

Voortgezet Programmeren

Lecture 3: Programming by contract

Tommi Tervonen

Econometric Institute, Erasmus University Rotterdam

```
public class DataSet {  
  
    private ArrayList<double[]> data;  
  
    public DataSet(String fname) throws IOException ,  
    FileNotFoundException { ... }  
  
    public int getNrColumns() { ... }  
  
    public double[] getColumn(int index) { ... }  
  
}
```

Private / public visibility

- public are visible to everyone
- private visible only within the **class**: also other objects of the same class can access them (motivation: if you modify the variable type, you can also modify the uses)

```
public class Matrix {  
    private double[] data;  
    private int nrows;  
  
    public Matrix(Matrix other) {  
        this.nrows = other.nrows;  
        copyData(other.data);  
    }  
    ...  
}
```

Private methods

- Maximize use of private methods for code clarity and to avoid redundancy
- Every method should fit in one screen of code

```
public DataSet(String fname) throws IOException {  
    this.fname = fname;  
    FileReader fr = new FileReader(fname);  
    BufferedReader rdr = new BufferedReader(fr);  
    loadData(rdr);  
}
```

```
private void loadData(Reader rdr)  
    throws IOException {  
    ...  
}
```

- Java passes primitives by value, objects by reference
- Side effects can occur when mutable objects are passed as parameters (or: object the method is called on can be modified)

```
/**  
 * Sorts the array in ascending order from  
 * from sIndex. I.e. guarantees the post-condition:  
 * array[sIndex] <= ... <= array[array.length-1]  
 *  
 * @param array the array to sort.  
 * @param sIndex the starting index.  
 * PRECOND: 0 <= sIndex < array.length  
 */  
void sortFromIndex(double[] array, int sIndex)
```

Violating pre-conditions

- Crash the program execution by throwing an unchecked exception (e.g. `IllegalArgumentException`)
- By convention, null references should never be passed in Java (or `NullPointerException` is thrown)

```
public void sortFromIndex(int [] array , int index) {  
    if(index < 0 || index >= array.length) {  
        throw new IllegalArgumentException("outofb");  
    }  
    // ... do the actual sorting  
}
```

- In addition to unchecked exceptions (e.g. `IllegalArgumentException`), java has `assert` keyword that checks for a condition
- Assertions are only enabled during development as they can do computationally expensive checks (similar convention in C, but not in Matlab!)
- Need to be enabled in Eclipse (run as / run configurations / arguments / VM arguments: add "-ea")
- Failed assertions throw `AssertionException`
- Do not use assertions to check pre-conditions of public methods!

How do you know your method works?

- Unit testing refers to automated testing of code functionality a "unit" at a time (e.g. method)
- We test only public methods (=the interface)
- Not tested = doesn't work
- A single unit test tests one functionality, and tests can be grouped to test suites (usually 1 test suite with all tests)
- In Eclipse the JUnit library needs to be added to build path

```
public void sortFromIndex (int[] array , int index)
```

```
public class SorterTest {  
  
    @Test  
    public void testSort() {  
        Sorter s = new Sorter();  
  
        int [] arr = new int []{3.0, 2.0, 1.0};  
        s.sortFromIndex(arr, 1);  
  
        assertEquals(3.0, arr[0]);  
        assertEquals(1.0, arr[1], 0.00001);  
        assertTrue(arr[2] == 2.0);  
  
        s.sortFromIndex(arr, 0);  
        assertEquals(new double []{1.0, 2.0, 3.0},  
            arr, 0.00001);  
    }  
}
```

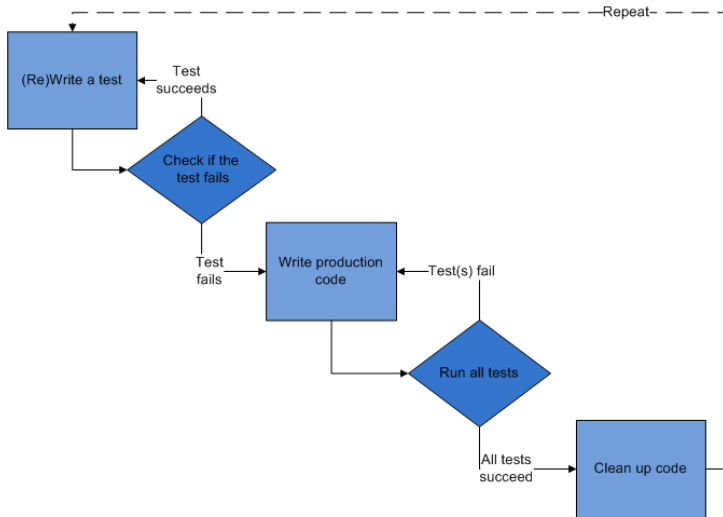
■ Example: `fi.smaa.jsmaa.model.ScaleCriterion` tests

```
public class StudentTest {  
    private Student s;  
    @Before  
    public void setUp() {  
        s = new Student("tommi" , 1212);  
    }  
    @Test  
    public void testConstructor() {  
        assertEquals("tommi" , s.getName());  
        assertEquals(1212, s.getNumber());  
    }  
    @Test  
    public void testSetGetName() {  
        assertEquals("tommi" , s.getName());  
        s.setName("x");  
        assertEquals("x" , s.getName());  
    }  
    @Test  
    public void testSetGetNumber() { ...
```

Why test?

- Unit tests document functionality
- Unit tests provide a safety net ("let me change this ... does it break something?")
- More tests = more trust in your code
- Bug = **lack of a test**
- Test-driven development

Test-driven development



- Classes can have **invariants** that hold after the constructor has finished, and before and after each public method call
- Throw `IllegalStateException` if the class invariant does not hold (often a sanity check)

```
public class CircularLinkedList {  
    // invariant: !isEmpty() -> list is circular  
    ...  
}
```

```
public class Date {  
    private int day; // invariant: 1 <= day <= 31  
    private int month; // invariant: 1 <= month <= 12  
  
    /**  
     * Sets the day.  
     *  
     * @param day New day, PRECOND: 1 <= day <= 31  
     */  
    public void setDay(int day) {  
        if (day < 1 || day > 31) {  
            throw new IllegalArgumentException("precond"  
                + " violation: day not valid");  
        }  
        this.day = day;  
    }  
    ...  
}
```

- Re-structure code without altering functionality
- Unit tests crucial
- Rename field/method, extract class, extract variable, convert local variable to field, inline variable, change method signature, move method, move field
- Pull up/push down methods in class hierarchy, extract interface
- Superb support in Eclipse

Method overloading

- Single method can have different implementations with different parameters. e.g.

```
public String() // constructs an empty string  
public String(char[] value)  
    // constructs a string with contents
```

- The constructor is **overloaded** (note: constructor name is fixed, otherwise only 1 way to construct an object)
- Overloading is defined by the method name and parameters (not including exceptions or the return value!)

- `final` keyword declares that the value of the variable cannot be re-set

```
final int x = 2;  
x = 3; // error
```

```
final Student s = new Student("tommi" );  
s.setName("tommi2" ); // ok  
s = new Student("tommi3" ); // error
```

Static variables and methods

- In OOP, most method calls are bound to an object
- `static` allows to create variables and methods that exist statically, i.e. can be called without an object

```
public class Math {  
    ...  
    public static final double PI = 3.141592654;  
    ...  
    public static double abs(double x) { ... }  
    ...  
}
```