Homework 3

1. JDK 库中的不变类

1. 寻找 JDK 库中的不变类（至少3类），并进行源码分析，分析其为什么是不变的？文档说明其共性。

在 Java 的 JDK 库中，不变类（Immutable Classes）是指那些一旦创建其实例之后，其状态（即其属性）就不能被更改的类。这些类的设计理念是为了提高代码的可读性和运行时的安全性

不变类示例

`String`

`String` 类在 Java 中代表字符串。它的不变性体现在，一旦一个 `String` 对象被创建，其内容就不能被改变。任何对字符串内容的修改操作都会导致创建一个新的 `String` 对象，而不是更改现有对象。这种设计是为了优化性能（通过字符串常量池）和提高安全性（字符串被广泛用作参数和系统属性）。

JDK 库的 `String` 类源码部分如下：

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {

    @Stable
    private final byte[] value;

    private final byte coder;

    private int hash; // Default to 0

    private boolean hashIsZero; // Default to false;

    /** use serialVersionUID from JDK 1.0.2 for interoperability */
    @java.io.Serial
    private static final long serialVersionUID = -6849794470754667710L;

    static final boolean COMPACT_STRINGS;

    static {
        COMPACT_STRINGS = true;
    }

    @java.io.Serial
    private static final ObjectStreamField[] serialPersistentFields =
        new ObjectStreamField[0];

    // ...

    public String(byte[] ascii, int hibyte, int offset, int count) {
        checkBoundsOffCount(offset, count, ascii.length);
        if (count == 0) {
            this.value = "".value;
            this.coder = "".coder;
```

```
            return;
        }
        if (COMPACT_STRINGS && (byte)hibyte == 0) {
            this.value = Arrays.copyOfRange(ascii, offset, offset + count);
            this.coder = LATIN1;
        } else {
            hibyte <<= 8;
            byte[] val = StringUTF16.newBytesFor(count);
            for (int i = 0; i < count; i++) {
                StringUTF16.putChar(val, i, hibyte | (ascii[offset++] & 0xff));
            }
            this.value = val;
            this.coder = UTF16;
        }
    }

    // ...

    public String substring(int beginIndex, int endIndex) {
        int length = length();
        checkBoundsBeginEnd(beginIndex, endIndex, length);
        if (beginIndex == 0 && endIndex == length) {
            return this;
        }
        int subLen = endIndex - beginIndex;
        return isLatin1() ? StringLatin1.newString(value, beginIndex, subLen)
                          : StringUTF16.newString(value, beginIndex, subLen);
    }

    // ...

}
```

从源码分析可以得知，`String` 类被 `final` 修饰，这意味着它不能被继承。同时，`String` 类的所有字段，例如 `value`、`coder`、`hash` 等都被 `final` 修饰，这意味着它们的值不能被更改。`String` 类的所有方法都不会改变 `String` 对象的状态，而是返回一个新的 `String` 对象。在通过 `byte[] ascii` 等可变对象构造 `String` 对象时，`String` 类会通过 `Arrays.copyOfRange` 等方法复制一份新的对象，而不是直接引用传入的对象。这些设计保证了 `String` 对象的不变性。

这样的设计使得 `String` 对象在多线程环境下是安全的，因为它的状态不会被更改。同时，`String` 对象的不变性也使得它可以被广泛用作参数和系统属性，而不用担心它的值会被更改。

`Integer`

`Integeer` 是一种基本数据类型的包装器。它的不变性与 `String` 类似，即一旦一个包装对象被创建，其内部的值就不能改变。任何修改操作都会返回一个新的对象。这样的设计有助于缓存常用的实例（例如小的整数），减少内存使用，并保证了线程安全。

JDK 库的 `Integer` 类源码部分如下：

```
@jdk.internal.ValueBased
public final class Integer extends Number
        implements Comparable<Integer>, Constable, ConstantDesc {

    // ...

    @IntrinsicCandidate
    public static Integer valueOf(int i) {
```

```
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }

    private final int value;

    // ...

}
```

从源码分析可以得知，`Integer` 类被 `final` 修饰，这意味着它不能被继承。同时，`Integer` 类的所有字段，例如 `value` 等都被 `final` 修饰，这意味着它们的值不能被更改。`Integer` 类的所有方法都不会改变 `Integer` 对象的状态，而是返回一个新的 `Integer` 对象。在通过 `int i` 等可变对象构造 `Integer` 对象时，`Integer` 类会通过 `new Integer(i)` 等方法复制一份新的对象，而不是直接引用传入的对象。这些设计保证了 `Integer` 对象的不变性。

BigInteger

`BigInteger` 类用于精确的数学计算。它的设计也遵循了不变性原则，即所有的算术操作都不会更改现有对象的状态，而是产生一个新的对象。这样做既保证了数学运算的准确性，又避免了并发环境下的问题。

```
public class BigInteger extends Number implements Comparable<BigInteger> {

    final int signum;

    final int[] mag;

    // ...

    public BigInteger(byte[] val, int off, int len) {
        if (val.length == 0) {
            throw new NumberFormatException("Zero length BigInteger");
        }
        Objects.checkFromIndexSize(off, len, val.length);

        if (val[off] < 0) {
            mag = makePositive(val, off, len);
            signum = -1;
        } else {
            mag = stripLeadingZeroBytes(val, off, len);
            signum = (mag.length == 0 ? 0 : 1);
        }
        if (mag.length >= MAX_MAG_LENGTH) {
            checkRange();
        }
    }

    private static int[] makePositive(byte[] a, int off, int len) {
        int keep, k;
        int indexBound = off + len;

        // Find first non-sign (0xff) byte of input
        for (keep=off; keep < indexBound && a[keep] == -1; keep++)
            ;


        /* Allocate output array.  If all non-sign bytes are 0x00, we must
```

```
         * allocate space for one extra output byte. */
        for (k=keep; k < indexBound && a[k] == 0; k++)
            ;

        int extraByte = (k == indexBound) ? 1 : 0;
        int intLength = ((indexBound - keep + extraByte) + 3) >>> 2;
        int result[] = new int[intLength];

        /* Copy one's complement of input into output, leaving extra
         * byte (if it exists) == 0x00 */
        int b = indexBound - 1;
        for (int i = intLength-1; i >= 0; i--) {
            result[i] = a[b--] & 0xff;
            int numBytesToTransfer = Math.min(3, b-keep+1);
            if (numBytesToTransfer < 0)
                numBytesToTransfer = 0;
            for (int j=8; j <= 8*numBytesToTransfer; j += 8)
                result[i] |= ((a[b--] & 0xff) << j);

            // Mask indicates which bits must be complemented
            int mask = -1 >>> (8*(3-numBytesToTransfer));
            result[i] = ~result[i] & mask;
        }

        // Add one to one's complement to generate two's complement
        for (int i=result.length-1; i >= 0; i--) {
            result[i] = (int)((result[i] & LONG_MASK) + 1);
            if (result[i] != 0)
                break;
        }

        return result;
    }

    // ...

}
```

从源码分析可以得知，`BigInteger` 类的所有字段，例如 `signum`、`mag` 等都被 `final` 修饰，这意味着它们的值不能被更改。`BigInteger` 类的所有方法都不会改变 `BigInteger` 对象的状态，而是返回一个新的 `BigInteger` 对象。在通过 `byte[] val` 等可变对象构造 `BigInteger` 对象时，`BigInteger` 类会通过 `makePositive` 等方法复制一份新的对象，而不是直接引用传入的对象。这些设计保证了 `BigInteger` 对象的不变性。

不变类的共性

- 所有字段都是 `final` 的，这意味着它们只能被赋值一次。
- 通常类本身被声明为 `final`，因此不能被继承。
- 不存在修改对象状态的方法。
- 如果类具有可变对象的字段，则必须通过深拷贝来防止外部修改。
- 安全性：不变对象在多线程环境下使用时，无需担心数据竞争或者同步问题。
- 缓存和重用：由于状态不变，这些对象的实例往往可以被安全地缓存和重用。
- 哈希表的键：由于其状态不变，这些对象特别适合用作哈希表的键。
- 创建和使用简单：不变对象通常更易于理解和使用。

`String` `StringBuilder` 与 `StringBuffer`

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

主要数据组织及功能实现

2.1. 分析其主要数据组织及功能实现，有什么区别?

`String`

- 主要数据组织：`String` 类的主要数据组织是 `byte[]` 数组，用于存储字符串的内容，其在源码中定义为 `private final byte[] value;`。`String` 类的其他字段包括 `coder`、`hash` 等，用于存储字符串的编码方式和哈希值。
- 功能实现：`String` 类的主要功能实现是字符串的操作，例如字符串的拼接、截取、查找、替换等。
- 主要区别：`String` 类的主要区别在于其不变性，即一旦一个 `String` 对象被创建，其内容就不能被改变。任何对字符串内容的修改操作都会导致创建一个新的 `String` 对象，而不是更改现有对象。

`StringBuilder`

- 主要数据组织：`StringBuilder` 继承自 `AbstractStringBuilder`，其主要数据组织是 `byte[]` 数组，用于存储字符串的内容，其在源码中定义为 `byte[] value;`，同时还有 `int count` 用于记录字符串的长度。源码部分截取如下：

```
abstract sealed class AbstractStringBuilder implements Appendable, CharSequence
    permits StringBuilder, StringBuffer {

    byte[] value;

    byte coder;

    boolean maybeLatin1;

    int count;

    private static final byte[] EMPTYVALUE = new byte[0];

    AbstractStringBuilder() {
        value = EMPTYVALUE;
    }

    AbstractStringBuilder(int capacity) {
        if (COMPACT_STRINGS) {
            value = new byte[capacity];
            coder = LATIN1;
        } else {
            value = StringUTF16.newBytesFor(capacity);
            coder = UTF16;
        }
    }

    AbstractStringBuilder(String str) {
        int length = str.length();
        int capacity = (length < Integer.MAX_VALUE - 16)
                ? length + 16 : Integer.MAX_VALUE;
        final byte initCoder = str.coder();
        coder = initCoder;
        value = (initCoder == LATIN1)
                ? new byte[capacity] : StringUTF16.newBytesFor(capacity);
```

```
        append(str);
    }

    // ...
}
```

- 功能实现：`StringBuilder` 类的主要功能实现是字符串的操作，例如字符串的拼接、截取、查找、替换等。他提供了高效的字符串操作方法，例如 `append`、`insert`、`delete`、`replace` 等，并且会在内部预留一定的空间，以便后续的字符串拼接操作。
- 主要区别：`StringBuilder` 类的主要区别在于其可变性，即其内部的字符串内容可以被修改。`StringBuilder` 类的所有方法都会改变 `StringBuilder` 对象的状态，而不是返回一个新的 `StringBuilder` 对象。

`StringBuffer`

- 主要数据组织：`StringBuffer` 同样继承自 `AbstractStringBuilder`，其主要数据组织是 `byte[]` 数组，用于存储字符串的内容，其在源码中定义为 `byte[] value;`，同时还有 `int count` 用于记录字符串的长度。
- 功能实现：`StringBuffer` 类的主要功能实现是字符串的操作，例如字符串的拼接、截取、查找、替换等。他同样提供了高效的字符串操作方法，例如 `append`、`insert`、`delete`、`replace` 等。并且 `StringBuffer` 的所有方法都是线程同步的，被 `synchronized` 修饰，因此在多线程环境下使用时，不需要额外的同步操作。
- 主要区别：`StringBuffer` 类的主要区别在于其可变性与线程安全性。`StringBuffer` 类的所有方法都会改变 `StringBuffer` 对象的状态，而不是返回一个新的 `StringBuffer` 对象。同时，`StringBuffer` 类的所有方法都是线程同步的。

设计原因与影响

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

`String`

`String` 类的设计遵循了不变性原则，即一旦一个 `String` 对象被创建，其内容就不能被改变。任何对字符串内容的修改操作都会导致创建一个新的 `String` 对象，而不是更改现有对象。这样的设计使得 `String` 对象在多线程环境下是安全的，因为它的状态不会被更改。同时，`String` 对象的不变性也使得它可以被广泛用作参数和系统属性，而不用担心它的值会被更改。这确保了字符串对象的安全性和共享性。

这样的设计导致在需要频繁修改字符串内容的情况下，由于每次修改都会创建新的 `String` 对象，可能会导致性能问题。

`StringBuilder`

`StringBuilder` 被设计为可变的，它使用一个可变的字符数组来存储字符串内容。这使得它在频繁修改字符串时更高效，因为它不需要每次都创建新的对象。它不是线程安全的，但性能更高。

这样的设计导致在多线程环境下使用 `StringBuilder` 时，需要额外的同步操作，否则可能会导致数据竞争和同步问题。

`StringBuffer`

`StringBuffer` 与 `StringBuilder` 的设计类似，也是可变的，但它是线程安全的，多个线程可以安全地同时访问和修改 `StringBuffer` 的内容。

这样的设计导致在单线程环境下使用 `StringBuffer` 时，由于需要额外的同步操作，可能会导致性能问题。在单线程环境下，`StringBuilder` 的性能更高。

适用场景

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

- `String` 适用于表示不可变的字符串，即字符串内容不会被修改的情况。例如字符串常量、字符串参数、系统属性、安全传递等。在 Java 中，用 `""` 表示字符串常量时，实际上是创建了一个 `String` 对象。

- `StringBuilder` 适用于频繁修改字符串内容的情况。例如字符串拼接、字符串替换等。例如动态字符串拼接、构建复杂格式的字符串等。
- `StringBuffer` 适用于多线程环境下频繁修改字符串内容的情况。例如多线程环境下的字符串拼接、字符串替换等。

示例

```
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`

- `s1` 和 `s3` 都是字符串常量，它们都是指向字符串常量池中的同一个对象。
- `s2` 是通过 `new` 关键字创建的字符串对象，它指向堆中的一个新的对象。
- `==` 操作符比较的是两个对象的引用，因此 `s1 == s2` 返回 `false`，而 `s1 == s3` 返回 `true`。

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` 类设计如下：

```
import java.util.Arrays;

public class Vector {
    private double[] data;

    public Vector(int size) {
        data = new double[size];
    }

    public Vector(double[] data) {
        // make a copy of the data
        this.data = Arrays.copyOf(data, data.length);
    }

    public Vector(Vector v) {
        this(v.data);
    }

    // get the size of the vector
    public int size() {
        return data.length;
    }

    // get the i-th element
    public double get(int i) {
        return data[i];
    }
```

```java
// set the i-th element
public void set(int i, double value) {
    data[i] = value;
}

// add two vectors
public Vector add(Vector v) {
    if (size() != v.size()) {
        throw new IllegalArgumentException("Vector sizes do not match");
    } else {
        Vector result = new Vector(size());
        for (int i = 0; i < size(); i++) {
            result.set(i, get(i) + v.get(i));
        }
        return result;
    }
}

// subtract two vectors
public Vector sub(Vector v) {
    if (size() != v.size()) {
        throw new IllegalArgumentException("Vector sizes do not match");
    } else {
        Vector result = new Vector(size());
        for (int i = 0; i < size(); i++) {
            result.set(i, get(i) - v.get(i));
        }
        return result;
    }
}

// multiply a vector by a scalar
public Vector mul(double scalar) {
    Vector result = new Vector(size());
    for (int i = 0; i < size(); i++) {
        result.set(i, get(i) * scalar);
    }
    return result;
}

// compute the dot product of two vectors
public double dot(Vector v) {
    if (size() != v.size()) {
        throw new IllegalArgumentException("Vector sizes do not match");
    } else {
        double result = 0;
        for (int i = 0; i < size(); i++) {
            result += get(i) * v.get(i);
        }
        return result;
    }
}

@Override
public String toString() {
    return Arrays.toString(data);
```

```
    }
}
```

`Matrix` 类设计如下：

```
import java.util.Arrays;

public class Matrix {
    private double[][] data;

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

    public Matrix(double[][] data) {
        this.data = new double[data.length][];
        int cols = data[0].length;
        for (int i = 0; i < data.length; i++) {
            this.data[i] = Arrays.copyOf(data[i], data[i].length);
            if (this.data[i].length != cols) {
                throw new IllegalArgumentException("Matrix rows have different lengths");
            }
        }
    }

    public Matrix(Matrix m) {
        this(m.data);
    }

    // get the number of rows
    public int rows() {
        return data.length;
    }

    // get the number of columns
    public int cols() {
        return data[0].length;
    }

    // get the (i, j)-th element
    public double get(int i, int j) {
        return data[i][j];
    }

    // set the (i, j)-th element
    public void set(int i, int j, double value) {
        data[i][j] = value;
    }

    // add two matrices
    public Matrix add(Matrix m) {
        if (rows() != m.rows() || cols() != m.cols()) {
            throw new IllegalArgumentException("Matrix sizes do not match");
        } else {
            Matrix result = new Matrix(rows(), cols());
            for (int i = 0; i < rows(); i++) {
                for (int j = 0; j < cols(); j++) {
```

```
                result.set(i, j, get(i, j) + m.get(i, j));
            }
        }
        return result;
    }
}

// subtract two matrices
public Matrix sub(Matrix m) {
    if (rows() != m.rows() || cols() != m.cols()) {
        throw new IllegalArgumentException("Matrix sizes do not match");
    } else {
        Matrix result = new Matrix(rows(), cols());
        for (int i = 0; i < rows(); i++) {
            for (int j = 0; j < cols(); j++) {
                result.set(i, j, get(i, j) - m.get(i, j));
            }
        }
        return result;
    }
}

// multiply a matrix by a scalar
public Matrix mul(double scalar) {
    Matrix result = new Matrix(rows(), cols());
    for (int i = 0; i < rows(); i++) {
        for (int j = 0; j < cols(); j++) {
            result.set(i, j, get(i, j) * scalar);
        }
    }
    return result;
}

// compute the dot product of two matrices
public Matrix mul(Matrix m) {
    if (cols() != m.rows()) {
        throw new IllegalArgumentException("Matrix sizes do not match");
    } else {
        Matrix result = new Matrix(rows(), m.cols());
        for (int i = 0; i < rows(); i++) {
            for (int j = 0; j < m.cols(); j++) {
                double sum = 0;
                for (int k = 0; k < cols(); k++) {
                    sum += get(i, k) * m.get(k, j);
                }
                result.set(i, j, sum);
            }
        }
        return result;
    }
}

// compute the transpose of a matrix
public Matrix transpose() {
    Matrix result = new Matrix(cols(), rows());
    for (int i = 0; i < rows(); i++) {
        for (int j = 0; j < cols(); j++) {
            result.set(j, i, get(i, j));
```

```
            }
        }
        return result;
    }

    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("[");
        for (int i = 0; i < rows(); i++) {
            sb.append("[");
            for (int j = 0; j < cols(); j++) {
                sb.append(get(i, j));
                if (j < cols() - 1) {
                    sb.append(", ");
                }
            }
            sb.append("]");
            if (i < rows() - 1) {
                sb.append(",\n");
            }
        }
        sb.append("]");
        return sb.toString();
    }
}
```

可以看到，我们的 `Vector` 和 `Matrix` 类都是可变的，具有 `get` 和 `set` 方法，可以获取和修改其中的元素。

- 实现 `UnmodifiableVector`，`UnmodifiableMatrix` 不可变类

`UnmodifiableVector` 类设计如下：

```
public final class UnmodifiableVector {
    private final Vector vector;    // final to make it immutable

    public UnmodifiableVector(int size) {
        vector = new Vector(size);
    }

    public UnmodifiableVector(double[] data) {
        // make a copy of the data
        vector = new Vector(data);
    }

    public UnmodifiableVector(Vector vector) {
        // make a copy of the vector to make it immutable
        this.vector = new Vector(vector);
    }

    public UnmodifiableVector(UnmodifiableVector v) {
        // no need to make a copy of the vector since it is already immutable
        this.vector = v.vector;
    }

    // get the size of the vector
    public int size() {
        return vector.size();
```

```
    }

    // get the i-th element
    public double get(int i) {
        return vector.get(i);
    }

    // No set method

    // add two vectors
    public UnmodifiableVector add(UnmodifiableVector v) {
        return new UnmodifiableVector(vector.add(v.vector));
    }

    // subtract two vectors
    public UnmodifiableVector sub(UnmodifiableVector v) {
        return new UnmodifiableVector(vector.sub(v.vector));
    }

    // multiply a vector by a scalar
    public UnmodifiableVector mul(double scalar) {
        return new UnmodifiableVector(vector.mul(scalar));
    }

    // dot product of two vectors
    public double dot(UnmodifiableVector v) {
        return vector.dot(v.vector);
    }

    @Override
    public String toString() {
        return vector.toString();
    }
}
```

`UnmodifiableMatrix` 类设计如下：

```
final public class UnmodifiableMatrix {
    private final Matrix matrix;    // final to make it immutable

    public UnmodifiableMatrix(int rows, int cols) {
        matrix = new Matrix(rows, cols);
    }

    public UnmodifiableMatrix(double[][] data) {
        // make a copy of the data
        matrix = new Matrix(data);
    }

    public UnmodifiableMatrix(Matrix matrix) {
        // make a copy of the matrix to make it immutable
        this.matrix = new Matrix(matrix);
    }

    public UnmodifiableMatrix(UnmodifiableMatrix m) {
        // no need to make a copy of the matrix since it is already immutable
        this.matrix = m.matrix;
```

```
    }

    // get the number of rows
    public int rows() {
        return matrix.rows();
    }

    // get the number of columns
    public int cols() {
        return matrix.cols();
    }

    // get the (i, j)-th element
    public double get(int i, int j) {
        return matrix.get(i, j);
    }

    // No set method

    // add two matrices
    public UnmodifiableMatrix add(UnmodifiableMatrix m) {
        return new UnmodifiableMatrix(matrix.add(m.matrix));
    }

    // subtract two matrices
    public UnmodifiableMatrix sub(UnmodifiableMatrix m) {
        return new UnmodifiableMatrix(matrix.sub(m.matrix));
    }

    // multiply a matrix by a scalar
    public UnmodifiableMatrix mul(double scalar) {
        return new UnmodifiableMatrix(matrix.mul(scalar));
    }

    // multiply two matrices
    public UnmodifiableMatrix mul(UnmodifiableMatrix m) {
        return new UnmodifiableMatrix(matrix.mul(m.matrix));
    }

    // get the transpose of the matrix
    public UnmodifiableMatrix transpose() {
        return new UnmodifiableMatrix(matrix.transpose());
    }

    @Override
    public String toString() {
        return matrix.toString();
    }
}
```

可以看到，`UnmodifiableVector` 和 `UnmodifiableMatrix` 类都是不可变的，它们的 `get` 方法可以获取其中的元素，但没有 `set` 方法，因此其中的元素是不可修改的。并且，它们的所有方法都会返回一个新的 `UnmodifiableVector` 或 `UnmodifiableMatrix` 对象，而不是修改现有对象。

- 实现 `MathUtils`，含有静态方法
  - `UnmodifiableVector getUnmodifiableVector(Vector v)`
  - `UnmodifiableMatrix getUnmodifiableMatrix(Matrix m)`

`MathUtils` 类设计如下：

```
public class MathUtils {
    private MathUtils() {
        // private constructor to prevent instantiation
    }

    public static UnmodifiableVector getUnmodifiableVector(Vector v) {
        return new UnmodifiableVector(v);
    }

    public static UnmodifiableMatrix getUnmodifiableMatrix(Matrix m) {
        return new UnmodifiableMatrix(m);
    }
}
```

测试代码如下：

```
public class Test {

    public static void main(String[] args) {
        Vector v1 = new Vector(new double[] {1, 2, 3});
        Vector v2 = new Vector(3);
        v2.set(0, 4);
        v2.set(1, 5);
        v2.set(2, 6);
        Vector v3 = v1.add(v2);
        System.out.println(v3);
        Vector v4 = v1.sub(v2);
        System.out.println(v4);
        Vector v5 = v1.mul(2);
        System.out.println(v5);
        double d1 = v1.dot(v2);
        System.out.println(d1);

        Matrix m1 = new Matrix(new double[][] {{1, 2, 3}, {4, 5, 6}});
        Matrix m2 = new Matrix(2, 3);
        m2.set(0, 0, 7);
        m2.set(0, 1, 8);
        m2.set(0, 2, 9);
        m2.set(1, 0, 10);
        m2.set(1, 1, 11);
        m2.set(1, 2, 12);
        Matrix m3 = m1.add(m2);
        System.out.println(m3);
        Matrix m4 = m1.sub(m2);
        System.out.println(m4);
        Matrix m5 = m1.mul(2);
        System.out.println(m5);
        Matrix m6 = m1.mul(m2.transpose());
        System.out.println(m6);

        UnmodifiableVector uv1 = new UnmodifiableVector(new double[] {1, 2, 3});
        UnmodifiableVector uv2 = new UnmodifiableVector(v2);
        UnmodifiableVector uv3 = uv1.add(uv2);
        System.out.println(uv3);
        System.out.println(uv3.get(1));
```

```
        UnmodifiableMatrix um1 = new UnmodifiableMatrix(new double[][] {{1, 2, 3}, {4, 5, 6}});
        UnmodifiableMatrix um2 = new UnmodifiableMatrix(m2);
        UnmodifiableMatrix um3 = um1.mul(um2.transpose());
        System.out.println(um3);
        System.out.println(um3.get(1, 1));

        // The following code should not compile
        // uv3.set(1, 10);
        // um3.set(1, 1, 10);
    }

}
```