Block 1

Block 1

1-OV Overview

1-OV.1 Contents of This Block

CP This block will briefly recapitulate the main constructs to write concurrent Java programs, as discussed during Module 1.2 (Software systems). Further, it will discuss the interleaving behaviour of concurrent programs, the meaning of thread safety, and different ways to share data between threads.

FP This block will introduce some fundamental concepts and demonstrate some elementary examples of functional programs. Some of these concepts are: function, definition, currying, recursion, types, polymorphism.

CC This block will introduce the architecture of a compiler, and how to separate a piece of text into words and symbols (scanning). You will also meet ANTLR, the tool used in this course to automatically generate scanners and parsers.

1-OV.2 Mandatory Activities

The sign-off exercises of this block should be completed (and signed off) in the course of the block itself. If you are not finished, make sure that your solutions are ready to be signed off during the first lab session of block 2.

1-OV.3 Expected Self-Study and Project Work

In addition to the scheduled activities, we estimate that you need approximately:

- 6 hours self-study to prepare the CP exercises;
- 2 hours self-study to read through the CC material and install ANTLR.

1-OV.4 Materials for this Block

- **CP**, from Module 1.2 (Software Systems): study Chapter 14, until *BlockingQueues* (p. 819 - 877) from C.S. Horstmann and G. Cornell, *Core Java, volume I: Fundamentals*, Prentice Hall, 9th edition (2012) again.

- **CP**, from JCIP: Chapters 1 and 3.
- **CC**, from EC: Chapters 1 (completely) and 2 (except for 2.4.1–2.4.4 and 2.6).

1-CP Concurrent Programming

Before this first exercise session, you should try to solve a collection of self-study exercises. You are asked to develop complete Java test programs, run these programas, and analyse the results of your tests. During the plenary exercise session, you will be asked to present your solutions, and the different solutions will be discussed in the group.

The objective of this first set of exercises is twofold. Firstly, we want you to brush up your Java knowledge with some simple multi-threaded Java applications. Secondly, you will discover that concurrency bugs are not just theory, but easily happen in practice.

Note that several exercises ask you to show that something is not thread safe. We recommend that you do not develop the entire test from scratch each time, but instead develop a reusable test class for concurrent structures and methods. In this way you do not have to write the boilerplate code, e.g., for spawning and joining threads, every time.

Unsafe Sequence. The goal of the first exercise is to get more insight in the possible behaviours a concurrent program can have. You are asked to implement a class that is not thread-safe, and to demonstrate that it is indeed not thread-safe by analysing the different behaviours.

Exercise 1-CP.1

1. Take the non-thread-safe sequence generator from Listing 1.1 (JCIP, page 6). Add a test that allows you to demonstrate that it is indeed *not* thread-safe. Due to the randomness of the thread interleaving, different runs will probably yield different results.
2. Take the output of one of your test runs and show where thread safety is violated. Give an interleaving that could have led to the behaviour you observed.
3. Develop different alternatives to make the class thread safe and show that these implementations are indeed working properly.

Hint: Even though the sequence generator is not thread-safe, your *test class* should be thread-safe in order to get valid results. □

Producer-Consumer Pipelines. Queues in their various different forms are essential building blocks for multithreaded processing pipelines. In order to achieve maximal throughput it is important that the queue allows for multiple producers to add messages to the queue, and for multiple consumers to process the messages, each running possibly in its own thread. Standard examples of queues are:

- stack (FILO, First In Last Out)
- buffer (FIFO, First In First Out)
- priority queue (the message with the highest priority gets consumed first)

Exercise 1-CP.2 In this exercise you are asked to investigate how such a producer-consumer pipeline can be used safely in a concurrent environment. Create your queue (or buffer) adhering to the following simple interface.

```
public interface Queue<T> {
    /** Push an element at the head of the queue. */
    public void push(T x);

    /** Obtain and remove the tail of the queue. */
    public T pull() throws QueueEmptyException;

    /** Returns the number of elements in the queue. */
    public int getLength();
}
```

1. Implement a queue, based on a linked list (i.e., a node contains a reference to the next node) without worrying about concurrency considerations.
2. Implement a test class and document the events that cause your data structure to fail for multiple consumers and producers.
   Hint: Again, make sure your test class is thread safe in order to obtain valid results.
3. Change your implementation to make it thread safe and show that your queue passes the tests from assignment 2.
4. Are there ways to further improve your implementation, using finer-grained concurrency, to improve concurrency? If possible, adapt your implementation accordingly.
5. Due to the randomness of thread interleaving it is *impossible* to prove the correctness of the synchronisation by merely running a test. Provide an informal proof why your synchronisation method achieves correctness.

□

Thread Safety. It is often hard to see from documentation whether a class is thread safe and under what conditions.

Exercise 1-CP.3 For this exercise we will analyse the class `java.util.Timer`. Before proceeding, carefully read the javadoc documentation of `java.util.Timer`.

1. Create a program which has a *single* `Timer` object with n `TimerTask`s associated. Each of these `TimerTask` objects should trigger at the exact same moment t_0. Use `Timer`'s `schedule` method to start the tasks at the same moment t_0 (or after a certain delay). In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTasks` objects. After all tasks have finished, check the number of updates to the variable.
2. Create a program which has a n `Timer` objects which trigger at the exact same moment t_0. Each of theses n `Timer` objects gets a single `TimerTask` object which is triggered at t_0. In the `run` method of the `TimerTask` objects you should increment a variable which is shared by all `TimerTask` objects. Again, after all tasks have finished, check the number of updates to the variable.
3. What do you observe? Under what conditions does everything work out and under what conditions does it go wrong? Based on your observations, what do you conclude about the implementation of `java.util.Timer`?

□

Exercise 1-CP.4 Paragraph 4.5 of JCIP (page 75) mentions that `java.text.SimpleDateFormat` is not thread safe (and so does the javadoc of `SimpleDateFormat`). Show that this is indeed the case.
Hint: Breaking things is usually easier than guaranteeing their correctness. Create some threads which share a `java.text.SimpleDateFormat` object. Let each thread update the shared object to random (but correct!) dates using the method `parse`. □

Assertions and Concurrency. Concurrent execution also has an impact on the claims that you can make about the behaviour of a program. As discussed in Section 3.5.2 of JCIP, concurrency can make assertions invalid, even though from a sequential point of view they would be considered obviously valid.

JML specifications also can be considered as assertions: every precondition translates to an assert at the beginning of the method; every postcondition translates to an assert at the end of the method.

Exercise 1-CP.5 Consider the JML-annotated class `Point`.

```
package pp.block1.cp.annotation;

public class Point {
  /*@ spec_public */ private int x;
  /*@ spec_public */ private int y;
```

```
    //@ ensures \result >= 0;
    /*@ pure */public int getX() {
        return x;
    }

    //@ ensures \result >= 0;
    /*@ pure */public int getY() {
        return y;
    }

    /*@
      requires n >= 0;
      ensures getX() == \old(getX()) + n;
     */
    public void moveX(int n) {
        x = x + n;
    }

    /*@
      requires n >= 0;
      ensures getY() == \old(getY()) + n;
     */
    public void moveY(int n) {
        y = y + n;
    }

}
```

This class `Point` is used in a class `RandomDrift`, which moves the point up and to the right with random steps.

```
package pp.block1.cp.annotation;

public class RandomDrift extends Thread {

    private Point p;

    public RandomDrift(Point p) {
        this.p = p;
    }

    public void run() {
        while (true) {
            int n = (int) (Math.random() * 10);
            p.moveX(n);
            int m = (int) (Math.random() * 10);
            p.moveY(m);
        }
    }
}
```

1. Class `RandomPoint` starts two `RandomDrift` threads, using the same instance of `Point`. Describe the state space of this program.
2. Discuss where JML specifications might become violated because of concurrency aspects. Show an example execution that violates the specifications.
3. Suppose the methods `moveX` and `moveY` in `Point` become synchronized. Would this solve the problem? Argue why, or why not.

□

1-FP Functional Programming

Exercise 1-FP.1 Introduction-exercise. Type in the next function

$$f\,x \quad = \quad 2x^2 + 3x - 5$$

and evaluate it for a few values of x. □

Remark. From now on every function definition should be preceded by mentioning its type.

Exercise 1-FP.2 Import the module *Data.Char*:

import *Data.Char*

Now the following functions are avaiable:

ord :: *Char* → *Int*
chr :: *Int* → *Char*

which translate a character into its code and back.

- Use *ord* and *char* to define a function *code* which changes a letter (including capital letters) by cyclicly shifting it three positions further in the alphabet, i.e., after 'z' one should continue with 'a' again.
  For example:

  code `'a'` = `'d'`
  code `'P'` = `'S'`
  code `'y'` = `'b'`

  The function *code* should leave all other characters (digits, spaces, etc.) unchanged. Hint: the relations $<, \leqslant, \geqslant, >$ also work for characters.
- Evaluate the expressions

  map code `"hello"`
  map code `"Tomorrow evening, 8 o'clock in Amsterdam"`

  (*map* applies the function *code* to all characters in a string).
- Generalise your function *code* in such a way that it can be given a number n as additional argument, which indicates how many positions the character has to be shifted (note that above $n = 3$). Note that the order in which the arguments are given to *code* is relevant for using the function *map* with the generalised coding function.

□

Exercise 1-FP.3 Define *recursively* a function *interest* which calculates how much money you have after n years, if you start with an amount a and receive r percent of interest per year. You have to take into account that you will have "interest over interest", where the interest only has to be computed once per year. □

Exercise 1-FP.4 Define two functions *root1* and *root2* which determine the roots of a quadratic equation of the form

$$ax^2 + bx + c = 0$$

where a, b, c are given, and $a \neq 0$. If the discriminant is negative, your function should give an error, to be defined as follows:

error `"negative discriminant"`

Write a function *discr* which calculates the discriminant and which can be used in the functions *root1* and *root2*.

Test your funktions for a number of values of a, b, c. □

Exercise 1-FP.5 A second order polynome is an expression of the form

$$ax^2 + bx + c$$

(assume $a \neq 0$).

1. Write a function *extrX* which calculates the value of x at which the polynome has its extreme value.
2. Write a function *extrY* which calculates this extreme value.

□

Exercise 1-FP.6 Write recursive definitions for the following functions on lists (give the type for every function you define):

1. *mylength* for the length of a list,
2. *mysum* which adds the elements in a list of numbers,
3. *myreverse* which reverses the order of the elements in a list,
4. *mytake* which gives the first n elements of a list (in case n is greater than the length of a list, the whole list should be delivered),
5. *myelem* which determines whether a given element is in a list,
6. *myconcat* which glues together a list of lists into one long list,
7. *mymaximum* which yields the maximum of a list of numbers,
8. *myzip* which transforms two lists into a list of pairs (the shortest of the two lists determines the length of the resulting list).

□

Exercise 1-FP.7 A sequence of numbers is *arithmetic* if, starting from some initial number a, every next number in the sequence can be determined from the previous number by adding a fixed difference d.

1. Write a recursive function r which generates the arithmetic sequence starting with a, and using difference d. Note that the type of r is:

   $$r :: Num\ a \Rightarrow a \rightarrow a \rightarrow [a]$$

2. Write a function *r1* which selects the n-th number from the sequence above (hint: use the function r),
3. Let i and j be two indices. Write a function *total* which calculates the sum of the i-th element upto (and included) the j-th element.

□

Exercise 1-FP.8

1. Write a function *allEqual* which determines whether all elements in a list are equal.
2. Write a function `isAS` which checks whether a sequence is arithmetical (hint: use the function *allEqual*).

□

Exercise 1-FP.9 A matrix can be defined as a list of lists of numbers, where the inner lists are the rows of the matrix. Write for the following cases a function which:

1. checks whether all rows in a matrix are equally long,
2. yields the list of totals of every row in a matrix,
3. transposes a matrix, i.e., every n-th row is transofrmed into the n-th column,
4. yields the list of totals of every column in a matrix.

□

1-CC Compiler Construction

1-CC.1 Chapter 1: Overview of Compilation

Exercise 1-CC.1 Compile for yourself a list of the following terms, with short definitions for each of them. Don't just copy from the book: make sure you understand your own definitions.

> *back end, front end, grammar, instruction scheduling, instruction selection, optimizer, parsing, register allocation, scanning, type checking*

□

Exercise 1-CC.2 Answer the following questions.

1. Using the grammar on Page 12 of EC, explain systematically, i.e., step by step, why the following string is an instance of the syntactic variable *Sentence*:
   "Most students is good programmers."
2. What is actually wrong with the above sentence? To what kind of programming error can you compare this? At what stage would a compiler detect it?
3. Which steps in the two sub-exercises above correspond to the following activities: *parsing*, *type checking*, *scanning*?

□

Exercise 1-CC.3 Extend the grammar on Page 12 of EC so that it can also cope with sentences of the form

> "Programming Paradigms is a diverse interesting module."
> "Programming Paradigms is a diverse interesting elective module."

without needing a new rule for every case where there are more successive adjectives. (A word like "a" is called a *particle* in English.) □

Exercise 1-CC.4 Consider the assignment

$$d \leftarrow d + 2 \times (a + b)$$

1. Manually carry out the (naive) instruction selection process informally described in §1.3.3 of the book (see Fig. 1.3) on this assignment.
2. Improve the sequence of instructions by minimizing the number of required registers
3. Improve your answer to the previous subquestion by rescheduling the sequence to a minimal execution time (where you may use more registers if that benefits the execution time). Compute the number of clock cycles of the original and the resulting schedule.

Give your answers in the form of tables such as the ones on Pages 17–19. □

1-CC.2 Chapter 2: Scanners

Exercise 1-CC.5 Answer Review Question 1 of Section 2.2 (Page 33), *with the proviso that the identifier should be exactly six characters*; i.e., for "zero to five alphanumeric characters" read "exactly five alphanumeric characters". □

Exercise 1-CC.6 Answer Review Question 2 of Section 2.3 (Page 42). Look for a concise RE; you may make use of all the notation introduced in the section. (The quotation mark, `"`, is an element of Σ.) □

Exercise 1-CC.7 Consider a language in with three different token kinds: (i) "`La`", (ii) "`La␣La`" and (iii) "`La␣La␣La␣Li`" (where the symbol ␣ denotes a space), with the proviso that

- Capitalisation must be as given: all `L`'s are upper case and all `a`'s and `i`'s lower case;
- The `a`'s may be repeated arbitrarily often (meaning there can be one or more), but the `i` may not; so `Laaaaaa` is allowed but not `Lii`;

- Every `La` and `Li` may be followed by zero or more spaces, *which are considered to be part of the token*.

Hence, for instance, "`LaaaaLaLaa␣Laaaa␣␣␣␣LaLiLaa`" is a valid input text; it consists of tokens "`LaaaaLa`", "`Laa␣Laaaa␣␣␣␣LaLi`" and "`Laa`".

1. Give regular expressions for the three token kinds of this language.
2. Give a single DFA that is able to recognise all three token kinds.
3. Answer the following questions, considering that scanning is *greedy*:
   - What is "`Laaaa LaLaa`" broken down into: "`Laaaa␣`" + "`LaLaa`", "`Laaaa␣La`" + "`Laa`" or "`Laaaa␣`" + "`La`" + "`Laa`"?
   - What is "`La␣La␣La␣La␣Li`" broken down into?

□

1-CC.3 Scanner implementation

Consider the following files, to be found on BLACKBOARD:

- `pp.block1.cc.dfa.State`. This is a straightforward implementation of a DFA — more precisely, a *state* of a DFA, but since all states are reachable from the initial state, it suffices to represent a DFA by its initial state. The class `State` includes a constant `DFA_ID6` that implements the 6-identifier scanner of Exercise 1-CC.5.
- `pp.block1.cc.dfa.Checker`. This is an interface offering the functionality of testing whether a given DFA accepts a given input text.
- `pp.block1.cc.test.CheckerTest`. This is a JUNIT test for an implementation of `Checker`.
- `pp.block1.cc.dfa.Scanner` This is an interface offering the functionality of applying a given DFA as a scanner to an input text.
- `pp.block1.cc.test.ScannerTest`. This is a JUNIT test for an implementation of `Scanner`.

Exercise 1-CC.8 Program an *efficient* implementation of `Checker`, and show it correct using `CheckerTest`. *Efficient* means that the execution time of your algorithm is *linear* in the length of the input string. You should be ready to argue why your solution is efficient. □

Exercise 1-CC.9 Program an *efficient* implementation of `Scanner`, and show it correct using `ScannerTest`. Efficiency here means the same as in Exercise 1-CC.8. Make sure that your solution correctly implements the notion of greediness, and that you are ready to argue why it is efficient. (*Hint:* reusing the `Checker` implementation of Exercise 1-CC.8 will *not* give rise to an efficient solution!) □

Exercise 1-CC.10 Add a constant `DFA_LALA` to `State`, analogous to `DFA_ID6`, that implements the DFA you developed in Exercise 1-CC.7. Add tests for this DFA to `CheckerTest` and `ScannerTest` that show the correctness of your DFA. □

1-CC.4 ANTLR

A grammar in ANTLR is defined in a file with extension `.g4`. In this course we will combine grammar files with JAVA source files. The ANTLR tool functionality consists of generating JAVA files from grammars that, when properly invoked, do the job of scanning, and later on, parsing and code generation.

This text assumes that you use ANTLR through its ECLIPSE plugin.

The lab files include the following example grammar:

```
1 lexer grammar Example;
2
3 @header{package pp.block1.cc.antlr;}
4
5 WHILE : 'while';              // Keyword
6 DO    : 'do';                 // Keyword
7 WS    : [ \t\r\n]+ -> skip ; // At least one whitespace char; don't make token
```

Here is an explanation of the most prominent features:

- Line 1: This declares the grammar. The name has to equal the file name (minus extension and not including any namespace information). The optional keyword **`lexer`** (which is used in the Compiler Construction world as a synonym for "scanner") means that this grammar only supports scanner rules; the default (obtained by leaving out the keyword) combines scanner and parser rules.
- Line 3: This specifies a line that should be inserted at the top of the Java files generated from the grammar. Since we want the generated files to be within the Java package "pp.block1.cc.antlr", we need a **`package`** declaration in the Java file; that is what this line achieves.
- Lines 5 and 6: This specifies that `while` and `do` are tokens of our `Example` language. `WHILE` and `DO` are the identifiers chosen for these tokens by the grammar designers; the fact that they equal the keywords is coincidental. Right now the names are themselves no used anywhere in the grammar.
- Line 7: This specifies that any non-empty sequence of characters from the set ' ' (space), '`\t`' (the TAB character), '`\r`' (the CR or Carriage Return character) and '`\n`' (the NL or Newline character) is considered a token; hoewever, the directive `->` **`skip`** then specifies that this token may actually be discarded. This construction is typically used for any information in the input file that does not need to be passed on for further processing: whitespace between other tokens (as here) or comments in the input file (in whatever comment syntax the input language supports).

Exercise 1-CC.11 To get acquainted with the ANTLR tool, carry out the following steps:

1. Generate the actual scanner `pp.block1.cc.antlr.Example` (a JAVA-file), and observe that, after this, all pre-defined lab files compile correctly. In the ECLIPSE plugin, this can be done by right-clicking the `g4`-file and selecting "Run As... → Generate Antlr Recognizer".
2. Study the syntax diagram of your language using the ECLIPSE plugin. For this purpose, go to "Window → Show View... → Other..." and select "ANTLR 4 → Syntax Diagram". This results in a so-called *railroad diagram* of your grammar rules; essentially a variation on a finite automaton where the labels are displayed as nodes and the nodes with incoming/outgoing edges as wiring between the nodes. Explain the difference between the railroad diagrams for `WHILE` and `WS`.
3. Run the JUNIT test `pp.block1.cc.test.ExampleTest`, and make sure you understand what's happening.
4. Run the JAVA class `pp.block1.cc.antlr.ExampleUsage`, and make sure you understand what's happening.

□

Exercise 1-CC.12 Consider again the grammar for 6-character identifiers from Exercise 1-CC.5.

1. Give an ANTLR lexer grammar that will recognize identifiers of this kind. Use so-called ANTLR *fragments* to make your rule(s) more readable (look up what fragments are in the online ANTLR documentation).
2. Test your grammar by providing a JUNIT test similar to `ExampleTest` (using the `LexerTester` class). Also test the acceptance of a sequence of identifiers. Does this behave as you expected?
3. What is the effect if you omit the keyword **`fragment`** from your grammar?

□

Exercise 1-CC.13 Consider again the grammar for PL/1 strings from Exercise 1-CC.6.

1. Give an ANTLR lexer grammar that will recognize strings of this kind, using the proper ANTLR syntax for exclusion (look it up!).
2. Test your grammar by providing a JUNIT test similar to `ExampleTest` (using the `LexerTester` class).

□

Exercise 1-CC.14 Consider the musical scanner of Exercise 1-CC.7.

1. Give an ANTLR lexer grammar that will recognize tokens of this kind.
2. Test your grammar by providing a JUNIT test.

□