深圳大学实验报告

课程名称：Java 程序设计

实验项目名称：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：毛斐巧

报告人：吴嘉楷 学号：2022150168 班级：国际班

实验时间：2024 年 10.29、11.5、12、19 日（周二）

实验报告提交时间：2024 年 10 月 29 日

教务部制

一、 实验目的

1. 掌握 Java 程序设计中的线程同步等技术。

2. 熟悉集合类的应用，掌握接口的定义、实现类的编写和接口回调等技术。

二、实验内容与要求

1. 运行以下三个程序（每个程序运行 10 次），并对输出结果给出分析。在报告中附上程序截图和详细的文字说明。（5 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);
    }
  }
}

public class TaskThreadDemo {
  public static void main(String[] args) {
    // Create tasks
    Runnable printA = new PrintChar('a', 100);
    Runnable printB = new PrintChar('b', 100);
    Runnable print100 = new PrintNum(100);

    // Create threads
    Thread thread1 = new Thread(printA);
    Thread thread2 = new Thread(printB);
    Thread thread3 = new Thread(print100);

    // Start threads
    thread1.start();
    thread2.start();
    thread3.start();
  }
}
```

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);
    }
  }
}
```

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

public class ExecutorDemo {
  public static void main(String[] args) {
    // Create a fixed thread pool with maximum three threads
    ExecutorService executor = Executors.newFixedThreadPool(3);

    // Submit runnable tasks to the executor
    executor.execute(new PrintChar('a', 100));
    executor.execute(new PrintChar('b', 100));
    executor.execute(new PrintNum(100));

    // Shut down the executor
    executor.shutdown();
  }
}
```

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;
    }
  }
}
```

2. 编写 Java 应用程序实现如下功能：第一个线程生成一个随机数，第二个线程每隔一段时间读取第一个线程生成的随机数，并判断它是否是奇数。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、完整的运行结果截图和简要文字说明。（10 分）

3. 编写 Java 应用程序实现如下功能：第一个线程输出数字 1-26，第二个线程输出字母 A-Z，输出的顺序为 1A2B3C...26Z，即每 1 个数字紧跟着 1 个字母的方式。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。（10 分）

4. 编写 Java 应用程序实现如下功能：创建工作线程，模拟银行现金账户存款操作。多个线程同时执行存款操作时，如果不使用同步处理，会造成账户余额混乱，要求使用 syncrhonized 关键字同步代码块，以保证多个线程同时执行存款操作时，银行现金账户存款的有效和一致。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、运行结果截图和详细的文字说明。（10 分）

5. 编写 Java 应用程序，根据用户输入的 5 个日期（每行一个日期），计算相邻两个日期之间间隔的时数，共 4 个结果（时数）。注：用户输入的时间格式为“××××年××月××日××时××分××秒”，输出的时间格式为“××日××时××分××秒”。在报告中应有程序截图、完整的运行结果截图和简要文字说明。（20 分）

6. 编写 Java 应用程序，实现稀疏矩阵的加法和乘法运算，其中稀疏矩阵是指矩阵中的大部分元素的值为 0。用户在命令行输入矩阵时矩阵的大小可能有错，因此需要使用异常处理。在报告中应有程序截图、完整的运行结果截图和简要文字说明。（15 分）

7. 编写 Java 应用程序，统计分析网页 https://en.szu.edu.cn/About/About2.htm 中关于深圳大学的介绍的英文文章（包括题目 About）中每个英文单词出现的次数（不区分大小写，不要写爬虫，可以把整篇文章的内容当作一个字符串读入），并输出出现次数最多的 50 个英文单词（按出现次数排序，每行输出 10 个英文单词，共 5 行）。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）

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。

（2）报告正文：请在指定位置填写。本次实验**需要**单独提交源程序文件，请将源程序文件压缩成为一个 code.zip 包提交。共提交两个文件：实验报告.doc 和 code.zip。

（3）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。

（4）提交方式：请在 Blackboard 平台中提交。

（5）发现雷同，所有雷同者该次作业记零分。

三、实验过程及结果

1. 运行以下三个程序（要求每个程序运行 10 次），并对输出结果给出分析。在报告中附上程序截图和详细的文字说明。（5 分）

程序 1:

程序源码：

```
class PrintChar implements Runnable {
    2 usages
    private char charToPrint;// The character to print
    2 usages
    private int times;// The number of times to repeat
    2 usages
    public PrintChar(char c,int t){ // Constructor
        charToPrint = c;
        times = t;
    }
    @Override
    public void run(){ // Override the run() method
        for(int i = 0; i < times; i++) // Repeat n times
            System.out.print(charToPrint);
    }
}
```

图 1.1.1　PrintChar 类源码

```
class PrintNum implements Runnable {
    2 usages
    private int lastNum; // The last number to print
    1 usage
    public PrintNum(int n){ // Constructor
        lastNum = n;
    }
    @Override
    public void run(){ // Override the run() method
        for(int i = 1; i <= lastNum; i++) // Print numbers from 1 to lastNum
            System.out.print(" " + i);
    }
}
```

图 1.1.2　PrintNum 类源码

```
public class TaskThreadDemo {
    public static void main(String[] args) {
        // Create tasks
        Runnable printA= new PrintChar( c: 'a', t: 100);
        Runnable printB= new PrintChar( c: 'b', t: 100);
        Runnable print100= new PrintNum( n: 100);
        // Create threads
        Thread thread1 = new Thread(printA);
        Thread thread2 = new Thread(printB);
        Thread thread3 = new Thread(print100);
        // Start threads
        thread1.start();
        thread2.start();
        thread3.start();
    }
}
```

图 1.1.3 测试类 TaskThreadDemo 源码

程序说明：
在程序一中，三个线程被创建并启动，每个线程负责不同的任务：

（1） PrintChar 类实现了 Runnable 接口，它接受一个字符和一个重复次数。在 run 方法中，该类负责将指定的字符打印指定次数。

- printA 任务使用字符 'a' 并打印 100 次。
- printB 任务使用字符 'b' 并打印 100 次。

（2） PrintNum 类也实现了 Runnable 接口，接受一个整数 lastNum。它的 run 方法负责从 1 打印到 lastNum。

- print100 任务负责打印从 1 到 100 的数字。

（3） 在 main 方法中，我们创建了三个线程 thread1、thread2 和 thread3，分别执行 printA、printB 和 print100 任务，然后依次启动它们。

输出结果：

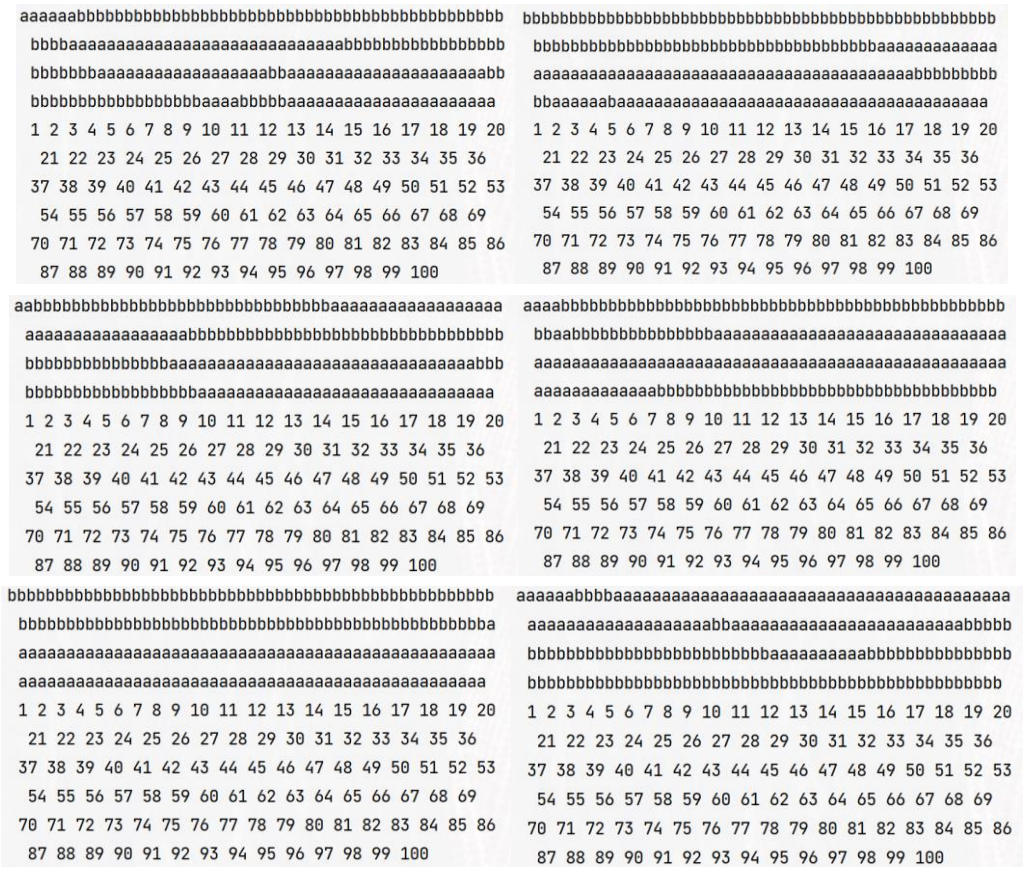

图 1.1.4　部分运行结果对比图

结果分析：

由于线程是并发执行的，每个线程都会按照自己的任务进行打印，并且它们的执行顺序由操作系统的调度决定，因此输出顺序是不可预测的。

三个线程的输出会交错在一起，因为它们是并发执行的，具体的交错方式由 CPU 调度决定。

但在上述运行结果对比图中，我们发现数字序列总是最后输出的，可能的原因是 thread3 的执行被延迟了，导致它在 thread1 和 thread2 完成之后才开始。这种情况的原因可能与线

程调度策略有关：

（1）线程调度的不可预测性：Java 的线程调度器会根据系统资源和负载情况来分配线程的执行顺序。thread3 有时可能被分配到稍后运行，从而导致它在 thread1 和 thread2 之后执行。

（2）CPU 密集型任务：PrintChar 的两个任务 thread1 和 thread2，都在进行密集的字符输出，这可能会暂时占用较多 CPU 资源，使得 thread3 的执行被推迟。

程序 2:

程序源码：

```java
package demo1;

import java.util.concurrent.*;

public class ExecutorDemo {

    public static void main(String[] args) {
        // Create a fixed thread pool with maximum three threads
        ExecutorService executor = Executors.newFixedThreadPool( nThreads: 3);
        // Submit runnable tasks to the executor
        executor.submit(new PrintChar( c: 'a', t: 100));
        executor.submit(new PrintChar( c: 'b', t: 100));
        executor.submit(new PrintNum( n: 100));
        // Shut down the executor
        executor.shutdown();
    }
}
```

图 1.2.1　ExecutorDemo 类源码

程序说明：

（1）**创建线程池**：代码使用 Executors.newFixedThreadPool(3) 创建了一个固定大小的线程池，最多可以同时运行 3 个线程。

（ ）**提交任务**：三个任务被提交到线程池中执行：

PrintChar('a', 100)：负责输出字符 'a' 100 次。

PrintChar('b', 100)：负责输出字符 'b' 100 次。

PrintNum(100)：负责输出从 1 到 100 的数字。

（3）**关闭线程池**：executor.shutdown() 表示不再接受新的任务，已提交的任务将继续执行，直到线程池中的所有任务完成为止。

输出结果：

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbaaabbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbba
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
 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 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 86 87 88 89 90 91
 92 93 94 95 96 97 98 99 100
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
 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 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 86 87 88 89 90 91
 92 93 94 95 96 97 98 99 100
```

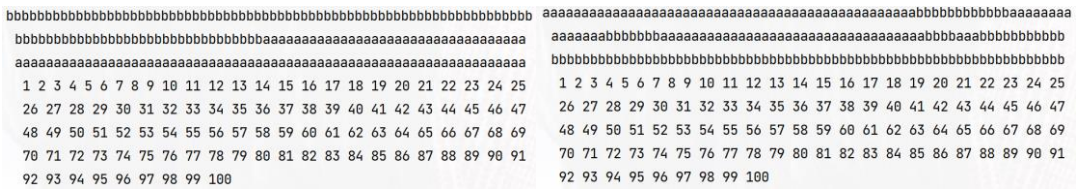

图 1.2.2　部分输出结果对比

结果分析：

由于线程池有三个线程，三个任务会并发执行。因此，输出结果仍是不确定的，三个任务的输出会交错：

（1）PrintChar('a', 100) 和 PrintChar('b', 100) 任务会输出 100 个 a 和 100 个 b，且会交错出现。

（2）PrintNum(100) 任务将输出数字 1 2 3 ... 100。

但在上述运行结果对比图中，我们发现数字序列总是最后输出的，可能的原因是 thread3 的执行被延迟了，导致它在 thread1 和 thread2 完成之后才开始。这种情况的原因可能与线程调度策略有关：

（1）线程调度的不可预测性：Java 的线程调度器会根据系统资源和负载情况来分配线程的执行顺序。thread3 有时可能被分配到稍后运行，从而导致它在 thread1 和 thread2 之后执行。

（2）CPU 密集型任务：PrintChar 的两个任务 thread1 和 thread2，都在进行密集的字符输出，这可能会暂时占用较多 CPU 资源，使得 thread3 的执行被推迟。

程序 3:

程序源码：

```
package demo1;
import java.util.concurrent.*;

public class AccountwithoutSync {
    2 usages
    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());
    }
```

图 1.3.1　AccountwithoutSync 类的部分源码

```java
// A thread for adding a penny to the account
1 usage
private static class AddAPennyTask implements Runnable {
    public void run() {
        account.deposit( amount: 1);
    }
}
// An inner class for account
2 usages
private static class Account {
    3 usages
    private int balance = 0;
    1 usage
    public int getBalance(){
        return balance;
    }
    1 usage
    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( millis: 5);
        } catch (InterruptedException ex) {
        }
        balance = newBalance;
    }
}
```

图 1.3.2　AccountwithoutSync 的内部类的源码

程序说明：

（1）Account 类：

- 该类表示一个简单的账户，包含一个整数字段 `balance`，表示账户余额。
- `getBalance()` 方法返回账户余额。
- `deposit(int amount)` 方法将指定金额存入账户，但由于没有同步机制，`balance` 的修改可能会受到并发线程的干扰。
- `deposit()` 方法中加入了 `Thread.sleep(5);` 来人为增加延迟，使问题更明显。

（2）AddAPennyTask 类：

- `AddAPennyTask` 实现了 `Runnable` 接口，表示一个任务线程。
- 在 `run()` 方法中，调用 `account.deposit(1);` 将 1 分钱存入账户。

（3）AccountwithoutSync 类：

- `main()` 方法中，首先创建了一个 `ExecutorService` 对象 `executor`，它是一个缓存线程池 (`newCachedThreadPool()`)，可以动态分配和回收线程资源。
- 然后，使用一个循环启动 100 个 `AddAPennyTask` 任务，每个任务都会调用 `account.deposit(1)` 方法存入一分钱。
- 调用 `executor.shutdown();` 后，不再接受新任务，并等待所有任务完成。
- `while (!executor.isTerminated()) {}` 会一直循环，直到所有线程任务完成。

- 当所有线程执行完毕后，输出账户余额 `account.getBalance()`。

输出结果：

```
What is balance? 2
What is balance? 1
What is balance? 1
What is balance? 1
```

图 1.3.3　部分输出结果对比图

结果分析：

理论上账户余额应该为 100，但实际输出却远小于 100，几乎全为 1。这是因为多个线程在写入新 balance 时彼此覆盖了结果。

由于 deposit() 方法没有同步，多个线程可能会在读取和修改 balance 时发生**竞态条件**，导致以下**数据不一致**的问题：

每个线程读取 balance 后都会等待 5 毫秒，再将新值写回 balance。如果多个线程几乎同时执行 deposit()，它们会读取相同的初始 balance 值，这导致最后 balance 的值可能小于预期值（100）。

修复方法：

要确保线程安全，可以对 deposit 方法进行同步控制。通过在 deposit 方法上添加 synchronized 关键字：

```
public synchronized void deposit(int amount){
```

图 1.3.4　修复方法

修复结果：

```
What is balance? 100
```

图 1.3.5　修复后的运行结果图

2. 编写 Java 应用程序实现如下功能：第一个线程生成一个随机数，第二个线程每隔一段时间读取第一个线程生成的随机数，并判断它是否是奇数。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、完整的运行结果截图和简要文字说明。（10 分）

程序源码：

```
public class RandomNumberChecker {
    // 共享资源：用来存放随机数
    5 usages
    private static volatile int randomNumber;

    public static void main(String[] args) {
        // 创建生成随机数的线程
        Thread generatorThread = new Thread(new RandomNumberGenerator());
        // 创建检查奇偶性的线程
        Thread checkerThread = new Thread(new OddNumberChecker());
        // 启动线程
        generatorThread.start();
        checkerThread.start();
    }
```

图 2.1　RandomNumberChecker 类主方法的内容

```
// 第一个线程：生成随机数
1 usage
private static class RandomNumberGenerator implements Runnable {
    1 usage
    private Random random = new Random();

    @Override
    public void run() { // 重写run()方法
        while (true) {
            randomNumber = random.nextInt( bound: 100); // 生成 0 到 99 之间的随机数
            System.out.println("Generated Random Number: " + randomNumber);
            try {
                Thread.sleep( millis: 1000); // 每隔一秒生成一个随机数
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

图 2.2　生成随机数的线程类 RandomNumberGenerator

```
// 第二个线程：判断随机数是否为奇数
1 usage
private static class OddNumberChecker implements Runnable {

    @Override
    public void run() { // 重写run()方法
        while (true) {
            if (randomNumber % 2 != 0) { // 判断随机数是否为奇数
                System.out.println("The number " + randomNumber + " is odd.");
            } else {
                System.out.println("The number " + randomNumber + " is even.");
            }
            try {
                Thread.sleep( millis: 2000); // 每隔 2 秒检查一次随机数
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

图 2.3　判断随机数是否为奇数的线程类 OddNumberChecker

程序说明：

（1）共享资源 **randomNumber**：这是两个线程共享的变量，用来存储随机生成的数。使用了 volatile 关键字，确保线程可以看到变量的最新值。

（2）**RandomNumberGenerator 类**：实现了 Runnable 接口，在 run 方法中**生成随机数**。每隔一秒生成一个新的随机数并更新 randomNumber。

（3）**OddNumberChecker 类**：实现了 Runnable 接口，在 run 方法中检查 randomNumber

是否为奇数。每隔 2 秒读取并判断当前 randomNumber 的奇偶性。
（4）**线程启动**：在 main 方法中，创建 Thread 实例时，将 RandomNumberGenerator 和 OddNumberChecker 对象传入构造方法，并启动两个线程。

运行结果：

```
The number 83 is odd.
Generated Random Number: 83
Generated Random Number: 89
The number 89 is odd.
Generated Random Number: 95
Generated Random Number: 94
The number 94 is even.
Generated Random Number: 95
Generated Random Number: 46
The number 46 is even.
```

图 2.4　部分输出结果

3. 编写 Java 应用程序实现如下功能：第一个线程输出数字 1-26，第二个线程输出字母 A-Z，输出的顺序为 1A2B3C...26Z，即每 1 个数字紧跟着 1 个字母的方式。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。（10 分）
程序源码：

```
public class NumberLetterPrinter {
    6 usages
    private static final Object lock = new Object(); // 用于同步的锁对象
    4 usages
    private static boolean numberTurn = true; // 用于控制输出顺序

    public static void main(String[] args) {
        // 创建数字输出线程
        Thread numberThread = new Thread(new NumberPrinter());
        // 创建字母输出线程
        Thread letterThread = new Thread(new LetterPrinter());
        // 启动线程
        numberThread.start();
        letterThread.start();
    }
```

图 3.1　NumberLetterPrinter 类

```
// 线程一：输出数字的线程
1 usage
private static class NumberPrinter implements Runnable {

    @Override
    public void run() {
        for (int i = 1; i <= 26; i++) { // 从 1 到 26 输出数字
            synchronized (lock) { // 获取锁
                while (!numberTurn) {
                    try {
                        lock.wait(); // 等待字母线程的通知
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                System.out.print(i); // 输出数字
                numberTurn = false;  // 切换到字母线程
                lock.notify();       // 通知字母线程
            }
        }
    }
}
```

图 3.2　输出数字的线程类

```java
// 线程二：输出字母的线程
1 usage
private static class LetterPrinter implements Runnable {
    @Override
    public void run() {
        for (char c = 'A'; c <= 'Z'; c++) { // 从 A 到 Z 输出字母
            synchronized (lock) { // 获取锁
                while (numberTurn) {
                    try {
                        lock.wait(); // 等待数字线程的通知
                    } catch (InterruptedException e) {
                        Thread.currentThread().interrupt();
                    }
                }
                System.out.print(c); // 输出字母
                numberTurn = true;   // 切换到数字线程
                lock.notify();       // 通知数字线程
            }
        }
    }
}
```

图 3.3　输出字母的线程

程序说明：

（1）**共享资源 lock**：使用一个 lock 对象作为**共享锁**，以控制两个线程的同步。

（ ）**控制输出顺序的标志 numberTurn**：该布尔变量 numberTurn 用于指示当前应该输出数字还是字母。true 表示数字线程的回合，false 表示字母线程的回合。

（??）**数字输出线程 NumberPrinter**：

- 每次输出数字前，先检查是否为自己的回合 (`numberTurn == true`)。
- 如果不是数字线程的回合（即 `numberTurn == false`），则调用 `lock.wait()` 进行等待。
- 输出数字后，将 `numberTurn` 设置为 `false`，表示轮到字母线程输出，并调用 `lock.notify()` 唤醒字母线程。

（4）**字母输出线程 LetterPrinter**：

类似于数字线程，字母线程在每次输出前检查 numberTurn。

如果不是字母线程的回合（即 numberTurn == true），则调用 lock.wait() 进行等待。

输出字母后，将 numberTurn 设置为 true，表示轮到数字线程输出，并调用 lock.notify() 唤醒数字线程。

（5）**同步控制**：两个线程通过 **synchronized** 块、wait() 和 notify() 实现同步控制，确保它们交替输出 1A2B3C...26Z 的模式。

运行结果：

```
1A2B3C4D5E6F7G8H9I10J11K12L13M14N15O16P17Q18R19S20T21U22V23W24X25Y26Z
Process finished with exit code 0
```

图 3.4 运行结果图

4. 编写 Java 应用程序实现如下功能：创建工作线程，模拟银行现金账户存款操作。多个线程同时执行存款操作时，如果不使用同步处理，会造成账户余额混乱，要求使用 syncrhonized 关键字同步代码块，以保证多个线程同时执行存款操作时，银行现金账户存款的有效和一致。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、运行结果截图和详细的文字说明。（10 分）

程序源码：

```
public class BankAccountDemo {
    public static void main(String[] args) {
        // 创建银行账户对象
        BankAccount account = new BankAccount();
        // 创建三个线程，模拟多线程存款操作
        Thread depositor1 = new Thread(new DepositTask(account, amount: 100));
        Thread depositor2 = new Thread(new DepositTask(account, amount: 200));
        Thread depositor3 = new Thread(new DepositTask(account, amount: 300));
        // 启动线程
        depositor1.start();
        depositor2.start();
        depositor3.start();
    }
}
```

图 4.1 BankAccountDemo 主类

```
// 银行账户类
4 usages
class BankAccount {
    3 usages
    private int balance = 0; // 账户余额
    // 获取账户余额
    no usages
    public int getBalance() {
        return balance;
    }
    // 存款操作，使用synchronized关键字保证线程安全
    1 usage
    public void deposit(int amount) {
        synchronized (this) { // 同步代码块
            int newBalance = balance + amount;
            System.out.println(Thread.currentThread().getName() + " 存入 " + amount + " 元，新的余额是：" + newBalance);
            balance = newBalance; // 更新余额
        }
    }
}
```

图 4.2 银行账户类 BankAccount

```
// 存款任务类，实现Runnable接口
3 usages
class DepositTask implements Runnable {
    2 usages
    private final BankAccount account; // 银行账户对象
    2 usages
    private final int amount; // 存款金额
    // 构造方法
    3 usages
    public DepositTask(BankAccount account, int amount) {
        this.account = account;
        this.amount = amount;
    }
    // 重写run方法
    @Override
    public void run() {
        account.deposit(amount); // 执行存款操作
    }
}
```

图 4.3 存款任务类 DepositTask

程序说明：

（1）BankAccount 类：表示银行账户。

- balance：存储账户余额的变量。
- deposit(int amount)：存款方法，将存款金额加入到 balance 中。
- synchronized (this)：使用同步块保证只有一个线程可以同时执行 deposit 方法，避免多个线程同时更新 balance 时的数据不一致问题。

（2）DepositTask 类：实现 Runnable 接口，表示一个存款任务。

- account：引用 BankAccount 对象，表示要进行存款的账户。
- amount：表示每次存款的金额。
- run() 方法中调用 account.deposit(amount) 来执行存款操作。

（3）main 方法：

- 创建 BankAccount 对象 account。
- 创建三个线程，每个线程使用不同的存款金额（100、200、300），模拟多个线程同时对账户进行存款。
- 启动三个线程执行存款操作。

运行结果：

```
Thread-0 存入 100 元，新的余额是：100
Thread-2 存入 300 元，新的余额是：400
Thread-1 存入 200 元，新的余额是：600
```

图 4.4 运行结果

5. 编写 Java 应用程序，根据用户输入的 5 个日期（每行一个日期），计算相邻两个日期之间间隔的时数，共 4 个结果（时数）。注：用户输入的时间格式为“××××年××月××日××时××分××秒”，输出的时间格式为“××日××时××分××秒”。在报告中应有程序截图、完整的运行结果截图和简要文字说明。（20 分）

程序源码：

```
public class DateIntervalCalculator {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        SimpleDateFormat dateFormat = new SimpleDateFormat( pattern: "yyyy年MM月dd日HH时mm分ss秒"); // 定义输入日期的格式
        ArrayList<Date> dates = new ArrayList<>(); // 用于存储输入的日期
        // 提示用户输入日期
        System.out.println("请输入5个日期（格式：××××年××月××日××时××分××秒）：");
        for (int i = 0; i < 5; i++) {
            System.out.print("输入第 " + (i + 1) + " 个日期：");
            String input = scanner.nextLine();
            try {
                Date date = dateFormat.parse(input); // 解析用户输入的日期
                dates.add(date); // 存储解析后的日期
            } catch (ParseException e) {
                System.out.println("日期格式错误，请输入正确格式！");
                i--; // 重试当前输入
            }
        }
        scanner.close(); // 关闭输入流
        // 计算相邻日期间隔的时数并输出结果
        System.out.println("\n相邻日期间隔时数：");
        for (int i = 0; i < dates.size() - 1; i++) {
            String formattedInterval = formatInterval(dates.get(i), dates.get(i + 1));
            System.out.println("第 " + (i + 1) + " 个间隔：  " + formattedInterval);
        }
    }
```

图 5.1 DateIntervalCalculator 类主方法的源码

```java
// 格式化时间间隔为"××日××时××分××秒"
1 usage
private static String formatInterval(Date date1, Date date2) {
    long diffInMillis = Math.abs(date2.getTime() - date1.getTime()); // 计算时间间隔（单位为毫秒）
    Duration duration = Duration.ofMillis(diffInMillis); // 将时间间隔转换为Duration对象
    // 计算时间间隔的日、时、分、秒
    long days = duration.toDays();
    long hours = duration.toHours() % 24;
    long minutes = duration.toMinutes() % 60;
    long seconds = duration.getSeconds() % 60;

    return String.format("%d日%d时%d分%d秒", days, hours, minutes, seconds);
}
```

图 5.2 计算时间间隔的函数

程序说明：

（1）LocalDateTime 和 DateTimeFormatter：使用 LocalDateTime 和 DateTimeFormatter 解析用户输入的日期，使代码更简洁，不需要使用 Date 类。

（2）Duration.between：通过 Duration.between(date1, date2) 直接计算时间差。

（3）toHoursPart、toMinutesPart、toSecondsPart：Duration 中的这些方法可以直接获取时间间隔中的小时、分钟和秒部分，无需手动计算。

运行结果：

```
请输入5个日期（格式：××××年××月××日××时××分××秒）：
输入第 1 个日期：2018年10月29日18时35分02秒
输入第 2 个日期：2024年10月29日23时35分31秒
输入第 3 个日期：2024年10月29日12时35分01秒
输入第 4 个日期：2022年10月29日18时35分01秒
输入第 5 个日期：2024年10月29日18时35分01秒

相邻日期间隔时数：
第 1 个间隔： 2192日5时0分29秒
第 2 个间隔： 0日11时0分30秒
第 3 个间隔： 730日18时0分0秒
第 4 个间隔： 731日0时0分0秒
```

图 5.3 程序运行结果图

6. 编写 Java 应用程序，实现稀疏矩阵的加法和乘法运算，其中稀疏矩阵是指矩阵中的大部分元素的值为 0。用户在命令行输入矩阵时矩阵的大小可能有错，因此需要使用异常处理。在报告中应有程序截图、完整的运行结果截图和简要文字说明。（15 分）

程序源码：

```
public class SparseMatrixDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            // 创建矩阵A
            System.out.print("请输入矩阵A的行数和列数（空格分隔）：");
            int rowsA = scanner.nextInt(); int colsA = scanner.nextInt();
            SparseMatrix matrixA = new SparseMatrix(rowsA, colsA);
            System.out.println("请输入矩阵A的非零元素（格式：行 列 值），输入-1结束：");
            while (true) { // 循环读取非零元素
                int row = scanner.nextInt();
                if (row == -1) break;
                int col = scanner.nextInt();
                int value = scanner.nextInt();
                matrixA.setElement(row, col, value);
            }
            // 创建矩阵B
            System.out.print("请输入矩阵B的行数和列数（空格分隔）：");
            int rowsB = scanner.nextInt(); int colsB = scanner.nextInt();
            SparseMatrix matrixB = new SparseMatrix(rowsB, colsB);
            System.out.println("请输入矩阵B的非零元素（格式：行 列 值），输入-1结束：");
            while (true) { // 循环读取非零元素
                int row = scanner.nextInt();
                if (row == -1) break;
                int col = scanner.nextInt();
                int value = scanner.nextInt();
                matrixB.setElement(row, col, value);
            }
            // 计算并输出矩阵相加的结果
            System.out.println("矩阵A+B结果：");
            try {
                SparseMatrix sumMatrix = matrixA.add(matrixB); // 计算矩阵相加
                sumMatrix.printMatrix(); // 输出结果
            } catch (IllegalArgumentException e) {
                System.out.println("加法错误：" + e.getMessage());
            }
            // 计算并输出矩阵相乘的结果
            System.out.println("矩阵A*B结果：");
            try {
                SparseMatrix productMatrix = matrixA.multiply(matrixB); // 计算矩阵相乘
                productMatrix.printMatrix(); // 输出结果
            } catch (IllegalArgumentException e) {
                System.out.println("乘法错误：" + e.getMessage());
            }
        } catch (Exception e) {
            System.out.println("输入错误：" + e.getMessage());
```

图 6.1 主程序源码

主程序说明：

用户首先输入两个矩阵的行数和列数，然后输入每个矩阵的非零元素。程序会尝试计算这两个矩阵的和与乘积，并输出结果。如果在计算过程中遇到非法参数异常，会捕获异常并输出错误信息。最后，关闭 Scanner 对象以释放资源。

```java
class SparseMatrix {
    7 usages
    private int rows; // 矩阵的行数
    8 usages
    private int cols; // 矩阵的列数
    6 usages
    private Map<String, Integer> elements; // 用于存储非零元素
    // 构造函数
    4 usages
    public SparseMatrix(int rows, int cols) {
        this.rows = rows;
        this.cols = cols;
        this.elements = new HashMap<>();
    }
    // 设置矩阵元素
    5 usages
    public void setElement(int row, int col, int value) {
        if (value != 0) {
            elements.put(row + "," + col, value);
        }
    }
    // 获取矩阵元素
    4 usages
    public int getElement(int row, int col) {
        return elements.getOrDefault( key: row + "," + col,  defaultValue: 0);
    }
```

图 6.2 矩阵类的基础成员和方法

```java
// 矩阵相加
public SparseMatrix add(SparseMatrix other) throws IllegalArgumentException {
    if (this.rows != other.rows || this.cols != other.cols) {
        throw new IllegalArgumentException("矩阵大小不一致，无法进行加法运算。");
    }
    SparseMatrix result = new SparseMatrix(this.rows, this.cols); // 创建结果矩阵
    // 添加当前矩阵的非零元素
    for (Map.Entry<String, Integer> entry : this.elements.entrySet()) {
        String key = entry.getKey();
        result.setElement(
                Integer.parseInt(key.split( regex: ",")[0]),
                Integer.parseInt(key.split( regex: ",")[1]),
                entry.getValue()
        );
    }
    // 添加other矩阵的非零元素
    for (Map.Entry<String, Integer> entry : other.elements.entrySet()) {
        String key = entry.getKey();
        int row = Integer.parseInt(key.split( regex: ",")[0]);
        int col = Integer.parseInt(key.split( regex: ",")[1]);
        result.setElement(row, col,  value: result.getElement(row, col) + entry.getValue());
    }
    return result;
}
```

图 6.3 矩阵加法

```java
// 矩阵相乘
1 usage
public SparseMatrix multiply(SparseMatrix other) throws IllegalArgumentException {
    if (this.cols != other.rows) {
        throw new IllegalArgumentException("矩阵维度不匹配，无法进行乘法运算。");
    }
    SparseMatrix result = new SparseMatrix(this.rows, other.cols); // 创建结果矩阵
    for (Map.Entry<String, Integer> entry : this.elements.entrySet()) {
        int row = Integer.parseInt(entry.getKey().split( regex: ",")[0]);
        int col = Integer.parseInt(entry.getKey().split( regex: ",")[1]);
        int value = entry.getValue();
        for (int k = 0; k < other.cols; k++) {
            int otherValue = other.getElement(col, k);
            if (otherValue != 0) {
                result.setElement(row, k,  value: result.getElement(row, k) + value * otherValue);
            }
        }
    }
    return result;
}
```

图 6.4 矩阵乘法

程序说明：

要实现稀疏矩阵的加法和乘法运算，我们可以创建一个 SparseMatrix 类来表示稀疏矩阵。由于大部分元素都是零，可以使用哈希表或列表来存储非零元素，以节省空间。然后，我们定义加法和乘法操作。

（1）SparseMatrix 类表示稀疏矩阵，其中非零元素存储在 elements 哈希表中，键是"行,列"的形式，值是对应的元素值。

（2）add 和 multiply 方法分别实现矩阵的加法和乘法，并进行矩阵大小和维度匹配的检查。若不匹配，则抛出 IllegalArgumentException。

（3）异常处理用于捕获输入错误或不匹配的矩阵大小。

运行结果：

```
请输入矩阵A的行数和列数（空格分隔）：3 3
请输入矩阵A的非零元素（格式：行 列 值），输入-1结束：
1 1 3
2 2 6
0 1 1
-1
请输入矩阵B的行数和列数（空格分隔）：3 3
请输入矩阵B的非零元素（格式：行 列 值），输入-1结束：
0 0 1
1 1 2
2 2 3
-1
矩阵A+B结果：
1 1 0
0 5 0
0 0 9
矩阵A*B结果：
0 2 0
0 6 0
0 0 18
```

图 6.5 正常情况下的运行结果

```
请输入矩阵A的行数和列数（空格分隔）：3 4
请输入矩阵A的非零元素（格式：行 列 值），输入-1结束：
1 1 1
 2 2 2
-1
请输入矩阵B的行数和列数（空格分隔）：4 3
请输入矩阵B的非零元素（格式：行 列 值），输入-1结束：
1 1 1
-1
矩阵A+B结果：
加法错误：矩阵大小不一致，无法进行加法运算。
矩阵A*B结果：
0 0 0
0 1 0
0 0 0
```

图 6.6 异常情况下的输出结果

7. 编写 Java 应用程序，统计分析网页 https://en.szu.edu.cn/About/About2.htm 中关于深圳大学的介绍的英文文章（包括题目 About）中每个英文单词出现的次数（不区分大小写，不要写爬虫，可以把整篇文章的内容当作一个字符串读入），并输出出现次数最多的 50 个英文单词（按出现次数排序，每行输出 10 个英文单词，共 5 行）。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）

程序源码：

```java
import java.util.*;
import java.util.stream.Collectors;

public class WordNumDemo {
    public static void main(String[] args) {
        // 将网页内容复制到content字符串中
        String content =  "About\n" +
                "Shenzhen University (SZU) is committed to excellence in teaching, research and social ser
                "SZU, which is based in Shenzhen, China's first Special Economic Zone and a key city in th
                "Established in 1983, SZU received support from top Chinese universities including Peking
                "SZU offers a wide array of undergraduate and graduate programs and provides students with
                "SZU is an integral part of the SEZ, a thriving technology and innovation hub. With two ca
                "SZU is accelerating its pace toward internationalization, providing a variety of global l
        // 1. 清理文本内容，拆分为单词列表（忽略大小写）
        String[] words = content.toLowerCase().split( regex: "[^a-z]+");
        // 2. 使用 Map 统计每个单词出现的次数
        Map<String, Integer> wordCountMap = new HashMap<>();
        for (String word : words) {
            if (!word.isBlank()) { // 跳过空字符串
                wordCountMap.put(word, wordCountMap.getOrDefault(word,  defaultValue: 0) + 1);
            }
        }
        // 3. 将 Map 转换为按出现次数降序排列的列表
        List<Map.Entry<String, Integer>> sortedWordList = wordCountMap.entrySet()   // 将Map转换为Entry集合
                .stream()   // 使用流处理
                .sorted((e1, e2) -> e2.getValue().compareTo(e1.getValue())) // 按出现次数降序排序
                .limit( maxSize: 50)  // 只取前50个
                .collect(Collectors.toList());  // 转换为列表

        // 4. 输出出现次数最多的前50个单词，每行10个单词
        for (int i = 0; i < sortedWordList.size(); i++) {
            System.out.print(sortedWordList.get(i).getKey() + " ");
            if ((i + 1) % 10 == 0) System.out.println(); // 每10个单词换行
        }
    }
}
```

图 7.1 程序源代码截图

程序说明：

（1）**清理和拆分：**

将内容转换为小写，然后使用正则表达式[^a-z]+将字符串拆分为单词数组，忽略所有非字母字符。

（2）**统计词频：**

通过 Map<String, Integer>存储每个单词的出现次数。其中，wordCountMap.put(word, wordCountMap.getOrDefault(word, 0) + 1); 这行代码是循环体的核心部分，用于更新 wordCountMap 中单词的计数。

wordCountMap.getOrDefault(word, 0)：这个方法尝试从 wordCountMap 中获取当前单词的计数。如果单词存在，则返回其计数；如果不存在，则返回 0。

（3）**排序和限制：**

按出现次数降序排序，限制输出前 50 个单词，从而达到输出 5 行每行 10 个单词的目的。

（4）**最后格式化输出：**

每行输出 10 个单词，共 5 行。

运行结果：

```
and the in of szu university to with a is
china students research an programs are sez has its for
education shenzhen self global variety innovation s excellence collaborations faculty
passion develop higher learning technology special universities city teaching top
established opportunities interests economic high zone academic from new including
```

图 7.2 运行结果

四、实验总结与体会

(写写感想、建议等)

实验总结：

1. **线程同步的重要性**。在程序 3 中，模拟银行账户存款操作时，如果不使用同步处理，多个线程同时执行存款操作会导致账户余额混乱。通过使用 synchronized 关键字同步代码块，我确保了多个线程同时执行存款操作时的银行现金账户存款的有效性和一致性。这让我认识到，在多线程环境下，正确地管理共享资源是保证程序正确性的关键。

2. **集合类的应用**。Java 的集合类在实验中也发挥了重要作用。例如，在统计分析网页中英文单词出现次数的实验中，我使用了 Map 接口来存储每个单词及其出现的次数。这不仅提高了代码的可读性，也使得数据的管理和操作变得更加高效。此外，集合类在处理稀疏矩阵时也显示出了其优势，通过存储非零元素，大大节省了内存空间。

3. **线程通信与协作**。在输出 1A2B3C...26Z 的实验中，我通过 wait()和 notify()方法实现了两个线程之间的同步控制，确保了输出的顺序。这个过程让我理解了线程间的协作机制，以及如何通过共享资源和同步机制来控制线程的执行顺序。

实验难点：

1. **线程调度的不可预测性**：在实验中，我遇到了线程执行顺序不可预测的问题。例如，在程序 1 和程序 2 中，数字序列总是最后输出，这让我意识到线程调度策略对程序输出的影响。

2. **同步机制的实现**：在实现线程同步时，我遇到了一些挑战，尤其是在理解 synchronized 关键字和 Lock 机制的区别和适用场景上。通过不断的实验和调试，我逐渐掌握了这些同步机制的使用方法。

3. **异常处理**：在实现稀疏矩阵的加法和乘法运算时，我需要处理用户输入错误或矩阵大小不匹配的情况。这要求我熟练掌握 Java 的异常处理机制，确保程序在遇到错误输入时能够给出正确的反馈。

五、成绩评定及评语

1.指导老师批阅意见：

2.成绩评定：

指导教师签字：毛斐巧

2024 年　　月　　日