# Java: Memory Consistency and Process Coordination* (Extended Abstract)

Lisa Higham and Jalal Kawash
*Department of Computer Science, The University of Calgary, Canada*
http://www.cpsc.ucalgary.ca/~{higham|kawash}
{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 introduce a form of synchronization that is weaker than `volatiles` and would suffice 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) [16] provides a global shared memory and a local memory for each Java thread. Because intricate rules [16] 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 [16, 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.

*This work was supported in part by research grant OGP0041900 and a post-graduate scholarship both from the Natural Sciences and Engineering Research Council of Canada.

Previous work by Gontmakher and Schuster [5, 6] provides such a definition (henceforth denoted $\text{Java}_1$) of Java memory consistency (Section 4.1). $\text{Java}_1$ 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 $\text{Java}_1$ is not correct for non-terminating computations such as those of a Java operating system. Section 4.3 extends and adjusts $\text{Java}_1$ to our new definition, $\text{Java}_\infty$, 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.

Existing definitions of weak memory consistency models ([15, 7, 1, 3, 12]) 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 suffice for some process coordination in Java.

Because of space limits, all the proofs have been omitted from this extended abstract. They can be found in the full version of the paper [10] available at the web cite http://www.cpsc.ucalgary.ca/~kawash. 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 [12, 11].

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 set of actions with write semantics and $O_r$ denote the set of actions with read semantics.

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 are used in the definitions of memory consistency models. One common partial order is $(O,\stackrel{prog}{\longrightarrow})$, called *program order*, which is defined by $o_1 \stackrel{prog}{\longrightarrow} o_2$, 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) [15], coherence [7], Pipelined-RAM (P-RAM) [17, 1], Goodman's processor consistency (PC-G) [7], weak

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

[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.

ordering (WO) [3], coherent weak ordering ($\text{WO}_{coherent}$), SPARC total store ordering (TSO) and partial store ordering (PSO) [19, 12].

Define the partial order $(O, \overset{weak-prog}{\longrightarrow})$, called *weak program order*, by: Action $o_1 \overset{weak-prog}{\longrightarrow} o_2$ if $o_1 \overset{prog}{\longrightarrow} o_2$ 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 $o_1 \overset{prog}{\longrightarrow} o' \overset{prog}{\longrightarrow} o_2$, 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 $(O, \overset{prog}{\longrightarrow}) \subseteq (O, <_L)$.

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

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

**PC-G** if for each process $p \in P$ there is a linearization $(O|p \cup O_w, <_{L_p})$ satisfying

1. $(O|p \cup O_w, \overset{prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, 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. $(O|p \cup O_w, \overset{weak-prog}{\longrightarrow}) \subseteq (O|p \cup O_w, <_{L_p})$, 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 [19]. Then, $O_w \cap O_r = O_a$.

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

1. $(O|p, \overset{prog}{\longrightarrow}) = (O|p, \overset{merge_p}{\longrightarrow})$, and
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. $((O|p \uplus O_w) \setminus (O_{invisible_p} \cup O_{memwrites_p}), \overset{merge_p}{\longrightarrow})$ is a linearization, where
   $O_{invisible_p} = \{w \mid w \in (O_w \setminus O|p) \cap O|x \wedge \exists w' \in O|x \cap O|p \cap O_w \wedge w'_{O|p} \overset{merge_p}{\longrightarrow} w \overset{merge_p}{\longrightarrow} w'_{O_w}\}$
   $O_{memwrites_p} = \{w_{O_w} \mid w \in O|p \cap O_w\}$, 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) [16] is an abstract machine introduced by SUN to support the Java programming language [8]. Its behavior is specified in the Java manuals [8, 16]. This section provides a simple, precise alternative but equivalent description; the proof is elsewhere [14].

The components and events of JVM are depicted in Figure 1(a) for a two-thread machine. 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 shown in Figure 1(b) where $t$ is a thread, $x$ an object, and $v$ a value (choice $\{f\}$ designates a non-deterministic choice to perform $f$ or not.)

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`$\}$[3] 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 $e_1 \overset{E}{\longrightarrow} e_2$ denote $e_1$ precedes $e_2$ in $E$.

1. If $o_1 \overset{prog}{\longrightarrow} o_2$ and $e_1$ (respectively, $e_2$) is the `use` or `assign` corresponding to $o_1$ (respectively, $o_2$), then $e_1 \overset{E}{\longrightarrow} e_2$.
2. If $\texttt{assign}(t,x,v) \overset{E}{\longrightarrow} \texttt{load}(t,x,u)$, then there is a $\texttt{store}(t,x,v)$ satisfying $\texttt{assign}(t,x,v) \overset{E}{\longrightarrow} \texttt{store}(t,x,v) \overset{E}{\longrightarrow} \texttt{load}(t,x,u)$.
3. Let $e \in \{\ \texttt{store}(t,x,v), \texttt{load}(t,x,v)\ \}$. If $e \overset{E}{\longrightarrow} \texttt{store}(t,x,u)$, then there exists an $\texttt{assign}(t,x,u)$ satisfying $e \overset{E}{\longrightarrow} \texttt{assign}(t,x,u) \overset{E}{\longrightarrow} \texttt{store}(t,x,u)$.

[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.

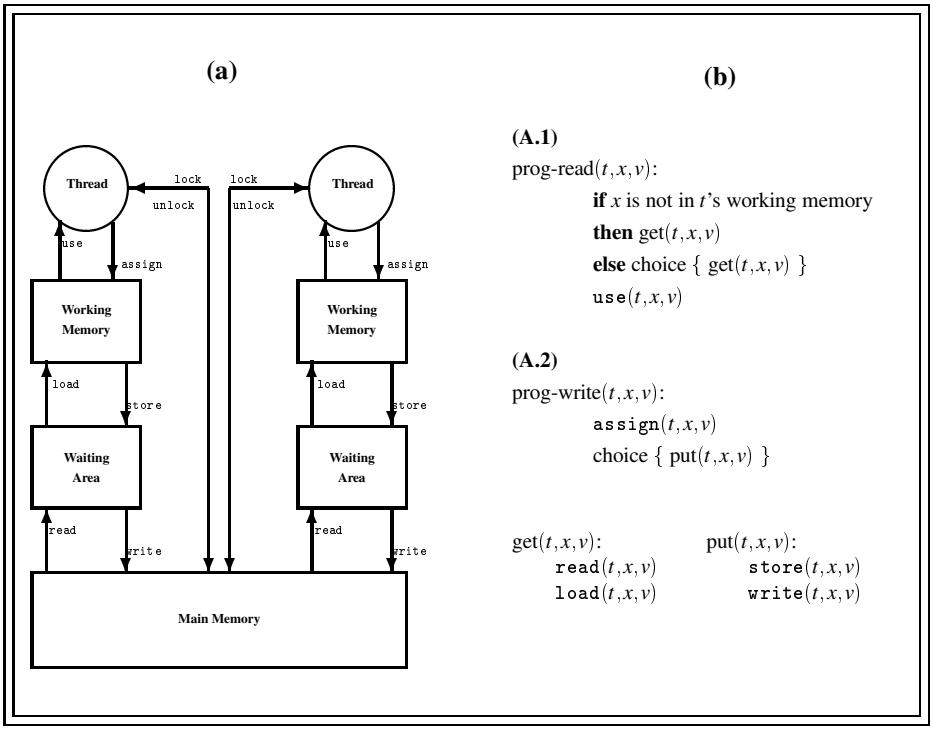


Figure 1: (a) A two-thread JVM architecture (b) Implementing program reads and writes by JVM events

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 consistency model arising from this description is equivalent to that of an even simpler machine where each get and put of Figure 1(b) are atomic events [14], and that these models are unchanged from that arising from the original set of rules describing JVM [14].

For the rest of the paper, we use the term *process* to refer to a Java *thread*.

# 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 $\text{Java}_1$ 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* $\text{Java}_1$ *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** $\begin{cases} p : [r(x)0], [r(x)0], [r(x)0], [r(x)0], \ldots \\ q : w(x)1 \end{cases}$

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 $\tilde{S}$ for $(\tilde{C},D)$ extend the satisfying sequences $\hat{S}$ 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 $(O_w, \stackrel{writes}{\longrightarrow})$ satisfying $\forall p \in P$:*

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

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

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 $(O_w, \stackrel{writes}{\longrightarrow})$ 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 $(O|p \cup O_{vis_p}, \stackrel{jpo}{\longrightarrow}) \subseteq (O|p \cup O_{vis_p}, <_{L_p})$, and*
2. *$(O_{vis_p}, <_{L_p}) = (O_{vis_p}, \stackrel{writes}{\longrightarrow})$.*

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 $p$, $(O_w, \stackrel{writes}{\longrightarrow}) = w(x)1$, $O_{vis_p} = \emptyset$, 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$_2$.*

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.

$$R \left\{ \begin{array}{l} \texttt{Order = (load-block | store-block)}^* \\ \texttt{load-block = load(use)}^* \\ \texttt{store-block = assign(use | assign)}^* \texttt{ store(use)}^* \end{array} \right.$$

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 ($\lambda$ denotes the empty string.)

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

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 $\text{Java}_1$ and coherence are the same. Computation 1 is $\text{Java}_\infty$ but it is not $\text{Java}_1$. Since it uses only one variable, it cannot be coherent. One important consequence of this is that $\text{Java}_\infty$ cannot support solutions to the nonterminating coordination problem $P_1C_1$-queue (see Section 6) even though it has been shown [9, 13] that coherence suffices to solve this problem. The following computation [5] is coherent but not Java.

**Computation 2** $\left\{ \begin{array}{l} p : r(x)1 \; w(y)1 \\ q : r(y)1 \; w(x)1 \end{array} \right.$

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-terminating) Computation 1 is not WO$_{coherent}$. Therefore, Java and WO$_{coherent}$ (consequently WO) are incomparable.

To see that PSO [19, 12] is 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 formalizes this intuition.

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

Since TSO and PSO are stronger than coherence [11, 12], Computation 1 is not possible in either PSO or TSO.

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) [18] or the producer/consumer problem (PCP) [2]. Note that solutions to these problems are assumed to be long-lived; that is, there is no bound on the number of times the protocols are invoked.

## 6.1 Critical section problem

Given that each process executes:

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

a solution to CSP must satisfy[4]:

- **Mutual Exclusion:** At any time there is at most one process in its <*critical section*>.

[4]Note that we are not listing fairness as a property. In fact, none of our impossibility results depend on fairness.

- **Progress:** If at least one process is in <*entry*>, then eventually one will be in <*critical section*>.

  If $p \in P$ is in <*entry*>, then $p$ will eventually be in <*critical section*>.

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

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:** items are consumed in an order consistent with that in which they were produced[5].

$P_mC_n$-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).

**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 previously presented [9, 13] 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 $m = 2$.*

**Corollary 6.5** *There is no Java$_1$ algorithm that solves $P_mC_n$-queue for $m + n \geq 3$.*

[5]Item $A$ precedes item $B$ if the production of item $A$ completes before the production of item $B$ begins.

# 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. Define a *read-volatile* variable to be one that only satisfies the following rule of those for `volatile` variables [16, 8]:

> "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*."

We suspect that read-volatiles would allow more parallelism than Java's `volatile` variables for many applications.

Figure 2 presents a multi-writer $P_1C_1$-queue algorithm, $ALG_M$, that is correct for coherence[6] [9, 13]. If every read of $ALG_M$ is a read-volatile, the algorithm solves $P_1C_1$-queue in Java.

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

void producer() {                    void consumer() {
     int in;                              int out;
     item it_p;                           item it_c;

     in = 1;                              out = 1;
     do {                                 do {
          while (Q[in] ≠ ⊥) nothing;           while (Q[out] = ⊥) nothing;
          ... produce it_p;                    it_c = Q[out];
          Q[in] = it_p;                        Q[out] = ⊥;
          in = in + 1 mod n+1;                 out = out + 1 mod n+1;
     } while true;                             ... consume it_c;
}                                         } 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.*

[6]The algorithm even works for P-RAM and WO with only ordinary reads and writes.

Since we have shown previously how to exploit a solution for $P_1C_1$-queue to build a solution for $P_mC_n$-set [9, 13], we have the following corollary.

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


## Acknowledgements

We thank Alan Covington for suggesting the problem of Java consistency.


# 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*, pages 56–63, December 1997.

[10] L. Higham and J. Kawash. Java: Memory consistency and process coordination. Technical Report 98/622/13, Department of Computer Science, The University of Calgary, April 1998.

[11] 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.

[12] 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 for publication.

[13] 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 for publication.

[14] J. Kawash. Process coordination issues in systems with weak memory consistency. Ph.D. dissertation draft, The University of Calgary.

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

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

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

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

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