java
import java.util.Arrays;import java.util.Comparator;
public class ProductSorter {

    static class Product {
         String name;
         double price;

         public Product(String name, double price) {
             this.name = name;
             this.price = price;
         }

         @Override
         public String toString() {
             return name + " ($" + price + ")";
        }
    }

    /**
      * Sorts an array of elements using the Insertion Sort algorithm with
a custom Comparator.
      * @param list The array of elements to be sorted.
      * @param comparator The comparator to determine the order of
elements.
      */
    public static <E> void insertionSort(E[] list, Comparator<? super E>
comparator) {
        if (list == null || list.length <= 1) {
           return; // Nothing to sort
        }

        // Task 1: Implement the outer loop for insertion sort (from the
second element).
        for (int i = 1; i < list.length; i++) {
           // Task 2: Store the current element to be inserted.
           E currentElement = list[i];
           int k = i - 1;

           // Task 3: Implement the inner loop to shift elements greater
than the current element to the right.
           // Use comparator to compare elements
           while (k >= 0 && comparator.compare(list[k], currentElement) >
0) {
                list[k + 1] = list[k];
                k--;
           }

           // Task 4: Place the current element in its correct sorted
position.
           list[k + 1] = currentElement;
        }
    }

    public static void main(String[] args) {
        Product[] products = {
           new Product("Laptop", 1200.0),
           new Product("Mouse", 25.0),
           new Product("Keyboard", 75.0),
           new Product("Monitor", 300.0),
           new Product("Webcam", 50.0)
        };

        System.out.println("Original Products: " +
Arrays.toString(products));

        // Sort products by price in ascending order
        insertionSort(products, new Comparator<Product>() {
             @Override
             public int compare(Product p1, Product p2) {
                 return Double.compare(p1.price, p2.price);
           }
        });

        System.out.println("Sorted by Price (Ascending): " +
Arrays.toString(products));
    }
}
```

编程题

Question 1:
Write a generic method `minElement` which returns the minimum element from an `ArrayList<E>` where `E` extends `Comparable<E>`. Return null if the list is empty.

```
public static <E extends Comparable<E>> E minElement(ArrayList<E> list)
{
    if (list == null || list.isEmpty()) return null;
    E min = list.get(0);
    for (E e : list) {
        if (e.compareTo(min) < 0) {
            min = e;
        }
    }
    return min;
}
```

该泛型方法遍历列表返回最小元素，空列表时返回 null。
This generic method iterates over the list to find the minimum element or returns null if list is empty.

Question 2:
Given three `PriorityQueue<String>`, representing three queues of passengers in "FirstClass", "BusinessClass" and "EconomyClass", implement a method to find passengers who exist only in "FirstClass" queue and not in the other two.

```
public static Set<String> findExclusiveFirstClassPassengers(
    PriorityQueue<String> firstClassQueue,
    PriorityQueue<String> businessClassQueue,
    PriorityQueue<String> economyClassQueue) {

    Set<String> firstSet = new HashSet<>(firstClassQueue);
    Set<String> businessSet = new HashSet<>(businessClassQueue);
    Set<String> economySet = new HashSet<>(economyClassQueue);

    firstSet.removeAll(businessSet);
    firstSet.removeAll(economySet);

    return firstSet;
}
```

该方法利用集合 `removeAll` 排除非专属乘客，返回独占首等舱的乘客。
This method uses `removeAll` to exclude passengers in other queues, returning exclusive FirstClass passengers.

LAB!!!!!!!!!!!!!!

Exercise 1.1 In-place Reverse

The reverse method in the lecture reverses an array of integers by copying it to a new array and
returning it.

Write a method reverseInPlace that reverses the array of doubles passed in the argument and returns nothing.

After that, write a test program that prompts the user to enter arbitrary numbers, invokes the method to reverse the numbers, and displays the array.

Your test program should first prompt the user to enter the array size.

Test the method on all kinds of input array.

Exercise 1.2 Deciding four consecutive numbers

Write the following method that tests whether the array has four consecutive numbers with the
same value:

public static boolean isConsecutiveFour(int[] values)

After that, write a test program that prompts the user to enter a series of integers and displays it if the series contains four consecutive numbers with the same value.

Your program should first prompt the user to enter the input size—i.e., the number of values in
the series.

Test the method on all kinds of input values.

Exercise 1.1 In-place Reverse
题意
写一个 reverseInPlace 方法，原地反转传入的 double 数组，方法无返回值。
然后写一个测试程序，

先让用户输入数组大小（int size）

再输入对应数量的数字填充数组

调用 reverseInPlace 方法反转数组

输出反转后的数组

思路分析
原地反转：不借助额外数组，只交换数组首尾元素直到中间。

交换方式：使用临时变量交换 values[i]和 values[n-1 - i]。

测试程序用 Scanner 读取输入，验证实现。

代码示范
```java
import java.util.Scanner;

public class ReverseArrayDemo {

    // 反转数组方法，原地修改，无返回值
    public static void reverseInPlace(double[] arr) {
        int n = arr.length;
        for (int i = 0; i < n / 2; i++) {
            double temp = arr[i];
            arr[i] = arr[n - 1 - i];
            arr[n - 1 - i] = temp;
        }
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 输入数组大小
        System.out.print("Enter array size: ");
        int size = scanner.nextInt();
        double[] numbers = new double[size];

        // 输入数组元素
        System.out.println("Enter " + size + " numbers:");
        for (int i = 0; i < size; i++) {
            numbers[i] = scanner.nextDouble();
        }

        // 调用反转方法
        reverseInPlace(numbers);

        // 打印结果
        System.out.println("Reversed array:");
        for (double num : numbers) {
            System.out.print(num + " ");
        }
    }
}
```

Exercise 1.2 Deciding four consecutive numbers

题意

写一个静态方法 isConsecutiveFour：判断一个整数数组中是否存在四个连续且相同的数字。返回 boolean。

写测试程序：

先让用户输入数组长度

再输入数字序列

判断是否存在 4 个连续相同数字，结果告诉用户。

思路分析

遍历数组，从 index 0 开始判断当前位置的数字是否等于接下来 3 个位置的数字。

一旦找到，即返回 true。

数组长度不足 4，直接返回 false。

代码示范

```
import java.util.Scanner;

public class ConsecutiveFourDemo {

    public static boolean isConsecutiveFour(int[] values) {
        if (values.length < 4) return false;

        for (int i = 0; i <= values.length - 4; i++) {
            int current = values[i];
            boolean allEqual = true;
            for (int j = i + 1; j < i + 4; j++) {
                if (values[j] != current) {
                    allEqual = false;
                    break;
                }
            }
            if (allEqual) return true;
        }
        return false;
    }

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // 输入数组长度
        System.out.print("Enter input size: ");
        int size = scanner.nextInt();

        int[] values = new int[size];

        System.out.println("Enter " + size + " integers:");
        for (int i = 0; i < size; i++) {
            values[i] = scanner.nextInt();
        }

        // 调用判断方法
        boolean result = isConsecutiveFour(values);

        if (result) {
            System.out.println("The series contains four consecutive numbers with the same
value.");
        } else {
            System.out.println("The series does NOT contain four consecutive numbers with
the same value.");
        }
    }
}
```

Exercise 2.1 My 2D Point Implementations

 Design a class named MyPoint to represent a point with x- and y-coordinates.

 The class contains:

o The data fields (instance variables) x and y that represent the coordinates with getter methods.

o An empty constructor that creates a point (0.0, 0.0).

o A constructor that constructs a point with specified coordinates.

o A (instance) method named distance that returns the distance from this point to another point of the MyPoint type.

o A static method (function) also named distance that returns the distance from two MyPoint objects.

 Write a test program that creates the three points (0.0, 0.0), (10.25, 20.8) and (13.25, 24.8) and

displays the distance between them using both distance implementations.

Notice the difference between invoking an instance method and a static method (functions)!

Exercise 2.2 Big Integers Divisible by 2 or 3

 Write a function that prints the first 10 numbers with 50 decimal digits that are divisible by 2 or

3.

Exercise 2.3 Person, Student, Employee, Faculty, Staff

 Design a class named Person and its two subclasses named Student and Employee.

 Make Faculty and Staff subclasses of Employee.

 A person has a name, address, phone number, and e-mail address.

 A student has a class status (freshman, sophomore, junior, or senior).

Define the status as a constant.

 An employee has an office, salary, and date hired.

Use the LocalDate class to create an object for date hired.

 A faculty member has office hours and a rank.

 A staff member has a title.

 Add a constructor that only takes name to each class (you may add other constructors).

 Override the toString method in each class to display the class name and the person’s name.

 What should the access modifier of name in Person be?

 Write a test program that creates a Person, Student, Employee, Faculty, and Staff

objects in an array, and invokes their toString() methods using polymorphism.

Exercise 2.4 MyStack using Inheritance

 In Lecture 2, we implement MyStack using composition.

 Now, implement a new MyStack class that extends ArrayList Instead.

o The class definition should be

public class MyStack extends ArrayList<Object>

o The methods are isEmpty(), getSize(), peek(), pop(), push(Object o), search(Object o) and toString().

Write a test program that prompts the user to enter five strings and displays them in reverse order.

Exercise 2.1 My 2D Point Implementations

题目要求

定义类 MyPoint，包含 double 类型实例变量 x 和 y，带 getter

两个构造器：无参默认点(0.0, 0.0)，带坐标参数

实例方法 distance(MyPoint p)：计算当前点到参数点的距离

静态方法 distance(MyPoint p1, MyPoint p2)：计算两个点之间的距离

编写测试程序，创建三个点，分别用实例和静态方法计算距离并打印

代码实现

java

复制

```
public class MyPoint {
    private double x;
    private double y;

    // 无参构造器，默认 0.0,0.0
    public MyPoint() {
        this.x = 0.0;
        this.y = 0.0;
    }

    // 带参数构造器
    public MyPoint(double x, double y) {
        this.x = x;
        this.y = y;
    }

    // getter 方法
    public double getX() { return x; }
    public double getY() { return y; }

    // 实例方法，计算 this 与 p 之间距离
    public double distance(MyPoint p) {
        double dx = this.x - p.x;
        double dy = this.y - p.y;
        return Math.sqrt(dx * dx + dy * dy);
    }

    // 静态方法，计算 p1 与 p2 之间距离
    public static double distance(MyPoint p1, MyPoint p2) {
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

测试代码

java

复制

```java
public class TestMyPoint {
    public static void main(String[] args) {
        MyPoint p1 = new MyPoint(0.0, 0.0);
        MyPoint p2 = new MyPoint(10.25, 20.8);
        MyPoint p3 = new MyPoint(13.25, 24.8);

        // 用实例方法计算距离
        System.out.printf("Distance p1 to p2 (instance): %.2f%n", p1.distance(p2));
        System.out.printf("Distance p2 to p3 (instance): %.2f%n", p2.distance(p3));

        // 用静态方法计算距离
        System.out.printf("Distance p1 to p3 (static): %.2f%n", MyPoint.distance(p1, p3));
        System.out.printf("Distance p3 to p2 (static): %.2f%n", MyPoint.distance(p3, p2));
    }
}
```

Exercise 2.2 Big Integers Divisible by 2 or 3

题目要求

打印前 10 个 50 位十进制能被 2 或 3 整除的数字

这个题用 java.math.BigInteger，利用除法和模运算实现

代码示范

java

复制

```java
import java.math.BigInteger;

public class BigIntegerDivisible {
    public static void main(String[] args) {
        BigInteger start = new BigInteger("1" + "0".repeat(49)); // 10^49，50 位数最小数
        int count = 0;
        BigInteger one = BigInteger.ONE;

        while (count < 10) {
            // 判断能否被 2 或 3 整除
            if (start.mod(BigInteger.valueOf(2)).equals(BigInteger.ZERO) ||
                start.mod(BigInteger.valueOf(3)).equals(BigInteger.ZERO)) {
                System.out.println(start);
                count++;
            }
            start = start.add(one);
        }
    }
}
```

Exercise 2.3 Person, Student, Employee, Faculty, Staff

题目要求与设计

Person 基类：name, address, phone, email；name 访问控制：建议用 protected 方便继承使用

Student extends Person：有 class status，定义为常量(freshman, sophomore, ...)

Employee extends Person：加 office, salary, date hired(LocalDate)

Faculty extends Employee：加 office hours, rank

Staff extends Employee：加 title

每类至少有带 name 参数的构造器，可重载

覆盖 toString()，打印类名和姓名

写 main 测试程序，用多态遍历对象数组并调用 toString()

代码实现示范

java

复制

```java
import java.time.LocalDate;

// Person 类
class Person {
    protected String name;
    private String address;
    private String phone;
    private String email;
    public Person(String name) {
        this.name = name;
    }

    // 可添加其他构造器和 getter/setter 略

    @Override
    public String toString() {
        return getClass().getSimpleName() + ": " + name;
    }
}

// Student 类
class Student extends Person {
    public static final String FRESHMAN = "Freshman";
    public static final String SOPHOMORE = "Sophomore";
    public static final String JUNIOR = "Junior";
    public static final String SENIOR = "Senior";

    private String status;

    public Student(String name) {
        super(name);
    }

    public String getStatus() { return status; }
    public void setStatus(String status) { this.status = status; }

    @Override
    public String toString() {
        return super.toString() + ", Status: " + status;
    }
}

// Employee 类
class Employee extends Person {
    private String office;
    private double salary;
    private LocalDate dateHired;

    public Employee(String name) {
        super(name);
    }
    public String getOffice() { return office; }
    public void setOffice(String office) { this.office = office; }
    public double getSalary() { return salary; }
    public void setSalary(double salary) { this.salary = salary; }
    public LocalDate getDateHired() { return dateHired; }
    public void setDateHired(LocalDate dateHired) { this.dateHired = dateHired; }

    @Override
    public String toString() {
        return super.toString() + ", Office: " + office;
    }
}

// Faculty 类
class Faculty extends Employee {
    private String officeHours;
    private String rank;

    public Faculty(String name) {
        super(name);
    }

    public String getOfficeHours() { return officeHours; }
    public void setOfficeHours(String officeHours) { this.officeHours = officeHours; }
    public String getRank() { return rank; }
    public void setRank(String rank) { this.rank = rank; }

    @Override
    public String toString() {
        return super.toString() + ", Rank: " + rank;
    }
}

// Staff 类
class Staff extends Employee {
    private String title;

    public Staff(String name) {
        super(name);
    }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
    @Override
    public String toString() {
        return super.toString() + ", Title: " + title;
    }
}
```
测试程序示范

java

复制

```
public class TestPersonHierarchy {
    public static void main(String[] args) {
        Person[] people = new Person[5];

        people[0] = new Person("Tom");
        Student s = new Student("Jerry");
        s.setStatus(Student.SENIOR);
        people[1] = s;
        Employee e = new Employee("John");
        e.setOffice("Room 101");
        people[2] = e;
        Faculty f = new Faculty("Susan");
        f.setRank("Associate Professor");
        people[3] = f;
        Staff st = new Staff("Bob");
        st.setTitle("Manager");
        people[4] = st;

        for (Person p : people) {
            System.out.println(p.toString());
        }
    }
}
```
Exercise 2.4 MyStack using Inheritance

题目要求

继承自 ArrayList<Object> 实现 MyStack 类

实现方法：

isEmpty() 返回是否栈空

getSize() 返回栈大小

peek() 返回栈顶元素但不弹出，空栈返回 null 或抛异常

pop() 弹出栈顶元素

push(Object o) 入栈

search(Object o) 查找元素位置（类似 Java Stack 的 search）

toString() 返回栈元素字符串表示

测试程序输入 5 个字符串，逆序输出

代码实现
java
复制
```java
import java.util.ArrayList;

public class MyStack extends ArrayList<Object> {

    public boolean isEmpty() {
        return super.isEmpty();
    }

    public int getSize() {
        return super.size();
    }

    public Object peek() {
        if (isEmpty()) return null; // 或 throw exception
        return get(getSize() - 1);
    }

    public Object pop() {
        if (isEmpty()) return null;
        return remove(getSize() - 1);
    }

    public void push(Object o) {
        add(o);
    }

    public int search(Object o) {
        int index = lastIndexOf(o);
        if (index >= 0) {
            // 返回栈顶起向下的距离，栈顶为 1
            return getSize() - index;
        }
        return -1;
    }

    @Override
    public String toString() {
        return super.toString();
    }
}
```
测试程序

java

复制
```
import java.util.Scanner;

public class TestMyStack {
    public static void main(String[] args) {
        MyStack stack = new MyStack();
        Scanner scanner = new Scanner(System.in);

        System.out.println("Enter five strings:");
        for (int i = 0; i < 5; i++) {
            String input = scanner.nextLine();
            stack.push(input);
        }

        System.out.println("Stack elements in reverse order:");
        while (!stack.isEmpty()) {
            System.out.println(stack.pop());
        }
    }
}
```

Exercise 3.1 ArrayList Shuffling

Write the following method that shuffles an ArrayList of numbers:

public static void shuffle(ArrayList<Number> list)

Hints:

1. Create the ArrayList first
2. In the shuffle() method, you may consider the Fisher-Yates shuffle algorithm, which is efficient for shuffling.
3. Generate a random index within the range of the list size using Math.random().
4. Swap the element at index 'i' with the element at the randomly generated index to perform the shuffle.

Exercise 3.2 Comparable Interface
Define a class named ComparableCircle that extends Circle and implements Comparable. Implement the compareTo method to compare the circles on the basis of area. Write a test class to find the larger of two instances of ComparableCircle objects
Hints:
1. Define a class named ComparableCircle that extends Circle and implements Comparable interface. You need to implement the compareTo method to compare the circles based on their radii.
2. Implement the compareTo method in the ComparableCircle class to compare circles based on their radii. You should compare the radii of two circles and return a positive integer if the radius of the current circle is greater, a negative integer if it's smaller, and zero if they are equal.
3. Write a test class to verify the functionality of the ComparableCircle class. Create two ComparableCircle objects and compare their sizes using the max method.

Exercise 3.3 Deep Copy
Rewrite the MyStack class in Lecture 2.2 (page 43) to perform a deep copy of the list field.
Hints:
1. Implement the Cloneable interface in the MyStack class to enable cloning functionality.
2. Override the clone method in the MyStack class to create a deep copy of the list field.3. Within the clone method, clone each object in the list to ensure a deep copy is made.

Exercise 3.1 ArrayList Shuffling
题目要求
实现静态方法 shuffle(ArrayList<Number> list)，使用 Fisher-Yates 洗牌算法随机打乱集合中的元素顺序。

代码示范
java
复制
```
import java.util.ArrayList;

public class ShuffleDemo {
    public static void shuffle(ArrayList<Number> list) {
        int n = list.size();
        for (int i = n - 1; i >= 1; i--) {
            int j = (int) (Math.random() * (i + 1)); // 随机索引 0~i
            // 交换 list[i] 与 list[j]
            Number temp = list.get(i);
            list.set(i, list.get(j));
            list.set(j, temp);
        }
    }

    // 测试
    public static void main(String[] args) {
        ArrayList<Number> numbers = new ArrayList<>();
        for (int i = 1; i <= 10; i++) numbers.add(i);

        System.out.println("Original list: " + numbers);
        shuffle(numbers);
        System.out.println("Shuffled list: " + numbers);
    }
}
```

Exercise 3.2 Comparable Interface

题目要求

定义一个 ComparableCircle 类，继承 Circle 并实现 Comparable<ComparableCircle>接口

实现 compareTo，按半径比较圆的大小

写测试类创建两个 ComparableCircle 实例，并用 max 功能比较大小

预设 Circle 类（简易）

java

复制

```
public class Circle {
    protected double radius;

    public Circle() { radius = 1.0; }

    public Circle(double radius) { this.radius = radius; }

    public double getRadius() { return radius; }

    public double getArea() { return Math.PI * radius * radius; }
}
```

ComparableCircle 类

java

复制

```
public class ComparableCircle extends Circle implements Comparable<ComparableCircle> {

    public ComparableCircle() { super(); }

    public ComparableCircle(double radius) { super(radius); }

    @Override
    public int compareTo(ComparableCircle o) {
        if (this.radius > o.radius) return 1;
        else if (this.radius < o.radius) return -1;
        else return 0;
    }

    @Override
    public String toString() {
        return "Circle radius: " + radius;
    }
}
```
测试类

java

复制
```
import java.util.Comparator;

public class TestComparableCircle {

    // 使用简单 max 方法找最大
    public static ComparableCircle max(ComparableCircle c1, ComparableCircle c2) {
        return (c1.compareTo(c2) >= 0) ? c1 : c2;
    }

    public static void main(String[] args) {
        ComparableCircle c1 = new ComparableCircle(5);
        ComparableCircle c2 = new ComparableCircle(7);

        ComparableCircle larger = max(c1, c2);
        System.out.println("Larger circle is: " + larger);
    }
}
```
Exercise 3.3 Deep Copy MyStack

题目要求

让自定义 MyStack 类实现 Cloneable 接口

重写 clone() 方法，实现对内部 list 字段的深拷贝

对 list 中的每个对象也调用 clone()确保深拷贝

说明

假设 MyStack 内部持有 ArrayList<E>类型的字段 list

元素 E 必须实现 Cloneable 接口和 clone()方法

clone()时需逐个克隆元素，然后新建新列表

示例代码
java
复制
```java
import java.util.ArrayList;

public class MyStack<E extends Cloneable> implements Cloneable {
    private ArrayList<E> list = new ArrayList<>();

    public void push(E e) {
        list.add(e);
    }

    public E pop() {
        if (list.isEmpty()) return null;
        return list.remove(list.size() - 1);
    }

    public boolean isEmpty() {
        return list.isEmpty();
    }

    public int getSize() {
        return list.size();
    }

    @Override
    public MyStack<E> clone() throws CloneNotSupportedException {
        MyStack<E> copy = (MyStack<E>) super.clone();
        copy.list = new ArrayList<>();
        for (E item : this.list) {
            // 注意：这里只演示调用 clone，实际需强制转换及异常处理
            try {
                E clonedItem = (E) item.getClass().getMethod("clone").invoke(item);
                copy.list.add(clonedItem);
            } catch (Exception e) {
                throw new CloneNotSupportedException("Element not cloneable");
            }
        }
        return copy;
    }
}
```

Part 2: Coding Questions

Question 1: Generic Method to Find Minimum Element

Write a generic method named findMin that takes an array of elements of type T, where T extends Comparable<T>. The method returns the minimum element in the array. If the array is empty, return null.

Answer:

```
public static <T extends Comparable<T>> T findMin(T[] array) {
        if (array == null || array.length == 0) return null;
        T min = array[0];
        for (T element : array) {
                if (element.compareTo(min) < 0) {
                        min = element;
                }
        }
        return min;
}
```

Question 2: Implement a Simple PhoneBook Using HashMap

Write a Java program to implement a simple phone book. It should have methods to:

Add a contact with a name and a phone number

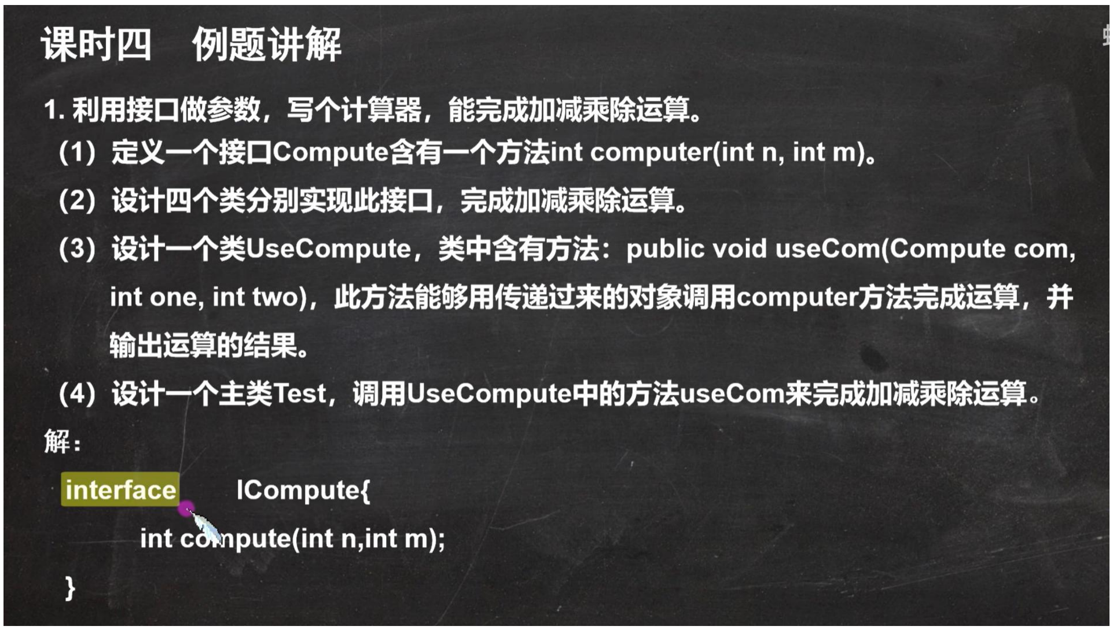
Lookup and print the phone number by name

Remove a contact by name

Use a HashMap<String, String> to store contacts.

Answer:

```
import java.util.HashMap;

public class PhoneBook {
        private HashMap<String, String> contacts = new HashMap<>();

        public void addContact(String name, String phone) {
                contacts.put(name, phone);
        }

        public void lookupContact(String name) {
                String phone = contacts.get(name);
                if (phone != null) {
                        System.out.println(name + "'s number is: " + phone);
        } else {
            System.out.println("Contact not found.");
        }
    }

    public void removeContact(String name) {
        if (contacts.remove(name) != null) {
            System.out.println(name + " removed from phone book.");
        } else {
            System.out.println("Contact not found.");
        }
    }

    public static void main(String[] args) {
        PhoneBook pb = new PhoneBook();
        pb.addContact("Alice", "12345");
        pb.addContact("Bob", "67890");

        pb.lookupContact("Alice");
        pb.removeContact("Bob");
        pb.lookupContact("Bob");
    }
}
```



课时四　例题讲解

1. 利用接口做参数，写个计算器，能完成加减乘除运算。

（1）定义一个接口Compute含有一个方法int computer(int n, int m)。

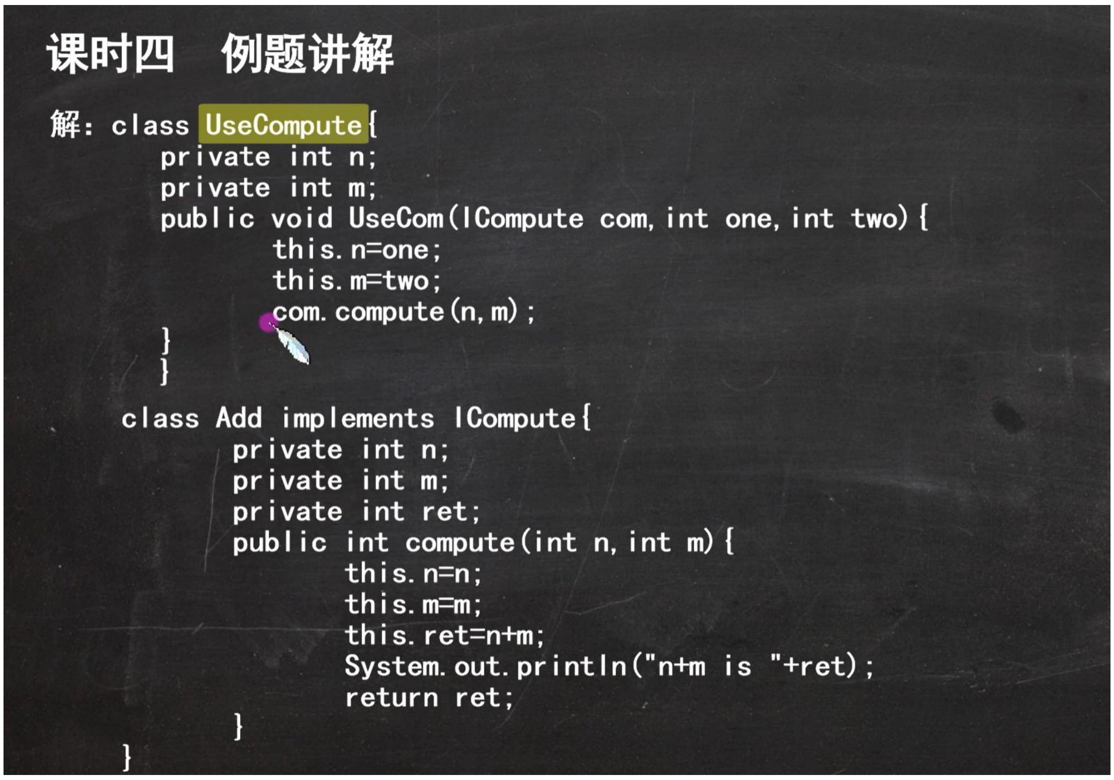
（2）设计四个类分别实现此接口，完成加减乘除运算。

（3）设计一个类UseCompute，类中含有方法：public void useCom(Compute com, int one, int two)，此方法能够用传递过来的对象调用computer方法完成运算，并输出运算的结果。

（4）设计一个主类Test，调用UseCompute中的方法useCom来完成加减乘除运算。

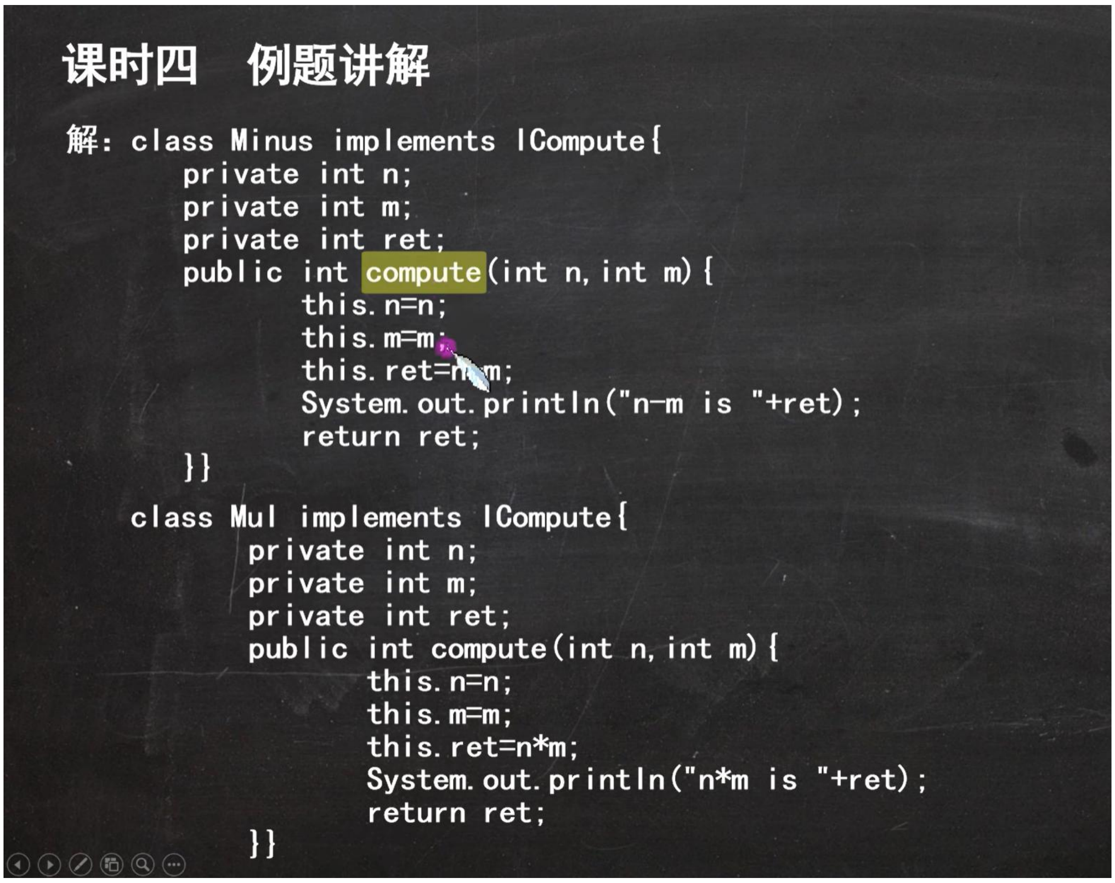
解：

```
interface      ICompute{
      int compute(int n,int m);
}
```



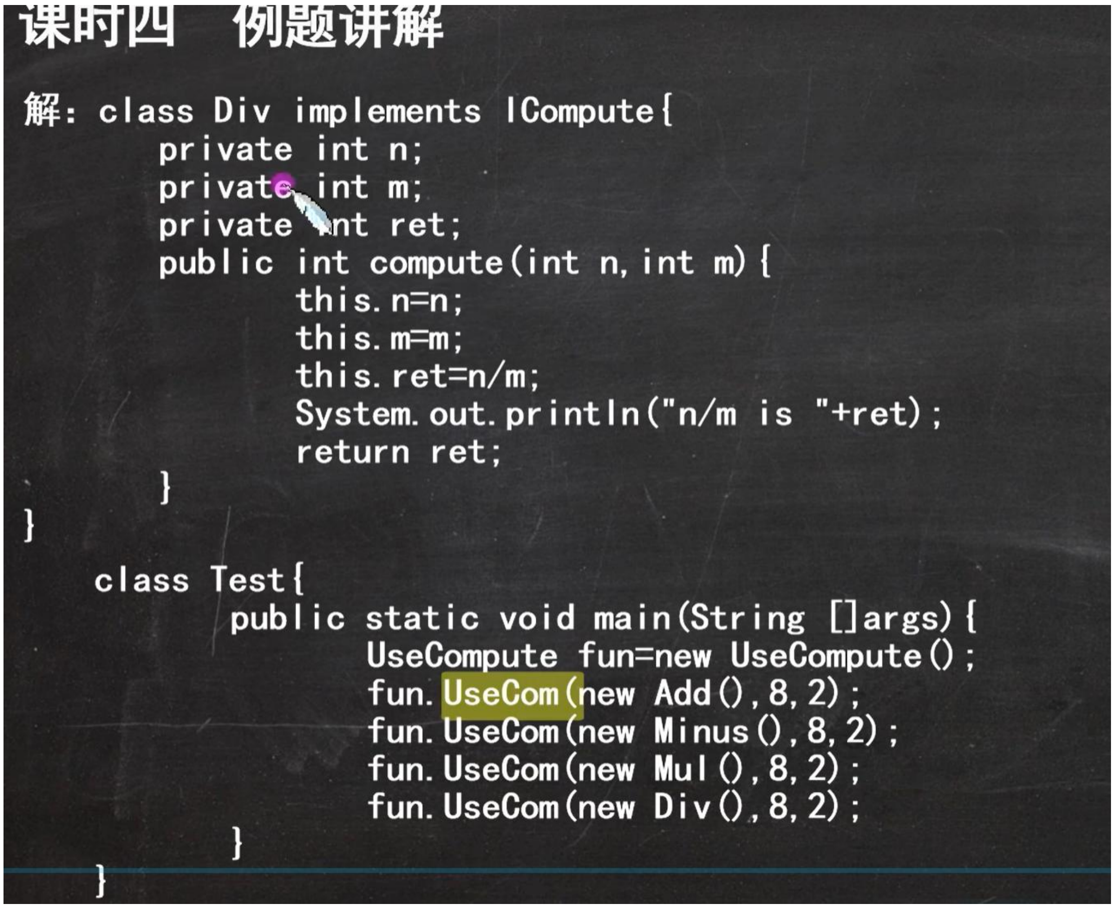
课时四　例题讲解

解：
```
class UseCompute{
      private int n;
      private int m;
      public void UseCom(ICompute com,int one,int two){
            this.n=one;
            this.m=two;
            com.compute(n,m);
      }
      }

    class Add implements ICompute{
          private int n;
          private int m;
          private int ret;
          public int compute(int n,int m){
                this.n=n;
                this.m=m;
                this.ret=n+m;
                System.out.println("n+m is "+ret);
                return ret;
          }
    }
```



课时四　例题讲解

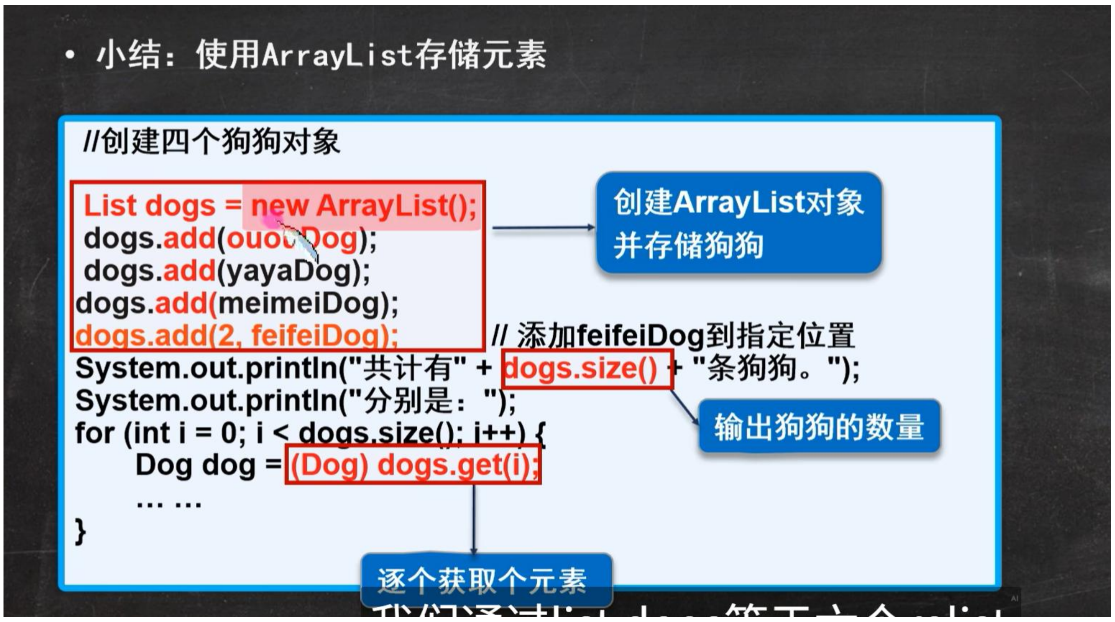
```
解：class Minus implements ICompute{
        private int n;
        private int m;
        private int ret;
        public int compute(int n,int m){
                this.n=n;
                this.m=m;
                this.ret=n-m;
                System.out.println("n-m is "+ret);
                return ret;
        }}
    class Mul implements ICompute{
            private int n;
            private int m;
            private int ret;
            public int compute(int n,int m){
                    this.n=n;
                    this.m=m;
                    this.ret=n*m;
                    System.out.println("n*m is "+ret);
                    return ret;
            }}
```



课时四　例题讲解

```
解：class Div implements ICompute{
        private int n;
        private int m;
        private int ret;
        public int compute(int n,int m){
                this.n=n;
                this.m=m;
                this.ret=n/m;
                System.out.println("n/m is "+ret);
                return ret;
        }
}
    class Test{
            public static void main(String []args){
                    UseCompute fun=new UseCompute();
                    fun.UseCom(new Add(),8,2);
                    fun.UseCom(new Minus(),8,2);
                    fun.UseCom(new Mul(),8,2);
                    fun.UseCom(new Div(),8,2);
            }
    }
```

LIST!!!!!!!!!!!!!!!



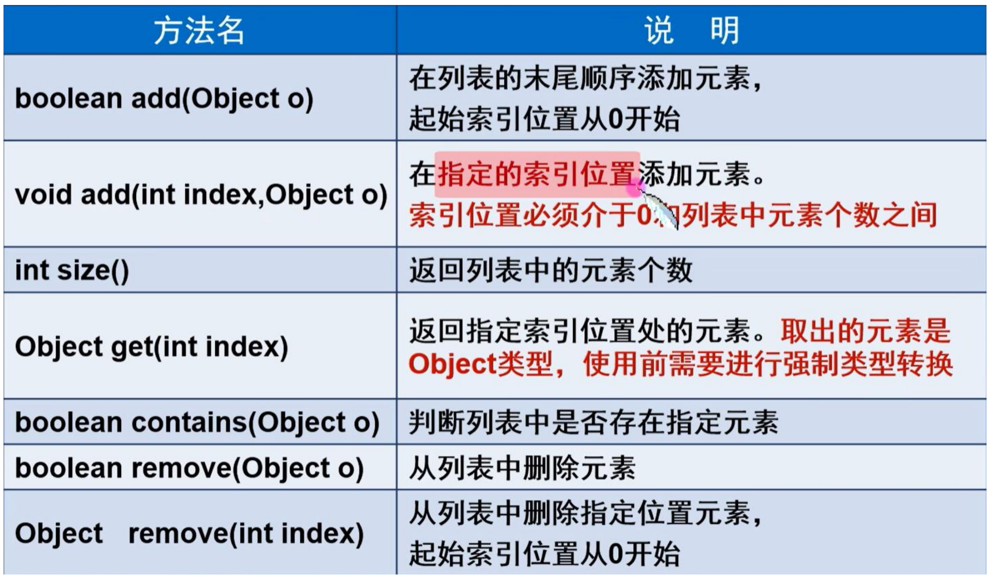
- 小结：使用ArrayList存储元素

```
//创建四个狗狗对象

List dogs = new ArrayList();
dogs.add(ououDog);
dogs.add(yayaDog);
dogs.add(meimeiDog);
dogs.add(2, feifeiDog);  // 添加feifeiDog到指定位置
System.out.println("共计有" + dogs.size() + "条狗狗。");
System.out.println("分别是：");
for (int i = 0; i < dogs.size(); i++) {
    Dog dog = (Dog) dogs.get(i);
    … …
}
```

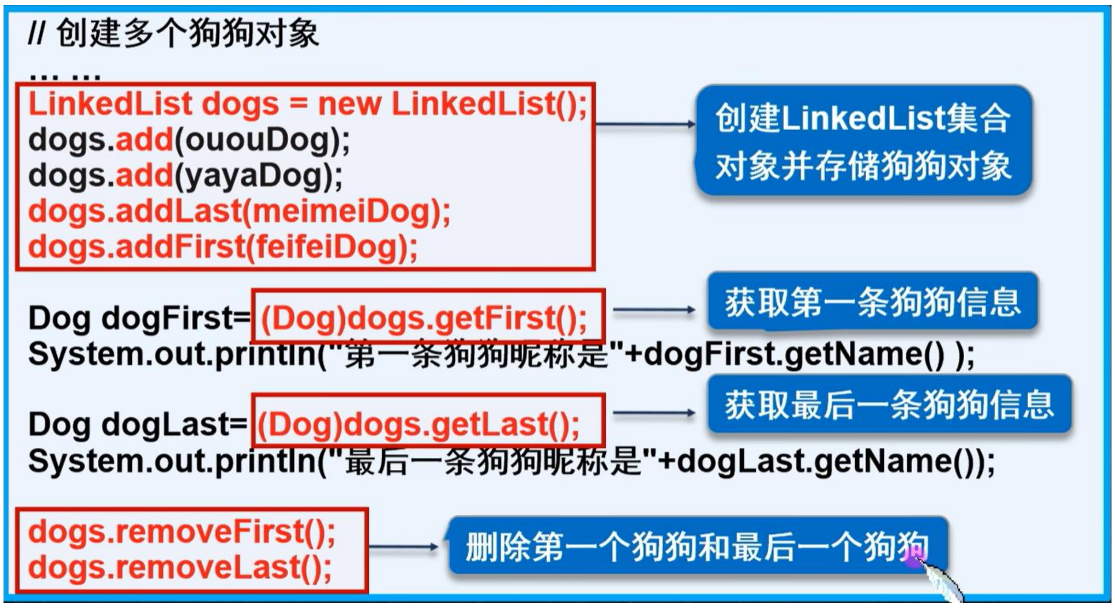
创建ArrayList对象并存储狗狗

输出狗狗的数量

逐个获取个元素



方法名	说　明
boolean add(Object o)	在列表的末尾顺序添加元素，起始索引位置从0开始
void add(int index,Object o)	在指定的索引位置添加元素。索引位置必须介于0和列表中元素个数之间
int size()	返回列表中的元素个数
Object get(int index)	返回指定索引位置处的元素。取出的元素是Object类型，使用前需要进行强制类型转换
boolean contains(Object o)	判断列表中是否存在指定元素
boolean remove(Object o)	从列表中删除元素
Object　remove(int index)	从列表中删除指定位置元素，起始索引位置从0开始



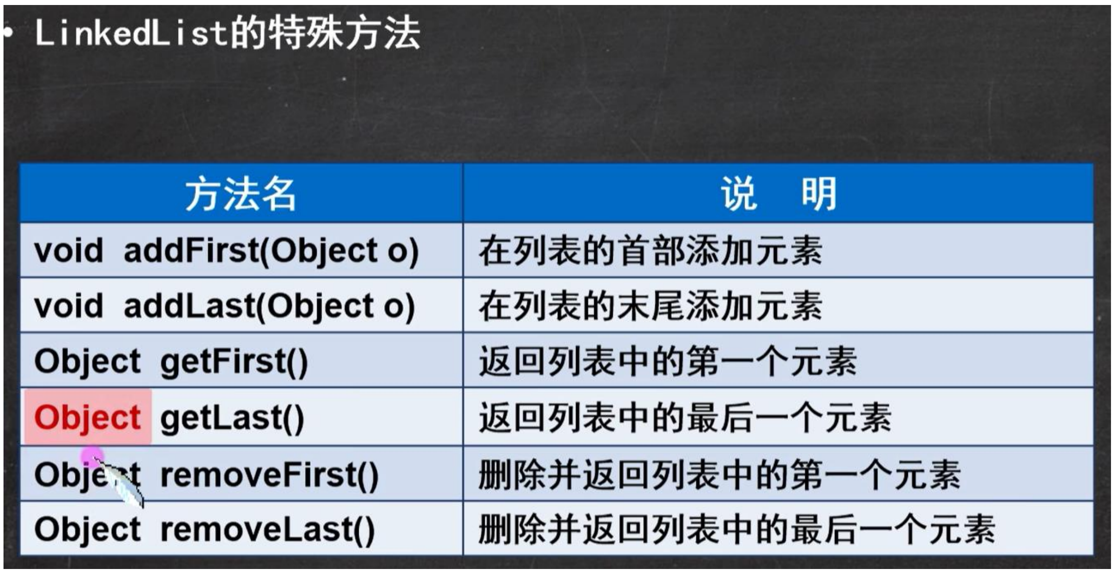
```
// 创建多个狗狗对象
… …
LinkedList dogs = new LinkedList();
dogs.add(ououDog);
dogs.add(yayaDog);
dogs.addLast(meimeiDog);
dogs.addFirst(feifeiDog);

Dog dogFirst= (Dog)dogs.getFirst();
System.out.println("第一条狗狗昵称是"+dogFirst.getName() );

Dog dogLast= (Dog)dogs.getLast();
System.out.println("最后一条狗狗昵称是"+dogLast.getName());

dogs.removeFirst();
dogs.removeLast();
```

创建LinkedList集合对象并存储狗狗对象

获取第一条狗狗信息

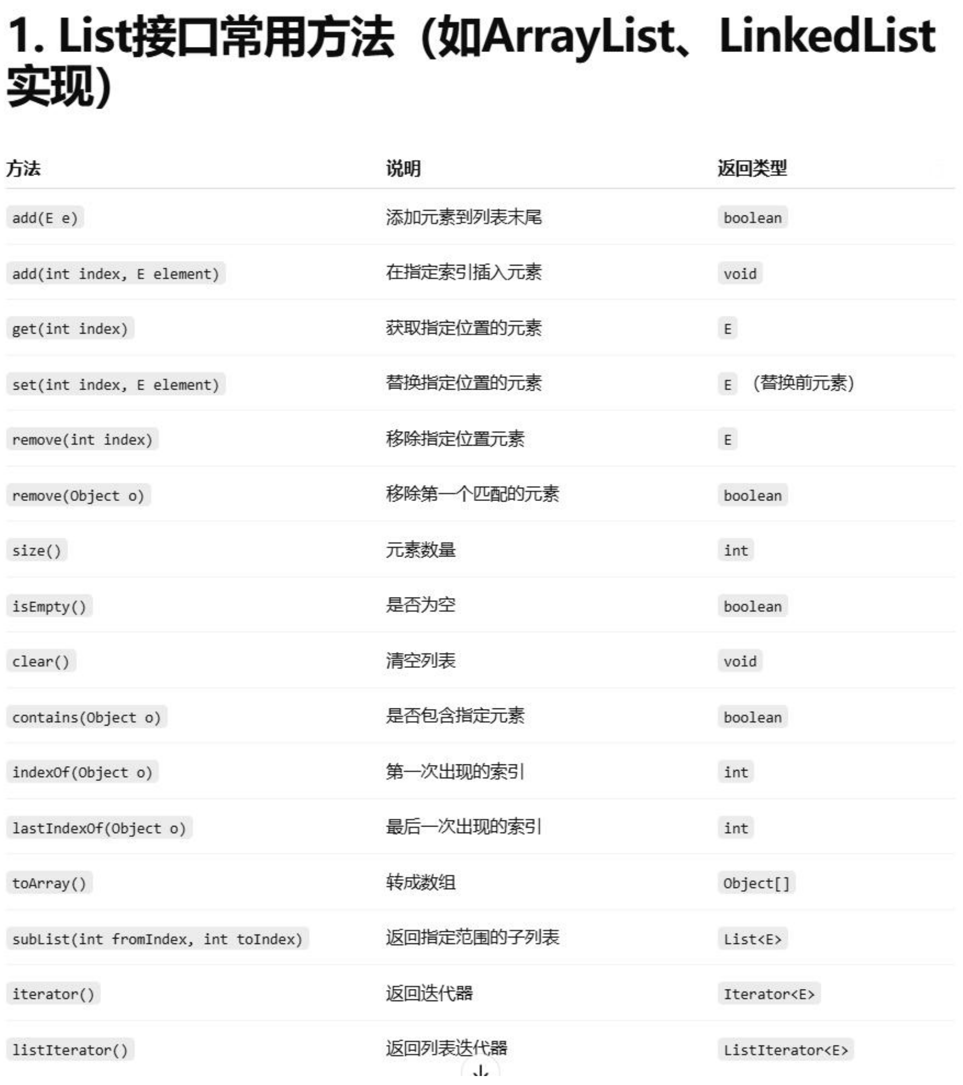
获取最后一条狗狗信息

删除第一个狗狗和最后一个狗狗



• LinkedList的特殊方法

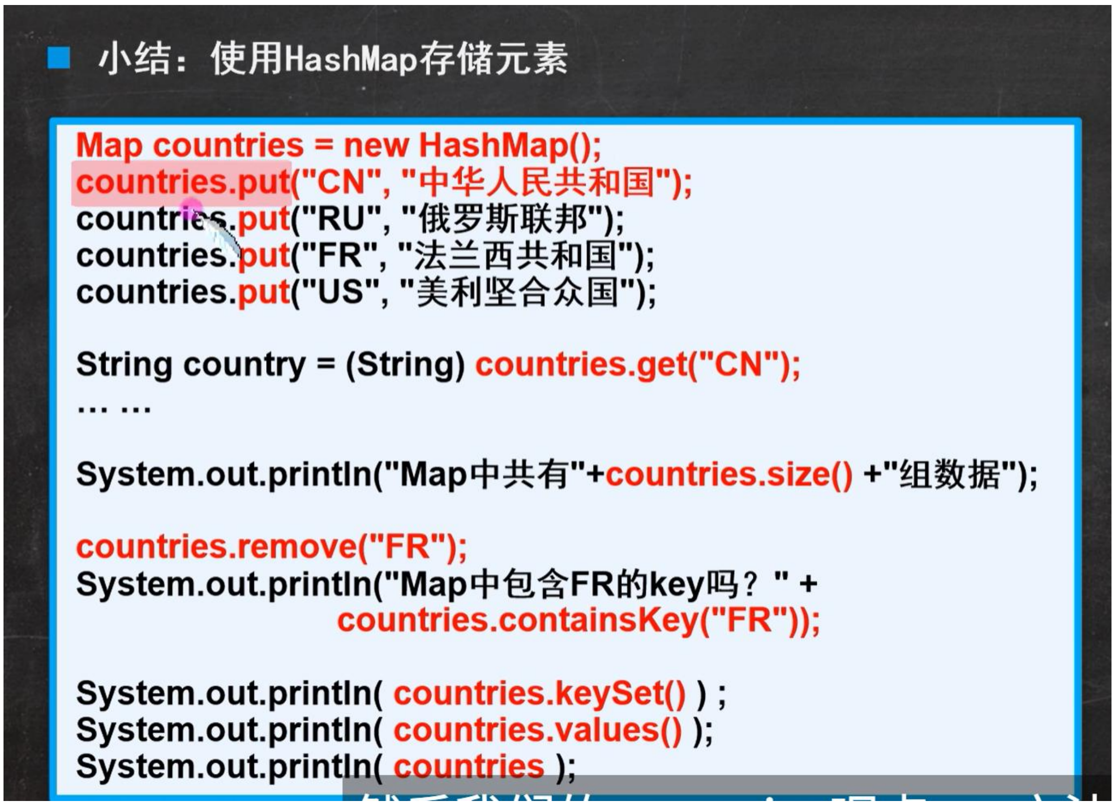
方法名	说　明
void addFirst(Object o)	在列表的首部添加元素
void addLast(Object o)	在列表的末尾添加元素
Object getFirst()	返回列表中的第一个元素
Object getLast()	返回列表中的最后一个元素
Object removeFirst()	删除并返回列表中的第一个元素
Object removeLast()	删除并返回列表中的最后一个元素



1. List接口常用方法（如ArrayList、LinkedList实现）

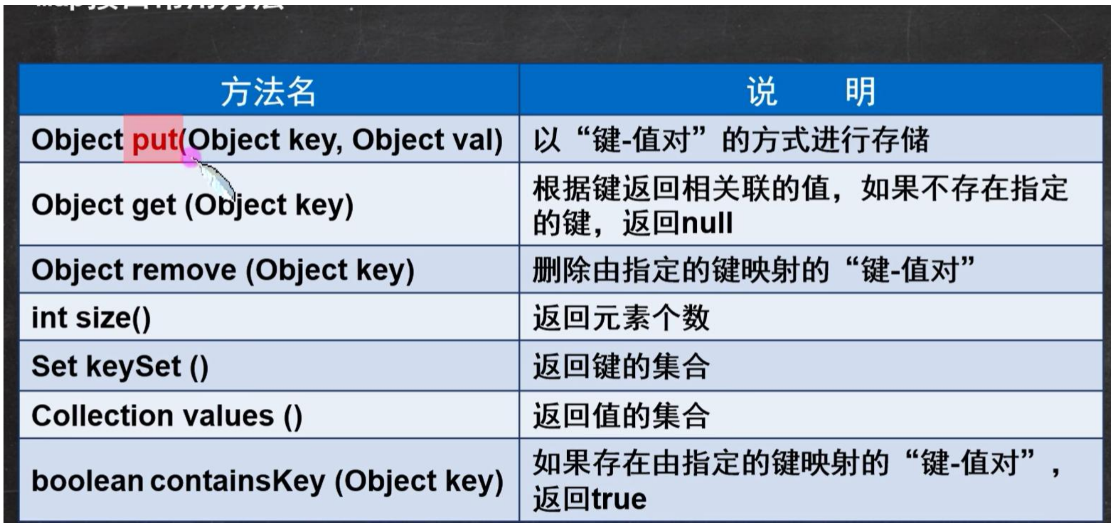
方法	说明	返回类型
`add(E e)`	添加元素到列表末尾	`boolean`
`add(int index, E element)`	在指定索引插入元素	`void`
`get(int index)`	获取指定位置的元素	`E`
`set(int index, E element)`	替换指定位置的元素	`E`（替换前元素）
`remove(int index)`	移除指定位置元素	`E`
`remove(Object o)`	移除第一个匹配的元素	`boolean`
`size()`	元素数量	`int`
`isEmpty()`	是否为空	`boolean`
`clear()`	清空列表	`void`
`contains(Object o)`	是否包含指定元素	`boolean`
`indexOf(Object o)`	第一次出现的索引	`int`
`lastIndexOf(Object o)`	最后一次出现的索引	`int`
`toArray()`	转成数组	`Object[]`
`subList(int fromIndex, int toIndex)`	返回指定范围的子列表	`List<E>`
`iterator()`	返回迭代器	`Iterator<E>`
`listIterator()`	返回列表迭代器	`ListIterator<E>`

MAP!!!!!!!!!!!!!!!

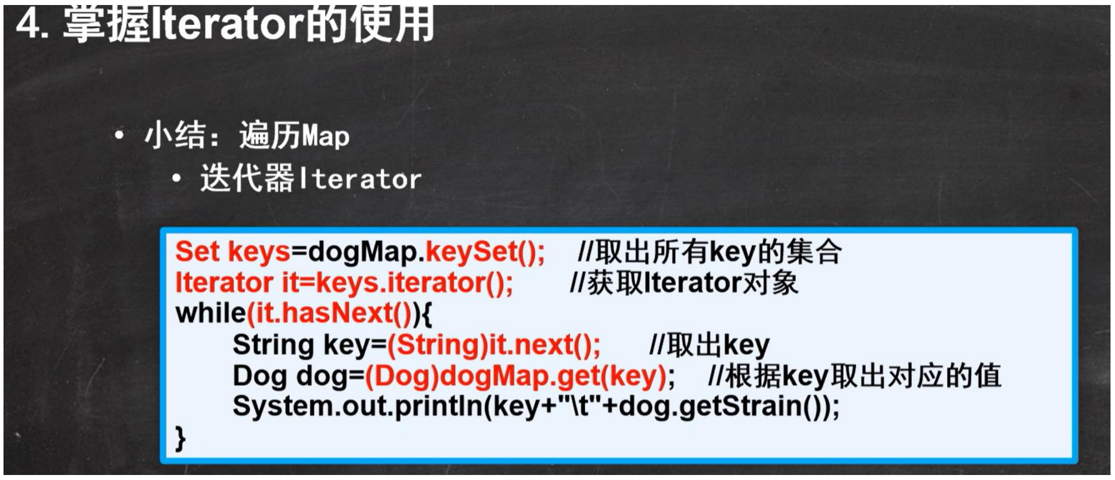


小结：使用HashMap存储元素

```
Map countries = new HashMap();
countries.put("CN", "中华人民共和国");
countries.put("RU", "俄罗斯联邦");
countries.put("FR", "法兰西共和国");
countries.put("US", "美利坚合众国");

String country = (String) countries.get("CN");
… …

System.out.println("Map中共有"+countries.size() +"组数据");

countries.remove("FR");
System.out.println("Map中包含FR的key吗？" +
                    countries.containsKey("FR"));

System.out.println( countries.keySet() ) ;
System.out.println( countries.values() );
System.out.println( countries );
```



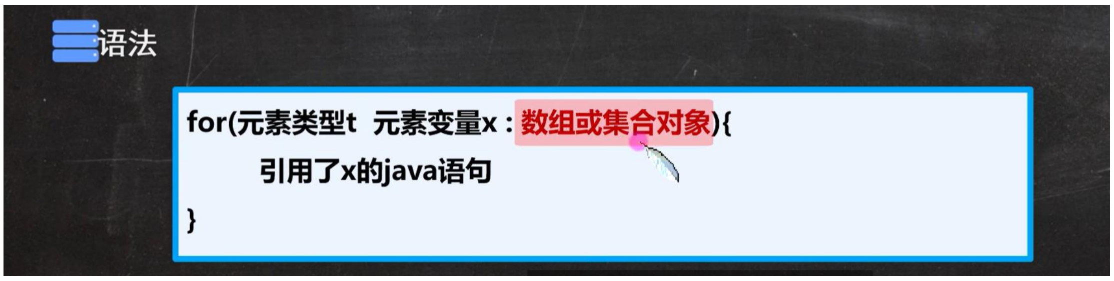
方法名	说　明
Object put(Object key, Object val)	以“键-值对”的方式进行存储
Object get (Object key)	根据键返回相关联的值，如果不存在指定的键，返回null
Object remove (Object key)	删除由指定的键映射的“键-值对”
int size()	返回元素个数
Set keySet ()	返回键的集合
Collection values ()	返回值的集合
boolean containsKey (Object key)	如果存在由指定的键映射的“键-值对”，返回true



4. 掌握Iterator的使用

- 小结：遍历Map
  - 迭代器Iterator

```
Set keys=dogMap.keySet();    //取出所有key的集合
Iterator it=keys.iterator();    //获取Iterator对象
while(it.hasNext()){
    String key=(String)it.next();    //取出key
    Dog dog=(Dog)dogMap.get(key);    //根据key取出对应的值
    System.out.println(key+"\t"+dog.getStrain());
}
```



语法

```
for(元素类型t  元素变量x : 数组或集合对象){
        引用了x的java语句
}
```

2. Map接口常用方法（如HashMap, TreeMap实现）

方法	说明	返回类型
`put(K key, V value)`	添加或更新键值对	`V`（旧值，如无返回null）
`get(Object key)`	根据键获取对应的值	`V`
`remove(Object key)`	删除键及对应的值	`V`
`containsKey(Object key)`	键是否存在	`boolean`
`containsValue(Object value)`	值是否存在	`boolean`
`size()`	键值对数量	`int`
`isEmpty()`	是否空	`boolean`
`clear()`	清空	`void`
`keySet()`	返回所有键的集合	`Set<K>`
`values()`	返回所有值的集合	`Collection<V>`
`entrySet()`	返回所有键值映射（Entry）的集合	`Set<Map.Entry<K,V>>`
`putAll(Map<? extends K, ? extends V> m)`	批量添加所有键值对	`void`
`getOrDefault(Object key, V defaultValue)`	如果键不存在返回默认值	`V`
`replace(K key, V value)`	替换指定键的值	`V`

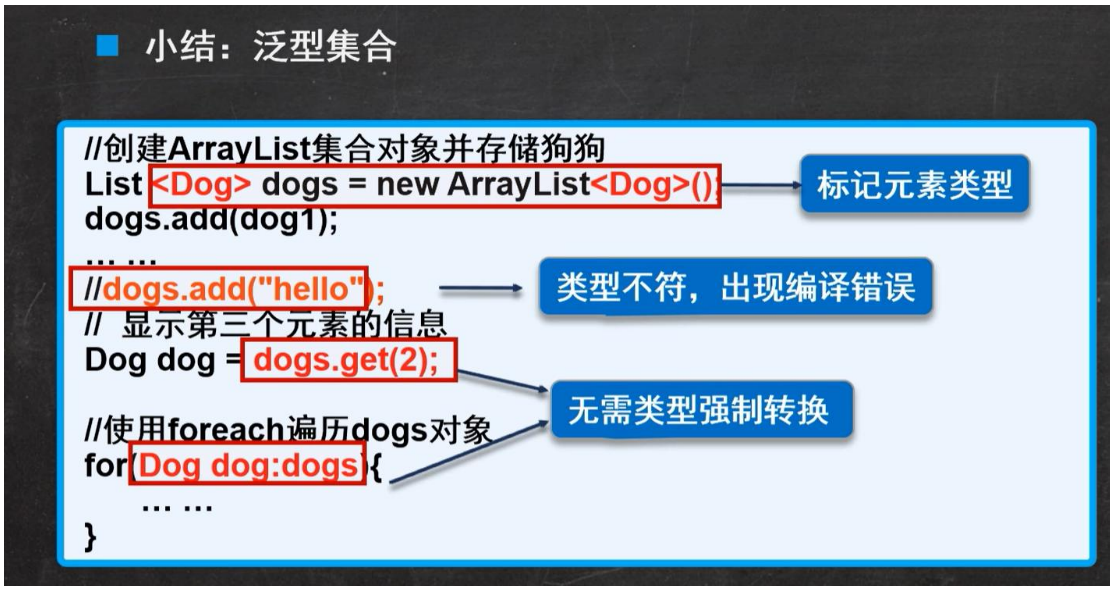
3. Set接口常用方法（如HashSet, TreeSet实现）

方法	说明	返回类型
`add(E e)`	添加元素，如果元素不重复则返回true	`boolean`
`remove(Object o)`	移除指定元素	`boolean`
`contains(Object o)`	判断是否包含元素	`boolean`
`size()`	元素数量	`int`
`isEmpty()`	是否空	`boolean`
`clear()`	清空	`void`
`iterator()`	返回元素迭代器	`Iterator<E>`
`toArray()`	转成数组	`Object[]`

4. 简单总结（常用的都是这些）

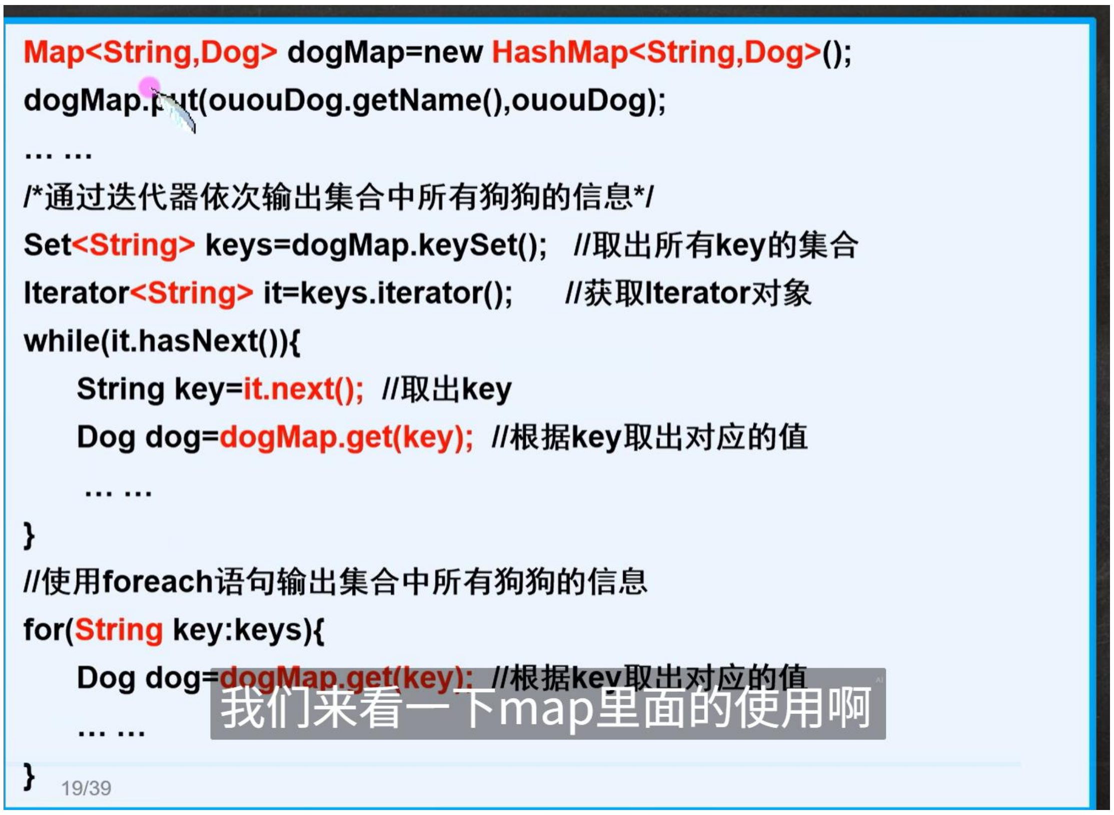
接口	典型实现类	常见用途	例子方法
List	ArrayList, LinkedList	有序集合，可按索引访问	`get()`, `add()`, `remove()`
Map	HashMap, TreeMap	键值映射集合	`put()`, `get()`, `remove()`
Set	HashSet, TreeSet	不重复元素集合	`add()`, `remove()`, `contains()`

泛型！！！



小结：泛型集合

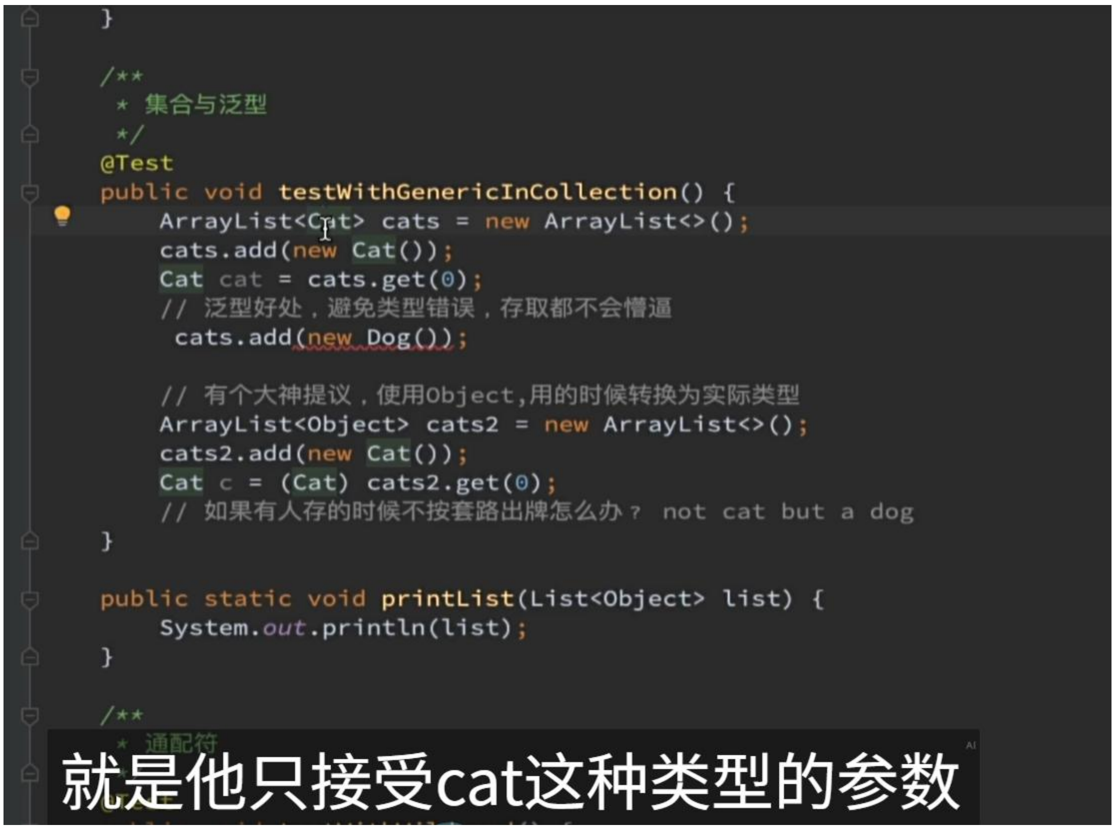
```
//创建ArrayList集合对象并存储狗狗
List<Dog> dogs = new ArrayList<Dog>();
dogs.add(dog1);
… …
//dogs.add("hello");
// 显示第三个元素的信息
Dog dog = dogs.get(2);

//使用foreach遍历dogs对象
for(Dog dog:dogs){
   … …
}
```



```
Map<String,Dog> dogMap=new HashMap<String,Dog>();
dogMap.put(ououDog.getName(),ououDog);
… …
/*通过迭代器依次输出集合中所有狗狗的信息*/
Set<String> keys=dogMap.keySet();   //取出所有key的集合
Iterator<String> it=keys.iterator();     //获取Iterator对象
while(it.hasNext()){
     String key=it.next();  //取出key
     Dog dog=dogMap.get(key);  //根据key取出对应的值
      … …
}
//使用foreach语句输出集合中所有狗狗的信息
for(String key:keys){
     Dog dog=dogMap.get(key);  //根据key取出对应的值
     … …
}
```

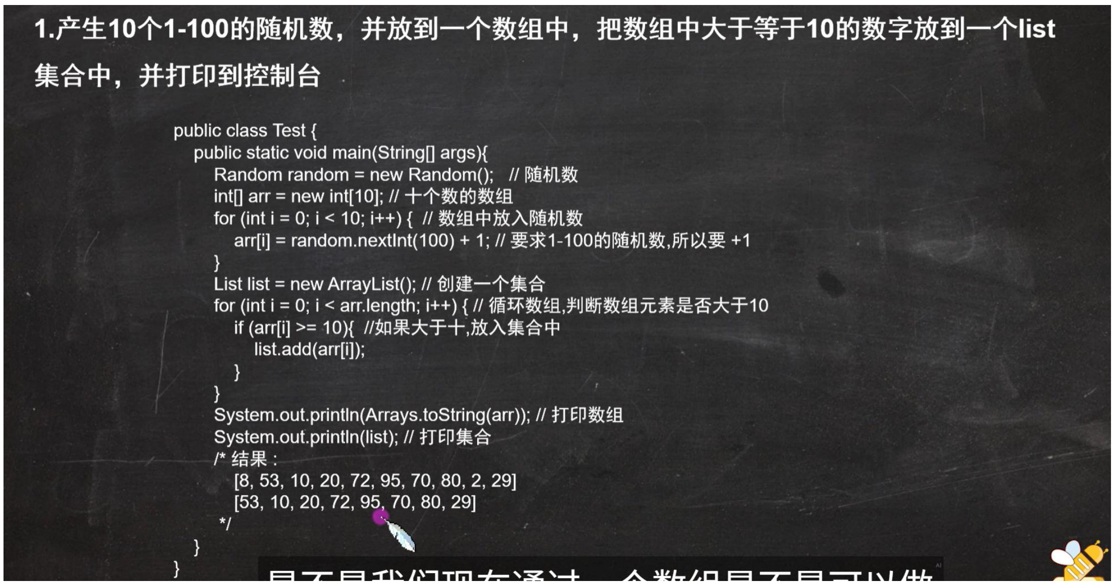
19/39

我们来看一下map里面的使用啊



```
    }

    /**
     * 集合与泛型
     */
    @Test
    public void testWithGenericInCollection() {
        ArrayList<Cat> cats = new ArrayList<>();
        cats.add(new Cat());
        Cat cat = cats.get(0);
        // 泛型好处，避免类型错误，存取都不会懵逼
         cats.add(new Dog());

        // 有个大神提议，使用Object,用的时候转换为实际类型
        ArrayList<Object> cats2 = new ArrayList<>();
        cats2.add(new Cat());
        Cat c = (Cat) cats2.get(0);
        // 如果有人存的时候不按套路出牌怎么办？ not cat but a dog
    }

    public static void printList(List<Object> list) {
        System.out.println(list);
    }

    /**
     * 通配符
```

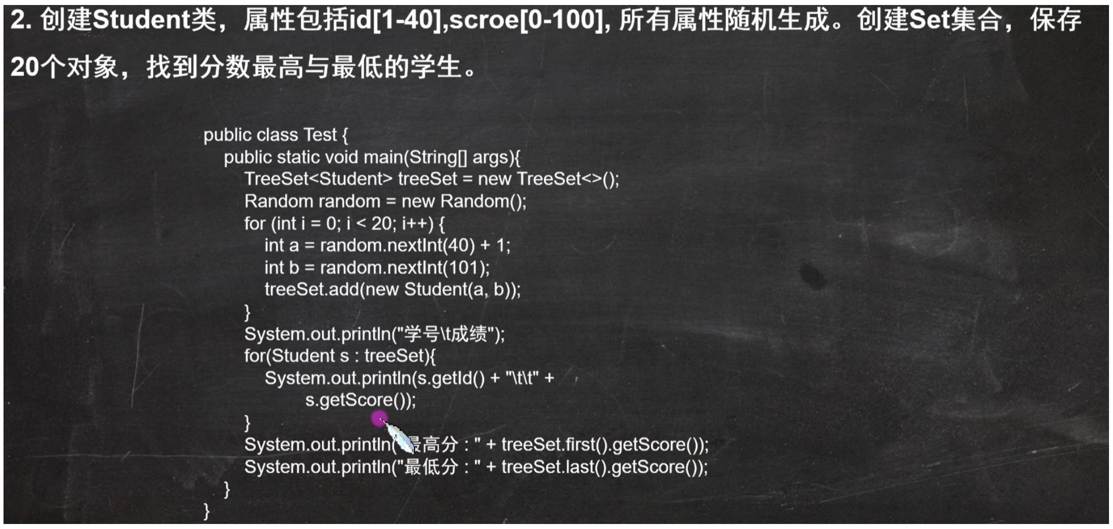
就是他只接受cat这种类型的参数



1.产生10个1-100的随机数，并放到一个数组中，把数组中大于等于10的数字放到一个list集合中，并打印到控制台

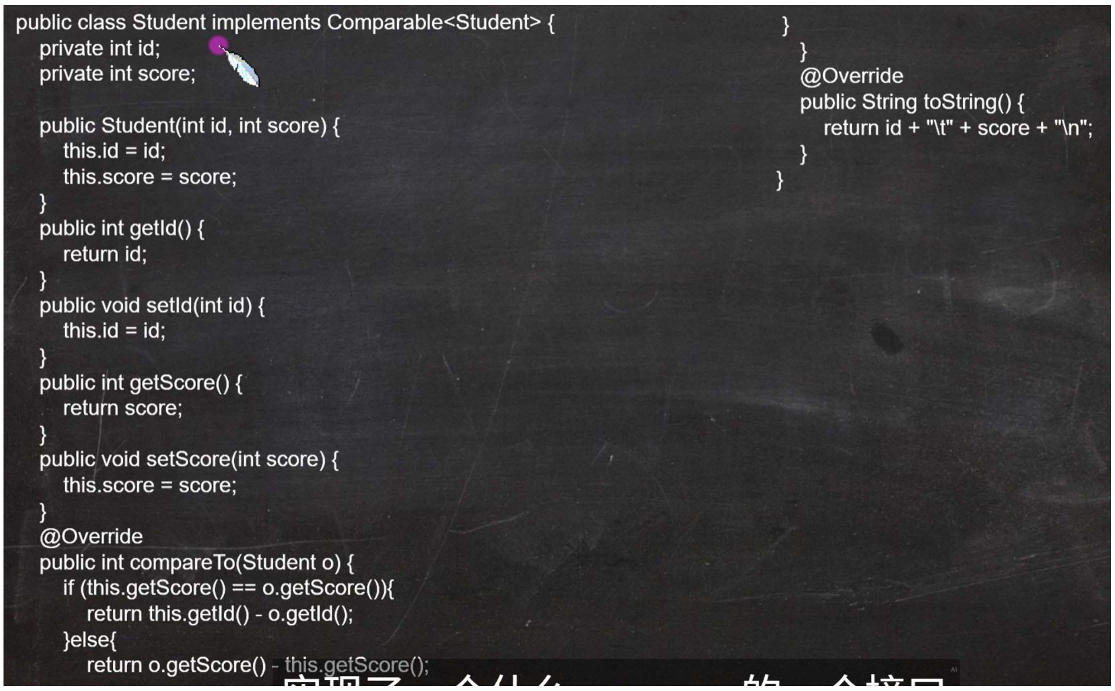
```
public class Test {
    public static void main(String[] args){
        Random random = new Random();   // 随机数
        int[] arr = new int[10]; // 十个数的数组
        for (int i = 0; i < 10; i++) {  // 数组中放入随机数
            arr[i] = random.nextInt(100) + 1; // 要求1-100的随机数,所以要 +1
        }
        List list = new ArrayList(); // 创建一个集合
        for (int i = 0; i < arr.length; i++) { // 循环数组,判断数组元素是否大于10
            if (arr[i] >= 10){  //如果大于十,放入集合中
                list.add(arr[i]);
            }
        }
        System.out.println(Arrays.toString(arr)); // 打印数组
        System.out.println(list); // 打印集合
        /* 结果 :
            [8, 53, 10, 20, 72, 95, 70, 80, 2, 29]
            [53, 10, 20, 72, 95, 70, 80, 29]
         */
    }
}
```

是不是我们现在通过一个数组是不是可以做

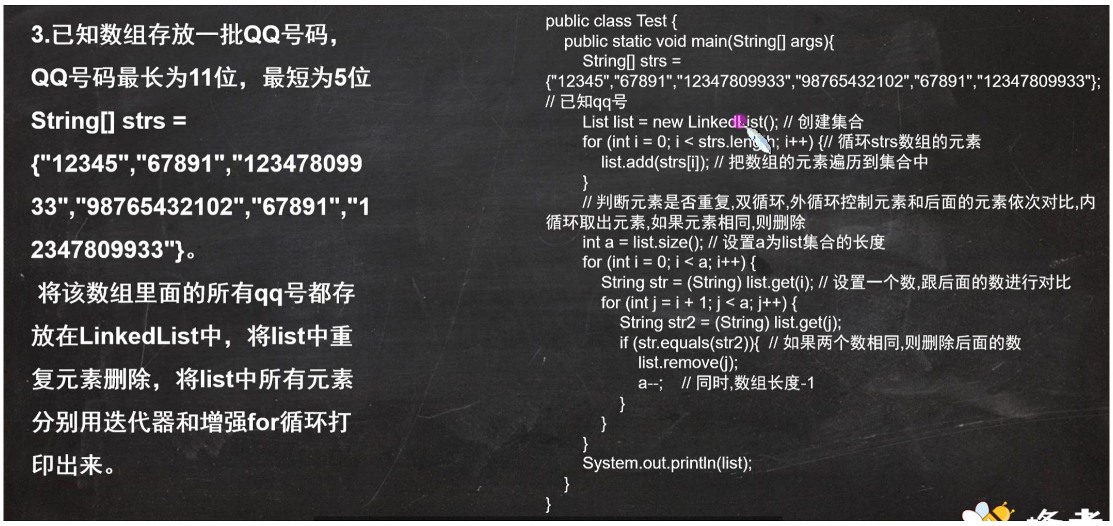


2. 创建Student类，属性包括id[1-40],scroe[0-100], 所有属性随机生成。创建Set集合，保存20个对象，找到分数最高与最低的学生。

```
public class Test {
    public static void main(String[] args){
        TreeSet<Student> treeSet = new TreeSet<>();
        Random random = new Random();
        for (int i = 0; i < 20; i++) {
            int a = random.nextInt(40) + 1;
            int b = random.nextInt(101);
            treeSet.add(new Student(a, b));
        }
        System.out.println("学号\t成绩");
        for(Student s : treeSet){
            System.out.println(s.getId() + "\t\t" +
                    s.getScore());
        }
        System.out.println("最高分 : " + treeSet.first().getScore());
        System.out.println("最低分 : " + treeSet.last().getScore());
    }
}
```



```java
public class Student implements Comparable<Student> {
    private int id;
    private int score;

    public Student(int id, int score) {
        this.id = id;
        this.score = score;
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public int getScore() {
        return score;
    }
    public void setScore(int score) {
        this.score = score;
    }
    @Override
    public int compareTo(Student o) {
        if (this.getScore() == o.getScore()){
            return this.getId() - o.getId();
        }else{
            return o.getScore() - this.getScore();
        }
    }
    @Override
    public String toString() {
        return id + "\t" + score + "\n";
    }
}
```



3.已知数组存放一批QQ号码，QQ号码最长为11位，最短为5位 String[] strs = {"12345","67891","12347809933","98765432102","67891","12347809933"}。

将该数组里面的所有qq号都存放在LinkedList中，将list中重复元素删除，将list中所有元素分别用迭代器和增强for循环打印出来。

```java
public class Test {
    public static void main(String[] args){
        String[] strs =
{"12345","67891","12347809933","98765432102","67891","12347809933"};
// 已知qq号
        List list = new LinkedList(); // 创建集合
        for (int i = 0; i < strs.length; i++) {// 循环strs数组的元素
            list.add(strs[i]); // 把数组的元素遍历到集合中
        }
        // 判断元素是否重复,双循环,外循环控制元素和后面的元素依次对比,内
循环取出元素,如果元素相同,则删除
        int a = list.size(); // 设置a为list集合的长度
        for (int i = 0; i < a; i++) {
            String str = (String) list.get(i); // 设置一个数,跟后面的数进行对比
            for (int j = i + 1; j < a; j++) {
                String str2 = (String) list.get(j);
                if (str.equals(str2)){  // 如果两个数相同,则删除后面的数
                    list.remove(j);
                    a--;    // 同时,数组长度-1
                }
            }
        }
        System.out.println(list);
    }
}
```