Software Mechanics for Java Multi-Threading

Jan Bergstra *Marijke Loots*

Preprint nr. 005 July 1999

Artificial Intelligence Preprint Series

Software mechanics for Java multi-threading

J.A. Bergstra* & M.E. Loots†

July 5, 1999

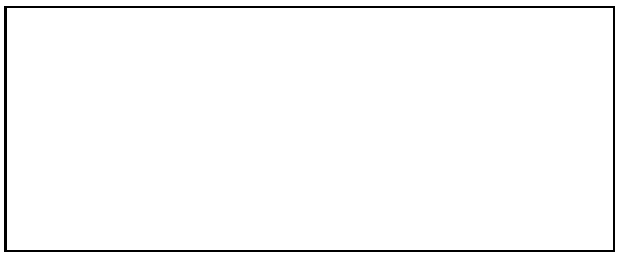

Figure 1: Java

Abstract

For a subset JavaTck (Java Thread Composition Kernel) of Java an empirical semantics has been developed.[1] Special emphasis is put on the role of synchronization features. The validity of empirical semantics is discussed in the light of a number of compiler postulates. A translation of process algebra with conditions and free merge to Java is used as an example.

Contents

1 Introduction **3**
- 1.1 Software mechanics and emperical semantics . . . 3
- 1.2 On the role of Java . . . 3
- 1.3 What is a thread? . . . 4
- 1.4 Empirical semantics . . . 4
  - 1.4.1 Limits of observation . . . 5
  - 1.4.2 Correctness as a default assumption . . . 5
  - 1.4.3 Incomplete coverage assumption 5
  - 1.4.4 Necessary Option Interpretation 5
- 1.5 Necessary Option Interpretation Barrier 6
- 1.6 Documentation constraint semantics . 6
- 1.7 Sequential virtual machine restriction 6
  - 1.7.1 Temporary projection semantics 7
  - 1.7.2 Randomized projection semantics . . . 7
  - 1.7.3 Randomized projections and distribution . . . 7
- 1.8 Compiler postulates . . . 7

2 JavaTck **8**
- 2.1 Software mechanics for JavaTck . . . 8
- 2.2 Java Class Families . . . 9
- 2.3 Flat Folder Hierarchy Notation . . . 9
  - 2.3.1 FFHN Primitives . . . 9
  - 2.3.2 Equations for flat folders . . . 9

3 Uniform class context **10**

4 Multi-threading: initial examples **10**
- 4.1 Non-atomic external actions . . . 10
- 4.2 Atomic external actions . . . 11
- 4.3 The collection of possible outputs . . . 12
- 4.4 The `Runnable` interface . . . 12
  - 4.4.1 Aliasing `start()` . . . 13
  - 4.4.2 Combining extension and implementation . . . 13

5 Modifying the threadvector **14**
  - 5.0.3 Preparatory thread classes . . . 14
  - 5.0.4 Rotating the thread vector . . 14

*University of Amsterdam, Programming Research Group, & Utrecht University, Department of Philosophy, Applied Logic Group, email: Jan.Bergstra@phil.uu.nl.

†Tilburg University, Faculty of Philosophy, email: M.E.Loots@kub.nl.

[1] Experiments were done with JDK1.2.1 on a SUN 10 with Solaris 2.5.

5.0.5 Comparison: no use of `yield` . 15
5.0.6 Setting priorities 15

6 Interleaving and locks 15
6.1 Turntaking logic 16
6.2 A base example 16
6.3 Threads can be interleaved 16
6.4 Restricting interleaving 17
6.4.1 Theads calling shared methods 17
6.4.2 Theads calling synchronized methods 18
6.5 Class locks and object locks 18
6.5.1 Competing for the object lock . 18
6.5.2 Different locks for different objects 19
6.5.3 Locking a block 19
6.5.4 Locking on independent objects 20
6.5.5 Locking on different objects II 21
6.5.6 Superobjects have the same lock 21

7 Thread inheritance 22
7.1 Thread inheritance: further examples . 23
7.1.1 Overrididing a public alias for `run()` 23
7.1.2 The 'Private' access specifier prevents overriding 24

8 Miscellaneous aspects of thread composition 25
8.1 A static resource class 25
8.2 A collection of (potential) threads . . 26
8.3 A catalogue of examples 26
8.3.1 Effect of priority updates . . . 26
8.3.2 The effect of `Thread.yield()` 27
8.3.3 Effects of locking 28
8.3.4 Priority updates without effect 29
8.3.5 Priority updates can go wrong 29
8.3.6 Priority updates need alive threads 29
8.3.7 Run-time determination of thread priority 30

9 Deadlocks and deadlock prevention 30
9.1 Absence of a potential deadlock 30
9.1.1 Only one class lock 31
9.2 A deadlocking system 31
9.2.1 A partial deadlock 32
9.2.2 Thematic deadlock removal . . 32
9.2.3 Synchronization as a cause . . 33

10 Livelock 34
10.1 Unlimited output 34
10.2 A livelock 34

11 Shared variables 35
11.1 Logical Processes 35
11.2 An environment 36
11.3 Turning process definitions into threads 36

12 Recursive Processes 37
12.1 The environment 37
12.2 Turning recursion equations into threads 38
12.3 Putting processes together 39
12.4 Relaxing the current-thread-persistence postulate 39

13 Exception handling 41
13.1 Exceptions defined 41
13.2 Exceptions motivated: pragmatic aspects 41
13.3 Exceptions motivated: fundamental aspects 41
13.4 Catching a self-defined exception . . . 42
13.5 Method overriding and Exceptions . . 42
13.5.1 Threads cannot throw exceptions 42
13.6 Handling an exception 43
13.7 Throwing a thread as an exception . . 44

14 Waiting for a thread to terminate 44
14.1 Aliasing `join()` 45
14.1.1 Joining dominates priorities . . 45
14.1.2 `InterrruptedException` not easily generated 46

15 Handing over object locks 46
15.1 An alias for `wait()` 47
15.2 The object lock compromised 47
15.3 It might go wrong 47
15.3.1 Notification is needed 48
15.3.2 Alternating handovers 48
15.3.3 Notifications may get lost . . . 48
15.3.4 Mutual exclusion still works . . 49

16 Global notification **49**
16.1 Replacing global notification by 'local' ones 50

17 Blocking actions and pipes **51**
17.1 Construction of a connected pipe . . . 51
17.2 Blocked reading demonstrated 51
17.3 One to one interconnection 52

18 Concluding remarks **53**

1 Introduction

Multi-threading is a very important facility for modern programming notations. (See 1) In particular Java™ has brought multi-threading home to the million-programmer market place. This paper explains multi-threading from a very narrow perspective, making it, hopefully, usable for readers with a professional interest in programming. The objective of the book is to provide a so-called empirical semantics for a selection of programming features for multi-threading. Java is taken as a carrier for the explanations, multi-threading being the subject at large, however. The purpose of the text more specifically is to provide readers with an understanding of multi-threaded programs independently of any computer practice or experiment.

The syntax used to express multi-threading is discussed, as well as a very specific approach towards semantics: empirical semantics. Empirical semantics has been developed in [5] for a selection of object-oriented features of Java. We refer to that source for a discussion of empirical semantics.

1.1 Software mechanics and emperical semantics

The term 'Software mechanics' has been coined in [5]) to denote the study of the working of program notations. Software mechanics for a particular program notation can be studied in a variety of ways, empirical semantics being just one of those. Empirical semantics consists of the development of a large family of simple examples, illustrating key aspects of program execution. The examples serve no other purpose than to illustrate the meaning of programming features and feature combinations. The examples are not supposed to have a deeper meaning or intention. None of the examples is reusable for any other purpose than for experimentation with language features. Alternative routes are: (i) reading the language documentations, (ii) reading the program code of compilers and of run-time systems, (iii) getting acquainted with a multitude of applications.

1.2 On the role of Java

Computer scientists differ so vastly in opinion regarding the importance and perspective of Java that some remarks concerning this matter are mandatory. We consider Java to have the following properties/qualities:

1. Java is an experimental language. It has been designed to fulfill a number of almost inconsistent objectives, as a consequence of which it suffers from disadvantages concerning speed and stability which can be observed by every user of the language. Important objectives were: (a) to obtain an Object-oriented language simpler than C++ yet capturing the critical advantages of objects, classification and inheritance, (b) to add multi-threading at an early stage, (c) to allow a flexible integration of application libraries and the language (and to use this to keep the language small), (d) to prepare all software written in Java for being down-loaded on other machines, guaranteeing a high level of security for both user and programmer.

2. Java may well be a worthy successor of PASCAL for teaching puposes, allowing its users to get acquainted with a wide variety of modern programming mechanisms. An understanding of Java will make people grasp the need for faster and more low-level and risk-prone programming notations (especially C). This direction of concept development is more efficient than the other way around.

3. There is a massive literature on Java allowing those who are interested efficient access to many aspects of the language and to its most useful programming styles. This is a definite advantage for Java as a vehicle for teaching activities.

4. Like FORTRAN and COBOL before, C and C^{++} will be around for many more years, quite irrespective of academic appreciation or fashion in program notation development. Java cannot and need not replace, supersede or make obsolete any of these notations in industrial practice in order to be itself a success in the world of teaching. The minimum needed (to make Java-based research and teaching rewarding) is that for the next 10 years or so there will be compilers around for the most popular platforms. This being not unlikely, Java can at least be granted the status of ML, LISP, PROLOG and HASKELL.[2] If Java's objectives prove unreachable (or currently unreachable) or are simply not reached for secondary reasons, its position on the market place will slowly marginalize. That risk is of no fundamental concern for either research or teaching.

5. The JDK thread class is by no means the most advanced or the most effective thread library around. Its virtue is simplicity rather than expressiveness. Its clear integration in the Java core language is a definite advantage. Some of the considerations below may be regarded quite Java specific. We have tried not to dig too far into aspects that are Java specific and of very limited relevance for the subject of multi-threading in general.

1.3 What is a thread?

Unlike in process algebra (e.g. [3]), where processes are given a meaning as purely mathematical objects per se, threads can hardly be considered in full isolation. Intuitively a program may be viewed as a map or a landscape. During execution a machine visits sites on the map and modifies data placed at such sites. From a distance a computation takes the form of a path on the map. Ignoring the data one might call the path a thread. The most prominent perspective on concurrency is to view a program as being executed in parallel with itself, different executions being characterized by the various paths they choose on the map. The multi-threading paradigm seeks to exploit this intuition. When different parts of a piece of code are run on different machines, threads come close to processes. When the same or quite related code is run on a single processor, the programming features expressing multi-threading may be considered a range of programming primitives for sequential deterministic programming. These primitives will invoke the support of a (hidden) scheduler in the way that recursion primitives will hide the use of a stack. We refer to [6] for a very elementary discussion of multi-threading along these lines.

1.4 Empirical semantics

At the heart of empirical semantics is an effort to provide a significant coverage by examples documenting the execution of a particular kind of programs. We refer to our [5] for an exposition of the scope and limits of empirical semantics, including an instance concerning single-threaded object-oriented programs. An important diference between the project reported in [5] and the present task is implied by the intrinsic incompleteness of language documentation as a means to predict the behavior of multi-threaded programs.[3] The execution of multi-threaded programs is not meant to be entirely specified by the language semantics, leaving compiler and executing machine significant degrees of freedom regarding the selection of threads from which to execute an action.

[2] Many readers may consider this description pessimistic. However, other readers may consider it too optimistic.

[3] In [5] the position is taken that, assuming by default that all observed behavior is reasonable (i.e. not in contradiction with the documentation) the observations reveal facts which are necessary, in the sense that future experiments will (can only) show the same behavior. It is implicitly assumed that when future compilers are used machines with correspondingly larger memories and faster processors will also be used. There is no point in running tomorrow's compiler on today's machine. It is further assumed that the language specification is stable in the sense that what must happen today at least may happen in the future (for Java at least).

1.4.1 Limits of observation

As a consequence a collection of examples of executions of multi-threaded programs cannot provide the first intuition of the constraints imposed by the intentions of language designers, compiler builders and machine architects. In order to obtain that intuition, non-experimental data will have to be investigated. The most plausible source of such data is the language documentation. With documentation constraint semantics we refer to the meaning of a program notation (or a fragment of it) emerging from a systematic investigation of the existing documentation. We claim that empirical semantics for multi-threading must be complemented with a documentation constaint semantics, for the very simple reason that no experimental set-up can be guaranteed to demonstrate all degrees of freedom intentionally left open by the language design within a reasonable amount of time.

1.4.2 Correctness as a default assumption

The following default assumption is used throughout: observed system behavior is reasonable. That implies that no attempt is made to classifiy some observations as compiler errors or problems in any other sense. This default assumption should not be interpreted as saying that compiler errors will not be observed. The default presupposition implies that the behavior of multi-threaded programs that we have observed is considered 'correct'. Having acquired an initial understanding, as a second stage of involvement one may begin to question the observations or their assesment. Unlike in the case of deterministic computations, the correctness of observed behavior is no indication of its necessity, however.

1.4.3 Incomplete coverage assumption

In order to make the previous remarks more explicit we formulate the so-called incomplete coverage assumption as follows: If a program notation leaves significant degrees of freedom to compiler/interpreter, run-time environment etc., not even the first intuition of the constraints satisfied by all legal executions can be acquired from mere experimentation with any given implementation.

The incomplete coverage assumption is not needed for all kinds of program notations. For single-threaded sequential programming, for instance, it may not be applicable. This does not mean that behavior is always predictable for single-threaded programming. If the language specification claims complete coverage, however, a full intuition should emerge from experiment alone (in some cases, and in principle).

1.4.4 Necessary Option Interpretation

In order to generalize from mere observation to a statement potentially affecting future program execution, we will formulate three default assumptions:

Bugfree setting assumption. The bugfree setting assumption asserts that none of the observations (to be reformulated in more general terms) report the finding of an (obvious) bug.

It is not easy to verify this assumption, but it is methodologically correct to assume it until contrasting signs cannot be ignored.

Sound documentation assumption. The documentation is sound in the sense that its statements will hold for future implementations of the language. Predicted behavior will occur, forbidden behavior will not.

If this assumption fails, portability of software to a new language implementation is at risk, even if the best possible precautions have been taken. Once a language has stabilized, the sound documentation assumption will be a major source of confidence for software engineers.

Rigid documentation assumption. The rigid documentation assumption asserts that the language specification will not change. In particular it will not be refined by disallowing behavior which was left as a possibility in an earlier version of the documentation.

This assumption is very likely wrong in most cases and probably for Java as well. Nevertheless, it enables one to cast the observations in a

conceptual framework, the effect of its rejection becoming clear in a second stage of reflection.

On the basis of the three assumptions just mentioned, the following interpretation of material acquired by observation has been formulated:

> (Necessary Option Interpretation) Let P be an observation of a program execution leading to an output. Then necessarily (i.e. in all possible future worlds, in particular after many stages of further development of the program notation and its underlying program environment) the behavior reflected in P should still be taken into account as a serious and realistic possibility by any programmer intending to write programs that can be ported to future language implementations. Stated in other words: in all future occasions it will be necessary to view P as a behavioral option of the program.

1.5 Necessary Option Interpretation Barrier

Not only is the necessary option interpretation a meaningful framework for interpreting the meaning of observations given the incomplete coverage assumption, it is also an upper limit to the potential of empirical work. This limit we call the Necessary Option Interpretation Barrier.

Experimentation alone will not produce any information about 'all the executions of a program that conform the language definition'. This is not the classical statement that testing cannot prove a system to work correctly in all cases; it implies that no form of intuitive induction, based on a limited amount of experimental data, can handle the combinatorial explosion of options left open by a language specification (to the extent that the incomplete coverage assumption is justified). We rule out the applicability of sound statistical techniques to explore the 'space of all possible implementations'. The exclusion of statistics cannot just be based on the combinatorial explosion; it is also induced by the unpredictable and complicated irregularities of modern program notation design.

1.6 Documentation constraint semantics

As a complement to empirical semantics, documentation constraint semantics takes the existing documention as its point of departure. Ideally the authors have contributed no single line of explanations themselves, collecting all descriptions from existing work, being explicit about varying degrees of reliability and authority of their sources. The text is arranged around a few simple example programs. Documentation semantics is an option, not necessary and often insufficient.

An early reference to Java multi-threading is [12], according to their explanation a thread of control is

> '..a path taken by a program during execution. This (*path*) determines what code will be executed...'.[4]

1.7 Sequential virtual machine restriction

We restrict our discussion to JBVM (Java Bytecode Virtual Machine) as a sequential virtual processor, mapping all threads on a list (called threadVector), giving turns to the elements of the list partially on the basis of information contained in the compiled program, partially on the basis of specific scheduling algorithms in the compiler (outside programmer control). For this restricted case a projection semantics in the style of [4] (extended in [6]) can be found by following the construction of the compiler. We will not provide a projection semantics here, as it would be too complicated. Besides, its mere existence would add too little to the relevant intuitions. The assumption of this restriction limits our considerations to the case of sequential multi-threading, leaving a discussion of parallelism-oriented multi-threading for another occasion. Parallel multi-threading is conceptually more complex than sequential multi-threading as it introduces non-determinism during computations.

[4] Italics between brackets have been added by the present authors in order to clarify a quotiation, otherwise harder to understand after having been cut out of its context.

1.7.1 Temporary projection semantics

The Java compiler of the JDK (forgetting version numbers for the moment) provides a translation of Java (and of its subset JavaTck) to bytecode, a low-level assembly language in binary form that can be processed by the JBVM. As stated above, the existence of compiler plus JBVM suffices for a projection semantics for JavaTck.

Unfortunately this semantics is an overspecification. It restricts program behavior where it need not do so, even against the intentions of the language developers.[5] The language semantics (as obtained via a projection) expresses several aspects where observations on compiler + JBVM (even after validation by means of code inspection) may show behavior which will not be guaranteed to be the same for all future versions of the JDK. For that reason a projection semantics developed via inspection of the compiler and JBVM (in some stage of their development) can only be temporarily valid. In any case, it is at least one way of obtaining a temporary projection semantics for JavaTck at some stage of development.

1.7.2 Randomized projection semantics

Is it principally possible to safeguard one's software engineering in JavaTck against the risk of using an overspecified temporary semantics? The most fundamental answer to this question would adopt denotational semantics in the style of [8]. This technique should be able to abstract from superfluous detail. As a consequence no unwarranted hidden assumptions are present in semantic arguments. The objection against it lies in the sheer complexity of the semantic models thus obtained.

As an upper limit of what can easily be done (not that easily, however, for Java!) a projection semantics may be adapted to protect its users against drawing unwarranted conclusions caused by semantic overspecification. In the case of JavaTck that may for instance be done by taking a computable random generator in order to schedule turntaking. In order to avoid unwarranted assumptions due to overspecification the code of the random generator should not be used while reasoning about a program. (One might conclude that in the process of program development, a random implementation for turntaking in the JDK is very useful, because it will be a great obstacle for the kind of program development that relies too much on a specific scheduling strategy. A choice in favor of randomized thread scheduling has been made for the ToolBus [2].)

[5] Semantic freedom is a strength for a language as it allows compiler writers to be more innovative, thereby inviting them to be more competitive.

1.7.3 Randomized projections and distribution

The argument in favor of randomized projection semantics is primarily valid in a non-distributed (i.e. sequential and deterministic) setting. Randomized projection semantics can also be useful to describe a distributed Java implementation, allowing different threads to be executed concurrently on independent parallel processors. From a fundamental point of view, it is difficult to defend the choice in favor of randomized projection semantics over a process semantics that properly incorporates run-time nondeterminism. For the distributed case, randomized projection semantics only provides an initial viewpoint, needing subsequent refinement.

1.8 Compiler postulates

Assuming that all the properties of an implementation are ultimately under the control of the compiler, another way of going about the uncertainties left open or even generated by the language specification is to stipulate postulates about the program execution (termed compiler postulates as the compiler can enforce the validity of them) and to derive or assert program properties on the basis of the postulates. Only if a compiler satisfies the postulates needed for a property is that property guaranteed. The advantage of this viewpoint is that it allows one to reason efficiently in those cases where the observed behavior is in accordance with the predictions of the postulates. The possible emergence of compilers (or compiler options) satisfying some underlying postulates would in its turn make the investment in postulate-dependent

reasoning more rewarding. Four compiler postulates are useful in the case of JavaTck.

Priority dominance postulate. The priority dominance postulate asserts that a thread cannot get a turn if a higher-priority thread is enabled as well.

Current thread persistence postulate. The current thread persistence postulate asserts that the current trace remains active until it either terminates, yields, stops, is interrrupted, waits, is put to sleep, or begins the competition for a lock. This implies that a step which is enabled will always be performed immediately after the action logically preceding it in the same thread (unless one of the aforementioned cases applies). The postulate excludes system events causing control to move to another thread. It also excludes fully randomized thread selection after each action. (Randomization may have an effect only when explicitly programmed instructions cause control to be transferred to another thread.)[6]

Strict rotation postulate. The strict rotation postulate expresses that a yield action will hand over control to another thread unless this violates another constraint. (In particular a call for `yield()` may be ignored when executed by the unique maximum priority thread, and when executed from within a synchronized method with all other threads competing for the lock currently held by the (current) thread.

Fair rotation postulate. The fair rotation postulate implies that if an infinite number of yields is performed, each thread will be given a turn infinitely often as well.

The two postulates last mentioned are important for our translation of process definitions with conditions and shared variables to JavaTck.

[6] A scheduler for thread operation satisfying current thread persistency is also called non-preemptive.

2 JavaTck

JavaTck is a subset of Java, introduced here only for expository reasons. JavaTck stands for 'Java Thread Composition Kernel'. It is a small subset of Java leaving out several OO-features (casting, interfaces, 'super', non-trivial initialization of static/instance fields), allowing full concentration on experimentation with multi-threaded programs. The following features are included in JavaTck: extending the `Thread` class and implementing the `Runnable` interface, starting a thread, overriding the method `run()`, setting priorities, testing `isAlive()`, handing over control to another thread using `yield()`, `synchronized` methods (class locks and object locks). Not included are Thread Groups, several features which have been labeled deprecated since the original Java versions appeared, or the method `destroy()`, which is currently considered a bad proposal anyhow.[7] The features `wait()`, `notify()`, `notifyAll()`, are covered in JavaTck, however, the use of real-time clock-based primitives is not.

Garbage collection is considered important for threads just as much as it is for other objects. The conditions under which threads can become garbage and can be 'garbage collected' have not been investigated below for reasons of brevity. From a philosophical point of view every unreachable thread (object of a class extending the `Thread` class) that either has not yet been started or that has terminated can be considered garbage and is a candidate for removal.

2.1 Software mechanics for JavaTck

Below we will explore the software mechanics of JavaTck, using the style of presentation of empirical semantics. JavaTck appears to be a rich and reasonable thread composition kernel, in spite of many features having been left out to obtain comprehensibility as well as generality.[8] In some cases one or

[7] Deprecation is an indication of 'past existence' a status that the 'hypothetical' method `destroy()` has not yet obtained. `destroy()` may be termed 'preemptively deprecated'.

[8] Our experiments demonstrate that the compiler/JBVM chooses execution of the (or rather a) highest-priority thread in all cases. Although not guaranteed by the language specification (rather the opposite) the priority effect could be made a

more of the compiler postulates will be assumed to argue in favor of the plausibility of a program. This is meaningful also if the postulates are not generally satisfied by an implementation at hand.

2.2 Java Class Families

Following [5] we will employ Java class families (JCF's) using Folder Hierarchy Notation (FHN) for which we refer to [5]. Using JCF's we can combine Java program descriptions out of existing and new classes and packages in a most flexible way. The use of JCF's and FHN below speaks for itself, the complexity of the FHN-equations mainly being connected with folders needed to denote packages. For the sake of completeness we have included the equations of FHN fixing its properties for flat folders. A flat folder is just an (unordered) set of named files. In addition FHN can acommodate different files under the same name in a single expression. The idea is that these files occur in an ordered fashion and that their contents should be concatenated in order to obtain a 'canonical form' presenting at most one file for each name. FFHN (flat FHN) is a subset of FHN, unable to cope with packages but sufficient for the needs of this document.

2.3 Flat Folder Hierarchy Notation

Being a subset of Java, JavaTck classes are produced as text files having a `.java` name extension. These files are collected in a folder. FFHN allows one to denote particular JCF's in full information outfit. The term folder is used rather than package because some folders denoted in FHN may fail to qualify as Java packages, for instance if the included files in no way contain Java classes.

Folders can be empty, or they can contain a number of files carrying their own names. In the description of FFHN A, A_i ranges over texts. Concatenation of texts is denoted with '$*$'.

reliable language feature that would significantly improve the transparency of the software mechanics of JavaTck.

2.3.1 FFHN Primitives

The primitives of FFHN are these:

- $\emptyset$ represents the empty folder. Its size is 0.
- `file:` *fileName*(A), denotes a file named *fileName* containing the character sequence A. The file `file:` *fileName*(A) is itself a folder of size 1.
- Given folders B_1 and B_2, $B_1 \cup B_2$ is the union of the two folders. There are some constraints:

  If B_1 contains a file `file:`*fileName*(A_1) and B_2 contains a file `file:` *fileName*(A_2), the combination $B_1 \cup B_2$ will contain a file `file:` *fileName*$(A_1 * A_2)$,($*$ denoting string concatenation.) This rule prevents folder union from being commutative. A folder can contain at most one file under some given name.

An FFHN-JavaTck expression is an FFHN-expression with all texts A listed in files being (Java) class descriptions. For such an expression to be syntactically correct the empirical criterion is that it is accepted by the compiler of JDK 1.2.1

2.3.2 Equations for flat folders

The rules for FFHN can easily be summarized in a set of conditional equations[9]. The use of the equations is as follows: in the process of writing a JCF, named files (and named folders) can be introduced in successive stages. The equations describe how to combine parts of files and folders into complete files and folders. This process has to be performed before the deletion of named folders or files is attempted. The flexibility of FFHN is helpful when a portfolio of JCF's is to be denoted.

$$\emptyset \cup B = B, B \cup \emptyset = B$$

[9] FFHN is called a notation in spite of its being an algebra according to even the most restricted definitions. Although FFHN is an algebra, its equations are neither deep nor difficult. The transformation rules expressed by the equations are all fairly obvious. FFHN is a bookkeeping device and no more than that. This lack of intrinsic interest of the FFHN equations motivates the decision to call it a notation rather than an algebra or a calculus.

$$(B_1 \cup B_2) \cup B_3 = B_1 \cup (B_2 \cup B_3)$$

$$\texttt{file} : x(A_1) \cup \texttt{file} : x(A_2) = \texttt{file} : x(A_1 * A_2)$$

$$x \neq y \rightarrow \texttt{file} : x(A_1) \cup \texttt{file} : y(A_2) =$$

$$\texttt{file} : y(A_2) \cup \texttt{file} : x(A_1)$$

3 Uniform class context

To simplify the presentation a number of classes is put in a context. These classes are added to all subsequent examples. FFHN is used to denote this context as well as the way of combining it with other classes. The context, named JCFp, is extended one step further in section 4.2.

We introduce JCFp = JCFps ∪ JCFpb ∪ JCFpc, with the diffferent parts as follows:

JCFps = `file:s.java(`

```
class s {static void main(String x[]){
      c.m();}
}
```

). JCFps contains the method `main()` which is called in all experiments. Running a program from the series of examples below results from the fixed command `javac s` which will call `main()` of class `s`. The execution of `main()` consists of a call of a method `m()` of class `c`. Many different instatiations of class `c` are presented below. For the sake of consistency different instances of the class `c` should of course occur in different class families. A single JCF can only contain one definition for a given class, in order to avoid being inconsistent. The three parts JCFpb, JCFpc and JCFpi contain classes with output methods for booleans, characters and integers.

It should be noticed that console output is by no means a central feature for Java, one of the original motivations for the design of Java being to obtain a high-level language for software that can be downloaded to control small equipment for which a console is by no means a likely peripheral. The use of console output (the methods `System.out.print()` and `System.out.println()`) is best encapsulated in such a way that examples are made independent of its syntax. This is achieved by the introduction of these classes.

JCFpb = `file:bco.java(`

```
public class bco {
public static void p(boolean p) {
      System.out.println(p);}
public static void pw(boolean p) {
      System.out.print(p +" ");}
}
```

), JCFpc = `file:cco.java(`

```
public class cco {
public static void p(char c) {
      System.out.println(c);}
public static void pw(char c) {
      System.out.print(c +" ");}
}
```

), and JCFpi = `file:ico.java(`

```
public class ico {
public static void p(int i) {
      System.out.println(i);}
public static void pw(int i) {
      System.out.print(i +" ");}
}
```

).

4 Multi-threading: initial examples

To obtain an initial grasp of the basics of multi-threading a number of 5-threaded systems are demonstrated below. Four threads are explicitly created, the fifth one being the thread originally offered by the system to call main.

4.1 Non-atomic external actions

The following JCF displays an example with 4 created threads (not counting the start-up thread for main): JCF1 = JCFp ∪ `file:c.java(`

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2();
thr1 t3 = new thr1();
thr2 t4 = new thr2();
cco.p('x');
t1.start();
t2.start();
cco.p('y');
t3.start();
t4.start();
cco.p('z');}
}

class thr1 extends Thread {
public void run() {
      bco.pw(false);bco.p(true);}
}

class thr2 extends Thread {
public void run() {
      cco.pw('a');cco.p('b');}
}
```

). This JCF produces:

```
x
y
z
false true
a b
false true
a b
```

According to the language specification it is not fixed in which order the threads will perform their actions. Not even the uninterrrupted printing of `true` can be guaranteed for JCF1.[10] To guarantee that outputs can proceed in an uninterrupted fashion, the output classes can be modified so as to have all output actions in one class and synchronized, which will guarantee their atomic execution.[11]

4.2 Atomic external actions

The class `syo` encapsulates the output actions offered by the classes in JCFp in so-called synchronized methods. All further examples will use methods of `syo` instead of methods of classes in JCFp directly. The principal virtue of this packaging of output actions in static synchronized methods of the same class is that whenever different threads attempt to perform output actions simultaneously the run-time system will guarantee that these actions exclude one another in time. In order to allow a thread to execute one of the methods of `syo` the thread must acquire the so-called class lock. After terminating the method execution the class lock is returned by the thread and the run-time system can allow another thread to acquire the class lock. Different threads can compete for the class lock of `syo` whenever they intend to perform one of the methods of `syo` at the same time. JCFsyo = `file: syo.java`(

```
public class syo {
static synchronized
   public void
      p(boolean b) {bco.p(b);}

static synchronized
   public void
      p(char c) {cco.p(c);}

static synchronized
```

[10] This is too pessimistic. The class `System.out` should be implemented so as to make all print actions atomic (but not the `write()` methods in fact. Nevertheless we will free ourselves from this assumption by explicitly wrapping the output actions in synchronized methods.

[11] The atomicity of these actions is relative: this particular collection of output actions (represented as synchronized methods of the same static class) can be regarded a collection of atomic actions with respect to one another. Below, the mechanism of syncrhronized methods will be discussed in far more detail. At this point the best interpretation of the synchronized methods, as occurring in the modified output class, is that by using synchronized methods one establishes the class `syo` as a collection of atomic actions on the basis of which subsequent discussions and illustrations can take place place. In particular a meticulous investigation of synchronized methods is simplified when based on a platform of actions excluding one another in time. A collection of atomic actions now being available, the investigation of synchronization primitives is more easily structured.

```
   public void
      p(int i) {ico.p(i);}

static synchronized
   public void
      pw(boolean b) {bco.pw(b);}

static  synchronized
   public void
      pw(char c) {cco.pw(c);}

static synchronized
   public void
      pw(int i) {ico.pw(i);}
}
```

). This output class leads to a further modification of JCF1: JCF2 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2();
thr1 t3 = new thr1();
thr2 t4 = new thr2();
syo.p('x');
t1.start();
t2.start();
syo.p('y');
t3.start();
t4.start();
syo.p('z');}
}

class thr1 extends Thread {
public void run() {
      syo.pw(false);syo.p(true);}
}

class thr2 extends Thread {
public void run() {
      syo.pw('a');syo.p('b');}
}
```

). Its output coincides with that of JCF1 (though not necessarilly!). For JCF2 it is reasonable to assume that all output actions are performed in an atomic way in view of the properties of the class `syo`.

4.3 The collection of possible outputs

JCF2 describes a system containing (during execution) 3 parallel processes (threads) at the same time. The language description being completely unspecific about when to give a thread the turn to execute, many different executions can be imagined, all consistent with the language specification. Two examples of possible outputs are:

```
x               x
y               false true
false true      a b
a b             y
a b             a b
z               false true
false true      z
```

4.4 The `Runnable` interface

In addition to the definition of extensions of the class `Thread` there is another option to generate threads. The interface `Runnable` is part of Java. Its only method is `run()`. If a class is specified to implement the `Runnable` interface, its method `run()` must be overridden and at least one constructor must be provided. In addition, this new constructor should start with calling the constructor of class `Thread` giving it the argument `this`. Thus the system is instructed that the objects constructed will in fact be threads. Moreover, a call of `start()` will now use the overridden form of `run()`. This somewhat unnatural procedure compensates for Java's lack of multiple inheritance (at least in some cases). Later we will see that it allows an object to behave like a thread and like an exception at the same time. The next example computes the same output as the previous two (though not necessarily!). JCF3 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2();
```

```
thr1 t3 = new thr1();
thr2 t4 = new thr2();
syo.p('x');
t1.t.start();
t2.t.start();
syo.p('y');
t3.t.start();
t4.t.start();
syo.p('z');}
}

class thr1 implements Runnable {
Thread t;
thr1() {t = new Thread(this);}
public void run() {
      syo.pw(false);syo.p(true);}
}

class thr2 implements Runnable {
Thread t;
thr2() {t = new Thread(this);}
public void run() {
      syo.pw('a');syo.p('b');}
}
```

).

4.4.1 Aliasing `start()`

By introducing a modified `start()` method the situation can be made quite similar to the thread extension case. JCF4 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2();
thr1 t3 = new thr1();
thr2 t4 = new thr2();
syo.p('x');
t1.Start();
t2.Start();
syo.p('y');
t3.Start();
t4.Start();
syo.p('z');}
}

class thr1 implements Runnable {
Thread t;
thr1() {t = new Thread(this);}
public void run() {
      syo.pw(false);syo.p(true);}
void Start() {t.start();}
}

class thr2 implements Runnable {
Thread t;
thr2() {t = new Thread(this);}
public void run() {
      syo.pw('a');syo.p('b');}
void Start() {t.start();}
}
```

). The output once more coincides with the outputs of the previous programs.

4.4.2 Combining extension and implementation

Of course implementing an interface can coincide with extending a class. That is a major reason for having interfaces at all, as it provides an approximation of multiple inheritance, the lack of which is a significant (but intentional) restriction of the Java syntax. In the example below (again producing the same output, this time repeated for clarification) the duplication of the alias of `start()` is removed. In addition the conditional operator is demonstrated and it is illustrated that the conditional operator can be applied to threads. It turns out to be critical that the second and third argument of `b? x:y` should be of the same type! JCF5 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
boolean b = true;
thr1 t1 = new thr1();
thr2 t2 = new thr2();
thr1 t3 = new thr1();
thr2 t4 = new thr2();
syo.p('x');
```

```
(b? t1:t3).Start();
t2.Start();
syo.p('y');
(!b? t1:t3).Start();
t4.Start();
syo.p('z');}
}

class TS extends Thread {
Thread t;
void Start() {t.start();}
}

class thr1 extends TS
implements Runnable {
thr1() {t = new Thread(this);}
public void run() {
      syo.pw(false);syo.p(true);}
}

class thr2 extends TS
implements Runnable {
thr2() {t = new Thread(this);}
public void run() {
      syo.pw('a');syo.p('b');}
}
```

). This program outputs the same as did several programs before:

```
x
y
z
false true
a b
false true
a b
```

5 Modifying the threadvector

According to the terminology of [5] during the execution of a multi-threaded program on a sequential machine a so-called threadvector is maintained in which alll threads feature in the order of priority. The action `yield()` from Java.lang (class Thread) represents a rotate or shift of this vector, thus giving the turn to the next one in the sequence. On the basis of these examples we can draw the following conclusions:

- There are actions (method calls) having the sole purpose to move control from one thread to another one. Notably `yield()`, a static method for the class `Thread` and `setPriority(int i)`, an instance method for the class `Thread` are such actions.
- As a warning, rather than as a conclusion: the operational effect of the actions just mentioned is not completely fixed by the language designers. These effects may vary from implementation to implementation, and may depend on parameters entirely outside the control of the programmer.

5.0.3 Preparatory thread classes

The following two classes are used in two subsequent examples: JCFthrs = `file:thr1.java`(

```
class thr1 extends Thread {
public void run() {
      syo.pw(false);syo.p(true);}
}
```

)∪ `file:thr2.java`(

```
class thr2 extends Thread {
char _x,_y;
thr2(char x,char y) {_x = x;_y = y;}
public void run() {
      syo.pw(_x);yield();syo.p(_y);}
}
```

). The second class provides a parametrized family of threads.

5.0.4 Rotating the thread vector

The method `yield()` can be used (by the current thread) to hand over control to another thread in the thread vector (taking priorities into account). Formally this amounts to a rotation (more general a permutation) in the thread vector, which is maintained in such a way as to have the current thread

as its head. JCF6 = JCFp ∪ JCFsyo ∪ JCFthrs ∪ `file:c.java`(

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2('a','b');
thr2 t3 = new thr2('c','d');
thr2 t4 = new thr2('e','f');
syo.p('x');
t1.start();
t2.start();
syo.p('y');
t3.start();
t4.start();
syo.p('z');}
}
```

). This JCF produces

```
x
y
z
false true
a c e b
d
f
```

5.0.5 Comparison: no use of `yield`

If, in JCFthrs, the instruction `yield();` is temporarily removed from the body of the constructor in `file:thr2.java`, changing JCF6 into JCF7, the following output is produced:

```
x
y
z
false true
a b
c d
e f
```

5.0.6 Setting priorities

Setting priorities has a significant effect on execution. The effect is not easy to grasp in all cases. It turns out that setting the priority of a thread above five gives it a higher priority, setting it below 5 will relatively delay its execution. The language specification is very clear however. An example shows the sort of effect that setting priorities can have, seeming to indicate that the main thread used by the system to start the program has the average default priority 5, which is also the initial priority of all other threads. JCF8 = JCFp ∪ JCFsyo ∪ JCFthrs ∪ `file:c.java`(

```
class c {
static void m(){
thr1 t1 = new thr1();
thr2 t2 = new thr2('a','b');
thr2 t3 = new thr2('c','d');
thr2 t4 = new thr2('e','f');
syo.p('x');
t2.setPriority(6);
t3.setPriority(4);
t1.start();
t2.start();
syo.p('y');
t3.start();
t4.start();
syo.p('z');}
}
```

), which produces:

```
x
a b
false true
y
z
e f
c d
```

6 Interleaving and locks

In this section a systematic survey is made of interleaving, the phenomenon that different threads may be operational simultaneously, and locking, the mechanism used to prevent interleaving in cases it is considered a disadvantage.

6.1 Turntaking logic

The key aspect of multi-threaded program execution is turntaking: which thread gets a turn to perform a step when? This is a difficult subject, made more difficult by incompleteness of the language specification as well as differences between implementation philosophies.

An approximation of the 'logic of turntaking' is obtained by considering a large family of examples. The compiler postulates can be of great help to structure reasoning about turntaking in concrete cases. The examples of this section will validate the following conclusions:

- Different threads can, in appropriate circumstances, perform their actions in an interleaved fashion.
- On most current implementations (e.g. the one used by the authors) this interleaving behavior must be explicity provoked by programming actions meant exclusively to influence the turntaking process (in particular `yield()` and `setPriority()`).
- Guarantees against interleaving can be obtained using sychronized blocks and methods. To each class and object a so-called lock is attached. Each thread can at any time be in possesion of a number of locks, each lock being acquired by one thread at most. A synchronised class method (static method) can only be executed by a thread in possesion of the lock of that class. A synchronized instance method can only be applied by a thread in possesion of the lock of the instance to which the method will be applied. A synchronized block can only be performed by a thread in possesion of the lock mentioned in the code of the synchronized block construct.
- Different locks (i.e. locks for different objects or classes) are unrelated. Threads dependent on different locks can proceed in any order. Only by making threads compete for the same lock can it be excluded that certain interleavings of actions occur.

6.2 A base example

The systematic development of examples can start with cases involving two new threads. In order to obtain symmetric examples the main thread will in most cases be designed in such a way that it terminates very quickly, leaving the concurrent composition of created threads active. Consider JCF9 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      (new T()).start();
      (new T()).start();
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      syo.p('a');
      }
}
```

). It outputs

```
true
a
a
```

6.3 Threads can be interleaved

It is not entirely trivial to demonstrate that threads can indeed be executed in a concurrent fashion. In fact, explicit precautions are required to invite our implementations to show an interleaving execution of threads. For instance, given a system that is in accordance with the four postulates of 1.8, the interleaved execution of threads can only occur if special methods such as `wait(), setPriority(), Thread.yield()` or `join()` are used. This is illustrated in the following example, for which the following facts should be taken into account:

1. The instruction (method call) `setPriority(2)` sets the priority of `this` thread to 2 (which is below the default value). The implementation used gives all threads a default priority level 5. The

priority modification will not by itself generate any outputs.

2. Although not prescribed by the language documentation, the implementation behaves in accordance with the four postulates (at least in the cases that we have tested). If the `setPriority()` instruction is dropped from the body of the `run()` method below the order of outputs is `true a b a b`. This behavior indicates absence of interleaving.

3. Unless interleaving takes place the order of the outputs cannot be explained.

We consider JCF10= JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      (new T()).start();
      (new T()).start();
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      syo.p('a');
      setPriority(2);
      syo.p('b');
      }
}
```

). Upon execution this JCF outputs

```
true
a
a
b
b
```

It is essential to notice that in spite of the 'trick' needed to demonstrate interleaving behavior, threaded programs must always be expected to be executed in an interleaved fashion. Stated in other words: unless careful measures are taken, no logically possible interleavings of thread operation can be excluded. The possibility of such interleavings may constitute a 'risk', making the use of threads less attractive.

6.4 Restricting interleaving

Multi-threading being only a programming feature designed to simplify programming its use can always be avoided. That will sometimes be at the cost of a prohibitive increase of code size and code complexity. Taking for granted the usefulness of multi-threaded programming as a means to simplify code it is obvious that techniques to restrict the possibilities for interleaved execution will be needed in practice. The feature Java has on offer to prevent interleaving is the so-called synchronized method call.

6.4.1 Theads calling shared methods

We will first modify the previous example by having the thread bodies consist of method calls to the static class methods of the same class. This will require another form of calling the priority change. It now works on the current thread as provided by one of the static methods of the thread class.: JCF11 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      (new T()).start();
      (new T()).start();
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      C.body();}
}

class C {
static void body() {
      syo.p('a');
      Thc.sP(2);
      syo.p('b');
      }
```

```
}

class Thc extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}
```

). Upon execution this JCF produces

```
true
a
a
b
b
```

6.4.2 Theads calling synchronized methods

The next example is very close to the previous one. It has been copied in full detail in order to clearly establish the crucial point that making methods synchronized will restrict interleaving. JCF12 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      (new T()).start();
      (new T()).start();
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      C.body();}
}

class C {
synchronized static void body() {
      syo.p('a');
      Thc.sP(2);
      syo.p('b');
      }
}

class Thc extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}
```

). Upon execution this program produces

```
true
a
b
a
b
```

The two threads will compete for access to the synchonized method (or methods if there are more) of a class (in this case the class `C`). The winning thread acquires the so-called class lock giving it the unique right to execute a synchronized static method of the class.

By granting the class lock to only one thread at a time the possibilities for interleaved execution are reduced. This is illustrated in the example above.

6.5 Class locks and object locks

Besides class locks there are also object locks. The next example modifies the previous one by making the thread bodies call synchronized instance methods on the same object. The object mantains its own object lock.

6.5.1 Competing for the object lock

Threads intending to use an instance method of the same object have to compete for the object lock. JCF13 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static OC oc = new OC();
static void m(){
      (new T()).start();
      (new T()).start();
      syo.p(true);
      }
}

class T extends Thread {
```

```
public void run() {
      c.oc.body();}
}

class OC {
synchronized void body() {
      syo.p('a');
      Thc.sP(2);
      syo.p('b');
      }
}

class Thc extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}
```

). Upon execution this program computes

```
true
a
b
a
b
```

6.5.2 Different locks for different objects

If the same example is rewritten so as to use different objects for the call by the two threads, the locking effect disappears and interleaved execution takes place. JCF14 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static OC oc1 = new OC();
static OC oc2 = new OC();
static void m(){
      (new T(oc1)).start();
      (new T(oc2)).start();
      syo.p(true);
      }
}

class T extends Thread {
OC _oc;
T(OC oc) {
      _oc = oc;}
public void run() {
      _oc.body();}
}

class OC {
synchronized void body() {
      syo.p('a');
      Thc.sP(2);
      syo.p('b');
      }
}

class Thc extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}
```

). Upon execution this program computes

```
true
a
a
b
b
```

6.5.3 Locking a block

If an entire method to be synchronized, a thread intending to execute a method has to acquire the lock of the owner of that method (i.e. the method scheduled for execution). JavaTck will also allow a block (a portion of code surrounded by braces) to be protected by means of a lock. This is exemplified in the next case. It was derived from the previous one by integrating the priority update in the definition of the thread class, changing the technique used to initialize the instance field of the threads and, most importantly, by having the objects in class `OC` ask for a lock from the same new object of another class. As a result interleaving will not take place. JCF15 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static OC oc1 = new OC();
```

```
static OC oc2 = new OC();
static OCL l = new OCL();
static void m(){
      (new T(oc1)).start();
      (new T(oc2)).start();
      syo.p(true);
      }
}

class T extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
OC oc;
T(OC oc) {
      this.oc = oc;}
public void run() {
      oc.body();}
}

class OC {
void body() {
      synchronized(c.l) {
            syo.p('a');
            T.sP(2);
            syo.p('b');
            }
      }
}

class OCL { }
```

). Upon execution this program computes

```
true
a
b
a
b
```

6.5.4 Locking on independent objects

The previous example can be modified as follows: The two instances of `OC` will independently synchronize on different objects. As a consequence interleaving takes places. JCF16 = JCFp ∪ JCFsyo ∪

```
file:c.java(

      class c {
      static OC oc1 = new OC();
      static OC oc2 = new OC();
      static OCL l1 = new OCL();
      static OCL l2 = new OCL();
      static void m(){
            (new T(oc1,l1)).start();
            (new T(oc2,l2)).start();
            syo.p(true);
            }
      }

      class T extends Thread {
      static void sP(int i) {
            (Thread.currentThread()).
                  setPriority(i);
            }
      OC oc;
      OCL l;
      T(OC oc, OCL l) {
            this.oc = oc;
            this.l = l;}
      public void run() {
            oc.body(l);}
      }

      class OC {
      void body(OCL l) {
            synchronized(l) {
                  syo.p('a');
                  T.sP(2);
                  syo.p('b');
                  }
            }
      }

      class OCL { }
```

). This program computes

```
true
a
a
b
b
```

6.5.5 Locking on different objects II

The example is a simplified version of the previous example involving synchronized blocks using different object locks. The required threads are now taken from different classes. JCF17 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static OC oc1 = new OC();
static OC oc2 = new OC();
static OCL l1 = new OCL();
static OCL l2 = new OCL();
static void m(){
      (new T1()).start();
      (new T2()).start();
      syo.p(true);
      }
}

class TsP extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}

class T1 extends TsP {
public void run() {
      c.oc1.body(c.l1);}

}

class T2 extends TsP {
public void run() {
      c.oc2.body(c.l2);}
}

class OC {
void body(OCL l) {
      synchronized(l) {
            syo.p('a');
            TsP.sP(2);
            syo.p('b');
            }
      }
}

class OCL { }
```

). This program computes the same output as the previous one:

```
true
a
a
b
b
```

6.5.6 Superobjects have the same lock

Once more the setting is changed and one of the objects is made a superobject of the other. It turns out that casting an object to a superclass will not introduce a new lock. This explains the absence of interleaving in the next example. JCF18 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static OC oc1 = new OC();
static OC oc2 = new OC();
static OCLe l = new OCLe();
static void m(){
      (new T1()).start();
      (new T2()).start();
      syo.p(true);
      }
}

class TsP extends Thread {
static void sP(int i) {
      (Thread.currentThread()).
            setPriority(i);
      }
}

class T1 extends TsP {
public void run() {
      c.oc1.body(c.l);}

}

class T2 extends TsP {
public void run() {
```

```
      c.oc2.body((OCL) c.l);}
}

class OC {
void body(OCL l) {
      synchronized(l) {
            syo.p('a');
            TsP.sP(2);
            syo.p('b');
            }
      }
void body(OCLe l) {
      synchronized(l) {
            syo.p('a');
            TsP.sP(2);
            syo.p('b');
            }
      }
}

class OCL { }
class OCLe extends OCL {}
```

). This program computes the same as the previous one:

```
true
a
b
a
b
```

7 Thread inheritance

The connection between threads and inheritance merits some attention. First of all inheritance is used when creating threads. By extending the class `Thread` a method `start` is inherited which in turn will call a public method `run()` which can be overridden by the class definition. Of interest in this section is the effect of successive extensions of the `Thread` class. The following example contains a number of cases. JCF19 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      T1 p1 = new T1();
      T2 p2 = new T2();
      T3 p3 = new T3();

      T2 q2 = new T2();
      T3 q3 = new T3();
      T3 qb3 = new T3();

      T1 r1,s1;
      T2 r2;

      r1 = (T1) q2;
      s1 = (T1) q3;
      r2 = (T2) qb3;

      p1.setPriority(8);
      p2.setPriority(7);
      p3.setPriority(6);
      r1.setPriority(4);
      s1.setPriority(3);
      r2.setPriority(2);

      p1.start();
      p2.start();
      p3.start();
      r1.start();
      s1.start();
      r2.start();

      syo.p(true);
      }
}

class T0 extends Thread {}

class T1 extends T0 {
int i = 1;
int f() {
      return 1;}
int g(int j) {
      return j;}
public void run() {
      syo.pw(1);
      syo.pw(i);
      syo.pw(f());
      syo.pw(g(i));
```

```
        syo.pw(g(f()));
        syo.p('a');
        }
}

class T2 extends T1 {
int i = 2;
int f() {
        return 2;}
int g(int j) {
        return j;}
public void run() {
        syo.pw(2);
        syo.pw(i);
        syo.pw(f());
        syo.pw(g(i));
        syo.pw(g(f()));
        syo.p('b');
        }
}

class T3 extends T2 {
int i = 3;
int f() {
        return 3;}
int g(int j) {
        return j;}
public void run() {
        syo.pw(3);
        syo.pw(i);
        syo.pw(f());
        syo.pw(g(i));
        syo.pw(g(f()));
        syo.p('c');
        }
}
```

). The outputs are as follows:

```
1 1 1 1 1 a
2 2 2 2 2 b
3 3 3 3 3 c
true
2 2 2 2 2 b
3 3 3 3 3 c
3 3 3 3 3 c
```

On the basis of this example we can draw the following conclusions:

- Successive extensions of the class `Thread` can be developed, the method `run()` can be overridden in subclasses in the usual way.
- Just like other objects, threads (i.e. objects from a class extending `Thread`) can be assigned to names for objects in superclasses.
- All issues regarding overriding and inheritance for threads follow the same pattern as in the absence of threads. (In practice this means that geat care must be taken because not all aspects of method overriding in inheritance hierarchies follow a simple and clear intuition.)

7.1 Thread inheritance: further examples

In order to illustrate the conclusion that care must be taken when combining threads, inheritance and overriding some additional examples are included.

The authors acknowledge information obtained in the form of a concrete example from Bart Jacobs [7] by personal e-mail regarding the fact that changing access specifiers from public to private may have effects as drastic as is shown below.

7.1.1 Overrididing a public alias for `run()`

In the next example the method `run()` is aliased (by introducing `run1()`. The alias need not necessarily be a public method. Therefore it is possible to experiment with the effect of making that alias private. First, however, the case is considered that the alias is public. JCF20 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
        T3 p31 = new T3();
        T3 p32 = new T3();
        T3 p33 = new T3();

        T1 q1 = p31;
        T2 q2 = p32;
```

```
      syo.pw(q1.i);
      syo.pw(q2.i);
      syo.pw(p33.i);syo.p(p33.j);

      q1.start();
      q2.start();
      p33.start();

      syo.p('a');
      }
}

class T0 extends Thread {}

class T1 extends T0 {
public int i = 15;
private int j = 16;
public void run() {run1();}
public void run1() {
      syo.pw(1);
      syo.p(j);
      }
}

class T2 extends T1 {
public int i = 25;
private int j = 26;
public void run1() {
      syo.pw(2);
      syo.p(j);
      }
}

class T3 extends T2 {
public int i = 35;
public int j = 36;
public void run1() {
      syo.pw(3);
      syo.p(j);
      }
}
```

). The output produced by the compiled version of this JCF reads:

```
15 25 35 36
a
3 36
3 36
3 36
```

7.1.2 The 'Private' access specifier prevents overriding

In the next example the method `run1()` is made private; as a consequence the method `run()` is protected against overriding. We notice that for instance fields this protection is not needed as the same form of overriding will not take place for fields. (This is the reason for including `i` and `j`.) JCF21 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      T3 p31 = new T3();
      T3 p32 = new T3();
      T3 p33 = new T3();

      T1 q1 = p31;
      T2 q2 = p32;

      syo.pw(q1.i);
      syo.pw(q2.i);
      syo.pw(p33.i);syo.p(p33.j);

      q1.start();
      q2.start();
      p33.start();

      syo.p('a');
      }
}

class T0 extends Thread {}

class T1 extends T0 {
public int i = 15;
private int j = 16;
public void run() {run1();}
private void run1() {
      syo.pw(1);
      syo.p(j);
```

```
      }
}

class T2 extends T1 {
public int i = 25;
private int j = 26;
private void run1() {
      syo.pw(2);
      syo.p(j);
      }
}

class T3 extends T2 {
public int i = 35;
public int j = 36;
private void run1() {
      syo.pw(3);
      syo.p(j);
      }
}
```

). The output produced by the compiled version of this JCF reads:

```
15 25 35 36
a
1 16
1 16
1 16
```

8 Miscellaneous aspects of thread composition

The examples in this section demonstrate a variety of program executions, which may be of help for readers who want to sharpen their intuitions. All essentials have been discussed before. From the point of view of empirical semantics, the cases contribute to the completeness of coverage.

8.1 A static resource class

The class `rs` containing only static methods carries a number of methods from which thread behavior will be composed in a subsequent class. JCFresource = JCFp ∪ JCFsyo ∪ `file:rs.java`(

```
class rs {
static void f(char x) {
      syo.pw(x);
      syo.pw(x);
      syo.pw(x);
      syo.p(x);
      }

static void g(char x) {
      syo.pw(x);
      syo.pw(x);
      (Thread.currentThread()).
            setPriority(4);
      syo.pw(x);
      syo.p(x);
      }

static void h(char x) {
      syo.pw(x);
      syo.pw(x);
      (Thread.currentThread()).
            setPriority(6);
      syo.pw(x);
      syo.p(x);
      }

static synchronized void
i(char x) {
      syo.pw(x);
      syo.pw(x);
      syo.pw(x);
      syo.p(x);
      }

static void j(char x) {
      syo.pw(x);
      Thread.yield();
      syo.pw(x);
      Thread.yield();
      syo.pw(x);
      Thread.yield();
      syo.p(x);
      }

static void k(char x) {
      syo.pw(x);
```

```
        syo.pw(x);
        Thread.yield();
        syo.pw(x);
        syo.p(x);
        }

    static synchronized void
    l(char x) {
        syo.pw(x);
        Thread.yield();
        syo.pw(x);
        (Thread.currentThread()).
            setPriority(3);
        syo.pw(x);
        (Thread.currentThread()).
            setPriority(7);
        syo.p(x);
        }
    }
```

)

8.2 A collection of (potential) threads

The next JCF contains a suitably parametrized thread constructor. The class T can play the role of Thread in the sequel. (Making descriptions shorter and easier to read.) JCFthrc = JCFp ∪ JCFsyo ∪ `file:P.java`(

```
    class T extends Thread { }

    class P extends T {
    char _l;
    char _x;
    P(char l,char x) {
        _l = l;
        _x = x;}
    public void run() {
        if(_l == 'f') {rs.f(_x);}
        else{
        if(_l == 'g') {rs.g(_x);}
        else{
        if(_l == 'h') {rs.h(_x);}
        else{
        if(_l == 'i') {rs.i(_x);}
        else{
        if(_l == 'j') {rs.j(_x);}
        else{
        if(_l == 'k') {rs.k(_x);}
        else{
        if(_l == 'l') {rs.l(_x);}
        else{;}}}}}}}}
    }
```

)

8.3 A catalogue of examples

We write JCFu for JCFthrc ∪ JCFresource.

8.3.1 Effect of priority updates

Three f-threads together are executed in the order of their introduction (which is not guaranteed). JCF22 = jCFu ∪ `file:c.java`(

```
    class c {static void m() {
        P p = new P('f','a');
        P q = new P('f','b');
        p.start();
        q.start();
        rs.f('c');
        }
    }
```

). It computes

```
    c c c c
    a a a a
    b b b b
```

Three h-threads together: JCF23 = JCFu ∪ `file:c.java`(

```
    class c {static void m() {
        P p = new P('h','a');
        P q = new P('h','b');
        p.start();
        q.start();
        rs.h('c');
        }
    }
```

). It computes

```
c c c c
a a a a
b b b b
```

Three g-threads together: JCF24 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      P p = new P('g','a');
      P q = new P('g','b');
      p.start();
      q.start();
      rs.g('c');
      }
}
```

). It computes

```
c c a a b b b b
c c
a a
```

The combination h-f-k. JCF25 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      P p = new P('h','a');
      P q = new P('f','b');
      p.start();
      q.start();
      rs.k('c');
      }
}
```

). It computes

```
c c a a a a
b b b b
c c
```

The combination g-g-g. JCF26 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      P p = new P('g','a');
      P q = new P('i','b');
      p.start();
      q.start();
      rs.g('c');
      }
}
```

). It computes

```
c c a a b b b b
c c
a a
```

The combination g-h-g. JCF27 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      P p = new P('g','a');
      P q = new P('h','b');
      p.start();
      q.start();
      rs.g('c');
      }
}
```

). It computes

```
c c a a b b b b
c c
a a
```

8.3.2 The effect of `Thread.yield()`

Adding a yield action in the beginning of the main thread immediately causes it to hand over transfer to another one, (unsurprisingly but not neccessarily) the next one that was introduced during execution.

The combination f-f-i. JCF28 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      X p = new X('f','a');
      X q = new X('f','b');
      p.start();
      q.start();
      rs.i('c');
      }
}
```

). It computes

```
c a a a a
b b b b
c c c
```

The `Thread.yield()` action after 2 actions. The combination h-f-k. JCF29 = JCFu ∪ `file:c.java(`

```
class c {static void m() {
      P p = new P('h','a');
      P q = new P('f','b');
      p.start();
      q.start();
      rs.k('c');
      }
}
```

). It computes

```
c c a a a a
b b b b
c c
```

The combination j-i-j. JCF30 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('j','a');
      P q = new P('i','b');
      p.start();
      q.start();
      rs.j('c');
      }
}
```

). It computes

```
c a b b b b
c a c a c
a
```

The combination j-l-j. JCF31 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('j','a');
      P q = new P('l','b');
      p.start();
      q.start();
      rs.j('c');
      }
}
```

). It computes

```
c a b c a b c a c
a
b b
```

8.3.3 Effects of locking

Although the synchronized bodies contain several actions that can affect turntaking, none of these has any influence in this case. The active thread has acquired the so-called class lock, which is needed for other threads to perform their body. The class lock can only be released upon exit of the body, giving the changed priorities and thread vector mutations no chance to have any impact if all alternatives are in need of the same lock.

The combination i-h-h. JCF32 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('i','a');
      P q = new P('h','b');
      p.start();
      q.start();
      rs.h('c');
      }
}
```

). It computes

```
c c c c
a a a a
b b b b
```

The combination l-h-l. JCF33 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('l','a');
      P q = new P('h','b');
      p.start();
      q.start();
      rs.l('c');
      }
}
```

). It computes

```
c b b b b
c c c
a a a a
```

The combination h-j-l. JCF34 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('h','a');
      P q = new P('j','b');
      p.start();
      q.start();
      rs.l('c');
      }
}
```

). It computes

```
c a a a a
b c b b b
c c
```

8.3.4 Priority updates without effect

The following JCF demonstrates a case where a sequence of priority updates might be expected to change the thread vector. No effects are observed however. JCF35 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      P p = new P('f','a');
      P q = new P('f','b');
      p.setPriority(6);
      q.setPriority(4);
      q.setPriority(5);
      p.setPriority(5);
      p.start();
      q.start();
      rs.f('c');
      }
}
```

). It computes

```
c c c c
a a a a
b b b b
```

8.3.5 Priority updates can go wrong

The following JCF demonstrates a case where a priority update takes place at the wrong moment and an exception is raised. JCF36 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      X p = new X('f','a');
      X q = new X('f','b');
      p.setPriority(6);
      q.setPriority(4);
      p.start();
      q.start();
      q.setPriority(5);
      p.setPriority(5);
      rs.f('c');
      }
}
```

). It computes

```
a a a a
Exception in thread "main"/
java.lang.NullPointerException
  at java.lang.Thread./
  setPriority(Compiled Code)
  at c.m(Compiled Code)
  at s.main(Compiled Code)
b b b b
```

(The slash (/) indicates a transition to a new line that we have introduced in the rendering of the error message).

8.3.6 Priority updates need alive threads

The priority update is incorrect on a dead thread. If it is performed on a live thread, the exception is removed. The exception is no big problem, however, since it can be caught and ignored. JCF37 = JCFu ∪ `file:c.java`(

```
class c {static void m() {
      X p = new X('f','a');
      X q = new X('f','b');
      p.setPriority(6);
      q.setPriority(4);
      p.start();
      q.start();
      syo.p(p.isAlive());
      syo.p(q.isAlive());
      q.setPriority(5);
      rs.f('c');
      }
}
```

). It computes

```
a a a a
false
true
c c c c
b b b b
```

8.3.7 Run-time determination of thread priority

Thread priority cannot only be set but also be inspected. JCF38 = JCFu ∪ JCFpi ∪ `file:c.java`(

```
class c {static void m() {
      X p = new X('l','a');
      X q = new X('j','b');
      syo.pw(p.getPriority());
      syo.pw(q.getPriority());
      p.start();
      p.setPriority(4);
      syo.pw(p.getPriority());
      q.setPriority(6);
      syo.p(q.getPriority());
      q.start();
      rs.k('c');
      }
}
```

). It computes

```
5 5 4 6
b b b b
c c c c
a a a a
```

9 Deadlocks and deadlock prevention

This section concerns the occurrence of deadlocks in multi-threaded systems. Examples will be given for unavoidable deadlocks, deadlocks that do but need not occur, and of executions that might run into deadlock on a different platform or after being compiled with a different language specification compliant compiler. The following conclusions relate to this section:

- JCF's can clearly demonstrate the occurrence of deadlocks as well as livelocks. In the case of a deadlock the system cannot proceed because a thread needs to acquire a class lock that is in the possesion of another thread that cannot release it. In the case of a livelock the system spends all its time on giving the turn to another thread, not making any meaningful progress at all.
- Deadlocks will only occur if locks (synchronized methods or blocks) are used. Livelocks can occur without the presence of these synchronization primitives.
- The latest versions of Java declare `Deprecated` all methods which were originally proposed to remove a deadlocked thread. Detecting a deadlock and subsequently clearing up the mess is not an option in Java anymore. Deadlocks have to be prevented by means of careful programming techniques instead.
- The use of synchronization primitives is the primary tool for deadlock-free programming. In a sense these primitives are the source and the solution for deadlocks at the same time. Needless to say without the use of synchronization primitives even more basic problems than deadlocks may occur in the multi-threaded case. For that reason JCFsyo has been introduced.

9.1 Absence of a potential deadlock

The following JCF prints `true` until a run-time error occurs. Calling a synchronized method from within is apparently admissible . JCF39 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class T extends Thread {
public void run(){A.f();}
}

class A {
synchronized static void f() {
      syo.p(true);f();syo.p(false);}
}
```

```
class c {static void m() {
      T t = new T();
      t.start();
      }
}
```

).

9.1.1 Only one class lock

The next example illustrates once more the outcome of the previous example. Once a thread has acquired the class lock for a class, this can be used to enter the code for all of the synchronized methods of that class. It follows that a single-threaded system cannot deadlock, and, a fortiori, that the addition of the `synchronize` key-word to JavaCck of [5] will not introduce the possibility of writing deadlocking programs. JCF40 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class T extends Thread {
public void run(){A.f1();}
}

class A {
synchronized static void f1() {
      syo.pw(true);
      B.g1();
      syo.p(false);}
synchronized static void f2() {
      syo.pw('a');
      syo.pw('b');}
}

class B {
synchronized static void g1() {
      syo.pw(1);
      A.f2();
      syo.pw(2);}
}

class c {static void m() {
      T t = new T();
      t.start();
      }
}
```

). This JCF produces `true 1 a b 2 false`.

9.2 A deadlocking system

A deadlock occurs in the following example. The deadlock is not necessary, however. Its occurrence is caused by the execution order taken by the implementation. This execution order is not unavoidable, taking an alternative one will avoid the deadlock. JCF41 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){A.f1();}
}

class TB extends Thread {
public void run(){B.g1();}
}

class A {
synchronized static void f1() {
      syo.pw(1);
      Thread.yield();
      B.g2();
      syo.p(2);}
synchronized static void f2() {
      syo.pw(3);}
}

class B {
synchronized static void g1() {
      syo.pw('a');
      A.f2();
      syo.p('b');}
synchronized static void g2() {
      syo.pw('c');}
}

class c {static void m() {
      TA t = new TA();
      TB s = new TB();
      t.start();
      s.start();
      }
}
```

). This JCF produces `1 a` and thereafter does not return. This constitutes a deadlock. The system keeps waiting for a possibility to make progress.

If the instruction `Thread.yield();` is removed from JCF41, thus yielding JCF42, the deadlock disappears and the following output results:

```
1 c 2
a 3 b
```

The 'solution' is ad hoc and not satisfactory because the language specification will point out clearly that JCF41 may deadlock and that JCF42 need not run into a deadlock. By adding an instruction `Thread.yield();` as the second instruction of the body of method `B.g1()`, thus obtaining JCF43, a case is found which must unavoidably run into deadlock.[12] Whichever of the threads goes first, it will not acquire both locks for classes A and B in one row, because the other thread has taken the second lock.

9.2.1 A partial deadlock

In the next case the aforementioned deadlock also appears but another thread with a lower priority can progress once the higher-priority threads have engaged in their (irreversible) deadly embrace. After the lower-priority thread has terminated the deadlock surfaces again. JCF44 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){A.f1();}
}

class TB extends Thread {
public void run(){B.g1();}
}

class TC extends Thread {
public void run(){
      syo.p(true);
      }
}

class A {
synchronized static void f1() {
      syo.pw(1);
      Thread.yield();
      B.g2();
      syo.p(2);}
synchronized static void f2() {
      syo.pw(3);}
}

class B {
synchronized static void g1() {
      syo.pw('a');
      Thread.yield();
      A.f2();
      syo.p('b');}
synchronized static void g2() {
      syo.pw('c');}
}

class c {
static TA t = new TA();
static void m() {
      TB s = new TB();
      TC r = new TC();
      r.setPriority(4);
      t.start();
      s.start();
      r.start();
      }
}
```

) outputs `1 a true` and diverges thereafter.

9.2.2 Thematic deadlock removal

Using the features of JavaTck, we can easily avoid the deadlock of JCF43. By wrapping the bodies of the two threads in synchronized methods of the same class the entire execution of the threads becomes a critical section, immune to interference from the other thread. JCF45 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){C.h('a');}
}
```

[12] On the basis of the strict-rotation postulate the deadlock is a necessity, without this postulate it is merely the 'more likely' thing to happen.

```
class TB extends Thread {
public void run(){C.h('b');}
}

class C {
synchronized static void
      h(char x) {
            if(x == 'a')
            {A.f1();}else{
            if(x == 'b')
            {B.g1();}else{;}}}
      }

class A {
synchronized static void f1() {
      syo.pw(1);
      Thread.yield();
      B.g2();
      syo.p(2);}
synchronized static void f2() {
      syo.pw(3);}
}

class B {
synchronized static void g1() {
      syo.pw('a');
      Thread.yield();
      A.f2();
      syo.p('b');}
synchronized static void g2() {
      syo.pw('c');}
}

class c {static void m() {
      TA t = new TA();
      TB s = new TB();
      t.start();
      s.start();
      }
}
```

). The output of this JCF equals that of JCF42.

9.2.3 Synchronization as a cause

If the methods `f1(),f2(),g1(),g2()` are not defined `synchronized` no deadlock will occur: JCF46 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){A.f1();}
}

class TB extends Thread {
public void run(){B.g1();}
}

class A {
static void f1() {
      syo.pw(1);
      Thread.yield();
      B.g2();
      syo.p(2);}
static void f2() {
      syo.pw(3);}
}

class B {
static void g1() {
      syo.pw('a');
      Thread.yield();
      A.f2();
      syo.p('b');}
static void g2() {
      syo.pw('c');}
}

class c {static void m() {
      TA t = new TA();
      TB s = new TB();
      t.start();
      s.start();
      }
}
```

). The output of JCF46 equals:

```
1 a c 2
3 b
```

Without synchronized methods deadlocks cannot occur.[13]

[13] This seems obvious but we have no formal proof.

10 Livelock

A livelock occurs if the system performs an unbounded number of internal steps. From the external viewpoint it is like a deadlock, the difference being that some actions are performed all the time.

10.1 Unlimited output

The following program produces an infinite output, running into a loop. It repeats the same outputs forever: JCF47 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){A.f();}
}

class TB extends Thread {
public void run(){B.g();}
}

class A {
static void f() {
      boolean b = true;
      syo.pw(1);
      while(b){
            syo.pw(3);
            Thread.yield();}
      syo.p(2);}
}

class B {
static void g() {
      boolean b = true;
      syo.pw('a');
      while(b){
           syo.pw('c');
           Thread.yield();}
      syo.p('b');}
}

class c {static void m() {
      TA t = new TA();
      TB s = new TB();
      t.start();
      s.start();
      }
}
```

). The output of JCF47 equals:

```
1 3 a c 3 c 3 c 3 c 3 c 3 c
3 c 3 c 3 c 3 c 3 c 3 c 3 c
3 c 3 c 3 c ..............
```

The conclusion can be drawn that `yield()` will indeed hand over control to another thread.

10.2 A livelock

JCF47 is changed in such a way that the repeating output is removed. It now seems reasonable to assume that the same cycle is being executed all the time. Therefore, the next JCF provides an example of a livelock. JCF48 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class TA extends Thread {
public void run(){A.f();}
}

class TB extends Thread {
public void run(){B.g();}
}

class A {
static void f() {
      boolean b = true;
      syo.pw(1);
      while(b){
            Thread.yield();}
      syo.p(2);}
}

class B {
static void g() {
      boolean b = true;
      syo.pw('a');
      while(b){
           Thread.yield();}
      syo.p('b');}
}

class c {static void m() {
```

```
        TA t = new TA();
        TB s = new TB();
        t.start();
        s.start();
        }
}
```

). The output of JCF48 is `1 a`, followed by a long silence (the absence of a prompt indicating divergence.)

11 Shared variables

With OS[14] process (system process) we denote an entire compiled program during execution. The cooperation between OS processes takes place via sockets. Different OS processes cannot access the same variables because these are private for the individual programs. Different threads in the same program, however, can use the manipulation of the same (shared) variables as a major means of cooperation and communication. This is the most prominent difference between multi-threading and multiprocessing. From the perspective of the OS, multiple OS processes are much further apart than multiple threads (originating from the same JCF).

Indeed it is common for different threads to have access to some shared variables. If there are shared variables systems can reliably communicate provided updates of the shared variables are only performed via synchronized methods. By encapsulating each shared variable in its own class, allowing modifications only by means of sychronized methods, a reasonable platform for concurrent programming can be achieved. To illustrate this situation we will translate a parallel composition of sequential processes into JavaTck in a systematic manner. We consider a single example, leaving its more or less obvious generalization unspecified.

As a case study we will describe how logical processes (not: OS processes) in process-algebra notation can be translated into Java multi-threading. The logical process notation may be considered a design pattern from the perspective of the Java programmer. We can draw the following conclusions from the example below:

- Logical processes specified by means of systems of recursion equations can systematically be translated into Java in a satisfactory way. The translations depend on the package of compiler postulates one is willing to assume. Assuming fewer postulates forces one to write more complicated translation schemes.
- Object orientation provides a nice way to separate the context of a logical process (here called environment) and the control aspects of a logical process under translation. The translation will work for each environment.
- Logical process definitions can be far shorter than their Java counterparts. As a design pattern logical process definitions make sense.
- Translating communicating logical processes instead of incorporating logical processes into JavaTck seems much harder.[15]

11.1 Logical Processes

We consider the following process definition for `Z`:

```
X = X1
X1 = b1:-> p1 . X2
     o+ b2:-> p3
X2 = b3:-> p2

Y = Y1
Y1 = c1:-> q1 . Y2
     o+ c2:-> q2
Y2 = c1:-> q3
```

[14] OS stands for Operating System.

[15] Communication between logical processes takes place if two ore more processes share an action. The obvious example is a handshake taking place between two people. Either participant provides a part of the same action. Handshaking communication is the simplest form of communication between logical processes. It is simpler than the use of shared variables because it can do without the very notion of shared variables. For that reason deadlock-free programming is considered easier in program notations based on handshaking communication than in notations based on shared variables. It is common to refer to handshaking communication as synchronous communication and to refer to communication by means of shared resources as asynchrounous communication.)

```
Z = X || Y
```

In this notation `Z` is the parallel composition of `X1` and `Y1`. `X1` and `Y1` are determined by means of a system of recursion equations. In the right-hand sides `o+` denotes the skew alternative composition of two processes (taking the left one if its guard is valid, taking the right one otherwise). The operator `+` represents alternative composition; it leads to a choice between its alternatives, taking only a choice if its guards avaluate to `true` (if any guards are present). In the context of shared variables the alternative composition of two processes chooses non-deterministically a process with a valid guard.

The relation between skew alternative composition and alternative composition is as follows:

```
a o+ X = a
(a . X) o+ Y = a . X
(b:-> X) o+ Y = b:-> (X o+ Y) + !b:-> Y
```

The expressions `b1..5` and `c1,2` denote boolean expressions, made up from static boolean variables only. The actions `p1,..,4` and `q1,.,3` represent calls to synchronized static methods of a single class. These method calls can be considered atomic (in the context of the processes). Sequential composition is determined by '.'. The symbol `:->` combines a condition and a process into a conditional (or guarded) process. Only if the guard is valid will the process be performed. The system is free, however, to execute any first step of an alternative composition for which the guarding condition is true at the time of execution. If a guard is found true its subsequent action is performed without any other action (of other processes) occurring in between.

11.2 An environment

The class `e` contains the environment information. It determines the static variables, the interpretation of conditions and of actions. JCF49 = JCFp ∪ JCFsyo ∪ `file:e.java`(

```
class e {
static boolean w1 = true;
static boolean w2 = true;
static boolean w3 = true;
static boolean w4 = true;
static boolean w5 = true;

synchronized static boolean
      b1() {return w1;}
synchronized static boolean
      b2() {return w2;}
synchronized static boolean
      b3() {return w3;}
synchronized static boolean
      c1() {return w4;}
synchronized static boolean
      c2() {return w5;}

synchronized static void
      p1() {w4 = true;
            syo.p('b');}
synchronized static void
      p2() {syo.p('d');}
synchronized static void
      p3() {;}
synchronized static void
      q1() {w1 = true;
            w4 = false;
            syo.p('a');}
synchronized static void
      q2() {syo.pw('d');}
synchronized static void
      q3() {w3 = true;
            syo.p('c');}
}
```

).

11.3 Turning process definitions into threads

The following class contains definitions for threads corresponding to the two processes `X1` and `Y1`. The process definitions can be considered a design pattern for this piece of code. JCF50 = JCFp ∪ JCFsyo ∪ `file:T.java`(

```
class T extends Thread { }
```

```
class X extends T {
public void run() {X1();}
void X1() {
      while(!e.b1() &&
            !e.b2()){
            Thread.yield();}
      if(e.b1()){
            e.p1();X2();}
      else{e.p3();}
      }
void X2() {
      while(!e.b3()){
            Thread.yield();}
      e.p2();
      }
}

class Y extends T {
public void run() {Y1();}
void Y1() {
      while(!e.c1() &&
            !e.c2()){
            Thread.yield();}
      if(e.c1()){
            e.q1();Y2();}
      else{e.q2();}
      }
void Y2() {
      while(!e.c1()){
            Thread.yield();}
      e.q3();
      }
}

class c {
static void m() {
      e.w1 = false;
      e.w2 = false;
      e.w3 = false;
      e.w4 = true;
      e.w5 = false;

      X x = new X();
      Y y = new Y();
      x.start();
      y.start();
            }
      }
```

). The result of execution of this JCF is:

```
a
b
c
d
```

12 Recursive Processes

We consider the following more involved process definition for `U`:

```
X = X1
X1 = b1:-> p1 . X2
     o+ b2:-> p3. X3
     o+ b3:-> p4
     o+ b4:-> p2
X2 = b3:-> p2 . X1
     o+ b4:-> p1
X3 = p5

Y = Y1
Y1 = (c1:-> q1 o+ c2:-> q2) . Y2
Y2 = q3

Z = Z1
Z1 = r1 . Z1

U = X || Y || Z
```

12.1 The environment

The class `e` contains once more the environment information. It determines the static variables, the interpretation of conditions and of actions. JCF51 = JCFp ∪ JCFsyo ∪ `file:e.java`(

```
class e {
static boolean w1 = true;
static boolean w2 = true;
static boolean w3 = true;
static boolean w4 = true;
static boolean w5 = true;
```

```
synchronized static boolean
      b1() {return w1;}
synchronized static boolean
      b2() {return w2;}
synchronized static boolean
      b3() {return w3;}
synchronized static boolean
      b4() {return w3;}
synchronized static boolean
      c1() {return w4;}
synchronized static boolean
      c2() {return w5;}

synchronized static void
      p1() {w4 = true;
            syo.pw('b');}
synchronized static void
      p2() {syo.pw('d');}
synchronized static void
      p4() {;}
synchronized static void
      p5() {;}
synchronized static void
      p3() {;}
synchronized static void
      q1() {w1 = true;
            w4 = false;
            syo.pw('a');}
synchronized static void
      q2() {syo.pw('d');}
synchronized static void
      q3() {w3 = true;
            syo.pw('c');}
synchronized static void
      r1() {syo.p('z');}
}
```

).

12.2 Turning recursion equations into threads

The following class contains definitions for threads corresponding to the two processes `X1` and `Y1`. The process definitions can be considered a design pattern for this piece of code, just like in the simpler case above. The translation has been changed so as to provide a `yield()` action after each successful execution of a translated action (unless the thread has terminated). The purpose of this modification is to maximize the chances for all processes to have a turn.
JCF52 = JCFp ∪ JCFsyo ∪ `file:T.java`(

```
class T extends Thread { }
class X extends T {
public void run() {X1();}
void X1() {
      while(!e.b1() &&
            !e.b2() &&
            !e.b3() &&
            !e.b4()){
            Thread.yield();}
      if(e.b1()){
            e.p1();
            Thread.yield();
            X2();}
      else{
      if(e.b2()){
            e.p3();
            Thread.yield();
            X3();}
      else{
      if(e.b3()){
            e.p4();}
      else{e.p2();}}}
      }
void X2() {
      while(!e.b3() &&
            !e.b4()){
            Thread.yield();}
      if(e.b3()){
            e.p2();
            Thread.yield();
            X1();}
      else{e.p1();}
      }
void X3() {e.p5();}
}

class Y extends T {
public void run() {Y1();}
```

```
void Y1() {
      while(!e.c1() &&
            !e.c2()){
            Thread.yield();}
      if(e.c1()){e.q1();}
      else{e.q2();}
      Thread.yield();
      Y2();
      }

void Y2() {
      while(!e.c1()){
            Thread.yield();}
      e.q3();
      }
}

class Z extends T {
public void run() {Z1();}
void Z1() {
      e.r1();
      Thread.yield();
      Z1();
      }
}
```

).

12.3 Putting processes together

The three processes in multi-thread form are composed into a single program: JCF53 = JCFp ∪ JCFsyo ∪ JCF52 ∪ `file:T.java`(

```
class c {//defines U
static void m() {
      e.w1 = false;
      e.w2 = false;
      e.w3 = false;
      e.w4 = true;
      e.w5 = false;

      X x = new X();
      Y y = new Y();
      Z z = new Z();
      x.start();
      y.start();
      z.start();
      }
}
```

). The result of executing this program is an infinitely progressing computation producing

```
a z
b c z
d z
b z
d z
b z
...
...
```

12.4 Relaxing the current-thread-persistence postulate

The translations above from process-algebra-styled recursive equations depend on all three of the postulates on multi-threaded execution 1.8. In particular the current thread persistence postulate is problematic as it is not at all implied by any of the current language-specification documents. The translating program can be modified in such a way that the current-thread-persistence postulate is not relevant anymore, now becoming dependent on the priority-dominance-postulate instead. In order to obtain this modification all checks of guarding conditions are placed in a self-designed critical section mechanism. Priorities are used to implement that critical section (in combination with synchronized methods). The output generated by the modified program is identical to that of the previous one. (That implies that the program remedies a complication which did not actually materialize in the program before modification.) JCF54 = JCFCFp ∪ JCFsyo ∪ JCF52 ∪ `file:T.java`(

```
class T extends Thread { }

class guardSynch {
synchronized static boolean
gS(Thread t, boolean b){
      if(b){return true;}
      else{ t.setPriority(6);
```

```
            return false;}
      }
}

class X extends T {
public void run() {X1();}
void X1() {
      while(guardSynch.gS(this,
            !e.b1() &&
            !e.b2() &&
            !e.b3() &&
            !e.b4())){
            Thread.yield();}
      if(e.b1()){
            e.p1();
            this.setPriority(5);
            Thread.yield();
            X2();}
      else{
      if(e.b2()){
            e.p3();
            this.setPriority(5);
            Thread.yield();
            X3();}
      else{
      if(e.b3()){
            e.p4();}
      else{e.p2();}}}
      }
void X2() {
      while(guardSynch.gS(this,
            !e.b3() &&
            !e.b4())){
            Thread.yield();}
      if(e.b3()){
            e.p2();
            this.setPriority(5);
            Thread.yield();
            X1();}
      else{e.p1();}
      }
void X3() {e.p5();}
}

class Y extends T {
public void run() {Y1();}
void Y1() {
      while(guardSynch.gS(this,
            !e.c1() &&
            !e.c2())){
            Thread.yield();}
      if(e.c1()){e.q1();}
      else{e.q2();}
            this.setPriority(5);
      Thread.yield();
      Y2();
      }

void Y2() {
      while(guardSynch.gS(this,
            !e.c1())){
            Thread.yield();}
      e.q3();
      this.setPriority(5);
      }
}

class Z extends T {
public void run() {Z1();}
void Z1() {
      guardSynch.gS(this,false);
      e.r1();
      this.setPriority(5);
      Thread.yield();
      Z1();
      }
}

class c {//defines U
static void m() {
      e.w1 = false;
      e.w2 = false;
      e.w3 = false;
      e.w4 = true;
      e.w5 = false;

      X x = new X();
      Y y = new Y();
      Z z = new Z();
      x.start();
      y.start();
      z.start();
```

```
        }
    }
```

).

13 Exception handling

Some features of multi-threaded Java may lead to exceptions which then have to be dealt with. To have a self-contained exposition we first provide some discussion of exceptions.

Exception handling is an important mechanism for computer programming. Well-known is the Java exception handling feature, largely inherited from the C++ programming language. Some important aspects of exception handling will be introduced and reviewed, using the Java programming language as a representation format for examples as well as for principles.[16]

It has been tried to provide as clearly as possible a description of what exceptions are and why their use is considered so attractive in programming. After all it is easy to imagine languages completely lacking the concept of an exception. It will be argued that the concept of an exception and of its proper handling emerges from the insistence on modularization, and from the objective of separation of concerns.

13.1 Exceptions defined

An exception is an output state for a computation which has been labeled an exception as a part of the design of the system carrying out the computation. More specifically the exception is an object containing information about the mentioned output state. It is reasonable to view exceptions as instances of specific exception classes. The very fact that exceptions encapsulate data that can be moved through a system indicates the plausibility of considering them objects. Exceptions testify that something has happened, rather than 'being themselves that event'.

[16] It has not been our intention to give a complete introduction to exception handling in Java. Complete coverage has been rated of lesser importance than clarity of exposition of fundamental aspects. The present focus is on aspects relating to threads.

13.2 Exceptions motivated: pragmatic aspects

The reason for using exceptions is that it has been practically useful to divide possible output states of a computation into two disjoint categories: outputs reporting the positive completion of a computation, including values that have been produced, and outputs reporting that 'something went wrong'. Going 'wrong' expresses the fact that during the computation evidence has emerged that the computation cannot be completed in an orderly fashion, for instance because the required data were found absent in a directory whish was supposed to contain them.

The important point is this: if abstract data types, or well-structured objects are used to format intended outputs, their structure will be unacceptably complicated by adding the structure of exception objects. There is no apriori relationship between the structure of 'positive' outputs and the structure of reports about things that went wrong (and were considered exceptional). Separating exceptions from correct outputs allows a separation of concerns for the design of output classes and of exception classes. Java proposes to inherit all exceptions from an original class exception. This would be too restrictive for output in general. It is therefore primarily a pragmatic matter of design modularization and separation of concerns, combined with all advantages for maintenance and use.

The abovementioned motivation for the use of exceptions is remarkably insensitive to the assumption that exceptions are rare. If exceptions are not rare, they are even more needed, the argument of their irregular structure, overly complicating the logic of object design, not being changed. (Even if one's printed crashes every second job, it would be justified to view the matter as an exception! The software report is an exception, unfortunately the event is not. That, however, will change for the better in time.)

13.3 Exceptions motivated: fundamental aspects

Along the fundamental arguments that can be put forward to motivate the use of exceptions it can be

mentioned that some program instructions bring the executing machine in a state that it will reside in for quite some time. During that time some feedback information may be generated about what is going on (e.g. a printer has no paper). This information cannot be predicted by the programmer, and often no test is available that can be used to check in advance whether a (faulty) condition will occur. In the case of popping an empty stack, the exception (`EmptyStackPopException`) can be avoided simply by testing in advance that the stack is non-empty. There may be no simple test at hand predicting that a long numerical computation will be dividing by zero. Again an `IllegalDivisionByZeroException` can easily be avoided by having all division preceded with appropriate tests. Nevertheless, such a test may be quite difficult to install in the case of third-party software. Ultimately, program execution may suffer from unpredictable hardware failure. Such failures can sometimes be observed. Then they can be meaningfully rendered as exceptions, cautious software engineering not being a strategy to avoid them .

13.4 Catching a self-defined exception

The next JCF introduces a new class of exceptions (called `Exceptional`). Consider JCF55 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m()
throws Exceptional {
      throw new Exceptional();}
}

class Exceptional
extends Exception {
public String toString() {
      return "exceptional case!";
      }
}
```

). The generated output reads

```
Exception in thread "main"\\
exceptional case!
        at c.m(Compiled Code)
        at s.main(Compiled Code)
```

13.5 Method overriding and Exceptions

If one method overrides another one the overriding method can only throw exceptions also thrown by the overridden one. The same holds for implementing an interface. Interface members may be specified to throw exceptions, their realizations in an implementation should throw the same errors.[17] As an example consider JCF56 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c extends C {
static void m()
throws Exceptional {
      throw new Exceptional();
      }
}

class C {
static void m()
throws Exceptional { }
}

class Exceptional
extends Exception {
public String toString() {
      return "exceptional case!";
      }
}
```

). The throws clause in the method `c.m()` is needed to allow the class extension from `C` to `c`. The output of the program is identical to the output of the previous program.

13.5.1 Threads cannot throw exceptions

As `run()` does not throw specific exceptions it cannot be overridden with any method throwing an existing or a user-defined exception. For that reason a running thread will not throw an exception. All exceptions thrown during its life must be caught and dealt with.

[17] More precisely: the realizations should announce the option to throw the same errors. Whether it will ever happen is of no importance.

This is a significant simplification. Indeed it would be quite unclear where to return to after an exception was raised.[18]

13.6 Handling an exception

Exceptions can be explicitly caught by means of the `try{...}catch(...){...}` construct. When caught a piece of code can be executed taking the exception as a parameter. In that case the string value of the exception will not be produced. Consider JCF57 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m()
throws Exceptional {
      try{  throw
            new Exceptional(
                  true);
      }
      catch(Exceptional e){
            syo.p(!e._b);
      }
      throw
      new Exceptional(true);
      }
}

class Exceptional
extends Exception {
boolean _b = true;
Exceptional(boolean b) {
      _b = b;
      }
public String toString() {
      if(_b){return "true";}
      else{return "false";}
      }
}
```

). It computes

```
 false
 Exception in thread "main" true
         at c.m(Compiled Code)
               at s.main(Compiled Code)
```

Java allows one to handle different cases of exceptions (depending on the class of the exception) by means of a number of successive `catch(..)` clauses. This may be considered syntactic sugar as it could be explicitly programmed using the `instanceof` operator. Another feature of Java is to allow a `finally{..}` clause following the catch clauses. The body of this instruction is executed after the preceding `try{..}` block, irrespective of whether an exception has been caught and handled.

The exception-handling code (the body of a `catch(E e){..}` clause has the option to rethrow the exception (or to throw any other exception.) There may seem to be an unfortunate feature interaction between rethrowing and the `finally{..}` option. An example indicates its resolution by effecting `finally{..}` before rethrowing an exception. In the example the `InterruptedException` is used. It is a standard exception for Java that can (for that reason) be used without definition. JCF58 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
    try{
      try{
            syo.pw(true);
            throw
            new InterruptedException();
      }catch(
      InterruptedException e) {
            syo.pw('a');
            throw e;}
      finally{syo.pw('b');}
    }catch(Exception e) {syo.pw('c');}
    finally{syo.pw('d');}
    syo.p(false);
    }
}
```

). It computes

```
true a b c d false
```

[18] In other words: where to put a corresponding `catch()` statement.

13.7 Throwing a thread as an exception

As an exception is evidently always caught within a method, its definition can go without the `throws ..` clause. This is exemplified in a case showing simultaneously that an exception can itself be a thread. In order to make objects that are both exceptions and threads the `Runnable` interface has to be implemented for an extension of the exception class. The syntax speaks more or less for itself. A key point is that a parametrized constructor will be needed for threads in a constructor for the exception class implementing `Runnable`. By giving `this` as a parameter it is established that `start()` will call the overriding version of `run()`. JCF59 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m() {
      try{
      throw new Exceptional();
      }catch(Exceptional e){
      (e.t).start();}
      }
}

class Exceptional
extends Exception
implements Runnable {
Thread t;
Exceptional() {
      t = new Thread(this);
      }
public void run() {
      syo.p(true);
      }
public String toString() {
      return "exceptional case!";
      }
}
```

). Its output simply consists of `true`. It should be noticed that the `Runnable` interface comes in as a 'magic' trick to get around the fact that Java is a single inheritance based program notation.[19]

14 Waiting for a thread to terminate

One thread may wait for another one to terminate, only to make progress once this termination has been observed. This mechanism is invoked by means of the `join()` method, to be applied to the thread whose termination is awaited. JCF60 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      T p = new T();
      p.start();
      try{
           p.join();
      }catch(
      InterruptedException e) { };
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      syo.pw('a');
      }
}
```

). It outputs `a true`. Because the `join()` method call can encounter an exceptional situation if its target process is interrupted, the `join()` method can give rise to an exception. As Java requires the exception to be explicitly 'thrown' by any method which can generate it unless it is explicitly 'caught' the exception will feature in the program text in some form or another. It is possible to define a local version of `join()`, moving the exception handling inside. As `join()` is a final method a new name is needed. JCF61 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
```

[19] A symmetric solution is conceivable in principle: taking a `Thread` extension to implement the `Exception` interface, if that were available in the language.

```
static void m(){
      T p = new T();
      p.start();
      p.Join();
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      syo.pw('a');
      }
void Join() {
      try{
          join();
      }catch(
      InterruptedException e) { };
      }
}
```

). It also outputs `a true`.

14.1 Aliasing `join()`

By extending Thread to TJ first and introducing `Join()` there, a class is obtained from which subsequent threads can be inherited in case the programmer prefers `Join()` throughout.[20] Let JCFaj = `file:TJ.java`(

```
class TJ extends Thread {
void Join() {
      try{join();}catch(
      InterruptedException e) { };
      }
}
```

). Then consider JCF62 =JCFaj ∪ JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      T p = new T(true);
      p.start();
      T q = new T(false);
      q.start();
      T2 r = new T2();
      r.start();
      p.Join();
      syo.pw(false);
      q.Join();
      r.Join();
      syo.p(true);
      }
}

class T2 extends TJ {
public void run() {syo.pw(0);}}

class T extends TJ {
boolean b = true;
T(boolean b) {this.b = b;}
public void run() {
      if(b){syo.pw('a');}
      else{syo.pw('b');}
      }
}
```

). This JCF produces `a b 0 false true`.

14.1.1 Joining dominates priorities

If a high-priority thread joins a low-priority thread its actions must wait. JCF63 =JCFaj ∪ JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static void m(){
      Q q1 = new Q('a');
      q1.setPriority(2);
      q1.start();
      Q q2 = new Q('b');
      q2.setPriority(3);
      q2.start();
      P r = new P('c',q1);
      r.setPriority(4);
      r.start();
      r.Join();
      syo.p(true);
      }
}
```

[20] This is relevant only if the cause of the exception is clearly absent.

```
class P extends TJ {
char _c;
TJ t;
P(char c,TJ t) {
      this.t = t;
      _c = c;}
public void run() {
      syo.pw(_c);
      syo.pw(_c);
      t.Join();
      syo.pw(_c);
      syo.pw(_c);
      }
}

class Q extends TJ {
char _c;
Q(char c) {_c = c;}
public void run() {
      syo.pw(_c);
      syo.pw(_c);
      syo.pw(_c);
      syo.pw(_c);
      }
}
```

). The following output is computed:
`c c b b b b a a a a c c true`.

14.1.2 `InterrruptedException` not easily generated

The next JCF illustrates that an `InterrruptedException` is not simply generated by the `interrupt()` method of the `Thread` class. The execution of the next JCF is somewhat hard to understand! JCF64 =JCFaj ∪ JCPp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static T p = new T();
static void m(){
      p.start();
      try{
           p.join();
      }catch(
      InterruptedException e) {
            syo.p('d');}
      syo.p(true);
      }
}

class T extends Thread {
public void run() {
      syo.pw('a');
      c.p.interrupt();
      syo.pw(interrupted());
      syo.p(isInterrupted());
      syo.pw('b');
      c.p.interrupt();
      syo.pw(isInterrupted());
      syo.p(interrupted());
      c.p.interrupt();
      syo.pw('c');
      }
}
```

) This JCF produces

```
a true false
b true true
c true
```

15 Handing over object locks

Object locks have been discussed in section 6.5 above. Their functionality was similar to that of class locks though far more flexible due to object creation. Synchronized methods applied to an object must compete for the lock of that object. Having been granted the lock, the method can start its run. An interesting option offered by JavaTck is to make the object lock in a sense transferable. Objects allow a method `wait()` which, when called, temporarily[21] releases the object lock for use by another thread. A second thread may now obtain the object lock just released and start the execution of a second synchronized method on the object. If at some time the second thread performs an action `notify` (of the aforementioned object) the first thread may proceed

[21] Until further notification.

with its execution of the body of its synchronized method. This simply means that the method call `wait()` can return. When waiting the thread might be interrupted. This would make the `wait()` fail. In order to give the first synchronized method an option to deal with that, the run-time system will generate an exception which is then thrown by `wait()`.

15.1 An alias for `wait()`

As we are not trying to illustrate the functionality of exceptions here, the examples would be needlessly complicated by the aspect of exception handling. Therefore, objects meant to carry a lock will inherit from OW, an extension of the Object class which uses an alias for `wait()` ignoring exceptions. JCFow = `file:TJ.java`(

```
class OW {
void Wait() {
      try{wait();}catch(
      InterruptedException e) { };
      }
}
```

).

15.2 The object lock compromised

The first illustration of the effect of `wait()` is an example indicating that two synchronized methods of one object can be simultaneously active using `wait()`. The following JCF contains a class with objects allowing synchronization. JCFsob1 = JCFow ∪ `file:R.java`(

```
class R extends OW {
synchronized void r1() {
      syo.p(0);
      Wait();
      syo.p(1);
      }
synchronized void r2() {
      syo.pw('a');
      notify();
      syo.p('b');
      }
}
```

). In order to let this class play its role the following context is provided.[22] JCFcontext =JCFaj ∪ JCFow ∪ JCFp ∪ JCFsyo ∪ `file:c.java`(

```
class c {
static R t = new R();
static void m(){
      T1 p = new T1();
      T2 q = new T2();
      p.start();
      q.start();
      syo.p(true);
      }
}

class T1 extends TJ {
public void run() {
      c.t.r1();}
}

class T2 extends TJ {
public void run() {
      c.t.r2();}
}
```

). Let JCF65 = JCFcontext ∪ JCFsob1. This JCF can be compiled and executed. The resulting output is:

```
true
0
a b
1
```

There is no other explanation for this run than that both methods `r1()` and `r2()` are active in a truly interleaved fashion. This indicates that the lock has been compromised, the very purpose of `wait()`.

15.3 It might go wrong

The program from the previous example can be modified by changing the order of introduction and initialization of the threads. In principle this does not matter, in practice the difference is substantial. Let JCF66 = JCFcontext ∪ `file:R.java`(

[22] The naming for the JCF's used is very explicit because various classes will be reused in subsequent examples.

```
class R extends OW {
synchronized void r1() {
      syo.p(0);
      notify();
      syo.p(1);
      }
synchronized void r2() {
      syo.pw('a');
      Wait();
      syo.p('b');
      }
}
```

). The resulting computation generates

```
true
0
a b
```

Subsequently a deadlock has occurred. The absence of this deadlock in previous examples has been caused by a (fortunate) style of interleaving.

15.3.1 Notification is needed

The body of the second method can be changed by removing the `notify()` action. The second thread can then be granted the lock, terminate and return it. However, this will not work becasue a deadlock results. The `wait()` action will not return unless a notification is given. Let JCF67 = JCFcontext ∪ `file:R.java`(

```
class R extends OW {
synchronized void r1() {
      syo.p(0);
      Wait();
      syo.p(1);
      }
synchronized void r2() {
      syo.pw('a');
      syo.p('b');
      }
}
```

). The resulting computation produces

```
true
0
a b
```

Subsequently it deadlocks as the waiting thread cannot restart.

15.3.2 Alternating handovers

Two threads can alternately wait for one another to provide a notification. Let JCF68 = JCFcontext ∪ `file:R.java`(

```
class R extends OW {
synchronized void r1() {
      syo.pw(0);
      Wait();
      syo.pw(1);
      notify();
      syo.pw(2);
      Wait();
      syo.p(3);
      }
synchronized void r2() {
      syo.pw('a');
      notify();
      syo.pw('b');
      Wait();
      syo.pw('c');
      notify();
      syo.p('d');
      }
}
```

). The resulting computation produces

```
true
0 a b 1 2 c d
3
```

15.3.3 Notifications may get lost

Two subsequent notifications may 'fuse' into one. As a result a deadlock may occur as follows. Let JCF69 = JCFcontext ∪ `file:R.java`(

```
class R extends OW {
synchronized void r1() {
      syo.pw(0);
```

```
        Wait();
        syo.pw(1);
        Wait();
        syo.pw(2);
        notify();
        syo.p(3);
        }
    synchronized void r2() {
        syo.pw('a');
        notify();
        syo.pw('b');
        notify();
        syo.pw('c');
        Wait();
        syo.p('d');
        }
}
```

). The resulting computation produces

```
true
0 a b c 1
```

Subsequently the computation deadlocks. Apparently the second notification has not been saved for later use. It cannot resolve the second complementary `wait()`.

15.3.4 Mutual exclusion still works

After a handover (a `wait()` action allowing temporary possesion of the lock by another thread) the second thread performing a synchronized method on the same object must now proceed until termination or until its next `wait()` action. In both cases it releases the lock (albeit temporary). In the next example two techniques are tried that may induce a thread to give control to the other thread prematurily after handover. Using the `currentThread()` method for the `Thread` class it can set its own priority low, or it may try rotation using a `yield()`. These attempts fail and the thread in control must proceed irrespective of adverse priority or attempts to rotate. Let JCF70 = JCFcontext ∪ `file:R.java`(

```
class R extends OW {
synchronized void r1() {
        syo.pw(0);
        Wait();
        syo.pw(1);
        syo.pw(2);
        syo.p(3);
        }
    synchronized void r2() {
        syo.pw('a');
        notify();
        syo.pw('b');
        Thread.yield();
        syo.pw('c');
        (Thread.currentThread()).
              setPriority(2);
        syo.p('d');
        }
}
```

). The resulting computation gives

```
true
0 a b c d
1 2 3
```

16 Global notification

A thread can notify several other threads at the same time. It is all or nothing (i.e. just one), there is no notification subscription service. A first example of `notifyAll()` illustrates the principle of the mechanism. Several aspects are illustrated (i) if several threads are waiting it need not be the highest priority one that is reactivated after a `notifyAll()` command,[23] (ii) two waiting threads are reactivated by a single `notifyAll()` command. (iii) the `notifyAll()` command can be mixed with the `notify()` command. Let JCF71 = JCFcontext ∪ `file:R.java`(

```
class c {
static Rgn r = new Rgn();
static void m(){
        Ta Pa = new Ta();
        T0 P0 = new T0();
        Tf Pf = new Tf();
```

[23] Rather one observes a stacking mechanism where `wait()` is like a recursive call to another thread.

```
      Pa.setPriority(6);
      P0.setPriority(7);
      Pf.setPriority(8);

      Pa.start();
      P0.start();
      Pf.start();

      syo.p(true);
      }
}

class Ta extends Thread {
public void run() {
      c.r.runa();
      }
}

class T0 extends Thread {
public void run() {
      c.r.run0();
      }
}

class Tf extends Thread {
public void run() {
      c.r.runf();
      }
}

class Rgn extends OW {
synchronized void runa() {
      syo.pw('a');
      syo.pw('b');
      Wait();
      syo.pw('c');
      syo.p('d');
      }
synchronized void run0() {
      syo.pw(0);
      syo.pw(1);
      Wait();
      syo.pw(2);
      syo.p(3);
      notify();
      }
      synchronized void runf() {
            syo.pw('f');
            notifyAll();
            syo.pw('g');
            syo.pw('h');
            Wait();
            syo.p('i');
            }
      }
```

). The resulting computation gives

```
a b 0 1 f g h 2 3
i
c d
true
```

16.1 Replacing global notification by 'local' ones

One can modify the previous example by changing the body of class `Rgn` so as to replace the single occurrence of `notifyAll()` by several occurences of `notify()`. Let JCF72 = JCFcontext ∪ `file:R.java` in which `Rgn` has been modified as follows: (

```
class Rgn extends OW {
      synchronized void runa() {
            syo.pw('a');
            syo.pw('b');
            Wait();
            syo.pw('c');
            syo.p('d');
            }
      synchronized void run0() {
            syo.pw(0);
            syo.pw(1);
            Wait();
            syo.pw(2);
            syo.p(3);
            notify();
            }
      synchronized void runf() {
            syo.pw('f');
            notify();
            notify();
            syo.pw('g');
```

```
        syo.pw('h');
        Wait();
        syo.p('i');
        }
}
```

) JCF72 has the same output as the earlier examples.

17 Blocking actions and pipes

A blocking action is a method call which will be executed only after some condition has become true. In the mean time the thread responsible for the call of that action can hand over its turn to another thread. Synchronized method calls are examples of blocking actions, but not the only ones. Another example is found in the use of pipes. Pipes are unidirectional communication channels with the behavior of a reliable FIFO queue. The queue may have a bounded capacity, ts maximum length depending on aspects of the implementation and probably not known in advance to the programmer. Reading from a pipe is a blocking action. If the pipe is empty the read action will wait until a message has arrived. Writing can block if the pipe is full. Another example of blocking actions is found in the use of sockets. Sockets allow communication between programs runnning on different machines. We will illustrate pipes and blocking effects below. Sockets will not be discussed, however, the 'blocking aspects' of sockets being similar to those of pipes.

17.1 Construction of a connected pipe

The example below provides a simple use of pipes. Java requires an object made for the input side of the pipe as well as an object attached to its output side. These objects must be explicitly connected. JCF73 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
import java.io.*;

class c {
static void m() {
  try{
      PipedInputStream x =
         new PipedInputStream();
      PipedOutputStream y =
         new PipedOutputStream();
      y.connect(x);
      y.write(3);
      y.write(5);
      int z = 0;
      z = x.read();
      syo.pw(z);
      z = x.read();
      syo.p(z);
     } catch (Exception e) { };
  }
}
```

). The resulting computation produces

```
3 5
```

17.2 Blocked reading demonstrated

The next example documents the fact that even in the presence of a difference in priority a thread cannot proceed if it is an empty pipe that is being read. In such a case, however, control is transferred to another thread if possible. JCF74 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
import java.io.*;

class c {
static PipedInputStream x =
      new PipedInputStream();
static PipedOutputStream y =
        new PipedOutputStream();
static S1 p = new S1();
static S2 q = new S2();

static void m() {
  try{
      y.connect(x);
     }catch (Exception e) { };
      p.start();
      p.setPriority(4);
      q.start();
      q.setPriority(6);
      syo.p(true);
```

```
  }
}

class S1 extends Thread {
public void run() {
  try{
      syo.p('c');
      c.y.write(3);
      syo.p('d');
      c.y.write(5);
     }catch (Exception e) { };
  }
}

class S2 extends Thread {
public void run() {
  try{
      int z = 0;
      syo.pw('a');
      z = c.x.read();
      syo.pw(z);
      syo.pw('b');
      z = c.x.read();
      syo.p(z);
     } catch (Exception e) { };
   }
}
```

). The resulting computation produces

```
a true
c
d
3 b 5
```

17.3 One to one interconnection

It is tempting to interconnect different pipes, thus allowing forks and joins. Although the compiler accepts such experiments, run-time errors arise immediately. It can be concluded that for pipes ony one to one connections of input objects and output objects make sense. As an example consider: JCF75 = JCFp ∪ JCFsyo ∪ `file:c.java`(

```
import java.io.*;

class c {
static PipedInputStream x =
      new PipedInputStream();
static PipedOutputStream y =
        new PipedOutputStream();
static PipedOutputStream y1 =
        new PipedOutputStream();
static S1 p = new S1();
static S2 q = new S2();

static void m() {
  try{
      y.connect(x);
      y1.connect(x);
     }catch (Exception e) { };
      p.start();
      p.setPriority(4);
      q.start();
      q.setPriority(6);
      syo.p(true);
  }
}

class S1 extends Thread {
public void run() {
  try{
      syo.p('c');
      c.y.write(3);
      syo.p('d');
      c.y1.write(5);
     }catch (Exception e) {
      syo.p(7);};
  }
}

class S2 extends Thread {
public void run() {
  try{
      int z = 0;
      syo.pw('a');
      z = c.x.read();
      syo.pw(z);
      syo.pw('b');
      z = c.x.read();
      syo.p(z);
     } catch (Exception e) { };
```

```
    }
}
```

). The corresponding computation produces

```
a true
c
d
7
```

For more information on the use of pipes and sockets we refer to [9] and [10].

18 Concluding remarks

Several aspects of the Java multi-threading subject have not been discussed, notably: interrupts, timers, object locks, thread groups. Many more feature combinations could have been investigated. Nevertheless we feel that a meaningful survey of elementary aspects of Java multi-threading has been obtained in the previous pages. Further reading on multi-threading can go in different directions. The original texts on Java should be mentioned, e.g. [1] but also the discussion of multi-threading from the perspective of Unix and POSIX based on the C programming language [11].

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addisson-Wesley, New York, 1996.

[2] J.A. Bergstra and P. Klint. The discrete time ToolBus–a coordination architecture. *Science of Computer Programming*, 31:205–229, 1998.

[3] J.A. Bergstra and J.-W. Klop. Process algebra for synchronous communication. *Information and Control*, 60 (1/3):109–137, 1984.

[4] J.A. Bergstra and M.E. Loots. Program algebra for component code. Technical Report P9811, Programming Research Group, University of Amsterdam, 1998.

[5] J.A. Bergstra and M.E. Loots. Empirical semantics for object-oriented programs. Technical report, Lecture Notes, Programming Research Group, University of Amsterdam, 1999.

[6] J.A. Bergstra and M.E. Loots. Programs, interfaces and components. Technical report, Lecture Notes, Programming Research Group, University of Amsterdam, 1999.

[7] B.Jacobs. A counter-example to PCA, 1999. Personal communication.

[8] J.W. de Bakker. *Mathematical theory of prgram correctness*. Prentice Hall International, 1980.

[9] E. Rusty Harold. *Java I/O*. O'Reilly, 1999.

[10] M. Hughes, M. Shoffer, and D. Hammer. *Java Network Programming*. Manning, Greenwich USA, 1999.

[11] S. Kleiman, D. Shah, and B. Smaalders. *Programming with Threads*. SunSoft Press, Prentice Hall International, Mountain View, California, 1996.

[12] S. Oaks and H. Wong. *Java Threads*. O'Reilly, 1997.