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

Proposed Final Draft

April 12, 2004, 6:15pm

This document is the proposed final draft version 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 document 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 unusually detailed and technical, involving insights and advances in a number of academic topics. This discussion is archived (and continues) at the JMM web site. The web site provides additional information that may help in understanding how this specification was arrived at; it is located at `http://www.cs.umd.edu/~pugh/java/memoryModel`.

The core semantics (Sections 4 – 7) is intended to describe semantics allowed by 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 could conceivably limit potential future optimizations and implementations.

The major change since the public review period is the new formalism for expressing the semantics. This only codifies the previous behavior described in previous versions of the specifications.

Readers are urged to examine closely the semantics on final fields (Sections 3.5 and 9). 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 confusing and counterintuitive. This specification describes the semantics of multithreaded Java programs, including rules for which values may be seen by a read of shared memory that is updated by multiple threads. As the specification is similar to the *memory models* for different hardware architectures, these semantics are referred to as the *Java memory model*.

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 observe r2 == 2, r1 == 1

Valid compiler transformation

Initially, A == B == 0

Thread 1	Thread 2
B = 1;	A = 2
r2 = A;	r1 = B

May observe r2 == 2, r1 == 1

Figure 1: Surprising results caused by statement reordering

The Java programming language neither prevents nor requires 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.

1.2 Notation in examples

The Java memory model is not fundamentally based in the Object-Oriented nature of the Java programming language. For concision 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. Most examples consist of two or more threads containing statements with access to local variables, shared global variables 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, either instruction 1 or instruction 3 must come first in an execution. If instruction 1 comes first, it should not be able to see the write at instruction 4. If instruction 3 comes first, it should not be able to see the write at instruction 2.

If some execution exhibited this behavior, then we would know that instruction 4 came before instruction 1, which came before instruction 2, which came before instruction 3, which came before instruction 4. This is, on the face of it, absurd.

However, compilers are allowed to reorder the instructions in each thread, when this does not affect the exeecution of that thread in isolation. 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
`a = p.x;`	`p.x = 3`
`b = q.x;`	
`c = p.x;`	

May observe `a == c == 0, b == 3`

Valid compiler transformation

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

Thread 1	Thread 2
`a = p.x;`	`p.x = 3`
`b = q.x;`	
`c = a;`	

May observe `a == c == 0`, `b == 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.

When this occurs, it is called a *data race*. When code contains a data race, counterintuitive results are often possible.

Several mechanisms can produce 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 can reorder and transform programs, but must do so only in the ways allowed by this specification.

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 `a` reused for `c`: 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 `a` and `c` will have the value 0, and `b` will have the value 3. This may seem counterintuitive 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: this was 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 counterintuitive 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 counterintuitive 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.

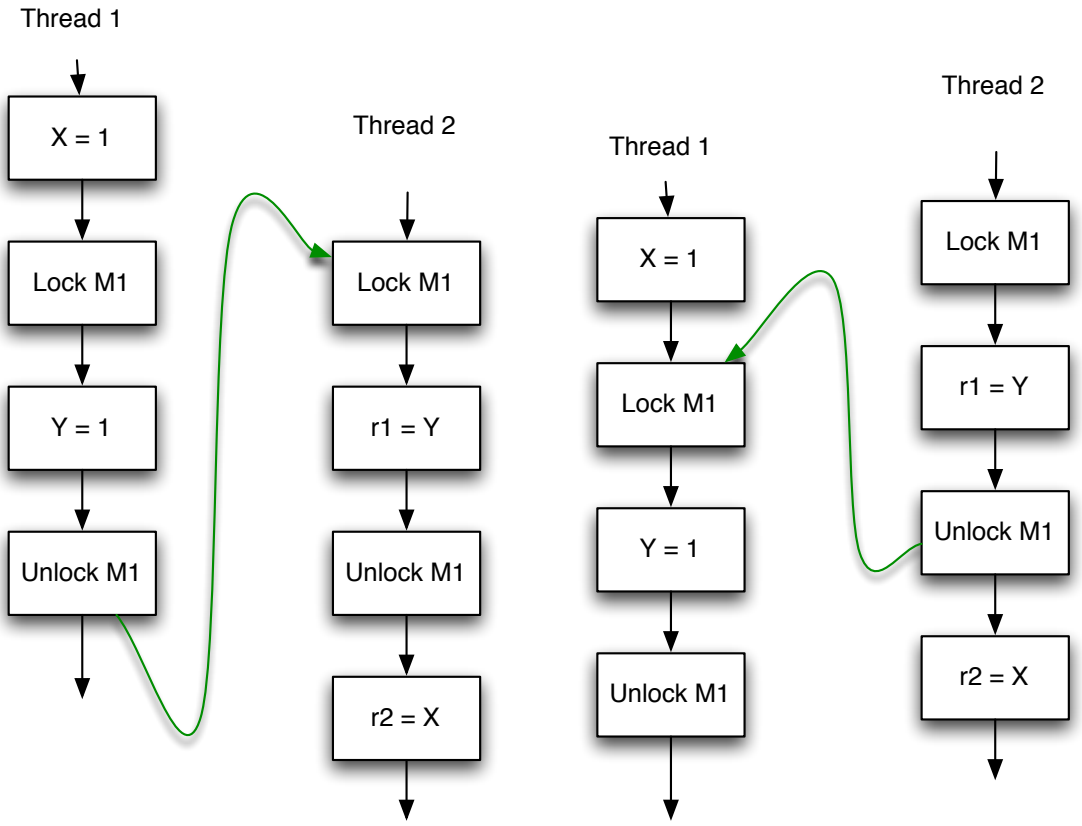
Happens-Before Relationship Two actions can be ordered by a *happens-before* relationship. If one action happens before another, then the first is visible to and ordered before the second. It should be stressed that a happens before relationship between two actions does not imply that those actions must occur in that order in a Java implementation. Rather, it implies that if they occur out of order, that fact cannot be detected. 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. In order to guarantee that the results of one action are observable to a second action, then the first must happen 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 no writes are performed to `done` in the first thread; the compiler may hoist the read of `done` out of the loop, transforming it into 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`. Conceptually, all actions on volatiles happen in a single order, and each write to a volatile field happens before any read of that volatile that occurs later in that order.

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 `threadOne()` gets executed in one thread and `threadTwo()` gets executed in another. Would it be possible for `threadTwo()` to return the value `true`?

The Java memory model allows this result, illustrating a violation of the ordering that a

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

  void threadOne() {
    a = true;
    b = true;
  }

  boolean threadTwo() {
    boolean r1 = b; // sees true
    boolean r2 = a; // sees false
    return r1 && !r2; // returns true
  }

}
```

Figure 5: Ordering example

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 assignments to `a` and `b` in `threadOne()` can be performed out of order. Compilers have substantial freedom to reorder code in the absence of synchronization. This might result in `threadTwo()` being executed after the assignment to `b`, but before the assignment to `a`.

To avoid this behavior, programmers must ensure that their code is correctly synchronized.

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 11 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 on a sequence of actions. A program can be free from data races without having this form of atomicity. However, enforcing this kind of atomicity is frequently as important to program correctness as enforcing 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

Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. 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. If a program has no data races, then all executions of the program will appear to be sequentially consistent. 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.

If we were to use sequential consistency as our memory model, many of the compiler and processor optimizations that we have discussed would be illegal. For example, in Figure 2, as soon as the write of 3 to `p.x` occurred, both subsequent reads of that location would be required to see that value.

Having discussed sequential consistency, we can use it to provide an important clarification regarding data races and correctly synchronized programs. A data race occurs in an execution of a program if there are conflicting actions in that execution that are not ordered by synchronization. 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

```
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

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 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's constructor is finished. If this is followed, then when the object is seen by another thread, that thread will always see the correctly constructed version of that object's final fields. It will also see versions of any object or array referenced by those final fields that are at least as up-to-date as the final fields are.

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,

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`

`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 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, even if malicious code is using data races to pass String references between threads.

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 9.

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.

The memory model predicts the possible behaviors of a program. An implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model. This provides a great deal of freedom for the Java implementor to perform a myriad of code transformations, including reordering actions and removing unnecessary synchronization. For more detail on some permitted optimizations, see Appendix A.

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 are not affected directly by the memory model or data races. 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 the VM, allow access outside an array 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. When we refer to this, we say that the program obeys *intra-thread semantics*. However, when threads interact, reads can return values written by writes from different threads.

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 locking or unlocking a monitor, reading or writing a volatile 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). As previously mentioned, all threads need to obey the correct intra-thread semantics for Java programs

Every inter-thread action is associated with information about the execution of that action; we refer to that information as *annotation*. 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 it would be performed in a single threaded context, as defined in the remainder of the Java Language Specification.

Each time the 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 determine 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. These include locks, unlocks, reads of and writes to volatile variables, actions that start a thread, and actions that detect that a thread is done.

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. We call the resulting edges *synchronized-with* edges, and they are defined as follows:

- 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 by any thread (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 detects that T1 has terminated. T2 may accomplish this by calling `T1.isAlive()` or doing a join action on T1.
- If thread T1 interrupts thread T2, there is a happens-before edge from the interrupted by T1 to the point where any other thread (including T2) determines that

T2 has been interrupted (by having an InterruptedException thrown or by invoking Thread.interruped or Thread.isInterrupted).

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 $x \stackrel{hb}{\rightarrow} z$.

It should be noted that the presence of a happens-before relationship between two actions does not necessarily imply that they have to take place in that order in an implementation. However, they have to appear to share an ordering to the rest of the program. If it cannot be caught, it is not illegal. For example, the write of a default value to every field of an object constructed by a thread need not occur before the beginning of that thread, as long as no read ever observes that fact.

Notice that this does not mean that actions that share a happens-before relationship have to appear to be ordered with respect to any code with which they do not share a happens-before relationship. The fact that the actions may be reordered might be detectable based on values seen by other threads. Writes, for example, might be seen to occur out of order by other threads involved in a data race with those writes.

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 13.

6 Requirements and Goals for the Java Memory Model

Before going into details about the properties of the Java Memory Model, it is first useful to detail the requirements and goals for the Java memory model. It is important to understand that this is not the specification of the Java Memory Model, but merely properties that the Java Memory Model will have.

6.1 Intra-thread Semantics

These are the standard semantics for a single thread's execution. They are defined more fully in Section 5.

6.2 Correctly Synchronized Programs have Sequentially Consistent Behavior

It is important for most programs in Java to have relatively straightforward semantics. As discussed in Section 3.4, if code is written so that it has no data races, it will behave as if

it is sequentially consistent; most of the unusual and counterintuitive behaviors discussed in this document will not appear if programmers adhere to this model. This behavior when programs are correctly synchronized is a property that has long been advocated and accepted for memory models.

6.3 Standard intra-thread compiler transformations are legal

Many compiler transformations have been developed over the years; these include the reordering of non-conflicting memory accesses and the storage of values in registers. It is a goal of the Java memory model for all such transformations to be legal. Of course, this blanket proclamation is true only for transformations that do not interact with thread semantics. For example, the reordering of a normal memory access and a volatile memory access is not legal in general, although it is legal in a number of more specific cases. Similarly, transformations may not be universally allowable if they depend upon analyzing what values can be seen by reads of variables shared with other threads.

One important exception is that it is not legal, in general, to introduce additional reads and writes into the program in a way that is detectable. For example, if `r1` is a local variable and `x` is a shared variable, the compiler may not replace `r1 = x; return r1+r1` with `return x+x`. The code fragment should always return a value that is twice that of a value read for `x`. The transformed fragment `return x+x` could return an odd value.

6.4 Reordering of memory accesses and synchronization actions

Some operations (e.g., writing a volatile variable, unlocking a monitor, starting a thread) have release semantics: they allow the actions that happen before the release to be visible to and ordered before following actions. Other operations (e.g., reading a volatile variable, locking a monitor) have acquire semantics: they allow actions in a different thread to see and be ordered after earlier actions.

It is generally legal to reorder a normal memory access and a following acquire action, or a release action and a following normal memory access. This has been referred to as "roach motel" semantics: variable accesses can be moved into a synchronized block, but cannot, in general, move out.

6.5 Standard processor memory models are legal

Different processor architectures (e.g., PowerPC, IA64, IA32, and SPARC) have formally defined memory models (some, such as SPARC, have more than one). Although the processors vary in terms of how actions such as volatile accesses and synchronization need to be implemented, the standard processor architectures can all directly implement non-volatile, non-synchronization actions without any additional memory barriers. Note, however, that some processor architectures (specifically, Alpha and DSM architectures) may have more implementation issues for correct handling of accesses to final fields.

Initially, A == B == 0

Thread 1	Thread 2
r1 = new Object();	r2 = new Object();
synchronized (r1) {	synchronized (r2) {
A = 1;	r3 = B;
}	}
synchronized (r1) {	synchronized (r2) {
B = 1;	r4 = A;
}	}

Figure 9: Useless synchronization may be removed; May observe `r3 = 1` and `r4 = 0`

6.6 Useless Synchronization can be Ignored

A synchronization action is useless in a number of situations, including lock acquisition on thread-local objects or the reacquisition of a lock on an object on which a thread already has a lock. A number of papers have been published showing compiler analyses that can detect and remove useless synchronization. The old JMM did not allow useless synchronization to be completely removed; the new JMM does.

For example, in Figure 9, the objects referenced by r1 and r2 are thread local and can be removed. After the synchronization is removed, simple reordering transformations could result in the behavior `r3 == 1` and `r4 = 0`.

6.7 Safety Guarantees for Incorrectly Synchronized Programs

It has been long held that one of the requirements for a memory model is that it should guarantee sequentially consistent semantics for correctly synchronized programs. However, the issue of what semantics or guarantees should be provided for incorrectly synchronized programs has not been addressed.

The Java memory model needs to provide such guarantees for two reasons:

- In Java, you should be able to load and run untrusted code inside a trusted process. It must be possible to limit the effect of malicious code, even if it is badly synchronized. Therefore, any program that passes the verifier must have defined semantics.

  This is in constrast to languages such as C and C++ that, for example, do not define the semantics of programs that write outside the bounds of an array. As a result, programs written in those languages are subject to attacks that can violate important security guarantees.

- Errors in general, and synchronization errors in particular, occur all too often in real programs. While it is preferable to avoid errors altogether, if one occurs, its effects should be as limited and localized as possible.

Although there is no complete list of the safety guarantees needed by incorrectly synchronized programs, here is an incomplete list of those provided by the Java memory model.

No out-of-thin-air reads Each read of a variable must see a value written by a write to that variable.

Type Safety Incorrectly synchronized programs are still bound by Java's type safety guarantees. This is reinforced by the fact that the programs must obey intra-thread semantics (as described in Section 5) and that there cannot be any out-of-thin-air reads.

Non-intrusive reads When incorrectly synchronized reads are added to a correctly synchronized program in a way that does not affect the behavior of the program, the program is no longer correctly synchronized. This can happen, for example, reads are added for the purposes of debugging. Such reads should be non-intrusive: the resulting program should still have sequentially consistent semantics, other than for the values seen by those reads. While this is a special case of limited applicability, it captures an important property that pushes the Java memory model in a desirable direction. Note that the semantics of final fields violate the non-intrusive reads property.

Causality This is, without a doubt, the hardest concept to understand and formalize. The Java memory model needs to forbid behaviors such as that shown in Figure 12, In this example, each thread justifies its write of 42 based on the fact that the other thread wrote 42. We say, therefore, that the value 42 appears *out of thin air*. While we prohibit this example, there are a number of other examples that seem to be violations of causality, but can, in fact, arise through combinations of standard and desirable compiler optimizations.

7 Specification of the Java Memory Model

We previously described sequential consistency (Section 3.4). It is too strict for use as the Java memory model since it forbids standard compiler and processor optimizations. We will now present a description of the Java memory model.

In sequential consistency, all actions occur in a total order (the execution order) that is consistent with program order, and each read r of a variable `v` sees the value written by the write w to `v` such that:

- w comes before r in the execution order, and
- there is no other write w' such that w comes before w' and w' comes before r in the execution order.

Initially, `A == B == 0`

Thread 1	Thread 2
`1: B = 1;`	`3: A = 2`
`2: r2 = A;`	`4: r1 = B`

May observe `r2 == 0`, `r1 == 0`

Figure 10: Behavior allowed by happens-before consistency, but not sequential consistency

7.1 Happens-Before Consistency

We retain from sequential consistency the idea that there is a total order over all actions that is consistent with the program order. Using this order, we can relax the rules by which writes can be seen by a read. We compute a partial order called the *happens-before order*, as described in Section 3.

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 is *happens-before consistent* if each read sees a write that it is allowed to see by the happens-before ordering. For example, the behavior shown in Figure 10 is happens-before consistent, since there is an execution order that allows each read to see the appropriate write. In this case, since there is no synchronization, each read can see either the write of the initial value or the write by the other thread. One such execution order is

```
1: B = 1
3: A = 2
2: r2 = A; // sees initial write of 0
4: r1 = B // sees initial write of 0
```

Similarly, the behavior shown in Figure 1 is happens-before consistent, since there is an execution order that allows each read to see the appropriate write. One such execution order is

```
1: r2 = A; // sees write of A = 2
3: r1 = B // sees write of B = 1
2: B = 1
4: A = 2
```

In this execution, the reads see writes that occur later in the execution order. This may seem counterintuitive, but is allowed by happens-before consistency. It turns out that allowing reads to see later writes can sometimes produce unacceptable behaviors.

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 11: Happens-Before Consistency is not sufficient

7.2 Causality

Happens-Before Consistency is a necessary, but not sufficient, set of constraints. Merely enforcing Happens-Before Consistency would allow for *unacceptable* behaviors – those that violate the requirements we have established for Java programs. For example, happens-before consistency allows values to appear "out of thin air".

For example, the code shown in Figure 11 is correctly synchronized. This may seem surprising, since it doesn't perform any synchronization actions. Remember, however, that a program is correctly synchronized if, when it is executed in a sequentially consistent manner, there are no data races among its non-volatile variables. 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.

Since this program is correctly synchronized, the only behaviors we can allow are sequentially consistent behaviors. However, there is an execution of this program that is happens-before consistent, but not sequentially consistent:

```
r1 = x; // sees write of x = 1
y = 1;
r2 = y; // sees write of y = 1
x = 1;
```

This result is happens-before consistent: there is no happens-before relationship that prevents it from occurring. However, it is clearly not acceptable: there is no sequentially consistent execution that would result in this behavior. The fact that we allow a read to see a write that comes later in the execution order can sometimes thus result in unacceptable behaviors.

Although allowing reads to see writes that come later in the execution order is sometimes undesirable in this model, it is also sometimes necessary. Consider Figure 1. Since the reads come first in each thread, the very first action in the execution order must be a read. If that read can't see a write that occurs later, then it can't see any value other than the initial value for the variable it reads. This is clearly not reflective of all behaviors.

We refer to the issue of when reads can see future writes as *causality*. Read operations do not actually use crystal balls or time machines to foretell the future to determine which value they can see. This issue actually arises because programs are often not actually executed in

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 12: An Unacceptable Violation of Causality

the order they are written. Compilers, JVMs and architectures often reorder operations or perform them in parallel.

Even when a program is incorrectly synchronized, there are certain behaviors that violate causality in a way that is considered unacceptable. An example of this is given in Figure 12; the behavior r1 == r2 == 42 is happens-before consistent, but unacceptable for a Java program.

Examples such as these reveal that the specification must define carefully whether a read can see a write that occurs later in the execution (bearing in mind that if a read sees a write that occurs later in the execution, it represents the fact that the write is actually performed early).

The rules that allow a read to see a later write are complicated and can be controversial, in part because they rely on judgments about acceptability that are not covered by traditional program semantics. There are, however, two uncontroversial properties that are easy to specify:

- If a write is absolutely certain to be performed in all executions, it may be seen early (thus, the behavior in Figure 1 is legal).
- If a write cannot occur unless it is seen by an earlier read, it cannot be seen by an earlier read (thus, the behavior in Figures 11 and 12 are impossible).

The Java memory model takes as input a given execution, and a program, and determines whether that execution is a legal execution of the program. It does this by gradually building a set of "committed" actions that reflect which actions were executed by the program. Usually, the next action to be committed will reflect the next action that can be performed by a sequentially consistent execution. However, as a read sometimes needs to see a later write, we allow some actions to be committed earlier than other actions that happen-before them.

Obviously, some actions may be committed early and some may not. If, for example, one of the writes in Figure 11 were committed before the read of that variable, the read could see the write, and the unacceptable result could occur.

An action a can only be committed earlier than actions that happen-before it for an execution E' under very specific circumstances. It must occur in some execution E where it was not performed early, but where all of the previously committed actions in E' occurred, and

- All of the reads that happened-before a in E must see writes that happen-before them.

- If `a` is a read, in both E and E' it sees a value that was written by some committed action.

This prevents behavior such as that of Figures 11 and 12; an action can only occur early if we can reason that it occurs based solely on the actions that happen-before it.

Finally, we could create a situation where we reason that an action `a` can occur early based on its happens-before relationships, and then commit it. However, after we commit `a`, we take a program path that does not maintain those happens-before relationships; this might void our reasoning. To avoid this problem, we require the happens-before edges that led to `a` to be present in the resulting execution.

In the remainder of this section, we formalize the memory model we have described so far.

7.3 Actions and Executions

An action a is described by a tuple $\langle t, k, v, u \rangle$, comprising:

t - the thread performing the action

k - the kind of action: volatile read, volatile write, (normal or non-volatile) read, (normal or non-volatile) write, lock or unlock. Volatile reads, volatile writes, locks and unlocks are synchronization actions.

v - the variable or monitor involved in the action

u - an arbitrary unique identifier for the action

An execution E is described by a tuple $\langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \rangle$, comprising:

P - a program

A - a set of actions

$\overset{po}{\rightarrow}$ - program order, which for each thread t, is a total order all actions performed by t in A

$\overset{so}{\rightarrow}$ - synchronization order, which is a total order over all synchronization actions in A

W - a write-seen function, which for each read r in A, gives $W(r)$, the write action seen by r in E.

V - a value-written function, which for each write w in A, gives $V(w)$, the value written by w in E.

$\overset{sw}{\rightarrow}$ - synchronizes-with, a partial order over synchronization actions.

$\overset{hb}{\rightarrow}$ - happens-before, a partial order over actions

Note that the synchronizes-with and happens-before are uniquely determined by the other components of an execution and the rules for well-formed executions.

7.4 Definitions

1. **Definition of synchronizes-with** If $x \xrightarrow{so} y$ and x is a volatile write or an unlock, and y is a volatile read of the same variable as x, or a lock of the same monitor as x, then $x \xrightarrow{sw} y$. Volatile writes and unlocks are referred to as *releases*, and volatile reads and locks are referred to as *acquires*.

2. **Definition of happens-before** The happens-before order $\xrightarrow{hb}$ is the transitive closure of $\xrightarrow{sw} \cup \xrightarrow{po}$.

3. **Restrictions of partial orders and functions** We use $f|_d$ to denote the function given by restricting the domain of f to d: for all $x \in d$, $f(x) = f|_d(x)$ and for all $x \notin d, f(x) = \bot$. Similarly, we use $\xrightarrow{e}|_d$ to represent the restriction of the partial order $\xrightarrow{e}$ to the elements in d: for all $x, y \in d$, $x \xrightarrow{e} y$ if and only if $x \xrightarrow{e}|_d y$. If either $x \notin d$ or $y \notin d$, then it is not the case that $x \xrightarrow{e}|_d y$.

7.5 Well-formed Executions

We only consider well-formed executions. An execution $E = \langle P, A, \xrightarrow{po}, \xrightarrow{so}, W, V, \xrightarrow{sw}, \xrightarrow{hb} \rangle$ is well formed if the following conditions are true:

1. **Each read sees a write in the execution. All volatile reads see volatile writes, and all non-volatile reads see non-volatile writes.** For all reads $r \in A$, we have $W(r) \in A$ and $W(r).v = r.v$. If $r.k$ is a volatile read, then $W(r).k$ is a volatile write, otherwise $r.k$ is a normal read, and $W(r).k$ is a normal write.

2. **Synchronization order is consistent with program order** There do not exist $x, y \in A$, such that $x \xrightarrow{so} y \wedge y \xrightarrow{po} x$. The transitive closure of synchronization order and program order is acyclic.

3. **The execution obeys intra-thread consistency** For each thread t, the actions performed by t in A are the same as would be generated by that thread in program-order in isolation, with each write w writing the value $V(w)$ and each read r seeing the value $V(W(r))$. The program-order must reflect the program order of P.

4. **The execution obeys happens-before consistency** For all reads $r \in A$, it is not the case that $r \xrightarrow{hb} W(r)$ or that there exists a write $w \in A$ such that $w.v = r.v$ and $W(r) \xrightarrow{hb} w \xrightarrow{hb} r$.

5. **The execution obeys synchronization-order consistency** For all volatile reads $r \in A$, it is not the case that $r \xrightarrow{so} W(r)$ or that there exists a write $w \in A$ such that $w.v = r.v$ and $W(r) \xrightarrow{so} w \xrightarrow{so} r$.

6. **The execution obeys a weak fairness property for synchronization actions.** Only a finite number of synchronization actions may occur before a given synchronization action occurs in the synchronization order.

7.6 Executions valid according to the Java Memory Model

A well-formed execution $E = \langle P, A, \overset{po}{\rightarrow}, \overset{so}{\rightarrow}, W, V, \overset{sw}{\rightarrow}, \overset{hb}{\rightarrow} \rangle$ is validated by committing actions from A. If all of the actions in A can be committed, then the execution is valid according to the Java memory model.

Starting with the empty set as C_0, we perform several steps where we take actions from the set of actions A and add them to a set of committed actions C_i to get a new set of committed actions C_{i+1}. To demonstrate that this is reasonable, for each C_i we need to demonstrate an execution E_i containing C_i that meets certain conditions.

Formally, there exists

- Sets of actions $C_0, C_1, \ldots, C_n$ such that
  - $C_0 = \emptyset$
  - $C_i \subseteq C_{i+1}$
  - $C_n = A$
- Well-formed executions $E_1, \ldots, E_n$, where $E_i = \langle P, A_i, \overset{po_i}{\rightarrow}, \overset{so_i}{\rightarrow}, W_i, V_i, \overset{sw_i}{\rightarrow}, \overset{hb_i}{\rightarrow} \rangle$.

Given these sets of actions $C_0 ... C_n$ and executions $E_1 ... E_n$, every action in C_i must be one of the actions in E_i. All actions in C_i must share the same relative happens-before order and synchronization order in both E_i and E. Formally,

1. $C_i \subseteq A_i$
2. $\overset{hb_i}{\rightarrow}|_{C_i} = \overset{hb}{\rightarrow}|_{C_i}$
3. $\overset{so_i}{\rightarrow}|_{C_i} = \overset{so}{\rightarrow}|_{C_i}$

The values written by the writes in C_i must be the same in both E_i and E. Only the reads in C_{i-1} need to see the same writes in E_i as in E. Formally,

4. $V_i|_{C_i} = V|_{C_i}$
5. $W_i|_{C_{i-1}} = W|_{C_{i-1}}$

All reads in E_i that are not in C_{i-1} must see writes that happen-before them. All reads in $C_i - C_{i-1}$ must see writes in C_{i-1} in both E_i and E. Formally,

6. For any read $r \in A_i - C_{i-1}$, we have $W_i(r) \overset{hb_i}{\rightarrow} r$
7. For any read $r \in C_i - C_{i-1}$, we have $W_i(r) \in C_{i-1}$ and $W(r) \in C_{i-1}$

Initially, `x = 0`

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

`r1 == 2` and `r2 == 1` is a legal behavior

Figure 13: An Unexpected Reordering

A set of synchronization edges is *sufficient* if it is the minimal set such that you can take the transitive closure of those edges with program order edges, and determine all of the happens-before edges in the program. This set is unique.

Given a set of sufficient synchronizes-with edges for E_i, if there is a release-acquire pair that happens-before an action you are committing, then that pair must be present in all E_j, where $j \geq i$. Formally,

8. Let $\overset{ssw_i}{\rightarrow}$ be the $\overset{sw_i}{\rightarrow}$ edges that are also in the transitive reduction of $\overset{hb_i}{\rightarrow}$ but not in $\overset{po_i}{\rightarrow}$. We call $\overset{ssw_i}{\rightarrow}$ the sufficient sychronizes-with edges for E_i. If $x \overset{ssw_i}{\rightarrow} y \overset{hb_i}{\rightarrow} z$ and $z \in C_i$, then $x \overset{sw_j}{\rightarrow} y$ for all $j \geq i$.

8 Illustrative Test Cases and Behaviors

In this section, we give a number of examples of behaviors that are either allowed or prohibited by the Java memory model. Most of these are either examples that show violations of our informal notion of causality, and thus are prohibited, or examples that seem to be a violation of causality but can result from standard compiler optimizations, and are in fact allowed.

The example in Figure 12 provides an example of a result that is clearly unacceptable. If, for example, the value that was being produced "out of thin air" was a reference to an object which the thread was not supposed to have, then such a transformation could be a serious security violation. There are no reasonable compiler transformations that produce this result.

8.1 Surprising Behaviors Allowed by the Memory Model

Figure 13 shows a small but interesting example. The behavior `r1 == 2` and `r2 == 1` is a legal behavior, although it may be difficult to see how it could occur. A compiler would not reorder the statements in each thread; this code must never result in `r1 == 1` or `r2 == 2`. However, the behavior `r1 == 2` and `r2 == 1` might be allowed by a processor architecture that performs the writes early, but in a way that they were not visible to local reads that came before them in program order. This behavior, while surprising, is a common optimization that is allowed by the Java memory model.

The behavior shown in Figure 14 is allowed. The compiler should be allowed to

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: r2 = r1;	
3: if (true) ;	

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

Figure 14: Effects of Redundant Read Elimination

Initially, `a = b = 0`

Thread 1	Thread 2
r1 = a;	r2 = b;
if (r1 == 1)	if (r2 == 1)
b = 1;	a = 1;
	else
	a = 1;

`r1 == r2 == 1` is legal behavior

Figure 15: Prescient Writes Can Be Performed Early

- eliminate the redundant read of `a`, replacing `r2 = a` 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` is always guaranteed to happen, because the reads of `a` always return the same value. Without this information, the assignment seems to cause itself to happen. Thus, simple compiler optimizations can lead to an apparent causal loop. Note that intra-thread semantics and out-of-thin-air safety guarantee that if `r1` $\neq$ `r2`, then Thread 1 will not write to `b`, `r3 == 1` and either `r1 == r2 == 0` or `r1 == r2 == 1`.

Another unusual example can be seen in Figure 15. This behavior would seem impossible, because thread 2 should not be able to decide which assignment statement it will execute until after it has read `b`. But if `r2 == 1`, then that suggests that the write `a = 1` was performed before the read `r1 = a`.

This behavior can result from a compiler detecting that in every execution *some* statement will perform `a = 1`. Thus, the action may be performed early, even though we don't know in advance which *statement* would have caused the action to occur. This may cause `r1` to see 1, `b` to be written to by Thread 1, Thread 2 to see `b == 1`, and `a` to be written to in a different place from the one it was originally.

Initially, `x == y == 0`

Thread 1	Thread 2
`r1 = x;`	`r3 = y;`
`r2 = 1 + r1*r1 - r1;`	`x = r3;`
`y = r2;`	

`r1 == r2 == r3 == 1` is legal behavior

Figure 16: Compilers Can Think Hard About When Actions Are Guaranteed to Occur

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

Thread 1	Thread 2
`r3 = x;`	`r2 = y;`
`if (r3 == 0)`	`x = r2;`
`  x = 42;`	
`r1 = x;`	
`y = r1;`	

`r1 == r2 == r3 == 42` is a legal behavior

Figure 17: A Complicated Inferrence

Figure 16 shows another seemingly impossible behavior. In order for `r1 == r2 == 1`, Thread 1 would seemingly need to write 1 to `y` before reading `x`. However, it appears as if Thread 1 can't know what value `r2` will be until after `x` is read.

In fact, it is easy for the compiler to perform an analysis that shows that `x` is guaranteed to be either 0 or 1. The write to `y` is therefore not dependent on the value seen for `x`. Knowing that, the compiler can determine that the quadratic equation always returns 1, resulting in Thread 1's always writing 1 to `y`. Thread 1 may, therefore, write 1 to `y` before reading `x`.

Now consider the code in Figure 17. A compiler could determine that the only values ever assigned to `x` are 0 and 42. From that, the compiler could deduce that, at the point where we execute `r1 = x`, either we had just performed a write of 42 to `x`, or we had just read `x` and seen the value 42. In either case, it would be legal for a read of `x` to see the value 42. It could then change `r1 = x` to `r1 = 42`; this would allow `y = r1` to be transformed to `y = 42` and performed earlier, resulting in the behavior in question.

8.2 Behaviors Prohibited by the Memory Model

The examples in Figures 18 and 19 are similar to the example in Figure 12, with one major distinction. In Figure 12, the value 42 could never be written to `x` in any sequentially consistent execution. In the examples in Figures 18 and 19, 42 is only sometimes written to `x`. Could it be legal for the reads in Threads 1 and 2 to see the value 42 even if Thread 4 does not write that value?

Note that the major difference between these two examples is the fact that in Figure 18, the writes in Threads 1 and 2 are dependent on data flow from the reads, and in Figure 19,

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

Thread 1	Thread 2	Thread 3	Thread 4
`r1 = x;`	`r2 = y;`	`z = 42;`	`r0 = z;`
`y = r1;`	`x = r2;`		`x = r0;`

Is `r0 == 0, r1 == r2 == 42` legal behavior?

Figure 18: Can Threads 1 and 2 See 42, if Thread 4 didn't write 42?

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

Thread 1	Thread 2	Thread 3	Thread 4
`r1 = x;`	`r2 = y;`	`z = 1;`	`r0 = z;`
`if (r1 != 0)`	`if (r2 != 0)`		`if (r0 == 1)`
`  y = r1;`	`  x = r2;`		`  x = 42;`

Is `r0 == 0, r1 == r2 == 42` legal behavior?

Figure 19: Can Threads 1 and 2 See 42, if Thread 4 didn't write to `x`?

the writes are dependent on control flow from the reads. For these purposes, the Java memory model should not make a distinction between control and data dependence.

This is a potential security issue; if 42 represents a reference to an object that Thread 4 controls, but does not want Threads 1 and 2 to see without Thread 4's first seeing 1 for `z`, then Threads 1 and 2 can be said to manufacture the reference out of thin air.

This sort of behavior is not known to result from any combination of known reasonable and desirable optimizations. However, there is also some question as to whether this reflects a real and serious security requirement. In Java, the semantics usually side with the principle of having safe, simple and unsurprising semantics when possible, and thus the Java Memory Model prohibits the behaviors shown in Figures 18 and 19.

9 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 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.

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.

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, where Y refers to any object which may transitively have a reference to X.

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 $F \stackrel{hb}{\rightarrow} a$, 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 20. An execution of this code is shown in Figure 21, with the reference links shown and labeled. Two reference link chains are shown. In order for the read of `t.f` to be correctly ordered with respect to the construction of the object referenced by `t`, there must exist some action on either chain that is forced by synchronization to occur after construction of that object. For this execution, since both of the assignments in Thread 1 happen after construction, the read of `t.f` is guaranteed to see the correctly initialized value.

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 $F \stackrel{hb}{\rightarrow} a$ no matter which read of Y is selected.

An example of this situation can be seen in Figure 22. 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

Thread 1	Thread 2	Thread 3
`Foo p = new Foo();`	`Foo q = G.x;`	`Bar t = G.y;`
`Bar b = new Bar()`	`Bar r = q.b;`	`int i = t.f;`
`p.b = b;`	`G.y = r;`	
`G.x = p;`		

Figure 20: When is Thread 3 guaranteed to see the correct value for final field `b.f`?

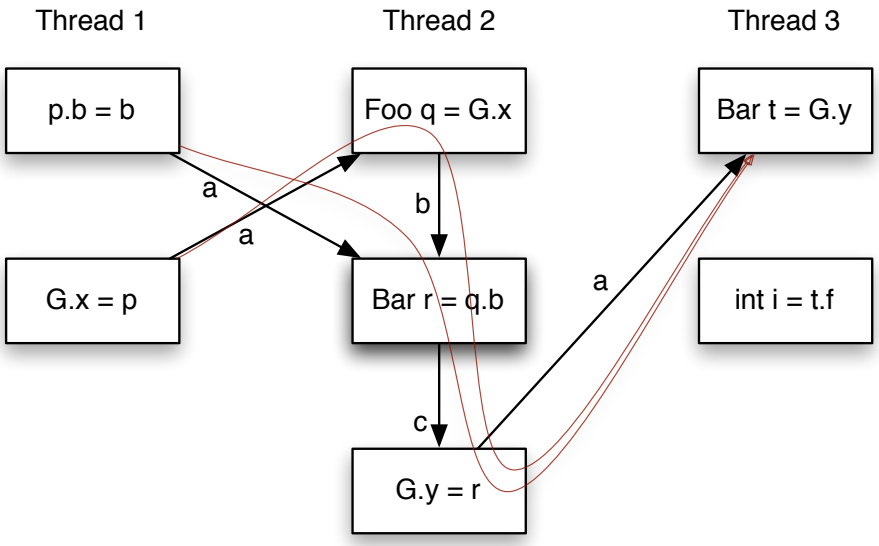


Figure 21: Reference links in an execution of Figure 20

`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 22: Final field example where Reference to object is read twice

through the global variable `b`, there is no action that is ordered after the freeze operation, 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.

In general, if a read R of a final field f in thread t_2 is correctly ordered with respect to a freeze 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 $w \stackrel{hb}{\rightarrow} F$.

A final field may only be written by bytecode once. A classfile may be rejected in any one of the following situations:

- it contains a blank final class variable that is not definitely assigned by a static initializer of the class in which it is declared, or
- it contains a blank final instance variable which is not definitely assigned by some constructor of the class in which it is declared, or
- it contains a final variable that is assigned to even if it is not definitely unassigned immediately prior to the assignment

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,

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 23: Sets used in the formalization of final fields

most of the guarantees for final fields of that object can go kerflooey. For details, consult the formal semantics.

9.1 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 B. Figure 23 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 any other writes to that final field, 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.

9.2 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.

```
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 24: Bytes must not be overwritten by writes to adjacent bytes

10 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 24 shows a test case to detect word tearing.

Thread 1	Thread 2

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

Figure 25: Fairness

11 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 programs correctly to avoid this.

12 Fairness

Without a *fairness* guarantee for virtual machines, it is possible for a thread that is capable of making progress never to 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 25. 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.

The Java threading model only provides an extremely weak fairness guarantee. For each synchronization action, only a finite number of synchronization actions occur before it in the synchronization order.

Any fairness guarantee stronger than this 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 might, in fact, never allow Thread 2 to execute. This behavior

would be legal (but perhaps undesirable) in the Java memory model.

However, the weak fairness guarantee does guarantee that if thread 2 terminates, that thread 1 cannot loop an infinite number of times.

13 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.

13.1 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.

Each thread must determine an order over the events that could cause it to be removed from a wait set. That order does not have to be consistent with other orderings, but the thread must behave as though those events occurred in that order. For example, if a thread t is in the wait set for m, and then both an interrupt of t and a notification of m occur, there must be an order over these events.

If the interrupt is deemed to have occurred first, then t will eventually return from `wait` by throwing InterruptedException, and some other thread in the wait set for m (if any exist at the time of the notification) must receive the notification. If the notification is deemed to have occurred first, then t will eventually return normally from `wait` with an interrupt still pending.

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 is allowed either to set t's interruption status to false and have the wait method throw InterruptedException, or return normally from the method.

13.2 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.

- 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.

13.3 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()`.

13.4 Interactions of Waits, Notification and Interruption

The above specifications allow us to determine several properties having to do with the interaction of waits, notification and interruption. If a thread is both notified and interrupted while waiting, it may either:

- return normally from `wait()`, while still having a pending interrupt (in other works, a call to `Thread.interrupted()` would return true)
- return from `wait()` by throwing an `InterruptedException`

The thread may not reset its interrupt status and return normally from the call to `wait()`.

Similarly, notifications cannot be lost due to interrupts. Assume that a set S of threads is in the wait set of a monitor M, and another thread performs a `notify()` on M. Then either

- at least one thread in S must return normally from `wait()`. By "return normally", we mean it must return without throwing `InterruptedException`, or
- all of the threads in S must exit `wait()` by throwing `InterruptedException`

Note that if a thread is both interrupted and woken via `notify()`, and that thread returns from `wait()` by throwing an `InterruptedException`, then some other thread in the wait set must be notified.

13.5 Sleep and Yield

Causes the currently executing thread to sleep (temporarily cease execution) for the specified duration, subject to the precision and accuracy of system timers and schedulers. The thread does not lose ownership of any monitors, and resumption of execution will depend on scheduling and the availability of processors on which to execute the thread.

A sleep for a period of zero time and yield operations need not have obversable effects.

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 thread 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 forbidden. Instead, the semantics only describe the allowed and forbidden behaviors.

The compiler, VM and processor may compile, transform and execute a program in any way 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

B Formal Definition of 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.

B.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:

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

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

B.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 B.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.

B.2.1 Freezes Seen as a Result of Reads

Consider a read r in thread t of field x of the object o 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), .

B.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:

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

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.

B.3 Single Threaded Guarantees for Final Fields

We have discussed in detail what guarantees are made for final fields seen in multiple threads. However, compiler transformations can cause a read of a final field to return a default value even if it is only accessed in a single thread. In this section, we discuss what guarantees are made if a final field is seen to change in a single thread.

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.

C 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 finalizer invocation 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 C.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.

C.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.