Java HW3

姓名：黄文杰

学号：3210103379

1. 寻找JDK库中的不变类

String类

源码摘要：

```java
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {
    // 内部字符数组
    @Stable
    private final byte[] value;
    private final byte coder;
    private int hash; // Default to 0
    private boolean hashIsZero; // Default to false;

    public String() {
        this.value = "".value;
        this.coder = "".coder;
    }

    public String(char[] value) {
        this(value, 0, value.length, null);
    }

    public String(String original) {
        this.value = original.value;
        this.coder = original.coder;
        this.hash = original.hash;
        this.hashIsZero = original.hashIsZero;
    }

    public String(char[] value, int offset, int count) {
        this(value, offset, count, rangeCheck(value, offset, count));
    }

    // ...其他构造函数和方法...

    public String substring(int beginIndex) {
        return substring(beginIndex, length());
    }
```

```
36
37      public String substring(int beginIndex, int endIndex) {
38          int length = length();
39          checkBoundsBeginEnd(beginIndex, endIndex, length);
40          if (beginIndex == 0 && endIndex == length) {
41              return this;
42          }
43          int subLen = endIndex - beginIndex;
44          return isLatin1() ? StringLatin1.newString(value, beginIndex,
    subLen)
45                            : StringUTF16.newString(value, beginIndex,
    subLen);
46      }
47      //    Tips：StringLatin1.newString()方法详情如下（StringUTF16.newString（）
    同理）：
48      //    public static String newString(byte[] val, int index, int len) {
49      //        if (len == 0) {
50      //            return "";
51      //        }
52      //        return new String(Arrays.copyOfRange(val, index, index + len),
53      //                LATIN1);
54      //    }
55
56      public static String valueOf(char[] data) {
57          return new String(data);
58      }
59
60      // ...其他方法...
61  }
62
```

源码分析：

1. `value` 字段是 `final` 的： `String` 类内部的字符数组 `value` 被声明为 `final`，这意味着一旦分配了内存并初始化了数组，就不能再改变数组的引用或内容。这是确保字符串内容不可变性的重要步骤。
2. 构造函数中初始化 `value` 和 `coder` 字段： 在构造函数中，`value` 和 `coder` 字段被初始化，一旦初始化，它们不能被修改。这确保了一旦创建了 `String` 对象，它的内容和编码方式是固定的。
3. 所有修改字符串内容（或其他成员变量）的方法都会返回新的 `String` 对象，而不是修改原有对象的内容。例如 `substring` 方法创建新的 `String` 对象： `substring` 方法根据给定的 `beginIndex` 和 `endIndex` 创建一个新的 `String` 对象来表示子字符串，而不是修改原有对象的内容。这保持了原始字符串的不可变性。
4. `String` 类本身是final类型，它不能被其他类所继承。
5. `String` 类不提供任何公共方法来修改字符串的字符内容。没有任何方法允许直接修改 `value` 字段，确保了字符串的不可变性。

这就是为什么 `String` 类是一个不变类的原因。

Integer类

源码摘要：

```
public final class Integer extends Number
        implements Comparable<Integer>, Constable, ConstantDesc {
    // 内部int值
    private final int value;

    public Integer(int value) {
        this.value = value;
    }

    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high) {
            return IntegerCache.cache[i + (-IntegerCache.low)];
        }
        return new Integer(i);
    }

    public int intValue() {
        return value;
    }

    // ...其他方法...
}

```

源码分析：

1. `value`字段是声明为`private final int`，这意味着一旦`Integer`对象被创建，其内部整数值就不能被修改。
2. `Integer`类中提供了一些静态工厂方法，如`valueOf`，它们返回现有的`Integer`对象，或者创建一个新的对象。这确保了整数值相同的两个`Integer`对象都是相等的。
3. `Integer`类没有提供任何公共方法来修改其内部整数值。没有方法允许直接修改`value`字段，确保了`Integer`对象的不可变性。
4. `Integer`类本身是final类型，它不能被其他类所继承。

这就是为什么`Integer`类是一个不变类的原因。

LocalDate类

源码摘要：

```
public final class LocalDate
        implements Temporal, TemporalAdjuster, ChronoLocalDate, Serializable
{
    private final int year;
    private final int month;
    private final int day;

    public static LocalDate of(int year, Month month, int dayOfMonth) {

```

```
        YEAR.checkValidValue(year);
        Objects.requireNonNull(month, "month");
        DAY_OF_MONTH.checkValidValue(dayOfMonth);
        return create(year, month.getValue(), dayOfMonth);
    }

    public static LocalDate of(int year, int month, int dayOfMonth) {
        YEAR.checkValidValue(year);
        MONTH_OF_YEAR.checkValidValue(month);
        DAY_OF_MONTH.checkValidValue(dayOfMonth);
        return create(year, month, dayOfMonth);
    }

    private static LocalDate create(int year, int month, int dayOfMonth) {
        if (dayOfMonth > 28) {
            int dom = switch (month) {
                case 2 -> (IsoChronology.INSTANCE.isLeapYear(year) ? 29 :
28);
                case 4, 6, 9, 11 -> 30;
                default -> 31;
            };
            if (dayOfMonth > dom) {
                if (dayOfMonth == 29) {
                    throw new DateTimeException("Invalid date 'February 29'
as '" + year
                                                    + "'is not a leap year");
                } else {
                    throw new DateTimeException("Invalid date '" +
Month.of(month).name()
                                                    + " " + dayOfMonth + "'");
                }
            }
        }
        return new LocalDate(year, month, dayOfMonth);
    }

    private LocalDate(int year, int month, int dayOfMonth) {
        // 构造函数用于创建对象，但不提供修改属性的方法
        this.year = year;
        this.month = (short) month;
        this.day = (short) dayOfMonth;
    }

    // ...其他构造函数和方法...

    public LocalDate plusDays(long daysToAdd) {
        if (daysToAdd == 0) {
            return this;
        }
        long dom = day + daysToAdd;
        if (dom > 0) {
            if (dom <= 28) {
                return new LocalDate(year, month, (int) dom);
            } else if (dom <= 59) { // 59th Jan is 28th Feb, 59th Feb is
31st Mar
```

```
                long monthLen = lengthOfMonth();
                if (dom <= monthLen) {
                    return new LocalDate(year, month, (int) dom);
                } else if (month < 12) {
                    return new LocalDate(year, month + 1, (int) (dom -
monthLen));
                } else {
                    YEAR.checkValidValue(year + 1);
                    return new LocalDate(year + 1, 1, (int) (dom -
monthLen));
                }
            }
        }

        long mjDay = Math.addExact(toEpochDay(), daysToAdd);
        return LocalDate.ofEpochDay(mjDay);
    }

    // ...其他方法...
}

```

源码分析：

1. `year`、`month` 和 `day` 属性声明为 `final`：这三个属性都被声明为`private final`，这意味着一旦对象被创建，它们的值不可被修改。
2. 该类不提供修改属性的公共方法。`LocalDate`类提供了一系列的静态工厂方法（例如`of`方法）和构造函数来创建新的LocalDate对象，但没有提供公共方法来修改已创建的对象的属性。这确保了对象的年、月和日属性不可更改。
3. 所有修改内部成员变量的方法都会返回新的 `LocalDate` 对象，而不是修改原有对象的内容。诸如`plusDays`等日期计算方法都返回一个新的`LocalDate`对象，以表示进行日期计算后的结果。这意味着不会修改原始对象，而是创建一个新的对象来表示新的日期。
4. 该类有内部的不可变性验证，在构造函数和创建新`LocalDate`对象的方法中，有内部验证确保日期值的有效性，但如果日期值无效，它会抛出`DateTimeException`异常，而不是修改原始对象。
5. `LocalDate`类本身是final类型，它不能被其他类所继承。

这就是为什么`LocalDate`类是一个不变类的原因。

不变类共性

1. 不变类不提供公共方法来修改其内部状态。没有setter方法或其他公共方法允许直接修改对象的属性。
2. 保证类不会被扩展。防止粗心或恶意的子类假装对象的状态已改变，一般做法是使这个类为final，或者让类的所有构造器都变为私有的或包级私有的，并添加公共的静态工厂来代替公有的构造器。
3. 所有域都是final的。
4. 所有域都是private的。
5. 都能确保对于任何可变组件的互斥访问。如果类有指向可变对象的域，则能确保该类的客户端无法获得指向这些对象的引用
6. 不变类的内部状态（成员变量、属性）一旦被设置，就不可再被修改。这通常通过将成员变量声明为 `final` 来实现。
7. 不变类都不能对原对象进行修改，都是返回一个新的对象。

2. 对String、StringBuilder以及StringBuffer进行源代码分析

分析其主要数据组织及功能实现，有什么区别?

由于String类的源码摘要和分析已在第一部分包含，这里不在赘述。下面主要分析StringBuilder和StringBuffer类。

StringBuilder

源码摘要:

```
public final class StringBuilder
    extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>, CharSequence
{
    // StringBuilder继承AbstractStringBuilder，实现Serializable接口和
CharSequence接口。

    public StringBuilder() {
        super(16); // 初始容量为16个字符
    }

    public StringBuilder(int capacity) {
        super(capacity);
    }

    public StringBuilder(String str) {
        super(str);
    }

    public StringBuilder append(String str) {
        super.append(str);
        return this;
    }

    // 其他append方法用于添加字符、字符数组、其他数据类型等

    public StringBuilder reverse() {
        super.reverse();
        return this;
    }

    // 其他操作方法如insert、delete、replace等
}

```

其父类源码:

```
abstract sealed class AbstractStringBuilder implements Appendable,
CharSequence
    permits StringBuilder, StringBuffer {
    byte[] value;
    byte coder;
    int count;
    boolean maybeLatin1;
    //...
}
```

AbstractStringBuilder这个类中定义了字符串的储存形式,但与 String 不同的是它没有加 final 修饰符,所以是可以动态改变的。

主要数据组织和功能实现:

- `StringBuilder` 继承自 `AbstractStringBuilder`，该类内部使用字符数组 `char[]` 来存储字符串的字符序列。
- 它提供了一系列 `append` 方法，用于将不同类型的数据添加到字符串中，这些方法会在现有字符序列上进行操作，而不会创建新的对象。
- `reverse` 方法用于反转字符串。`insert`、`delete`、`replace` 等方法允许在指定位置执行插入、删除、替换等操作。

StringBuffer

源码摘要:

```
public final class StringBuffer
    extends AbstractStringBuilder
    implements Serializable, Comparable<StringBuffer>, CharSequence
{
    // StringBuffer继承AbstractStringBuilder，实现Serializable接口和
CharSequence接口。

    public StringBuffer() {
        super(16); // 初始容量为16个字符
    }

    public StringBuffer(int capacity) {
        super(capacity);
    }

    public StringBuffer(String str) {
        super(str.length() + 16); // 初始容量为字符串长度 + 16个字符
        append(str);
    }

    public synchronized StringBuffer append(String str) {
        super.append(str);
        return this;
    }

```

```
    // 其他append方法用于添加字符、字符数组、其他数据类型等，都使用同步机制来确保线程安全性

    public synchronized StringBuffer reverse() {
        super.reverse();
        return this;
    }

    @Override
    public synchronized int compareTo(StringBuffer another) {
        return super.compareTo(another);
    }

    @Override
    public synchronized int length() {
        return count;
    }

    @Override
    public synchronized int capacity() {
        return super.capacity();
    }


    @Override
    public synchronized void ensureCapacity(int minimumCapacity) {
        super.ensureCapacity(minimumCapacity);
    }

    @Override
    public synchronized void trimToSize() {
        super.trimToSize();
    }

    /**
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @see        #length()
     */
    @Override
    public synchronized void setLength(int newLength) {
        toStringCache = null;
        super.setLength(newLength);
    }

    /**
     * @throws IndexOutOfBoundsException {@inheritDoc}
     * @see        #length()
     */
    @Override
    public synchronized char charAt(int index) {
        return super.charAt(index);
    }

    // 其他操作方法如insert、delete、replace等，也都使用同步机制来确保线程安全性
}
```

其父类源码：

```
abstract sealed class AbstractStringBuilder implements Appendable,
CharSequence
    permits StringBuilder, StringBuffer {
    byte[] value;
    byte coder;
    int count;
    boolean maybeLatin1;
    //...
}
```

与StringBuilder继承自同一个父类。

主要数据组织和功能实现：

- `StringBuffer` 与 `StringBuilder` 类似，内部也使用字节数组 `byte[]` 来存储字符串的字符序列。
- 它提供了一系列 `append` 方法，用于将不同类型的数据添加到字符串中，这些方法都使用同步机制来确保线程安全性。
- `reverse` 方法用于反转字符串，也使用同步机制来确保线程安全。`insert`、`delete`、`replace` 等方法都使用同步机制来确保线程安全。
- 添加了synchronized关键字（一种线程锁机制）所以是线程安全的。

> 区别：
>
> 1. **不可变性 vs. 可变性**：
>   - **String** 是不可变的，一旦创建，其内容不能被修改。每次对字符串执行操作时，都会创建一个新的字符串对象，原始字符串对象不变。这意味着对String的操作会导致内存的大量分配和释放。
>   - **StringBuilder** 和 **StringBuffer** 都是可变的。它们允许在现有字符串上进行操作，而不会创建新的对象。这在需要频繁修改字符串内容时可以提供更好的性能。
> 2. **性能**：
>   - 由于String是不可变的，它的性能受到频繁字符串操作的限制，因为每次操作都会导致新的字符串对象的创建。
>   - StringBuilder和StringBuffer是可变的，它们更适合在需要频繁修改字符串内容的情况下，因为它们不需要创建新的字符串对象，因此通常比String更高效。StringBuilder在单线程环境中通常比StringBuffer更快，因为它没有额外的同步开销。
> 3. **线程安全性**：
>   - **String** 是线程安全的，因为它是不可变的，多个线程可以安全地共享String对象。
>   - **StringBuilder** 不是线程安全的，它适合在单线程环境下使用。
>   - **StringBuffer** 是线程安全的，它使用同步机制来确保多线程环境下的安全性。但这会导致一定的性能开销。

说明为什么这样设计，这么设计对String, StringBuilder及StringBuffer的影响?

1. **String 的设计**：
   - **不可变性**：String的不可变性使其在多线程环境中更安全，因为不需要担心其他线程修改字符串内容。这是出于线程安全性的考虑。
   - **性能和缓存**：由于不可变性，相同的字符串常常可以在内存中共享，这样可以节省内存并提高性能。Java使用字符串池（String Pool）来缓存常用字符串，这可以减少重复的字符串对象的创建。

   影响：
   - 不可变性保证了线程安全性，但对于频繁的字符串操作可能会导致性能问题，因为每次操作都会创建新的字符串对象。
2. **StringBuilder 的设计**：
   - **可变性**：StringBuilder被设计成可变的，以便在需要频繁修改字符串内容的情况下提供高性能的字符串操作。
   - **性能优化**：由于可变性，StringBuilder不需要创建新的字符串对象，而是在现有对象上执行操作，这提高了性能。

   影响：
   - 在单线程环境中，StringBuilder通常比String更快，因为没有不必要的对象创建。
   - 不适合在多线程环境中，因为它不是线程安全的。
3. **StringBuffer 的设计**：
   - **可变性和线程安全性**：StringBuffer与StringBuilder类似，也是可变的，但它使用同步机制来确保线程安全性。这是为了在多线程环境中提供安全的字符串操作。
   - **性能和线程安全的权衡**：StringBuffer的设计是在性能和线程安全之间取得平衡。虽然性能可能不如StringBuilder，但它可以在多线程环境中使用，而不需要额外的同步操作。

   影响：
   - 在多线程环境中，StringBuffer是线程安全的，但性能可能比StringBuilder差。
   - 如果不需要线程安全性，建议在单线程环境中使用StringBuilder以获得更好的性能。

String, StringBuilder及StringBuffer分别适合哪些场景?

- String 适合用于不经常更改的字符串，如常量字符串、配置信息等。
- StringBuilder 适合用于需要频繁修改字符串内容的单线程环境。
- StringBuffer 适合用于需要频繁修改字符串内容的多线程环境，但请注意，它的性能可能不如 StringBuilder。如果在单线程环境下使用 StringBuffer，建议使用 StringBuilder 以提高性能。

常量池问题

```
String s1 = "Welcome to Java";

String s2 = new String("Welcome to Java");

String s3 = "Welcome to Java";

System.out.println("s1 == s2 is " + (s1 == s2));
System.out.println("s1 == s3 is " + (s1 == s3));
```

为什么s1==s2 返回false，而s1==s3返回true

解答：

这是由于Java中字符串的内存管理和字符串池（String Pool）的工作方式导致的。

1. `String s1 = "Welcome to Java";` 这行代码创建了一个字符串字面值（literal）`"Welcome to Java"`，并将 `s1` 引用指向这个字面值。在Java中，字符串字面值会被放入字符串池，以便重复使用。因此，`s1` 实际上引用的是字符串池中的对象。
2. `String s2 = new String("Welcome to Java");` 这行代码创建了一个新的String对象，通过构造函数传递了一个字符串字面值。这会在堆内存中创建一个新的字符串对象，与字符串池中的对象不同。
3. `String s3 = "Welcome to Java";` 这行代码再次创建了一个字符串字面值 `"Welcome to Java"`，并将 `s3` 引用指向字符串池中的对象，因为该字面值已存在于字符串池中。

在Java中，对象的 '==' 符号判断的是两个引用是否指向同一片内存区域。

- 对于 `s1 == s2` ，因为 `s1` 指向字符串池中的对象，而 `s2` 指向堆内存中的新对象，所以它们的引用地址不同，因此表达式返回 `false`。
- 对于 `s1 == s3` ，由于两者都指向字符串池中的相同对象（`"Welcome to Java"`），所以它们的引用地址相同，因此表达式返回 `true`。

3. 设计不变类

实现Vector, Matrix类，可以进行向量、矩阵的基本运算、可以得到（修改）Vector和Matrix中的元素，如Vector的第k维，Matrix的第i,j位的值。

Vector类

源码实现：

```
1  package Immutable;
2
3  public class Vector {
4      private double[] elements;
5
6      public Vector(int size) {
```

```
        this.elements = new double[size];
    }

    public Vector(double[] elements) {
        this.elements = elements;
    }

    // getsize
    public int getSize() {
        return elements.length;
    }

    // get方法
    public double getElement(int index) {
        if (index < 0 || index >= elements.length) {
            throw new IndexOutOfBoundsException("Invalid index");
        }
        return elements[index];
    }

    // set方法
    public void setElement(int index, double value) {
        if (index < 0 || index >= elements.length) {
            throw new IndexOutOfBoundsException("Invalid index");
        }
        elements[index] = value;
    }

    // 向量和
    public Vector add(Vector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same
size for addition.");
        }
        Vector result = new Vector(this.getSize());
        for (int i = 0; i < this.getSize(); i++) {
            result.setElement(i, this.getElement(i) + other.getElement(i));
        }
        return result;
    }

    // 向量差
    public Vector subtract(Vector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same
size for subtraction.");
        }
        Vector result = new Vector(this.getSize());
        for (int i = 0; i < this.getSize(); i++) {
            result.setElement(i, this.getElement(i) - other.getElement(i));
        }
        return result;
    }

    // 向量点积
```

```
    public double dotProduct(Vector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same size for dot product.");
        }
        double product = 0.0;
        for (int i = 0; i < this.getSize(); i++) {
            product += this.getElement(i) * other.getElement(i);
        }
        return product;
    }

}

```

该Vector类是一个用于处理向量操作的简单类。它包含了以下关键要点：

- 该类具有一个私有成员变量`elements`，用于存储向量的元素。
- 该类提供了两个构造方法，一个允许创建具有指定大小的向量，另一个允许创建包含给定元素的向量。
- `getSize()`方法返回向量的大小，即元素的个数。
- `getElement(int index)`方法用于获取向量中特定位置的元素。
- `setElement(int index, double value)`方法用于设置向量中特定位置的元素的值。
- `add(Vector other)`方法执行向量的加法操作，它会检查向量大小是否一致，然后返回一个新的向量作为结果。
- `subtract(Vector other)`方法执行向量的减法操作，类似于`add`方法，也会检查向量大小并返回新的向量作为结果。
- `dotProduct(Vector other)`方法计算两个向量的点积，同样也会检查向量大小，最后返回点积结果。

Matrix类

源码实现：

```
package Immutable;

public class Matrix {
    private int rows;
    private int columns;
    private double[][] data;

    public Matrix(int rows, int columns) {
        this.rows = rows;
        this.columns = columns;
        data = new double[rows][columns];
    }

    public Matrix(int rows, int columns, double[][] data) {
        this.rows = rows;
        this.columns = columns;
        this.data = new double[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
```

```
                this.data[i][j] = data[i][j];
            }
        }
    }

    public int getRows() {
        return rows;
    }

    public int getColumns() {
        return columns;
    }

    public double getElement(int row, int column) {
        if (row < 0 || row >= rows || column < 0 || column >= columns) {
            throw new IndexOutOfBoundsException("Invalid row or column
index");
        }
        return data[row][column];
    }

    public void setElement(int row, int column, double value) {
        if (row < 0 || row >= rows || column < 0 || column >= columns) {
            throw new IndexOutOfBoundsException("Invalid row or column
index");
        }
        data[row][column] = value;
    }

    // 矩阵加法
    public Matrix add(Matrix other) {
        if (this.getRows() != other.getRows() || this.getColumns() !=
other.getColumns()) {
            throw new IllegalArgumentException("Matrices must have the same
dimensions for addition.");
        }
        Matrix result = new Matrix(this.getRows(), this.getColumns());
        for (int i = 0; i < this.getRows(); i++) {
            for (int j = 0; j < this.getColumns(); j++) {
                result.setElement(i, j, this.getElement(i, j) +
other.getElement(i, j));
            }
        }
        return result;
    }

    // 矩阵减法
    public Matrix subtract(Matrix other) {
        if (this.getRows() != other.getRows() || this.getColumns() !=
other.getColumns()) {
            throw new IllegalArgumentException("Matrices must have the same
dimensions for addition.");
        }
        Matrix result = new Matrix(this.getRows(), this.getColumns());
        for (int i = 0; i < this.getRows(); i++) {
```

```
            for (int j = 0; j < this.getColumns(); j++) {
                result.setElement(i, j, this.getElement(i, j) -
other.getElement(i, j));
            }
        }
        return result;
    }

    // 矩阵乘法
    public Matrix multiply(Matrix other) {
        if (this.getColumns() != other.getRows()) {
            throw new IllegalArgumentException("Number of columns in the
first matrix must match the number of rows in the second matrix.");
        }
        int resultRows = this.getRows();
        int resultColumns = other.getColumns();
        Matrix result = new Matrix(resultRows, resultColumns);
        for (int i = 0; i < resultRows; i++) {
            for (int j = 0; j < resultColumns; j++) {
                double sum = 0.0;
                for (int k = 0; k < this.getColumns(); k++) {
                    sum += this.getElement(i, k) * other.getElement(k, j);
                }
                result.setElement(i, j, sum);
            }
        }
        return result;
    }

    // 矩阵转置（matrix transpose）
    public Matrix transpose() {
        int resultRows = this.getColumns();
        int resultColumns = this.getRows();
        Matrix result = new Matrix(resultRows, resultColumns);
        for (int i = 0; i < resultRows; i++) {
            for (int j = 0; j < resultColumns; j++) {
                result.setElement(i, j, this.getElement(j, i));
            }
        }
        return result;
    }

    // 矩阵数乘
    public Matrix scalarMultiply(double scalar) {
        Matrix result = new Matrix(this.getRows(), this.getColumns());
        for (int i = 0; i < this.getRows(); i++) {
            for (int j = 0; j < this.getColumns(); j++) {
                result.setElement(i, j, this.getElement(i, j) * scalar);
            }
        }
        return result;
    }
}

```

该Vector类是一个用于处理矩阵操作的类。它包含了以下关键要点：

成员变量：

- `private int rows`：用于存储矩阵的行数。
- `private int columns`：用于存储矩阵的列数。
- `private double[][] data`：一个二维数组，用于存储矩阵的元素。

构造方法：

- `public Matrix(int rows, int columns)`：构造方法，用于创建一个矩阵对象，指定行数和列数。它初始化了矩阵的维度和元素数组。

内部方法：

- `public int getRows()`：返回矩阵的行数。
- `public int getColumns()`：返回矩阵的列数。
- `public double getElement(int row, int column)`：用于获取矩阵中指定行和列位置的元素的值。如果行或列索引越界，它会抛出IndexOutOfBoundsException异常。
- `public void setElement(int row, int column, double value)`：用于设置矩阵中指定行和列位置的元素的值。如果行或列索引越界，它会抛出IndexOutOfBoundsException异常。

矩阵运算方法：

- `public Matrix add(Matrix other)`：执行矩阵加法，它会检查两个矩阵的维度是否相同，然后返回一个新的矩阵，其中每个元素是两个矩阵对应位置元素的和。
- `public Matrix subtract(Matrix other)`：执行矩阵减法，类似于add方法，也会检查两个矩阵的维度，并返回一个新的矩阵，表示两个矩阵的差。
- `public Matrix multiply(Matrix other)`：执行矩阵乘法，它会检查第一个矩阵的列数是否等于第二个矩阵的行数，然后返回一个新的矩阵，表示两个矩阵的乘积。
- `public Matrix transpose()`：计算矩阵的转置，将行和列互换，得到一个新的矩阵。
- `public Matrix scalarMultiply(double scalar)`：执行矩阵的标量乘法，将矩阵的每个元素与给定标量相乘，然后返回一个新的矩阵。

实现UnmodifiableVector, UnmodifiableMatrix不可变类

UnmodifiableVector类

源码实现：

```
1  package Immutable;
2  import java.util.Arrays;
3
4  public final class UnmodifiableVector {
5      private final double[] elements;
6
7      public UnmodifiableVector(int size) {
8          this.elements = new double[size];
9      }
10
11     // Use Arrays.copyOf to copy the input array, ensuring the immutability
   of the UnmodifiableVector class
12     public UnmodifiableVector(double[] elements) {
```

```
        this.elements = Arrays.copyOf(elements, elements.length); // Copy the array to ensure immutability
    }

    public int getSize() {
        return elements.length;
    }

    public double getElement(int index) {
        if (index < 0 || index >= elements.length) {
            throw new IndexOutOfBoundsException("Invalid index");
        }
        return elements[index];
    }

    // Generate a copy of the array
    public double[] getElements() {
        return Arrays.copyOf(elements, elements.length);
    }

    public UnmodifiableVector setElement(int index, double value) {
        if (index < 0 || index >= elements.length) {
            throw new IndexOutOfBoundsException("Invalid index");
        }
        double[] newElements = Arrays.copyOf(elements, elements.length);
        newElements[index] = value;
        return new UnmodifiableVector(newElements);
    }

    // Vector addition
    public UnmodifiableVector add(UnmodifiableVector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same size for addition.");
        }
        double[] resultElements = new double[this.getSize()];
        for (int i = 0; i < this.getSize(); i++) {
            resultElements[i] = this.getElement(i) + other.getElement(i);
        }
        return new UnmodifiableVector(resultElements);
    }

    // Vector subtraction
    public UnmodifiableVector subtract(UnmodifiableVector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same size for subtraction.");
        }
        double[] resultElements = new double[this.getSize()];
        for (int i = 0; i < this.getSize(); i++) {
            resultElements[i] = this.getElement(i) - other.getElement(i);
        }
        return new UnmodifiableVector(resultElements);
    }

```

```
    // Vector dot product
    public double dotProduct(UnmodifiableVector other) {
        if (this.getSize() != other.getSize()) {
            throw new IllegalArgumentException("Vectors must have the same size for dot product.");
        }
        double product = 0.0;
        for (int i = 0; i < this.getSize(); i++) {
            product += this.getElement(i) * other.getElement(i);
        }
        return product;
    }
}

```

该类是Vector类的不可变类，为了实现它的不可变性，做了如下修改：

- 将Class设置为final类型，使它不能被继承
- 将成员变量都设计为private fina类型，使它们不能被修改
- 删除了可以修改内置成员变量的方法，所有set方法都是返回一个新的对象而不是直接对原对象进行更改。
- 通过在第二个构造函数中使用`Arrays.copyOf`来复制传入的数组，确保了UnmodifiableVector类的不可变性。
- 对外开放的接口方法都不对内置成员变量进行修改，而是返回一个新的对象。

UnmodifiableMatrix类

源码实现：

```
public final class UnmodifiableMatrix {
    private final int rows;
    private final int columns;
    private final double[][] data;

    public UnmodifiableMatrix(int rows, int columns, double[][] data) {
        this.rows = rows;
        this.columns = columns;
        this.data = new double[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                this.data[i][j] = data[i][j];
            }
        }
    }

    public int getRows() {
        return rows;
    }

    public int getColumns() {
        return columns;
    }

```

```
    public double getElement(int row, int column) {
        if (row < 0 || row >= rows || column < 0 || column >= columns) {
            throw new IndexOutOfBoundsException("Invalid row or column
index");
        }
        return data[row][column];
    }

    public UnmodifiableMatrix setElement(int row, int column, double value)
{
        if (row < 0 || row >= rows || column < 0 || column >= columns) {
            throw new IndexOutOfBoundsException("Invalid row or column
index");
        }
        double[][] newData = new double[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                newData[i][j] = this.data[i][j];
            }
        }
        newData[row][column] = value;
        return new UnmodifiableMatrix(rows, columns, newData);
    }

    // Matrix addition
    public UnmodifiableMatrix add(UnmodifiableMatrix other) {
        if (this.getRows() != other.getRows() || this.getColumns() !=
other.getColumns()) {
            throw new IllegalArgumentException("Matrices must have the same
dimensions for addition.");
        }
        double[][] resultData = new double[this.getRows()]
[this.getColumns()];
        for (int i = 0; i < this.getRows(); i++) {
            for (int j = 0; j < this.getColumns(); j++) {
                resultData[i][j] = this.getElement(i, j) +
other.getElement(i, j);
            }
        }
        return new UnmodifiableMatrix(this.getRows(), this.getColumns(),
resultData);
    }

    // Matrix subtract
    public UnmodifiableMatrix subtract(UnmodifiableMatrix other) {
        if (this.getRows() != other.getRows() || this.getColumns() !=
other.getColumns()) {
            throw new IllegalArgumentException("Matrices must have the same
dimensions for subtraction.");
        }
        double[][] resultData = new double[this.getRows()]
[this.getColumns()];
        for (int i = 0; i < this.getRows(); i++) {
            for (int j = 0; j < this.getColumns(); j++) {
```

```
                resultData[i][j] = this.getElement(i, j) -
other.getElement(i, j);
            }
        }
        return new UnmodifiableMatrix(this.getRows(), this.getColumns(),
resultData);
    }

    // Matrix multiply
    public UnmodifiableMatrix multiply(UnmodifiableMatrix other) {
        if (this.getColumns() != other.getRows()) {
            throw new IllegalArgumentException("Number of columns in the
first matrix must match the number of rows in the second matrix.");
        }
        int resultRows = this.getRows();
        int resultColumns = other.getColumns();
        double[][] resultData = new double[resultRows][resultColumns];
        for (int i = 0; i < resultRows; i++) {
            for (int j = 0; j < resultColumns; j++) {
                double sum = 0.0;
                for (int k = 0; k < this.getColumns(); k++) {
                    sum += this.getElement(i, k) * other.getElement(k, j);
                }
                resultData[i][j] = sum;
            }
        }
        return new UnmodifiableMatrix(resultRows, resultColumns,
resultData);
    }

    // Matrix transpose
    public UnmodifiableMatrix transpose() {
        int resultRows = this.getColumns();
        int resultColumns = this.getRows();
        double[][] resultData = new double[resultRows][resultColumns];
        for (int i = 0; i < resultRows; i++) {
            for (int j = 0; j < resultColumns; j++) {
                resultData[i][j] = this.getElement(j, i);
            }
        }
        return new UnmodifiableMatrix(resultRows, resultColumns,
resultData);
    }

    // Matrix scalarMultiply
    public UnmodifiableMatrix scalarMultiply(double scalar) {
        double[][] resultData = new double[this.getRows()]
[this.getColumns()];
        for (int i = 0; i < this.getRows(); i++) {
            for (int j = 0; j < this.getColumns(); j++) {
                resultData[i][j] = this.getElement(i, j) * scalar;
            }
        }
        return new UnmodifiableMatrix(this.getRows(), this.getColumns(),
resultData);
```

```
    }
}

```

该类是Matrix类的不可变类，为了实现它的不可变性，做了如下修改：

- 将Class设置为final类型，使它不能被继承
- 将成员变量都设计为private fina类型，使它们不能被修改
- 删除了可以修改内置成员变量的方法，所有set方法都是返回一个新的对象而不是直接对原对象进行更改。
- 修改构造方法，对于传入构造函数中的可变参数通过new一个副本来接受。
- 对外开放的接口方法都不对内置成员变量进行修改，而是返回一个新的对象。

实现MathUtils，含有静态方法

• UnmodifiableVector getUnmodifiableVector(Vector v)

• UnmodifiableMatrix getUnmodifiableMatrix(Matrix m)

源码实现：

```java
package Immutable;

public class MathUtils {
    public static UnmodifiableVector getUnmodifiableVector(Vector v) {
        double[] elements = new double[v.getSize()];
        for (int i = 0; i < v.getSize(); i++) {
            elements[i] = v.getElement(i);
        }
        return new UnmodifiableVector(elements);
    }

    public static UnmodifiableMatrix getUnmodifiableMatrix(Matrix m) {
        int rows = m.getRows();
        int columns = m.getColumns();
        double[][] data = new double[rows][columns];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                data[i][j] = m.getElement(i, j);
            }
        }
        return new UnmodifiableMatrix(rows, columns, data);
    }
}

```

这些方法将 `Vector` 和 `Matrix` 对象转换为 `UnmodifiableVector` 和 `UnmodifiableMatrix` 对象，以确保它们不可变。

测试说明

为了测试方便，封装了一个`Test`类，其中提供了测试四个类（Vector、Matrix、UnmodifiableVector、UnmodifiableMatrix）的方法。

源码实现：

```
package Immutable;

public class Test {

    public static void main(String[] args) {
        testVectorOperations();
        testMatrixOperations();
        testUnmodifiableVector();
        testUnmodifiableMatrix();
    }

    public static void testVectorOperations() {
        Vector v1 = new Vector(new double[]{1.0, 2.0, 3.0});
        Vector v2 = new Vector(new double[]{4.0, 5.0, 6.0});

        // Test vector addition
        Vector resultAdd = v1.add(v2);
        System.out.println("Vector Addition Result:");
        printVector(resultAdd);

        // Test vector subtraction
        Vector resultSubtract = v1.subtract(v2);
        System.out.println("Vector Subtraction Result:");
        printVector(resultSubtract);

        // Test vector dot product
        double dotProduct = v1.dotProduct(v2);
        System.out.println("Vector Dot Product Result: " + dotProduct);
    }

    public static void testMatrixOperations() {
        Matrix m1 = new Matrix(2, 2);
        m1.setElement(0, 0, 1.0);
        m1.setElement(0, 1, 2.0);
        m1.setElement(1, 0, 3.0);
        m1.setElement(1, 1, 4.0);

        Matrix m2 = new Matrix(2, 2);
        m2.setElement(0, 0, 5.0);
        m2.setElement(0, 1, 6.0);
        m2.setElement(1, 0, 7.0);
        m2.setElement(1, 1, 8.0);

        // Test matrix addition
        Matrix resultAdd = m1.add(m2);
        System.out.println("Matrix Addition Result:");
        printMatrix(resultAdd);

```

```
        // Test matrix subtraction
        Matrix resultSubtract = m1.subtract(m2);
        System.out.println("Matrix Subtraction Result:");
        printMatrix(resultSubtract);

        // Test matrix multiplication
        Matrix resultMultiply = m1.multiply(m2);
        System.out.println("Matrix Multiplication Result:");
        printMatrix(resultMultiply);

        // Test matrix transpose
        Matrix resultTranspose = m1.transpose();
        System.out.println("Matrix Transpose Result:");
        printMatrix(resultTranspose);

        // Test matrix scalar multiplication
        Matrix resultScalarMultiply = m1.scalarMultiply(2.0);
        System.out.println("Matrix Scalar Multiplication Result:");
        printMatrix(resultScalarMultiply);
    }

    public static void testUnmodifiableVector() {
        double[] elements = { 1.0, 2.0, 3.0 };
        Vector vector = new Vector(elements);
        UnmodifiableVector unmodifiableVector =
MathUtils.getUnmodifiableVector(vector);


        // Test getElement method
        double element = unmodifiableVector.getElement(1);
        System.out.println("Element at index 1: " + element);

        // Test setElement method
        UnmodifiableVector modifiedVector =
unmodifiableVector.setElement(1, 5.0);
        System.out.println("Original vector: ");
        printVector(unmodifiableVector);
        System.out.println("Modified vector: ");
        printVector(modifiedVector);

        // Verify that the original vector remains unchanged
        System.out.println("Original vector after modification: ");
        printVector(unmodifiableVector);
    }

    public static void testUnmodifiableMatrix() {
        double[][] data = {{1.0, 2.0}, {3.0, 4.0}};
        Matrix matrix = new Matrix(2, 2, data);
        UnmodifiableMatrix unmodifiableMatrix =
MathUtils.getUnmodifiableMatrix(matrix);

        // Test getElement method
        double element = unmodifiableMatrix.getElement(1, 1);
        System.out.println("Element at (1, 1): " + element);

```

```
        // Test setElement method
        UnmodifiableMatrix modifiedMatrix =
unmodifiableMatrix.setElement(1, 1, 5.0);
        System.out.println("Original matrix:");
        printMatrix(unmodifiableMatrix);
        System.out.println("Modified matrix:");
        printMatrix(modifiedMatrix);

        // Verify that the original matrix remains unchanged
        System.out.println("Original matrix after modification:");
        printMatrix(unmodifiableMatrix);
    }

    // Printing function for the Vector class
    public static void printVector(Vector vector) {
        int size = vector.getSize();
        System.out.print("[");
        for (int i = 0; i < size; i++) {
            System.out.print(vector.getElement(i));
            if (i < size - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

    // Printing function for the UnmodifiableVector class
    public static void printVector(UnmodifiableVector vector) {
        int size = vector.getSize();
        System.out.print("[");
        for (int i = 0; i < size; i++) {
            System.out.print(vector.getElement(i));
            if (i < size - 1) {
                System.out.print(", ");
            }
        }
        System.out.println("]");
    }

    // Printing function for the Matrix class
    public static void printMatrix(Matrix matrix) {
        int rows = matrix.getRows();
        int columns = matrix.getColumns();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                System.out.print(matrix.getElement(i, j) + " ");
            }
            System.out.println();
        }
    }

    // Printing function for the UnmodifiableMatrix class
    public static void printMatrix(UnmodifiableMatrix matrix) {
        int rows = matrix.getRows();
        int columns = matrix.getColumns();
```

```
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < columns; j++) {
                System.out.print(matrix.getElement(i, j) + " ");
            }
            System.out.println();
        }
    }
}

```

测试结果如下：

对vector类的测试结果：

```
Vector Addition Result:
[5.0, 7.0, 9.0]
Vector Subtraction Result:
[-3.0, -3.0, -3.0]
Vector Dot Product Result: 32.0
```

对Matrix类的测试结果：

```
Matrix Addition Result:
6.0 8.0
10.0 12.0
Matrix Subtraction Result:
-4.0 -4.0
-4.0 -4.0
Matrix Multiplication Result:
19.0 22.0
43.0 50.0
Matrix Transpose Result:
1.0 3.0
2.0 4.0
Matrix Scalar Multiplication Result:
2.0 4.0
6.0 8.0
```

对UnmodifiableVector类的测试结果：

```
Element at index 1: 2.0
Original vector: [1.0, 2.0, 3.0]
Modified vector: [1.0, 5.0, 3.0]
Original vector after modification: [1.0, 2.0, 3.0]
```

对UnmodifiableMatrix类的测试结果：

```
Element at (1, 1): 4.0
Original matrix:
1.0 2.0
3.0 4.0
Modified matrix:
1.0 2.0
3.0 5.0
Original matrix after modification:
1.0 2.0
3.0 4.0
```

其中由于在 `testUnmodifiableMatrix` 和 `testUnmodifiableVector` 这两个测试方法中调用了 `MathUtils` 中的 `getUnmodifiableVector` 和 `getUnmodifiableMatrix` 方法，所以这也变相测试了 `MathUtils` 类。