The Java Memory Model

Authors: Jeremy Manson, William Pugh, Sarita V. Adve

Presenter: Keren Zhou

COMP 522

Why Java needs a well-formed memory model

- Java supports threads running on shared memory
- Java memory model defines multi-threaded Java program semantics
- Key concerns: Java memory model specifies legal behaviors and provides safety and security properties

Why should we care about Java Memory Model

- A programmer should know
  - Junior level
    - Use *monitor*, *locks*, and *volatile* properly
    - Learn safety guarantees of Java
  - Intermediate level
    - Reason the correctness of concurrent programs
    - Use concurrent data structures (e.g ConcurrentHashMap)
  - Expert level
    - Understand and optimize utilities in java.util.concurrent
      - AbstractExecutorService
      - Atomic variables

Create a singleton to get a static instance

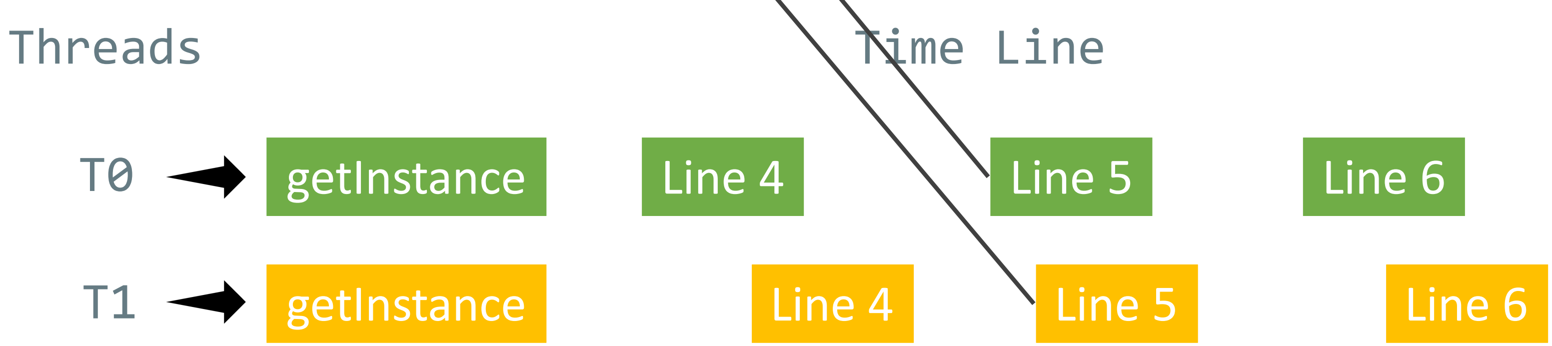
- Singleton: only want a single instance in a program
  - Database
  - Logging
  - Configuration

```java
public class Instance {
  private static Instance instance;
  public static Instance getInstance() {
    if (instance == null)
      instance = new Instance();
    return instance;
  }
}
```

The simple singleton is thread-unsafe

```java
1 public class Instance {
2   private static Instance instance;
3   public static Instance getInstance() {
4     if (instance == null)
5       instance = new Instance();
6     return instance;
7   }
8 }
```

Create two instances at the same time!

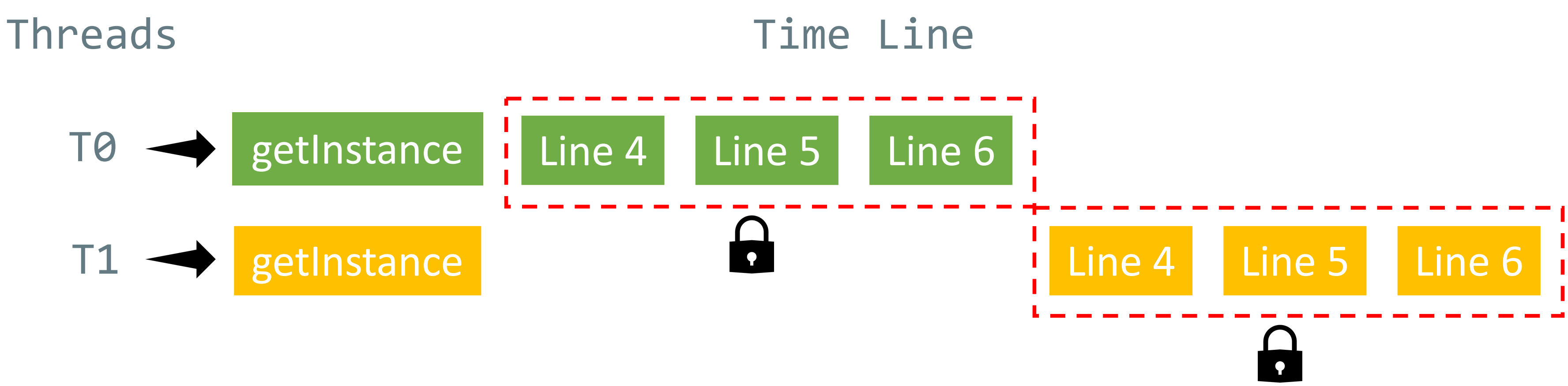


Use synchronized keyword

- Java *synchronized* keyword can be used in different contexts
  - Instance methods
  - Code blocks
  - Static methods
    - Only **one thread** can execute inside a static synchronized method per class, irrespective of the number of instances it has.

The synchronized singleton has low performance

```
1 public class Instance {
2   private static Instance instance;
3   public synchronized static Instance getInstance() {
4     if (instance == null)
5       instance = new Instance();
6     return instance;
7   }
8 }
```



Use double-checked lock to make it more efficient

- Motivation: In the early days, the cost of synchronization could be quite high
- Idea: Avoid the costly synchronization for all invocations of the method except the first
- Solution:
  - First check if the instance is null or not
  - If instance is null, enter a critical section to create the object
  - If instance is not null, return instance

Double checked-lock implementation

```
public class Instance {
  private static Instance instance;
  public static Instance getInstance() {
    if (instance == null) {
      synchronized (Instance.class) {
        if (instance == null) {
            instance = new Instance();
        }
      }
    }
    return instance;
  }
}
```

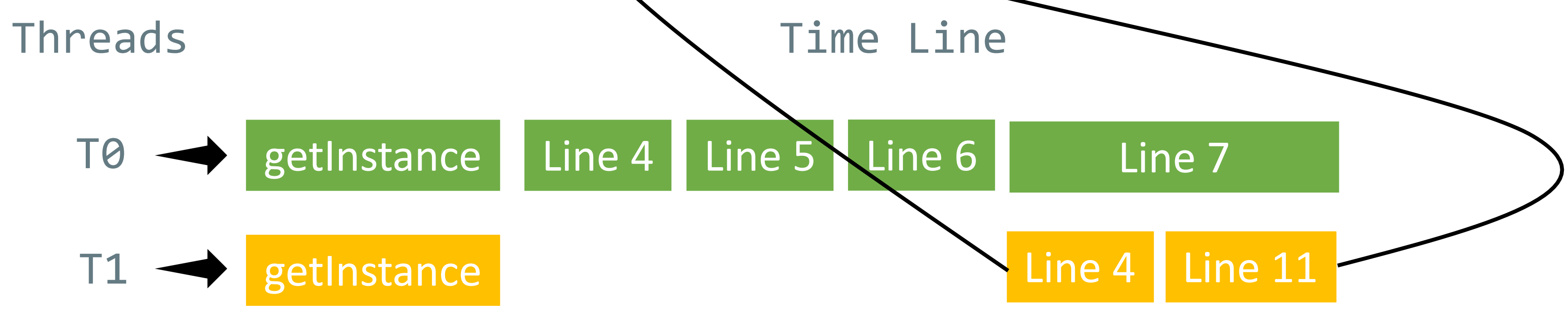
How double-checked lock goes wrong

- Brief answer: instruction reorder
- Suppose T0 is initializing with the following three steps
  1) mem = allocate(); //Allocate memory for Singleton
  2) ctorSingleton(mem); //Invoke constructor
  3) object.instance = mem; //initialize instance.
- What if step 2 is interchanged with step 3?
  - Another thread T1 might see the instance before being fully constructed

A thread returns an object that has not been constructed

```
3   public static Instance getInstance() {
4     if (instance == null) {
5       synchronized (UnsafeInstance.class) {
6         if (instance == null) {
7             instance = new Instance();
8         }
9       }
10    }
11    return instance;
12  }
```

```
mem = allocate();
object.instance = mem;
ctorSingleton(mem);
```



T1 returns before constructor completes!

Use volatile to avoid reordering

- The behavior of *volatile* differs significantly between programming languages
- C/C++
  - Volatile keyword means always read the value of the variable memory
  - Operations on volatile variables are not atomic
  - Cannot be used as a portable synchronization mechanism
- Java
  - Prevent reordering
  - Derive a synchronization order on top of Java Memory Model

Use volatile to avoid reordering

- Java’s *volatile* was not consistent with developers intuitions
  - The original Java memory model allowed for volatile writes to be reordered with nonvolatile reads and writes
- Under the new Java memory model (from JVM v1.5), volatile can be used to fix the problems with double-checked locking

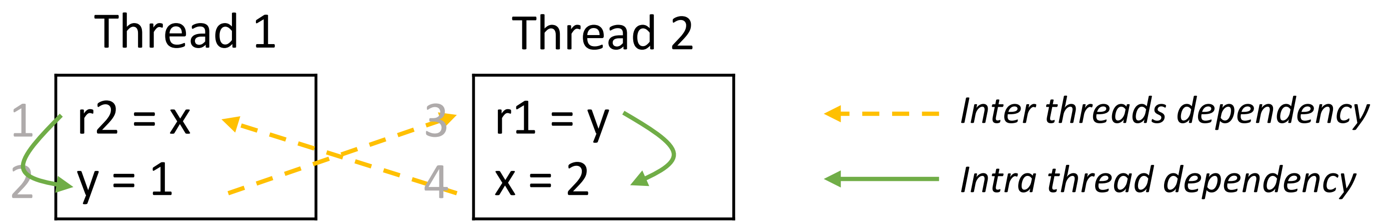
Java memory model history

- **1996**: An Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects, *Douglas C. Schmidt* and etc.
- **1996**: The Java Language Specification, chapter 17, *James Gosling* and etc.
- **1999**: Fixing the Java Memory Model, *William Pugh*
- **2004**: JSR 133---Java Memory Model and Thread Specification Revision

Review: sequential consistency

- **Total order**: Memory actions must appear to execute one at a time in a single total order
- **Program order**: Actions of a given thread must appear in the same order in which they appear in the program

Review: sequential consistency violation



Circular dependency!

Initial conditions: x = y = 0

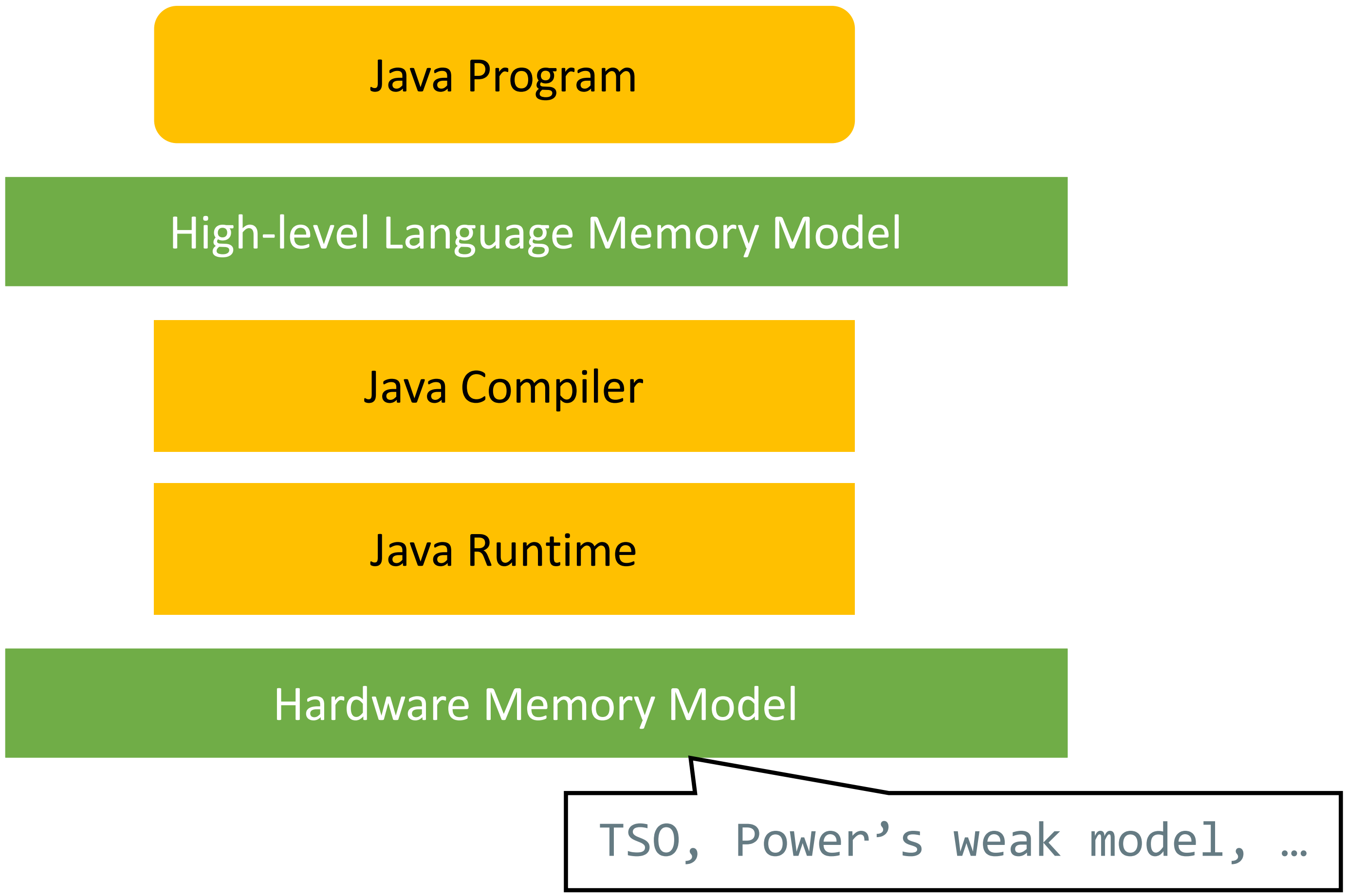
Final results: r2 == 2 and r1 == 1?

Decision: Disallowed. Violates sequential consistency

Java memory model balances between performance and safety

- Sequential consistency
  - Easy to understand
  - Restricts the use of many compiler and hardware transformations
- Relaxed memory models
  - Allow more optimizations
  - Hard to reason about the correctness

Java memory model hides underlying hardware memory model



Java defines data-race-free model

- *Data race* occurs when two threads access the same memory location, at least one of the accesses is a write, and there is no intervening synchronization
- A *data-race-free* Java program guarantees sequential consistency (Correctly synchronized)

Another definition of data-race from Java Memory Model's perspective

- Two accesses *x* and *y* form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by **happens-before**

Happens-before memory model

- A simpler version than the full Java Memory Model
  - *Happens-before order*
    - The transitive closure of *program order* and the *synchronizes-with order*
  - *Happens-before consistency*
    - Determines the value that a non-volatile read can see
  - *Synchronization order consistency*
    - Determines the value that a volatile read can see
- Solves part of the Java mysterious problems

Program order definition

- The program order of thread *T* is a total order that reflects the order in which these actions would be performed according to intra-thread semantics of *T*
  - If *x* and *y* are actions of the same thread and *x* comes before *y* in program order, then *hb(x, y)* (i.e. *x* happens-before *y*)

Eliminate ambiguity in program order definition

- Given a program in Java

```
y = 6
x = 5
```

- Program order does not mean that *y = 6* must be subsequent to *x = 5* from a wall clock perspective. It only means that the sequence of actions executed must be *consistent* with that order

What should be consistent in program order

- Happens-before consistency
  - A read *r* of a variable *v* is allowed to observe a write *w* to *v* if
    - *r* does not happen-before *w* (i.e., it is not the case that *hb(r, w)*) – a read cannot see a write that happens-after it, and
    - There is no intervening write *w0* to *v* (i.e., no write *w0* to *v* such that *hb(w, w0), hb(w0, r)*) –the write *w* is not overwritten along a happens-before path.
- Given a Java program

```
1 y = 6
2 x = 5
3 z = y
```

- *hb(1, 2) & hb(2, 3) -> hb(1, 3),* so *z* sees 6 be written to *y*

Synchronization Order

- A synchronization order is a total order over all of the synchronization actions of an execution
  - A write to a volatile variable *v* *synchronizes-with* all subsequent reads of *v* by any thread;
  - An unlock action on monitor *m* *synchronizes-with* all subsequent lock actions on *m* that were performed by any thread;
  - …
- If an action *x* *synchronizes-with* a following action *y*, then we have *hb(x, y)*

What should be consistent in synchronization order

- Synchronization order consistency
  - Synchronization order is consistent with program order
  - Each read *r* of a volatile variable *v* sees the last write to *v* to come before it in the synchronization order

Happens-before memory model example

Thread 1		Thread 2
1 x = 1		3 **if** (ready)
2 ready = **true**		4 r1 = x

synchronization-with

program order

Initial conditions: x = 0, ready = false, ready is *volatile*

Final results: r1 == 1?

Decision: Allowed. The program is correctly synchronized

Proof:

Program order: *hb(L1, L2)* and *hb(L3, L4)*
Synchronization order: *hb(L2, L3)*
Transitive: *hb(L1, L4)*

Recall data-race definition:
Two accesses *x* and *y* form a data race in an execution of a program if they are from different threads, they conflict, and they are not ordered by **happens-before**

Correct double-checked lock with volatile

```java
public class Instance {
  private volatile static Instance instance;
  public static Instance getInstance() {
    if (instance == null) {
      synchronized (Instance.class) {
        if (instance == null) {
            instance = new Instance();
        }
      }
    }
    return instance;
  }
}
```

```
1. mem = allocate();
2. ctorSingleton(mem);
3. object.instance = mem;
```

hb(2, 3) ensures that L2's write to mem can be seen at L3

Happens-before doesn’t solve all problems

Thread 1

```
r1 = x
if (r1 != 0)
    y = 42
```

Thread 2

```
r2 = y
if (r2 != 0)
    x = 42
```

Initial conditions: x = y = 0

Final results: r1 == r2 == 42?

Decision: Disallowed. Because the values are out-of-thin-air.

In a future aggressive system, Thread 1 could speculatively write the value 42 to y.
How to propose a methodology to disallow these behaviors?

Happens-before doesn't solve all problems

Thread 1

```
1 r1 = a
2 r2 = a
3 if (r1 == r2)
4    b = 2
```

Thread 2

```
5 r3 = b
6 a = r3
```

⟶

Thread 1

```
1 b = 2
2 r1 = a
3 r2 = r1
4 if (true)
```

Thread 2

```
4 r3 = b
5 a = r3
```

Initial conditions: a = 0, b = 1

Final results: r1 == r2 == r3 == 2?

Decision: Allowed. A compiler may determines that *r1* and *r2* have the same value and eliminate ***if** r1 == r2* (L3). Then, *b = 2* (L4) can be moved to an earlier position (L1)

What's the difference between two programs

- One difference between the acceptable and unacceptable results is that in latter program, the write that we perform (i.e. $b = 2$) would also have occurred if we had carried on the execution in a sequentially consistent way.
- In the former program, value 42 in any sequentially consistent execution will not be written.

Well-behaved execution

- We distinguish two programs by considering whether those writes could occur in a sequentially consistent execution.
- Well-behaved execution
    - A read that must return the value of a write that is ordered before it by *happens-before*.

Disallowed examples-data dependency

Thread 1

```
1 r1 = x
2 a[r1] = 0
3 r2 = a[0]
4 y = r2
```

Thread 2

```
5 r3 = y
6 x = r3
```

Initial conditions: x = y = 0; a[0] = 1, a[1] = 2

Final results: r1 == r2 == r3 == 1?

Decision: Disallowed. Because values are out-of-thin-air.

Proof: We have *hb(L1, L2)*, *hb(L2, L3)*. To let *r2 == 1*, *a[0]* must be *0*. Since initially *a[0] == 1* and *hb(L2, L3)*, we have *r1 == 0* at *a[r1] = 0* (L2). *r1* at *L2* is the final value Because *hb(L1, L2)*, *r1* at L2 must see the write to *r1* at *r1 = x* (L1).

Disallowed examples-control dependency

Thread 1

```
1  z = 1
```

Thread 2

```
2  r1 = z
3  if (r1 == 0)
4      x = 1
```

Thread 3

```
5  r2 = x
6  y = r2
```

Thread 4

```
7  r3 = y
8  x = r3
```

Initial conditions: x = y = z = 0

Final results: r1 == r2 == r3 == 1?

Decision: Disallowed. Because values are out-of-thin-air.

Proof: Because we have *hb(L5, L6)*, to let *r2 == 1* (so that y = 1), *x = 1* (L4) must be executed. If *L4* is executed, *if (r1 == 0)* (L3) must be **true**. However, since *r1 == 1* and *hb(L2, L3), L4* cannot be executed.

Disallowed examples-control dependency

Thread 1

```
r1 = x
if (r1 == 0)
    x = 1
```

Thread 2

```
r2 = x
y = r2
```

Thread 3

```
r3 = y
x = r3
```

Initial conditions: x = y = 0

Final results: r1 == r2 == r3 == 1?

Decision: Disallowed. Because values are out-of-thin-air

Proof: The same reason as the previous example.

Causality

- Actions that are committed earlier may cause actions that are committed later to occur
- The behavior of incorrectly synchronized programs is bounded by causality
- The causality requirement is strong enough to respect the safety and security properties of Java and weak enough to allow standard compiler and hardware optimizations

Justify a correct execution

- Build up causality constraints to *justify* executions
  - Ensures that the occurrence of a committed action and its value does not depend on an uncommitted data race
- Justification steps
  - Starting with the empty set as C_0
  - Perform a sequence of 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 containing C_i that meets certain conditions

Justification examples-reorder

Thread 1	Thread 2		Thread 1	Thread 2
1 r1 = x 2 y = 1	3 r2 = y 4 x = r2	→	1 y = 1 2 r1 = x	3 r2 = y 4 x = r2

Initial conditions: x = y = 0

Final results: r1 == r2 == 1?

Decision: Allowed, because of compiler transformation. *y = 1* (L2) is a constant that does not affect *r1 = x* (L2).

C1:	**C2:**	**C3:**	**C4:**
→ y = 1	y = 1	y = 1	y = 1
r2 = y (0)	→ r2 = y (1)	r2 = y (1)	r2 = y (1)
x = r2 (0)	x = r2 (1)	→ x = r2 (1)	x = r2 (1)
r1 = x (0)	r1 = x (0)	r1 = x (0)	→ r1 = x (1)

Justification examples-redundant elimination

Thread 1

```
r1 = a
if (r1 == 1)
   b = 1
```

Thread 2

```
r2 = b
if (r2 == 1)
   a = 1
if (r2 == 0)
   a = 1
```

$\longrightarrow$

Thread 1

```
r1 = a
if (r1 == 1)
   b = 1
```

Thread 2

```
a = 1
r2 = b
```

Initial conditions: a = b = 0

Final results: r1 == r2 == 1?

Decision: Allowed. A compiler could determine that Thread 2 always writes *1* to *a* and hoists the write to the beginning of Thread 2.

C1:	**C2:**	**C3:**	**C4:**
→ a = 1	a = 1	y = 1	y = 1
r1 = a (0)	→ r1 = a (1)	r1 = a (1)	r1 = a (1)
b = 1 (0)	b = 1 (0)	→ b = 1 (1)	b = 1 (1)
r2 = b (0)	r2 = b (0)	r2 = b (0)	→ r2 = b (1)

Justification examples-inter thread analysis

Thread 1

```
r1 = x
r2 = 1 + r1*r1 – r1
y = r2
```

Thread 2

```
r3 = y
x = r3
```

→

Thread 1

```
r2 = 1
y = r2
r1 = x
```

Thread 2

```
r3 = y
x = r3
```

Initial conditions: x = y = 0

Final results: r1 == r2 == 1?

Decision: Allowed. Interthread analysis could determine that *x* and *y* are always either *0* or *1*, and thus determine that *r2* is always *1*. Once this determination is made, the write of *1* to *y* could be moved early in Thread *1*.

C1:	**C2:**	**C3:**	**C4:**	**C5:**
r2 = 1	r2 = 1	r2 = 1	r2 = 1	r2 = 1
y = r2 (0)	y = r2 (1)	y = r2 (1)	y = r2 (1)	y = r2 (1)
r3 = y (0)	r3 = y (0)	r3 = y (1)	r3 = y (1)	r3 = y (1)
x = r3 (0)	x = r3 (0)	x = r3 (0)	x = r3 (1)	x = r3 (1)
r1 = x (0)	r1 = x (0)	r1 = x (0)	r1 = x (0)	r1 = x (1)

Comparison between allowed examples and disallowed examples

Thread 1	Thread 2	Thread 3
1 r1 = x 2 **if** (r1 == 0) 3 x = 1	4 r2 = x 5 y = r2	6 r3 = y 7 x = r3

Initial conditions: x = y = 0

Final results: r1 == r2 == r3 == 1?

Thread 1	Thread 2
1 r1 = x 2 **if** (r1 == 0) 3 x = 1 4 r2 = x 5 y = r2	6 r3 = y 7 x = r3

Decision: Allowed. Interthread analysis could determine that *x* is always 0 or 1. So we can replace *r2 = x* by *r2 = 1* and *y = r2* by *y = 1*. After moving *y = 1* and *r2 = 1* to an earlier position, we get *r1 == r2 == r3*.

Comparison between allowed examples and disallowed examples

Thread 1
```
1 r1 = x
2 if (r1 == 0)
3     x = 1
```

Thread 2
```
4 r2 = x
5 y = r2
```

Thread 3
```
6 r3 = y
7 x = r3
```

Initial conditions: a = b = 0

Final results: r1 == r2 == r3 == 2?

Thread 1
```
1 r1 = a
2 r2 = a
3 if (r1 == r2)
4     b = 2
```

Thread 2
```
1 r3 = b
2 a = r3
```

Decision: Allowed. Although there are some SC executions in which *r1 != r2* (L3), we can hoist *b = 2* (L4) to an earlier position and there is an SC execution such that *r1 == r2*. For the above case, there's no SC execution such that *r1 == 0* (L2) is **true** and *r1 == r2 == r3 == 1*. That is, if we hoist *x = 1* to an earlier position, L2 must be **false**.

Practical issues-final field

- "For space reasons, we omit discussion of two important issues in the Java memory model: the treatment of **final fields**, and **finalization / garbage collection**."
- Java's *final* field also allows programmers to implement thread-safe immutable objects without synchronization

Rule of thumb

- Set the final fields for an object in that object's constructor; and 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.
- What happens in the constructor
  - If a read occurs after the field is set in the constructor, it sees the value the final field is assigned, otherwise it sees the default value.

Can we change a final field?

- Reflection introduces problems
- The specification allows aggressive optimization of final fields. Within a thread, it is permissible to reorder reads of a final field with those modifications of a final field that do not take place in the constructor.

Practical issues-final field

- **new** A().f() could return -1, 0, or 1

```
class A {
  final int x;
  A() { x = 1; }
  int f() { return d(this,this); }
  int d(A a1, A a2) {
    int i = a1.x;
    g(a1);
    int j = a2.x;
    return j - i;
  }
  static void g(A a) {
      // uses reflection to change a.x to 2
  }
}
```

reorder!

Practical issues-final field

- **new** A().f() could return -1, 0, or 1

```
class A {
  final int x;
  A() { x = 1; }
  int f() { return d(this,this); }
  int d(A a1, A a2) {
    int i = a1.x;
    g(a1);
    int j = a2.x;
    return j - i; return 1
  }
  static void g(A a) {
      // uses reflection to change a.x to 2
  }
}
```

Practical issues-final field

- **new** A().f() could return -1, 0, or 1

```
class A {
  final int x;
  A() { x = 1; }
  int f() { return d(this,this); }
  int d(A a1, A a2) {
    g(a1);
    int i = a1.x;
    int j = a2.x;
    return j - i; return 0
  }
  static void g(A a) {
    // uses reflection to change a.x to 2
  }
}
```

Practical issues-final field

- **new** A().f() could return -1, 0, or 1

```
class A {
  final int x;
  A() { x = 1; }
  int f() { return d(this,this); }
  int d(A a1, A a2) {
    int j = a2.x;
    g(a1);
    int i = a1.x;
    return j - i; return -1
  }
  static void g(A a) {
    // uses reflection to change a.x to 2
  }
}
```

Practical issue-efficient singleton

- The *initialization-on-demand holder* (design pattern) idiom is a lazy-loaded singleton. In all versions of Java, the idiom enables a safe, highly concurrent lazy initialization with good performance.

```
public class SafeInstance {
  private SafeInstance() {}
  private static class LazyHolder {
    static final SafeInstance INSTANCE = new SafeInstance();
  }
  public static SafeInstance getInstance() {
    return LazyHolder.INSTANCE;
  }
}
```

Why initialization-on-demand holder is safe?

- When the class is initialized?
- A class or interface type T will be initialized immediately before the first occurrence of any one of the following:
  - A static field declared by T *is used* and the field is not a *constant variable*
    - A variable of primitive type or type String, that is final and initialized with a compile-time constant expression is called a *constant variable*.
  - …

Why initialization-on-demand holder is safe?

- Why initialization is safe?
- For each class or interface *C*, there is a unique initialization lock *LC*. The mapping from *C* to *LC* is left to the discretion of the Java Virtual Machine implementation.
- We can also implement singleton by ENUM in Java.

Conclusion

- Following happens-before rules allows us to write a data-race-free program that is correctly synchronized.
- Java memory model provides a clear definition of well-behaved executions, preventing values come out-of-thin-air in the presence of data race.
- Double-checked lock is thread-safe for JVM later than v1.5.

Reference Books

- Goetz, Brian, et al. *Java concurrency in practice*. Pearson Education, 2006.
- Gosling, James, et al. *The Java language specification*. Pearson Education, 2014.
- Lea, Doug. "The JSR-133 cookbook for compiler writers." (2008).

Reference URLs

- Double-checked locking

https://www.ibm.com/developerworks/java/library/j-dcl/index.html

- Causality test cases

http://www.cs.umd.edu/~pugh/java/memoryModel/unifiedProposal/testcases.html