JavaOne℠

Sun's 2000 Worldwide Java Developer Conference

Correct and Efficient Synchronization of Java™ Technology-based Threads

Doug Lea and William Pugh
http://gee.cs.oswego.edu
http://www.cs.umd.edu/~pugh

Audience

- **Assume you are familiar with basics of Java™ technology-based threads ("Java threads")**
  - creating, starting and joining threads
  - synchronization
  - wait and notifyAll
- **Will talk about things that surprised a lot of experts**
  - including us, James Gosling, Guy Steele, …
  - (others discovered many of these)

Overview

- **Java Thread Spec**
- **Synchronization**
  - Properties
  - Costs
- **Some problems and solutions**
  - field access
  - initialization
  - collections

Java Thread Specification

- **Chapter 17 of the Java Language Spec**
  - Chapter 8 of the Virtual Machine Spec
- **Very, very hard to understand**
  - not even the authors understood it
  - has subtle implications
    - that forbid standard compiler optimizations
  - all existing JVM's violate the specification
    - some parts should be violated

Safety Issues in Multithreaded Systems

- **Many intuitive assumptions do not hold**
- **Some widely used idioms are not safe**
  - double-check idiom
  - checking non-volatile flag for thread termination
- **Can't use testing to check for errors**
  - some anomalies will occur only on some platforms
    - e.g., multiprocessors

Revising the Thread Spec

- **Work is underway to consider revising the Java Thread Spec**
  - http://www.cs.umd.edu/~pugh/java/memoryModel
- **Goals**
  - Clear and easy to understand
  - Foster reliable multithreaded code
  - Allow for high performance JVM's
- **May effect JVM's**
  - and badly written existing code
    - including parts of Sun's JDK

What to do Today?

- **Guidelines we will provide should work under both existing and future thread specs**
- **Don’t try to read the official specs**
- **Avoid corner cases of the thread spec**
  - not needed for efficient and reliable programs

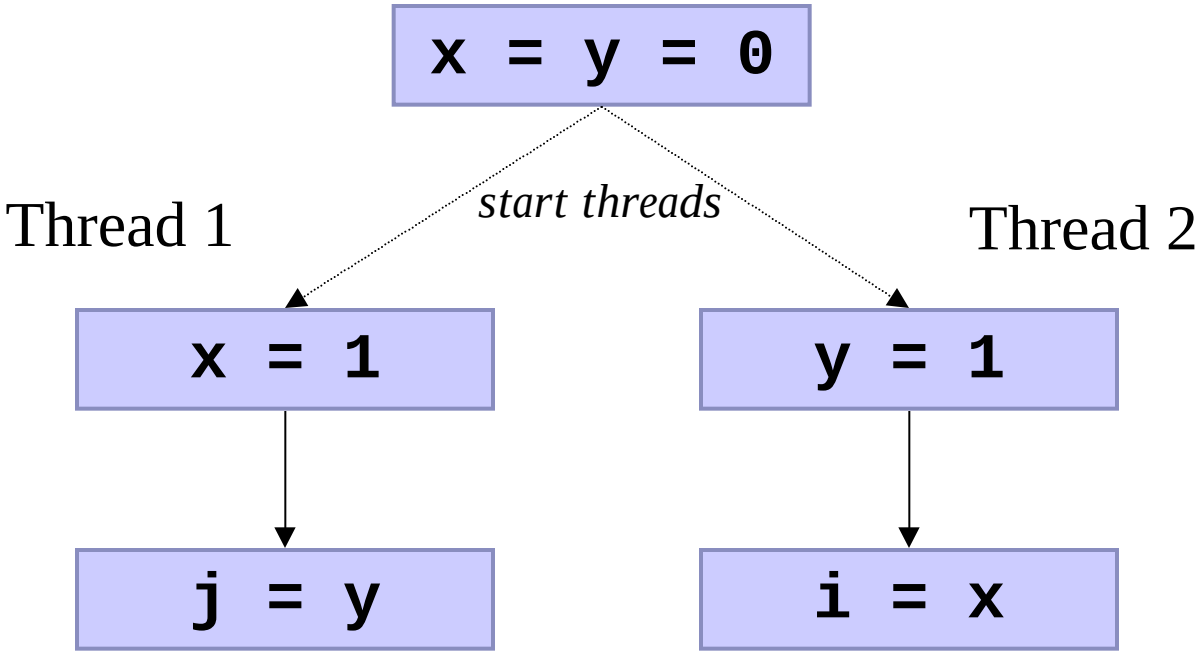
Three Aspects of Synchronization

- **Atomicity**
  - Locking to obtain mutual exclusion
- **Visibility**
  - Ensuring that changes to object fields made in one thread are seen in other threads
- **Ordering**
  - Ensuring that you aren't surprised by the order in which statements are executed

Don’t be too clever

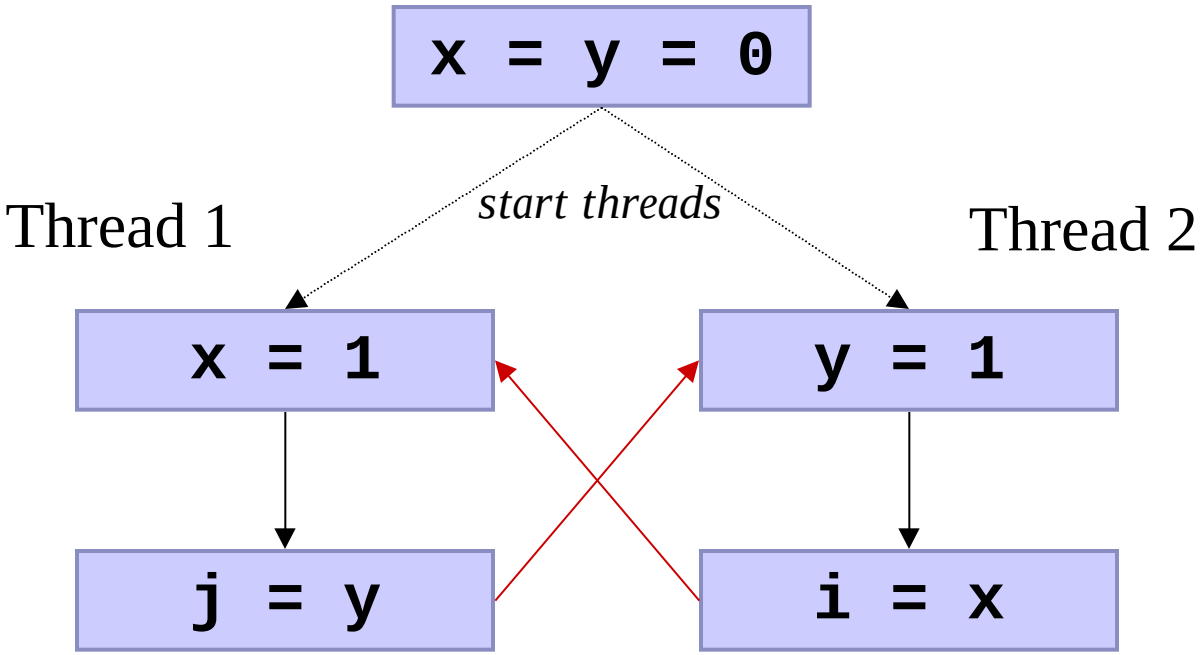
- **People worry about the cost of synchronization**
  - Try to devise schemes to communicate between threads
    - without using synchronization
- **Very difficult to do correctly**
  - Inter-thread communication without synchronization is not intuitive

Quiz time



Can this result in `i=0` and `j=0` ?

Answer: Yes!



How can i = 0 and j = 0?

How can this happen?

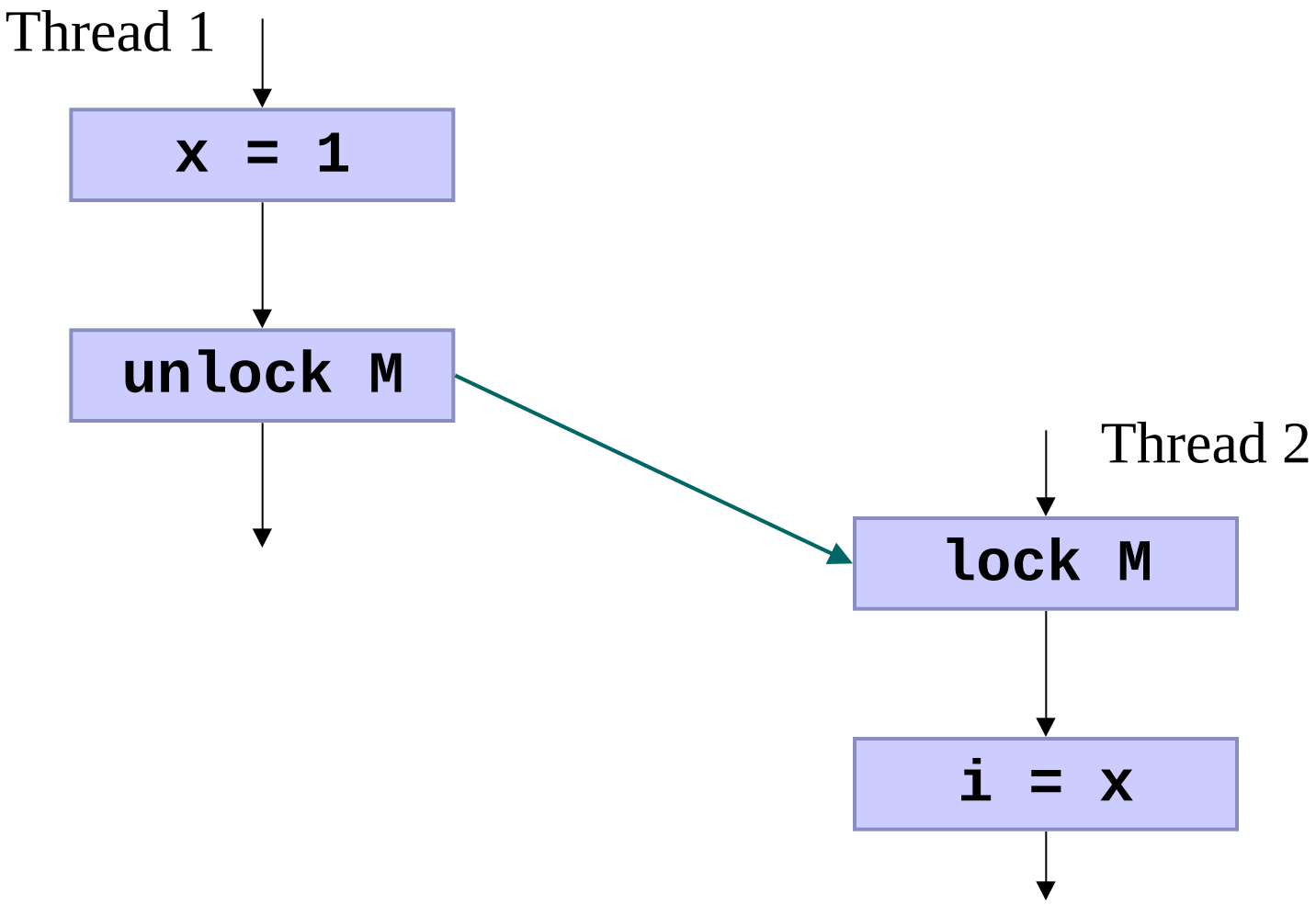
- **Compiler can reorder statements**
  - or keep values in registers
- **Processor can reorder them**
- **On multi-processor, values not synchronized in global memory**

- **Must use synchronization to enforce visibility and ordering**
  - as well as mutual exclusion

Synchronization Actions (*approximately*)

```
// block until obtain lock
synchronized(anObject) {
   // get main memory value of field1 and field2
   int x = anObject.field1;
   int y = anObject.field2;
   anObject.field3 = x+y;
   // commit value of field3 to main memory
}
// release lock
moreCode();
```

When are actions visible to other Threads?



What does volatile mean?

- **C/C++ spec**
  - ***there is no implementation independent meaning of volatile***
- **Situation a little better with Java technology**
  - volatile reads/writes guaranteed to go directly to main memory
    - can't be cached in registers or local memory
  - reads/writes of volatile longs/doubles guaranteed to be atomic
    - enforced on all JVM's?

Using Volatile

- **Volatile used to guarantee visibility of writes**
  - stop ***must*** be declared volatile
  - otherwise, compiler could keep in register

```
class Animator implements Runnable {
  private volatile boolean stop = false;
  public void stop() { stop = true; }
  public void run() {
    while (!stop)
      oneStep();
  }
  private void oneStep() { /*...*/ }
}
```

Using Volatile to Guard Other Fields Doesn't Work

- **Do not use - Does not work**

```
class Future {
   private volatile boolean ready = false;
   private Object data = null;
   public Object get() {
      if (!ready) return null;
      return data;
      }
   // only one thread may ever call put
   public void put(Object o) {
      data = o;
      ready = true;
      }
   }
```

Nobody Implements Volatile Correctly

- **Existing JMM requires sequential consistency for volatile variables**
  - In quiz example, if x and y are volatile
  - should be impossible to see i = 0 and j = 0
- **Haven't found any JVM's that enforce it**
  - Some JVM's completely ignore volatile flag

Volatile Compliance

	No Compiler Optimizations	Sequential Consistency	Atomic Longs/Doubles
Solaris JVM 1.2.2 EVM	Pass	Fail	Pass
Solaris JVM 1.2.2 Hotspot 1.0.1	Fail	Fail	Pass
Windows JVM 1.3 Hotspot Client	Fail	Fail	Fail
Windows JVM 1.3 Hotspot Server	Pass	Fail	Fail
Windows IBM JVM 1.1.8	Pass	Fail	Fail

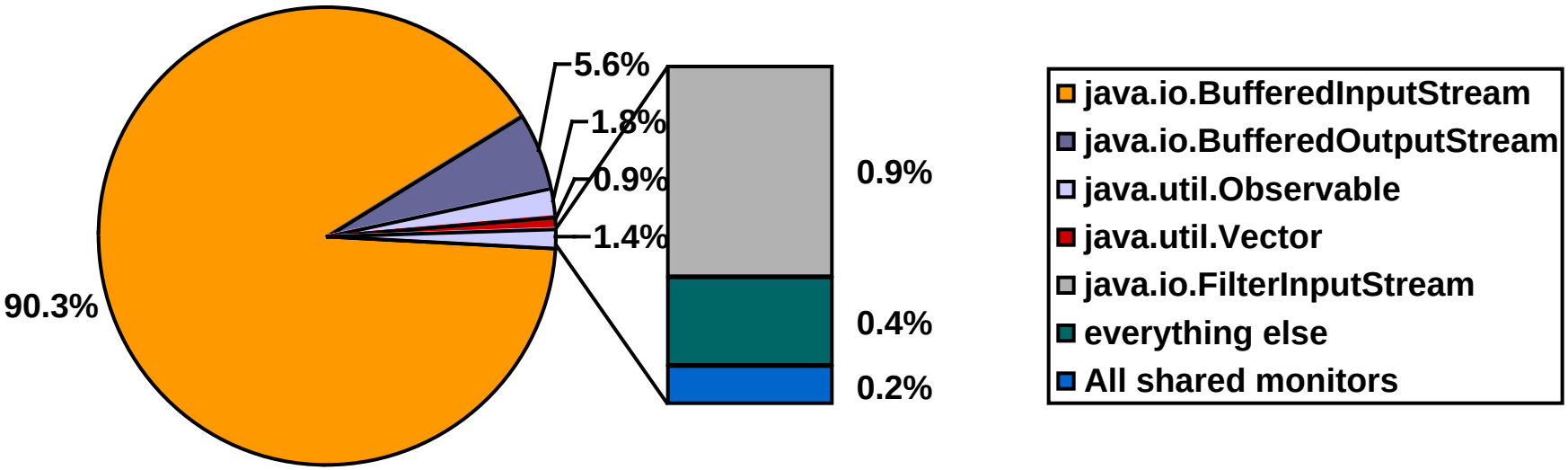
Why Use Volatile?

- **Since the semantics are implemented inconsistently**
- **Future-proof your code**
  - prohibit optimizations compilers might do in the future
- **Works well for flags**
  - more complicated uses are tricky
- **Revising the thread spec...**
  - Test compliance
  - Strengthen to make easier to use

Cost of Synchronization

- **Few good public multithreaded benchmarks**
  - See us if you want to help
- **Volano Benchmark**
  - most widely used server benchmark
  - multithreaded chat room server
  - Client performs 4.8M synchronizations
    - 8K useful
  - Server 43M synchronizations
    - 1.7M useful

Synchronization in VolanoMark Client



7,684 synchronizations on shared monitors
4,828,130 thread local synchronizations

Cost of Synchronization in VolanoMark

- **Removed synchronization of**
  - java.io.BufferedInputStream
  - java.io.BufferedOutputStream
- **Performance (2 processor Ultra 60)**
  - HotSpot
    - original: 4503
    - altered: 4828 (+7%)
  - Exact VM
    - original: 6649
    - altered: 6874 (+3%)

Most Synchronization is on Thread Local Objects

- **Synchronization on thread local object**
  - is useless
  - current spec says it isn't quite a no-op
    - but very hard to use usefully
  - revised spec may make it a no-op
- **Largely arises from using synchronized classes**
  - in places where not required

Synchronize when Needed

- **Places where threads interact**
  - need synchronization
  - need careful thought
  - need documentation
  - cost of required synchronization not significant
    - for most applications
    - no need to get tricky
- **Elsewhere, using a synchronized class can be expensive**

Synchronized classes

- **Some classes are synchronized**
  - Vector, Hashtable, Stack
  - most Input/Output Streams
- **Contrast with 1.2 Collection classes**
  - by default, not synchronized
  - can request synchronized version
- **Using synchronized classes**
  - often doesn't suffice for concurrent interaction

Synchronized Collections Aren't Always Enough

- **Transactions (DO NOT USE)**

```
ID getID(String name) {
        ID x = (ID) h.get(name);
        if (x == null) {
             x = new ID();
             h.put(name, x);
             }
        return x; }
```

- **Iterators**
  - can't modify collection while another thread is iterating through it

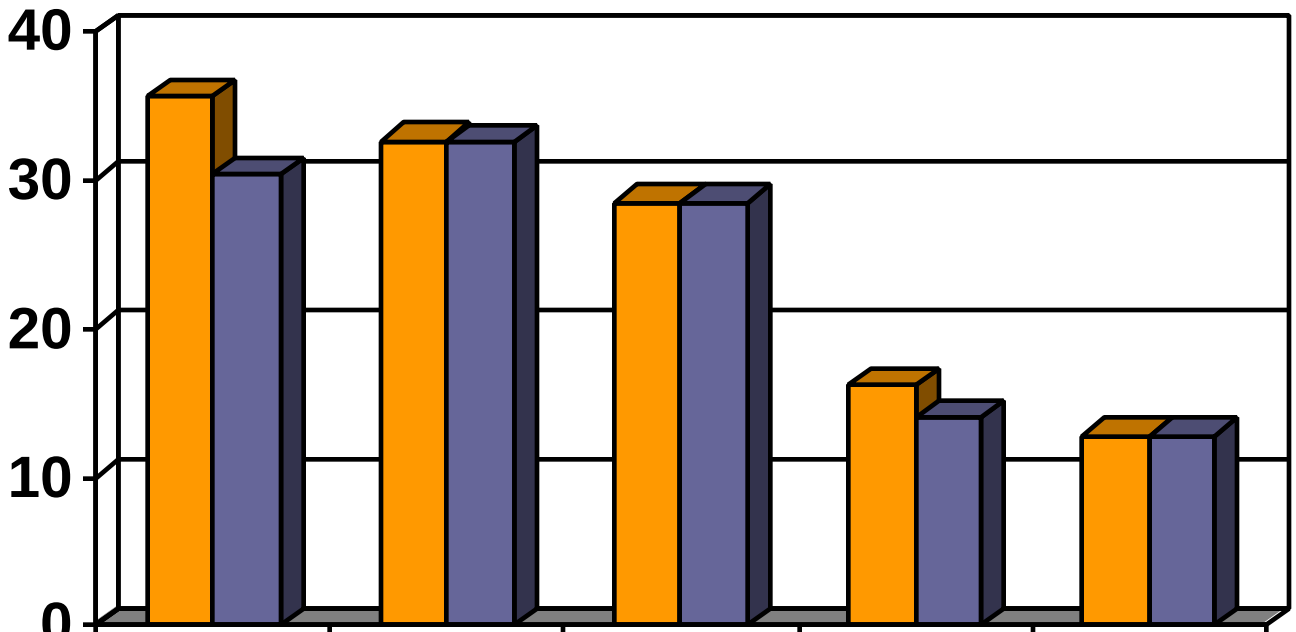
Concurrent Interactions

- **Often need entire transactions to be atomic**
  - reading and updating a Map
  - Writing a record to an OutputStream
- **OutputStreams are synchronized**
  - can have multiple threads trying to write to the same OutputStream
  - output from each thread is nondeterministicly interleaved
  - essentially useless

Cost of Synchronization in SpecJVM DB benchmark

- **Program in the Spec Java benchmark**
- **Does lots of synchronization**
  - > 53,000,000 syncs
    - 99.9% comes from use of Vector
  - Benchmark is single threaded, all of it is useless
- **Tried**
  - remove synchronizations
  - switching to ArrayList
  - improving the algorithm

Execution time of Spec JVM _209_db, Hotspot Server



	Original	Use ArrayList	Use ArrayList and other minor	Change Shell Sort to Merge Sort	All
Original	35.5	32.6	28.5	16.2	12.8
Without Syncs	30.3	32.5	28.5	14.0	12.8

Lessons

- **Synchronization cost can be substantial**
  - 10-20% for DB benchmark
  - Consider replacing all uses of Vector, Hashtable and Stack
- **Use profiling**
- **Use better algorithms!**
  - Used built-in merge sort rather than hand-coded shell sort

Designing Fast Code

- **Make it right before you make it fast**
- **Avoid synchronization**
  - Avoid sharing across threads
  - Don't lock already-protected objects
  - Use immutable fields and objects
  - Use volatile
- **Avoid contention**
  - Reduce lock scopes
  - Reduce lock durations

Isolation in Swing

- **Swing relies entirely on *Isolation***
  - AWT thread owns all Swing components
    - No other thread may access them
  - Eliminates need for locking
    - Still need care during initialization
    - Can be fragile
      - Every programmer must obey rules
  - Rules are usually easy to follow
  - Most Swing components accessed in handlers triggered within AWT thread

Accessing isolated objects

- **Need safe inter-thread communication**
  - Swing uses via runnable Event objects
    - created by some other thread
    - serviced by AWT thread

```
SwingUtilities.invokeLater(new Runnable(){
  public void run() {
    statusMessage.setText("Running");
  }});
```

GetX/SetX access methods

- **Not synchronizing access methods**
  - `int thermometer.getTemperature()`
  - (doesn't work for references)
- **Synchronizing access methods**
  - `account.getTotalBalance()`
- **Omitting access methods**
  - `queue` doesn't need `getSize()`

Things That Don't Work

- **Double-Check Idiom**
  - also, unsynchronized reads/writes of refs
- **Non-volatile flags**
- **Depending on sleep for visibility**

Initialization check - v1 - OK

Basic version:

```
class Service {
  Parser parser = null;
  public synchronized void command() {
    if (parser == null)
      parser = new Parser(...);
    doCommand(parser.parse(...));
   }
  // ...
}
```

Initialization checks - v2 - OK

Isolate check:

```
class ServiceV2 {
  Parser parser = null;
  synchronized Parser getParser() {
    if (parser == null)
      parser = new Parser();
    return parser;
  }
  public void command(...) {
    doCommand(getParser().parse(...));
    }}
```

Single-check - DO NOT USE

Try to do it without synchronization:

```
class ServiceV3 { // DO NOT USE
  Parser parser = null;
  Parser getParser() {
     if (parser == null)
          parser = new Parser();
     return parser;
  }}
```

Double-check - DO NOT USE

Try to minimize likelihood of synch:

```
class ServiceV4 { // DO NOT USE
  Parser parser = null;
  Parser getParser() {
    if (parser == null)
      synchronized(this) {
        if (parser == null)
          parser = new Parser();
      }
    return parser;
  }}
```

Problems with double-check

- **Can reorder**
  - initialization of Parser object
  - store into parser field
- **…among other reasons**
  - see JMM web page for gory details
- **Can go wrong uniprocessors**
- **Using volatile doesn't help**
  - under current JMM

Alternatives to Double–Check

- **Eagerly initialize**
  - especially for Singletons
  - especially if ref can be final
- **If static, put in separate class**
  - first use forces initialization
  - later uses guaranteed to see initialization
  - no explicit check needed
- **Double check OK for primitive values**
  - hashCode caching
  - (still technically a data race)

Unsynchronized Reads/Writes of References

- **Beware of unsynchronized getX/setX methods that return a reference**
  - same problems as double check
  - doesn't help to synchronize only setX

```
private Color color;
void setColor(int rgb) {
     color = new Color(rgb);
     }
Color getColor() {
     return color;
     }
```

Thread Termination in Sun's Demo Applets

```
Thread blinker = null;
public void start() {
        blinker = new Thread(this);
        blinker.start();
    }

    public void stop() {
        blinker = null;}

    public void run() {
       Thread me = Thread.currentThread();
       while (blinker == me) {
           try {Thread.currentThread().sleep(delay);}
           catch (InterruptedException e) {}
           repaint();
        }
    }
```

unsynchronized access to blinker field

confusing but not wrong: sleep is a static method

Problems

- **Don’t assume another thread will see your writes**
  - just because you did them
- **Calling sleep doesn’t guarantee you see changes made while you slept**
  - Nothing to force thread that called stop to push change out of registers/cache

Wrap-up

- **Cost of synchronization operations can be significant**
  - but cost of ***needed*** synchronization rarely is
- **Thread interaction needs careful thought**
  - but not too clever
- **Need for synchronization...**

Wrapup - Synchronization

- **Communication between threads**
  - requires both threads to synchronize
    - or communicate through volatile fields
- **Synchronizing everything**
  - is rarely necessary
  - can be expensive (5%-20% overhead)
  - may lead to deadlock
  - may not provide enough synchronization
    - e.g., transactions

JavaOne℠

Sun's 2000 Worldwide Java Developer Conference℠