3. homework assignment; JAVA, Academic year 2013/2014; FER

First: read page 12. I mean it! You are back? OK. This homework consists of four problems.

As a preparation for this homework, read in book about, and install the following tools: ant, CheckStyle, PMD, FindBugs, Junit, JaCoCo. Create a single build.xml capable of calling all of them, as described at the end of this homework.

Problem 1.

We are building a simple library for Digital logic. We will start with a definition of a legal values for a Boolean function. These are:

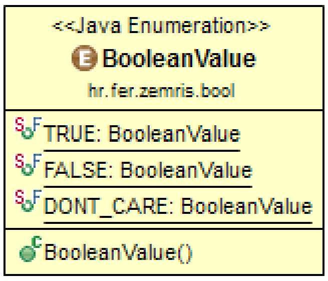
- logical value zero
- logical value one
- logical value don't care

We will treat *don't care* as a situation in which actual value of function is either zero or one, but we do not care about its actual value. When defining boolean algebra over this three-valued set, how will you define what is the result of logical OR between one and don't care? Do likewise for logical AND and logical NOT.

We will use Java's enum construct to define legal values. Please read about enums:

http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html

Define it as illustrated in following diagram.



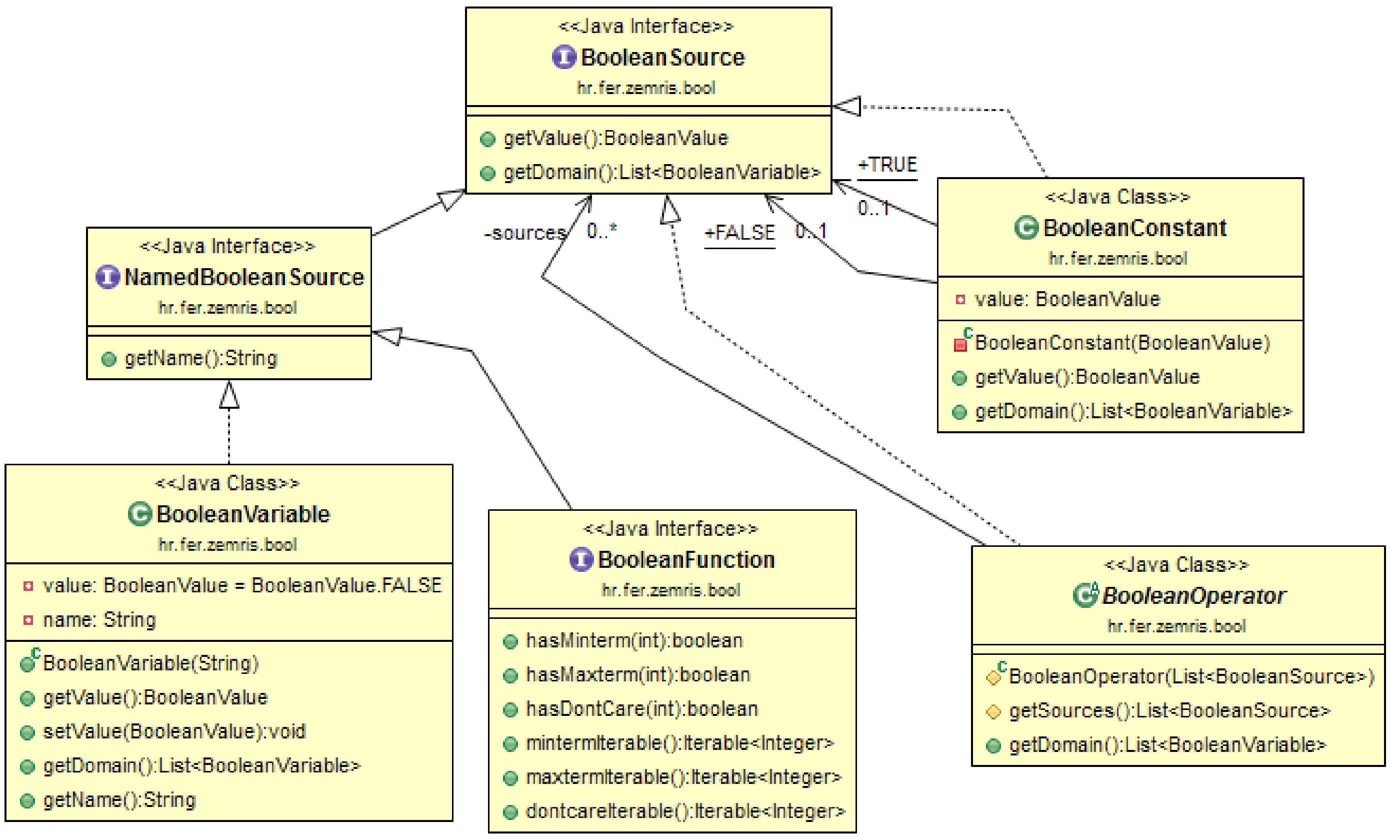
Now we will start with definition of several interfaces and classes as show on following diagram.

Interface `BooleanSource` represents any source capable of producing legal `BooleanValue`. It declares that it is capable of producing this value (method `getValue()`) and that it can provide an information based on which variables is this value produced (method `getDomain()`).

Class `BooleanConstant` is its first implementation: it has a private `value` which is populated through it only *private* constructor and it offers public getter for it. It is an unmodifiable instance. It has two public final static class instances denoted `TRUE` and `FALSE` which are statically initialized to hold an instance of constant always providing value `TRUE` and an instance of constant always providing value `FALSE`. Thanks to this design, you can write something like this:

```
BooleanSource tautology = BooleanConstant.TRUE;
```

The domain for instances of `BooleanConstant` is always an empty collection.



Interface `NamedBooleanSource` represents a source with associated name.

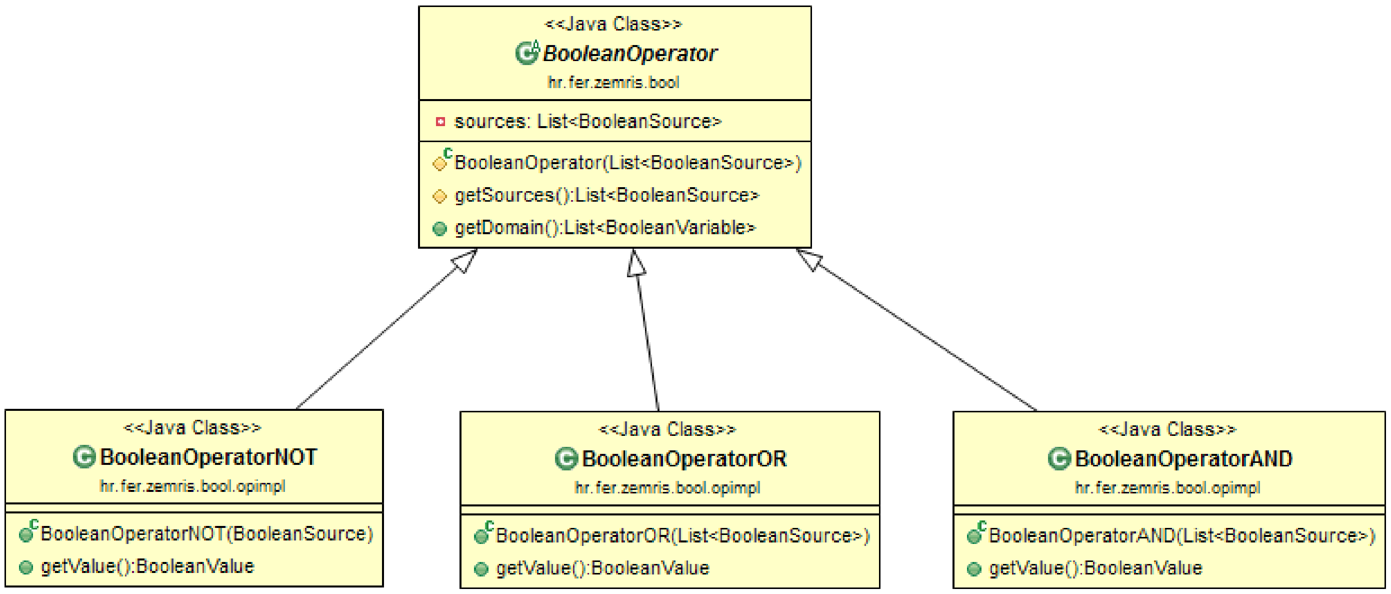
Class `BooleanVariable` is an implementation of `NamedBooleanSource`. It has a single public constructor which accepts name; by default, variable's value is set to `FALSE`. It offers getter and setter for variable's value. Variable's domain is a collection containing the variable itself.

Interface `BooleanFunction` represents an abstract boolean function. It declares methods like hasMinterm, hasMaxterm and hasDontCare; they accept a single argument: index, and check if function contains appropriate minterm/maxterm/dontcare. Please note that index uniquely specifies the values for domain's variables. For example, if domain is list {A,B,C}, index 3, binary 011, represents value assignment A=0, B=1, C=1. However, if domain is list {B,C,A}, index 3, binary 011, represents value assignment B=0, C=1, A=1. Order of variables, as returned by `getDomain()` must be used when interpreting value assignment inferred from numerical index. We say that function contains minterm *i* if for inferred value assignment function value is `TRUE`, it contains maxterm *i* if for inferred value assignment function value is `FALSE`, and it contains dontcare *i* if for inferred value assignment function value is `DONT_CARE`. Regarding these functions, they partition index space in three non-overlapping (disjoint) sets, which union is total index space (e.g. for functions of three variables, total index space is set {0, 1, 2, 3, 4, 5, 6, 7}). BooleanFunction also has three methods which ensures that we can use it for iterations:

```
BooleanFunction f = …;
for(Integer i : f.mintermIterable()) {
  System.out.println("I contain minterm: " + i);
}
```

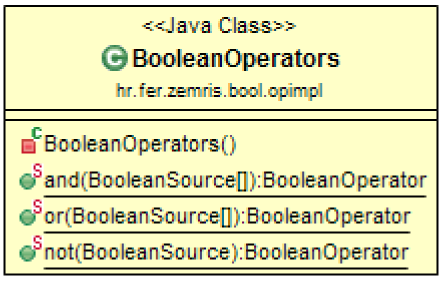
Finally, BooleanOperator is an abstraction over boolean operators. It has private list of sources based on which final result is calculated; this list must be provided through BooleanOperator's *protected* constructor as its only argument. Operators determine their domain by inspecting domains of given sources and producing an union (here, the ordering of boolean variables is not important).

We have three implementations of operators, as indicated in following diagram.



<<Java Class>>
BooleanOperator
hr.fer.zemris.bool
sources: List<BooleanSource>
BooleanOperator(List<BooleanSource>)
getSources():List<BooleanSource>
getDomain():List<BooleanVariable>
<<Java Class>>
BooleanOperatorNOT
hr.fer.zemris.bool.opimpl
BooleanOperatorNOT(BooleanSource)
getValue():BooleanValue
<<Java Class>>
BooleanOperatorOR
hr.fer.zemris.bool.opimpl
BooleanOperatorOR(List<BooleanSource>)
getValue():BooleanValue
<<Java Class>>
BooleanOperatorAND
hr.fer.zemris.bool.opimpl
BooleanOperatorAND(List<BooleanSource>)
getValue():BooleanValue

Please also implement a class providing convenient factory methods for producing concrete instances:



<<Java Class>>
BooleanOperators
hr.fer.zemris.bool.opimpl
BooleanOperators()
and(BooleanSource[]):BooleanOperator
or(BooleanSource[]):BooleanOperator
not(BooleanSource):BooleanOperator

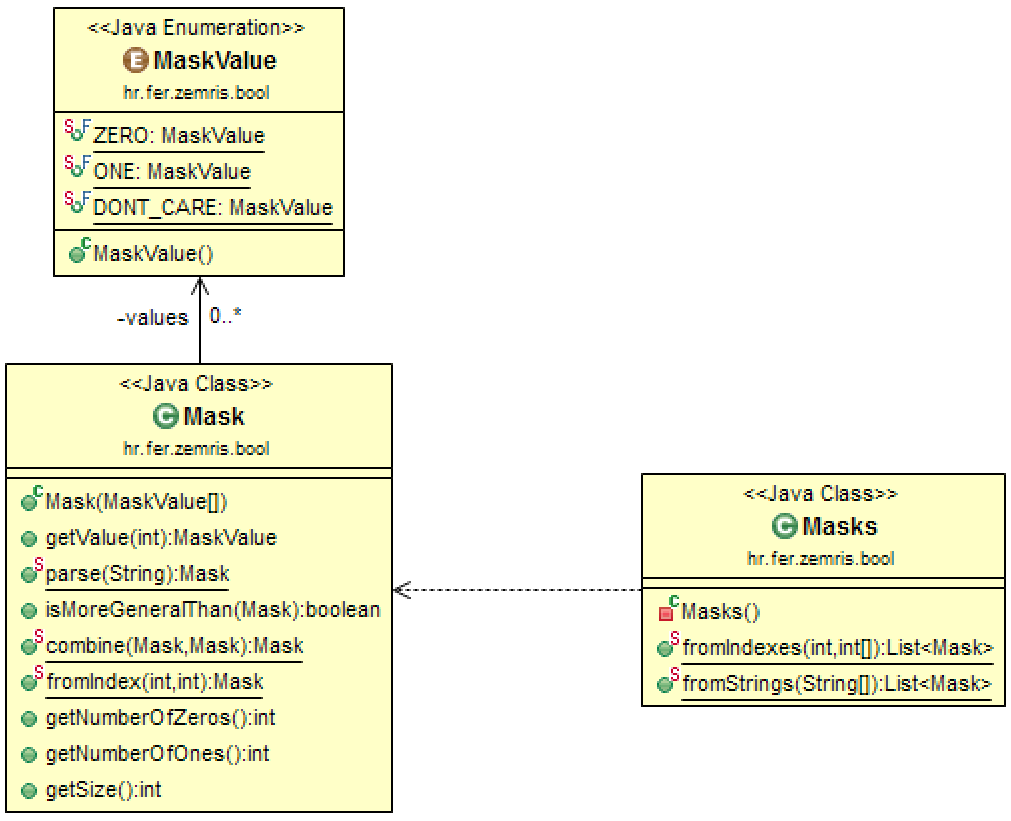
The prototype for factory methods `and` and `or` should be:

```
public static BooleanOperator and(BooleanSource ... sources);
public static BooleanOperator or(BooleanSource ... sources);
```

To support definition of Boolean functions as sums of (partial) products and as products of (partial) sums, we will define a concept of *mask* (something that should be familiar to you from Quine-McCluskey minimization algorithm). For example, consider a Boolean function defined over variables A, B, C and D (i.e. its domain is {A, B, C, D}). Mask “0x1x” represents all value assignments in which A=0 and C=1 (we don't care for B and D); here there are for such assignments which we can specify by more specific masks: “0010”, “0011”, “0110” and “0111”. We will define one enum and two classes (see following diagram).

Distinct values for each variable in mask will be modeled as enum `MaskValue`. Instances of class `Mask` represent a single mask. Each instance stores mask as an array of `MaskValue`. Class Mask offers a single public constructor which accepts an array of `MaskValue`. It also offers two public static factory methods: `parse` which accepts a string representation of a mask and `fromIndex` which accepts two arguments: mask size and index. The idea is that you can write the following (both examples generate the same mask):

```
Mask m1 = new Mask(
  new MaskValue[] {
    MaskValue.ZERO, MaskValue.DONT_CARE, MaskValue.ONE, MaskValue.DONT_CASE
  }
);
Mask m2 = Mask.parse("0x1x");
```



Using `fromIndex`, you can specify masks which do not contain don't cares. For example:

```
Mask m3 = Mask.fromIndex(3, 1);
```

creates mask of size 3 whose string representation is "001".

```
Mask m4 = Mask.fromIndex(5, 11);
```

creates mask of size 5 whose string representation is "01011".

Instance method `getValue` accepts an index and returns `MaskValue` at requested position. For example, using previous example, `m3.getValue(2)` would return `MaskValue.ONE`.

Class Masks provides a static factory methods for creation of multiple masks in a single method call. For example:

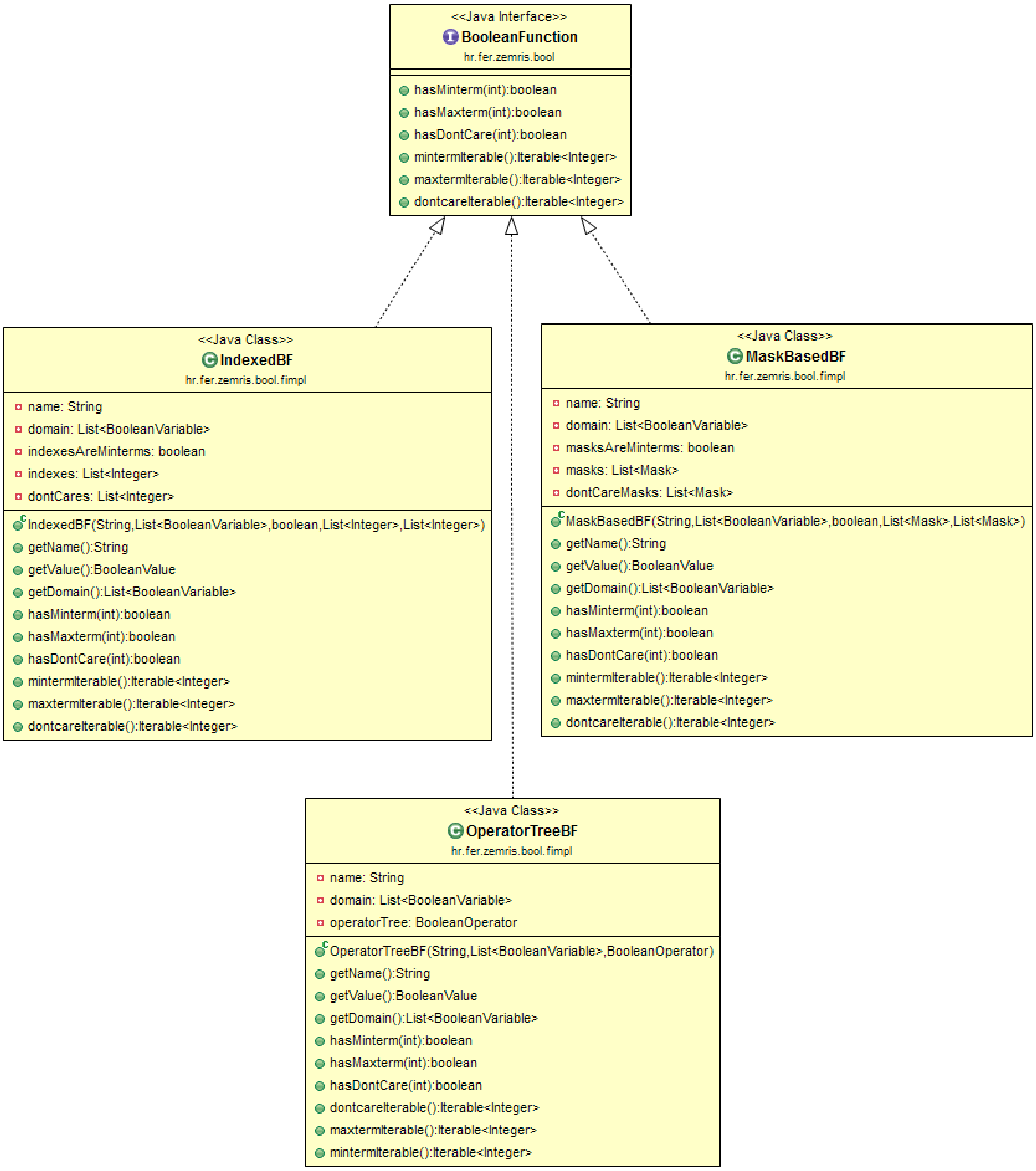
```
List<Mask> masks = Masks.fromStrings("0x1x", "1110", "x00x");
List<Mask> masks = Masks.fromIndexes(3, 0, 1, 7);
```

The latter call produces masks of size 3: "000", "001", "111".

Please note that although method prototypes rendered in diagram depicts arrays, the idea is that you declare metods as methods with variable number of arguments (read in book about it / google it).

The method `Mask.combine` generates a new mask if this is possible. For example, you can combine "0x01" and "0x00" into "0x0x" because expanding "0x0x" would produce exactly "0x01" and "0x00" and nothing more. If given masks are not combineable, method must return `null` unless there is something wrong with masks (one is `null`, or they are of different sizes) – in this case throw an exception.

Finally, you must support three implementations of Boolean functions, as depicted in following diagram.



<<Java Interface>>
BooleanFunction
hr.fer.zemris.bool
hasMinterm(int):boolean
hasMaxterm(int):boolean
hasDontCare(int):boolean
mintermIterable():Iterable<Integer>
maxtermIterable():Iterable<Integer>
dontcareIterable():Iterable<Integer>
<<Java Class>>
IndexedBF
hr.fer.zemris.bool.fimpl
name: String
domain: List<BooleanVariable>
indexesAreMinterms: boolean
indexes: List<Integer>
dontCares: List<Integer>
IndexedBF(String,List<BooleanVariable>,boolean,List<Integer>,List<Integer>)
getName():String
getValue():BooleanValue
getDomain():List<BooleanVariable>
hasMinterm(int):boolean
hasMaxterm(int):boolean
hasDontCare(int):boolean
mintermIterable():Iterable<Integer>
maxtermIterable():Iterable<Integer>
dontcareIterable():Iterable<Integer>
<<Java Class>>
MaskBasedBF
hr.fer.zemris.bool.fimpl
name: String
domain: List<BooleanVariable>
masksAreMinterms: boolean
masks: List<Mask>
dontCareMasks: List<Mask>
MaskBasedBF(String,List<BooleanVariable>,boolean,List<Mask>,List<Mask>)
getName():String
getValue():BooleanValue
getDomain():List<BooleanVariable>
hasMinterm(int):boolean
hasMaxterm(int):boolean
hasDontCare(int):boolean
mintermIterable():Iterable<Integer>
maxtermIterable():Iterable<Integer>
dontcareIterable():Iterable<Integer>
<<Java Class>>
OperatorTreeBF
hr.fer.zemris.bool.fimpl
name: String
domain: List<BooleanVariable>
operatorTree: BooleanOperator
OperatorTreeBF(String,List<BooleanVariable>,BooleanOperator)
getName():String
getValue():BooleanValue
getDomain():List<BooleanVariable>
hasMinterm(int):boolean
hasMaxterm(int):boolean
hasDontCare(int):boolean
dontcareIterable():Iterable<Integer>
maxtermIterable():Iterable<Integer>
mintermIterable():Iterable<Integer>

Class `IndexedBF` represents a boolean function which is defined by specifying indexes of minterms (or maxterms) and indexes of don't cares. The idea is to enable user to specify functions formally defined as:

$$f(A,B,C)=\sum m(1,2,5)+\sum d(0,4,7)$$

$$g(A,B,C)=\prod M(3,6)\cdot\prod d(0,4,7)$$

Observe that these two functions are equal. It's constructor has prototype:

```
public IndexedBF(String name, List<BooleanVariable> domain, boolean indexesAreMinterms,
                 List<Integer> indexes, List<Integer> dontCares);
```

Class `MaskedBasedBF` represents a boolean function which is defined by specifying minterm/maxterm and don't care masks. The prototype of its constructor is:

```
public MaskBasedBF(String name, List<BooleanVariable> domain, boolean masksAreMinterms,
                          List<Mask> masks, List<Mask> dontCareMasks);
```

Here is an example (assume domain is {A,B,C,D}).

```
BooleanFunction bf = new MaskBasedBF(
    "f",
    domain,
    true,
    Masks.fromStrings("0x00","000x"),
    Masks.fromStrings("10x0")
);
```

The function specified in previous example is:

$$f(A,B,C,D)=\sum m(0,1,4)+\sum d(8,10)$$

Finally, the class `OperatorTreeBF` allows us to construct a boolean function by specifying an operator tree. It has a single constructor with prototype:

```
public OperatorTreeBF(String name, List<BooleanVariable> domain, BooleanOperator operatorTree)
;
```

Please note that you can determine the actual boolean variables on which operator tree depends by calling appropriate function. In constructor, you must provide a list of variables on which you wish this boolean function to declare its dependency. Having this in mind, you can always declare that you have boolean function *f(A,B,C) = A AND B*. The domain for operator tree *A AND B* would be {A, B} (or {B, A}, the order here is not important). The function declares that it depends on superset of this domain and specifies variable ordering which is to be used for resolving value assignments from numerical indexes. However, it illegal to declare that function does not depend on some variable on which operator tree depends: in this case constructor must throw an appropriate exception.

In order to implement minterm/maxterm/dontcare iterators for this function representation, please note that you can use function's domain variables, set them during iteration to every legal value (try assigning to each variable either `BooleanValue.TRUE` or `BooleanValue.FALSE`) and compute the result produced by operator tree. Since you will need this functionality for different iterators (minterms, maxterms, dontcares), try to refactor code so that you do not have code duplications.

Here are three simple examples.

Example 1. Constructing operator tree.

```java
package test;

import hr.fer.zemris.bool.BooleanConstant;
import hr.fer.zemris.bool.BooleanFunction;
import hr.fer.zemris.bool.BooleanOperator;
import hr.fer.zemris.bool.BooleanVariable;
import hr.fer.zemris.bool.fimpl.OperatorTreeBF;
import hr.fer.zemris.bool.opimpl.BooleanOperators;

import java.util.Arrays;

public class Primjer1 {

	public static void main(String[] args) {

		BooleanVariable varA = new BooleanVariable("A");
		BooleanVariable varB = new BooleanVariable("B");
		BooleanVariable varC = new BooleanVariable("C");

		BooleanOperator izraz1 = BooleanOperators.or(
			BooleanConstant.FALSE,
			varC,
			BooleanOperators.and(varA, BooleanOperators.not(varB))
		);

		BooleanFunction f1 = new OperatorTreeBF(
			"f1",
			Arrays.asList(varA, varB, varC),
			izraz1);

		for(Integer i : f1.mintermIterable()) { // Ispis: 1, 3, 4, 5, 7
			System.out.println("Imam minterm: "+i);
		}
		for(Integer i : f1.maxtermIterable()) { // Ispis: 0, 2, 6
			System.out.println("Imam maxterm: "+i);
		}
		for(Integer i : f1.dontcareIterable()) { // Ispis:
			System.out.println("Imam dontcare: "+i);
		}

	}

}
```

Example 2. Specifying function as sum of minterms.

```java
package test;

import hr.fer.zemris.bool.BooleanFunction;
import hr.fer.zemris.bool.BooleanVariable;
import hr.fer.zemris.bool.fimpl.IndexedBF;

import java.util.Arrays;

public class Primjer2 {

	public static void main(String[] args) {

		BooleanVariable varA = new BooleanVariable("A");
		BooleanVariable varB = new BooleanVariable("B");
		BooleanVariable varC = new BooleanVariable("C");

		BooleanFunction f1 = new IndexedBF(
			"f1",
			Arrays.asList(varA, varB, varC),
			true,
			Arrays.asList(0,1,5,7),
			Arrays.asList(2,3)
		);

		for(Integer i : f1.mintermIterable()) { // Ispis: 0, 1, 5, 7
			System.out.println("Imam minterm: "+i);
		}
		for(Integer i : f1.maxtermIterable()) { // Ispis: 4, 6
			System.out.println("Imam maxterm: "+i);
		}
		for(Integer i : f1.dontcareIterable()) { // 2, 3
			System.out.println("Imam dontcare: "+i);
		}

	}

}
```

Example 3. Specifying function using masks.

```java
package test;

import hr.fer.zemris.bool.BooleanFunction;
import hr.fer.zemris.bool.BooleanVariable;
import hr.fer.zemris.bool.Masks;
import hr.fer.zemris.bool.fimpl.MaskBasedBF;

import java.util.Arrays;

public class Primjer3 {

	public static void main(String[] args) {

		BooleanVariable varA = new BooleanVariable("A");
		BooleanVariable varB = new BooleanVariable("B");
		BooleanVariable varC = new BooleanVariable("C");
		BooleanVariable varD = new BooleanVariable("D");

		BooleanFunction f1 = new MaskBasedBF(
			"f1",
			Arrays.asList(varA, varB, varC, varD),
			true,
			Masks.fromStrings("00x0", "1xx1"),
			Masks.fromStrings("10x0")
		);

		for(Integer i : f1.mintermIterable()) { // Ispis: 0, 2, 9, 11, 13, 15
			System.out.println("Imam minterm: "+i);
		}
		for(Integer i : f1.maxtermIterable()) { // Ispis: 1, 3, 4, 5, 6, 7, 12, 14
			System.out.println("Imam maxterm: "+i);
		}
		for(Integer i : f1.dontcareIterable()) { // Ispis: 8, 10
			System.out.println("Imam dontcare: "+i);
		}

	}

}
```

Problem 2.

Open PowerPoint presentation `java_tecaj_03_prezentacija.pdf`. Solve the problem from slide 37. Put the implementation class in package `hr.fer.zemris.java.tecaj.hw4`, and name it `AboveAverage`.

Problem 3.

Open PowerPoint presentation `java_tecaj_03_prezentacija.pdf`. Solve the problem from slide 43. Put the implementation class in package `hr.fer.zemris.java.tecaj.hw4`, and name it `NamesCounter`.

Problem 4.

Write a simple database emulator. Put the implementation classes in package `hr.fer.zemris.java.tecaj.hw4.db`. In repository on Ferko you will find a file named `database.txt`. It is a simple textual form in which each row contains data on single student. Atributes are: *jmbag*, *lastName*, *firstName*, *finalGrade*. Name your program `StudentDB`. When started, program reads this file from current directory from file `database.txt`. In order to achieve this, write a class `StudentRecord`; instances of this class will represent records for each student. Assume that there can not exist multiple records for the same student. Implement equals and hashCode methods so that two students are equal if jmbags are equal.

Write a class `StudentDatabase`: its constructor must get a list of `String` (the content of `database.txt`). It must create an internal list of student records. Additionally, it must create an index for fast retrieval of student records when jmbag is known (use map for this). Add the following two public methods in this class as well:

```
public StudentRecord forJMBAG(String jmbag);
public List<StudentRecord> filter(IFilter filter);
```

The first method uses index to obtain requested record in O(1); if record does not exists, the method returns `null`.

The second method accepts a reference to an object which is an instance of IFilter interface:

```
public interface IFilter {
   public boolean accepts(StudentRecord record);
}
```

The method `filter` in `StudentDatabase` loops through all student records in its internal list; it calls `accepts` method on given object and current record; each record for which `accepts` returns **`true`** is added to temporary list and this list is then returned by the `filter` method.

The system reads user input from console. You must support a single command: **query**, which comes in two forms. First form allows user to specify JMBAG and writes a record for this JMBAG. The second form allows user to specify last name (or part of it using wildcard `*`; wildcard character, if present, can occur at most once, but it can be at the beginning, at the end or somewhere in the middle).

Create a class `LastNameFilter` which implements `IFilter`. It has a single public constructor which receives one argument: mask (i.e. last name pattern). Implement the second form of query command using this class. Here is an example of interaction with program, as well as expected output and formatting.

Simbol for prompt which program writes out is “> ”.

```
> query jmbag="0000000003"
+============+========+========+===+
| 0000000003 | Bosnić | Andrea | 4 |
+============+========+========+===+
Records selected: 1

> query lastName="B*"
+============+===========+===========+===+
| 0000000002 | Bakamović | Petra     | 3 |
| 0000000003 | Bosnić    | Andrea    | 4 |
| 0000000004 | Božić     | Marin     | 5 |
| 0000000005 | Brezović  | Jusufadis | 2 |
+============+===========+===========+===+
Records selected: 4

> query lastName="Be*"
Records selected: 0
```

Please observe how the table is automatically resized and how columns are aligned using spaces. The order in which records are the same in which they are given in database file.

If users input is invalid, write an appropriate message. Allow user to write spaces: following is also OK:

```
query     lastName="Be*"
query lastName    ="Be*"
query lastName=   "Be*"
query lastName  =    "Be*"
```

Everything else is illegal. You **do not** have to support any more complex questions, such:

```
query lastName="Be*" or (lastName != "*e" and firstName = "Ivan")
```

Note: for reading from file please use the code snippet which I have explained during lectures (repeated below). In this example, we are reading the content of `prva.txt` located in current directory:

```
List<String> lines = Files.readAllLines(
    Paths.get("./prva.txt"),
    StandardCharsets.UTF_8
);
```

Important notes

Solve all of the problems in a single Eclipse project. Configure Eclipse to use two source directories: `src/main/java` for your source files and `src/test/java` for sources files of unit tests.

You are required to write the adequate number of unit tests for all of the classes developed in problem 1.

You must equip your project with `build.xml` script so that the project can be build from the command line. In that script, you must integrate all of the quality-checking tools which I have described in the book. It should be possible to run at least the following targets: `init`, `compile`, `compile-tests`, `run-tests`, `quality`, `reports`, `clean`.

Target `quality` must run unit tests and all of the quality checks (i.e. *checkstyle*, *pmd*, *findbugs*). Unit tests must be run with the code coverage analysis. Target `reports` is a wrapper that will run all of the unit tests, the quality checks and the javadoc generation.

All of the classes in all three problems should have appropriate javadoc.

Configuring ant build script

In order to get *portable* build script, you should not define paths to additional tools directly in `build.xml` file (for example, variables such as `checkstyle.home`, `pmd.home` etc). Instead, create a new file `config.properties` in the same directory as `build.xml`. It must be a text file which defines all of the necessary home variables. Here is a sample content:

```
checkstyle.home=d:/usr/checkstyle-5.6
pmd.home=d:/usr/pmd-bin-5.0.2
findbugs.home=d:/usr/findbugs-2.0.2
junit.home=d:/usr/junit-4.11
jacoco.home=d:/usr/jacoco-0.6.3
xalan.home=d:/usr/xalan-j_2_7_1
```

Then delete in `build.xml` declarations for those variables (*and only that!*) and just add the following line:

```
<property file="config.properties"/>
```

Now, *`ant`* will read file `config.properties` and will define variables as defined in that file.

Please note. You can consult with your peers and exchange ideas about this homework *before* you start actual coding. Once you open you IDE and start coding, consultations with others (except with me) will be regarded as cheating. You can not use any of preexisting code or libraries which is not part of Java standard edition (Java SE) unless explicitly provided by me. This means that from this point on, you can use Java Collection Framework classes or its derivatives (moreover, I recommend it). Document your code!

In order to solve this homework, create a blank Eclipse Java Project and write your code inside. You must name your project's main directory (which is usually also the project name) `HW04-`*yourJMBAG*; for example, if your JMBAG is 0012345678, the project name and the directory name must be `HW04-0012345678`. Once you are done, export the project as a ZIP archive and upload this archive to Ferko before the deadline. Do not forget to lock your upload or upload will not be accepted. Deadline is April 7th 2014. at Noon.