JSR-133: Java^{TM} Memory Model and Thread Specification

This document is the community review draft of the JSR-133 specification, the Java Memory Model (JMM) and Thread Specification. This specification is intended to be part of the JSR-176 umbrella for the Tiger (1.5) release of Java, and is intended to replace Chapter 17 of the Java Language Specification and Chapter 8 of the Java Virtual Machine Specification. The current draft has been written generically to apply to both, the final version will include two different versions, essentially identical in semantics but using the appropriate terminology for each.

The discussion and development of this specification has been unusally detailed and technical, involving insights and advances in a number of academic topics. This discussion is archived (and continues) at the JMM web site `http://www.cs.umd.edu/~pugh/java/memoryModel`. The web site provides additional information that may help in understanding how this specification was arrived at.

The core semantics (Sections 4 – 7) is intended to describe semantics allowed by existing JVMs. The existing chapters of the JLS and JVMS specify semantics that are at odds with optimizations performed by many existing JVMs. The proposed core semantics should not cause issues for existing JVM implementations, although they may limit potential future optimizations and implementations.

The expert group seeks feedback on two technical issues: a choice of a weak or strong causality model (Section 7.4), and an issue regarding the semantics of notify and interrupt (Section 12.4). The JSR-133 expert group urges all readers to examine these issues and provide the expert group with feedback on them.

The weak causality model could allow more compiler optimizations, the strong model could provide more safety guarantees. However, we have not found any compelling and realistic examples of optimizations or guarantees that would guide the choice between the strong and weak causality model. This document proposes the strong causality model, since any JVM that is compliant with the strong causality model will also be compliant with the weak causality model.

JCP members are also urged to closely read the semantics on final fields (Sections 3.5 and 8). This is the one place most likely to require JVM implementors to change their implementation to be compliant with JSR-133. In particular, memory barriers or other techniques may be required to ensure that other threads see the correct values for final fields of immutable objects, even in the presence of data races.

Contents

List of Figures

1 Introduction

Java virtual machines support multiple *threads* of execution. Threads are represented in Java by the `Thread` class. The only way for a user to create a thread is to create an object of this class; each Java thread is associated with such an object. A thread will start when the `start()` method is invoked on the corresponding `Thread` object.

The behavior of threads, particularly when not correctly synchronized, can be particularly confusing and intuitive. This specification describes the semantics of multithreaded Java programs, including rules on what values may be seen by a read of shared memory that is updated by other threads. Similar to the *memory model* for different hardware architectures, these semantics have been referred to as the *Java memory mode*.

These semantics do not describe how a multithreaded program should be executed. Rather, they describe only the behaviors that are allowed by multithreaded programs. Any execution strategy that generates only allowed behaviors is an acceptable execution strategy. This is discussed more in Appendix A.

1.1 Locks

Java provides multiple mechanisms for communicating between threads. The most basic of these methods is *synchronization*, which is implemented using *monitors*. Each object in Java is associated with a monitor, which a thread can *lock* or *unlock*. Only one thread at a time may hold a lock on a monitor. Any other threads attempting to lock that monitor are blocked until they can obtain a lock on that monitor.

A thread t may lock a particular monitor multiple times; each unlock reverses the effect of one lock operation.

The Java programming language does not provide a way to perform separate lock and unlock actions; instead, they are implicitly performed by high-level constructs that always arrange to pair such actions correctly.

Note, however, that the Java virtual machine provides separate monitorenter and monitorexit instructions that implement the lock and unlock actions.

The `synchronized` statement computes a reference to an object; it then attempts to perform a lock action on that object's monitor and does not proceed further until the lock action has successfully completed. After the lock action has been performed, the body of the `synchronized` statement is executed. If execution of the body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

A `synchronized` method automatically performs a lock action when it is invoked; its body is not executed until the lock action has successfully completed. If the method is an instance method, it locks the monitor associated with the instance for which it was invoked (that is, the object that will be known as `this` during execution of the body of the method). If the method is static, it locks the monitor associated with the Class object that represents the class in which the method is defined. If execution of the method's body is ever completed, either normally or abruptly, an unlock action is automatically performed on that same monitor.

Original code	
Initially, A == B == 0	
Thread 1	Thread 2
1: r2 = A;	3: r1 = B
2: B = 1;	4: A = 2
May return r2 == 2, r1 == 1	

Valid compiler transformation	
Initially, A == B == 0	
Thread 1	Thread 2
B = 1;	A = 2
r2 = A;	r1 = B
May return r2 == 2, r1 == 1	

Figure 1: Surprising results caused by statement reordering

The Java programming language does not prevent, nor require detection of, deadlock conditions. Programs where threads hold (directly or indirectly) locks on multiple objects should use conventional techniques for deadlock avoidance, creating higher-level locking primitives that don't deadlock, if necessary.

There is a total order over all lock and unlock actions performed by an execution of a program.

1.2 Notation in examples

The Java memory model is not substantially intertwined with the Object-Oriented nature of the Java programming language. For terseness and simplicity in our examples, we often exhibit code fragments that could as easily be C or Pascal code fragments, without class or method definitions, or explicit dereferencing. Instead, most examples consists of two or more threads containing statements with access to local variables (e.g., local variables of a method, not accessible to other threads), shared global variables (which might be static fields) or instance fields of an object.

2 Incorrectly synchronized programs can exhibit surprising behaviors

The semantics of the Java programming language allow compilers and microprocessors to perform optimizations that can interact with incorrectly synchronized code in ways that can produce behaviors that seem paradoxical.

Consider, for example, Figure 1. This program contains local variables $r1$ and $r2$; it also contains shared variables A and B, which are fields of an object. It may appear that the result $r2 == 2, r1 == 1$ is impossible. Intuitively, if $r2$ is 2, then instruction 4 came before instruction 1. Further, if $r1$ is 1, then instruction 2 came before instruction 3. So, if $r2 == 2$ and $r1 == 1$, then instruction 4 came before instruction 1, which comes before instruction 2, which came before instruction 3, which comes before instruction 4. This is, on the face of it, absurd.

However, compilers are allowed to reorder the instructions in each thread. If instruction 3 is made to execute after instruction 4, and instruction 1 is made to execute after instruction 2, then the result $r2 == 2$ and $r1 == 1$ is perfectly reasonable.

Original code

Initially: `p == q`, `p.x == 0`

Thread 1	Thread 2
m = p.x;	p.x = 3
n = q.x;	
o = p.x;	

May return m == o == 0, n == 3

Valid compiler transformation

Initially: `p == q`, `p.x == 0`

Thread 1	Thread 2
m = p.x;	p.x = 3
n = q.x;	
o = m;	

May return m == o == 0, n == 3

Figure 2: Surprising results caused by forward substitution

To some programmers, this behavior may make it seem as if their code is being "broken" by Java. However, it should be noted that this code is improperly synchronized:

- there is a write in one thread,
- a read of the same variable by another thread,
- and the write and read are not ordered by synchronization.

This is called a *data race*. When code contains a data race, counter-intuitive results are often possible.

Several mechanisms can affect this reordering: the just-in-time compiler and the processor may rearrange code. In addition, the memory hierarchy of the architecture on which a virtual machine is run may make it appear as if code is being reordered. For the purposes of simplicity, we shall simply refer to anything that can reorder code as being a compiler. Source code to bytecode transformation, which is traditionally thought of as compilation, is outside the scope of this document.

Another example of surprising results can be seen in Figure 2. This program is incorrectly synchronized; it accesses shared memory without enforcing any ordering between those accesses. One common compiler optimization involves having the value read for m reused for o: they are both reads of $p.x$ with no intervening write.

Now consider the case where the assignment to $p.x$ in Thread 2 happens between the first read of $p.x$ and the read of $q.x$ in Thread 1. If the compiler decides to reuse the value of $p.x$ for the second read, then m and o will have the value 0, and n will have the value 3. This may seem counter-intuitive as well: from the perspective of the programmer, the value stored at $p.x$ has changed from 0 to 3 and then changed back.

Although this behavior is surprising, it is allowed by most JVMs. However, it is forbidden by the original Java memory model in the JLS and JVMS, one of the first indications that the original JMM needed to be replaced.

3 Informal Semantics

A program must be *correctly synchronized* to avoid the kinds of non-intuitive behaviors that can be observed when code is reordered. The use of correct synchronization does not ensure that the overall behavior of a program is correct. However, its use does allow a programmer to reason about the possible behaviors of a program in a simple way; the behavior of a correctly synchronized program is much less dependent on possible reorderings. Without correct synchronization, *very* strange, confusing and counter-intuitive behaviors are possible.

There are two key ideas to understanding whether a program is correctly synchronized:

Conflicting Accesses Two accesses (reads of or writes to) the same shared field or array element are said to be *conflicting* if at least one of the accesses is a write.

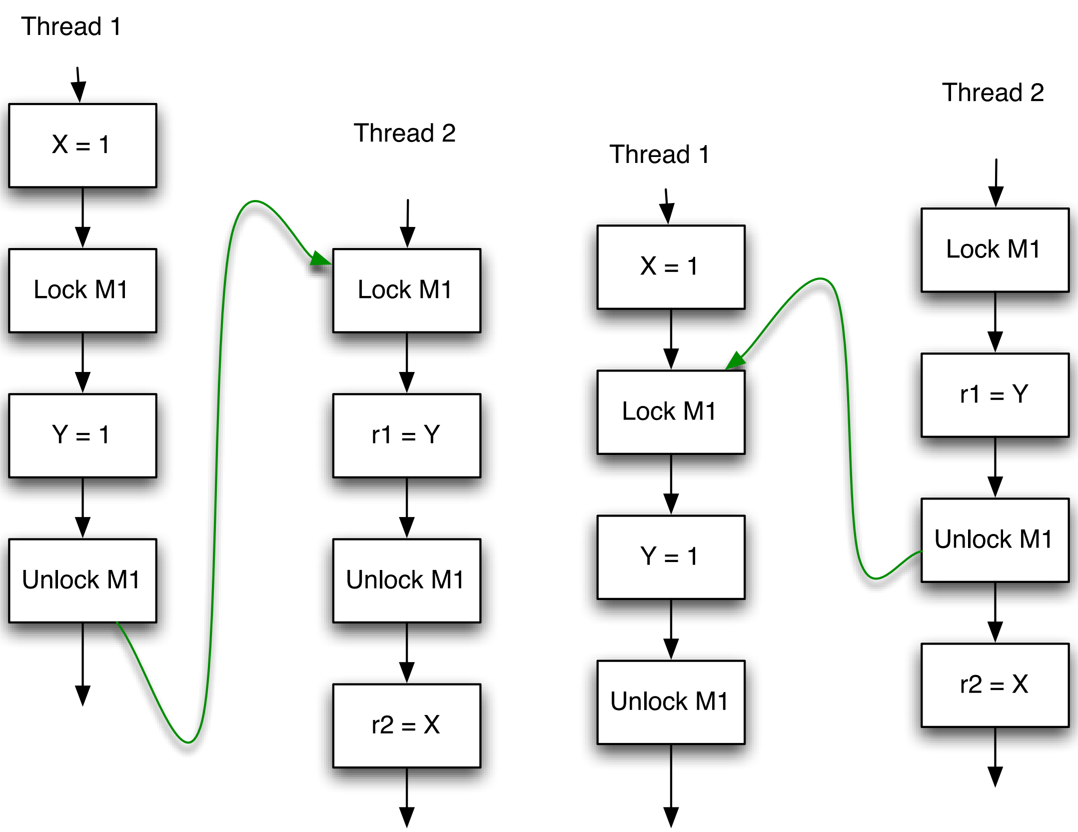
Happens-Before Relationship Two actions can be ordered by a *happens-before* relationship. If one action happens before another, then the first is visible to and ordered before the second. There are a number of ways to induce a happens-before ordering in a Java program, including:

- Each action in a thread happens before every subsequent action in that thread.
- An unlock on a monitor happens before every subsequent lock on that monitor.
- A write to a volatile field happens before every subsequent read of that volatile.
- A call to `start()` on a thread happens before any actions in the started thread.
- All actions in a thread happen before any other thread successfully returns from a `join()` on that thread.
- If an action a happens before an action b, and b happens before an action c, then a happens before c.

When a program contains two conflicting accesses that are not ordered by a happens-before relationship, it is said to contain a *data race*. A correctly synchronized program is one that has no data races (Section 3.4 contains a subtle but important clarification).

An example of incorrectly synchronized code can be seen in Figure 3, which shows two different executions of the same program, both of which contain conflicting accesses to shared variables X and Y. In Figure 3a, the two threads lock and unlock a monitor M1 so that, in this execution, there is a happens-before relationship between all pairs of conflicting accesses. However, a different execution, shown in Figure 3b, shows why this program is incorrectly synchronized; there is no happens-before ordering between the conflicting accesses to X.

If a program is not correctly synchronized, then three types of problems can appear: *visibility*, *ordering* and *atomicity*.



(a) Correctly ordered

(b) Accesses to X not correctly ordered

Figure 3: Ordering by a happens-before ordering

```
class LoopMayNeverEnd {
  boolean done = false;

  void work() {
    while (!done) {
      // do work
    }
  }

  void stopWork() {
    done = true;
  }
}
```

Figure 4: Visibility Example

3.1 Visibility

If an action in one thread is *visible* to another thread, then the result of that action can be observed by the second thread. The results of one action are only guaranteed to be visible to a second action if the first happens before the second.

Consider the code in Figure 4. Now imagine that two threads are created, and that one thread calls `work()`, and at some point, the other thread calls `stopWork()`. Because there is no happens-before relationship between the two threads, the thread in the loop may never see the update to `done` performed by the other thread. In practice, this may happen if the compiler detects that there are no updates to `done` in the first thread: the loop can be transformed to an infinite loop.

To ensure that this does not happen, there must be a happens-before relationship between the two threads. In `LoopMayNeverEnd`, this can be achieved by declaring `done` to be `volatile`. All actions on volatiles happen in a total order, and each write to a volatile field happens before any subsequent read of that volatile.

3.2 Ordering

Ordering constraints govern the order in which multiple actions are seen to have happened. The ability to perceive ordering constraints among actions is only guaranteed to actions that share a happens-before relationship with them.

The code in Figure 5 shows an example of where the lack of ordering constraints can produce surprising results. Consider what happens if `sideOne()` gets executed in one thread and `sideTwo()` gets executed in another. Would it be possible for `temp1` and `temp2` both to be true?

```
class BadlyOrdered {
  boolean a = false;
  boolean b = false;

  boolean sideOne() {
    boolean temp1 = b; // 1
    a = true; // 2
    return temp1;
  }

  boolean sideTwo() {
    boolean temp2 = a; // 3
    b = true; // 4
    return temp2;
  }
}
```

Figure 5: Ordering example

The Java memory model allows this result, illustrating a violation of the ordering that a user might have expected. This code fragment is not correctly synchronized (the conflicting accesses are not ordered by a happens-before ordering).

If ordering is not guaranteed, then the actions labeled 2 and 4 can appear to happen before the actions labeled 1 and 3; both reads can then see the value `true`. Compilers have substantial freedom to reorder code in the absence of synchronization, so a compiler could move the assignments to `a` and `b` earlier, resulting in $temp1$ and $temp2$ both being *true*.

3.3 Atomicity

If an action is (or a set of actions are) *atomic*, its result must be seen to happen "all at once", or indivisibly. Section 10 discusses some atomicity issues for Java; other than the exceptions mentioned there, all individual read and write actions take place atomically.

Atomicity can also be enforced between multiple actions. A program can be free from data races without having this form of atomicity. However, it is frequently just as important to enforce appropriate atomicity in a program as it is to enforce freedom from data races. Consider the code in Figure 6. Since all access to the shared variable `balance` is guarded by synchronization, the code is free of data races.

Now assume that one thread calls `deposit(5)`, while another calls `withdraw(5)`; there is an initial balance of 10. Ideally, at the end of these two calls, there would still be a balance of 10. However, consider what would happen if:

- The `deposit()` method sees a value of 10 for the balance, then

```
class BrokenBankAccount {
  private int balance;

  synchronized int getBalance() {
    return balance;
  }

  synchronized void setBalance(int x) throws IllegalStateException {
    balance = x;
    if (balance < 0) {
      throw new IllegalStateException("Negative Balance");
    }
  }

  void deposit(int x) {
    int b = getBalance();
    setBalance(b + x);
  }

  void withdraw(int x) {
    int b = getBalance();
    setBalance(b - x);
  }
}
```

Figure 6: Atomicity Example

- The `withdraw()` method sees a value of 10 for the balance **and** withdraws 5, leaving a balance of 5, and finally
- The `deposit()` method uses the balance it originally saw to calculate the new balance.

As a result of this lack of atomicity, the balance is 15 instead of 10. This effect is often referred to as a *lost update* because the withdrawal is lost. A programmer writing multi-threaded code must use synchronization carefully to avoid this sort of error. For this example, making the `deposit()` and `withdraw()` methods `synchronized` will ensure that the actions of those methods take place atomically.

3.4 Sequential Consistency

If a program has no data races, then executions of the program are *sequentially consistent*: very strong guarantees are made about visibility and ordering. Within a sequentially consistent execution, there is a total order over all individual actions (such as a read or a write) which is consistent with program order. Each individual action is atomic and is immediately visible to every thread. As noted before, sequential consistency and/or freedom from data races still allows errors arising from groups of operations that need to be perceived atomically, as shown in Figure 6.

Having defined sequential consistency, we can use it to provide an important clarification regarding correctly synchronized programs. A program is correctly synchronized if and only if all sequentially consistent executions are free of data races. Programmers therefore only need to reason about sequentially consistent executions to determine if their programs are correctly synchronized.

A more full and formal treatment of memory model issues for normal fields is given in Sections 4–7.

3.5 Final Fields

Fields declared `final` can be initialized once, but never changed. The detailed semantics of final fields are somewhat different from those of normal fields. In particular, compilers have a great deal of freedom to move reads of final fields across synchronization barriers and calls to arbitrary or unknown methods. Correspondingly, compilers are allowed to keep the value of a final field cached in a register and not reload it from memory in situations where a non-final field would have to be reloaded.

Final fields also allow programmers to implement thread-safe immutable objects without synchronization. A thread-safe immutable object is seen as immutable by all threads, even if a data race is used to pass references to the immutable object between threads. This can provide safety guarantees against misuse of the immutable class by incorrect or malicious code.

Final fields must be used correctly to provide a guarantee of immutability. An object is considered to be *completely initialized* when its constructor finishes. A thread that can only

```
class FinalFieldExample {

  final int x;
  int y;
  static FinalFieldExample f;

  public FinalFieldExample() {
    x = 3;
    y = 4;
  }

  static void writer() {
    f = new FinalFieldExample();
  }

  static void reader() {
    if (f != null) {
      int i = f.x;
      int j = f.y;
    }
  }
}
```

Figure 7: Example illustrating final fields semantics

see a reference to an object after that object has been completely initialized is guaranteed to see the correctly initialized values for that object's final fields.

The usage model for final fields is a simple one. Set the final fields for an object in that object's constructor. Do not write a reference to the object being constructed in a place where another thread can see it before the object is completely initialized. When the object is seen by another thread, that thread will always see the correctly constructed version of that object's final fields, and any object or array referenced by those final fields.

Figure 7 gives an example that demonstrates how final fields compare to normal fields. The class `FinalFieldExample` has a final int field x and a non-final int field y. One thread might execute the method `writer()`, and another might execute the method `reader()`. Because `writer()` writes f *after* the object's constructor finishes, the `reader()` will be guaranteed to see the properly initialized value for $f.x$: it will read the value 3. However, $f.y$ is not final; the `reader()` method is therefore not guaranteed to see the value 4 for it.

Final fields are designed to allow for necessary security guarantees.

Consider the code in Figure 8. `String` objects are intended to be immutable and string operations do not perform synchronization. While the `String` implementation does not have any data races, other code could have data races involving the use of `String`s, and the JLS

Thread 1	Thread 2
	`String myS = Global.s;`
`Global.s = "/tmp/usr".substring(4);`	`if (myS.equals("/tmp"))`
	`  System.out.println(myS);`

Figure 8: Without final fields or synchronization, it is possible for this code to print `/usr`

makes weak guarantees for programs that have data races. In particular, if the fields of the `String` class were not final, then it would be possible (although unlikely in the extreme) that thread 2 could initially see the default value of 0 for the offset of the string object, allowing it to compare as equal to `"/tmp"`. A later operation on the `String` object might see the correct offset of 4, so that the `String` object is perceived as being `"/usr"`. Many security features of the Java programming language depend upon `String`s being perceived as truly immutable.

This is only an overview of the semantics of final fields. For a more detailed discussion, which includes several cases not mentioned here, consult Section 8.

4 The Java Memory Model

A *memory model* describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program. Java's memory model works by examining each read in an execution trace and checking that the write observed by that read is valid.

A high level, informal overview of the memory model shows it to be a set of rules for when writes by one thread are visible to another thread. Informally, a read r can see the value of any write w such that w does not occur after r and w is not seen to be overwritten by another write w' (from r's perspective).

When we use the term "read" in this memory model, we are only referring to values returned from fields or array elements. There are other actions performed by a virtual machine, including reads of array lengths, executions of checked casts, and invocations of virtual methods, that must always return the correct value. Although these may be implemented with reads at the machine level, these actions cannot throw an exception or otherwise cause the VM to misbehave (e.g., crash or report the wrong array length).

The memory semantics determine what values can be read at every point in the program. The actions of each thread in isolation must behave as governed by the semantics of that thread, with the exception that the values seen by each read are determined by the memory model. We refer to this as obeying intra-thread semantics.

However, when threads interact, reads can return values written by writes from different threads. The model provides two main guarantees for the values seen by reads.

- *Consistency* requires that behavior is consistent with both intra-thread semantics and the write visibility enforced by the happens-before ordering.

- *Causality* means that an action cannot cause itself to happen. In other words, it must be possible to explain how the actions occurred in an execution without depending on one of the actions that you are trying to explain.

  Causality is necessary to ensure that correctly synchronized programs have sequentially consistent semantics. This is covered in more detail in Section 6.

5 Definitions

Shared variables/Heap memory Memory that can be shared between threads is called *shared* or *heap* memory. All instance fields, static fields and array elements are stored in heap memory. We use the term variable to refer to both fields and array elements. Variables local to a method are never shared between threads.

Inter-thread Actions An inter-thread action is an action performed by a thread that could be detected by or be directly influenced by another thread. Inter-thread actions include reads and writes of shared variables and synchronization actions, such as obtaining or releasing a lock, reading or writing a shared variable, or starting a thread.

We do not need to concern ourselves with intra-thread actions (e.g., adding two local variables and storing the result in a third local variable).

An inter-thread action is annotated with information about the execution of that action. All actions are annotated with the thread in which they occur and the program order in which they occur within that thread. Some additional annotations include:

write	The variable written to and the value written.
read	The variable read and the write seen (from this, we can determine the value seen).
lock	The monitor which is locked.
unlock	The monitor which is unlocked.

For brevity's sake, we usually refer to inter-thread actions as simply *actions*.

Program Order Among all the inter-thread actions performed by each thread t, the program order of t is a total order that reflects the order in which these actions would be performed according to the intra-thread semantics of t.

Intra-thread semantics *Intra-thread semantics* are the standard semantics for single threaded programs, and allow the complete prediction of the behavior of a thread based on the values seen by read actions within the thread. To determine if the actions of thread t in an execution are legal, we simply evaluate the implementation of thread t as would be performed in a single threaded context, as defined in the remainder of the Java Language Specification.

Each time evaluation of thread t generates an inter-thread action, it must match the inter-thread action a of t that comes next in program order. If a is a read, then further evaluation of t uses the value seen by a.

Simply put, intra-thread semantics are what result from the execution of a thread in isolation; when values are read from the heap, they are determined by the memory model.

Synchronization Actions All inter-thread actions other than reads and writes of normal and final variables are synchronization actions.

Synchronization Order In any execution, there is a *synchronization order* which is a total order over all of the synchronization actions of that execution. For each thread t, the synchronization order of the synchronization actions in t is consistent with the program order of t.

Happens-Before Edges If we have two actions x and y, we use $x \stackrel{hb}{\rightarrow} y$ to mean that x *happens-before* y. If x and y are actions of the same thread and x comes before y in program order, then $x \stackrel{hb}{\rightarrow} y$.

Synchronization actions also induce happens-before edges:

- There is a happens-before edge from an unlock action on monitor m to all subsequent lock actions on m (where subsequent is defined according to the synchronization order).
- There is a happens-before edge from a write to a volatile variable v to all subsequent reads of v (where subsequent is defined according to the synchronization order).
- There is a happens-before edge from an action that starts a thread to the first action in the thread it starts.
- There is a happens-before edge between the final action in a thread T1 and an action in another thread T2 that allows T2 to detect that T1 has terminated. T2 may accomplish this by calling `T1.isAlive()` or doing a join action on T1.

In addition, we have two other rules for generating happens-before edges.

- There is a happens-before edge from the write of the default value (zero, false or null) of each variable to the first action in every thread.
- Happens-before is transitively closed. In other words, if $x \stackrel{hb}{\rightarrow} y$ and $y \stackrel{hb}{\rightarrow} z$, then $x \stackrel{hb}{\rightarrow} z$.

Execution Trace An *execution trace* (which we sometimes simply call an *execution*) E of a program P consists of four parts:

- A set of inter-thread actions, including, for each thread t, the program order among actions in thread t.
- A synchronization order over the synchronization actions in the execution.
- The *happens-before* relationships in the program, derived from the program order and the synchronization order.
- A causal order, discussed in Section 7.

This tuple is written as $\langle S, so, \stackrel{hb}{\rightarrow}, co\rangle$. An execution trace E is a *valid execution trace* if and only if

- the actions of each thread obey intra-thread semantics
- and the values seen by the reads in E are valid according to the memory model (as defined in Section 6 – 10).

The use of fields marked `final` changes the guarantees for write visibility somewhat. Final fields are frequently guaranteed to see their correctly initialized value regardless of happens-before orderings. This is discussed in detail in Section 8.

The `wait` methods of class `Object` have lock and unlock actions associated with them; their happens-before relationships are defined by these associated actions. These methods are described further in Section 12.

6 Consistency

We first introduce a simple memory model called *consistency*.

The happens-before relationship defines a partial order over the actions in an execution trace; one action is ordered before another in the partial order if one action happens-before the other. We say that a read r of a variable v is *allowed* to observe a write w to v if, in the happens-before partial order of the execution trace:

- r is not ordered before w (i.e., it is not the case that $r \stackrel{hb}{\rightarrow} w$), and
- there is no intervening write w' to v (i.e., no write w' to v such that $w \stackrel{hb}{\rightarrow} w' \stackrel{hb}{\rightarrow} r$).

Informally, a read r is allowed to see the result of a write w if there is no happens-before ordering to prevent that read. An execution trace is *consistent* if all of the reads in the execution trace are allowed.

Because consistent execution traces do not have causal orders, they are represented by a tuple $\langle S, so, \stackrel{hb}{\rightarrow}\rangle$.

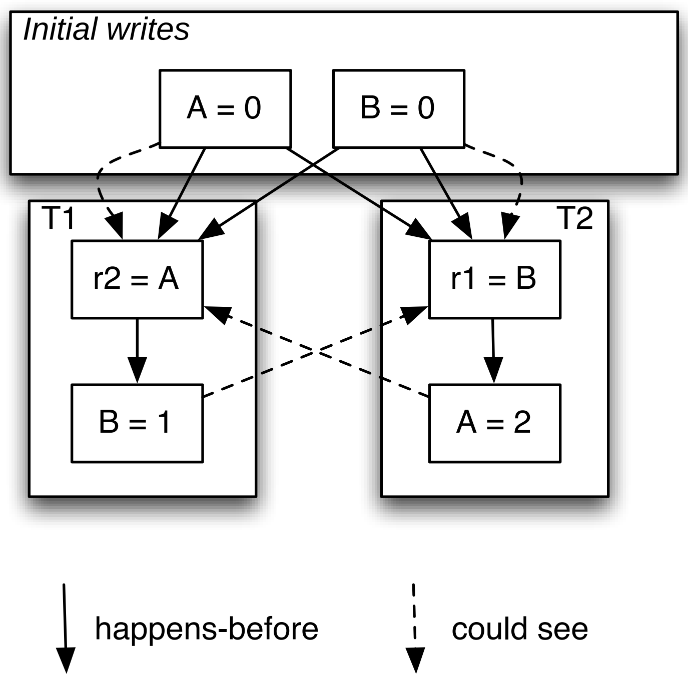


Figure 9: Execution trace of Figure 1

As an example of this simple model, consider Figure 1, and the corresponding graph in Figure 9. The solid lines represent happens-before relationships between the actions. The dotted lines between a write and a read indicate a write that the read is allowed to see. For example, the read at `r2 = A` is allowed to see the write at `A = 0` or the write `A = 2`. An execution is consistent, and valid according to the Consistency memory model, if all reads see writes they are allowed to see. So, for example, an execution that has the result $r1 == 1, r2 == 2$ would be a valid one.

7 Causality

Consistency is a necessary, but not sufficient, set of constraints. In other words, we need the requirements imposed by Consistency, but they allow for unacceptable behaviors.

In particular, one of our key requirements is that correctly synchronized programs may exhibit only sequentially consistent behavior. Consistency alone will violate this requirement.

Consider the code in Figure 10. If this code is executed in a sequentially consistent way, each action will occur in program order, and neither of the writes will occur. Since no writes occur, there can be no data races: the program is correctly synchronized. We therefore only want the program to exhibit sequentially consistent behavior.

Could we get a non-sequentially consistent behavior from this program? Consider what would happen if both $r1$ and $r2$ saw the value 1. Can we argue that this relatively nonsensical result is legal under Consistency?

The answer to this is "yes". The read in Thread 2 is allowed to see the write in Thread 1, because there is no happens-before relationship to prevent it. Similarly, the read in Thread 1 is allowed to see the read in Thread 2: there is no synchronization to prevent that, either. Consistency is therefore inadequate for our purposes.

Even for incorrectly synchronized programs, Consistency is too weak: it can allow situations in which an action causes itself to happen, when it could happen no other way. We say that an execution that contains this behavior contains a *causal loop*. At the extreme, this might allow a value to appear out of thin air. An example of this is given in Figure 11. If we decide arbitrarily that the writes in each thread will be of the value 42, the behavior `r1 == r2 == 42` can be validated as consistent: each read each sees a write in the execution, without any intervening happens-before relation.

To avoid problems such as this, we require that executions respect causality. It turns out that formally defining causality in a multithreaded context is tricky and subtle.

7.1 Causal Orders

For any execution trace, we require the existence of a *causal order*, which is an ordered list of the reads, writes and synchronization actions in that execution. The causal order can be considered an explanation of how an execution happened. For example, first we can show that x could happen; once we know x will happen, we can show that y can happen; once we know that both x and y will happen, we can show that z can happen.

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
if (r1 != 0)	if (r2 != 0)
y = 1;	x = 1;

Correctly synchronized, so r1 == r2 == 0 is the only legal behavior

Figure 10: Consistency is not sufficient

Initially, x == y == 0

Thread 1	Thread 2
r1 = x;	r2 = y;
y = r1;	x = r2;

Incorrectly Synchronized: But r1 == r2 == 42 Still Cannot Happen

Figure 11: Motivation for disallowing some cycles

The causal order does *not* have to be consistent with the program order or the happens-before order. Any total order over actions in an execution trace is potentially a valid causal order. The causal order could, for example, reflect the order in which the actions might occur after compiler transformations have taken place.

The intuition behind causal orders is that for each prefix of that causal order, the next action in the order is *justified* by the actions in the prefix. Actions that do not involve potential cycles do not need to be explicitly justified; we only require the justification of *prescient* actions. An action x in a trace $\langle S, so, \stackrel{hb}{\rightarrow}, co\rangle$ is prescient if and only if either:

- there exists an action y that occurs after x in the causal order such that $y \stackrel{hb}{\rightarrow} x$, or
- x observes a write that occurs after it in the causal order.

All prescient actions must be justified. To justify a prescient action x in trace E, let α be the actions that occur before x in the causal order. We need to show that x will be allowed to occur in all executions with a causal order that starts with α and contains no prescient actions after α. Section 7.4 describes an alternative weak causality model, in which we would only require that there exist some execution in which x would be allowed to occur.

It should be fairly clear that there are no causal orders for which Figure 11 will produce 42: there is no sequence of actions that will guarantee that 42 will be written to x or y.

In addition, the reads in Figure 10 will not be able to see the value 1. The first action in the causal order would have to be a write of 1 or a read of 1. Since neither of those are allowed in any non-prescient execution, they cannot be the first action in a causal order.

Before compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
1: r1 = a;	5: r3 = b;
2: r2 = a;	6: a = r3;
3: if (r1 == r2)	
4: b = 2;	

Is r1 == r2 == r3 == 2 possible?

After compiler transformation

Initially, a = 0, b = 1

Thread 1	Thread 2
4: b = 2;	5: r3 = b;
1: r1 = a;	6: a = r3;
2: j = r1;	
3: if (true) ;	

r1 == r2 == r3 == 2 is sequentially consistent

Figure 12: Motivation for allowing prohibited reads

Formally defining causality is somewhat involved. To do so, it is necessary to define what it means for one action to be in more than one execution. This definition will be omitted here; the full definition can be found, along with the formal model, in Appendix B.

7.2 When is a race read justified in a causal order?

We need to state more formally what it means for a read r to be justified in occurring in an execution E whose causal order is $\alpha r \beta$. If r is involved in a data race, then execution can non-deterministically choose which of the multiple writes visible to r is seen by r, which makes it difficult to guarantee that r will see a particular value. The read r is justified if in all executions whose causal order consists of α followed by non-prescient actions, there is a corresponding read r' that is allowed to observe the same value that r observed.

Intuitively, we are only able to justify a read if it will always be *allowed to occur* based on the actions we have already justified.

7.3 Prohibited Sets

In order to perform an action presciently, we must be guaranteed that the action will occur. The JMM allows for many possible behaviors of incorrectly synchronized programs. Compilers may perform transformations that prevent some behaviors that are allowed by the JMM. When these behaviors are prohibited by compiler transformations, other actions may be guaranteed that were not guaranteed given the full freedom of the JMM. The semantics need to allow such actions to be performed presciently. In Figure 12, we see an example of such an optimization. The compiler can

- eliminate the redundant read of a, replacing 2 with $r2 = r1$, then
- determine that the expression $r1 == r2$ is always true, eliminating the conditional branch 3, and finally

- move the write `4: b = 2` early.

Here, the assignment `4: b = 2` seems to cause itself to happen. Thus, simple compiler optimizations can lead to an apparent cyclic execution trace without a workable causal order. We must allow these cases, but also prevent cases where, if $r1 \neq r2$, $r3$ is assigned a value other than 1.

To validate such an execution we would need a causal order that makes $r1 == r2 == r3 == 2$ a causally consistent execution of Figure 12. To see this behavior, we need a causal order over valid executions that would justify this behavior in an execution.

How would we go about trying to construct a causal order to validate this behavior? In this case, we are trying to capture a potential behavior of the transformed program: the case where `4` happens first, then all of Thread 2, and finally `1 - 3`. This would suggest { `4, 5, 6, 1, 2, 3` } as a potential causal order.

However, we cannot use this causal order assuming only Causality and Consistency. The prefix of `4 (b = 2)` is empty, so all of the validated executions for which the empty set is a prefix (i.e., all validated executions) must allow the write `4` to occur. The problem is that `4` is not guaranteed to occur in all non-prescient executions; it only occurs whenever $r1$ and $r2$ see the same value. If we were able to exclude all executions in which $r1$ and $r2$ see different values, then we could use the causal order { `4, 5, 6, 1, 2, 3` }

In short, compiler transformations can make certain executions (such as the ones in which `1` and `2` do not see the same value) impossible. This prohibition, in turn, can lead to additional executions that seem cyclic.

For the purposes of showing that a prescient action x is justified, a set of behaviors that are not possible on a particular implementation of a JVM may be specified. This, in turn, allows other actions to be guaranteed and performed presciently, allowing for new behaviors.

If we allowed arbitrary executions to be prohibited, we could conceivably, for example, prohibit all executions. We could then vacuously justify any action we liked, because it would occur in every execution. This sort of behavior is nonsensical; we therefore cannot allow arbitrary executions to be prohibited.

Instead, an execution can be prohibited only if it contains a read that could see multiple values, and there is an *alternate execution* that is not prohibited that contains that read, but sees a different value.

Consider prescient action x in an execution $\langle S, so, \stackrel{hb}{\rightarrow}, \alpha x \beta \rangle$ we wish to justify, and the set of executions J of executions whose causal order starts with α. We allow the specification of a set of prohibited executions $P \subset J$, and only require that x be allowed to occur in all executions in $J - P$.

For each prohibited execution $E' \in P$, we must identify a read r in E', but not in α, that is allowed by the happens before constraint to see multiple writes. Assume that in E' the read r sees a write w. Then there must exist a different write w' that could be seen by r in E' and an execution $E'' \in J - P$ such that

- r and w' occur in E'', and in E'', the read r sees w'.
- E'' includes all of the actions that come before r in the causal order of E'.

Initially, x == y == 0

Thread 1	Thread 2

r1 = x;
if (r1 != 0)
y = 1;
	r2 = y;
	if (r2 != 0)
	x = 1;
	else
	x = 1;

Incorrectly Synchronized: r1 == r2 == 1 Can Happen

Figure 13: Seemingly Cyclic Behavior that is allowed

Initially, x == y == z == 0

Thread 1	Thread 2	Thread 3	Thread 4

r1 = x;
if (r1 != 0)
y = 1;
	r2 = y;
	if (r2 != 0)
	x = 1;
		z = 1	r3 = z
			if (r3 != 0)
			x = 1

Incorrectly Synchronized: r1 == r2 == 1, r3 == 0 can happen in Weak Causality Model, but not in the Strong Causality Model.

Figure 14: Border Line case; differentiates between Strong and Weak Causality Model

Note that since r represents a choice at which E' and E'' diverge, the remainder of their executions can be substantially different.

Using prohibited executions, we can show that the execution in Figure 12 respects causality. This can be done by prohibiting all executions where `h1` and `h2` do not return the same value. Execution traces where they do return the same value can be provided as alternate executions.

7.4 Causality Model Summary and a Possible Weakening

Figure 15 summarizes, using the informal notation used in this section, the strong causality model proposed for the Java Memory Model. However, the JSR-133 expert group is debating whether instead to adopt a weaker model (Figure 16) that would make fewer guarantees about the behavior of incorrectly synchronized programs and might allow for more compiler and other optimizations of Java programs. At the moment, we do not have a good handle on whether guarantees necessary for safety and defensive programming might be violated by the weaker model, or whether there are any compiler or architectural optimizations that would be forbidden by the stronger causality model. The expert group solicits feedback on this issue. The weak causality model allows the behavior shown in Figure 14, while this behavior is forbidden by the strong causality model.

In both the strong and weak models, there are restrictions as to when an operation may

For every execution, there is a total order over actions called the *causal order*. An action x is *prescient* if either:

- x is a read that sees a write that occurs later in the causal order, or
- there exists an action y that occurs after x in the causal order and $y \stackrel{hb}{\to} x$.

Each prescient action x in an execution E must be justified. Let α be the sequence of actions that precedes x in the causal order of E. Let J be the set of all consistent executions whose causal order consists of α followed by non-prescient actions. To prove x is justified, we need to demonstrate a set of prohibited executions $P \subset J$ such that the following is true (P may be, and often is, empty):

- If x is not a read, then we need to show that x occurs in each $E' \in J - P$.
- If x is a read, then we need to show that x occurs in each $E' \in J - P$, with the caveat that x may see different values in E and E'. If different values are seen by x in E and E', there must be a write $w \in E'$ such that w writes the value seen by x in E and x would be allowed to see w in E' by the happens before constraint.
- If the set of prohibited executions P is non-empty, we must show that P is valid. For each prohibited execution $E' \in P$, we must identify a read r in E', but not in α, that is allowed by the happens before constraint to see multiple writes. Assume that in E' the read r sees a write w. Then there must exist a write w' of a different value that could be seen by r in E' and an execution $E'' \in J - P$ such that
  - r and w' occur in E'', and in E'', the read r sees w'.
  - E'' includes all of the actions that come before r in the causal order of E'.

    Note that since r represents a choice at which E' and E'' diverge, the remainder of their executions can be substantially different.

Given these semantics, any execution E where all of the actions are justified is legal. In addition, any execution where the only observable actions are a prefix of the causal order is also legal; this reflects a situation where execution stalls due to deadlock or lack of fairness. The treatment of the memory model in this section is informal; for a formal model, consult Appendix B.

Figure 15: Strong Causality Model for the Java Memory Model

For every execution, there is a total order over actions called the *causal order*. An action x is *prescient* if either:

- x is a read that sees a write that occurs later in the causal order, or
- there exists an action y that occurs after x in the causal order and $y \stackrel{hb}{\to} x$.

Each prescient action x in an execution E must be justified. Let α be the sequence of actions that precedes x in the causal order of E. Let E' be some consistent execution whose causal order consists of α followed by non-prescient actions. To prove x is justified, we need to demonstrate:

- If x is not a read, then we need to show that x occurs in E'.
- If x is a read, then we need to show that x occurs in each E', with the caveat that x may see different values in E and E'. If different values are seen by x in E and E', there must be a write $w \in E'$ such that w writes the value seen by x in E and x would be allowed to see w in E' by the happens before constraint.

Given these semantics, any execution E where all of the actions are justified is legal. In addition, any execution where the only observable actions are a prefix of the causal order is also legal; this reflects a situation where execution stalls due to deadlock or lack of fairness.

Figure 16: Weak Causality Model for the Java Memory Model

be speculatively/presciently performed and end up justifying itself.

- In both the strong and weak model, self-justifying speculation is not allowed if the only way the speculative operation could have occurred is through self-justifying speculation. Figures 10 and 11 show examples of the behavior that both models forbid.

- If an operation is guaranteed to occur without speculation, then it is allowed to occur in a manner that appears to be the result of self-justifying speculation. This behavior is shown in Figure 13. Simple compiler transformations can result in this behavior, so it has to be allowed.

- If an operation *could* occur without speculation, but *is not guaranteed* to occur without speculation, then it is allowed to occur in a manner that appears to be the result of self-justifying speculation under the weak causality model, but not under the strong causality model. The behavior in Figure 14 is allowed by the weak causality model but forbidden by the strong causality model.

8 Final Field Semantics

Final fields were discussed briefly in Section 3.5. Such fields are initialized once and not changed. This annotation can be used to pass immutable objects between threads without synchronization.

Final field semantics are finely based around several competing goals:

- The value of a final field is not intended to change. The compiler should not have to reload a final field because a lock was obtained, a volatile variable was read, or an unknown method was invoked. In fact, the compiler is allowed to hoist reads within thread t of a final field f of an object X to immediately after the very first read of a reference to X by t; the thread need never reload that field.

- Objects that have only final fields and are not made visible to other threads during construction should be perceived as immutable even if references to those objects are passed between threads via data races.

  - Storing a reference to an object X into the heap during construction of X does not necessarily violate this requirement. For example, synchronization could ensure that no other thread could load the reference to X during construction. Alternatively, during construction of X a reference to X could be stored into another object Y; if no references to Y are made visible to other threads until after construction of X is complete, then final field guarantees still hold.

- Making a field f final should impose minimal compiler/architectural cost when reading f.

Thread 1	Thread 2	Thread 3
Foo f = new Foo();	Foo f = G.x;	Bar b = G.y;
Bar b = new Bar()	Bar b = f.b;	int i = b.f;
f.b = b;	G.y = b;	
G.x = f;		

Figure 17: When is Thread 3 guaranteed to correctly see the final field b.f

The use of final fields adds constraints on which writes are considered ordered before which reads, for the purposes of determining if an execution is consistent. Final fields have no direct influence on causality, except in that they influence consistency.

Informally, the semantics for final fields are as follows. Assume a *freeze* action on a final field f of an object X takes place when the constructor for X in which f is written exits.

Let F refer to the freeze action on final field f of object X by thread t_1, and let R refer to a read of $X.f$ in another thread t_2. When is R guaranteed to see the correctly initialized value of $X.f$?

For the moment, assume each thread only reads a single reference to each object. For any object X, thread t_2 must have obtained its address via a chain of the following reference links:

a. Thread t_i wrote a reference to an object Y which was read by another thread t_j

b. Thread t_i read a reference to an object Y, and then read a field of Y to see a reference to another object Z

c. Thread t_i read a reference to an object Y, and later wrote a reference to Y.

If there is an action a in this chain such that $F \xrightarrow{hb} a$, then R is correctly ordered with respect to F, and the thread will observe the correctly constructed value of the final field. If there is no such action, then R does not get that guarantee.

Consider the example shown in Figure 17. An execution of this code is shown in Figure 18, with the reference links shown and labeled. Two reference link chains are shown. In order for the read of `b.f` to be correctly ordered with respect to the construction of the object referenced by `b`, there must exist some action on either chain that is forced by synchronization to occur after construction of that object.

In the more general case, thread t_i may read multiple references to an object Y from different locations. To make the guarantees associated with final fields, it must be possible to find an action a in the chain such that $F \xrightarrow{hb} a$ no matter which read of Y is selected.

An example of this situation can be seen in Figure 19. An object o is constructed in Thread 1 and read by Threads 2 and 3. The reference chain for the read of `t.f` in Thread 2 must be traceable through all reads by Thread 2 of a reference to o. On the chain that goes through the global variable `b`, there is no action that is ordered after the freeze operation,

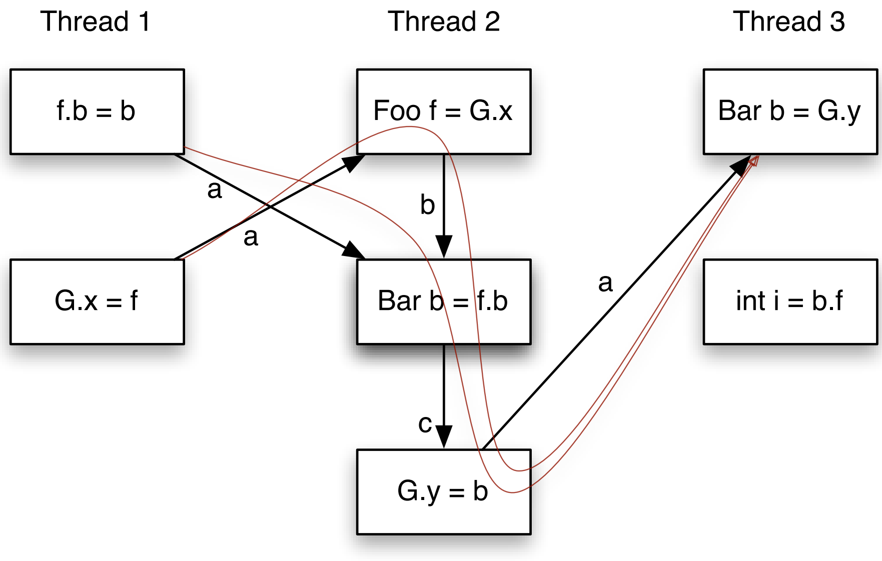


Figure 18: Reference links in an execution of Figure 17

so the read of `t.f` is not correctly ordered with regards to the freeze operation. Therefore, k is not guaranteed to see the correctly constructed value for the final field.

The fact that k does not receive this guarantee reflects legal transformations by the compiler. A compiler can analyze this code and determine that `r.f` and `t.f` are reads of the same final field of the same object. Since final fields are not supposed to change, it could replace `k = t.f` with `k = i` in Thread 2.

All possible reference chains for the read of `s.f` in Thread 3 include the write to `q` in Thread 1. The read is therefore correctly ordered with respect to the freeze operation, and guaranteed to see the correct value.

If a read R of a final field f in thread t_2 is correctly ordered with respect to a freeze

f is a final field; its default value is 0

Thread 1	Thread 2	Thread 3
o.f = 42;	r = p;	s = q;
p = o;	i = r.f;	j = s.f;
freeze o.f;	t = q;	
q = o;	if (t == r)	
	k = t.f;	

We assume r and s do not see the value null. i and k can be 0 or 42, and j must be 42.

Figure 19: Final field example where Reference to object is read twice

Notation	Description
G	The set of freezes associated with a write of an address
freezeBeforeRead(r)	The freezes seen at a read r; if r sees address a, it is used to calculate the set freezesBeforeDereference(t, a)
freezesBeforeDereference(t, a)	The freezes seen before any dereference of a in t. It consists of only the freezes seen at every read in the thread in isolation.
writesBeforeRead(r)	The writes seen at a read r; if r sees address a, it is used to calculate the set writesBeforeDereference(t, o)
writesBeforeDereference(t, o)	The writes seen before every dereference of o in t. It consists of only the writes seen at every read in the thread in isolation.

Figure 20: Sets used in the formalization of final fields

F in thread t_1, then the read is guaranteed to see the value of f set before the freeze F. Furthermore, in thread 2, when reading elements of any object reached in thread 2 only by following a reference loaded from f, those reads are guaranteed to occur after all writes w such that $w \stackrel{hb}{\to} F$.

A final field may only be written by bytecode once. Other techniques, such as deserialization, may cause a final field to be modified after the end of the enclosing object's constructor. There must be a freeze of the final field after each such write. If a reference to an object is shared with other threads between the initial construction of an object and when deserialization changes the final fields of the object, most of the guarantees for final fields of that object can go kerflooey. For details, consult the formal semantics.

8.1 Overview of Formal Semantics of Final Fields

The following is a discussion of the formal semantics of final fields. The semantics themselves can be found in Appendix C. Figure 20 contains a table of all of the sets mentioned below, and their definition.

Each field $o.x$ has an *enclosing object* o, and a set of objects that are reachable by following a chain of dereferences from it. A *final field* may be written to multiple times: once by bytecode in a constructor, and otherwise by VM actions. After the constructor for the enclosing object, a final field is explicitly frozen. After the other writes, the VM may optionally choose to freeze the final field.

For the purposes of this discussion, *freeze* can be considered a noun: a freeze can be copied from thread to thread, and the set of freezes visible to a given thread for a field are the ones that provide the guarantees for that field. A set of freezes G are written at every write of an enclosing object, and a set of freezes freezesBeforeDereference(t, a) are observed at every read of an enclosing object at address a in thread t.

The set G of freezes that are written at every write w of an enclosing object at address a include:

- All the freezes that happen before w, and
- The set freezesBeforeDereference(t, a) consisting of all the freezes that were observed by that thread's read of a.

Each reference a to an object may be stored in fields of several different objects. Each read r in thread t of one of these fields has a set freezeBeforeRead(r) associated with it. This set contains:

- All the freezes that happen before r
- The set G (defined above) that was associated with the write of a, and
- The set of freezes freezesBeforeDereference(t, b) associated with a's enclosing object b (the last object on the dereference chain before a).

The set freezeBeforeRead(r) that is associated with a single read is, however, not the set that determines what freezes are seen when the field is accessed. This set is called freezesBeforeDereference(t, a), and is the intersection of all of the sets freezeBeforeRead(r) whose read saw the address a. The set freezesBeforeDereference(t, a) therefore only contains those freezes that are associated with all of the reads of a given field.

Once we have the set freezesBeforeDereference(t, a) for a given address, we must determine what writes we are guaranteed to see; this is the set writesBeforeDereference(t, a).

To calculate writesBeforeDereference(t, a), we look at all of the places a thread can read a reference to an object. Each of these reads r has a set writesBeforeRead(r) associated with it.

- If the reference to the object was a non-final field, then the writesBeforeRead(r) set is the same as the writesBeforeDereference(t, o) set for the enclosing object.
- If the reference to the object was a final field, then the writesBeforeRead(r) set contains:
  - The writesBeforeDereference(t, o) set for the enclosing object.
  - The set of writes that happen before each freeze of the enclosing object that is present in the set freezesBeforeDereference(t, o) for the object.

The set writesBeforeDereference(t, a) is the intersection of all of the writesBeforeRead(r) sets whose reads saw a ; this gives us only those writes that are associated with all of the reads of a given field.

8.2 Write Protected Fields

Normally, final static fields may not be modified. However `System.in`, `System.out`, and `System.err` are final static fields that, for legacy reasons, must be allowed to be changed by the methods `System.setIn()`, `System.setOut()` and `System.setErr()`. We refer to these fields as being *write-protected* to distinguish them from ordinary final fields.

The compiler needs to treat these fields differently from other final fields. For example, a read of an ordinary final field is "immune" to synchronization: the barrier involved in a lock or volatile read does not have to affect what value is read from a final field. Since the value of write-protected fields may be seen to change, synchronization events should have an effect on them.

Therefore, the semantics dictate that these fields be treated as normal fields that cannot be changed by user code, unless that user code is in the `System` class.

9 Word Tearing

One implementation consideration for Java virtual machines is that every field and array element is considered distinct; updates to one field or element do not interact with reads or updates of any other field or element. In particular, two threads that update adjacent elements of a byte array must not interfere or interact and do not need synchronization to ensure sequential consistency.

Some processors (notably early Alphas) do not provide the ability to write to a single byte. It would be illegal to implement byte array updates on such a processor by simply reading an entire word, updating the appropriate byte, and then writing the entire word back to memory. This problem is sometimes known as *word tearing*, and on processors that cannot easily update a single byte in isolation some other approach will be required. Figure 21 shows a test case to detect word tearing.

10 Treatment of Double and Long Variables

Some Java implementations may find it convenient to divide a single write action on a 64-bit long or double value into two write actions on adjacent 32 bit values. For efficiency's sake, this behavior is implementation specific; Java virtual machines are free to perform writes to long and double values atomically or in two parts.

For the purposes of this memory model, a single write to a non-volatile long or double value is treated as two separate writes: one to each 32-bit half. This can result in a situation where a thread sees the first 32 bits of a 64 bit value from one write, and the second 32 bits from another write. Write and reads of volatile long and double values are always atomic. Writes to and reads of references are always atomic, regardless of whether they are implemented as 32 or 64 bit values.

VM implementors are encouraged to avoid splitting their 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their

```
public class WordTearing extends Thread {
    static final int LENGTH = 8;
    static final int ITERS = 10000;
    static byte[] counts = new byte[LENGTH];
    static Thread[] threads = new Thread[LENGTH];

    final int id;

    WordTearing(int i) { id = i; }

    public void run() {
        for (; counts[id] < ITERS; counts[id]++);
        if (counts[id] != ITERS) {
            System.err.println("Word-Tearing found: " +
                               "counts["+id+"] = " +
                               counts[id]);
            System.exit(1);
        }
    }

    public static void main(String[] args) {
        for (int i = 0; i < LENGTH; ++i)
            (threads[i] = new WordTearing(i)).start();
    }
}
```

Figure 21: Bytes must not be overwritten by writes to adjacent bytes

Thread 1	Thread 2
while (true)	synchronized (o) {
synchronized (o) {	// does nothing.
// does not call	}
// Thread.yield(),	
// Thread.sleep()	
}	

Figure 22: Fairness

programs correctly to avoid this.

11 Fairness

Without a *fairness* guarantee for virtual machines, it is possible for a running thread to be capable of making progress and never do so. One example of a such a guarantee would state that if a thread is infinitely often allowed to make progress, it will eventually do so. Java has no official fairness guarantee, although, in practice, most JVMs do provide it to some extent.

An example of how this issue can impact a program can be seen in Figure 22. Without a fairness guarantee, it is perfectly legal for a compiler to move the `synchronized` block outside the while loop; Thread 2 will be blocked forever.

Any potential fairness guarantee would be inextricably linked to the threading model for a given virtual machine. A threading model that only switches threads when `Thread.yield()` is called will, in fact, never allow Thread 2 to execute. A fairness guarantee makes this sort of implementation illegal; it would force Thread 2 to be scheduled. Because this kind of implementation is often desirable, this specification does not include a fairness guarantee. In particular, for any execution shown to be legal by the semantics for consistency and causality, it would also be legal to execute just the instructions in any prefix of the causal order of that execution.

12 Wait Sets and Notification

Every object, in addition to having an associated lock, has an associated wait set. A wait set is a set of threads. When an object is first created, its wait set is empty. Elementary actions that add threads to and remove threads from wait sets are atomic. Wait sets are manipulated in Java solely through the methods `Object.wait`, `Object.notify` , and `Object.notifyAll`.

Wait set manipulations can also be affected by the interruption status of a thread, and by the `Thread` class methods dealing with interruption. Additionally, `Thread` class methods for sleeping and joining other threads have properties derived from those of wait and notification actions.

12.1 Wait

Wait actions occur upon invocation of `wait()`, or the timed forms `wait(long millisecs)` and `wait(long millisecs, int nanosecs)`. A call of `wait(long millisecs)` with a parameter of zero, or a call of `wait(long millisecs, int nanosecs)` with two zero parameters, is equivalent to an invocation of `wait()`.

Let thread t be the thread executing the wait method on Object m, and let n be the number of lock actions by t on m that have not been matched by unlock actions. One of the following actions occurs.

- If n is zero (i.e., thread t does not already possess the lock for target m) an IllegalMonitorStateException is thrown.
- If this is a timed wait and the nanosecs argument is not in the range of 0-999999 or the millisecs argument is negative, an IllegalArgumentException is thrown.
- If thread t is interrupted, an InterruptedException is thrown and t's interruption status is set to false.
- Otherwise, the following sequence occurs:
  1. Thread T is added to the wait set of object m, and performs n unlock actions on m.
  2. Thread t does not execute any further Java instructions until it has been removed from m's wait set. The thread may be removed from the wait set due to any one of the following actions, and will resume sometime afterward.
     - A notify action being performed on m in which t is selected for removal from the wait set.
     - A notifyAll action being performed on m.
     - An interrupt action being performed on t.
     - If this is a timed wait, an internal action removing t from m's wait set that occurs after at least `millisecs` milliseconds plus `nanosecs` nanoseconds elapse since the beginning of this wait action.
     - An internal action by the Java VM implementation. Implementations are permitted, although not encouraged, to perform "spurious wake-ups" – to remove threads from wait sets and thus enable resumption without explicit Java instructions to do so. Notice that this provision necessitates the Java coding practice of using wait() only within loops that terminate only when some logical condition that the thread is waiting for holds.
  3. Thread t performs n lock actions on m.
  4. If thread t was removed from m's wait set in step 2 due to an interrupt, t's interruption status is set to false and the wait method throws InterruptedException.

If t was not removed due to an interrupt, but t is interrupted before it completes step 3, then the system may be allowed to set t's interruption status is set to false and the wait method throws InterruptedException. Otherwise the method returns normally.

12.2 Notification

Notification actions occur upon invocation of methods `notify` and `notifyAll()`. Let thread t be the thread executing either of these methods on Object m, and let n be the number of lock actions by t on m that have not been matched by unlock actions. One of the following actions occurs.

- If n is zero an `IllegalMonitorStateException` is thrown. This is the case where thread t does not already possess the lock for target m.

- If n is greater than zero and this is a notify action, then, if m's wait set is not empty, a thread u that is a member of m's current wait set is selected and removed from the wait set. (There is no guarantee about which thread in the wait set is selected.) This removal from the wait set enables u's resumption in a wait action. Notice however, that u's lock actions upon resumption cannot succeed until some time after t fully unlocks the monitor for m. Also notice that the behavior of wait implies that notify causes some thread (if one exists) to return normally after re-locking m's monitor, rather than throwing an InterruptedException. However, Java programs cannot rely on exactly which wait set removal action occurs when a notification and an interruption action execute at approximately the same time.

- If n is greater than zero and this is a notifyAll action, then all threads are removed from m's wait set, and thus resume. Notice however, that only one of them at a time will lock the monitor required during the resumption of wait.

12.3 Interruptions

Interruption actions occur upon invocation of method `Thread.interrupt()`, as well as methods defined to in turn invoke it, such as `ThreadGroup.interrupt()`. Let t be the thread invoking `U.interrupt()`, for some thread u, where t and u may be the same. This action causes u's interruption status to be set to true.

Additionally, if there exists some object m whose wait set contains u, u is removed from m's wait set. This enables u to resume in a wait action, in which case this wait will, after re-locking m's monitor, throw `InterruptedException`.

Invocations of `Thread.isInterrupted()` can determine a thread's interruption status. Any thread may observe and clear its own interruption status by invoking (static) method `Thread.interrupted()`.

12.4 Unresolved Question on wait/notify/interrupt semantics

The JSR133 expert group is still trying to decide whether to use a strong or weak version of the semantics associated with wait(), notify() and interrupts. Feedback from the community review is solicited on this issue.

The question has to do with whether an interrupt can cause a notification to be lost. In particular, consider the situation where threads $T1$ and $T2$ have both performed a wait on an object `X` and are in the waitset for `X`. Then the following two actions are performed, in either order:

- $T1$ is interrupted
- `X.notify()` is invoked

Which of the following behaviors is allowed?

a) $T1$ returns from the call to `wait()` by throwing `InterruptedException`, $T2$ returns normally from the call to `wait()`

b) $T1$ returns normally from the call to `wait()` and still has a pending interrupt; $T2$ remains in the waitset for `X`

c) $T1$ returns from the call to `wait()` by throwing `InterruptedException`; $T2$ remains in the waitset for `X`

Behavior (a) is the preferred behavior, and the strongest version of the wait/interrupt semantics allows only behavior (a). Weaker versions of the wait/interrupt semantics would also allow either (b), (c) or both.

12.5 Sleep

An invocation of method `Thread.sleep(long millisecs)` is not guaranteed to be behaviorally distinguishable from the action:

```
if (millisecs != 0) {
   Object s = new Object();
   synchronized (s) {
    long startTime = System.currentTimeMillis();
    long waitTime = millisecs;
    for (;;) {
      s.wait(waitTime);
      long now = System.currentTimeMillis();
      waitTime = millisecs - (now - startTime);
      if (waitTime <= 0)
        break;
    }
```

```
    }
  }
```

In this code, s is an object that is not otherwise used in any way: it is not accessed by another thread. The method `Thread.sleep(long millisecs, int nanosecs)` operates identically to `Thread.sleep(long millisecs)`, except that it accommodates nanosecond timing arguments.

It is important to note that neither `Thread.sleep` nor `Thread.yield` have any synchronization semantics. In particular, the compiler does not have to flush writes cached in registers out to shared memory before a call to sleep or yield, nor does the compiler have to reload values cached in registers after a call to sleep or yield. For example, in the following (broken) code fragment, assume that `this.done` is a non-volatile boolean field:

```
while (!this.done)
  Thread.sleep(1000);
```

The compiler is free to read the field `this.done` just once, and reuse the cached value in each execution of the loop. This would mean that the loop would never terminate, even if another field changed the value of `this.done`.

A Compiler and Architectural Optimizations Allowed

The language semantics do not describe which optimizations and transformations are allowed and which are prohibited. Instead, the semantics only describe the allowed and prohibited behaviors.

The compiler, VM and processor may compile, transform and execute a program in anyway that exhibits only allowed behaviors of the original program. For example, the compiler may perform transformations that seem at odds with the spirit of the semantics, so long as the compiler can prove that the transformation is not detectable. In other words, if the compiler can't be caught, it isn't illegal.

Now, many transformations can have effects on efficiency, fairness, and other important issues. These issues are consider to be quality of service issues, rather than semantics. Compilers that make transformations that hurt efficiency or fairness would be legal, but undesirable.

Although this specification is not defined in terms of which transformations as legal, you can derive proofs that certain transformations are legal. These include:

- in the absence of synchronization, performing all of the standard reordering transformations allowed in a single threaded context.
- removing/ignoring synchronization on thread local objects
- removing/ignoring redundant synchronization (e.g., when a synchronized method is called from another synchronized method on the same object).
- treating volatile fields of thread local objects as normal fields

$$
\begin{aligned}
&E = \langle S, so, \xrightarrow{hb}, co\rangle \in \text{valid} \iff E \in \text{consistent} \wedge \\
&\quad \forall x \in co, x \in \text{prescient} \Rightarrow \\
&\qquad co = \alpha x \beta \wedge \\
&\qquad \textbf{let } J = \{E' = \langle S', so', \xrightarrow{hb'}, \alpha'\beta'\rangle \mid \langle S', so', \xrightarrow{hb'}\rangle \in \text{consistent} \\
&\qquad\qquad\qquad \wedge \text{ length}(\alpha) = \text{length}(\alpha') \\
&\qquad\qquad\qquad \wedge\ \beta' \text{ does not contain prescient actions} \\
&\qquad\qquad\qquad \wedge\ \alpha \preceq \alpha'\} \\
&\qquad\quad \textbf{in } \exists P \subset J \\
&\qquad\qquad \forall E' = \langle S', so', \xrightarrow{hb'}, \alpha'\beta'\rangle \in J - P \\
&\qquad\qquad\qquad \exists x' \in \beta' : x' \mapsto x \wedge \\
&\qquad\qquad \forall E_p \in P \\
&\qquad\qquad\qquad E_p = \langle S_p, so_p, \xrightarrow{hb_p}, \alpha_p r_p \beta_p\rangle \wedge \\
&\qquad\qquad\qquad \exists E'_p = \langle S'_p, so'_p, \xrightarrow{hb'_p}, \alpha'_p r'_p \beta'_p\rangle \in J - P \wedge \\
&\qquad\qquad\qquad\quad \alpha_p \preceq \alpha'_p \wedge \\
&\qquad\qquad\qquad\quad r_p \mapsto r'_p \wedge \\
&\qquad\qquad\qquad\quad r_p \text{ and } r'_p \text{ observe different values}
\end{aligned}
$$

Given this, any execution $E \in$ valid is legal. In addition, any execution where the only observable actions are a prefix of the causal order is also legal; this reflects a situation where due to deadlock or lack of fairness, execution stalls.

Figure 23: Full Semantics With Strong Causality Model

B Formal Model of Strong Causality

This section describes, in more detail, the formal model for Strong Causality of Java programs. It includes formal definitions for what it means for an action to be present in multiple executions. In addition, the full semantics for strong causality of a program P are detailed in Figure 23.

B.1 Definitions

For the model to work in the way we described, it is necessary to define what it means

- For one action to be contained in two different executions (*action correspondence*), and
- For one execution to be a prefix of another (*execution containment*).

Given $E = \langle S, so, \xrightarrow{hb}, \alpha x \beta \rangle$, $E' = \langle S', so', \xrightarrow{hb'}, \alpha' \beta' x' \gamma' \rangle$,

- $(x, \alpha) \cong (x', \alpha') \iff$
  - $\text{length}(\alpha) = \text{length}(\alpha')$
  - $\forall i, 0 \leq i < \text{length}(\alpha) : (\alpha_i, \alpha_{k<i}) \cong (\alpha'_i, \alpha'_{k<i}) \wedge$
    $\alpha_i \xrightarrow{hb} x \iff \alpha'_i \xrightarrow{hb'} x' \wedge$
    $x \xrightarrow{hb} \alpha_i \iff x' \xrightarrow{hb'} \alpha'_i$
- $\alpha \preceq \alpha' \iff$
  - $\forall i, 0 \leq i < \text{length}(\alpha) : (\alpha_i, \alpha_{k<i}) \cong (\alpha'_i, \alpha'_{k<i})$
  - $\forall i, 0 \leq i < \text{length}(\alpha)$: all of the information with which α_i is annotated (including the monitor accessed, the variable read or written and the value read or written) is the same as that for α'_i.
- $x \in \text{prescient}_E \iff$
  - $\exists y \in \beta$:
    * $y \xrightarrow{hb} x \vee$
    * (x is a read)$\wedge$(y is a write)$\wedge$(x observes the write performed by y)
- $x \mapsto x' \iff$
  - $(x, \alpha) \cong (x', \alpha')$
  - if x' is a read, it is allowed to observe the same value that x observes

Figure 24: Definitions

B.1.1 Action Correspondence

To use causal orders, we must first define the notion of what it means for two actions to correspond to each other in two different executions. Two actions and the causal orders that justify them correspond to each other in separate executions if

- The causal orders that justify each are the same length, and
- All of the elements in each of the causal orders are the same, and
- If the i^{th} element of the causal order that justifies one happens before the action, then the i^{th} element of the causal order that justifies the other happens before the action.

For two actions a and b, with causal orders that justify them α and β, this is written $(a, \alpha) \cong (b, \beta)$. This is laid out more formally in Figure 24. Additionally, if $(a, \alpha) \cong (b, \beta)$, a and b are reads, and b is allowed to read the same value that a read, we say $a \mapsto b$.

B.1.2 Execution Containment

For causal orders to work, it must be clear what it means for one execution to contain another.

The causal order co' of an execution $E = \langle S', so', \stackrel{hb'}{\rightarrow}, co' \rangle$ contains a causal order co (written $co \preceq co'$) if, for the i^{th} action co_i of co , $(co_i, co_{k<i}) \cong (co'_i, co'_{k<i})$ and all of the information with which co_i is annotated (including the monitor accessed, the variable read or written and the value read or written) is the same as that for co'_i.

The formal statement of these definitions can be found in Figure 24.

C Final Field Semantics

The formal semantics of final fields are different from those of normal fields. For final fields, they supersede the ordinary rules for happens-before edges (as described in Section 5); for non-final fields, they may be considered a supplement.

C.1 Freezes Associated with Writes

When an address a is stored in the heap by thread t at write w, it is stored as a pair $\langle a, G \rangle$, where G is a set of freeze actions defined as:

$$G = \{f \mid f \stackrel{hb}{\rightarrow} w\} \cup \text{freezesBeforeDereference}(t, a)$$

The set freezesBeforeDereference(t, a) is the set of freezes associated with the address a in thread t, as defined below.

C.2 The Effect of Reads

A read r in thread t of field x of the object at address c returns a tuple $\langle a, G \rangle$, where a is the value returned and G is a set of freeze actions as defined in Section C.1. Each such read has two corresponding sets. The first, the set freezeBeforeRead(r), is a set of freezes associated with the read. The second, the set writesBeforeRead(r), is a set of writes associated with the read. These sets are used to compute the values that are legal for final fields.

C.2.1 Freezes Seen as a Result of Reads

Consider a read r in thread t of field x of the object at address c that returns a tuple $\langle a, G \rangle$. The set of freezes freezeBeforeRead(r) associated with a read r of address a is:

$$\text{freezeBeforeRead}(r) = G \cup \{f \mid f \xrightarrow{hb} r\} \cup \text{freezesBeforeDereference}(t, c)$$

The set freezesBeforeDereference(t, a) is the intersection of the sets of freezes that the thread saw every time it read a reference to o: this is the set freezeBeforeRead(r). Let sawAddress(t, a) be the set of reads in thread t that returned the address a.

$$\text{freezesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{freezeBeforeRead}(r)$$

If a thread t allocated a (including all situations where sawAddress(t, a) is empty), then the set freezesBeforeDereference(t, a) is empty.

The actual freezesBeforeDereference sets are defined by the least fixed point solution to these equations (i.e., the smallest sets that satisfy these equations). This is because the definition of freezesBeforeDereference(t, a) uses freezeBeforeRead(t, c), .

C.2.2 Writes Visible at a Given Read

For any read instruction r, there is a set of writes, writesBeforeRead(r), that is known to be ordered before r due to the special semantics of final fields. These ordering constraints are taken into account in determining which writes are visible to the read r. However, these ordering constraints do not otherwise compose with the standard happens-before ordering constraints.

We define the set writesBeforeRead(r) in terms of the writes that are known to occur before any dereference of an address c by thread t. These writes are given by the set writesBeforeDereference(t, c). Like the equations for freezes, these equations are recursive; the solution is defined to be the least fixed point solution.

Result set for non-final fields or array elements Consider a read r in thread t of non-final field or element x of the object at address c. The set of writes writesBeforeRead(r) is defined as:

$$\text{writesBeforeRead}(r) = \text{writesBeforeDereference}(t, c)$$

Result set for final fields Consider a read r in thread t of final field x of the object at address c. The set of writes writesBeforeRead(r) is defined as:

$$\begin{aligned}
&\text{writesBeforeRead}(r) = \\
&\quad \text{writesBeforeDereference}(t, c) \cup \\
&\qquad\qquad \{w \mid \exists f \text{ s.t. } \; f \in \text{freezesBeforeDereference}(t, c) \\
&\qquad\qquad\qquad\qquad \wedge \;\; f \text{ is a freeze of } c.x \\
&\qquad\qquad\qquad\qquad \wedge \;\; w \overset{hb}{\rightarrow} \; f\}
\end{aligned}$$

Result set for static fields The set writesBeforeRead(r) associated with a read r of a static field is the empty set.

Visible Write Set The set writesBeforeDereference(t, a) is defined to be the intersection of the writesBeforeRead sets for all reads that see the value a.

$$\text{writesBeforeDereference}(t, a) = \bigcap_{r \in \text{sawAddress}(t,a)} \text{writesBeforeRead}(r)$$

If a thread t allocated a then writesBeforeDereference(t, a) is empty. This includes any situations where sawAddress(t, a) is empty. As with freezesBeforeDereference, these equations are recursive and the solution is defined to be the least fixed point solution to the equations (i.e., the smallest sets that satisfy these equations).

When a read r examines the contents of any field $a.x$ in thread t, all of the writes in writesBeforeRead(r) are considered to be ordered before r. If $a.x$ is a final field, these are the only writes considered to be ordered before r. In addition, if $a.x$ is a final static field, then r will always return $a.x$'s correctly constructed value, unless r happens in the thread that performed the class initialization, before the field was written.

C.3 Single Threaded Guarantees for Final Fields

For cases where a final field is set once in the constructor, the rules are simple: the reads and writes of the final field in the constructing thread are ordered according to program order.

We must treat cases such as deserialization, where a final field can be modified after the constructor is completed, a little differently. Consider the situation where a program:

- Reads a final field, then
- calls a method that rewrites that final field, and finally
- re-reads the final field.

Because reads of final fields can be reordered around method boundaries, the compiler may reuse the value of the first read for the second read. The limitation we place on this is that if the method returns a "new" reference to the final field's enclosing object, and the final field is read via that reference, then the program will see the rewritten value of the final field. If it uses the "old" reference to the final field's enclosing object, then the program may see either the original value or the new one.

Conceptually, before a program modifies a frozen final field, the system must call a `realloc()` function, passing in a reference to the object, and getting out a reference to the object through which the final fields can be reassigned. The only appropriate way to use this `realloc()` function is to pass the only live reference to the object to the `realloc()` function, and only to use that value `realloc()` returns to refer to the object after that call.

After getting back a "fresh" copy from `realloc()`, the final fields can be modified and refrozen. The `realloc()` function need not actually be implemented at all; the details are hidden inside the implementation. However, it can be thought of as a function that might decide to perform a shallow copy.

In more detail, each reference within a thread essentially has a version number. Passing a reference through `realloc()` increments that version number. A read of a final field is ordered according to program order with all writes to that field using the same or smaller version number.

Two references to the same object but with different version numbers should not be compared for equality. If one reference is ever compared to a reference with a lower version number, then that read and all reads of final fields from that reference are treated as if they have the lower version number.

D Finalization

This appendix details changes to Section 12.6 of the Java language specification, which deals with finalization. The relevant portions are reproduced here.

The class `Object` has a protected method called `finalize`; this method can be overridden by other classes. The particular definition of `finalize` that can be invoked for an object is called the *finalizer* of that object. Before the storage for an object is reclaimed by the garbage collector, the Java virtual machine will invoke the finalizer of that object.

Finalizers provide a chance to free up resources that cannot be freed automatically by an automatic storage manager. In such situations, simply reclaiming the memory used by an object would not guarantee that the resources it held would be reclaimed.

The Java programming language does not specify how soon a finalizer will be invoked, except to say that it will happen before the storage for the object is reused. Also, the language does not specify which thread will invoke the finalizer for any given object. It is guaranteed, however, that the thread that invokes the finalizer will not be holding any user-visible synchronization locks when the finalizer is invoked. If an uncaught exception is thrown during the finalization, the exception is ignored and finalization of that object terminates.

It is important to note that many finalizer threads may be active (this is sometimes needed on large SMPs), and that if a large connected data structure becomes garbage, all of the finalize methods for every object in that data structure could be invoked at the same time, each running in a different thread.

The `finalize` method declared in class Object takes no action.

The fact that class `Object` declares a `finalize` method means that the `finalize` method for any class can always invoke the `finalize` method for its superclass. This should always be done, unless it is the programmer's intent to nullify the actions of the finalizer in the superclass. Unlike constructors, finalizers do not automatically invoke the finalizer for the superclass; such an invocation must be coded explicitly.)

For efficiency, an implementation may keep track of classes that do not override the `finalize` method of class Object, or override it in a trivial way, such as:

```
protected void finalize() throws Throwable {
   super.finalize();
}
```

We encourage implementations to treat such objects as having a finalizer that is not overridden, and to finalize them more efficiently, as described in Section D.1.

A finalizer may be invoked explicitly, just like any other method.

The package java.lang.ref describes weak references, which interact with garbage collection and finalization. As with any API that has special interactions with the language, implementors must be cognizant of any requirements imposed by the java.lang.ref API. This specification does not discuss weak references in any way. Readers are referred to the API documentation for details.

D.1 Implementing Finalization

Every object can be characterized by two attributes: it may be *reachable*, *finalizer-reachable*, or *unreachable*, and it may also be *unfinalized*, *finalizable*, or *finalized*.

A *reachable* object is any object that can be accessed in any potential continuing computation from any live thread. Any object that may be referenced from a field or array element of a reachable object is reachable. Finally, if a reference to an object is passed to a JNI method, then the object must be considered reachable until that method completes.

A class loader is considered reachable if any instance of a class loaded by that loader is reachable. A class object is considered reachable if the class loader that loaded it is reachable.

Optimizing transformations of a program can be designed that reduce the number of objects that are reachable to be less than those which would naïvely be considered reachable. For example, a compiler or code generator may choose to set a variable or parameter that will no longer be used to null to cause the storage for such an object to be potentially reclaimable sooner.

Another example of this occurs if the values in an object's fields are stored in registers. The program then may access the registers instead of the object, and never access the object again. This would imply that the object is garbage.

Note that this sort of optimization is only allowed if references are on the stack, not stored in the heap. For example, consider the *Finalizer Guardian* pattern:

```
class Foo {
  private final Object finalizerGuardian = new Object() {
    protected void finalize() throws Throwable {
      /* finalize outer Foo object */
    }
  }
}
```

The finalizer guardian forces a super.finalize() to be called if a subclass overrides finalize and does not explicitly call super.finalize().

If these optimizations are allowed for references that are stored on the heap, then the compiler can detect that the $finalizerGuardian$ field is never read, null it out, collect the object immediately, and call the finalizer early. This runs counter to the intent: the programmer probably wanted to call the `Foo` finalizer when the `Foo` instance became unreachable. This sort of transformation is therefore not legal: the inner class object should be reachable for as long as the outer class object is reachable.

Transformations of this sort may result in invocations of the `finalize` method occurring earlier than might be otherwise expected. In order to allow the user to prevent this, we enforce the notion that synchronization may keep the object alive. *If an object's finalizer can result in synchronization on that object, then that object must be alive and considered reachable whenever a lock is held on it.*

Note that this does not prevent synchronization elimination: synchronization only keeps an object alive if a finalizer might synchronize on it. Since the finalizer occurs in another thread, in many cases the synchronization could not be removed anyway.

A *finalizer-reachable* object can be reached from some finalizable object through some chain of references, but not from any live thread. An *unreachable* object cannot be reached by either means.

An *unfinalized* object has never had its finalizer automatically invoked; a *finalized* object has had its finalizer automatically invoked. A *finalizable* object has never had its finalizer automatically invoked, but the Java virtual machine may eventually automatically invoke its finalizer. An object cannot be considered finalizable until its constructor has finished. Every pre-finalization write to a field of an object must be visible to the finalization of that object. Furthermore, none of the pre-finalization reads of fields of that object may see writes that occur after finalization of that object is initiated.