Java: Memory Consistency and Process Coordination*

Lisa Higham†and Jalal Kawash‡
Department of Computer Science, The University of Calgary, Canada, T2N 1N4
Fax: +1 (403) 284 4707, Phone: +1 (403) 220 7696, 220 7681
{higham|kawash}@cpsc.ucalgary.ca

Abstract

In Java, some memory updates are necessarily visible to some threads but never to others. A definition of Java memory consistency must take this fact into consideration to capture the semantics of non-terminating systems, such as a Java operating system. This paper presents a programmer-centered formal definition of Java memory behavior that captures those semantics.

Our definition is employed to prove that it is impossible to provide fundamental process coordination in Java, such as critical sections and producer/consumer coordination, without the use of the `synchronized` and `volatile` constructs. However, we show that a weaker form of synchronization suffices to solve some of these problems in Java.

keywords: Java, Java Virtual Machine, memory consistency models, process coordination, critical section problem, producer/consumer problem, non-terminating systems.

1 Introduction

The Java Virtual Machine (JVM) [15] provides a global shared memory and a local memory for each Java thread. Because intricate rules (see Appendix A) determine the communication between these memories, and because much of this communication is optional (at the discretion of the implementor), the possible behaviors of multi-threaded Java programs are complicated. For instance, memory accesses can be visible to some threads but not to others [15, 8] (henceforth called the *invisibility* phenomenon.) These complicated interactions of threads and memories make it imperative to provide programmers with a formal and precise definition of the memory behavior of JVM. The definition should be given in the programmer's terms, by specifying the constraints that Java imposes on the outcomes of the read and write operations used by the programmer.

Previous work by Gontmakher and Schuster [5, 6] provides such a definition (henceforth denoted Java_1) of Java memory consistency (Section 4.1). Java_1 captures the possible outcomes of any terminating Java computation. However, as will be seen, for terminating computations it is possible to circumvent dealing explicitly with the invisibility phenomenon. We show (Section 4.2) that Java_1 is not correct for non-terminating computations such as those of a Java operating system. Section 4.3 extends and adjusts Java_1 to our new definition, Java_∞, which does deal with invisibility and which is correct for both terminating and non-terminating Java computations. We also provide a precise and short operational definition of the memory behavior of JVM (Section 3) that captures all the Java ordering constraints.

*Department of Computer Science, The University of Calgary Research Report # 98/622/13.
†Supported in part by the Natural Sciences and Engineering Research Council of Canada grant OGP0041900.
‡Supported in part by a Natural Sciences and Engineering Research Council of Canada post-graduate scholarship(B).

Existing definitions of weak memory consistency models ([14, 7, 1, 3, 11]) apply to terminating computations. However, process coordination is required in non-terminating systems, such as distributed operating systems, and interesting subtilities arise when extending from terminating to potentially non-terminating computations. Section 4.2 examines and formalizes what is required for a non-terminating computation to satisfy a given memory consistency condition. Later, this formalism is used to show that although Java is coherent when restricted to terminating computations (as proved by Gontmakher and Schuster), non-terminating Java is not (Section 5.1). Java consistency is also compared with SPARC's total store ordering, partial store ordering, and weak ordering in Section 5.2.

Section 6 shows that Java cannot support solutions to fundamental process coordination problems, such as the critical section and producer/consumer problems, without the use of expensive synchronization constructs such as `volatile` variables or `locks`. However, Section 7 shows that a form of "in-between" synchronization would allow Java to support some process coordination much more cheaply than is possible with what Java does provide.

Before proceeding with the technical results, we need the definitions of Section 2.

2 Preliminaries

2.1 Memory consistency framework

A multiprocessor machine consists of a collection of processors together with various memory components and communication channels between these components. The behavior of such a machine can be described by specifying the sequence of events that the machine executes when implementing a given program instruction. Alternatively, it can be described by precisely specifying the constraints on the perceived outcomes and orderings of the instructions that can result from an execution. Given a particular machine architecture, our goal is to formulate these constraints on computations. This subsection overviews our general framework for specifying a memory consistency model, and for modeling the corresponding machine architecture. Various memory consistency models that are used in this paper are defined in the next subsection. A comprehensive treatment appears elsewhere [11, 10].

We model a multiprocess system as a collection of processes operating via *actions* on a collection of shared data *objects*. In general, these objects may be of any type, but in this paper it suffices to consider only read(p,x,v) (process p reads value v from register x) and write(p,x,v) (process p writes value v to register x) actions.

A *process* is a sequence of invocations of actions, and the *process computation* is the sequence of actions created by augmenting each invocation in the process with its matching outcome. A *(multiprocess) system*, (P,J), is a collection P of processes and a collection J of objects, such that the actions of each process in P are applied to objects in J. A *system computation* is a collection of process computations, one for each p in P.

Let (P,J) be a multiprocess system, and O be all the (read and write) actions in a computation of this system. $O|p$ denotes all the actions that are in the process computation of p in P. $O|x$ are all the actions that are applied to object x in J. Let O_w denote the write actions and O_r denote the read actions.

A sequence of read and write actions to the same object is *valid* if each read returns the value of the most recent preceding write. A *linearization* of a collection of read and write actions O, is a linear order[1] $(O, <_L)$ such that for each x, the subsequence $(O|x, <_L)$ of $(O, <_L)$ is valid for x.

A *(memory) consistency model* is a set of constraints on system computations. These constraints are given in terms of partial order requirements on the actions O of a computation. Several partial orders

[1] A linear order is an irreflexive partial order (S,R) such that $\forall x,y \in S\ x \neq y$, either xRy or yRx.

are used in the definitions of memory consistency models. One common partial order is $(O, \xrightarrow{prog})$, called *program order*, which is defined by $o_1 \xrightarrow{prog} o_2$, if and only if o_2 follows o_1 in the computation of some process p. A computation satisfies some consistency model D if the computation meets all the constraints of D. A system provides memory consistency D if every computation that can arise from the system satisfies the consistency model D.

A multiprocessor machine *implements an action* by proceeding through a sequence of *events* that depend on the particular machine and that occur at the various components of the machine. The events in this sequence and the action that is implemented by them are said to *correspond*. A processor of a machine *implements a process* by initiating, in program order, the implementation of the actions corresponding to the action-invocations of the process. A multiprocessor machine *implements a system* (P,J) by having each processor implement a process in P. A *machine execution* is described by the sequence of resulting machine events.[2]

2.2 Memory consistency models

Following are the definitions for sequential consistency (SC) [14], coherence [7], Pipelined-RAM (P-RAM) [16, 1], Goodman's processor consistency (PC-G) [7], weak ordering (WO) [3], coherent weak ordering ($\text{WO}_{coherent}$), SPARC total store ordering (TSO) and partial store ordering (PSO) [18, 11].

Define the partial order $(O, \xrightarrow{weak-prog})$, called *weak program order*, by: Action $o_1 \xrightarrow{weak-prog} o_2$ if $o_1 \xrightarrow{prog} o_2$ and either 1) at least one of $\{o_1, o_2\}$ is a synchronization action, or 2) $\exists o'$ such that o' is a synchronization action and $o_1 \xrightarrow{prog} o' \xrightarrow{prog} o_2$, or 3) o_1 and o_2 are to the same object.

Let O be all the actions of a computation C of the multiprocess system (P,J). Then C is:

SC if there is a linearization $(O, <_L)$ satisfying $(O, \xrightarrow{prog}) \subseteq (O, <_L)$.

coherent if for each object $x \in J$ there is a linearization $(O|x, <_{L_x})$ satisfying $(O|x, \xrightarrow{prog}) \subseteq (O|x, <_{L_x})$.

P-RAM if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$.

PC-G if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ and $\forall x \in J$ $(O_w \cap O|x, <_{L_p}) = (O_w \cap O|x, <_{L_q})$.

WO if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying 1) $(O|p \cup O_w, \xrightarrow{weak-prog}) \subseteq (O|p \cup O_w, <_{L_p})$, and 2) $\forall q \in P$ $(O_w \cap O_{synch}, <_{L_p}) = (O_w \cap O_{synch}, <_{L_q})$.

$\text{WO}_{coherent}$ if for each process $p \in P$ there is some linearization $(O|p \cup O_w, <_{L_p})$ satisfying the two conditions of WO and $\forall q \in P$ and $\forall x \in J$ $(O_w|x, <_{L_p}) = (O_w|x, <_{L_q})$.

In the following, $(A \uplus B)$ denotes the disjoint union of sets A and B, and if $x \in A \cap B$ then the copy of x in A is denoted x_A and the copy of x in B is denoted x_B. Let O_a denote the set of swap atomic actions and O_{sb} denote the set of store barrier actions provided by the SPARC architecture [18]. Let O_r denote the set of actions with read semantics. Then, $O_w \cap O_r = O_a$.

TSO if there exists a total order $(O_w, \xrightarrow{writes})$ such that $(O_w, \xrightarrow{prog}) \subseteq (O_w, \xrightarrow{writes})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \xrightarrow{merge_p})$, satisfying:

1. $(O|p, \xrightarrow{prog}) = (O|p, \xrightarrow{merge_p})$, and

[2] Events in a multiprocessor can be simultaneous. For example, two different working memories may be simultaneously updated. However, because the same outcome would arise if these simultaneous events were ordered one after the other in arbitrary order, we can assume that the outcome of a machine execution arises from a *sequence* of events.

2. $(O_w, \overset{writes}{\longrightarrow}) = (O_w, \overset{merge_p}{\longrightarrow})$, and

3. if $w \in (O|p \cap O_w)$ then $w_{O|p} \overset{merge_p}{\longrightarrow} w_{O_w}$, and

4. $((O|p \uplus O_w) \backslash (O_{invisible_p} \cup O_{memwrites_p}), \overset{merge_p}{\longrightarrow})$ is a linearization, where
$O_{invisible_p} = \{w \mid w \in (O_w \backslash O|p) \cap O|x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \overset{merge_p}{\longrightarrow} w \overset{merge_p}{\longrightarrow} w'_{O_w}\}$
$O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, and

5. let $w \in (O|p \cap O_w)$ and $a \in (O|p \cap O_a)$, if $w \overset{prog}{\longrightarrow} a$, then $w_{O_w} \overset{merge_p}{\longrightarrow} a$, and if $a \overset{prog}{\longrightarrow} w$, then $a \overset{merge_p}{\longrightarrow} w_{O|p}$

PSO if there exists a total order $(O_w, \overset{writes}{\longrightarrow})$ such that $\forall x$, $(O_w \cap O|x, \overset{prog}{\longrightarrow}) \subseteq (O_w \cap O|x, \overset{writes}{\longrightarrow})$ and $\forall p \in P$ there is a total order $(O|p \uplus O_w, \overset{merge_p}{\longrightarrow})$, satisfying items 1 through 4 of TSO and (5) if $sb \in (O|p \cap O_{sb})$ and $w, u \in (O|p \cap O_w)$ and $w \overset{prog}{\longrightarrow} sb \overset{prog}{\longrightarrow} u$, then $w_{O_w} \overset{merge_p}{\longrightarrow} u_{O_w}$.

3 Java Virtual Machine

The Java Virtual Machine (JVM) [15] is an abstract machine introduced by SUN to support the Java programming language [8]. Its behavior is specified in the Java manuals [8, 15] and is quoted in Appendix A. This section provides a simple, precise alternative but equivalent description [13].

The components and events of JVM are depicted in Figure 1 for a two-thread machine. The *working*

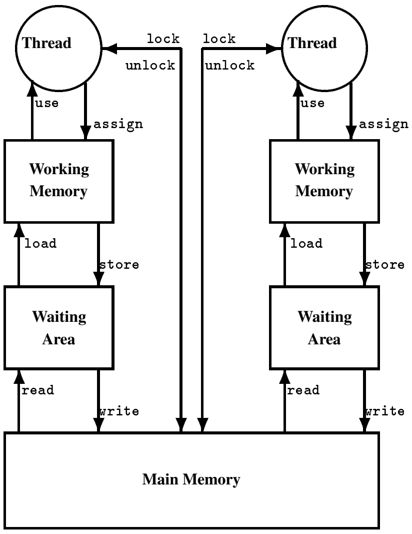

Figure 1: A two-thread JVM architecture

memory is local to a thread and is accessible by `use` and `assign` events. The *main memory* is accessible to a thread by `load`, `store`, `read`, and `write` events [3]. To commit an `assign` to main memory, the working memory performs a `store` event. Later, the main memory performs a subsequent `write` event, which updates main memory. Before a thread attempts to `use` a variable that is not in its working memory, main

[3]Main memory is also accessible by `lock` and `unlock` events. This paper does not deal in detail with these events since we are interested in the memory consistency of Java in the context of ordinary reads and writes.

memory performs a `read` event. Later, the working memory issues a `load` bringing the variable to the working memory. A component called the *waiting area* is introduced to model the delay between `stores` and `writes` and between `reads` and `loads`.

For memory consistency concerns in Java, a thread is considered to be a sequence of prog-read and prog-write actions, which are implemented in the JVM machine as follows. Let t be a thread, x an object, and v a value. Also choice $\{f\}$ designates a non-deterministic choice to perform f or not.

(A.1)

prog-read(t,x,v):
- **if** x is not in t's working memory
- **then** get(t,x,v)
- **else** choice { get(t,x,v) }
- `use`(t,x,v)

(A.2)

prog-write(t,x,v):
- `assign`(t,x,v)
- choice { put(t,x,v) }

where get(t,x,v) and put(t,x,v) are defined by:

get(t,x,v):
- `read`(t,x,v)
- `load`(t,x,v)

put(t,x,v):
- `store`(t,x,v)
- `write`(t,x,v)

A Java program S is a collection of threads. Any Java machine execution E of S is a sequence of events of the types {`assign`, `use`, `store`, `load`, `write`, `read` } satisfying the additional constraints that follow. Let o_1 and o_2 be actions in { prog-read, prog-write}, e_1, e_2, and e be events, and let $e_1 \xrightarrow{E} e_2$ denote e_1 precedes e_2 in E.

1. If $o_1 \xrightarrow{prog} o_2$ and e_1 (respectively, e_2) is the `use` or `assign` corresponding to o_1 (respectively, o_2), then $e_1 \xrightarrow{E} e_2$.
2. If `assign`$(t,x,v) \xrightarrow{E}$ `load`(t,x,u), then there is a `store`(t,x,v) satisfying `assign`$(t,x,v) \xrightarrow{E}$ `store`(t,x,v) $\xrightarrow{E}$ `load`(t,x,u).
3. Let $e \in$ { `store`(t,x,v), `load`(t,x,v) }. If $e \xrightarrow{E}$ `store`(t,x,u), then there exists an `assign`(t,x,u) satisfying $e \xrightarrow{E}$ `assign`$(t,x,u) \xrightarrow{E}$ `store`(t,x,u).
4. Let o_1 and $o_2 \in O|x \cap O|p$ and $o_1 \xrightarrow{prog} o_2$ and let e_1 and e_2 be any events corresponding to o_1 and o_2 respectively, then $e_1 \xrightarrow{E} e_2$.

It is easily confirmed that the memory consistency model arising from this description is equivalent to that of an even simpler machine where each get and put (Algorithms A.1 and A.2) are atomic events [13], and that these models are unchanged from that arising from the original set of rules describing JVM [13].

For consistency with previously defined memory models, we use the term *process* to refer to a Java *thread* in the rest of the paper.

4 Java Memory Consistency Model

The rules of Java that determine the interaction between working memories and main memory permit a process's write action to be invisible to another process. This is highlighted by the appearance of the choice function in algorithms A.1 and A.2 for prog-read and prog-write. We distinguish two kinds of invisibilities. First, certain `stores` are optional, which makes some `assigns` visible to the process that issued them, but invisible to others. We use the term *covert* to refer to this kind of invisibility. Second, a `load` is optional when a process already has a value for the required variable recorded in its working memory, which can cause a `use` to retrieve a stale value rather than seeing a newly written value. We use the term *fixate* for this kind of invisibility.

To define the memory consistency model of Java, the obstacles that arise from covert and fixate invisibilities can be cleanly and elegantly finessed as long as computations are finite [5, 6] as shown in Section 4.1. Those ideas, however, do not suffice for non-terminating Java computations. After resolving exactly what is meant by a consistency condition for a non-terminating system in Section 4.2, we provide a new definition of consistency that is correct for both terminating and non-terminating Java computations (Section 4.3).

4.1 Consistency of terminating Java computations

Gontmakher and Schuster [5, 6] gave non-operational definitions for Java memory behavior. We use Java$_1$ for their "programmer's view" characterization, after translation to our framework. Given two actions o_1 and o_2 both in $O|p$ for some $p \in P$, the *Java partial program order*, denoted ($\xrightarrow{jpo}$), is defined by: $o_1 \xrightarrow{jpo} o_2$ if $o_1 \xrightarrow{prog} o_2$ and one of the following holds:

1. $o_1,o_2 \in O|x$, or
2. $o_1 \in O_r$, $o_2 \in O_w$, and o_1 returns the value written by o' where $o' \in O_w|q\ q \neq p$, or
3. there exists o' such that $o_1 \xrightarrow{jpo} o'$ and $o' \xrightarrow{jpo} o_2$.

Definition 4.1 *[5] Let O be all the actions of a computation C of the multiprocess system* (P,J). *Then C is* Java$_1$ *if there is some linearization* $(O,<_L)$ *satisfying* $(O,\xrightarrow{jpo}) \subseteq (O,<_L)$.

Notice that this definition requires one linearization for all actions. Gontmakher and Schuster [5, 6] prove that their definition does capture exactly all terminating Java computations. There are two essential ideas in forming the linearization:

- certain covert writes can be moved to the end of the linearization so that these writes are never read by any other process and hence do not negate validity.
- fixate reads could be moved earlier in the linearization to precede the writes that are invisible to the reader so that the stale value returned is valid.

Furthermore, Java partial program order is just enough to permit these writes and reads to move as described.

A problem arises with Definition 4.1 when a system is non-terminating because the end of the computation is not defined.

4.2 Consistency models for non-terminating systems

Consider Computation 1, where process p continues to read 0 for x even though q at some point writes 1 to x. This could arise as a Java computation either 1) from a fixate invisibility of p to the updated value of x because (after its first `load`) none of p's `uses` is preceded by a matching `load`, or 2) from a covert invisibility of x because q's `assign` was not succeeded by a `store`.

Computation 1 $\left\{ \begin{array}{l} p:[r(x)0],[r(x)0],[r(x)0],[r(x)0],\ldots \\ q:w(x)1 \end{array} \right.$

Does Computation 1 satisfy Definition 4.1? Certainly for any finite prefix of p's computation, say after i reads by p, it is Java$_1$, since the linearization $[r(x)0]^i\ w(x)1$ satisfies the definition. However the linearization(s) required by a consistency model are meant to capture each system component's "view" of the computation. For Java, the given linearization means that $[r(x)0]^i\ w(x)1$ is consistent with each process's view. We expect that, as the computation continues, processes extend their respective views, but do not "change their minds" about what happened earlier. We will return to this example after we capture what it means for a non-terminating system to satisfy a given consistency condition.

Let O be all actions of some finite computation C of a system (P,J), and let D be a memory consistency model. To establish that C satisfies consistency model D, we provide a set of sequences $\mathcal{S}$, each composed of actions in O, that satisfy the constraints of D. Each sequence is meant to capture a component's "view" of the computation, or some kind of agreement between such views. Call such an $\mathcal{S}$ a set of *satisfying sequences* for (C,D).

For the definition D to hold for a non-terminating computation, C, we (informally) have two requirements. First, if C is paused, then the prefix, say $\hat{C}$, that has been realized so far, should have satisfying sequences for $(\hat{C},D)$. Second, if C is resumed and paused again later, say at $\tilde{C}$, then there are satisfying sequences for $(\tilde{C},D)$ that are "extensions" of the satisfying sequences for $(\hat{C},D)$. That is, we do not want to allow a component to reconstruct its view of what the computation did in the past. We formalize this intuition as follows.

A sequence s *extends* $\hat{s}$ if $\hat{s}$ is a prefix of s. A set of sequences $S = \{s_1,\ldots,s_n\}$ *extends* a set of sequences $\hat{S} = \{\hat{s_1},\ldots,\hat{s_n}\}$ if for each i, s_i extends $\hat{s_i}$.

Definition 4.2 *Let D be a memory consistency model for finite computations. A non-terminating computation $C = \bigcup_{p\in P}\{C_p\}$* satisfies *$D$ if $\forall p \in P$ and for every finite prefix $\hat{C}_p$ of C_p, there is a finite prefix $\hat{C}_q$ of C_q $\forall q \neq p$, such that*

1. *$\hat{C} = \bigcup_{q\in P}\{\hat{C}_q\}$ satisfies D, and*
2. *for any finite $\tilde{C}_p$ that extends $\hat{C}_p$ and is a prefix of C_p, there is a finite prefix $\tilde{C}_q$ of C_q $\forall q \neq p$, such that*
   - *$\tilde{C} = \bigcup_{q\in P}\{\tilde{C}_q\}$ extends $\hat{C}$, and*
   - *$\tilde{C}$ satisfies D, and*
   - *the satisfying sequences $\tilde{\mathcal{S}}$ for $(\tilde{C},D)$ extend the satisfying sequences $\hat{\mathcal{S}}$ for $(\hat{C},D)$.*

If we apply Definition 4.2 to Definition 4.1, Computation 1 is not Java$_1$. That is, any linearization of a finite prefix of the computation that contains q's write and satisfies Definition 4.1 cannot be extended to a linearization for a longer prefix of the computation that still satisfies the definition. (Instead, the write action by q would have to be moved to the new end of the linearization.) We need a definition of Java that is equivalent to Definition 4.1 for finite computations but that preserves semantic commitments in the course of non-terminating computations.

4.3 Java consistency

We first define Java$_2$, which is equivalent to Java$_1$ but is described from the point of view of processes.

Definition 4.3 *Let O be all the actions of a computation C of the multiprocess system (P,J). Then C is* Java$_2$ *if there is a total order $(O_w, \xrightarrow{writes})$ satisfying $\forall p \in P$:*

1. *there is a linearization $(O|p \cup O_w, <_{L_p})$ such that $(O|p \cup O_w, \xrightarrow{jpo}) \subseteq (O|p \cup O_w, <_{L_p})$, and*
2. *$(O_w, <_{L_p}) = (O_w, \xrightarrow{writes})$.*

Claim 4.4 *Java$_1$ is equivalent to Java$_2$.*

Proof: Java$_1$ $\Rightarrow$ Java$_2$. Given $(O,<_L)$ guaranteed by Definition 4.1, the total order $(O_w, \xrightarrow{writes})$ is built by projecting $(O,<_L)$ to write actions. Similarly, $(O|p \cup O_w, <_{L_p})$ is built from $(O,<_L)$ by deleting all read actions by $q \neq p$. The argument that Definition 4.3 holds is trivial.

Java$_2$ $\Rightarrow$ Java$_1$. By condition 2 of Definition 4.3, for any p the order of O_w in $<_{L_p}$ is the same as that in $\xrightarrow{writes}$. Let $(O_w, \xrightarrow{writes})$ be $w_1, w_2, ..., w_n$. Then $(O|p \cup O_w, <_{L_p}) = S_0, w_1, S_1, ..., S_n, w_n, S_n$ where the S_i's contain only p's read actions. Construct $(O,<_L)$ by adding the S_i's to $(O_w, \xrightarrow{writes})$ each after w_i but before

w_{i+1}. Since each $<_{L_p}$ is a linearization and since only read actions are added, validity follows immediately. ■

We further adjust Definition 4.3 to cope with invisibility and hence capture both terminating and non-terminating Java computations.

Definition 4.5 *Let O be all the actions of a computation C of the multiprocess system (P,J). Then C is* Java$_\infty$ *if there is some total order $(O_w, \overset{writes}{\longrightarrow})$ and $\forall p \in P$ there is a subset O_{vis_p} of O_w satisfying:*

1. *there is a linearization $(O|p \cup O_{vis_p}, <_{L_p})$ such that $(O|p \cup O_{vis_p}, \overset{jpo}{\longrightarrow}) \subseteq (O|p \cup O_{vis_p}, <_{L_p})$, and*
2. $(O_{vis_p}, <_{L_p}) = (O_{vis_p}, \overset{writes}{\longrightarrow})$.

Notice that the set O_{vis_p} in Definition 4.5 is the set of writes that "so far" are visible to process p. Notice also that Computation 1 does satisfy Definition 4.5. For any prefix of i reads by p, $(O_w, \overset{writes}{\longrightarrow}) = w(x)1$, $O_{vis_p} = \emptyset$, and $O_{vis_q} = O_w = w(x)1$. So, $(O|p \cup O_{vis_p}, <_{L_p}) = [r(x)0]^i$ and $(O|q \cup O_{vis_q}, <_{L_q}) = w(x)1$. Also, for each i these satisfying sequences are extensions of the satisfying sequences for $i-1$. The revised definition captures exactly what "happened" in the sense that $w(x)1$ took place from q's view but not from p's.

Theorem 4.6 *For finite computations, Java$_\infty$ is equivalent to Java$_2$.*

Proof: Java$_2$ $\Rightarrow$ Java$_\infty$. For all p set O_{vis_p} to O_w. In this case Definition 4.5 is identical to Definition 4.3.

Java$_\infty$ $\Rightarrow$ Java$_2$. Let C be a finite computation satisfying Definition 4.5. Denote by $L(p)$ $(O|p \cup O_{vis_p}, <_{L_p})$ guaranteed by Definition 4.5. We build the linearizations $(O|p \cup O_w, <_{\widehat{L_p}})$, denoted $\widehat{L(p)}$, from $L(p)$ and show they satisfy Definition 4.3.

Initially set $\widehat{L(p)}$ to $L(p)$. If $O_{vis_p} = O_w$, then the case is trivial. Otherwise, $O_{vis_p} \subset O_w$ and there exists a non empty set of invisible writes with respect to p, $O_{inv_p} = O_w \setminus O_{vis_p}$.

Let $(O_w, \overset{writes}{\longrightarrow})$ given by Definition 4.5 be $w_1\ w_2\ ...\ w_n$, and let i be the smallest index in $(O_w, \overset{writes}{\longrightarrow})$ such that $w_i \in O_{inv_p}$. Locate w_j in $L(p)$ where j is the smallest index in $L(p)$ such that $i < j$ (note that $w_j \in O_{vis_p}$). Insert w_i into $\widehat{L(p)}$ immediately before w_j but after any preceding action of w_j. If both w_i and w_j are to the same object, no further action is needed. Otherwise, certain reads in $\widehat{L(p)}$ will be moved as follows.

Let $\widehat{L(p)}$ be $S_1\ w_1\ S_2\ w_2\ ...\ S_j\ w_i\ w_j\ ...\ S_n\ w_n$. Note that S_i are finite sequences of reads. Let w_k be the first write in $\widehat{L(p)}$ such that $k > i$ and w_k and w_i are to the same object, say x. Move $(S_{j+1}\ S_{j+2}\ ...\ S_k)|x$ to the place sandwiched by S_j and w_i. The whole procedure is repeated for the next smallest i.

The resulting $\widehat{L(p)}$ are linearizations. First of all, note that initially $\widehat{L(p)}$ are linearizations because they are set to $L(p)$. If validity has been violated by the construction, then w_i and w_j must be to two different objects. Otherwise, it could not have been violated because w_i was inserted immediately before w_j in $\widehat{L(p)}$.

Thus, there must be a read, r, of x returning a value that is different from the value written by w_i and r succeeds w_i in $\widehat{L(p)}$. If r precedes w_k, then the insertion of w_i did not violate validity. Therefore, r must precede w_k which means that r is in the sequence $(S_{j+1}\ S_{j+2}\ ...\ S_k)|x$ which was moved to precede w_i. That is, the validity violation that the insertion of w_i introduced was restored by construction.

We still need to show that $\widehat{L(p)}$ is consistent with $\overset{jpo}{\longrightarrow}$. Since $L(p)$ is consistent with $\overset{jpo}{\longrightarrow}$, we need to argue that the above construction did not violate it. Note first of all that $(O_w, \overset{writes}{\longrightarrow})$ maintains $\overset{jpo}{\longrightarrow}$ by Definition 4.5, and that our construction maintains $\overset{writes}{\longrightarrow}$. So, we need only consider actions in $O|p$. In other words, we need only show that the movement of $(S_{j+1}\ S_{j+2}\ ...\ S_k)|x$ did not violate $\overset{jpo}{\longrightarrow}$.

Note that such a movement is moving only reads backwards; i.e., $\overset{jpo}{\longrightarrow}$ could be only violated if $\overset{prog}{\longrightarrow}$ is violated between actions on the same object. However, this simply can not be the case because $(S_{j+1}\ S_{j+2}\ ...\ S_k)|x$ precedes w_k and there is no other write to x in the interval between w_i and w_k. Furthermore, the original order of the reads in $(S_{j+1}\ S_{j+2}\ ...\ S_k)|x$ is not affected by such a movement. ■

For the remainder of this paper, Java and Java$_\infty$ are used interchangeably.

5 Comparing Java with Various Consistency Models

5.1 Java versus coherence

Gontmakher and Schuster [5, 6] argue that Java is Coherent. Their proof relies on the regular language R, which is an elegant distillation of the rules for a single Java process (Appendix A Section A.1 Rules 2, 3, 4 and 5).

$$R \begin{cases} \texttt{Order = (load-block | store-block)}^* \\ \texttt{load-block = load(use)}^* \\ \texttt{store-block = assign(use | assign)}^* \texttt{ store(use)}^* \end{cases}$$

They claim that the events corresponding to the actions of a single process to a fixed variable satisfy R. In fact, if an $\texttt{assign}(t,x,v)$ is not followed by a $\texttt{load}(t,x,u)$, then a subsequent `store` is optional. A modification of R to $\hat{R}$ captures this more general situation (λ denotes the empty string.)

$$\widehat{R} \begin{cases} \texttt{Order = (load-block | store-block)}^* \texttt{ tail-block} \\ \texttt{load-block = load(use)}^* \\ \texttt{store-block = assign(use | assign)}^* \texttt{ store(use)}^* \\ \texttt{tail-block = assign(use | assign)}^* \mid \lambda \end{cases}$$

Coherence still holds for any computation such that for each variable, and for each process, the events corresponding to the actions of that process on that variable satisfy $\hat{R}$. Their proof requires only a slight modification so that in the linearization each `tail-block` follows every `load-block` and `store-block`. Thus we can conclude that all finite Java computations are coherent.

Unfortunately, non-terminating Java computations are not necessarily coherent. When there is only one variable, notice that Java$_1$ and coherence are the same. Computation 1 is Java$_\infty$ but it is not Java$_1$. Since it uses only one variable, it cannot be coherent. One important consequence of this is that Java$_\infty$ cannot support solutions to the nonterminating coordination problem P_1C_1-queue (see Section 6) even though it has been shown [9, 12] that coherence suffices to solve this problem. The following computation [5] is coherent but not Java.

Computation 2 $\begin{cases} p : r(x)1\ w(y)1 \\ q : r(y)1\ w(x)1 \end{cases}$

Hence, Java and coherence are not comparable (except for finite computations.)

5.2 Java versus other consistency models

Gontmakher and Schuster [5, 6] show by examples that:

- Java and P-RAM are incomparable.
- Java and PC-G are incomparable.

Since their examples are finite, these same conclusions apply to Java$_\infty$.

It is easily verified that Computation 2 is WO$_{coherent}$, and (the non-treminating) Computation 1 is not WO$_{coherent}$. Therefore, Java and WO$_{coherent}$ (consequently WO) are incomparable.

To see that PSO [18, 11] is strictly stronger than Java, imagine a situation in which the working memory in JVM mimics the behavior of the store buffer in PSO. Specifically, (1) every `assign` is paired with a store and (2) every use that follows a store by the same process on the same variable is paired with a load that follows the store. The following claim makes this intuition evident.

Claim 5.1 *PSO (consequently, TSO) is strictly stronger than Java.*

Proof:

We show that PSO is strictly stronger than Java$_2$. Let $L(p)$ denote the linearizations $((O|p \uplus O_w)\backslash(O_{invisible_p} \cup O_{memwrites_p}), \xrightarrow{merge_p})$ guaranteed by PSO. We construct the linearizations $(O|p \cup O_w, <_{L_p})$, denoted $\widehat{L(p)}$, and show that they satisfy Definition 4.3.

Initially, $\widehat{L(p)}$ is set to $L(p)$ for all p. We proceed by adding the actions in $O_{invisible_p}$ to $\widehat{L(p)}$. Recall that, $O_{invisible_p} = \{w \mid w \in (O_w \backslash O|p) \cap O|x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \xrightarrow{merge_p} w \xrightarrow{merge_p} w'_{O_w}\}$. Every $w \in O_{invisible_p}$ is added to $\widehat{L(p)}$ before w' but after any preceding action to w'. If there are more than one such w by the same process on the same object, these are inserted into $\widehat{L(p)}$ such that their program order is maintained. Since w and w' are on the same object, $\widehat{L(p)}$ are obviously linearizations.

Since program order is maintained in $(O|p, \xrightarrow{merge_p})$ and in $(O_w \cap O|x, \xrightarrow{merge_p})$ (by PSO), $\xrightarrow{jpo}$ ($\xrightarrow{jpo} \subseteq \xrightarrow{prog}$) is maintained in $((O|p \uplus O_w)\backslash(O_{invisible_p} \cup O_{memwrites_p}), \xrightarrow{merge_p})$. Therefore, we need to show that the insertion of w does not violate $\xrightarrow{jpo}$. $\widehat{L(p)}$ contains no reads by $q \neq p$, and $\xrightarrow{prog}$ was maintained among actions in $O_{invisible_p}$ We still need to argue that inserting w before w' does not cross past a w'' by q on x. That is, there is no w'' such that both w and w'' are in $O|q \cap O|x$ for $q \neq p$ satisfying $w'' \xrightarrow{merge_p} w$ (or, $w'' \xrightarrow{prog} w$), but $w <_{L_p} w''$. Note that $w'_{O|p} \xrightarrow{merge_p} w \xrightarrow{merge_p} w'_{O_w}$. We have two cases, either $w'_{O|p} \xrightarrow{merge_p} w'' \xrightarrow{merge_p} w \xrightarrow{merge_p} w'_{O_w}$, or $w'' \xrightarrow{merge_p} w'_{O|p}$. In the first case, $w'' \in O_{invisible_p}$ and must have been inserted to $\widehat{L(p)}$ before w. In the second, w is placed between w'' and w' by construction. Therefore, $\xrightarrow{jpo}$ is maintained.

Finally, we note that Java Computation 1 is impossible in a PSO or TSO system.

Since TSO is strictly stronger than PSO and since Java$_2$ is strictly stronger than Java, we conclude that TSO and PSO are strictly stronger than Java. ■

The following table summarizes these observations.

M	coherence	P-RAM	PC-G	WO$_{coherent}$	WO	TSO	PSO
Java $\Rightarrow M$	NO	NO [6]	NO [6]	NO	NO	NO	NO
$M \Rightarrow$ Java	NO [6]	NO [6]	NO [6]	NO	NO	YES	YES

6 Coordination Impossibilities

This section confirms that without the synchronized or volatile constructs, fixate and covert invisibilities make Java too weak to support solutions to coordination problems such as the critical section problem (CSP) [17] or the producer/consumer problem (PCP) [2].

6.1 Critical section problem

Given that each process executes:

repeat
    <remainder>
    <entry>
    <critical section>
    <exit>
until *false*

a solution to CSP must satisfy:

- **Mutual Exclusion:** At any time there is at most one process in its <*critical section*>.
- **Progress:** If at least one process is in <*entry*>, then eventually one will be in <*critical section*>.

Theorem 6.1 *There is no Java$_1$ algorithm that solves CSP even for two processes.*

Proof: Assume for the sake of contradiction that there is a mutual exclusion algorithm A that solves CSP in Java$_1$. for processes p and q. If A runs with p in $<entry>$ and with q in $<remainder>$, then by the Progress property, p must enter its $<critical\ section>$ producing a partial computation of the form of Computation 3.

Computation 3 $\begin{cases} p: o_1^p, o_2^p, ..., o_k^p \ \ (p \text{ is in its } <critical\ section>) \\ q: \lambda \end{cases}$

where λ denotes the empty sequence and o_i^p denotes the i^{th} action of p. Similarly, if A runs with q in $<entry>$ and without p participating, a computation of the form of Computation 4 results.

Computation 4 $\begin{cases} p: \lambda \\ q: o_1^q, o_2^q, ..., o_l^q \ \ (q \text{ is in its } <critical\ section>) \end{cases}$

Now consider Computation 5.

Computation 5 $\begin{cases} p: o_1^p, o_2^p, ..., o_k^p \ \ (p \text{ is in its } <critical\ section>) \\ q: o_1^q, o_2^q, ..., o_l^q \ \ (q \text{ is in its } <critical\ section>) \end{cases}$

We show that Computation 5 satisfies the conditions of Definition 4.1. First, build $<_L$ as follows. $(O, <_L) = P_f, o_1^p, o_2^p, \ldots, o_k^p, S_f$. Initially, P_f is empty and $S_f = o_1^q, o_2^q, \ldots, o_l^q$. Certain actions are removed from S_f and appended to P_f. Examine o_i^q in S_f in order from $i = 1$ to l. If $o_i^q \in O_r|x$ for some x and there is no o_j^q such that $j < i$ and $o_j^q \in O_w|x$, then append o_i^q to P_f and remove it from S_f.

Sequence P_f contains only reads by q returning initial values, $o_1^p, o_2^p, ..., o_k^p$ is Computation 3, and for every read in S_f there is a preceding write to the same object. Hence, $P_f, o_1^p, o_2^p, ..., o_k^p, S_f$ is a linearization. Moreover $(O, \xrightarrow{jpo}) \subseteq (O, <_L)$ by construction. Therefore, Computation 5 is a possible Java$_1$ computation. However, in this computation both p and q are in their critical sections simultaneously, contradicting the requirement that A satisfies Mutual Exclusion. ∎

Since Java$_1$ is stronger than Java the following corollary is immediate.

Corollary 6.2 *There is no Java algorithm using only ordinary actions that solves CSP even for two processes.*

6.2 Producer/consumer problems

Producers and consumers are assumed to have the following forms:

producer:	*consumer:*
repeat	**repeat**
<*entry*>	<*entry*>
<*producing*>	<*consuming*>
<*exit*>	<*exit*>
until *false*	**until** *false*

We denote the producer/consumer queue problem as P_mC_n-queue where m and n are respectively the number of producer and consumer processes. A solution to P_mC_n-queue must satisfy the following:

- **Safety:** There is a one-to-one correspondence between produced and consumed items.
- **Progress:** If a producer (respectively consumer) is in <*entry*>, then it will eventually be in <*producing*> ((respectively <*consuming*>) and subsequently in <*exit*>.

- **Order:** consumers consume items in the same order as that in which the items were produced.

P_mC_n-set denotes the producer/consumer set problem. A solution for P_mC_n-set must satisfy Safety and Progress only.

The fixate or covert invisibility makes consumers (respectively, producers) unware of actions of production (respectively, consumption). The following theorem requires no proof.

Theorem 6.3 *There is no Java algorithm using only ordinary actions that solves P_mC_n-queue or P_mC_n-set even for $n = m = 1$.*

Even Java_1 is too weak to support a solution for general cases of P_mC_n-queue. The solutions for P_1C_1-queue and P_mC_n-set we presented before [9, 12] are correct for any coherent system. Thus, they are correct for Java_1.

Theorem 6.4 *There is no Java_1 algorithm that solves P_1C_n-queue or P_mC_1-queue even for $n = 2$ or $m = 2$.*

Proof: The proof is similar to that of Theorem 6.1. We give the proof for P_1C_n-queue, the P_mC_1-queue case is very similar.

Suppose there is a solution to P_1C_n-queue for some system, with producer p and two consumers c and d. Consider Computation 6 where p places item ι in the queue and quits, then c removes item ι while d is idle:

$$\textbf{Computation 6}\ \begin{cases} p : o^p_1, o^p_2, ..., o^p_k \ \ (p \textit{ has produced item } \iota.) \\ c : o^c_1, o^c_2, ..., o^c_l \ \ (c \textit{ has consumed item } \iota.) \\ d : \lambda \end{cases}$$

By Progress, this computation must be possible in the system. Similarly the following computation is also possible in the system. The Order property guarantees that c will consume item ι in Computation 6, and that d will consume the same item ι in Computation 7

$$\textbf{Computation 7}\ \begin{cases} p : o^p_1, o^p_2, ..., o^p_k \ \ (p \textit{ has produced item } \iota.) \\ c : \lambda \\ d : o^d_1, o^d_2, ..., o^d_m \ \ (d \textit{ has consumed item } \iota.) \end{cases}$$

Notice that the sequence for p is identical in both Computation 6 and Computation 7, and that p completes this sequence before c or d begin.

We will show that if Computation 6 and Computation 7 are possible then so is the computation:

$$\textbf{Computation 8}\ \begin{cases} p : o^p_1, o^p_2, ..., o^p_k \ \ (p \textit{ has produced item } \iota.) \\ c : o^c_1, o^c_2, ..., o^c_l \ \ (c \textit{ has consumed item } \iota.) \\ d : o^d_1, o^d_2, ..., o^d_m \ \ (d \textit{ has consumed item } \iota.) \end{cases}$$

We show that Computation 8 satisfies Definition 4.3. Since Java_1 is equivalent to Java_2 this will be enough to prove the claim.

Let $(O_w, \overset{writes}{\longrightarrow})$ be $(o^p_1, o^p_2, ..., o^p_k,\ o^c_1, o^c_2, ..., o^c_l,\ o^d_1, o^d_2, ..., o^d_m)|w$[4]. Let $(O|p \cup O_w, <_{L_p})$ be $o^p_1, o^p_2, ..., o^p_k$, $(o^c_1, o^c_2, ..., o^c_l,\ o^d_1, o^d_2, ..., o^d_m)|w$. This is clearly a linearization that is consistent with $\overset{prog}{\longrightarrow}$ (thus with $\overset{jpo}{\longrightarrow}$) as well as with $\overset{writes}{\longrightarrow}$. Also let $(O|c \cup O_w, <_{L_c})$ be $(o^p_1, o^p_2, ..., o^p_k)|w, o^c_1, o^c_2, ..., o^c_l, (o^d_1, o^d_2, ..., o^d_m)|w$. $(o^p_1, o^p_2, ..., o^p_k)|w$, $o^c_1, o^c_2, ..., o^c_l$ is a prefix of Computation 6; therefore, it is valid. Since $(o^d_1, o^d_2, ..., o^d_m)|w$ contains only writes, $(O|c \cup O_w, <_{L_c})$ is a linearization. Furthermore, it is consistent with both $\overset{writes}{\longrightarrow}$ and $\overset{prog}{\longrightarrow}$. Now construct $(O|d \cup O_w, <_{L_d})$ to be $(o^p_1, o^p_2, ..., o^p_k)|w$, P_f, $(o^c_1, o^c_2, ..., o^c_l)|w$, S_f. Initially, P_f is empty and $S_f = (o^d_1, o^d_2, ..., o^d_m)$. Certain actions are removed from S_f and appended to P_f. Examine o^d_i in S_f in order from $i = 1$ to j. If $o^d_i \in O_r|x$ for some x and there is no o^d_j such that $j < i$ and $o^d_j \in O_w|x$, then append o^d_i to P_f and remove it from S_f.

[4] If S is a sequence of actions, then $S|w$ denotes the subsequence of S including only write actions.

At the end, P_f contains only reads by q returning values written by p or returning initial values. Thus, $(o_1^p, o_2^p, ..., o_k^p)|w$, P_f is valid because it is part of Computation 7. Furthermore, for every read in S_f there is a preceding write to the same object. Hence, $P_f, (o_1^c, o_2^c, ..., o_k^c)|w, S_f$ is a linearization and consequently so is $(O|d \cup O_w, <_{L_d})$. Moreover, it is consistent with $\xrightarrow{writes}$ and $\xrightarrow{jpo}$.

However, in Computation 8 both c and d have consumed the same item, contradicting the Safety requirement. Thus we can conclude that a solution to P_1C_n-queue is impossible in Java_2.

■

Corollary 6.5 *There is no Java_1 algorithm that solves P_mC_n-queue for $m+n \geq 3$.*

7 Coordination Possibilities

Solving CSP or PCP is trivial with the use of `volatile` and `synchronized` constructs. However, `synchronized` methods and `volatile` variables are expensive in execution time; `volatiles` guarantee sequential consistency (SC) [5], which is not a necessary requirement to solve coordination problems. For example, PC-G suffices to solve CSP [1]. This section shows that a significant weakening of the constraints on `volatiles` , which we call "read-volatile", suffices to support solutions to some common coordination problems. We suspect that read-volatiles would allow much more parallelism than Java's `volatile` variables.

Define a *read-volatile* variable to be one that only satisfies rule A.4.1 for volatile variables. The next claim shows that read-volatiles would suffice to solve the P_1C_1-queue problem.

```
class ProducerConsumer {
item[] Q = new item[n+1]; (initialized to ⊥)

void producer() {
     int in;
     item itp;

     in = 1;
     do {
         while (Q[in] ≠ ⊥) nothing;
         ...  produce itp;
         Q[in] = itp;
         in = in + 1 mod n+1;
     } while true;
}

void consumer() {
     int out;
     item itc;

     out = 1;
     do {
         while (Q[out] = ⊥) nothing;
         itc = Q[out];
         Q[out] = ⊥;
         out = out + 1 mod n+1;
         ...  consume itc;
     } while true;
}
}
```

Figure 2: ALG_M, a multi-writer P_1C_1-queue algorithm

Claim 7.1 *Read-volatiles are sufficient to solve P_1C_1-queue in Java.*

Proof: ALG_M (Figure 2) solves P_1C_1-queue in Java if every read is a read-volatiles. For simplicity, assume that Q is of size one (one item). The extension to the general case is straight forward. Also assume the producer is producing integers incrementally from 1 to n, and let $\perp = 0$. All reads in ALG_M are assumed to be read-volatiles. We will refer to the producer and consumer by p and c respectively.

- Safety:
  To prove Safety we need to prove that p can neither over-produce nor secret-produce and that c can neither over-consume nor secret-consume.

  For p to over produce, we have to have two consecutive writes by p as such: $w(\mathtt{Q})i\ w(\mathtt{Q})i+1$. This means that we have the following ordering from the point of view of p: $w(\mathtt{Q})i\ r(\mathtt{Q})v\ w(\mathtt{Q})i+1$ by $\xrightarrow{jpo}$. By the algorithm, $v = 0$. Because $r(\mathtt{Q})v$ is a read-volatile, its corresponding `use` is preceded by a `load`. Rule B.1.2 guarantees that there is a matching `store` for the `assign` corresponding to $w(\mathtt{Q})i$ (keep in mind that all actions are to the same location). In other words, $r(\mathtt{Q})v$ must have seen a 0 which means that $w(\mathtt{Q})i$ was overwritten. Only c writes 0's. By a similar argument, we can conclude that i was consumed before $i+1$ was produced.

  To see that p can not secret-produce, note that any $w(\mathtt{Q})i$ is always followed by $r(\mathtt{Q})v$. Rule B.1.2 guarantees that there is always a matching `store` for $w(\mathtt{Q})i$.

  The argument for c is similar.

- Progress:
  Assume that ALG_M gets to a deadlock state. Therefore, p is stuck in its while loop and sees $\mathtt{Q} \neq \perp$. Similarly, c sees $\mathtt{Q} = \perp$. By rule B.4.1, both c and p are reading from main memory. By the register property, Q is either $\perp$ or different but can not be both. This means that either p or c can proceed.

- Order:
  Assume c consumes two items out of order. This means that p produces $w(\mathtt{Q})i\ w(\mathtt{Q})i+1$ in this order, but c consumes $i+1$ before i. This is simply impossible because by Definition 4.5 there is a total order on writes that both p and c must agree on. Or, simply Java does not allow the two writes ($w(\mathtt{Q})i$ and $w(\mathtt{Q})i+1$) to be written out to main memory out of program order.

Therefore, read-volatiles are sufficient to solve P_1C_1-queue in Java. ■

We have shown before that we can exploit a solution for P_1C_1-queue to build a solution for P_mC_n-set [9, 12]. Thus the following corollary.

Corollary 7.2 *Read-volatiles are sufficient to solve P_mC_n-set in Java.*

A Java Ordering Constraints

In this appendix, we quote the ordering rules from the Java manuals [8, 15]. Let T be a thread, V be a variable, and L be a lock variable.

A.1 Rules for one thread

1. "A *use* or *assign* by T of V is permitted only when dictated by execution by T of the Java program according to the standard Java execution model. For example, an occurrence of V as an operand of the + operator requires that a single *use* operation occur on V; an occurrence of V as the left-hand operand of the assignment operator = requires that a single *assign* operation occur. All *use* and *assign* actions by a given thread must occur in the order specified by the program being executed by the thread. If the following rules forbid T to perform a required use as its next action, it may be necessary for T to perform a load first in order to make progress."
2. "A *store* operation by T on V must intervene between an *assign* by T of V and a subsequent *load* by T of V. (Less formally: a thread is not permitted to lose the most recent assign.)"
3. "An *assign* operation by T on V must intervene between a *load* or *store* by T of V and a subsequent *store* by T of V. (Less formally : a thread is not permitted to write data from its working memory back to main memory for no reason.)"
4. "After a thread is created, it must perform an *assign* or *load* operation on a variable before performing a *use* or *store* operation on that variable. (Less formally: a new thread starts with an empty working memory.)"
5. "After a variable is created, every thread must perform an *assign* or *load* operation on that variable before performing a *use* or *store* operation on that variable. (Less formally: a new variable is created only in main memory and is not initially in any thread's working memory.)"

A.2 Rules for thread-main memory interaction

1. "For every *load* operation performed by any thread T on its working copy of a variable V, there must be a corresponding preceding *read* operation by the main memory on the master copy of V, and the *load* operation must put into the working copy the data transmitted by the corresponding *read* operation."
2. "For every *store* operation performed by any thread T on its working copy of a variable V, there must be a corresponding following *write* operation by the main memory on the master copy of V, and the *write* operation must put into the master copy the data transmitted by the corresponding *store* operation."
3. "Let action A be a *load* or *store* by thread T on variable V, and let action P be the corresponding *read* or *write* by the main memory on variable V. Similarly, let action B be some other *load* or *store* by thread T on that same variable V, and let action Q be the corresponding *read* or *write* by the main memory on variable V. If A precedes B, then P must precede Q. (Less formally: operations on the master copy of any given variable on behalf of a thread are performed by the main memory in exactly the order that the thread requested.)"

A.3 Rules for locks

1. "With respect to a lock, the *lock* and *unlock* operations performed by all the threads are performed in some total sequential order. This total order must be consistent with the total order on the operations of each thread."

2. “A *lock* operation by T on L may occur only if, for every thread S other than T, the number of preceding *unlock* operations by S on L equals the number of preceding *lock* operations by S on L. (Less formally: only one thread at a time is permitted to lay claim to a lock; moreover, a thread may acquire the same lock multiple times and does not relinquish ownership of it until a matching number of unlock operations have been performed.)”
3. “An *unlock* operation by thread T on lock L may occur only if the number of preceding *unlock* operations by T on L is strictly less than the number of preceding *lock* operations by T on L. (Less formally: a thread is not permitted to unlock a lock it does not own.)”
4. “Between an *assign* operation by T on V and a subsequent *unlock* operation by T on L, a *store* operation by T on V must intervene; moreover, the *write* operation corresponding to that *store* must precede the *unlock* operation, as seen by main memory. (Less formally: if a thread is to perform an unlock operation on any lock, it must first copy all assigned values in its working memory back out to main memory.)”
5. “Between a *lock* operation by T on L and a subsequent use or *store* operation by T on a variable V, an *assign* or *load* operation on V must intervene; moreover, if it is a *load* operation, then the *read* operation corresponding to that *load* must follow the *lock* operation, as seen by main memory. (Less formally: a lock operation behaves as if it flushes all variables from the thread’s working memory, after which it must either assign them itself or load copies anew from main memory.)”

A.4 Rules for Volatiles

1. “A *use* operation by T on V is permitted only if the previous operation by T on V was *load*, and a *load* operation by T on V is permitted only if the next operation by T on V is *use*. The *use* operation is said to be ”associated” with the *read* operation that corresponds to the *load*.”
2. “A *store* operation by T on V is permitted only if the previous operation by T on V was *assign*, and an *assign* operation by T on V is permitted only if the next operation by T on V is *store*. The *assign* operation is said to be ”associated” with the *write* operation that corresponds to the *store*.”
3. “Let action A be a *use* or *assign* by thread T on variable V, let action F be the *load* or *store* associated with A, and let action P be the *read* or *write* of V that corresponds to F. Similarly, let action B be a *use* or *assign* by thread T on variable W, let action G be the *load* or *store* associated with B, and let action Q be the *read* or *write* of V that corresponds to G. If A precedes B, then P must precede Q. (Less formally: operations on the master copies of volatile variables on behalf of a thread are performed by the main memory in exactly the order that the thread requested.) ”

A.5 Other rules

These rules can be inferred from several places in the Java manuals. We quote these from [5].

1. “The operations performed by any one thread are totally ordered. A *use* of V or a *store* to V in one of the program orders always uses the most recent value that was given to V by an *assign* or a *load* operation in that order.”
2. “The operations performed by the main memory for any one variable are totally ordered. A *read* in the order of one of the variables always yields the value that was written by the last *write* in that order. If there is no preceding *write* in the order, the value yielded by *read* is sone initial value.”
3. “ It is not permitted for an instruction to follow itself.”

References

[1] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pages 251–260, June 1993. Also available as College of Computing, Georgia Institute of Technology technical report GIT-CC-92/34.

[2] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, the Netherlands, 1965. Reprinted in [4].

[3] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *Proc. of the 13th Annual Int'l Symp. on Computer Architecture*, pages 434–442, June 1986.

[4] F. Genuys, editor. *Programming Languages*. Academic Press, 1968.

[5] A. Gontmakher and A. Schuster. Java consistency: Non-operational characterizations of Java memory behavior. Technical Report CS0922, Computer Science Department, Technion, November 1997.

[6] A. Gontmakher and A. Schuster. Characterizations of Java memory behavior. In *Proc. of the 12th Int'l Parallel Processing Symp.*, April 1998.

[7] J. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface Working Group, March 1989.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specifications*. Addison-Wesley, 1996.

[9] L. Higham and J. Kawash. Critical sections and producer/consumer queues in weak memory systems. In *Proc. of the 1997 Int'l Symp. on Parallel Architectures, Algorithms, and Networks*, December 1997.

[10] L. Higham, J. Kawash, and N. Verwaal. Defining and comparing memory consistency models. In *Proc. of the 10th Int'l Conf. on Parallel and Distributed Computing Systems*, pages 349–356, October 1997.

[11] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part I: Definitions and comparisons. Technical Report 98/612/03, Department of Computer Science, The University of Calgary, January 1998. Submitted to IEEE Trans. on Computers.

[12] L. Higham, J. Kawash, and N. Verwaal. Weak memory consistency models part II: Process coordination problems. Technical Report 98/613/04, Department of Computer Science, The University of Calgary, January 1998. Submitted to IEEE Trans. on Computers.

[13] J. Kawash. Process coordination in modern distributed systems. Ph.D. Dissertation Draft, The University of Calgary.

[14] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):690–691, September 1979.

[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[16] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report 180-88, Department of Computer Science, Princeton University, September 1988.

[17] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.

[18] D. L. Weaver and T. Germond, editors. *The SPARC Architecture Manual version 9*. Prentice-Hall, 1994.