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

May return r2 == 2, r1 == 1

Figure 1: Behaves Surprisingly

This document is intended to resolve all of the issues that have been discussed about Java's memory model that do not fall into the broad categories of the descriptions of semantics for final, normal and volatile fields. Discussion of immutability is confined to final fields, and so will not be addressed here.

Part I
Synchronization Idioms - Good and Bad

1 Simple Reordering

1.1 Read After Write

Initially, A == B == 0

We will start off with a simple example. Consider Figure 1. It may appear that the result $r2 == 2, r1 == 1$ is impossible. Intuitively, if $r2$ is 2, then instruction 4 came before instruction 1. Further, if $r1$ is 1, then instruction 2 came before instruction 3. So, if $r2 == 2$ and $r1 == 1$, then instruction 4 came before instruction 1, which comes before instruction 2, which came before instruction 3, which comes before instruction 4. This is, on the face of it, absurd.

However, compilers don't care about what you think might be absurd. They are perfectly happy to reorder the instructions in each thread. If instruction 3 does not come before instruction 4, and instruction 1 does not come before instruction 2, then the result $r2 == 2$ and $r1 == 1$ is perfectly

reasonable.

This may seem counter-intuitive. However, it should be noted that this code is improperly synchronized: there is no ordering of the accesses by synchronization. When synchronization is missing, weird and bizarre results are allowed.

Finally, it should be mentioned that $r1$ cannot have any value other than 0 or 1, and $r2$ cannot have any value other than 0 or 2. Values for most variables cannot appear as if out of thin air.

1.2 Object Publication and Construction

Consider Figure 2. A programmer unfamiliar with the Java memory model has written some code. That programmer is trying to guarantee that Thread 2 sees *value* for $r1.field$. However, the code is flawed. In the first example, it would be perfectly reasonable for a compiler to reorder the write to $Global.a$ above the write to $f.field$. Thread 2 doesn't have the guarantee that $f.field$ has the right value - it can see $Global.a$ early.

The second, third and fourth examples are a little more clever, but still wrong. The programmer is trying to prevent the reordering by placing synchronization in the thread. However, a little knowledge can be a dangerous thing: a lock that has no effect in another thread can be removed. Then, the write to $Global.a$ can be moved early.

A programmer can avoid these synchronization pitfalls by declaring $Global.a$ to be `volatile` - writes to and reads from volatile variables cannot be reordered, so the programmer gets the desired semantics. Alternatively, the code may be synchronized so that the writes to $Global.a$ are in a synchronized block; this will provide the same guarantees.

For most of the guarantees needed for object initialization, it suffices to use final fields. See the documents on final fields for more information on this topic.

2 Volatile

2.1 Simple Volatile Example

When all of the declared fields are normal, as they are in Figure 1, there is a great deal of reordering that can take place. Figure 3 is fairly similar: if none

Thread 1	Thread 2
// Initializer Code	r1 = Global.a;
Foo f = new Foo();	if (r1 != null)
f.field = value;	b1 = r1.field;
Global.a = f;	

Thread 1	Thread 2
// Initializer Code	r1 = Global.a;
Foo f = new Foo();	if (r1 != null)
f.field = value;	b1 = r1.field;
lock f;	
Global.a = f;	
unlock f;	

Thread 1	Thread 2
// Initializer Code	r1 = Global.a;
Foo f = new Foo();	if (r1 != null)
f.field = value;	b1 = r1.field;
lock f;	
unlock f;	
Global.a = f;	

Thread 1	Thread 2
// Initializer Code	r1 = Global.a;
Foo f = new Foo();	if (r1 != null)
lock f;	b1 = r1.field;
f.field = value;	
unlock f;	
Global.a = f;	

None of these code examples guarantees $b1 == value$.

Figure 2: Code that **does not work**

B is `volatile`

Initially, $A == B == 0$.

Thread 1	Thread 2
A = 1;	y = B;
B = 1;	x = A;

If $y == 1$, then $x == 1$.

Figure 3: Reorderings are Prevented with Volatiles

Initially all variables are 0. $v1, v2$ are volatile.

Thread 1	Thread 2
A = 1;	while (B != 1);
v1 = 1;	r2 = v2;
B = 1;	r3 = A;

Figure 4: $r3 == 1$ is **not** guaranteed

of the variables were declared volatile, many reorderings would be possible. For example, the reads of B and A in Thread 2 could be reversed; this would allow y to be 1 and x to be 0.

However, B was declared volatile. What does this mean? Writes cannot be moved past a write to a volatile, and reads cannot be moved before a read of a volatile. We cannot change the order of the instructions in Thread 1 or Thread 2. This gives us additional guarantees about our code: if $B == 1$, then $A == 1$.

This makes a change in the semantics of volatile from the original specification for volatiles. Regardless of this, code written to the original specification should still work.

It should also be mentioned that there is no guarantee made about thread communication on different volatile variables. In Figure 4, Thread 1 writes to shared variable A, writes to a volatile, and then writes to shared variable B. Thread 2 reads the value written by Thread 1 to B, then reads a volatile, and finally reads A.

Thread 2 is never guaranteed to see the write to B. For this example, however, we will assume that the value 1 is returned from B at some point.

Even though Thread 2 sees the write of 1 to B, it is not guaranteed to see the write of 1 to A. It may seem that the write of 1 to A occurred before the write of 1 to B; this ordering would seem to be enforced by the write to

Initially A=B=C=D=0

Thread 1	Thread 2	Thread 3	Thread 4
A = 1;	A = 2;	while (B != 1);	while (B != 1);
B = 1;	C = 1;	while (C != 1);	while (C != 1);
		r1 = A;	r2 = A;

Figure 5: Can this result in $r1 == 1$, $r2 == 2$?

volatile variable $v1$. In thread 2, it may seem that the read of 1 from B must occur before the read of A; this ordering would seem to be enforced by the read of volatile variable $v2$.

However, interthread orderings of this sort can only be guaranteed by reads and writes of the *same* volatile variable. In other words, for this idiom to work (still assuming that B sees the value 1), Thread 1 must write and Thread 2 must read the same variable; further, Thread 2's read of the volatile variable must return the value written in Thread 1.

2.1.1 Write Atomicity

One question that arises when dealing with volatile variables is whether writes to a volatile appear in the same order to every thread. If they do, we refer to the writes as being *atomic*. Consider Figure 5. If Thread 3 and Thread 4 are forced to see the writes to A from Threads 1 and 2 in the same order, then all executions will result in $r1 == r2$. If they can be seen out of order, then $r1 != r2$ is allowed.

Write atomicity is, in fact, a property of volatile variables in Java.

2.1.2 Multithreaded Singleton

This code is very similar to the Multithreaded Singleton pattern, also known as "double checked locking" (Figure 6). Here, the the programmer has tried to initialize the *helper* field lazily.

If the *helper* field is not volatile, the assignment to *helper* could occur before the fields of that object are initialized. Another thread could read *helper* without synchronization and try to access its fields before they were initialized. This is clearly not what the programmer intended. Use of a `volatile` field here would prevent this reordering.

Fundamentally, the question for most programmers will be, "when can I guarantee that the objects I create will be threadsafe without synchroniza-

```
class Foo {
  private [volatile] Helper helper = null;
  public Helper getHelper() {
    if (helper == null)
      synchronized(this) {
        if (helper == null)
          helper = new Helper();
      }
    return helper;
  }
  // other functions and members...
}
```

Figure 6: Multithreaded Singleton - `helper` should be `volatile`

tion?". The answer is simply that the only way to do this is to make the fields of those objects immutable using final fields, and to use final fields correctly. See other documents on final fields for details on this.

2.2 More Uses for Volatile

2.2.1 Detect Termination

In Figure 7, a compiler could reasonably detect that Thread 2 never changes the *terminateProgram* variable; it could therefore change the loop into an infinite loop. However, the restrictions imposed on volatile variables would prevent this from happening: the loop would have to check whether *terminateProgram* was updated on each iteration. If this is an idiom you use, remember to make *terminateProgram* volatile.

2.2.2 Communication on Other Fields

This category is something of a catchall: `volatile` is, in general, used for interthread communication.

Consider Figure 8. We have already seen how the read of the volatile p will guarantee that a read of p's fields will return an up to date value. But what does it guarantee for a read of those fields under another name? A

```
Thread 1:

if (command.equals("quit")) {
  terminateProgram = true;
  return;
}

Thread 2:

while (!terminateProgram) {
  try { Thread.sleep(100); }
  catch (Exception e) {}
  // update state
  repaint();
}
```

Figure 7: Example Requires `volatile` terminateProgram

p is `volatile`

Thread 1	Thread 2
Node r1 = new Node();	Node r2 = p;
q = r1;	if (r2 != null) {
r1.x = o;	int r3 = q.x;
o.y = 1;	int r4 = r3.y;
p = r1;	}

$r3$ should not be null.

Figure 8: More Uses for Volatile

```
boolean flag_i, flag_j;
int turn;

do {
  // acquire lock
  flag_i = true;
  while (flag_j) {
    if (turn == j) {
      flag_i = false;
      while (turn == j);
      flag_i = true;
    }
  }

  // critical section

  // release lock
  turn = j;
  flag_i = false;
} while (1);
```

Figure 9: Dekker's Algorithm for Mutual Exclusion

read of a volatile guarantees that **all** writes performed before that volatile was written will be seen: $r3$ will be a pointer to o.

In addition to this, $o.y$ was written before the write to p; by the same logic, Thread 2 is ensured that it will see the value 1 for $r3.y$.

2.2.3 Dekker's Algorithm

The code is Figure 9 is usually referred to as *Dekker's Algorithm*. It was the first provably correct code that implemented mutual exclusion.

The version shown here works for two threads. There are three shared variables, the $flag$ variables and the *turn* variable. The thread displayed here is thread i, and the other thread is thread j. In the other thread, i and j would be reversed.

To acquire the lock, a thread first sets its flag and then checks the other thread's flag. If that flag is set, the thread then checks the *turn* variable, which indicates whether it is that thread's turn to acquire the variable or not. If it is the thread's turn, it acquires the lock, and if it isn't, it sets the flag to false, and spins until it is. To release the lock, it is only necessary to set the *turn* variable so that it is the other thread's turn, and set your own $flag$ variable to false.

It would be nice to be able to implement this in Java. However, because there is no explicit synchronization, there is no guarantee that the other thread will see the updates made to the *turn* and $flag$ variables. For example, the compiler can analyze the `do` loop and determine that after the second iteration, *turn* will have the same value as j. Since that value is not changed in the `while` loop, the test can be removed; this will make the `while` loop infinite.

This sort of behavior can be avoided with the application of the `volatile` modifier. Setting both *turn* and $flag$ to `volatile` will force their values to be reloaded at every access. This will prevent the while loop from being transformed into an infinite loop.

A more subtle problem involves the relationship between two of the volatile variables. If the write to $flag_i$ is reordered with the read from $flag_j$, then both threads could enter their critical section without correctly seeing if the other has set their flag variable. This would violate the necessary mutual exclusion properties.

Setting the $flag$ variables to `volatile` prevents this. There is now a happens-before ordering between the write to $flag_i$ and the read of $flag_j$; in the other thread, it will be an ordering between the write to $flag_j$ and the read of $flag_i$.

What does this mean? Well, for *both* threads to enter their critical section at the same time, they both have to decide that the other thread's flag variable is false. Therefore, both of the reads have to happen before both of the writes. The read of $flag_j$, for example, must happen before the write to $flag_j$ in the other thread. In turn, the write to $flag_j$ happens before the read of $flag_i$, as we have established. The read of $flag_i$ must happen before the write to $flag_i$ in the other thread. And finally, the write to $flag_i$ must happen before the read of $flag_j$. We have now set up a cycle of happens-before relationships, which is illegal under the semantics. ***DRAW A PICTURE***

Since this reordering cannot take place, it cannot enable both threads to

Initially, $v = a = b = local = 0$. v is volatile.

Thread 1	Thread 2
a = x;	t1 = v;
b = x;	x = a;
v = local;	y = b;
// other	r = a*b;
v = local + 1;	t2 = v;
a = y;	if (t1 == t2 && t1 % 2 == 1)
b = y;	return r;
v = local + 2;	

Thread 2 can return values other than x^2 or y^2

Figure 10: Optimistic Reader Example

enter their critical section simultaneously. This is not a correctness proof; that is left as an exercise for the reader.

2.3 Non-Uses of Volatile

2.3.1 Optimistic Readers

The author of the code in Figure 10 wishes Thread 2 to see either the first set of writes to a and b by Thread 1 (i.e., the writes of x) or the second (y), but does not want a mixture. This is an example of an implementation of optimistic locking. However, the use of `volatile` here does not guarantee this; the coder should have used explicit synchronization.

Let us examine this more carefully. The guarantee that volatile gives us is that a thread that reads the volatile variable will see writes performed by another thread before that volatile was written. However, it does not explicitly guarantee that a reader thread will not see writes that occur after the write to the volatile.

As a result of this, the second write to (for example) a can be moved to before the first write to v. This might result in Thread 2 seeing the x for a, but y for b: Thread 2 returns xy instead of x^2 or y^2.

One way of making this idiom work would be to add a volatile boolean field $flag$. After the second increment of v in Thread 1, $flag$ would get read. Since the reads of y cannot be moved above a volatile read, this would ensure

```
public class ThreadStart extends Thread {
  static int x;
  static int y;
  public void run() {
    int r1 = x;
    int r2 = y;
  }
  public static void main(String args[]) {
    x = 1;
    ThreadStart t = new ThreadStart();
    y = 1;
    t.start();
  }
}
```

$r1$ and $r2$ must **both** see 1

Figure 11: What do Initialized Threads See?

that the (dependent) writes to a and b also not be moved above the volatile read.

In addition to this, before Thread 2 checks v for the second time, $flag$ must be assigned the value false. Without this assignment, the second read of v could be moved to before the reads of a and b. These changes enable the optimistic readers idiom in Java.

3 Thread Initialization and Termination

In the code in Figure 11, the newly started thread must be able to see all of the writes performed before the thread was started. There is an implicit release when `main()` performs the `start()`, and an implicit acquire when Thread t invokes its `run()` method.

There are other actions on threads as well, but most of these are defined in terms of locking actions, and therefore do not need explicit memory semantics. For example, `Thread.join()` is defined in terms of `Object.wait()`,

a is an array of length ¿ 2
Initially, a[0] = 1, a[1] = 3, a[2] = 4
`synchronized(a)` is not removed

Thread 1	Thread 2
a[2] = 5;	r1 = a[0];
synchronized(a) ...	r2 = a[r1];
a[0] = 2;	

Figure 12: Code that **does not work**

and therefore inherits the semantics of its locking and unlocking behavior.

4 Other Reorderings

4.0.2 Hardware Memory Models

Now consider a similar reordering, on the code shown in Figure 12. In this code, for some reason, a compiler has decided that it cannot remove the synchronization block. This enforces an ordering between the writes to $a[2]$ and $a[0]$. This would seem to prevent $r2$ from seeing $a[2]$ unless $a[2]$ is set to 5.

However, this does not work either. Processors with weak memory guarantees (such as the Alpha and IA-64) allow $a[2]$ to be cached even if $a[0]$ is not. The "new" value of $a[0]$ might be seen even if the "old" value is seen for $a[2]$.

Our new semantics reflect this. As Thread 2 does not perform any synchronization on a, no ordering is enforced between Thread 1 and Thread 2. Thread 2 is free to read 4 for $r2$.

4.1 Useless Synchronization

In Figure 13, we have some examples of synchronization that does not have to have any effect. Synchronizing on a new object, as we do in the first example, may have no effect because no other thread can obtain this lock: no orderings are enforced with other threads.

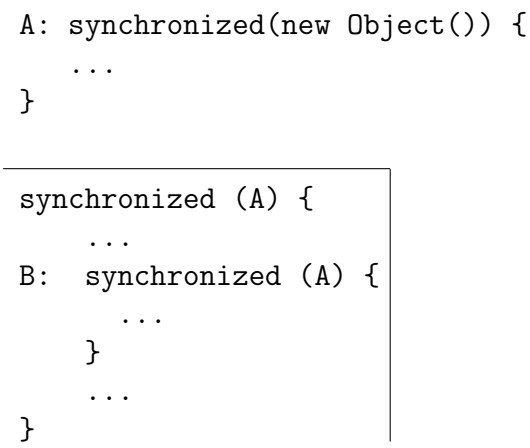


Figure 13: The labelled statements have no effect

```
synchronized(A) {
  ...
}
synchronized(A) {
  ...
}
```

Figure 14: The locks may be merged

In the second example, we obtain the same lock twice. This, too, may have no effect: no other thread can have done anything that forces an ordering with that lock while that lock is held.

4.2 Lock Merging

In Figure 14, the compiler may merge the lock regions, but a correctly synchronized program should not display the effects of this. Any other thread that obtains a lock on A will simply have to wait until the second locking region has been exited (as it might have had to do anyway).

Thread 1	Thread 2
synchronized(global1) {	synchronized(global1) {
thread2.start();	}
S = 1;	r1 = S;
}	

$r1$ is guaranteed to see 1

Figure 15: This synchronization is not useless

Initially, A == B == 0

Thread 1	Thread 2
synchronized(X) {	synchronized(X) {
r2 = A;	r1 = B;
B = 1;	A = 2;
}	}

Must not return 2 for A, 1 for B

Figure 16: Behaves Just as Expected

Empty Synchronization Blocks The reader should not be fooled into thinking that any empty synchronization block can be removed. In Figure 15, we see an example of where empty synchronization can be quite useful. In Thread 2, the empty synchronization block acts as a way of guaranteeing that the second thread will see the write to S performed by the first. This is a legal idiom under the memory model.

4.3 Correctly Synchronized Programs

4.3.1 No Surprising Results

Compare Figure 16 to Figure 1. Here, the proper synchronization protocols are obeyed: the code in each thread will be executed without interference from the other. Here, it would be impossible to get the result $r1 == 1, r2 == 2$.

Part II
VM Safety Guarantees

5 Type Safety

One issue that needs to be addressed by the VM is that of seeing the correct default value for a field. If a thread attempts to read a field, and it has a garbage value, then that VM is violating the semantics' guarantees.

To prevent this, fields of an object must be pre-zeroed: this must occur early enough that every thread that sees that object is guaranteed to see the zeroed value. In most cases, this will mean setting free memory to zero during garbage collection.

5.1 Word Tearing

Another consideration, related to type safety, is that values must not appear to come out of thin air. In Figure 5.1, multiple threads are writing to adjascent bytes in a byte array. Some processors (notably early Alphas) do not provide the ability to write to a single byte; writing to a byte naïvely would overwrite a full word. This issue must not arise in Java; a write to a single byte should write to nothing other than that byte.

6 Safety for Internal Data Structures

6.1 Safety for Object Headers

One issue to address is is that of synchronization errors and data races when accessing constructs that are not modifiable for the programmer; this includes, for example, object headers and vtables. From the formal perspective, such structures can be considered to have been written once, before each thread started. This constructs a happens-before ordering between all writes to these structures and any reads.

The upshot of this is that a VM cannot have a synchronization error that involves (say) a vtable: the proper value for a vtable must always be visible to every thread.

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

There is a simple way to accomplish this. As we have discussed, all free memory will be preallocated to contain null values. This will prevent the VM from reading a garbage value for any vtable pointer. The VM should check to make sure that a vtable pointer is not null; if it is, it should perform a memory barrier operation and reload the pointer. Since a vtable pointer can never be null, enough repetitions of this should load the appropriate values eventually. The code to do the memory barrier and reload the vtable would rarely, if ever, be taken.

6.2 Safety for Arrays

The main issues for array safety is that of the array length; if access to the array is not synchronized, then an uninitialized value could be seen for the array length. This problem can be resolved by considering that an array length is a field of an array object. If this field is final, then other threads are guaranteed to see its correctly constructed value.

7 Compiler Analyses

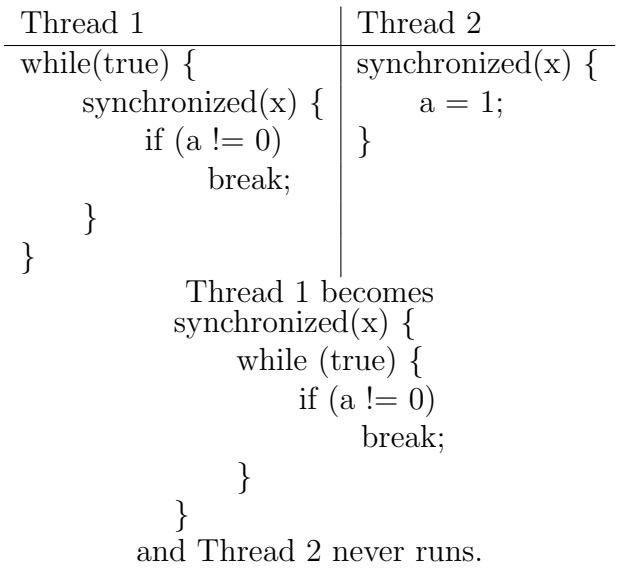
8 Escape Analysis

The optimizations that typically result from escape analysis should all be legal in a way that is simple to understand. For example, escape analysis is often used to determine which locks are thread-local, and remove them; this is legal if a thread local lock has no effect on the actions of another thread.

Since synchronization actions on a variable v in an execution only create an ordering with other synchronization actions on v, we can conclude that if v is thread local, no other thread will be able to order itself with respect to synchronization actions on v. This enables thread-local lock elimination.

9 Lock Merging

As with lock elimination (see Section 8 above), merging adjacent locks on the same monitor is a useful optimization. Again, the issue is that a synchronization operation can only be removed if no other thread can see the result of its being removed. If two lock regions are merged, then the resulting



execution of the program is equivalent to the execution where the lock was not acquired by another thread between the two original locking regions.

This means that a compiler could conceivably create a situation where one thread *always* holds a given lock, thereby starving the other threads. Thread 1 of Figure 9 can be transformed by hoisting the synchronization out of the loop. This would result in a situation where the code in Thread 2 never occurred. Although this code transformation may reflect poorly on the quality of a Java implementation, it is still legal.

10 Fairness

As can be inferred from Section 9, the memory model does not provide any fairness guarantees. Threads that may seem as if they can make progress may, in fact, never make any progress at all. This allows for an implementation where thread switching only occurs cooperatively, at invocations of `Thread.yield()`.

This does enable some code transformations, such as the one in Figure 9, that are legal, but reflect a questionable JVM design decision. Another example of where a legal transformation might be questionable can be seen in Figure 10.

volatile boolean consumed = true;

Thread 1	Thread 2
while (true) {	while (true) {
synchronized(this) {	synchronized(this) {
work();	if (!consumed) {
consumed = false;	consumeWork();
}	consumed = true;
while (!consumed);	}
}	}
	}

What would happen if the loop that checks the status of the volatile *consumed* flag in Thread 1 were to be moved inside the `synchronized` block? This would cause deadlock: Thread 2 could never set *consumed* to true, and so stop the loop.

Since the model does not guarantee that every thread will make progress, this transformation is legal. The "correct" way to program this idiom is to use the built-in `wait()` and `notify()` methods. Regardless of this, it is unlikely that such a transformation will take place.