

深圳大学实验报告

课程名称：Java 程序设计

实验项目名称：课程实验 3：常用集合类和线程

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：潘微科

报告人：曹秦鲁 学号：202315020 班级：国际班

实验时间：2024 年 11 月 1 日（周五）-2024 年 11 月 20 日（周三）

实验报告提交时间：

教务部制

实验目的与要求:

实验目的: 掌握常用的集合类, 能够较为熟练的查阅 Java 提供的常见的类, 并进行程序设计, 掌握 Java 程序设计中的线程同步等技术。

实验要求:

Part 1 (25 分)

(1.1). 编写 Java 应用程序, 实现浮点数 (float) 稀疏矩阵的乘法和加法运算, 其中稀疏矩阵是指矩阵中的绝大部分元素的值为 0。在命令行读入和输出矩阵中的元素的时候, 采用三元组的方式, 即行号、列号和数值, 例如“第 2 行、第 3 列、数值 3.2”表示为 2 3 3.2。要求以 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和一个 4*5 的矩阵 ([0 1.1 1.5 0 0; 0 0 3.2 0 0; 0 1.3 0 0 -3.2; -1.0 6.2 0 0 0]) 相乘, 以及 1 个 3*4 的矩阵 ([1.2 0 0 0; 0 0 3.1 0; 0 0 0 2.2]) 和 1 个 3*4 的矩阵 ([1.2 -3 -5.3 0; 0 1 0.1 -0.4; 2 2 1 0.2]) 相加为例, 在报告中附上程序截图、完整的运行结果截图和简要文字说明。(10 分)

```
import java.util.*;

public class SparseMatrix {

    // 定义稀疏矩阵, 使用Map 表示
    private Map<Integer, Map<Integer, Float>> matrix;

    // 构造函数初始化矩阵
    public SparseMatrix() {
        matrix = new HashMap<>();
    }

    // 向矩阵中添加一个元素
    public void addElement(int row, int col, float value) {
        if (value != 0) {
            matrix.putIfAbsent(row, new HashMap<>());
            matrix.get(row).put(col, value);
        }
    }

    // 输出矩阵的稀疏格式 (三元组形式)
    public void printMatrix() {
        for (Map.Entry<Integer, Map<Integer, Float>> rowEntry : matrix.entrySet()) {
            int row = rowEntry.getKey();
            for (Map.Entry<Integer, Float> colEntry : rowEntry.getValue().entrySet()) {
                int col = colEntry.getKey();
                float value = colEntry.getValue();
                System.out.println(row + " " + col + " " + value);
            }
        }
    }
}
```

```

// 矩阵加法
public SparseMatrix add(SparseMatrix other) {
    SparseMatrix result = new SparseMatrix();

    // 复制当前矩阵中的元素
    for (Map.Entry<Integer, Map<Integer, Float>> rowEntry : this.matrix.entrySet())
    {
        int row = rowEntry.getKey();
        for (Map.Entry<Integer, Float> colEntry : rowEntry.getValue().entrySet()) {
            int col = colEntry.getKey();
            float value = colEntry.getValue();
            result.addElement(row, col, value);
        }
    }

    // 添加另一个矩阵中的元素
    for (Map.Entry<Integer, Map<Integer, Float>> rowEntry : other.matrix.entrySet())
    {
        int row = rowEntry.getKey();
        for (Map.Entry<Integer, Float> colEntry : rowEntry.getValue().entrySet()) {
            int col = colEntry.getKey();
            float value = colEntry.getValue();
            result.addElement(row, col, result.matrix.get(row).getOrDefault(col, 0f)
+ value);
        }
    }

    return result;
}

// 矩阵乘法
public SparseMatrix multiply(SparseMatrix other) {
    SparseMatrix result = new SparseMatrix();

    // 遍历当前矩阵的行
    for (Map.Entry<Integer, Map<Integer, Float>> rowEntry : this.matrix.entrySet())
    {
        int row = rowEntry.getKey();
        // 遍历另一个矩阵的列
        for (Map.Entry<Integer, Map<Integer, Float>> otherRowEntry :
other.matrix.entrySet()) {
            int col = otherRowEntry.getKey();
            float sum = 0;

```

```

        // 对于每一行，计算与另一矩阵列的点积
        for (Map.Entry<Integer, Float> colEntry : rowEntry.getValue().entrySet())
        {
            int currentCol = colEntry.getKey();
            float value = colEntry.getValue();
            if (otherRowEntry.getValue().containsKey(currentCol)) {
                sum += value * otherRowEntry.getValue().get(currentCol);
            }
        }

        if (sum != 0) {
            result.addElement(row, col, sum);
        }
    }
}

return result;
}

public static void main(String[] args) {
    // 示例矩阵 1
    SparseMatrix matrix1 = new SparseMatrix();
    matrix1.addElement(1, 1, 1.2f);
    matrix1.addElement(2, 3, 3.1f);
    matrix1.addElement(3, 4, 2.2f);

    // 示例矩阵 2
    SparseMatrix matrix2 = new SparseMatrix();
    matrix2.addElement(1, 2, 1.1f);
    matrix2.addElement(1, 3, 1.5f);
    matrix2.addElement(2, 3, 3.2f);
    matrix2.addElement(3, 2, 1.3f);
    matrix2.addElement(3, 5, -3.2f);
    matrix2.addElement(4, 1, -1.0f);
    matrix2.addElement(4, 2, 6.2f);

    // 矩阵乘法
    SparseMatrix product = matrix1.multiply(matrix2);
    System.out.println("Matrix multiplication result:");
    product.printMatrix();

    // 示例矩阵 3
    SparseMatrix matrix3 = new SparseMatrix();
    matrix3.addElement(1, 1, 1.2f);

```

```

        matrix3.addElement(2, 3, 3.1f);
        matrix3.addElement(3, 4, 2.2f);

        // 示例矩阵4
        SparseMatrix matrix4 = new SparseMatrix();
        matrix4.addElement(1, 1, 1.2f);
        matrix4.addElement(1, 2, -3f);
        matrix4.addElement(1, 3, -5.3f);
        matrix4.addElement(2, 2, 1f);
        matrix4.addElement(2, 3, 0.1f);
        matrix4.addElement(2, 4, -0.4f);
        matrix4.addElement(3, 1, 2f);
        matrix4.addElement(3, 2, 2f);
        matrix4.addElement(3, 3, 1f);
        matrix4.addElement(3, 4, 0.2f);

        // 矩阵加法
        SparseMatrix sum = matrix3.add(matrix4);
        System.out.println("Matrix addition result:");
        sum.printMatrix();
    }
}

```

```

"C:\Program Files\Eclipse Adoptium\jdk-21.0.4.
Matrix multiplication result:
1 4 -1.2
2 1 4.6499996
2 2 9.92
Matrix addition result:
1 1 2.4
1 2 -3.0
1 3 -5.3
2 2 1.0
2 3 3.1999998
2 4 -0.4
3 1 2.0
3 2 2.0
3 3 1.0
3 4 2.4

Process finished with exit code 0

```

代码解释

SparseMatrix 类:

matrix 是一个存储稀疏矩阵的内部数据结构, 使用了 `Map<Integer, Map<Integer, Float>>` 形式, 外层 `Map` 代表行, 内层 `Map` 代表列。

addElement(int row, int col, float value): 向矩阵中添加元素, 只有当值不为零时才添加。

`printMatrix()`: 按三元组格式输出矩阵元素。

`add(SparseMatrix other)`: 实现两个稀疏矩阵的加法。

`multiply(SparseMatrix other)`: 实现两个稀疏矩阵的乘法。

主函数:

`matrix1`、`matrix2` 分别是示例中的稀疏矩阵 1 和稀疏矩阵 2, 进行矩阵乘法操作。

`matrix3`、`matrix4` 分别是示例中的稀疏矩阵 3 和稀疏矩阵 4, 进行矩阵加法操作。

(1.2). 有 12 个国家 (美国、中华人民共和国、德国、日本、英国、印度、法国、意大利、加拿大、韩国、以色列、俄罗斯), 其属性有 `name`、`GDP2023` 和 `Olympics2024`, 分别表示国家名称、世界银行公布的 2023 年的国内生产总值 (单位: 百万美元) 和在 2024 年巴黎奥会上获得的奖牌数量。

编写一个 Java 应用程序, 要求使用 `TreeSet`。(i) 按照 `Olympics2024` 从大到小排序输出这些国家的信息; (ii) 按照 `GDP2023` 从大到小排序输出这些国家的信息。要求以上(i)和(ii)两小题都通过以下两种方式实现: 通过实现 `Comparator` 接口或通过实现 `Comparable` 接口。在报告中附上程序截图、完整的运行结果截图和详细的文字说明。(15 分)

```
import java.util.Comparator;import java.util.TreeSet;

public class CountryApp {

    // 创建一个Country 类来表示国家
    static class Country implements Comparable<Country> {
        String name;
        double GDP2023;
        int Olympics2024;

        public Country(String name, double GDP2023, int Olympics2024) {
            this.name = name;
            this.GDP2023 = GDP2023;
            this.Olympics2024 = Olympics2024;
        }

        // 输出国家信息
        public String toString() {
            return "Country: " + name + ", GDP2023: " + GDP2023 + " Million USD, Olympics2024
Medals: " + Olympics2024;
        }

        // 使用GDP2023 进行排序 (降序)
        @Override
        public int compareTo(Country other) {
            // 按照 GDP2023 从大到小排序
            return Double.compare(other.GDP2023, this.GDP2023);
        }
    }
}
```

```

    }
}

// Olympics2024 排序的比较器（降序）
static class OlympicsComparator implements Comparator<Country> {
    @Override
    public int compare(Country o1, Country o2) {
        // 按照 Olympics2024 从大到小排序
        return Integer.compare(o2.Olympics2024, o1.Olympics2024);
    }
}

// GDP2023 排序的比较器（降序）
static class GDPComparator implements Comparator<Country> {
    @Override
    public int compare(Country o1, Country o2) {
        // 按照 GDP2023 从大到小排序
        return Double.compare(o2.GDP2023, o1.GDP2023);
    }
}

public static void main(String[] args) {
    // 创建 12 个国家的实例，使用提供的真实数据
    Country[] countries = {
        new Country("美国", 26704170, 113),
        new Country("中华人民共和国", 18084810, 93),
        new Country("德国", 5072707, 48),
        new Country("日本", 4231143, 58),
        new Country("英国", 3070677, 51),
        new Country("印度", 3730314, 28),
        new Country("法国", 3171209, 40),
        new Country("意大利", 2610766, 38),
        new Country("加拿大", 2184804, 32),
        new Country("韩国", 1968019, 24),
        new Country("以色列", 554173, 9),
        new Country("俄罗斯", 1799000, 50)
    };

    // 第一种方式：通过实现 Comparable 接口（按 GDP2023 排序）
    TreeSet<Country> setByGDP = new TreeSet<>();
    for (Country country : countries) {
        setByGDP.add(country);
    }
}

```

```
System.out.println("排序方式 1: 按 GDP2023 从大到小排序输出: ");
for (Country country : setByGDP) {
    System.out.println(country);
}

// 第二种方式: 通过实现 Comparator 接口 (按 Olympics2024 排序)
TreeSet<Country> setByOlympics = new TreeSet<>(new OlympicsComparator());
for (Country country : countries) {
    setByOlympics.add(country);
}

System.out.println("\n 排序方式 2: 按 Olympics2024 奖牌数量从大到小排序输出: ");
for (Country country : setByOlympics) {
    System.out.println(country);
}

// 第三种方式: 通过实现 Comparator 接口 (按 GDP2023 排序)
TreeSet<Country> setByGDPComparator = new TreeSet<>(new GDPComparator());
for (Country country : countries) {
    setByGDPComparator.add(country);
}

System.out.println("\n 排序方式 3: 按 GDP2023 从大到小排序输出 (使用 Comparator): ");
for (Country country : setByGDPComparator) {
    System.out.println(country);
}
}
```



```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.4.7-hotspot\bin\java.exe" "-javaag
排序方式1: 按GDP2023从大到小排序输出:
Country: 美国, GDP2023: 2.670417E7 Million USD, Olympics2024 Medals: 113
Country: 中华人民共和国, GDP2023: 1.808481E7 Million USD, Olympics2024 Medals: 93
Country: 德国, GDP2023: 5072707.0 Million USD, Olympics2024 Medals: 48
Country: 日本, GDP2023: 4231143.0 Million USD, Olympics2024 Medals: 58
Country: 印度, GDP2023: 3730314.0 Million USD, Olympics2024 Medals: 28
Country: 法国, GDP2023: 3171209.0 Million USD, Olympics2024 Medals: 40
Country: 英国, GDP2023: 3070677.0 Million USD, Olympics2024 Medals: 51
Country: 意大利, GDP2023: 2610766.0 Million USD, Olympics2024 Medals: 38
Country: 加拿大, GDP2023: 2184804.0 Million USD, Olympics2024 Medals: 32
Country: 韩国, GDP2023: 1968019.0 Million USD, Olympics2024 Medals: 24
Country: 俄罗斯, GDP2023: 1799000.0 Million USD, Olympics2024 Medals: 50
Country: 以色列, GDP2023: 554173.0 Million USD, Olympics2024 Medals: 9

排序方式2: 按Olympics2024奖牌数量从大到小排序输出:
Country: 美国, GDP2023: 2.670417E7 Million USD, Olympics2024 Medals: 113
Country: 中华人民共和国, GDP2023: 1.808481E7 Million USD, Olympics2024 Medals: 93
Country: 日本, GDP2023: 4231143.0 Million USD, Olympics2024 Medals: 58
Country: 英国, GDP2023: 3070677.0 Million USD, Olympics2024 Medals: 51
Country: 俄罗斯, GDP2023: 1799000.0 Million USD, Olympics2024 Medals: 50
Country: 德国, GDP2023: 5072707.0 Million USD, Olympics2024 Medals: 48
Country: 法国, GDP2023: 3171209.0 Million USD, Olympics2024 Medals: 40
Country: 意大利, GDP2023: 2610766.0 Million USD, Olympics2024 Medals: 38
Country: 加拿大, GDP2023: 2184804.0 Million USD, Olympics2024 Medals: 32
Country: 印度, GDP2023: 3730314.0 Million USD, Olympics2024 Medals: 28
Country: 韩国, GDP2023: 1968019.0 Million USD, Olympics2024 Medals: 24
Country: 以色列, GDP2023: 554173.0 Million USD, Olympics2024 Medals: 9

排序方式3: 按GDP2023从大到小排序输出 (使用Comparator):
Country: 美国, GDP2023: 2.670417E7 Million USD, Olympics2024 Medals: 113
Country: 中华人民共和国, GDP2023: 1.808481E7 Million USD, Olympics2024 Medals: 93
Country: 德国, GDP2023: 5072707.0 Million USD, Olympics2024 Medals: 48
Country: 日本, GDP2023: 4231143.0 Million USD, Olympics2024 Medals: 58
```

```
排序方式3: 按GDP2023从大到小排序输出 (使用Comparator):
Country: 美国, GDP2023: 2.670417E7 Million USD, Olympics2024 Medals: 113
Country: 中华人民共和国, GDP2023: 1.808481E7 Million USD, Olympics2024 Medals: 93
Country: 德国, GDP2023: 5072707.0 Million USD, Olympics2024 Medals: 48
Country: 日本, GDP2023: 4231143.0 Million USD, Olympics2024 Medals: 58
Country: 印度, GDP2023: 3730314.0 Million USD, Olympics2024 Medals: 28
Country: 法国, GDP2023: 3171209.0 Million USD, Olympics2024 Medals: 40
Country: 英国, GDP2023: 3070677.0 Million USD, Olympics2024 Medals: 51
Country: 意大利, GDP2023: 2610766.0 Million USD, Olympics2024 Medals: 38
Country: 加拿大, GDP2023: 2184804.0 Million USD, Olympics2024 Medals: 32
Country: 韩国, GDP2023: 1968019.0 Million USD, Olympics2024 Medals: 24
Country: 俄罗斯, GDP2023: 1799000.0 Million USD, Olympics2024 Medals: 50
Country: 以色列, GDP2023: 554173.0 Million USD, Olympics2024 Medals: 9
```

Part 2 (25 分)

(2.1) 将第 8 章讲义 (JavaPD-Ch08) 中的 5 个应用程序 (Example8_1, Example8_2, Example8_3, Example8_4, Example8_6) 在 Eclipse 中运行, 如运行结果不唯一, 则需要运行多次并至少得到两个不同的结果。对重要语句加上注释。在报告中附上程序截图、运行结果截图和简要文字说明 (对运行结果做出解释)。(5 分)

Example8_1:

```
// 定义一个继承自 Thread 类的类
class WriteWordThread extends Thread {
    WriteWordThread(String s) {
```

```

        setName(s); // 设置线程的名字
    }

    public void run() {
        for (int i = 1; i <= 3; i++) {
            System.out.println("Thread: " + getName()); // 打印当前线程的名称
        }
    }
}

public class Example8_1 {
    public static void main(String args[]) {
        WriteWordThread zhang, wang;

        zhang = new WriteWordThread("Zhang"); // 创建线程 Zhang
        wang = new WriteWordThread("Wang"); // 创建线程 Wang

        zhang.start(); // 启动线程 Zhang
        for (int i = 1; i <= 3; i++) {
            System.out.println("Main Thread"); // 主线程打印信息
        }
        wang.start(); // 启动线程 Wang
    }
}

```

代码解释：

WriteWordThread 类：

该类继承了 Java 的 Thread 类，用于表示一个线程。

在构造方法中，通过 setName(s)设置线程的名称。setName()是 Thread 类提供的方法，可以为线程设置一个名称（在本程序中，线程会被命名为"Zhang"和"Wang"）。

run()方法是线程的执行体，重写 run()方法后，定义线程的具体工作。在此，run()方法中有一个循环，循环输出当前线程的名字（通过 getName()方法获取）。

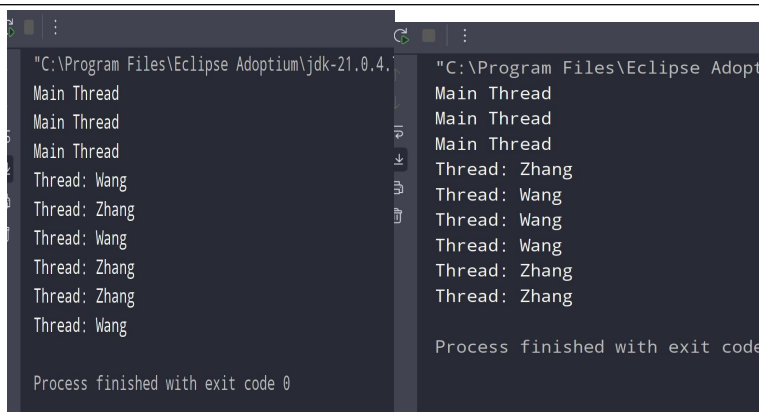
main()方法是程序的入口点。

在 main()方法中，我们首先创建了两个线程对象 zhang 和 wang，分别赋予它们名字 "Zhang"和"Wang"。

通过 zhang.start()启动 zhang 线程，start()方法会使线程开始执行，线程会执行 run()方法中的内容。

紧接着，主线程执行一个循环，输出 3 次"Main Thread"。

最后，调用 wang.start()启动 wang 线程，使得 wang 线程开始执行其 run()方法中的内容。



```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.4.
Main Thread
Main Thread
Main Thread
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Zhang
Thread: Zhang
Thread: Wang
Process finished with exit code 0

"C:\Program Files\Eclipse Adopt
Main Thread
Main Thread
Main Thread
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Zhang
Thread: Zhang
Process finished with exit code 0
```

不同的输出结果是由于 线程调度的不可预测性。具体来说，以下几个因素决定了每次程序执行时的输出顺序：

线程启动顺序：

`zhang.start()` 和 `wang.start()` 的调用顺序是确定的，但是它们的执行顺序由操作系统的调度器决定。

线程在启动后，会被操作系统分配 CPU 时间片来执行任务。由于操作系统的调度机制是不确定的，可能会出现 `zhang` 先执行，也可能是 `wang` 先执行，甚至可能是两者交替执行。

主线程和子线程的交替执行：

主线程在调用 `zhang.start()` 后，会继续执行并打印 "Main Thread"。此时，主线程和子线程之间的执行顺序取决于操作系统的调度。主线程可能会先打印完 3 次 "Main Thread"，也可能与子线程交替执行。

线程的执行时间：

`Thread` 的 `run()` 方法内没有显式的 `sleep()` 或其他阻塞操作，所以两个线程会尽可能地执行完它们的循环。每次 `System.out.println()` 输出后，线程会被操作系统挂起，等待下一次调度。由于 `zhang` 和 `wang` 的执行时间和操作系统调度的不确定性，输出顺序会有所不同。

Example8_2:

```
// 继承 Thread 类创建一个自定义线程类 class WriteWordThread extends Thread {
    int n = 0; // 定义一个整数 n，用来设置线程休眠的时间（单位：毫秒）

    // 构造方法，接受线程名称和休眠时间 n
    WriteWordThread(String s, int n) {
        setName(s); // 设置线程的名称
        this.n = n; // 设置休眠时间
    }
}
```

```

// 重写 run 方法，定义线程的执行任务
public void run() {
    for (int i = 1; i <= 3; i++) {
        System.out.println("Thread: " + getName()); // 打印当前线程的名称
        try {
            sleep(n); // 让线程休眠 n 毫秒，模拟耗时操作
        } catch (InterruptedException e) {
            // 如果线程被中断，捕获异常
        }
    }
}

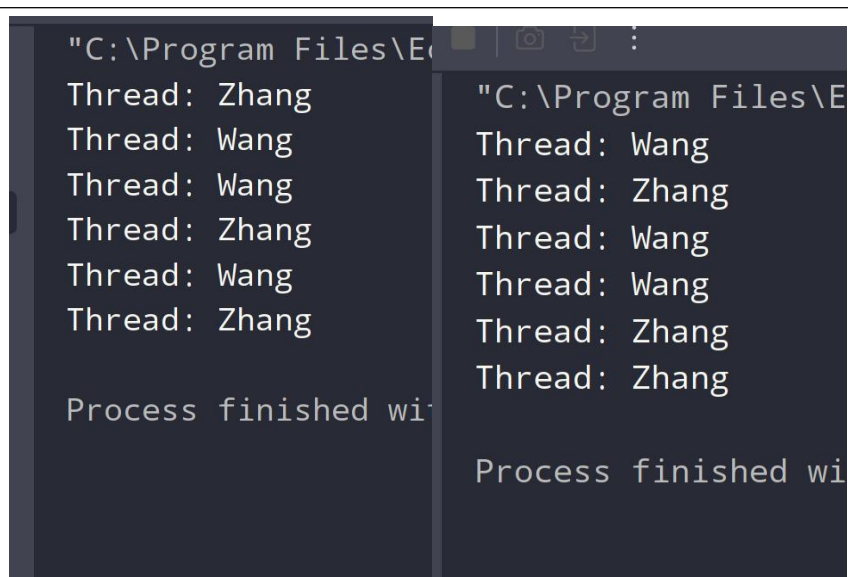
}

public class Example8_2 {
    public static void main(String args[]) {
        // 创建两个线程对象，并指定线程名称和休眠时间
        WriteWordThread zhang, wang;
        zhang = new WriteWordThread("Zhang", 200); // 线程 Zhang 休眠 200 毫秒
        wang = new WriteWordThread("Wang", 100); // 线程 Wang 休眠 100 毫秒

        // 启动两个线程
        zhang.start();
        wang.start();
    }
}

```

- WriteWordThread 类：
 - 该类继承了 Thread 类，并重写了 run() 方法。在 run() 方法中，通过 sleep(n) 来让线程“休眠”一定的时间，从而控制线程执行的间隔。
n 是一个整数，表示线程休眠的时间（单位是毫秒）。
通过 setName(s) 来设置线程的名称，getName() 可以获取线程的名称并在输出中显示。
 - main() 方法：
 - zhang 线程的休眠时间为 200 毫秒，wang 线程的休眠时间为 100 毫秒。
两个线程 zhang 和 wang 被创建并启动。
由于每个线程在打印输出后都会休眠一定的时间，因此线程的输出顺序受到 sleep(n) 的影响。
 - sleep(n) 方法：
 - Thread.sleep(n) 使得当前线程暂停执行指定的毫秒数。调用这个方法后，线程进入休眠状态，直到休眠时间结束。
- 注意：如果线程在休眠期间被中断，会抛出 InterruptedException 异常，但这里的代码并未做特别处理，仅仅捕获异常。



```
"C:\Program Files\E
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Zhang

Process finished wi

"C:\Program Files\E
Thread: Wang
Thread: Zhang
Thread: Wang
Thread: Wang
Thread: Zhang
Thread: Zhang

Process finished wi
```

使用了两个线程（WriteWordThread），并给每个线程设置了不同的睡眠时间（200 毫秒和 100 毫秒）。这种设置会导致线程在运行时交替执行，从而产生不同的执行顺序和输出结果。由于线程调度是由操作系统和 JVM 控制的，每次程序的输出可能会不同。

线程的输出顺序是由操作系统的线程调度器决定的，这取决于：

1. **线程的优先级：**在默认情况下，Java 中的线程优先级是平等的，操作系统根据资源的可用性来决定哪个线程优先执行。在多核处理器上，操作系统可能会根据 CPU 的空闲时间给线程分配不同的执行顺序。
2. **线程的睡眠时间：**Zhang 和 Wang 线程的 `sleep()` 时间不同，这意味着在 `sleep` 时间较短的线程（例如 Wang）在每次输出后会更快恢复执行，从而获得更多的 CPU 时间片。

Example8_3:

```
class Left extends Thread {
    int n = 0; // 计数器

    // 重写run方法，线程执行的任务
    public void run() {
        while(true) { // 无限循环
            n++; // 每次循环计数器加1
            System.out.println(n + " Left"); // 输出计数值和"Left"
            try {
                // 让线程休眠一个随机时间，单位是毫秒
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
                // 如果线程被中断，捕获并处理异常
            }
        }
    }
}
```

```

    }
}
}
}class Right extends Thread {
    int n = 0; // 计数器

    // 重写run 方法，线程执行的任务
    public void run() {
        while(true) { // 无限循环
            n++; // 每次循环计数器加1
            System.out.println(n + " Right"); // 输出计数值和"Right"
            try {
                // 让线程休眠一个随机时间，单位是毫秒
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {
                // 如果线程被中断，捕获并处理异常
            }
        }
    }
}

}
}public class Example8_3 {
    public static void main(String args[]) {
        Left left = new Left(); // 创建左边线程
        Right right = new Right(); // 创建右边线程

        left.start(); // 启动左边线程
        right.start(); // 启动右边线程

        while(true) { // 无限循环，直到满足终止条件
            try {
                Thread.sleep(100); // 主线程每100 毫秒休眠一次，允许子线程继续运行
            } catch (InterruptedException e) {
                e.printStackTrace(); // 捕获并打印异常
            }

            // 如果任意一个线程的计数器n 达到或超过8，则退出程序
            if(left.n >= 8 || right.n >= 8) {
                System.out.println(left.n + "," + right.n); // 输出当前两个线程的计数值
                System.exit(0); // 退出程序
            }
        }
    }
}
}
}

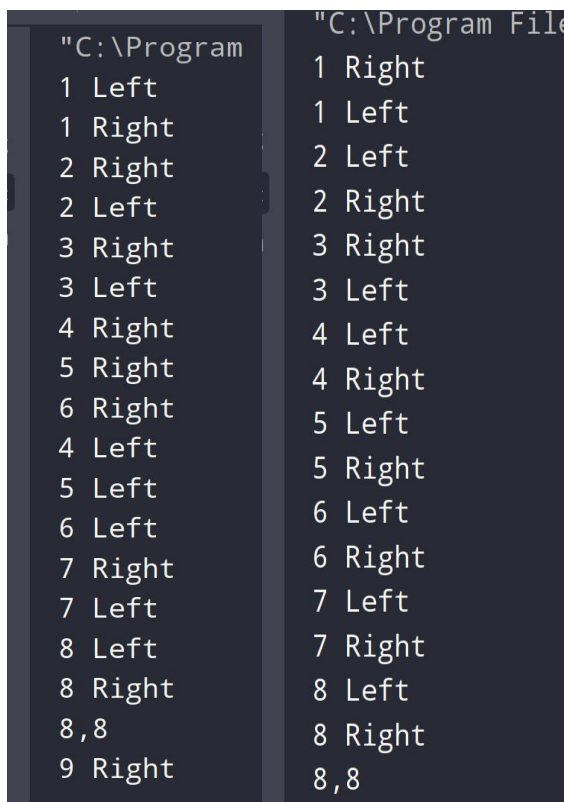
```


程序流程：

1. 主线程启动 Left 和 Right 两个子线程。
2. Left 和 Right 线程开始分别增加各自的计数器 n，并输出 "n Left" 或 "n Right"。
3. 主线程每隔 100 毫秒检查一次两个线程的计数器值。
4. 当任意一个线程的 n 达到 8 时，主线程输出计数值并调用 `System.exit(0)` 终止程序。

线程调度：

- 由于线程是并发执行的，具体的执行顺序无法预测，因此输出的顺序是非确定性的。
- 主线程会不断检查两个子线程的计数器，一旦满足条件就会输出结果并退出。



```
"C:\Program  "C:\Program Files
1 Left       1 Right
1 Right      1 Left
2 Right      2 Left
2 Left       2 Right
3 Right      3 Right
3 Left       3 Left
4 Right      4 Left
5 Right      4 Right
6 Right      5 Left
4 Left       5 Right
5 Left       6 Left
6 Left       6 Right
7 Right      7 Left
7 Left       7 Right
8 Left       8 Left
8 Right      8 Right
8,8          8,8
9 Right      8,8
```

Left 和 Right 两个线程的计数器 n 达到 8 的顺序，因为两个线程是并发执行的，因此它们的执行顺序和输出顺序是不确定的。

Example8_4:

```
class TaskBank implements Runnable {
    private int money = 0;
    String name1, name2;
```

```

// 构造函数，用于设置两个线程的名称
TaskBank(String s1, String s2) {
    name1 = s1;
    name2 = s2;
}

// 设置初始金额
public void setMoney(int amount) {
    money = amount;
}

// 线程执行的任务
public void run() {
    while (true) {
        money = money - 10; // 每次扣除 10 单位
        if (Thread.currentThread().getName().equals(name1)) {
            // 如果是“会计”线程
            System.out.println(name1 + ": " + money);
            if (money <= 100) {
                System.out.println(name1 + ": Finished");
                return; // 会计完成任务后结束线程
            }
        } else if (Thread.currentThread().getName().equals(name2)) {
            // 如果是“出纳”线程
            System.out.println(name2 + ": " + money);
            if (money <= 60) {
                System.out.println(name2 + ": Finished");
                return; // 出纳完成任务后结束线程
            }
        }
        try {
            Thread.sleep(800); // 每次操作后线程休眠 800 毫秒
        } catch (InterruptedException e) {}
    }
}

public class Example8_4 {
    public static void main(String args[]) {
        String s1 = "treasurer zhang"; // 会计线程名称
        String s2 = "cashier cheng"; // 出纳线程名称

        TaskBank taskBank = new TaskBank(s1, s2);
        taskBank.setMoney(120); // 设置初始余额为 120
    }
}

```



```

        // 创建并启动两个线程
        Thread zhang;
        Thread cheng;

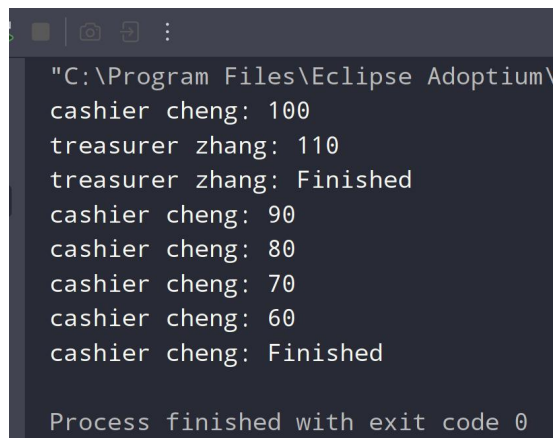
        zhang = new Thread(taskBank); // 创建会计线程
        cheng = new Thread(taskBank); // 创建出纳线程

        zhang.setName(s1); // 设置线程名称
        cheng.setName(s2); // 设置线程名称

        zhang.start(); // 启动会计线程
        cheng.start(); // 启动出纳线程
    }
}

```

输出结果通常会是一样的



```

"C:\Program Files\Eclipse Adoptium\
cashier cheng: 100
treasurer zhang: 110
treasurer zhang: Finished
cashier cheng: 90
cashier cheng: 80
cashier cheng: 70
cashier cheng: 60
cashier cheng: Finished

Process finished with exit code 0

```

Example8_5:

```

class Task implements Runnable {
    private int number = 0;

    // 设置数字初值
    public void setNumber(int n) {
        number = n;
    }

    // 线程执行的方法
    public void run() {
        while(true) {
            if (Thread.currentThread().getName().equals("add")) {
                // 如果是加法线程
                number++;
                System.out.printf("%d\n", number); // 输出当前值
            }
        }
    }
}

```

```

        if (Thread.currentThread().getName().equals("sub")) {
            // 如果是减法线程
            number--;
            System.out.printf("%12d\n", number); // 输出当前值
        }
        try {
            Thread.sleep(1000); // 休眠1秒
        } catch (InterruptedException e) {}
    }
}

public class Example8_5 {
    public static void main(String args[]) {
        Task taskAdd = new Task();
        taskAdd.setNumber(10); // 设置加法线程初值为10
        Task taskSub = new Task();
        taskSub.setNumber(-10); // 设置减法线程初值为-10

        // 创建线程
        Thread threadA, threadB, threadC, threadD;

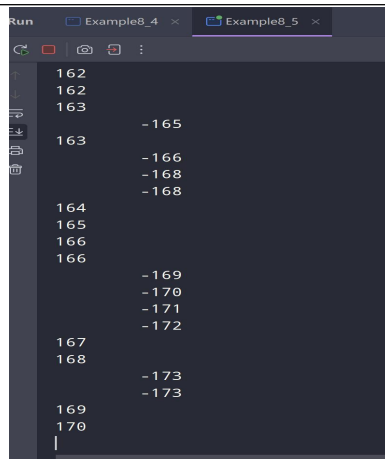
        threadA = new Thread(taskAdd);
        threadB = new Thread(taskAdd);
        threadA.setName("add");
        threadB.setName("add");

        threadC = new Thread(taskSub);
        threadD = new Thread(taskSub);
        threadC.setName("sub");
        threadD.setName("sub");

        // 启动线程
        threadA.start();
        threadB.start();
        threadC.start();
        threadD.start();
    }
}

```

输出的顺序取决于线程调度：例如，add 和 sub 线程的输出可能交替，可能先是 add 输出，再是 sub 输出，或者反过来。



(2.2). 运行以下三个程序（每个程序运行 5 次），并对输出结果给出分析。在报告中附上程序截图和简要的文字说明（包括对结果的分析）。（10 分）

程序 1:

```
// The task for printing a character a specified number of times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The number of times to repeat

    /** Construct a task with a specified character and number of
     * times to print the character
     */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    @Override /** Override the run() method to tell the system
     * what task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;

    /** Construct a task for printing 1, 2, ..., n */
    public PrintNum(int n) {
        lastNum = n;
    }

    @Override /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}
```

```
C:\Program Files\Eclipse Adoptium\jdk-21.0.4-hotspot\bin\java.exe "-javaagent:C:\Users\caozh\IntelliJ IDEA Community Edition 2024.1.4\lib\idea_rt.jar:aabbbbbbbaaaaaaaaaabbbaaaaaaaaaaaabbbbaaaaaaaaaaaaaaaaabbbbaaaaaaaabbbbbbbaaaaaaaaaabaabbbbbbaaaaaaaaaabbbbbbbaaaaaaaabbbbbb Process finished with exit code 0
```

```
JTF-8 --classpath C:\Users\caozh\Desktop\java程序设计\lab3\out\production\lab3 TaskThreadDemo  
35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85
```

- **并发执行：**三个线程并发运行，因此它们的输出会混合在一起。具体的输出顺序取决于线程调度，具有不确定性。
- **线程调度器：**操作系统的线程调度器决定了哪个线程先执行、哪个线程后执行，因此输出可能是字符 'a'、字符 'b' 和数字 1 到 100 之间的任意交替组合。
- **输出的结果：**每次运行程序时，输出结果可能不同。你可能看到连续的一段 'a'，然后是一段 'b'，或者字符和数字交替输出。

程序 2:

```
// The task for printing a character a specified number of times
class PrintChar implements Runnable {
    private char charToPrint; // The character to print
    private int times; // The number of times to repeat

    /** Construct a task with a specified character and number of
     * times to print the character
     */
    public PrintChar(char c, int t) {
        charToPrint = c;
        times = t;
    }

    @Override /** Override the run() method to tell the system
     * what task to perform
     */
    public void run() {
        for (int i = 0; i < times; i++) {
            System.out.print(charToPrint);
        }
    }
}

// The task class for printing numbers from 1 to n for a given n
class PrintNum implements Runnable {
    private int lastNum;

    /** Construct a task for printing 1, 2, ..., n */
    public PrintNum(int n) {
        lastNum = n;
    }

    @Override /** Tell the thread how to run */
    public void run() {
        for (int i = 1; i <= lastNum; i++) {
            System.out.print(" " + i);
        }
    }
}
```

```
"C:\Program Files\Eclipse Adoptium\jdk-21.0.4.7-hotspot\bin\java.exe" "-javaagent:C:\Users\caozh\IntelliJ IDEA Community Edition 2024.1.4\lib\idea_rt.jar=
aaaaaaaaaaaaabbbbbbbbaaaaaaaaaaaaabbbbbbbbaaaaaaaaaaaaabbbbbbbbaaaaaaaabbbbbbbbbaaaaaaaabbbbbbbbaaaaaaaabbbbbbbbaaaaaaaabbbbbbbbaaaaaaaab
Process finished with exit code 0
```

```

C:\Users\caozh\Desktop>java -Dfile.encoding=UTF-8 -Dsun.stdout.encoding=UTF-8 -Dsun.stderr.encoding=UTF-8 -classpath C:\Users\caozh\Desktop\java程序设计\lab3\
bbby 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53

```

任务提交顺序:

任务是按顺序提交到线程池中的，打印字符的任务先于打印数字的任务进入线程池，因此字符任务优先执行。

线程池调度机制:

线程池的调度机制和任务的队列特性使得字符打印任务总是先执行完毕，数字打印任务在字符任务结束后才开始。

打印顺序的确定性:

字符输出顺序可能交替或连续，具体取决于调度器的行为。

数字的输出顺序是确定的，从 1 到 100，因为 `PrintNum` 的任务是一个单独线程完成的顺序任务，且不会与其他任务交错。

因此，程序的输出永远是先字符后数字，且数字的顺序是从 1 到 100。这种现象是由线程池任务的提交顺序以及线程调度的行为共同决定的。

程序 3:

```
import java.util.concurrent.*;

public class AccountWithoutSync {
    private static Account account = new Account();

    public static void main(String[] args) {
        ExecutorService executor = Executors.newCachedThreadPool();

        // Create and launch 100 threads
        for (int i = 0; i < 100; i++) {
            executor.execute(new AddAPennyTask());
        }

        executor.shutdown();

        // Wait until all tasks are finished
        while (!executor.isTerminated()) {
        }

        System.out.println("What is balance? " + account.getBalance());
    }

    // A thread for adding a penny to the account
    private static class AddAPennyTask implements Runnable {
        public void run() {
            account.deposit(1);
        }
    }

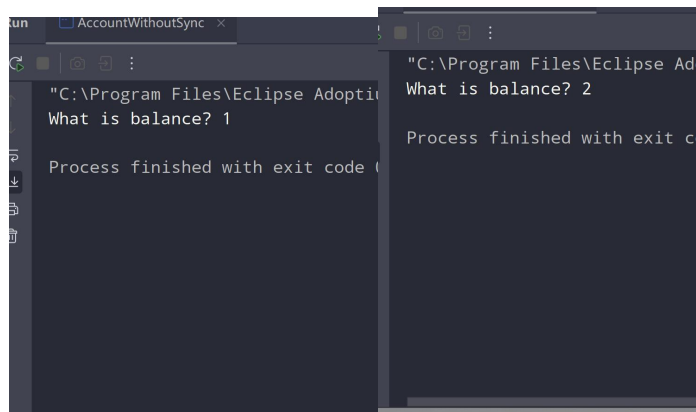
    // An inner class for account
    private static class Account {
        private int balance = 0;

        public int getBalance() {
            return balance;
        }

        public void deposit(int amount) {
            int newBalance = balance + amount;

            // This delay is deliberately added to magnify the
            // data-corruption problem and make it easy to see.
            try {
                Thread.sleep(5);
            }
            catch (InterruptedException ex) {
            }

            balance = newBalance;
        }
    }
}
```



总结:

- **原因:** 输出结果总是 1 或 2, 是由于多个线程在没有同步控制的情况下, 几乎同时读取并更新共享变量 `balance`, 导致数据竞争。
- **延迟的影响:** `Thread.sleep(5)` 放大了这种数据竞争, 使得多个线程在更新余额时覆盖彼此的更改。
- **解决方案:** 使用同步机制来确保每次只有一个线程能够更新余额, 可以避免数据竞争, 确保最终余额的正确性。

(2.3) 第 8 章讲义 (JavaPD-Ch08) 中的第 5 个应用程序 (Example8_5) 存在线程间不同步的问题, 请修改该程序, 以解决不同步的问题。在报告中附上程序截图、运行结果截图和详细的文字说明 (包括设计的思路和合理性分析)。(10 分)

```
class Task_fixed implements Runnable {  
    private int number = 0;  
  
    // 设置数字初值  
    public void setNumber(int n) {  
        number = n;  
    }  
  
    // 线程执行的方法  
    public void run() {  
        while (true) {  
            if (Thread.currentThread().getName().equals("add")) {  
                synchronized (this) {  
                    // 如果是加法线程  
                    number++;  
                    System.out.printf("%d\n", number); // 输出当前值  
                }  
            } else if (Thread.currentThread().getName().equals("sub")) {  
                synchronized (this) {  
                    // 如果是减法线程  
                    number--;  
                    System.out.printf("%12d\n", number); // 输出当前值  
                }  
            }  
            try {  
                Thread.sleep(1000); // 休眠1秒  
            } catch (InterruptedException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



```

    }
}
}
public class Example8_5_fixed {
    public static void main(String[] args) {
        Task taskAdd = new Task();
        taskAdd.setNumber(10); // 设置加法线程初值为10

        Task taskSub = new Task();
        taskSub.setNumber(-10); // 设置减法线程初值为-10

        // 创建线程
        Thread threadA, threadB, threadC, threadD;

        threadA = new Thread(taskAdd);
        threadB = new Thread(taskAdd);
        threadA.setName("add");
        threadB.setName("add");

        threadC = new Thread(taskSub);
        threadD = new Thread(taskSub);
        threadC.setName("sub");
        threadD.setName("sub");

        // 启动线程
        threadA.start();
        threadB.start();
        threadC.start();
        threadD.start();
    }
}

```

同步块的使用：

在原来的代码中，对共享变量 `number` 的读写操作没有同步控制，多个线程可能同时访问 `number`，导致结果不一致。

使用 `synchronized (this)` 来确保每次只有一个线程能够进入同步块，从而对 `number` 进行修改。

这样可以保证对 `number` 的更新操作是原子操作，不会被其他线程打断。

修改的位置：

在加法线程和减法线程各自的逻辑中使用了同步块来保护 `number` 的操作。

使用 `synchronized` 保证线程在对 `number` 执行读取、修改、写回的操作时，不会被

其他线程同时进行干扰。

线程命名：

线程 `threadA` 和 `threadB` 的名称均为 "add"，它们共享同一个 `taskAdd` 对象。

线程 `threadC` 和 `threadD` 的名称均为 "sub"，它们共享同一个 `taskSub` 对象。

由于加法和减法任务分别属于不同对象，使用 `synchronized (this)` 会保证对每个任务对象的访问是独占的，因此加法和减法操作可以并发进行，但同一个任务对象中的操作不会并发。

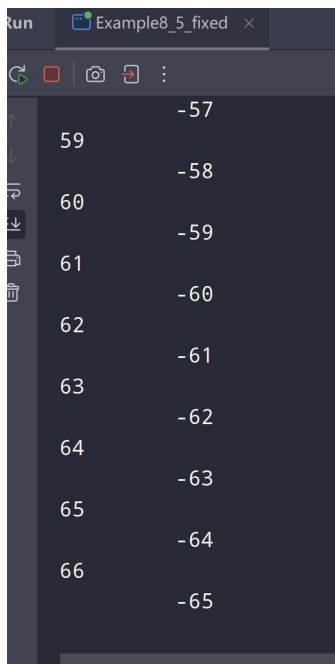
修改后的运行结果分析：

在修改之后，每个加法线程和减法线程在更新 `number` 时，都通过同步机制保证了线程安全。

结果输出会有以下特点：

加法线程和减法线程分别操作自己对应的 `number`，输出的数值变化是按顺序递增或递减的。

`synchronized` 确保了每次对 `number` 的操作都是完整的，不会因多个线程的并发而导致操作丢失或数据不一致。



Part 3 (30 分)

(3.1). 编写 Java 应用程序实现如下功能：第一个线程不停地随机生成 $[0,1)$ 之间的浮点数（float）并输出到屏幕，第二个线程将第一个线程输出的第 1-5 个浮点数的和与平均值输出到屏幕（紧跟在第一个线程输出的第 5 个浮点数之后）、将第一个线程输出的第 6-10 个 $[0,1)$ 之间的浮点数的和与平均值输出到屏幕（紧跟在第一个线程输出的第 10 个浮点数之后）...。要求线程间实现通信。要求采用实现 `Runnable` 接口和 `Thread` 类的构造方法的方式创建线程，而不是通过 `Thread` 类的子类的方式。在报告中附上程序截图、运行结果截图和详细的文字说明（包括设计的思路 and 合理性分析）。（10 分）

```
Generated: 0.5301
Sum: 3.2525, Average: 0.6505
Generated: 0.6777
Generated: 0.5416
Generated: 0.6684
Generated: 0.3412
Generated: 0.9553
Sum: 3.1842, Average: 0.6368
Generated: 0.7833
Generated: 0.3977
Generated: 0.3934
Generated: 0.1106
Generated: 0.5527
Sum: 2.2377, Average: 0.4475
Generated: 0.0030
Generated: 0.0776
Generated: 0.4311
Generated: 0.8515
Generated: 0.9323
Sum: 2.2956, Average: 0.4591
Generated: 0.4954
```

```
import java.util.ArrayList;import java.util.List;
public class RandomNumberSumCalculator {
    public static void main(String[] args) {
        SharedData sharedData = new SharedData();

        // 创建生成随机数的线程
        Thread producerThread = new Thread(new RandomNumberProducer(sharedData));
        producerThread.setName("Producer");

        // 创建计算和与平均值的线程
        Thread consumerThread = new Thread(new RandomNumberConsumer(sharedData));
        consumerThread.setName("Consumer");

        // 启动线程
        producerThread.start();
        consumerThread.start();
    }
}

class SharedData {
    private final List<Float> numbers = new ArrayList<>();
    private boolean isReady = false;

    public synchronized void addNumber(float number) {
        numbers.add(number);
        System.out.printf("Generated: %.4f\n", number);

        if (numbers.size() % 5 == 0) {
            isReady = true;
        }
    }
}
```

```

        notifyAll(); // 通知消费者线程进行计算
    }
}

public synchronized List<Float> getNumbers() {
    while (!isReady) {
        try {
            wait(); // 等待生成足够的数据
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Consumer thread interrupted");
        }
    }

    // 复制前 5 个元素用于计算，并从原始列表中移除
    List<Float> subList = new ArrayList<>(numbers.subList(0, 5));
    numbers.subList(0, 5).clear();
    isReady = false;
    notifyAll(); // 通知生产者线程继续生成数据
    return subList;
}
}

class RandomNumberProducer implements Runnable {
    private final SharedData sharedData;

    public RandomNumberProducer(SharedData sharedData) {
        this.sharedData = sharedData;
    }

    @Override
    public void run() {
        while (true) {
            float randomNum = (float) Math.random(); // 生成 [0, 1) 的随机浮点数
            synchronized (sharedData) {
                sharedData.addNumber(randomNum);
            }
            try {
                Thread.sleep(200); // 模拟一些延迟
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("Producer thread interrupted");
            }
        }
    }
}

```

```

}
class RandomNumberConsumer implements Runnable {
    private final SharedData sharedData;

    public RandomNumberConsumer(SharedData sharedData) {
        this.sharedData = sharedData;
    }

    @Override
    public void run() {
        while (true) {
            List<Float> numbers = sharedData.getNumbers(); // 获取生成的 5 个随机数
            float sum = 0;
            for (float num : numbers) {
                sum += num;
            }
            float average = sum / numbers.size();
            System.out.printf("Sum: %.4f, Average: %.4f\n", sum, average);
        }
    }
}

```

线程通信机制：

使用 wait() 和 notifyAll()：

wait()：当消费者线程调用 getNumbers() 方法时，如果当前没有足够的浮点数（即少于 5 个），则消费者线程会等待。

notifyAll()：当生产者线程添加了足够的浮点数（即 5 个）时，它会调用 notifyAll()，通知消费者线程可以开始计算了。

同样，消费者线程在完成计算后会调用 notifyAll()，通知生产者线程继续生成新的浮点数。

互斥访问：

为了确保线程之间互不干扰，addNumber() 和 getNumbers() 都使用了同步块（synchronized），这样可以保证对共享资源的访问是线程安全的。

设计合理性分析：

线程的独立性和协同工作：

生产者线程和消费者线程各自独立，通过共享的数据结构进行通信。这种设计使得每个线程专注于自己特定的任务，符合单一职责原则。

使用 wait() 和 notifyAll() 实现线程间的协同工作，使得消费者线程能及时地在 5 个数生成后进行计算，保证了输出的及时性和一致性。

线程安全性：

使用 `synchronized` 保证对共享数据的互斥访问，避免了多线程对共享数据同时操作导致的 inconsistency。

`wait()` 和 `notifyAll()` 机制使得线程之间能够正确协调，避免出现死锁或者空等待的情况。

合理的任务划分：

一个线程负责生成数据，另一个线程负责计算，逻辑清晰，任务明确。

这种设计使得程序更具扩展性，如果需要进一步增加复杂的计算或改变生成数据的方式，只需修改对应的线程类即可。

(3.2). 编写 Java 应用程序实现如下功能：创建工作线程，模拟银行现金账户取款和存款操作。多个线程同时执行取款和存款操作时，如果不使用同步处理，会造成账户余额混乱，要求使用 `synchronized` 关键字同步代码块，以保证多个线程同时执行取款和存款操作时，银行现金账户取款和存款的有效和一致。要求采用实现 `Runnable` 接口和 `Thread` 类的构造方法的方式创建线程，而不是通过 `Thread` 类的子类的方式。在报告中附上程序截图（假设银行存款有 100 元，有 3 个取款线程和 2 个存款线程，每次取款和存款均为 10 元）、运行结果截图（显示每次存取款操作后的余额等信息，以说明线程间同步正确）和详细的文字说明。（10 分）

```
import java.util.ArrayList;import java.util.List;
public class RandomNumberSumCalculator {
    public static void main(String[] args) {
        SharedData sharedData = new SharedData();

        // 创建生成随机数的线程
        Thread producerThread = new Thread(new RandomNumberProducer(sharedData));
        producerThread.setName("Producer");

        // 创建计算和与平均值的线程
        Thread consumerThread = new Thread(new RandomNumberConsumer(sharedData));
        consumerThread.setName("Consumer");

        // 启动线程
        producerThread.start();
        consumerThread.start();
    }
}

class SharedData {
    private final List<Float> numbers = new ArrayList<>();
    private boolean isReady = false;

    public synchronized void addNumber(float number) {
```

```

        numbers.add(number);
        System.out.printf("Generated: %.4f\n", number);

        if (numbers.size() % 5 == 0) {
            isReady = true;
            notifyAll(); // 通知消费者线程进行计算
        }
    }

    public synchronized List<Float> getNumbers() {
        while (!isReady) {
            try {
                wait(); // 等待生成足够的数据
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
                System.out.println("Consumer thread interrupted");
            }
        }

        // 复制前5个元素用于计算，并从原始列表中移除
        List<Float> subList = new ArrayList<>(numbers.subList(0, 5));
        numbers.subList(0, 5).clear();
        isReady = false;
        notifyAll(); // 通知生产者线程继续生成数据
        return subList;
    }
}

class RandomNumberProducer implements Runnable {
    private final SharedData sharedData;

    public RandomNumberProducer(SharedData sharedData) {
        this.sharedData = sharedData;
    }

    @Override
    public void run() {
        while (true) {
            float randomNum = (float) Math.random(); // 生成 [0, 1) 的随机浮点数
            synchronized (sharedData) {
                sharedData.addNumber(randomNum);
            }
            try {
                Thread.sleep(200); // 模拟一些延迟
            } catch (InterruptedException e) {

```

```

        Thread.currentThread().interrupt();
        System.out.println("Producer thread interrupted");
    }
}
}
}
}
class RandomNumberConsumer implements Runnable {
    private final SharedData sharedData;

    public RandomNumberConsumer(SharedData sharedData) {
        this.sharedData = sharedData;
    }

    @Override
    public void run() {
        while (true) {
            List<Float> numbers = sharedData.getNumbers(); // 获取生成的 5 个随机数
            float sum = 0;
            for (float num : numbers) {
                sum += num;
            }
            float average = sum / numbers.size();
            System.out.printf("Sum: %.4f, Average: %.4f\n", sum, average);
        }
    }
}
}

```



```
C:\Program Files\Eclipse Adoptium\jdk-
Withdraw-1 withdrew 10. Balance: 90
Deposit-2 deposited 10. Balance: 100
Deposit-1 deposited 10. Balance: 110
Withdraw-3 withdrew 10. Balance: 100
Withdraw-2 withdrew 10. Balance: 90
Withdraw-1 withdrew 10. Balance: 80
Deposit-2 deposited 10. Balance: 90
Withdraw-3 withdrew 10. Balance: 80
Deposit-1 deposited 10. Balance: 90
Withdraw-2 withdrew 10. Balance: 80
Withdraw-1 withdrew 10. Balance: 70
Withdraw-3 withdrew 10. Balance: 60
Deposit-1 deposited 10. Balance: 70
Deposit-2 deposited 10. Balance: 80
Withdraw-2 withdrew 10. Balance: 70
Deposit-2 deposited 10. Balance: 80
Deposit-1 deposited 10. Balance: 90
Withdraw-1 withdrew 10. Balance: 80
Withdraw-3 withdrew 10. Balance: 70
Withdraw-2 withdrew 10. Balance: 60
Deposit-2 deposited 10. Balance: 70
Deposit-1 deposited 10. Balance: 80
Withdraw-1 withdrew 10. Balance: 70
Withdraw-3 withdrew 10. Balance: 60
Withdraw-2 withdrew 10. Balance: 50
```

- **取款和存款交替输出：**从输出中可以看到，取款和存款的操作是交替进行的，取款线程和存款线程在系统调度器的控制下轮流执行。
- **余额变化有序且一致：**
 - 每次取款减少 10，每次存款增加 10，余额变化合理且符合预期，没有出现负数或跳跃式的增减。
 - 由于使用了 `synchronized` 同步代码块，线程之间是互斥访问共享资源的，因此余额的修改操作不会相互覆盖。

设计合理性分析

线程同步：

通过使用 `synchronized` 关键字确保对共享资源的互斥访问，使得每次存款和取款的修改操作都能完整地完，避免了因多个线程同时访问而引发的数据不一致性问题。

线程的独立性：

取款和存款操作分别封装在 `WithdrawTask` 和 `DepositTask` 中，符合单一职责原则，

每个类只负责一个操作。

线程的创建使用了 `Runnable` 接口，并通过 `Thread` 类进行启动，这种方式使得线程的逻辑与线程的控制解耦，增强了代码的可维护性和可扩展性。

任务的合理性：

每个线程进行 5 次操作，每次取款或存款的金额为 10 元，这样的设计可以直观地观察到每次存取款操作对余额的影响，从而验证线程间的同步是否正确。

模拟并发环境：

通过同时启动 3 个取款线程和 2 个存款线程，模拟了多线程对同一账户进行并发存取的情况，显示了使用同步机制的重要性。

(3.3). 有一座南北向的桥，只能容纳一个人，桥的南边有 1000 个人（记为 $S_1, S_2, \dots, S_{1000}$ ）和桥的北边有 1000 个人（记为 $N_1, N_2, \dots, N_{1000}$ ），编写 Java 应用程序让这些到达对岸，每个人用一个线程表示，桥为共享资源，在过桥的过程中输出谁正在过桥（不同人之间用逗号隔开）。运行 10 次，分别统计南边的 1000 人和北边的 1000 人先全部到达对岸的次数（第 i 行输出格式为：第 i 次运行，南边/北边先完成过桥）。要求采用实现 `Runnable` 接口和 `Thread` 类的构造方法的方式创建线程，而不是通过 `Thread` 类的子类的方式。在报告中附上程序截图、运行结果截图和详细的文字说明（包括对结果的分析）。（10 分）

```
import java.util.ArrayList;import java.util.List;import
java.util.concurrent.atomic.AtomicInteger;
public class BridgeCrossingSimulation {
    private static final int NUM_PEOPLE = 1000;
    private static final Object bridgeLock = new Object(); // 桥的同步锁
    private static AtomicInteger southCount = new AtomicInteger(0);
    private static AtomicInteger northCount = new AtomicInteger(0);

    public static void main(String[] args) {
        int southWins = 0;
        int northWins = 0;

        for (int run = 1; run <= 10; run++) {
            // 重置计数器
            southCount.set(0);
            northCount.set(0);

            // 创建南边和北边的人线程
            List<Thread> threads = new ArrayList<>();
            for (int i = 1; i <= NUM_PEOPLE; i++) {
                threads.add(new Thread(new Person("S" + i, "South")));
                threads.add(new Thread(new Person("N" + i, "North")));
            }
        }
    }
}
```

```

    }

    // 启动所有线程
    for (Thread thread : threads) {
        thread.start();
    }

    // 等待所有线程完成
    for (Thread thread : threads) {
        try {
            thread.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.out.println("Thread interrupted: " + e.getMessage());
        }
    }

    // 判断哪一侧先完成
    if (southCount.get() == NUM_PEOPLE) {
        southWins++;
        System.out.printf("第%d 次运行, 南边先完成过桥。\\n", run);
    } else if (northCount.get() == NUM_PEOPLE) {
        northWins++;
        System.out.printf("第%d 次运行, 北边先完成过桥。\\n", run);
    }
}

// 输出最终统计结果
System.out.println("南边先全部完成的次数: " + southWins);
System.out.println("北边先全部完成的次数: " + northWins);
}

static class Person implements Runnable {
    private final String name;
    private final String direction;

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

    @Override
    public void run() {
        try {

```

```

        // 每个人花费随机时间到达桥口
        Thread.sleep((int) (Math.random() * 100));
        // 过桥过程
        synchronized (bridgeLock) {
            System.out.println(name + " from " + direction + " is crossing the
bridge...");

            Thread.sleep(10); // 模拟过桥时间
        }

        // 统计过桥人数
        if (direction.equals("South")) {
            southCount.incrementAndGet();
        } else if (direction.equals("North")) {
            northCount.incrementAndGet();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
        System.out.println("Thread interrupted: " + e.getMessage());
    }
}
}
}

```

```

S119 from South is crossing the bridge...
N164 from North is crossing the bridge...
S142 from South is crossing the bridge...
S41 from South is crossing the bridge...
N60 from North is crossing the bridge...
N103 from North is crossing the bridge...
N7 from North is crossing the bridge...
S35 from South is crossing the bridge...
S94 from South is crossing the bridge...
S67 from South is crossing the bridge...
S66 from South is crossing the bridge...
N101 from North is crossing the bridge...
N135 from North is crossing the bridge...
N84 from North is crossing the bridge...
N83 from North is crossing the bridge...
S80 from South is crossing the bridge...
N27 from North is crossing the bridge...
N39 from North is crossing the bridge...
S120 from South is crossing the bridge...
N79 from North is crossing the bridge...
第10次运行，南边先完成过桥。
南边先全部完成的次数：10
北边先全部完成的次数：0

```

可能的原因分析

线程启动顺序的偏向性：

在主线程中，南边的 1000 个线程首先被创建和启动，然后是北边的 1000 个线程。

由于启动顺序，南边的线程可能更早被操作系统调度，从而在竞争桥资源时总是占据优势，导致南边的人更快完成过桥。

在 `for` 循环中创建和启动线程时，程序首先启动了南边的线程。操作系统调度器可能会优先调度已经创建的线程，这就让南边的人在调度上占据了先发优势。

线程竞争中的不公平性：

在 `Java` 的线程调度中，线程的竞争并不是严格公平的。由于南边的线程先启动，它们在争夺锁对象（桥的锁）时可能会占据一定的优势，使得南边的人每次都可以更早过桥。

线程调度的随机性对每个线程的优先级可能不均衡，特别是在大量线程同时争夺有限资源的情况下，先创建和启动的线程可能会更有机会获得锁。

锁的竞争和资源获取的先发优势：

过桥的共享资源使用 `synchronized` 关键字进行同步，意味着每次只有一个线程可以获取到锁并完成过桥操作。

南边的线程先启动并获得锁的概率更高，导致北边的线程始终落后。

由于南边线程的启动顺序在前，它们有更多机会首先获取桥的锁，这样的竞争优势使得南边的线程始终领先。

桥的同步机制对公平性的影响：

使用 `synchronized` 锁来管理桥的访问没有引入任何形式的公平性控制，这使得调度的随机性可能造成南边线程在锁竞争中持续占据上风。

如果多个线程同时等待锁，操作系统会根据具体的调度策略来决定哪个线程可以获得锁。在当前实现中，调度器可能偏向于南边线程，因为它们先启动。

报告写作。要求：主要思路有明确的说明，重点代码有详细的注释，行文逻辑清晰可读性强，报告整体写作较为专业。（20 分）

说明：

- （1）本次实验课作业满分为 100 分，占总成绩的比例 7%。
- （2）本次实验课作业截至时间 2024 年 11 月 20 日（周三）21:59。
- （3）报告正文：请在**指定位置填写**，本次实验**不需要单独提交源程序文件**。
- （4）个人信息：WORD 文件名中的“姓名”、“学号”，请改为你的**姓名和学号**；实验报告的首页，**请准确填写“学院”、“专业”、“报告人”、“学号”、“班级”、“实验报告提交时间”**等信息。
- （5）提交方式：截至时间前，请在 Blackboard 平台中提交。
- （6）发现抄袭（包括复制&粘贴整句话、整张图），**抄袭者和被抄袭者的成绩记零分**。
- （7）延迟提交，不得分；如有特殊情况，请于截至日期之后的**48 小时内**发邮件到 panweike@szu.edu.cn，并在邮件中注明课程名称、作业名称、姓名、学号等信息，以及特殊情况的说明，我收到后会及时回复。
- （8）期末考试阶段补交无效。

+++++

其他（例如感想、建议等等）。

<p>指导教师批阅意见：</p>	
<p>成绩评定：</p>	
<p>指导教师签字：</p>	<p>2024 年 月 日</p>

指导教师签字：_____

2024 年 月 日

2024 年 月 日

备注:	
-----	--

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。