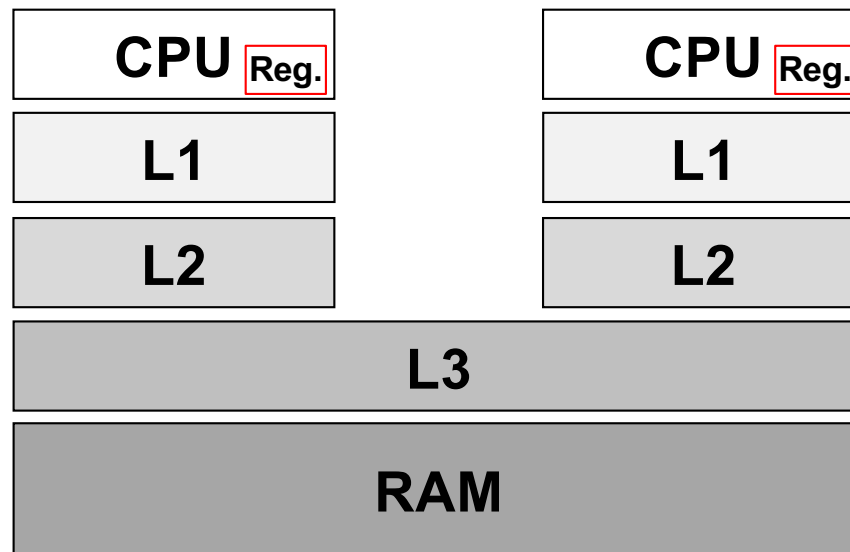


Java Memory Model: Content

- **Memory Models**
- **Java Memory Model: Happens Before Relation**
- **Volatile: in depth**
- **Double Checked Locking Problem**

Java Memory Model

- **Key Ideas:**
 - All threads share the main memory
 - Each thread uses a local working memory (conceptually)
 - In reality several cache levels are provided
 - **Flushing** and **Refreshing** working memory to and from the main memory must comply with the Memory Model



Java Memory Model

- **JMM specifies guarantees given by the JVM**
 - About when writes to variables become visible to other threads
 - Revised as part of JSR-133 (=> part of JDK 1.5)
- **JMM is an abstraction on top of hardware memory models**

Java Memory Model

Threads read and write to variables

Hardware Memory Model

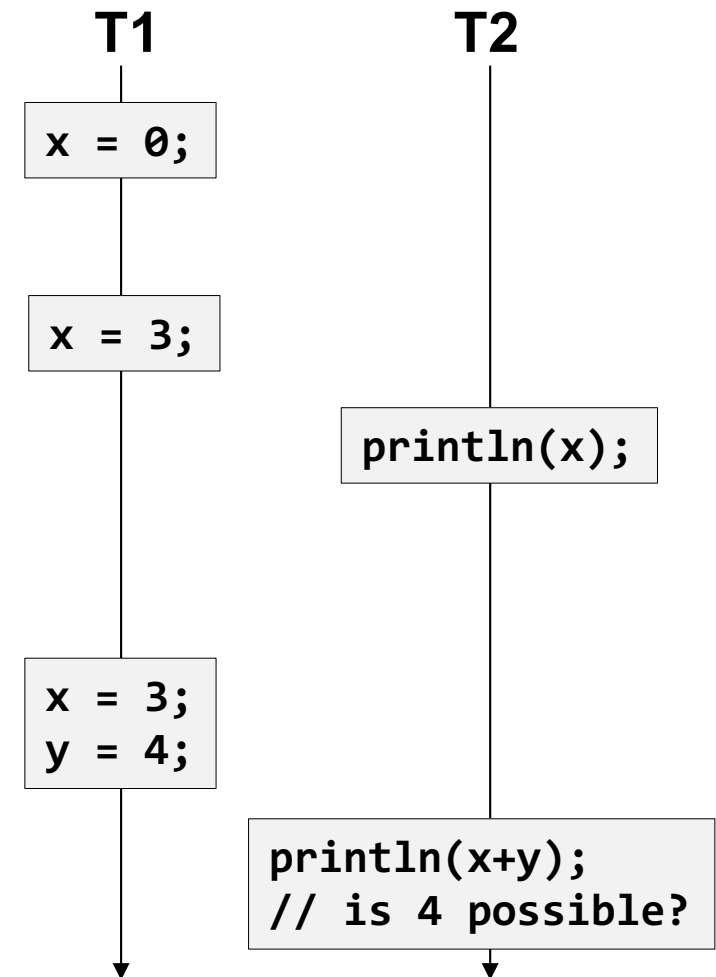
Processors read and write to registers

Processors read and write in caches

Processors read and write in main memory

Memory Model Guarantees

- **Atomicity**
 - Which operations are indivisible?
- **Visibility**
 - Under which condition are the effects of operations executed by one thread visible to other threads?
- **Ordering**
 - Under which conditions can the effects of operations appear out of order to any given thread?
 - Related to visibility issue



Memory Consistency Models

- **Sequential Consistency [unrealistic model]**
 - Each read of a variable will see the last write in the execution order
 - *"... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."*

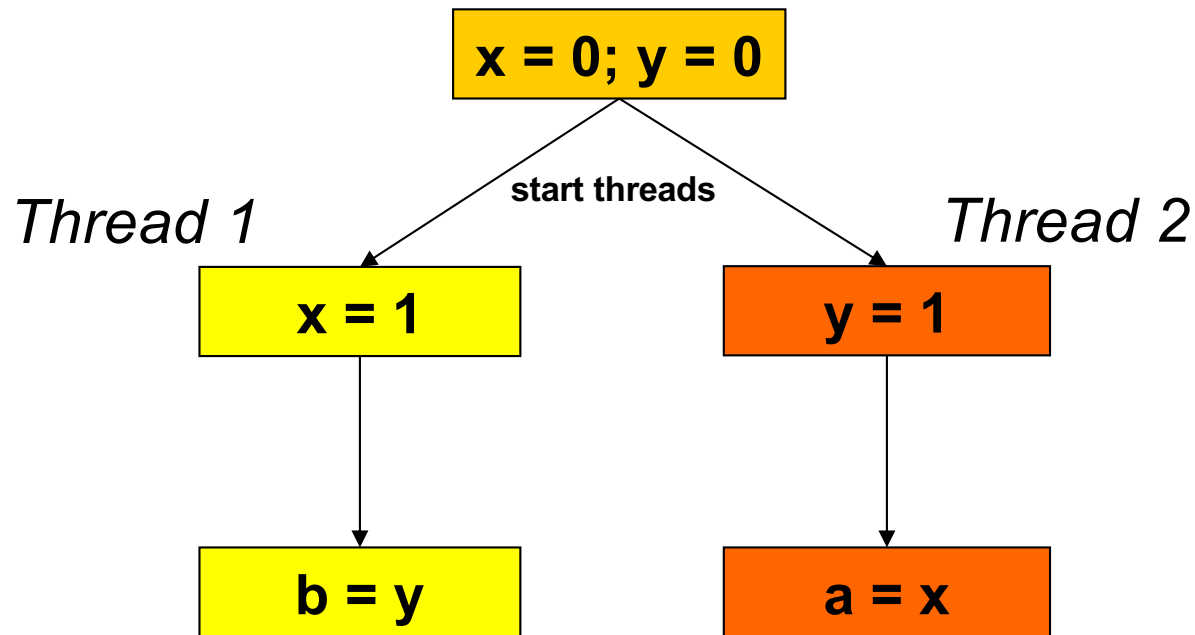
[Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs",
IEEE Transactions on Computers 28,9 (Sept. 1979), 690-691.]
 - Unpractical: caching mechanisms would be useless
 - Reorderings done by the compiler would not be permitted
- **=> JMM does not guarantee sequential consistency**

Java Memory Model

- **JLS only requires the JVM to maintain *within-thread as-if-serial semantics***
 - As long as a thread has the same result as if it were executed in program order in a strictly sequential environment, the following *games* are permissible
 - Caching
 - Compilers may store variables in registers
 - Caches may vary the order in which writes to variables are committed to main memory
 - Reordering
 - Processors may execute instructions in parallel or out of order (pipelining)
 - Dynamic instruction scheduling / Speculative execution

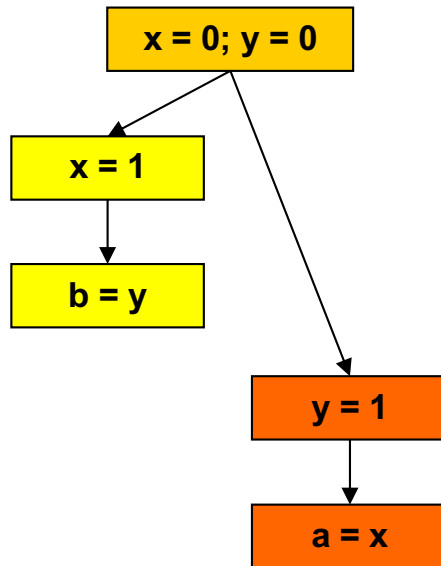
We need some rules how inter-thread memory actions are processed!

Example

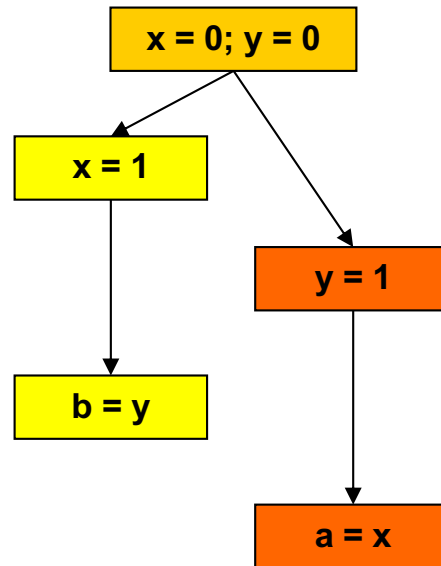


- Which values can fields a and b have after the execution of these statements

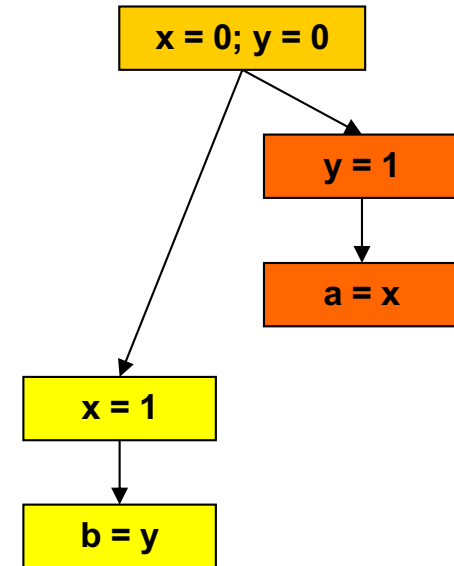
Example



$a = 1 / b = 0$

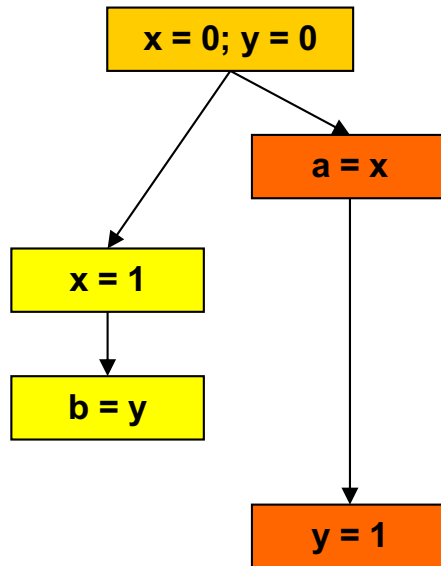


$a = 1 / b = 1$

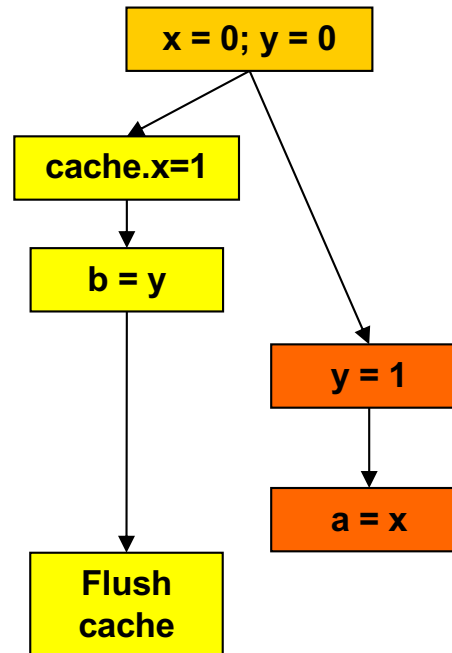


$a = 0 / b = 1$

Example



Compiler Reordering
a = 0 / b = 0

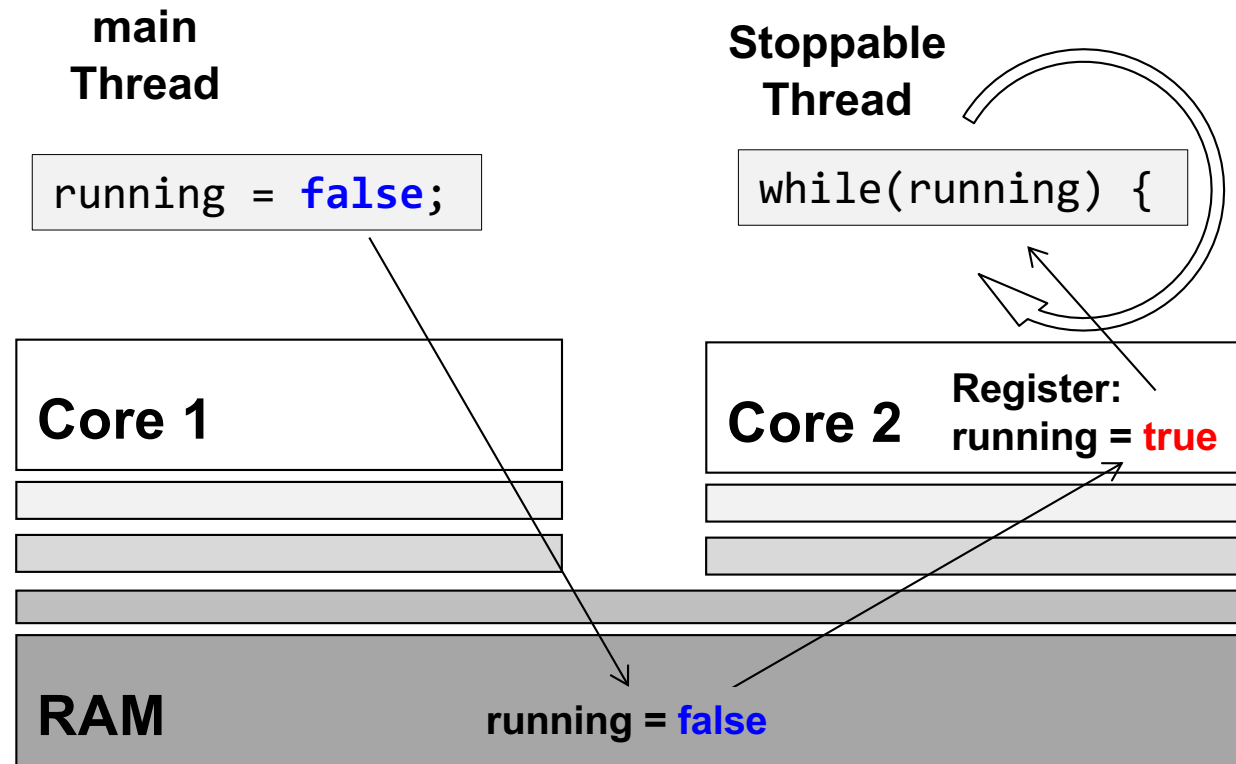


Cache Effect
a = 0 / b = 0

Java Memory Model: Content

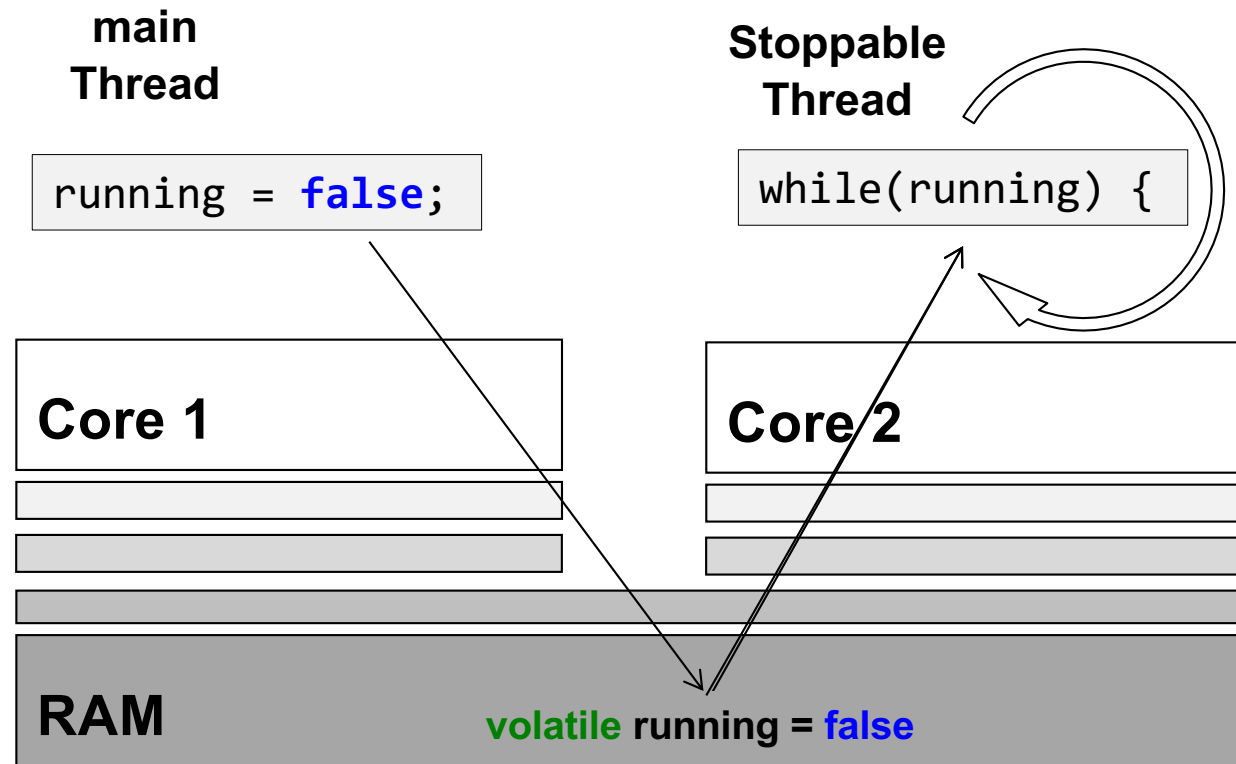
- **Memory Models**
- **Java Memory Model: Happens Before Relation**
- **Volatile: in depth**
- **Double Checked Locking Problem**

Stopping Threads Explained



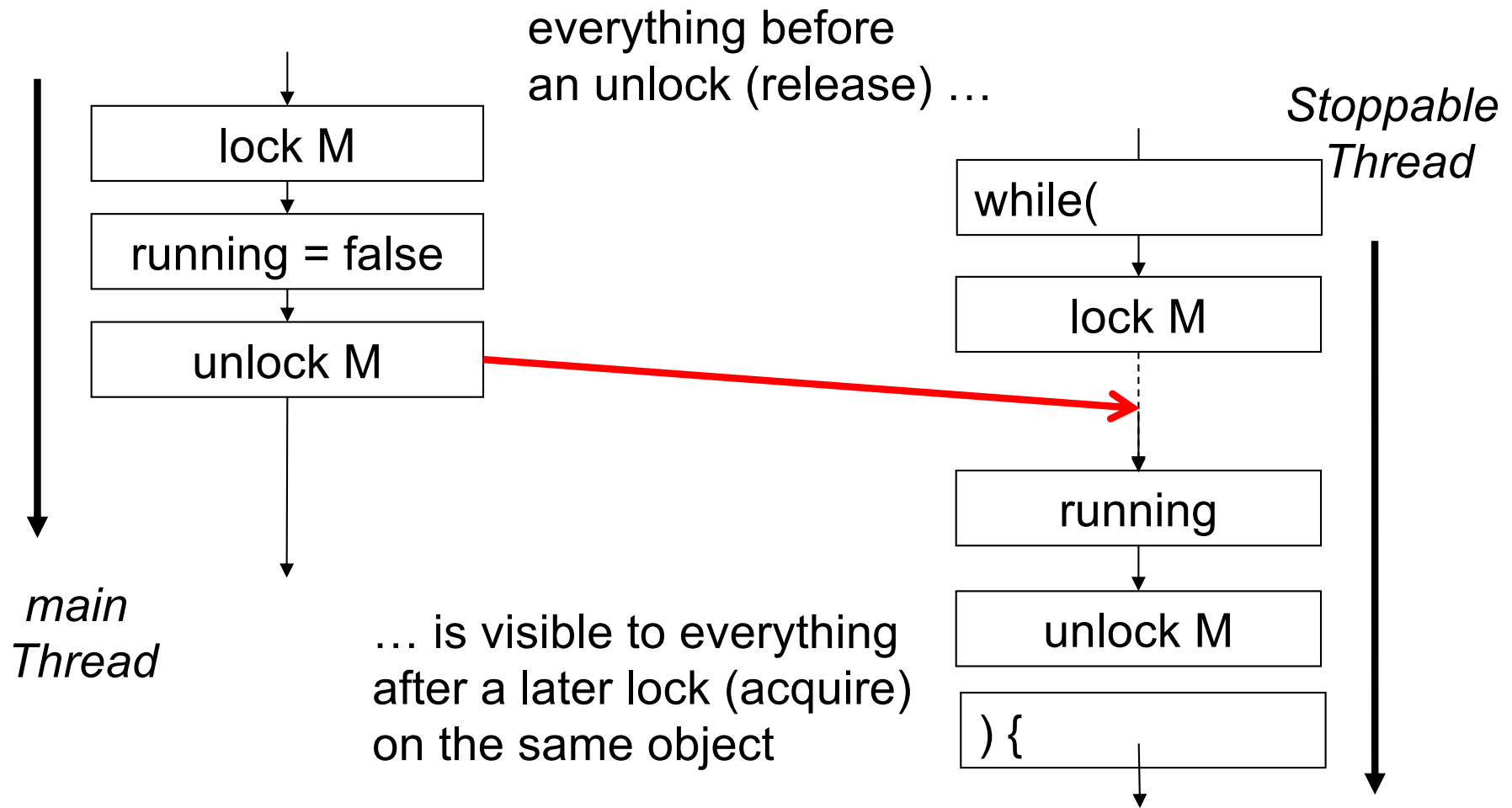
- StoppableThread caches a copy of the running flag in its cpu register
- main Thread writes `running = false` to the heap
- StoppableThread does not see the modification made by the main thread

Stopping Threads Explained

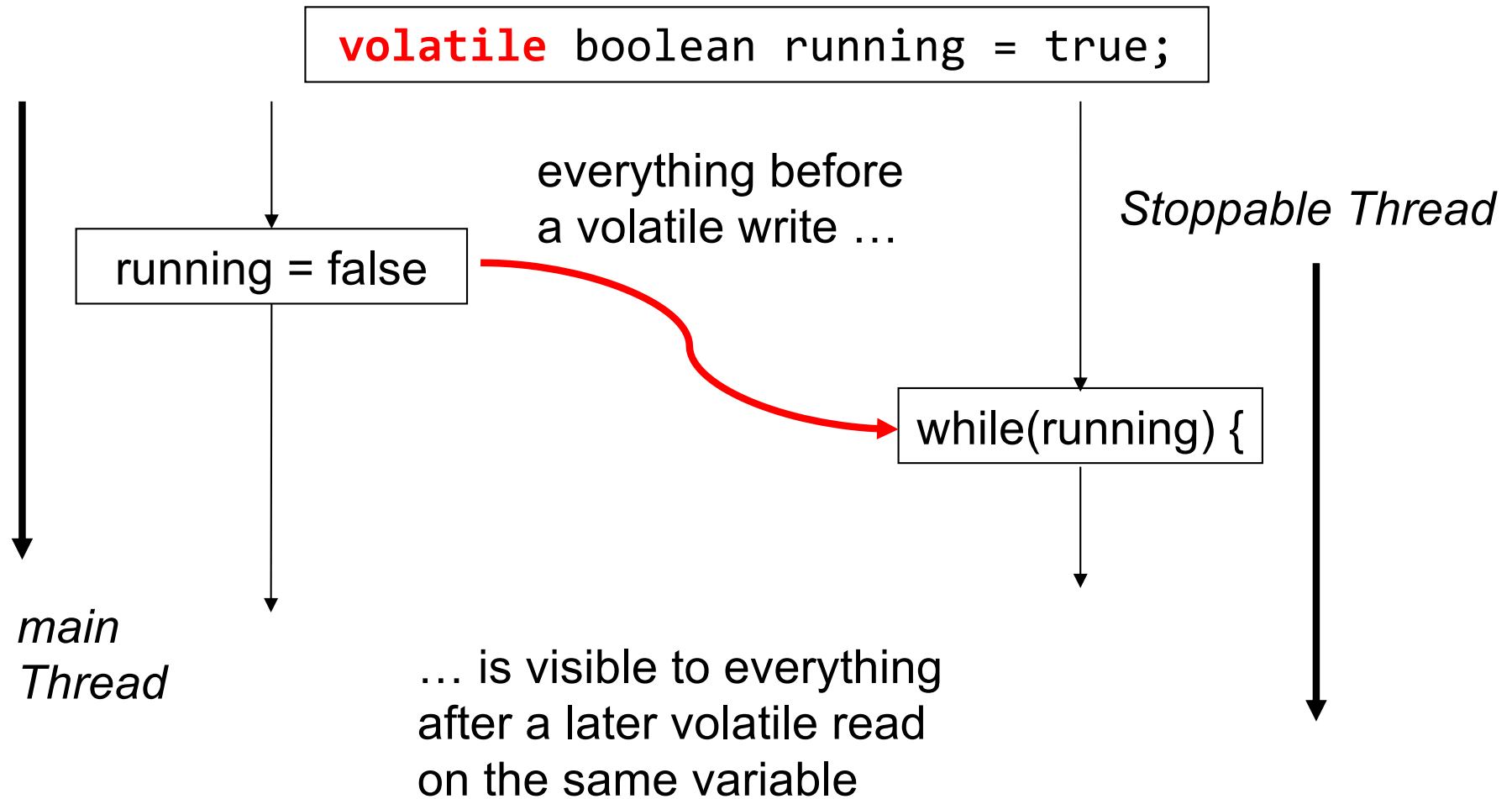


- For variables which are marked as **volatile**, local caching is not allowed!

Stopping Threads Explained



Stopping Threads Explained



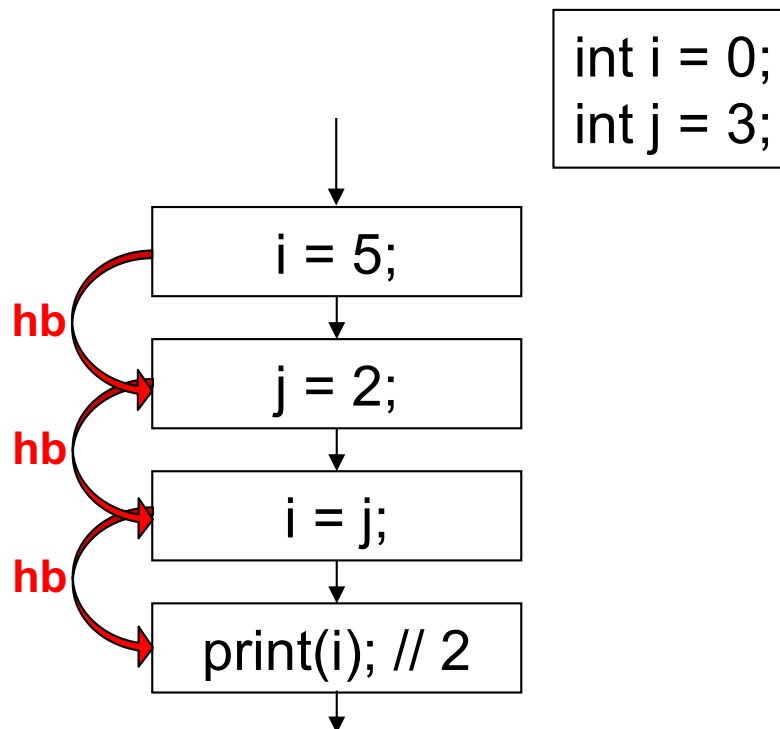


Java Memory Model: Visibility & Ordering

- **JMM defines a partial order called *happens-before* on actions**
 - Actions = variable read/write, monitor lock/unlock, thread start/join
 - To guarantee that a thread executing action B can **see the results of action A** (same or different thread) there must be a happens-before relationship between A and B, otherwise there is no guarantee!
 - Rules
 1. Each action in a thread *happens-before* every action in that thread that comes later in the program order
 2. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock
 3. A write to a volatile field *happens-before* every subsequent read of that same field
 4. A call to Thread.start on a thread *happens-before* every subsequent action in the started thread
 5. Actions in a thread t *happens-before* another thread detects its termination
 6. The *happens-before* order is transitive ($A \text{ --hb--> } B \ \&\& \ B \text{ --hb--> } C \Rightarrow A \text{ --hb--> } C$)

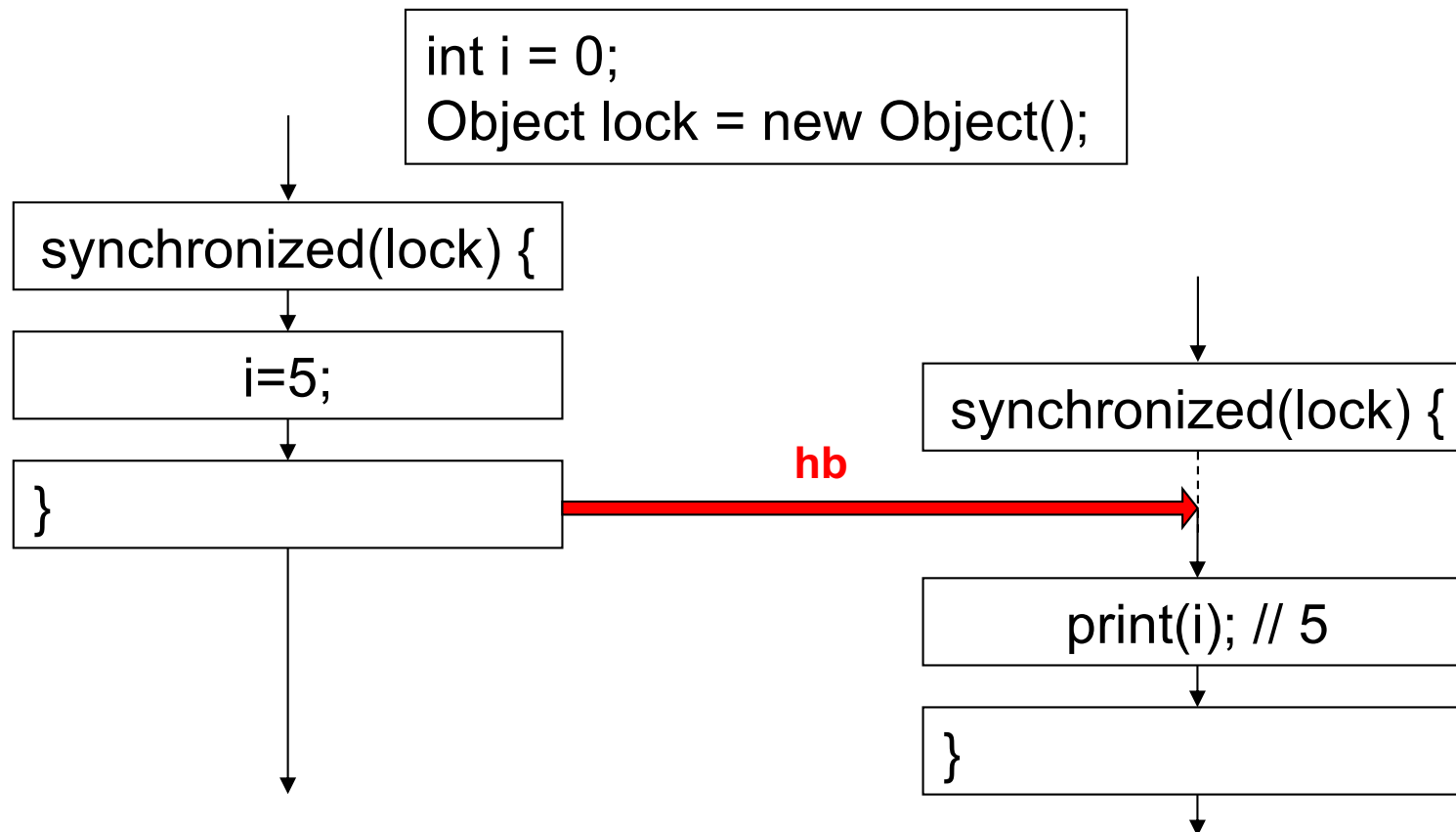
Java Memory Model Happens-Before Rules

- Each action in a thread *happens-before* every action in that thread that comes later in the program order



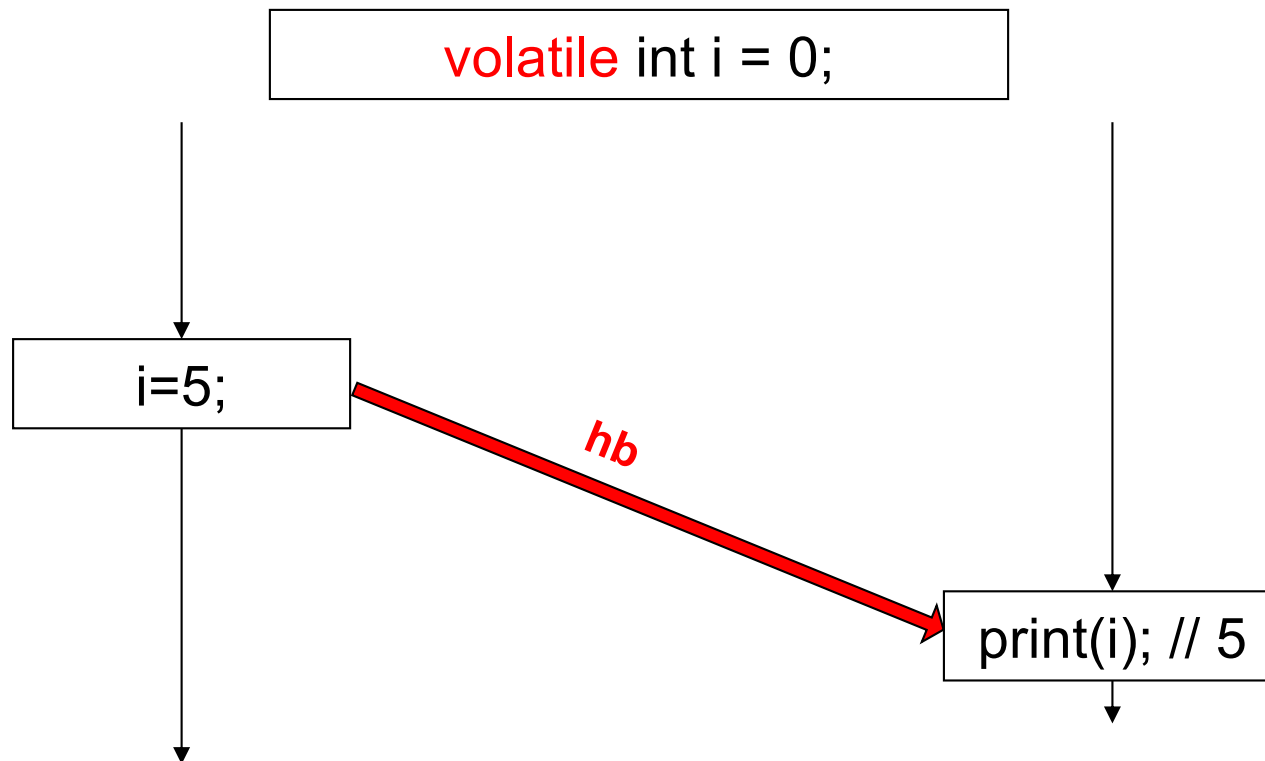
Java Memory Model Happens-Before Rules

- An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock



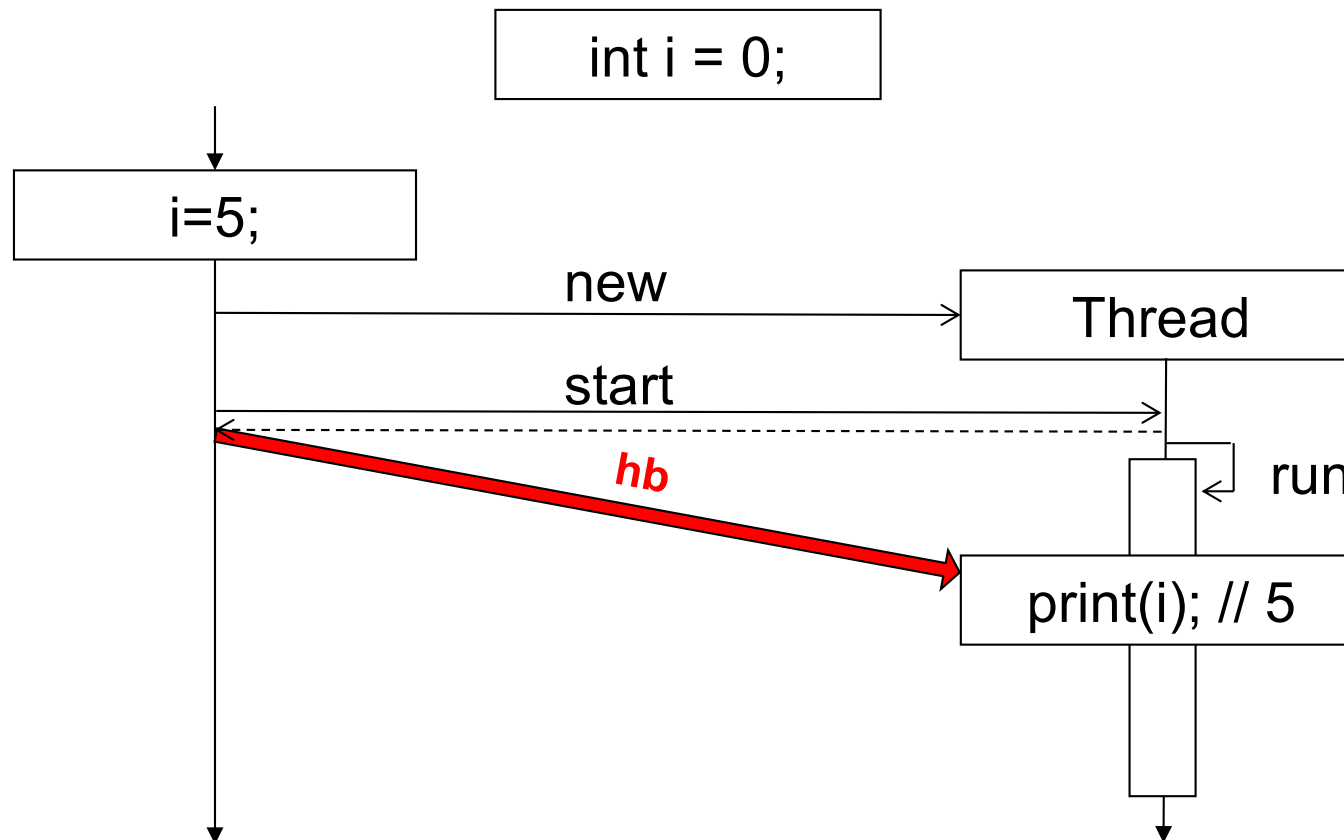
Java Memory Model Happens-Before Rules

- A write to a volatile field *happens-before* every subsequent read of that same field



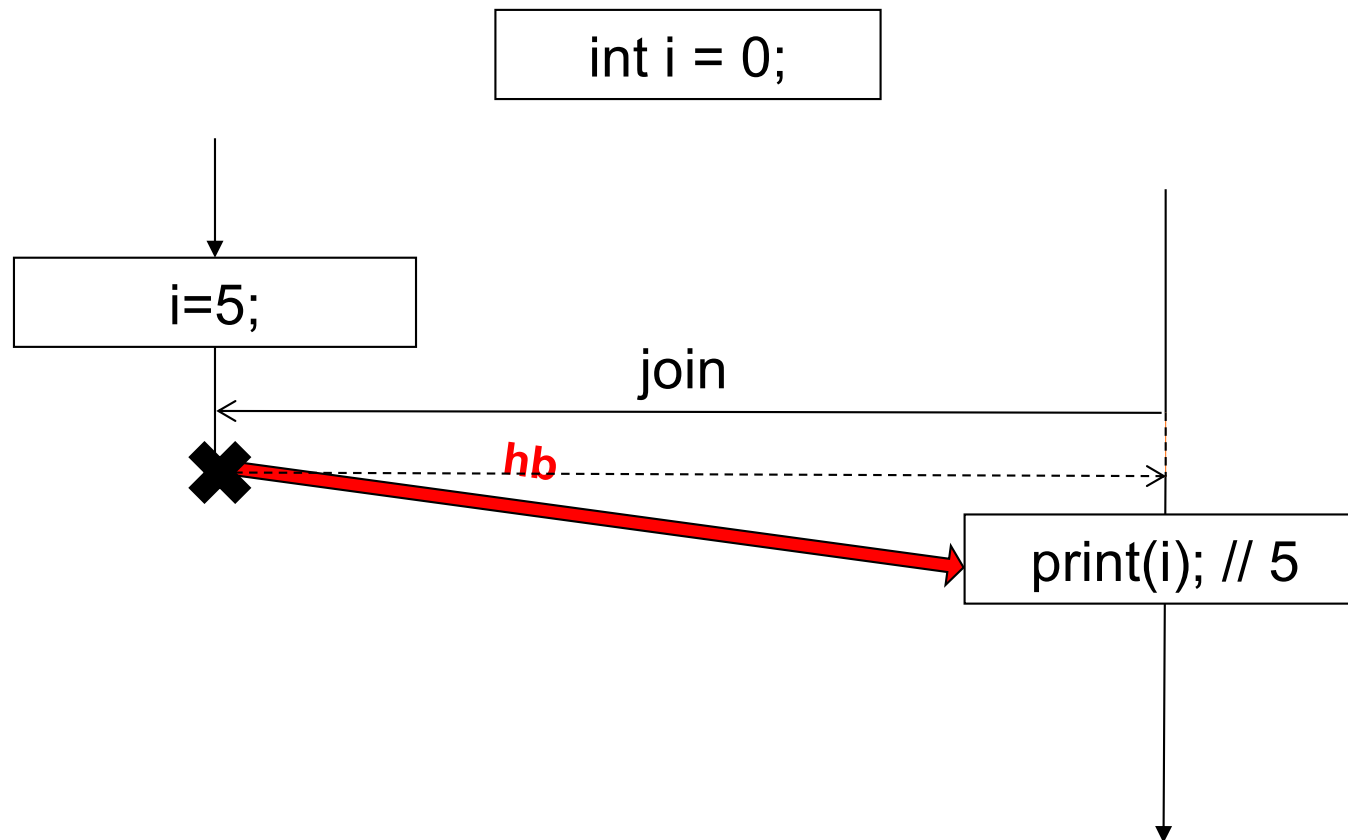
Java Memory Model Happens-Before Rules

- A call to `Thread.start` on a thread *happens-before* every subsequent action in the started thread



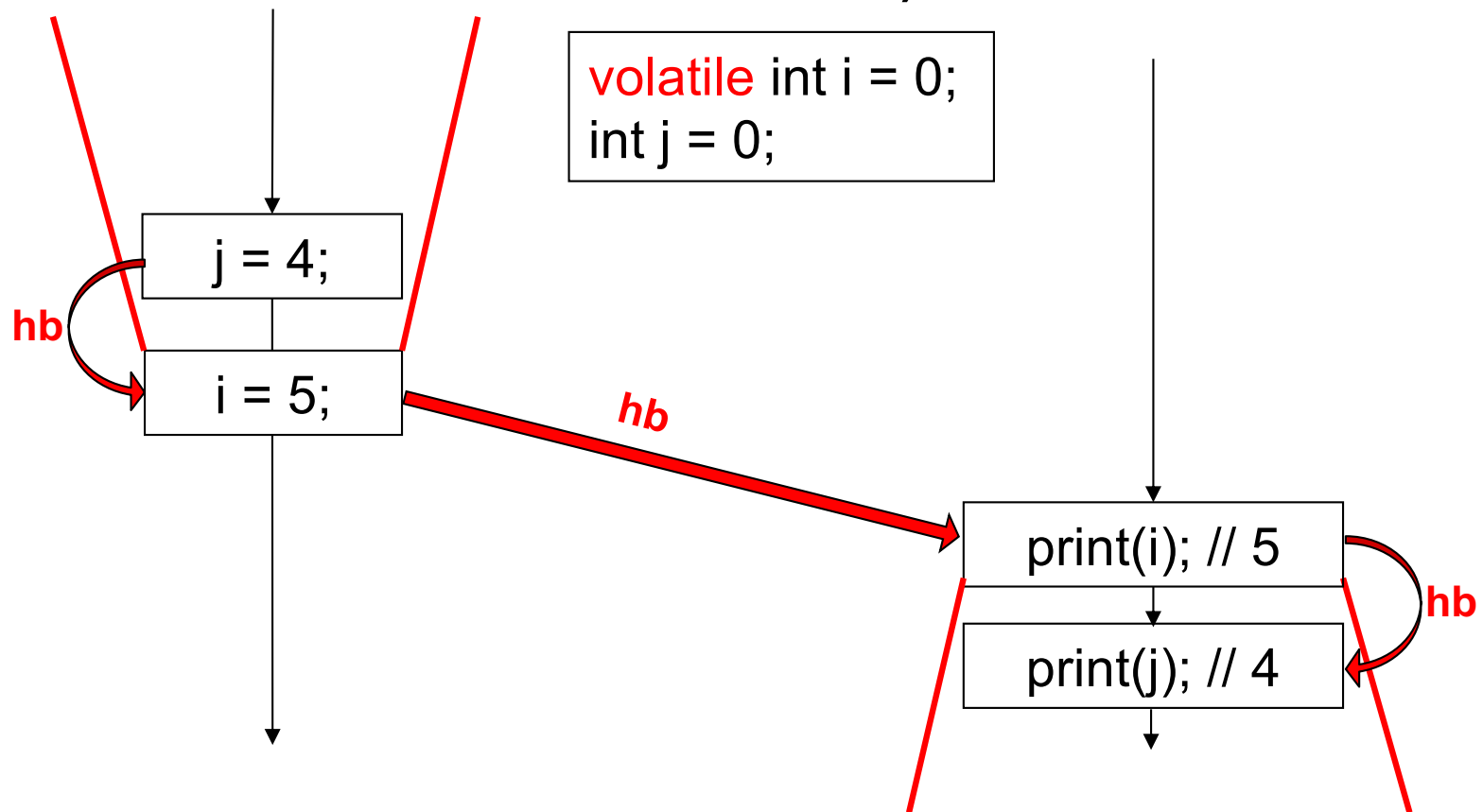
Java Memory Model Happens-Before Rules

- Actions in a thread *t* ***happens-before*** another thread detects its termination



Java Memory Model Happens-Before Rules

- The *happens-before* order is transitive
(A --hb-- B && B --hb-- C \Rightarrow A --hb-- C)



Java Memory Model: Visibility & Ordering

- **Synchronization**

- Synchronization guarantees that changes to object fields made in one thread are visible by other threads
 - Releasing a lock forces the thread to flush out all pending writes
 - Obtaining a lock forces the thread to get fresh values from memory

- **Thread-Start**

- Changes made in the starting thread are visible in the started thread
 - The start of a thread implies a flush of pending writes in the starting thread
 - Started thread will get fresh values from memory before run is executed

- **Thread-Stop**

- Changes made in a thread are visible by (actively) waiting threads
 - The end of a thread implies a flush
 - A thread which is informed about the end of another thread (*join*, *isAlive*) sees the changes performed by the terminated thread on shared fields

Java Memory Model: Visibility & Ordering

- **Volatile Fields**

```
volatile int x = 0, y = 0;
```

- Guarantees visibility of writes (i.e. volatile variables are never cached)
 - Read access to a volatile implies to get fresh values from memory
 - Write access to a volatile forces the thread to flush out all pending writes
- reads/writes of volatile longs/doubles are guaranteed to be atomic
- Volatile in Java ≥ 1.5
 - Declaring a variable x "volatile" is equivalent to wrapping all read/write access on x in synchronized blocks sharing the same lock object, except that "volatile" doesn't enforce mutual exclusion
 - Writes which *happen before* a volatile write access are visible by other threads reading **the same** volatile field

Volatile: Visibility 1

```
static int value = 0;  
static volatile boolean ready = false;
```

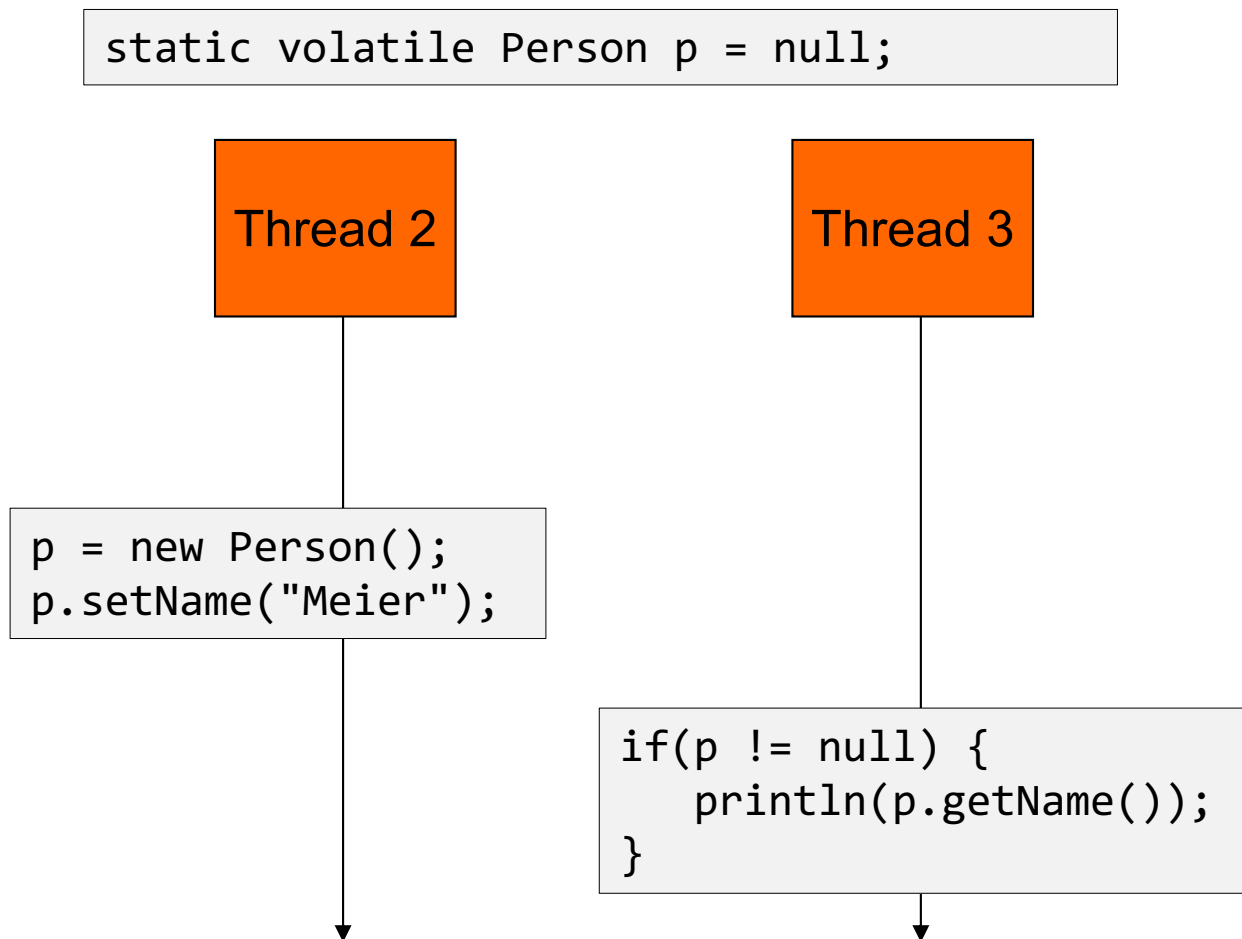
Thread 2

```
value = 77;  
ready = true;
```

Thread 3

```
if(ready)  
    println(value);
```


Volatile: Visibility 2



Volatile: Visibility 3

- **Method size**

```
class Stack {  
    private int[] array = new int[10];  
    private int  tos = 0;           // top of stack  
    public synchronized void push(int x) {  
        array[tos++] = x;  
    }  
    public synchronized int pop() {  
        return array[--tos];  
    }  
    public int size() { return tos; }  
}
```

- Size need not be declared as synchronized for data consistency
- int access is atomic, so a valid value is read

Java Memory Model: Content

- **Memory Models**
- **Java Memory Model: Happens Before Relation**
- **Volatile: in depth**
- **Double Checked Locking Problem**

Volatile: Performance

- **Costs of volatile access**
 - Compared to synchronized no management of wait queues and locks
 - Reading / writing volatiles (architecture dependent)
 - Cost for writing a volatile variable
 - "No cost" for reading a volatile variable (compared to non-volatile)
 - Access to a volatile variable inside a loop can be more expensive than synchronization of the whole loop

Volatile: Side effects to other variables

- **Example of a (simple) Exchanger (without synchronization!)**

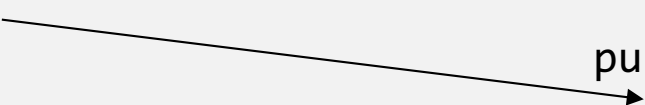
```
class Exchanger {  
    private volatile boolean ready = false;  
    private Object data = null;  
  
    public Object getResult() {  
        if(ready);  
        return data;  
    }  
  
    public void putResult(Object obj) { // only called once!  
        if(ready) throw new IllegalStateException();  
        data = obj; ready = true;  
    }  
}
```

- Access to volatile fields also has effects to the visibility of other fields
- Compiler cannot optimize away access to volatile fields

Volatile: Side effects to other variables

- **Example of a (simple) Exchanger (without synchronization!)**

```
class Exchanger {  
    private volatile boolean ready = false;  
    private Object data = null;  
  
    public void putResult(Object obj) {  
        if(ready) throw new IllegalStateException();  
        data = obj;  
        ready = true;  
    }  
  
    public Object getResult() {  
        if(ready);  
        return data;  
    }  
}
```

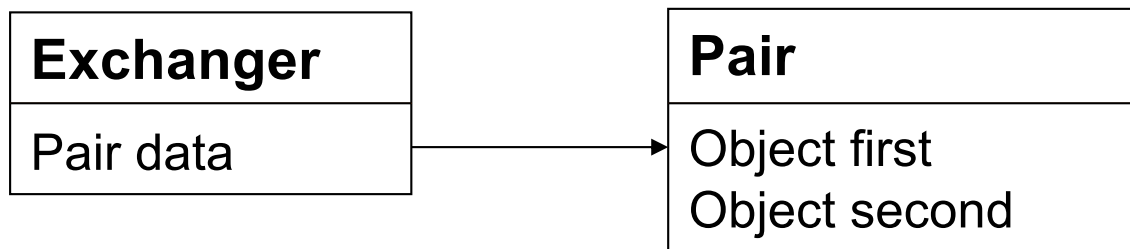
A black arrow originates from the line 'ready = true;' in the 'putResult' method and points to the 'if(ready);' line in the 'getResult' method, illustrating how the state of the 'ready' variable is checked after being updated.

- Access to volatile fields also has effects to the visibility of other fields
- Compiler cannot optimize away access to volatile fields

Volatile: Reference variables

- **Similar exchanger example, but with a reference**

```
class Pair {  
    private Object first, second;  
    public void setFirst(Object first) { this.first = first; }  
    public void setSecond(Object second) { this.second = second; }  
    public String toString() {  
        return "[" + first + ", " + second + "];"  
    }  
}
```



Volatile: Reference variables

- **Similar exchanger example, but with a reference**

```
class Exchanger {  
    private volatile Pair data = null;  
  
    public String getPairAsString() {  
        return data == null ? null : data.toString();  
    }  
    public boolean isReady() {  
        return data != null;  
    }  
    public void setPair(Object first, Object second) {  
        Pair tmp = new Pair();  
        tmp.setFirst(first);  
        tmp.setSecond(second);  
        data = tmp;  
    }  
}
```


Volatile: Reference variables

- **Discussion**

- Fields first/second are visible as they are written before the write to the volatile data field
- The following implementations would be wrong

```
public void setPair(Object first, Object second) {  
    data = new Pair();  
    data.setFirst(first);  
    data.setSecond(second);  
}
```

```
public void setPair(Object first, Object second) {  
    data = new Pair();  
    data.setFirst(first);  
    data.setSecond(second);  
    data = data;  
}
```

Volatile: Rules for the use of volatile

- **Rule 1: Independence of old value**
 - The new value to be written on a volatile does not depend on the old value
 - If the new value depends on the old value we have a read-modify-write sequence, atomicity can only be achieved with synchronization
- **Rule 2: Independence of other values**
 - If a variable depends on other variables, then several statements have to be executed in order to change an object from one consistent state to another consistent state
 - Again, synchronization is necessary

Volatile: Typical Applications

- **One-time events (latch)**
 - Variables which are changed only once can be defined as volatile
 - Latch / Barrier
 - Stopping of threads
- **Multiple Publication (blackboard)**
 - New information object completely replaces old one
 - `volatile double stockExchangePrice`
 - `volatile Object reference;` `// also works for references`
- **Lock / Volatile Combination**
 - Atomicity guaranteed with synchronization
 - Visibility guaranteed with volatile
 - `volatile int size`

Java Memory Model: Atomicity

- **Unsynchronized field access**
 - Access to variables of primitive type (except long/double) is atomic
 - Access to references is atomic (does *not* include access to referenced object)
 - Access to **volatile** variables (including long/double) is atomic
 - Access to atomic variables is atomic (=> lock free programming)

Java Memory Model: Atomicity

- **Unsynchronized field access**

- Atomicity does not mean that we get the most recent value! => visibility
- Atomicity only means that we will not get arbitrary bits

```
volatile long id = 0x0000000000000000L;
```

```
// Thread 1  
id = 0x0000000000000000L;
```

```
// Thread 2  
id = 0x1111111111111111L;
```

```
// Thread 3  
System.out.printf("%x\n", id);
```

- Possible results:

```
0000000000000000  
1111111111111111
```

JLS 17.7

Java Memory Model: Content

- **Memory Models**
- **Java Memory Model: Happens Before Relation**
- **Volatile: in depth**
- **Double Checked Locking Problem**

Double Checked Locking Problem

- **Singleton with eager initialization**

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    public static Singleton getInstance() { return instance; }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

- Instance is created upon loading of the class

Double Checked Locking Problem

- **Singleton with lazy initialization**

```
public class Singleton {  
    private static Singleton instance;  
    public synchronized static Singleton getInstance() {  
        if(instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

- Instance is only created upon access
- Disadvantage: Synchronization of getInstance

Double Checked Locking Problem

- **Singleton with double checked locking**

```
public class Singleton {  
    private static Singleton instance;  
    public static Singleton getInstance() {  
        if(instance == null) {  
            synchronized(Singleton.class) {  
                if(instance == null){  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

Volatile: Double Checked Locking Problem

- Singleton with double checked locking

```
public class Singleton {  
    private volatile static Singleton instance;  
    public static Singleton getInstance() {  
        if(instance == null) {  
            synchronized(Singleton.class) {  
                if(instance == null) {  
                    instance = new Singleton();  
                }  
            }  
        }  
        return instance;  
    }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

**Correct under
JMM (≥ 1.5)**

Double Checked Locking Problem

- **Singleton with eager initialization [Revisited]**

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    public static Singleton getInstance() { return instance; }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

- Is it correct?
 - T1 is invoking Singleton.getInstance() and loads class
 - T2 is invoking Singleton.getInstance() – does it see the initialized value?

Excursus: Class Initialization

- A class (or interface) of type T will be initialized *immediately* before the first occurrence of any of the following:
[JLS 12.4.1]
 - T is a class and an instance of T is created
 - T is a class and a static method declared by T is invoked
 - A static field declared by T is assigned
 - A static field declared by T is used and the field is not a constant variable
 - T is a top-level class, and an assert statement lexically nested within T is executed

Excursus: Class Initialization

- **Initialization Procedure [JLS 12.4.2]**
 1. **Synchronize on the Class object that represents the class or interface to be initialized.** This involves waiting until the current thread can obtain the lock for that object.
 2. If initialization is in progress for the class or interface *by some other thread*, then **wait** on this Class object (which temporarily releases the lock). When the current thread awakens from the wait, repeat this step.
 3. If initialization is in progress for the class or interface *by the current thread*, then this must be a recursive request for initialization. Release the lock on the Class object and complete normally.
 4. If the class or interface has already been initialized, then no further action is required. Release the lock on the Class object and complete normally (**but the lock establishes a happens-before relation**)
 5. If the Class object is in an erroneous state, then initialization is not possible. Release the lock on the Class object and throw a `NoClassDefFoundError`.

Excursus: Class Initialization

6. Otherwise, record the fact that initialization of the Class object is now in progress by the current thread and **release the lock** on the Class object.
7. Next, if the Class object represents a class rather than an interface, and the superclass of this class has not yet been initialized, then recursively perform this entire procedure for the superclass. If necessary, verify and prepare the superclass first. If the initialization of the superclass completes abruptly because of a thrown exception, then lock this Class object, label it erroneous, notify all waiting threads, release the lock, and complete abruptly, throwing the same exception that resulted from initializing the superclass.
8. Next, determine whether assertions are enabled for this class by querying its defining class loader.
9. Next, **execute either the class variable initializers** and static initializers of the class, or the field initializers of the interface, in textual order, as though they were a single block, except that final class variables and fields of interfaces whose values are compile-time constants are initialized first.

Excursus: Class Initialization

10. If the execution of the initializers completes normally, then lock this Class object, label it fully initialized, **notify all waiting threads**, release the lock, and complete this procedure normally.
11. Otherwise, the initializers must have completed abruptly by throwing some exception E . If the class of E is not Error or one of its subclasses, then create a new instance of the class `ExceptionInInitializerError`, with E as the argument, and use this object in place of E in the following step. But if a new instance of `ExceptionInInitializerError` cannot be created because an `OutOfMemoryError` occurs, then instead use an `OutOfMemoryError` object in place of E in the following step.
12. Lock the Class object, label it erroneous, notify all waiting threads, release the lock, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.

Double Checked Locking Problem

- **Singleton with eager initialization [Revisited]**

```
public class Singleton {  
    private static Singleton instance = new Singleton();  
    public static Singleton getInstance() { return instance; }  
    private Singleton() { /* initialization */ }  
    // other methods  
}
```

- Is it correct?

- T1 is invoking Singleton.getInstance() and loads class
- T2 is invoking Singleton.getInstance() – does it see the initialized value?

**Also correct under the
old memory model (< 1.5)**

**Correct under
JMM (≥ 1.5)**

Summary

- **Java Memory Model**

- Guarantees about visibility of shared memory
- Requires the JVM to maintain only *within-thread as-if-serial semantics*
- Defines happens-before relation across threads
 - Locking / Unlocking (using synchronized-blocks or `java.util.concurrent.locks`)
 - Volatile read / write
 - ... and some other rules

The Java Memory Model (JMM) is a relaxed memory model which acts as a contract between Java programmers, compiler writers and JVM implementers.