## Questions (in the real exam there will be 3 questions)

1. Describe how interfaces differ from abstract classes, and give an example of a programming problem in which abstract classes are more appropriate to use than interfaces or other mechanisms available in object-oriented programming languages.

2. Describe **three** errors/breaches of good programming practices in the following `Matrix` class, and **write unit tests** for it (or if you don't remember the JUnit syntax, describe what should be tested in the unit tests).

```
public class Matrix implements Iterable<Double> {
        private double[][] data;

        /**
         * Constructor.
         *
         * @param data contents of the matrix
         */
        public Matrix(double[][] data) {
                this.data = data;
        }

        /**
         * Gets the number of rows.
         *
         * @return number of rows
         */
        public int getNrRows() {
                return data.length;
        }

        /**
         * Gets the number of columns.
         *
         * @return number of columns
         */
        public int getNrColumns() {
                return data[0].length;
        }
```

```
/**
 * Constructs a deep copy of this matrix.
 *
 * @return a deep copy of this matrix
 */
public Matrix copy() {
        return new Matrix(this.data);
}

/**
 * Sets an element.
 *
 * @param row the row index. PRECOND: 0 <= row < getNrRows()
 * @param col the column index. PRECOND: 0 <= col < getNrCols()
 * @param val the new element value
 */
public void setElement(int row, int col, double val) {
        if (row < 0 || row >= getNrRows()) {
                throw new IllegalArgumentException("PREC V row");
        }
        if (col < 0 || col >= getNrCols()) {
                throw new IllegalArgumentException("PREC V col");
        }
        data[row][col] = val;
}

/**
 * Gets an element.
 *
 * @param row the row index. PRECOND: 0 <= row < getNrRows()
 * @param col the column index. PRECOND: 0 <= col < getNrCols()
 * @return the element at [row][col]
 */
public int getElement(int row, int col) {
        return data[row][col];
}

/**
 * Gets the iterator
 *
```

```
         * @see Iterable#iterator
         */
        public Iterator<Double> iterator() {
                return new Iterator<Double>(this);
        }
}
```

## Answers

1. Interfaces describe new object types and method signatures for these types. A class can implement an interface, in which case it has to provide an implementation for all the methods (with the exact same signatures) defined in the interface class. Abstract classes are classes where some of the methods are without an implementation (like with interfaces), but for other methods there can be an implementation. Another difference is that interface classes cannot have instance variables (fields), whereas abstract classes may contain these as well.
   Abstract classes are more appropriate than interfaces for programming problems where there are classes that have a common part for which the implementation is the same, but also a part that varies with the subtypes and that is used by the common part. For example, in the Java Collections Framework, the AbstractSet class is extended by different concrete set implementations (e.g. TreeSet and HashSet), with different iterator implementations. Now, all concrete implementations have the same removeAll method implemented in AbstractSet. removeAll uses the subclass-specific iterator implementations for iterating over the elements and removing them. Without abstract classes, this functionality would have to be repeated in each subclass, because the iterator implementation is not known for the AbstractSet.

2. First mistake: Iterator<Double> is an interface type, and cannot be instantiated in the iterator() method. Second mistake: the copy() should do deep copying, but copies the data reference, and is therefore constructing a shallow copy. Third mistake: getElement() has preconditions that are documented but not checked in the beginning of the method.

```
public class MatrixTest {

        private double[][] data;
        private Matrix m;
        @Before
        public void setUp() {
                data = new double[2][1];
                data[0][0] = 3.0;
                data[1][0] = 2.0;
                m = new Matrix(data);
        }

        @Test
        public void testGetNrRows() {
                assertEquals(2, m.getNrRows());
        }

        @Test
        public void testGetNrColumns() {
                assertEquals(1, m.getNrColumns());
        }

        @Test
        public void testCopy() {
                m2 = m.copy();
                assertEquals(2, m2.getNrRows());
                assertEquals(1, m2.getNrColumns());
                assertEquals(3.0, m2.getElement(0, 0), 0.001);
                assertEquals(2.0, m2.getElement(1, 0), 0.001);
                m.setElement(0, 0, 1.0);
                assertEquals(3.0, m2.getElement(0, 0), 0.001);
        }
}
```