Java应用技术 Homework

基本信息

- 个人信息：徐文皓，3210102377
- 课程班级：2023-2024学年秋冬学期，鲁伟明，周三9,10节
- 作业日期：2023.11.10

作业要求

- 对JDK不变类进行源码分析

  寻找 JDK 库中的任意3个不变类，进行源码分析，分析其为什么是不变的？文档说明其共性。

- 对字符串相关类进行源码分析

  对`String`、`StringBuilder`、`StringBuffer`进行源码分析。

  - 分析其主要数据组织及功能实现，有什么区别?
  - 说明为什么这样设计，这么设计对`String`，`StringBuilder`及`StringBuffer`的影响?
  - `String`，`StringBuilder`及`StringBuffer`分别适合哪些场景?
  - 回答以下问题

    为什么在以下代码段中，s1==s2 返回false，而s1==s3返回true?

    ```
    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));
    ```

- 设计实现不变类
  - 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。
  - 实现`UnmodifiableVector`，`UnmodifiableMatrix`不可变类
  - 实现`MathUtils`，含有静态方法
    - `UnmodifiableVector getUnmodifiableVector (Vector v)`
    - `UnmodifiableMatrix getUnmodifiableMatrix (Matrix m)`

作业内容

对JDK不变类进行源码分析

本次实验将使用`jdk-20.0.1`进行。

打开`jdk-20.0.1/lib/src.zip`文件，在此文件的`java.base/java/lang/`目录即可找到我们即将分析的三个不变类，它们分别具有不同的特征——

- `Integer.java`，`Integer`类的属性为一个值，它存在一些常用的实例。

- `Double.java`，`Double`类的属性为一个值，它很难找到一些常用的实例。
- `String.java`，`String`类的属性为一组值。

由于它们的源码都比较长，我们对于每个类只选取一些常用的属性和方法，以及在讨论“不变类”这一话题时必要的一些内容。

`Integer`不变类

```
@jdk.internal.ValueBased
public final class Integer extends Number
        implements Comparable<Integer>, Constable, ConstantDesc {
    @Native public static final int   MIN_VALUE = 0x80000000;
    @Native public static final int   MAX_VALUE = 0x7fffffff;
    private final int value;
    ...
    public static String toString(int i, int radix);
    public static int parseInt(String s, int radix) throws
NumberFormatException;
    ...
    public Integer(String s) throws NumberFormatException;
    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;
    }
    public String toString();
    ...
    private static class IntegerCache {
        static final int low = -128;
        static final int high;
        static final Integer[] cache;
        static Integer[] archivedCache;

        static {
            // high value may be configured by property
            int h = 127;
            String integerCacheHighPropValue =
                VM.getSavedProperty("java.lang.Integer.IntegerCache.high");
            if (integerCacheHighPropValue != null) {
                try {
                    h = Math.max(parseInt(integerCacheHighPropValue), 127);
                    // Maximum array size is Integer.MAX_VALUE
                    h = Math.min(h, Integer.MAX_VALUE - (-low) -1);
                } catch( NumberFormatException nfe) {
                    // If the property cannot be parsed into an int, ignore
it.
                }
            }
```

```
            high = h;

            // Load IntegerCache.archivedCache from archive, if possible
            CDS.initializeFromArchive(IntegerCache.class);
            int size = (high - low) + 1;

            // Use the archived cache if it exists and is large enough
            if (archivedCache == null || size > archivedCache.length) {
                Integer[] c = new Integer[size];
                int j = low;
                for(int i = 0; i < c.length; i++) {
                    c[i] = new Integer(j++);
                }
                archivedCache = c;
            }
            cache = archivedCache;
            // range [-128, 127] must be interned (JLS7 5.1.7)
            assert IntegerCache.high >= 127;
        }

        private IntegerCache() {}
    }
    ...
    public boolean equals(Object obj) {
        if (obj instanceof Integer) {
            return value == ((Integer)obj).intValue();
        }
        return false;
    }
    ...
}
```

`Interger`类被声明为`final`，不可继承。`Integer`类是对原始数据类型`int`的包装，在类的注释中提到：

> An object of type `Integer` contains a single field whose type is `int`.

可以注意到其实例的核心成员变量`int value`、包括一些相关量如`MIN_VALUE`等均被声明为`private final`，表明其私有不可修改。对于其构造方法`Integer(int value)`，在实例被构造时对`value`进行赋值。在类实例存在期间，考虑`int intValue()`方法，由于`int value`是原始数据类型而非对象，因此其返回值并不属于一个引用，无法通过此对实例进行修改。通过以上操作，`Integer`类的实例被创建后其属性就不再可以被修改，使得`Integer`成为了一个不变类。

由于`int`变量使用极其频繁，`Integer`类内部提供了一个静态工厂，即`private static class IntegerCache`，在调用`Integer valueOf(int i)`方法时优先在静态工厂中取用，会首先判断所需值是否在静态工厂内，只有不在时才会创建一个新实例。

下面观察其静态内部类`IntegerCache`。在`Integer`类被加载时，`IntergetCache`将在`Integer valueOf(int i)`方法前被加载。在它被加载时，首先检查JVM是否自定义了`integerCacheHighPropValue`，如有则将`high`设定为相应值，否则设置为`127`。该类提供了一个包括值在`-128,high`之间的所有`Integer`的缓存池，检查存档内是否存在大小符合要求的`IntegerCache`，如有则直接使用，否则重新创建一个`Integer`数组并初始化。该类的构造方法`IntegerCahce()`被声明为`private`，表明仅在类加载时进行一次构造。

此外，`Integer`类还向我们提供了`String toString(int i, int radix)`、`int parseInt(String s, int radix)`、`boolean equals(Object obj)`等常用方法。

`Double`不变类

```
@jdk.internal.ValueBased
public final class Double extends Number
        implements Comparable<Double>, Constable, ConstantDesc {
    public static final double POSITIVE_INFINITY = 1.0 / 0.0;
    public static final double NEGATIVE_INFINITY = -1.0 / 0.0;
    public static final double NaN = 0.0d / 0.0;
    public static final double MAX_VALUE = 0x1.fffffffffffffP+1023;
    public static final double MIN_NORMAL = 0x1.0p-1022;
    public static final double MIN_VALUE = 0x0.0000000000001P-1022;
    public static final int SIZE = 64;
    public static final int PRECISION = 53;
    public static final int MAX_EXPONENT = (1 << (SIZE - PRECISION - 1)) -
1; // 1023
    public static final int MIN_EXPONENT = 1 - MAX_EXPONENT; // -1022
    ...
    public static String toString(double d);
    public static double parseDouble(String s) throws NumberFormatException;
    public static boolean isNaN(double v);
    public static boolean isInfinite(double v);
    ...
    private final double value;
    public static Double valueOf(double d) {
        return new Double(d);
    }
    public Double(double value) {
        this.value = value;
    }
    public double doubleValue() {
        return value;
    }
    public boolean equals(Object obj) {
        return (obj instanceof Double)
               && (doubleToLongBits(((Double)obj).value) ==
                      doubleToLongBits(value));
    }
}
```

`Double`类被声明为`final`，不可继承。`Double`类是对原始数据类型`double`的包装，在类的注释中提到：

> An object of type `Double` contains a single field whose type is `double`.

与`Integer`类相似地，可以注意到其实例的核心成员变量`double value`、包括一些相关量如`NaN`等均被声明为`private final`，表明其私有不可修改。对于其构造方法`Double(double value)`，在实例被构造时对`value`进行赋值。在类实例存在期间，考虑`double doubleValue()`方法，由于`double value`是原始数据类型而非对象，因此其返回值并不属于一个引用，无法通过此对实例进行修改。通过以上操作，`Double`类的实例被创建后其属性就不再可以被修改，使得`Double`成为了一个不变类。

与`Integer`类相比，`Double`类由于`double`本身的特殊性，增加了许多常量。而和`Integer`不同的是，`Double`很难找到类似`...,-1,0,1,2,3,4,...`这样使用频率极高的实例，从而并没有提供缓存池。可见其`Double valueOf(double d)`方法会直接创建一个新的`Double`实例并返回。

`Double`类向我们提供了`String toString(double d)`、`double parseDouble(String s)`、`boolean isNaN(double v)`、`boolean isInfinite(double v)`、`boolean equals(Object obj)`等常用方法。

`String`不变类

```
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc {
    private final byte[] value;
    private final byte coder;
    private int hash; // Default to 0
    private boolean hashIsZero; // Default to false;
    ...
    public String(String original) {
        this.value = original.value;
        this.coder = original.coder;
        this.hash = original.hash;
        this.hashIsZero = original.hashIsZero;
    }
    ...
    public char charAt(int index) {
        if (isLatin1()) {
            return StringLatin1.charAt(value, index);
        } else {
            return StringUTF16.charAt(value, index);
        }
    }
    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);
    }
    public String toUpperCase(Locale locale) {
        return isLatin1() ? StringLatin1.toUpperCase(this, value, locale)
                          : StringUTF16.toUpperCase(this, value, locale);
    }
    public String toUpperCase() {
        return toUpperCase(Locale.getDefault());
    }
    public boolean equals(Object anObject) {
        if (this == anObject) {
            return true;
        }
```

```
        return (anObject instanceof String aString)
                && (!COMPACT_STRINGS || this.coder == aString.coder)
                && StringLatin1.equals(value, aString.value);
    }
    public static String valueOf(char[] data) {
        return new String(data);
    }
    public static String valueOf(Object obj) {
        return (obj == null) ? "null" : obj.toString();
    }
    public String toString();
}
```

相较于之前提到的两个类，`String`的属性更多，其中包括一个数组`byte[] value`，这要求我们在讨论不可变类时注意引用的影响。

`String`类被声明为`final`，不可继承。其实例的核心成员变量`byte[] value`和`byte coder`，均被声明为`private final`。但是，与上面两个类不同，`byte[] value`这一用于存储字符串内容的数组被声明为`final`之后，其元素仍然是可变的，只是限制了数组引用为不可变状态。因此，`String`本身的不可变，从内部而言，`final`只是起了一部分左右，而更多地要依靠实现时的"谨慎"；从外部而言，其`private`访问权限以及后续会讲到的不存在修改方法为`String`成为不可变类提供了坚实保障。而`byte coder`这一值用于标注`byte[] value`的情况，如果所有字符都是`ASCII`可以表示的字符，则`byte coder = 0 (LATIN1)`，在后续的处理中每个字节都将被视作一个字符。否则，存在`ASCII`无法表示的字符，`byte coder = 1 (UTF16)`，在后续的处理中每两个字节将被视作一个字符。从而，`byte[] value`和`byte coder`这两个量可以唯一确定字符串的内容信息，它们不被修改，实例的内容就没有发生实质上的改变。

关于获取其成员、尤其是在与`byte[] value`相关的方法中，均较好地避免了对`byte[] value`直接操作或者返回其引用。例如，在`String substring(int beginIndex, int endIndex)`中，返回值均为新创建的实例等。

```
public static String toUpperCase(String str, byte[] value, Locale locale) {
        ...
        byte[] result = new byte[len];
        System.arraycopy(value, 0, result, 0, first);  // Just copy the
first few
                                                       // upperCase
characters.
        for (int i = first; i < len; i++) {
            int cp = value[i] & 0xff;
            ...
            result[i] = (byte)cp;
        }
        return new String(result, LATIN1);
    }
```

我们以`String toUpperCase()`为例讨论它能够维持`String`的不变性。以`Latin1`编码为例，在将所有字符转为大写模式的过程中，先调用`System.arraycopy(/*Omitted*/)`将`byte[] value`拷贝到`byte[] result`中，在对`byte[] result`中的内容进行操作，最后再以此创建一个新的`String`实例作为返回值。这一过程有效地避免了对`byte[] value`进行任何相关变更。

```java
// java.lang.Object
public String toString() {
        return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

值得一提的是，我们观察到`String`类提供的`String valueOf(Object obj)`方法，相比于`Object`类本身的`String toString()`方法，它提供了`null`判别，给我们带来了便利。

此外，`String`提供的常量池技术很好地发挥了不可变类的优势，在此处我们不做展开。

小结

经过对以上源码的分析，我们发现它们具有这样的共性：

- 类本身不可被继承

  通过`final`关键字，我们将类声明为不可继承类，从而避免了在其子类中对其方法进行重写，之后使用多态机制在以该类为数据类型的实例中通过实例方法的动态绑定而更改预期结果这一现象。

- 类本身不提供公开的属性修改器

  无法从外部通过类的某个静态或实例方法对其属性作出修改。

- 类的关键属性被声明为`private final`

  类的关键属性被声明为`private`，使得无法从外部访问这一属性。

  类的关键属性被声明为`final`，使得这一属性无法被修改。

  但是，对于数组等非原始属性，`final`关键字只保证引用不发生变化，而其具体内容的不变还需要在实现过程中保证。

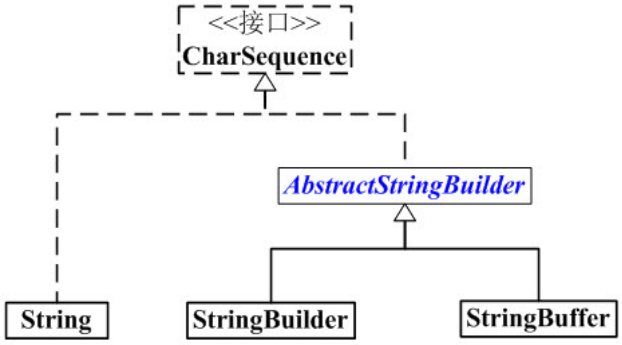
- 类的方法不返回可变对象的引用

一般地，在对不可变类的属性进行操作时，如果是引用属性，则将其克隆并在克隆上进行操作。如果内容有变，方法的返回值一般为一个新创建的实例。

对字符串相关类进行源码分析

```java
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence,
               Constable, ConstantDesc{}
public final class StringBuffer extends AbstractStringBuilder
    implements Serializable, Comparable<StringBuffer>, CharSequence{}
public final class StringBuilder extends AbstractStringBuilder
    implements java.io.Serializable, Comparable<StringBuilder>,
CharSequence{}
abstract sealed class AbstractStringBuilder implements Appendable,
CharSequence
    permits StringBuilder, StringBuffer{}
```

首先，我们可以发现，与`String`类不同，`StringBuilder`和`StringBuffer`由`AbstractStringBuilder`派生，后两者的重合度较高，我们在比较时可以侧重比较`String`类与`AbstractStringBuilder`的区别。从类头部来看，几个类都可序列化、可比较，实现了`CharSequence`接口，其中`String`为常量，`AbstractStringBuilder`是可追加的。



为了突出它们的区别，我们接下来忽略Latin1/UTF16编码方式相关的属性和方法，仅就其数据进行讨论。

```
private final byte[] value; // in String

byte[] value; // in AbstractStringBuilder
int count; // in AbstractStringBuilder

/**
 * A cache of the last value returned by toString. Cleared
 * whenever the StringBuffer is modified.
 */
private transient String toStringCache; // in StringBuffer
```

就其主要属性而言，`String`类为一个被声明`private final`的字节数组，用于存储字符串内容；其余两者为`byte[] value`和`int count`，分别用于存储字符串内容和长度。`StringBuffer`中还维护了一个不可序列化的`String toStringCache`，用于存储当前字符串内容最后一次被调用`toString()`时的返回值。

不可变类`String`的字符串内容是固定的，而`AbstractStringBuilder`提供了`private int newCapacity(int minCapacity)`使得字符串内容变量在数组长度不够时可以扩展。

`AbstractStringBuilder`对字符串的操作更加灵活，相对于`String`，它主要增加了一些关于字符串追加、插入、删除、逆置等常用操作：

```
public AbstractStringBuilder append(String str);
public AbstractStringBuilder delete(int start, int end);
public AbstractStringBuilder insert(int index, char[] str, int offset, int
len);
public AbstractStringBuilder reverse();
public void setCharAt(int index, char ch);
```

值得注意的是，相较于`StringBuilder`，`StringBuffer`在对字符串内容做变更的方法中增加了对`toStringCache`的维护，同时也加入了`synchronized`关键字，保证了线程安全。

```java
public synchronized StringBuffer append(String str) { // in StringBuffer
    toStringCache = null;
    super.append(str);
    return this;
}

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

特别地，我们考虑连接操作。在连接操作中，如果连接对象非空，`String`离不开创建新实例的过程。

```java
// in String
public String concat(String str) {
    if (str.isEmpty()) {
        return this;
    }
    return StringConcatHelper.simpleConcat(this, str);
}

// in StringConcatHelper
static String simpleConcat(Object first, Object second) {
        String s1 = stringOf(first);
        String s2 = stringOf(second);
        if (s1.isEmpty()) {
            // newly created string required, see JLS 15.18.1
            return new String(s2);
        }
        if (s2.isEmpty()) {
            // newly created string required, see JLS 15.18.1
            return new String(s1);
        }
        // start "mixing" in length and coder or arguments, order is not
        // important
        long indexCoder = mix(initialCoder(), s1);
        indexCoder = mix(indexCoder, s2);
        byte[] buf = newArray(indexCoder);
        // prepend each argument in reverse order, since we prepending
        // from the end of the byte array
        indexCoder = prepend(indexCoder, buf, s2);
        indexCoder = prepend(indexCoder, buf, s1);
        return newString(buf, indexCoder);
    }
```

相比之下，`AbstractStringBuilder`在原实例上操作，较大地提升了操作性能。

```java
public AbstractStringBuilder append(String str) {
    if (str == null) {
        return appendNull();
    }
    int len = str.length();
    ensureCapacityInternal(count + len);
    putStringAt(count, str);
```

```
    count += len;
    return this;
}

private void putStringAt(int index, String str) {
    inflateIfNeededFor(str);
    str.getBytes(value, index, coder);
}
```

事实上，一旦需要发生改变，`String`一般离不开实例的创建、回收，而`AbstractStringBuilder`一般不需要相关操作。

另外，值得注意的是`StringBuilder`和`StringBuffer`的`toString()`方法。

```
public synchronized String toString() { // in StringBuffer
    if (toStringCache == null) {
        return toStringCache = new String(this, null);
    }
    return new String(toStringCache);
}

public String toString() { // in StringBuilder
    return new String(this);
}
```

正如前文提到的，`StringBuffer`会额外维护一个`toStringCache`。经查，这个属性因为持有着字符串内容的String引用，可以减少一部分内存占用，不过可能解释更为合理的还是由于历史遗留问题不便取缔，它的必要性并不很大。

此外，它们在序列化、反序列化的实现方式上也有所不同，此处不做展开。

小结

我们对`String`、`StringBuffer`、`StringBuilder`这三个类的区别进行小结，并对任务要求做出回应。

- 分析主要数据组织及功能实现，它们有什么区别？
  - 不考虑编码、哈希等次要因素，`String`中只存储了字符串本身，而后两个类维护了字符串的长度。
  - 在实现时，`String`多创建新实例，而后两个类在字符串本身上进行操作，并增设了对字符串的追加、插入、删除、逆置等操作。
  - `StringBuffer`在对字符串进行修改时多用`synchronized`关键字。
- 说明为什么这样设计，这么设计对`String`，`StringBuilder`及`StringBuffer`的影响？
  - 后两个类的字符串是可变长的，因此需要长度属性来支持。
  - `String`维持了自身作为不可变类的优势，而后两个类提高了自身的灵活性。
  - `StringBuffer`在作为可变类的同时保证了线程安全。

类	性能	安全性
String	低	不可变类
StringBuffer	较高	线程安全

类	性能	安全性
StringBuilder	高	线程非安全

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

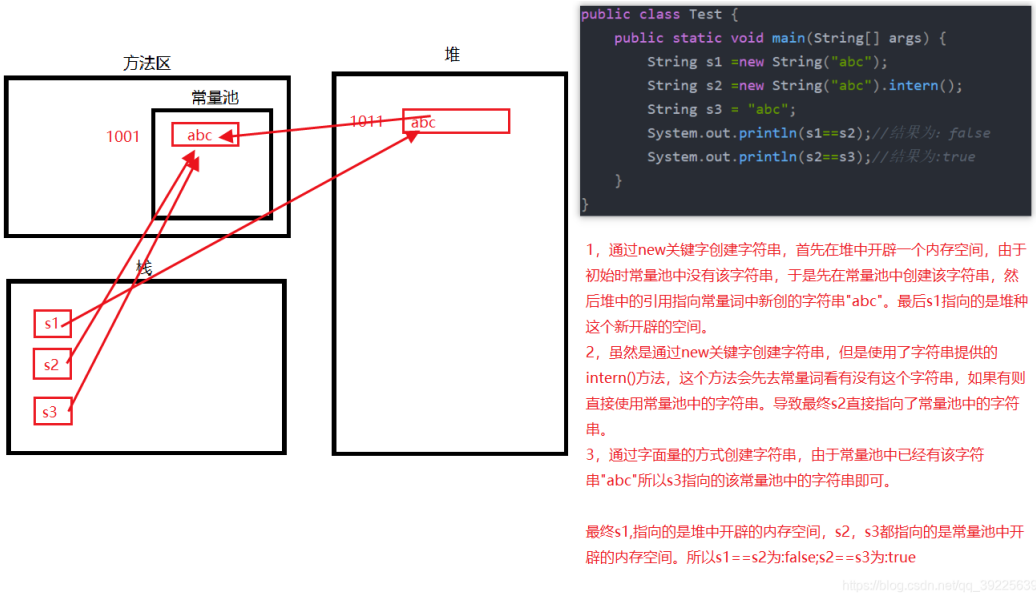
  结合它们各自的特点，

  - `String`适用于存储不需要频繁修改的字符串、不需要频繁操作的字符串
  - `StringBuilder`适用于在单线程操作字符串缓冲区下操作大量数据
  - `StringBuffer`适用于在多线程操作字符串缓冲区下操作大量数据

- 为什么在以下代码段中，s1==s2 返回false，而s1==s3返回true?

```
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));
```

`String`作为不可变类，拥有常量池，对字面常量可以实现复用。对于类实例，`==`运算符用于判断它们是否是同一实例的引用。`s1`被创建后，`"Welcome to Java"`被加入到常量池中，在创建`s3`时，JVM在常量池中找到该实例并将`s3`赋值为其引用，两者是同一实例的引用，因此`s1 == s3`结果为`true`。而`s2`是直接由`new`关键字创建的，JVM会直接在堆上创建一个新实例，与`s1`引用的实例不同，因此`s1 == s2`结果为`false`。



设计实现不变类

为突出工作重点，在此处我们不考虑异常情况，假设对相关类的所有操作都是符合操作规则的。

设计实现`Vector`、`Matrix`可变类

我们实现可变类`DoubleVector`，用于存放向量。该类的属性有向量内容`double[] value`、向量维数`int size`和向量内容容量`capacity`，主要方法有在向量末尾追加分量(`boolean pushBack(double x)`、`boolean pushBackAll(double[] c)`)、获取和修改某一分量(`double get(int index)`、`boolean set(int index, double newValue)`)、向量的加减和数乘运算(`DoubleVector add(DoubleVector opr)`、`DoubleVector minus(DoubleVector opr)`、`DoubleVector scalar(double factor)`)，以及`boolean equals(Object anObject)`和`String toString()`。

```
public class DoubleVector {

    double[] value;
    private int capacity;
    private int size;

    private int FindMinPowerOf2(int x);
    private void ensureCapacity(int minCapacity);

    public DoubleVector();
    public DoubleVector(double[] value);
    public DoubleVector(DoubleVector c);

    public boolean pushBack(double x);
    public boolean pushBackAll(double[] c);

    public int capacity() ;
    public int size();

    public double get(int index);
    public boolean set(int index, double newValue);

    public DoubleVector add(DoubleVector opr);
    public DoubleVector minus(DoubleVector opr);
    public DoubleVector scalar(double factor);

    public boolean equals(Object anObject) ;
    public String toString();
}
```

使用以下的测试代码，

```
DoubleVector v1 = new DoubleVector(new double[] { 1.0, 2.0 });
System.out.println("v1 after initialize: " + v1
        + ", with size " + v1.size() + " and capacity " + v1.capacity());
v1.pushBackAll(new double[] { 3.0, 4.0 });
System.out.println("v1 after pushback: " + v1
        + ", with size " + v1.size() + " and capacity " + v1.capacity());
v1.set(3, 5.0);
System.out.println("v1 after set: " + v1
        + ", with size " + v1.size() + " and capacity " + v1.capacity());
DoubleVector v2 = new DoubleVector(new double[] { 1.0, 2.0, -1.0, 3.0 });
System.out.println("v2 after initialize: " + v2
        + ", with size " + v2.size() + " and capacity " + v2.capacity());
System.out.println("v1 + v2 = " + v1.add(v2));
System.out.println("v1 - v2 = " + v1.minus(v2));
System.out.println("v1 * 1.5 = " + v1.scalar(1.5));
for (int i = 0; i < 5; i++) {
    v1.pushBack(i);
    System.out.println("v1 after pushback: " + v1
            + ", with size " + v1.size() + " and capacity " +
v1.capacity());
}
```

可以观察到输出结果如下：

```
v1 after initialize: [ 1.0, 2.0], with size 2 and capacity 2
v1 after pushback: [ 1.0, 2.0, 3.0, 4.0], with size 4 and capacity 4
v1 after set: [ 1.0, 2.0, 3.0, 5.0], with size 4 and capacity 4
v2 after initialize: [ 1.0, 2.0, -1.0, 3.0], with size 4 and capacity 4
v1 + v2 = [ 2.0, 4.0, 2.0, 8.0]
v1 - v2 = [ 0.0, 0.0, 4.0, 2.0]
v1 * 1.5 = [ 1.5, 3.0, 4.5, 7.5]
v1 after pushback: [ 1.0, 2.0, 3.0, 5.0, 0.0], with size 5 and capacity 8
v1 after pushback: [ 1.0, 2.0, 3.0, 5.0, 0.0, 1.0], with size 6 and capacity 8
v1 after pushback: [ 1.0, 2.0, 3.0, 5.0, 0.0, 1.0, 2.0], with size 7 and capacity 8
v1 after pushback: [ 1.0, 2.0, 3.0, 5.0, 0.0, 1.0, 2.0, 3.0], with size 8 and capacity 8
v1 after pushback: [ 1.0, 2.0, 3.0, 5.0, 0.0, 1.0, 2.0, 3.0, 4.0], with size 9 and capacity 16
```

与`DoubleVector`不同，`DoubleMatrix`类的行数和列数在构造时就已经被确定。该类的属性有矩阵内容`double[][] value`、矩阵行数`numOfRow`和列数`numOfCol`，主要方法有获取和修改某个元素(`double get(int i, int j)`、`boolean set(int i, int j, double newValue)`)，矩阵的加减、数乘和乘法运算(`DoubleMatrix add(DoubleMatrix opr)`、`DoubleMatrix minus(DoubleMatrix opr)`、`DoubleMatrix scalar(double factor)`、`DoubleMatrix multiply(DoubleMatrix opr)`)，以及`boolean equals(Object anObject)`和`String toString()`。

```java
public class DoubleMatrix {
    double[][] value;
    int numOfRow;
    int numOfCol;

    public DoubleMatrix(int numOfRow, int numOfCol);
    public DoubleMatrix(int n);
    public DoubleMatrix(double[][] matrix);
    public DoubleMatrix(DoubleMatrix c);

    public double get(int i, int j);
    public boolean set(int i, int j, double newValue);

    public DoubleMatrix add(DoubleMatrix opr);
    public DoubleMatrix minus(DoubleMatrix opr);
    public DoubleMatrix scalar(double factor);
    public DoubleMatrix multiply(DoubleMatrix opr);

    public boolean equals(Object anObject);
    public String toString();
}
```

使用以下的测试代码：

```java
DoubleMatrix m1 = new DoubleMatrix(new double[][] { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } });
System.out.println("m1: " + m1);
DoubleMatrix m2 = new DoubleMatrix(new double[][] { { 1, 0, 0 }, { 1, 1, 0 }, { 0, 1, 1 } });
System.out.println("m2: " + m2);
System.out.println("m1 + m2: " + m1.add(m2));
System.out.println("m1 * 1.5: " + m1.scalar(1.5));
System.out.println("m1 * m2: " + m1.multiply(m2));
DoubleMatrix m3 = new DoubleMatrix(new double[][] { { 1 }, { 2 }, { 1 } });
System.out.println("m3: " + m3);
System.out.println("m1 * m3: " + m1.multiply(m3));
```

可以观察到输出结果如下：

```
m1: Matrix[3,3]
1.0,2.0,3.0
4.0,5.0,6.0
7.0,8.0,9.0

m2: Matrix[3,3]
1.0,0.0,0.0
1.0,1.0,0.0
0.0,1.0,1.0

m1 + m2: Matrix[3,3]
2.0,2.0,3.0
5.0,6.0,6.0
7.0,9.0,10.0

m1 * 1.5: Matrix[3,3]
1.5,3.0,4.5
6.0,7.5,9.0
10.5,12.0,13.5

m1 * m2: Matrix[3,3]
3.0,5.0,3.0
9.0,11.0,6.0
15.0,17.0,9.0

m3: Matrix[3,1]
1.0
2.0
1.0

m1 * m3: Matrix[3,1]
8.0
20.0
32.0
```

设计实现`UnmodifiableVector`和`UnmodifiableMatrix`不变类

`UnmodifiableVector`类的属性和方法与`DoubleVector`类似，我们将类声明为`final`，将其属性声明为`private final`，去掉了会对实例状态进行修改的方法，并增加了`UnmodifiableVector(DoubleVector) c`这一构造方法和`indexOf`相关方法，便于后续使用。

```java
public final class UnmodifiableVector {

    private final double[] value;
    private final int size;

    public UnmodifiableVector();
    public UnmodifiableVector(double[] value);
    public UnmodifiableVector(DoubleVector c);
    public UnmodifiableVector(UnmodifiableVector c);

    public int size();
    public double get(int index);

    public UnmodifiableVector add(UnmodifiableVector opr);
    public UnmodifiableVector minus(UnmodifiableVector opr);
    public UnmodifiableVector scalar(double factor);

    public UnmodifiableVector indexOf(double[] value);
    public UnmodifiableVector indexOf(DoubleVector value);
    public UnmodifiableVector indexOf(UnmodifiableVector value);

    public boolean equals(Object anObject);
    public String toString();
}
```

`UnmodifiableMatrix`类的属性和方法与`DoubleMatrix`类似，我们将类声明为`final`，将其属性声明为`private final`，去掉了会对实例状态进行修改的方法，并增加了`UnmodifiableVector(DoubleVector) c`这一构造方法和`indexOf`相关方法，便于后续使用。

```java
public final class UnmodifiableMatrix {
```

```
    private final double[][] value;
    private final int numOfRow;
    private final int numOfCol;

    public UnmodifiableMatrix();
    public UnmodifiableMatrix(double[][] matrix);
    public UnmodifiableMatrix(DoubleMatrix c);
    public UnmodifiableMatrix(UnmodifiableMatrix c);

    public double get(int i, int j);

    public UnmodifiableMatrix add(UnmodifiableMatrix opr);
    public UnmodifiableMatrix minus(UnmodifiableMatrix opr);
    public UnmodifiableMatrix scalar(double factor);
    public UnmodifiableMatrix multiply(UnmodifiableMatrix opr);

    public UnmodifiableMatrix indexOf(double[][] value);
    public UnmodifiableMatrix indexOf(DoubleMatrix value);
    public UnmodifiableMatrix indexOf(UnmodifiableMatrix value);

    public boolean equals(Object anObject);
    public String toString();
}
```

实现 `MathUtils`

`MathUtils`将获取不变向量和获取不变矩阵的方法包装到一起即可。

```
public class MathUtils {
    public static UnmodifiableVector getUnmodifiableVector(DoubleVector v){
        return new UnmodifiableVector(v);
    }
    public static UnmodifiableMatrix getUnmodifiableMatrix(DoubleMatrix m){
        return new UnmodifiableMatrix(m);
    }
}
```

使用以下的测试代码：

```
DoubleVector v1 = new DoubleVector(new double[]{2,3,7,7});
DoubleVector v2 = new DoubleVector(v1);
UnmodifiableVector uv1 = MathUtils.getUnmodifiableVector(v1);
UnmodifiableVector uv2 = MathUtils.getUnmodifiableVector(v2);
System.out.println("uv1 = "+ uv1);
System.out.println("uv2 = "+ uv2);
System.out.println("uv1 + uv2 = " + uv1.add(uv2));
System.out.println("uv1 - uv2 = " + uv1.minus(uv2));
System.out.println("uv1 * 0.5 = " + uv1.scalar(0.5));
System.out.println();

DoubleMatrix m1 = new DoubleMatrix(new double[][] { { 2, 0, 2, 3 }, { 1, 1, 1, 2 } });
DoubleMatrix m2 = new DoubleMatrix(new double[][] { { 2, 0 }, { 2, 3 }, { 1, 1 }, { 1, 2 } });
UnmodifiableMatrix um1 = MathUtils.getUnmodifiableMatrix(m1);
```

```java
UnmodifiableMatrix um2 = MathUtils.getUnmodifiableMatrix(m2);
System.out.println("um1 = " + um1);
System.out.println("um2 = " + um2);
System.out.println("um1 + um1 = " + um1.add(um1));
System.out.println("um2 - um2 = " + um2.minus(um2));
System.out.println("um1 * 0.5 = " + um1.scalar(0.5));
System.out.println("um1 * um2 = " + um1.multiply(um2));
```

可以观察到输出结果如下：

```
uv1 = [ 2.0, 3.0, 7.0, 7.0]
uv2 = [ 2.0, 3.0, 7.0, 7.0]
uv1 + uv2 = [ 4.0, 6.0, 14.0, 14.0]
uv1 - uv2 = [ 0.0, 0.0, 0.0, 0.0]
uv1 * 0.5 = [ 1.0, 1.5, 3.5, 3.5]

um1 = Matrix[2,4]
2.0,0.0,2.0,3.0
1.0,1.0,1.0,2.0

um2 = Matrix[4,2]
2.0,0.0
2.0,3.0
1.0,1.0
1.0,2.0

um1 + um1 = Matrix[2,4]
4.0,0.0,4.0,6.0
2.0,2.0,2.0,4.0

um2 - um2 = Matrix[4,2]
0.0,0.0
0.0,0.0
0.0,0.0
0.0,0.0

um1 * 0.5 = Matrix[2,4]
1.0,0.0,1.0,1.5
0.5,0.5,0.5,1.0

um1 * um2 = Matrix[2,2]
9.0,8.0
7.0,8.0
```