Proposed Informal Semantics for Programmers

William Pugh

Dept. Of Computer Science
Univ. of Maryland

http://www.cs.umd.edu/~pugh/java

Reality Check

Realities

- Not going to change Java threading model
  - even if people don't like it
- Have to keep in mind that most Java programmers haven't taken an OS course
  - Can't hold them to high standards
- Incorrectly synchronized programs must have a (safe) meaning
  - can't allow a cracker to use improperly synchronized code to attack a system.

Goals for new Memory Model

- Preserve existing and/or necessary safety guarantees
  - even in the presence of data races
- Have a clear specification we can reason about
- Allow efficient immutable classes
- New MM should not break “reasonable” existing code

Goals for new MM (continued)

- In code that doesn't involve locks or volatile variables, use as much as possible of the standard compiler optimization techniques
- Data-race-free programs should be guaranteed sequentially consistent results
  - Constraints not necessary to ensure SC for data-race-free programs should be imposed with "care and deliberation".

Proposed Changes

- Make it clear
- Allow standard compiler optimizations
- Remove corner cases of synchronization
  - enable additional compiler optimizations
- Strengthen volatile
  - make easier to use
- Strengthen final
  - Enable compiler optimizations
  - Fix security concerns

VM Safety Guarantees

Safety Guarantees

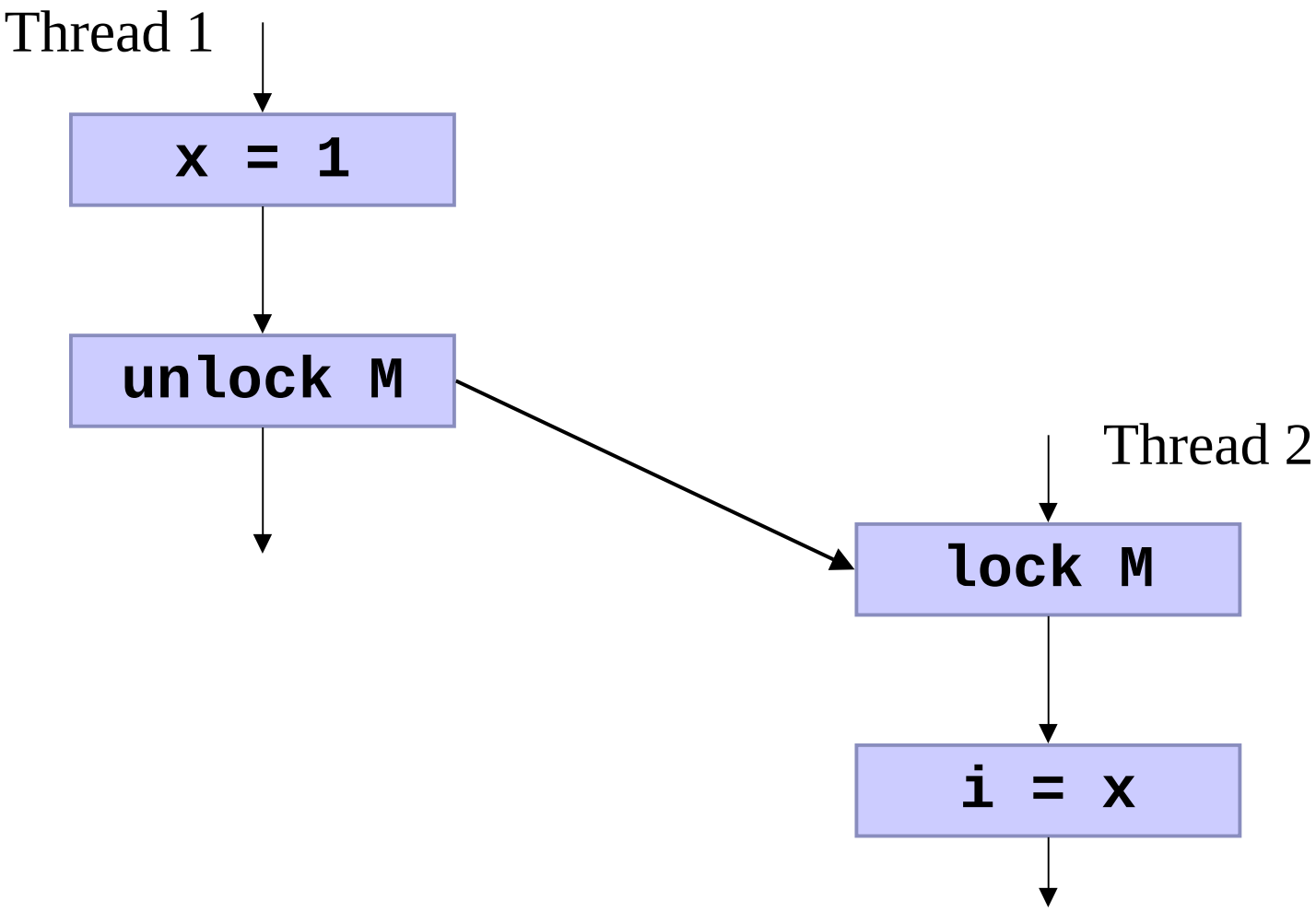
- For reads of fields and arrays
  - type safety
  - not-out-of-thin-air safety
- VM safety - despite lack of synchronization
  - All operations other than reading a field or array are as usual
    - can't crash/violate VM
    - No new exceptions
    - array length is always correct

synchronization

Synchronization

- Synchronization on thread local objects
  - e.g., synchronized(new Object()) {}
  - is not a no-op under current semantics
  - but it isn't a memory barrier
- Proposal: make it a no-op
  - and allow other compiler optimizations
- Programming model is release consistency

When Are Actions Visible to Other Threads?



Synchronization semantics

- Once you acquire a monitor
  - you are properly synchronized with regards to all actions previous to a previous release of the same monitor by any thread
  - must be same monitor

Notes

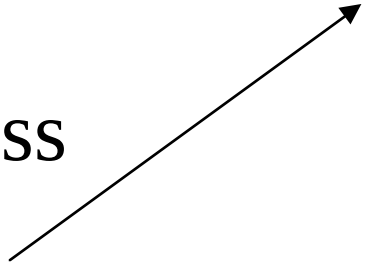
- Synchronization blocks can be expanded
  - Lock operations can be moved upwards
  - Unlock operations downwards
- Consider

```
synchronized (A.class) {
        A tmp = new A();
        }
A.foo = tmp;
```

Example

Lock A.class
tmp = new A
tmp.<init>()
unlock A.class
A.foo = tmp

Lock A.class
tmp = new A
A.foo = tmp
tmp.<init>()
unlock A.class



New Optimizations Allowed

- Turning synchronizations into no-ops
  - locks on objects that aren't ever locked by any other threads
  - reentrant locks
  - enclosed locks
- Lock coarsening
  - merging two calls to synchronized methods on same object
    - need to be careful about starvation issues

Volatile

Existing Semantics of Volatile

- No compiler optimizations
  - Can't hoist read out of loop
  - reads/writes go directly to memory
- Reads/writes of volatile are sequentially consistent
  - can not be reordered
  - but access to volatile and non-volatile variables can be reordered
- Reads/writes of long/doubles are atomic

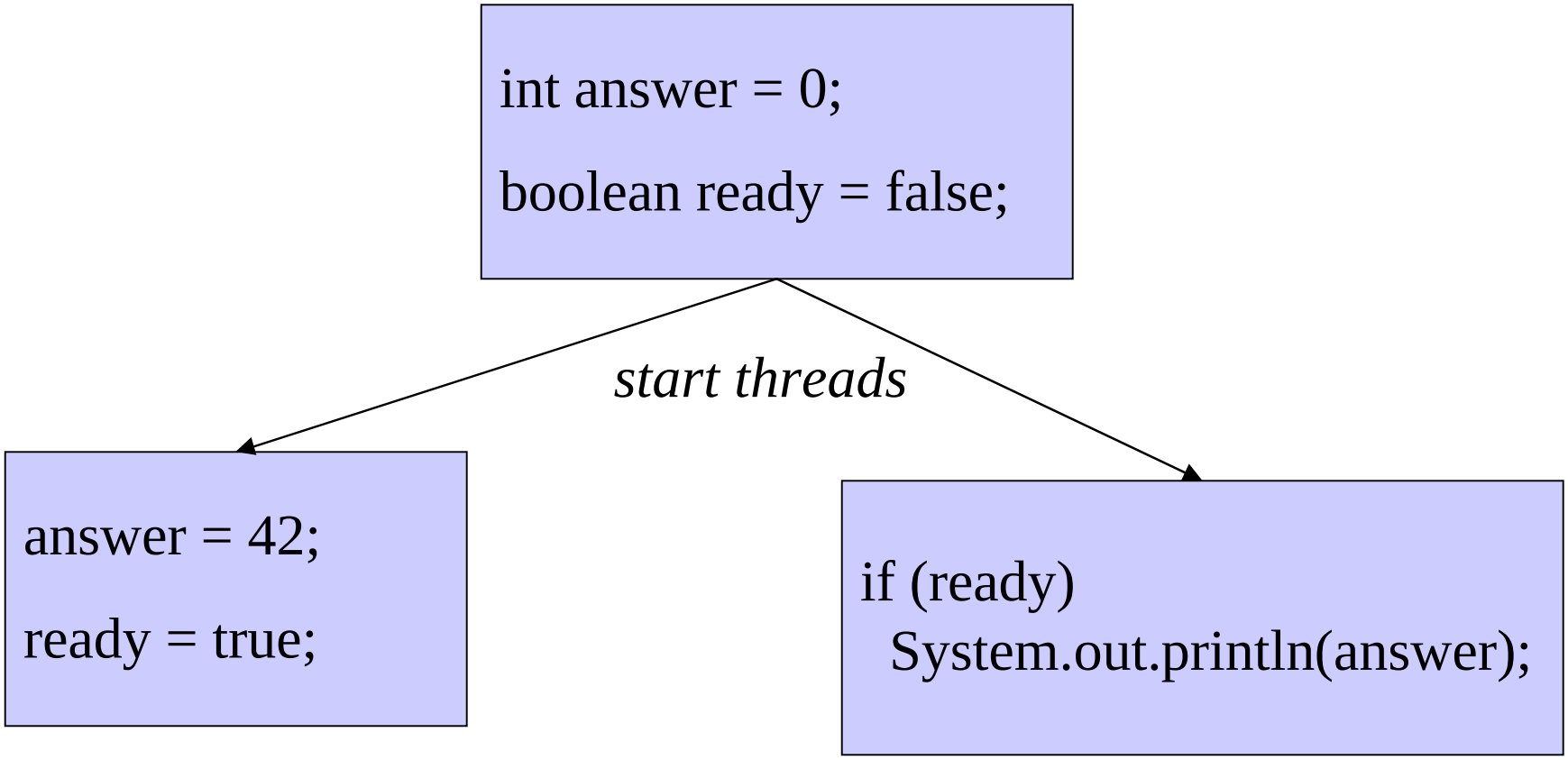
Existing Volatile Compliance

- Very poor
  - some JVMs completely ignore volatile
- No one enforces sequential consistency
- Atomic longs/doubles isn't enforced on most

Volatile Compliance

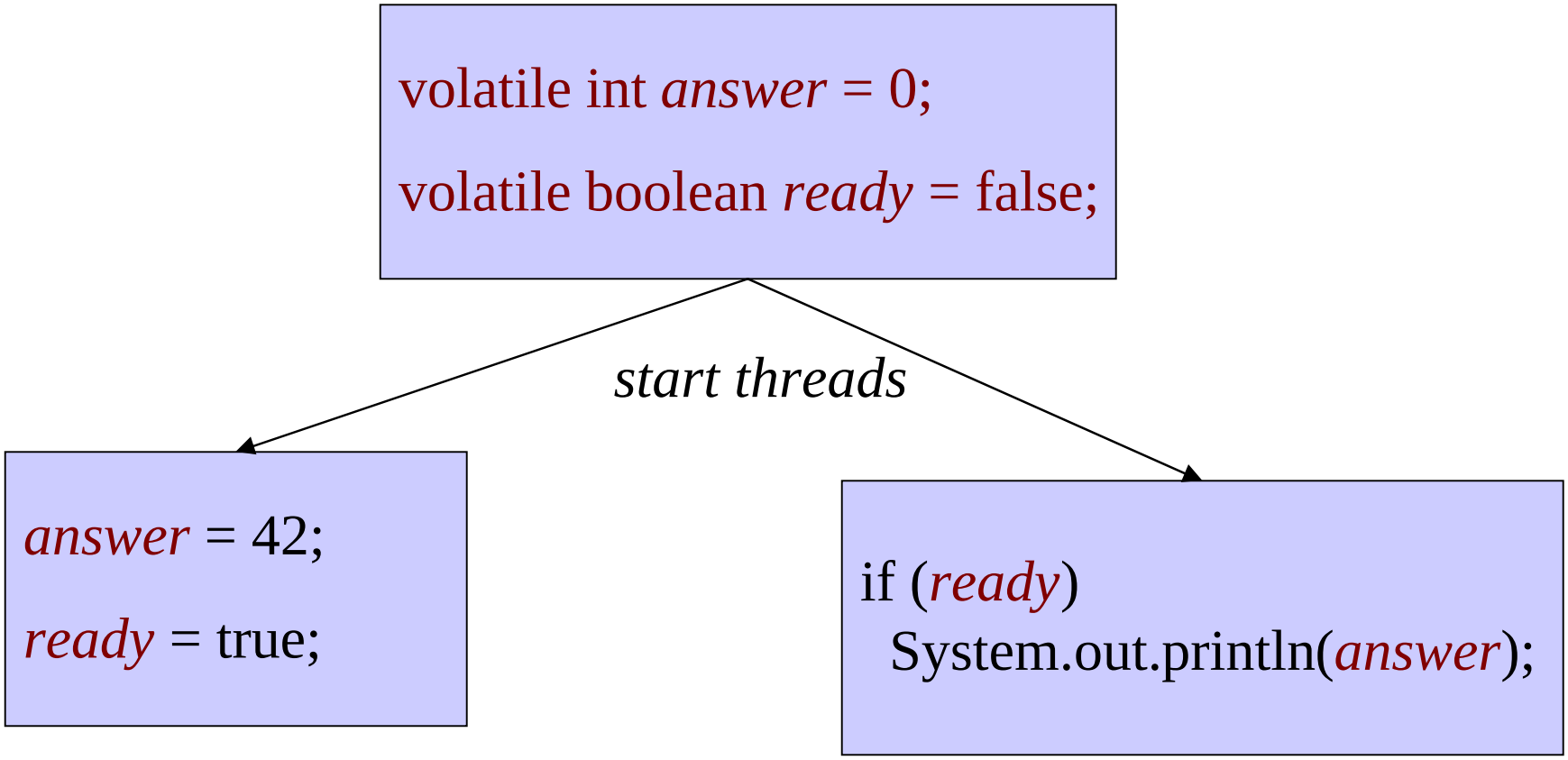
	No Compiler Optimizations	Sequential Consistency	Atomic Longs/Doubles
Solaris JDK 1.2.2 EVM	Pass	Fail	Pass
Solaris JDK 1.3.0 beta Hotspot Client	Fail	Fail	Fail
Windows JDK 1.3.0 Hotspot Client	Fail	Fail	Fail
Solaris JDK 1.3.0 beta Hotspot Server	Pass	Fail	Fail
Windows JDK 1.3.0 Hotspot Server	Pass	Fail	Fail
Windows IBM JDK 1.1.8	Pass	Fail	Fail

Need for volatile



Can print 0

Need for volatile

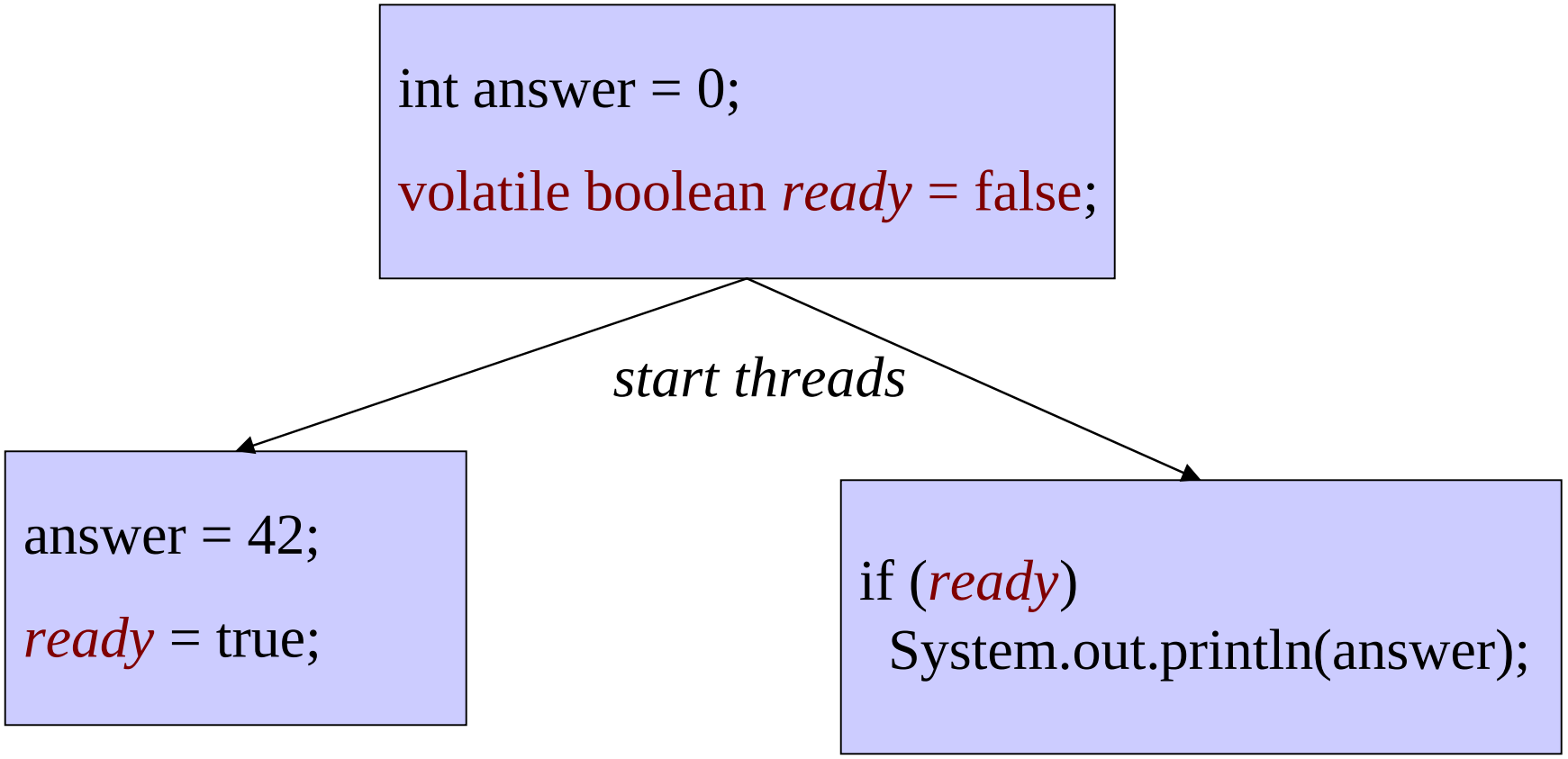


Must *not* print 0

Proposed New Semantics for Volatile

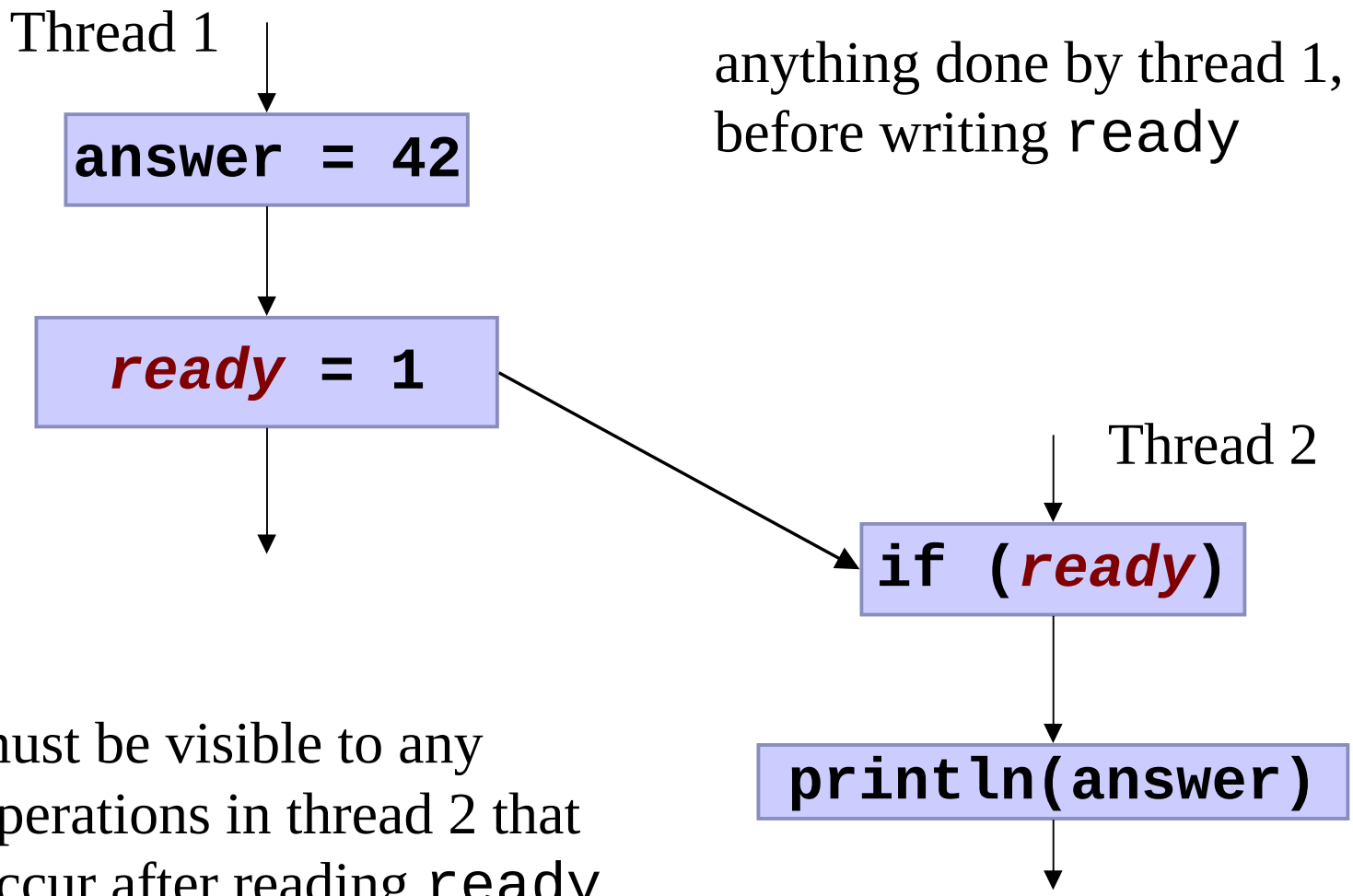
- Write to a volatile acts as a release
- Read of a volatile acts as an acquire
- If a thread reads a volatile
  - all writes done by any other thread,
  - before earlier writes to the same volatile,
  - are guaranteed to be visible

New semantics for volatile



Existing semantics: can print 0
Proposed semantics: must not print 0

When Are Actions Visible to Other Threads?



Naïve Implementation of Volatile

- On SMP with weak memory model (Alpha)
  - Membar before & after each volatile write
  - Membar after each volatile read
- On SMP with TSO (e.g. Sparc)
  - Membar after each volatile write
- On IA-64
  - use ld.acq and st.rel for volatile fields
  - also, memory barrier after each volatile write

Implementation Cost of Proposed Change in Semantics

- Naïve implementation handles new semantics
  - unclear if only enforcing only existing semantics would incur fewer memory barriers
- New semantics will prohibit some compiler optimizations
  - reading a volatile will force all values cached in registers to be reloaded

Reordering of Volatiles

- Can reorder
  - normal access and following volatile read
  - volatile write and following normal access
- Volatiles are not quite a data-race-free-0 sync access

Volatile Summary

- Few programmers will use all these features
  - Do we really need sequential consistency, on top of acquire/release semantics?
- But it is simple and easy to understand
  - more likely to be used correctly

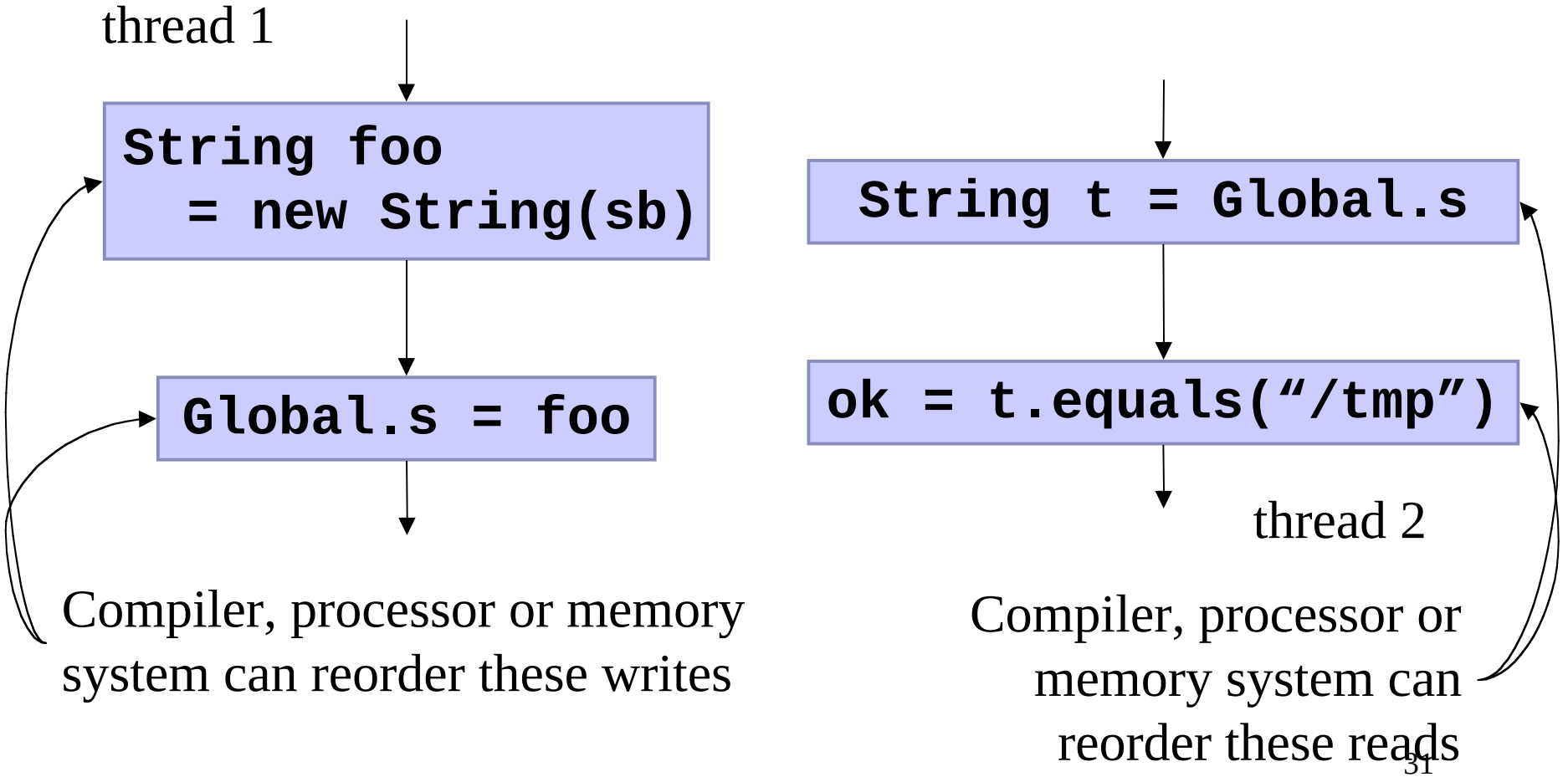
Immutable Objects and final fields

Immutable Objects

- Many Java classes represent immutable objects
  - e.g., String
- Creates many serious security holes if Strings are not truly immutable
  - probably other classes as well
  - should do this in String implementation, rather than in all uses of String

Strings aren't immutable

just because thread 2 sees new value for **`Global.s`** doesn't mean it sees all writes done by thread 1 before store to **`Global.s`**



Why aren't Strings immutable?

- A String object is initialized to have default values for its fields
- *then* the fields are set in the constructor
- Thread 1 could create a String object
- pass it to Thread 2
- which calls a sensitive routine
- which sees the fields change from their default values to their final values

Making String immutable

- Could make String methods synchronized
  - most programmers don't think methods for immutable objects need to be synchronized
  - synchronization would slow down String methods on all platforms
    - only needs to be synchronized on SMP's with weak memory models
    - doesn't need synchronization on SPARC, Intel or MAJC SMP's

Need for syncs in immutable objects

- No issues arise if proper synchronization is used when passing a reference to an immutable object between threads
  - no synchronization, final fields, or anything else needs to be done to get true immutability
  - issues only arise when references are passed via a data race

Are immutable objects like an int?

- Reads/writes of an int are atomic
  - can be done without synchronization
  - you get old value or new value
- Are reads/writes of references to immutable objects atomic
  - should we encourage a programming style in which they are written without synchronization

Final = Immutable?

- Existing Java memory model doesn't mention final
  - no special semantics
- Would be nice if compiler could treat final fields as constant
  - Don't have to reload at memory barrier
  - Don't have to reload over unknown function call

Existing semantics require that final fields need to be reloaded at synchronization points

```
class A extends Thread
  {
    final int x;
    A() {

  synchronized(this) {
        start();
        sleep(10);
        }
      x = 42;
      };
```

```
public void run() {
    int i,j;
    i = x;
    synchronized(this) {
        j = x;
        }
    System.out.println(i+j);
    }
}
```

Must *not* print 0

Proposed Semantics for Final

- Read of a final field always sees the value set in constructor
  - If,
    - a final field is read before set
      - (by the constructing thread)
    - or, a reference to the object becomes visible to another thread before object is constructed
    - semantics are ugly
- Can assume final fields never change
- Makes string immutable?

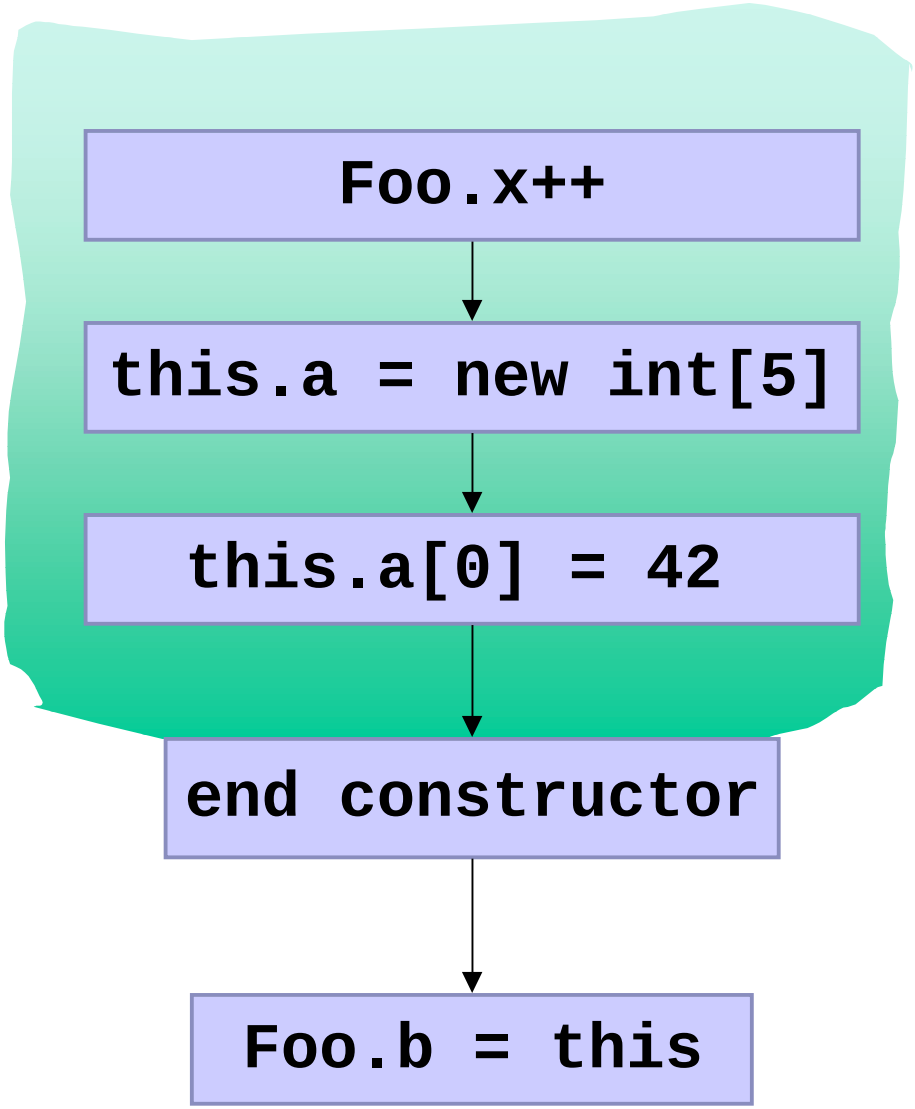
Problems

- JNI code can change final fields
  - setIn, setOut, setErr
  - **Propose to remove this ability**
  - (reflection appears to be safe)
- Objects that escape their constructor before final fields are set
  - Base class “registers” object, derived class has final fields
- Doesn’t suffice to make strings immutable

Doesn’t make Strings immutable

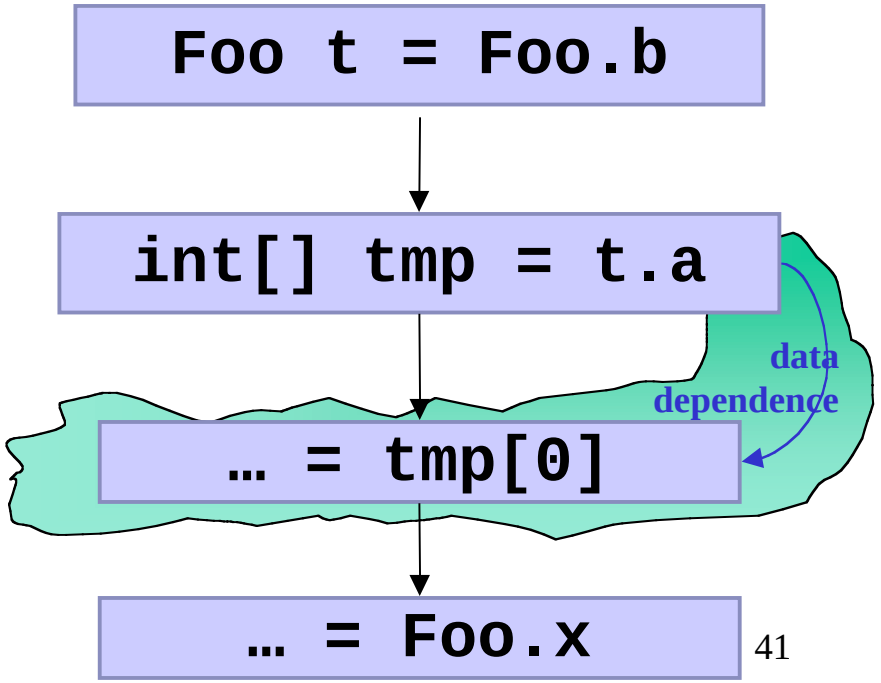
- No way for elements of an array to be final
- For Strings, have to see final values for elements of character array
- So…
  - Read of final field is treated as a weak acquire
    - matching a release done when object is constructed
  - weak in that it only effects things dependent on value read
    - no compiler impact

Visibility enforced by final field a

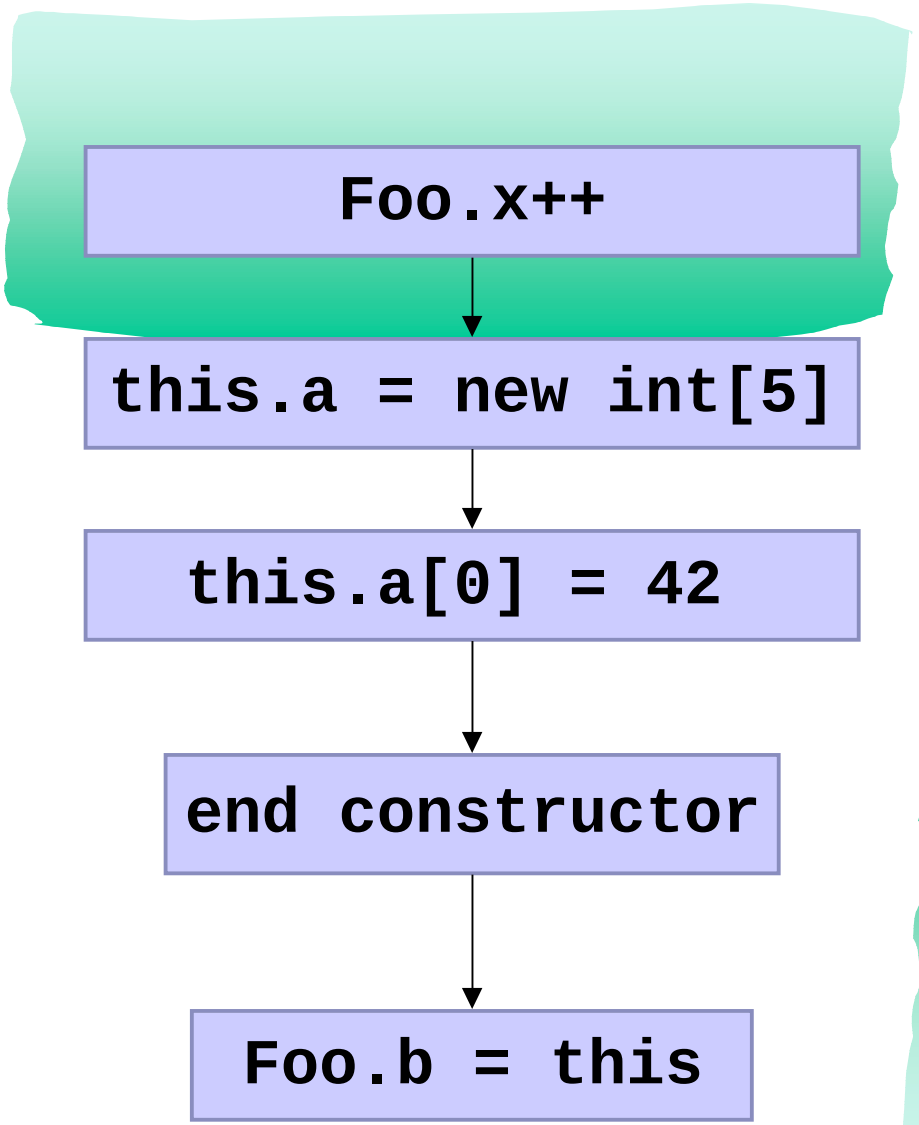


All actions done before completion of constructor

must be visible to any action that is data dependent on the read of a final field set in that constructor

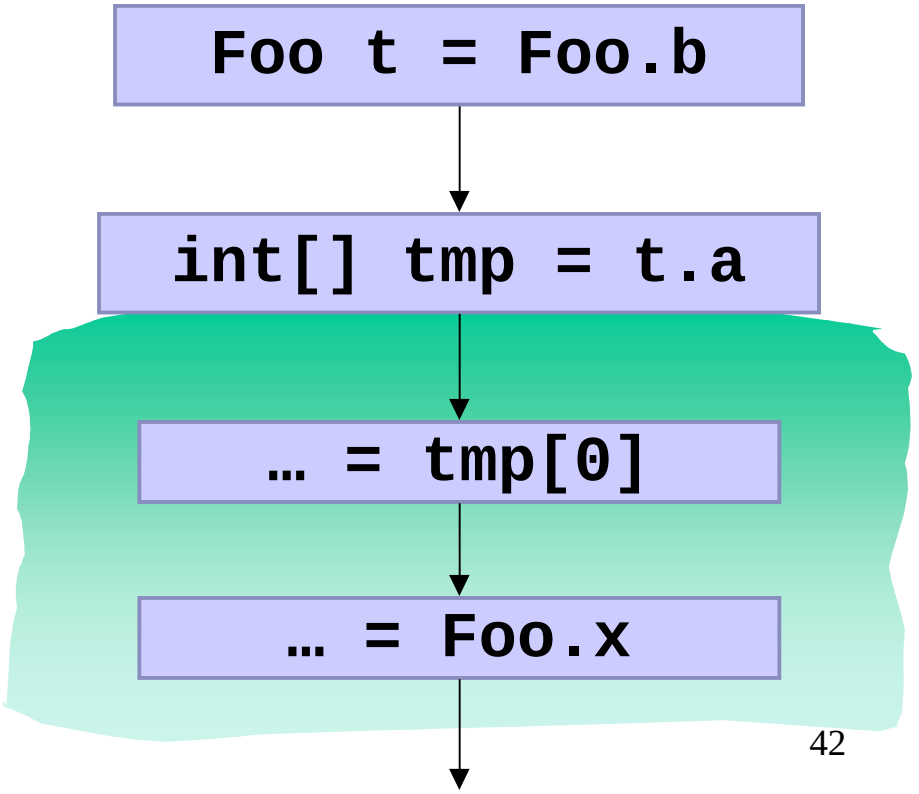


Contrast with volatile

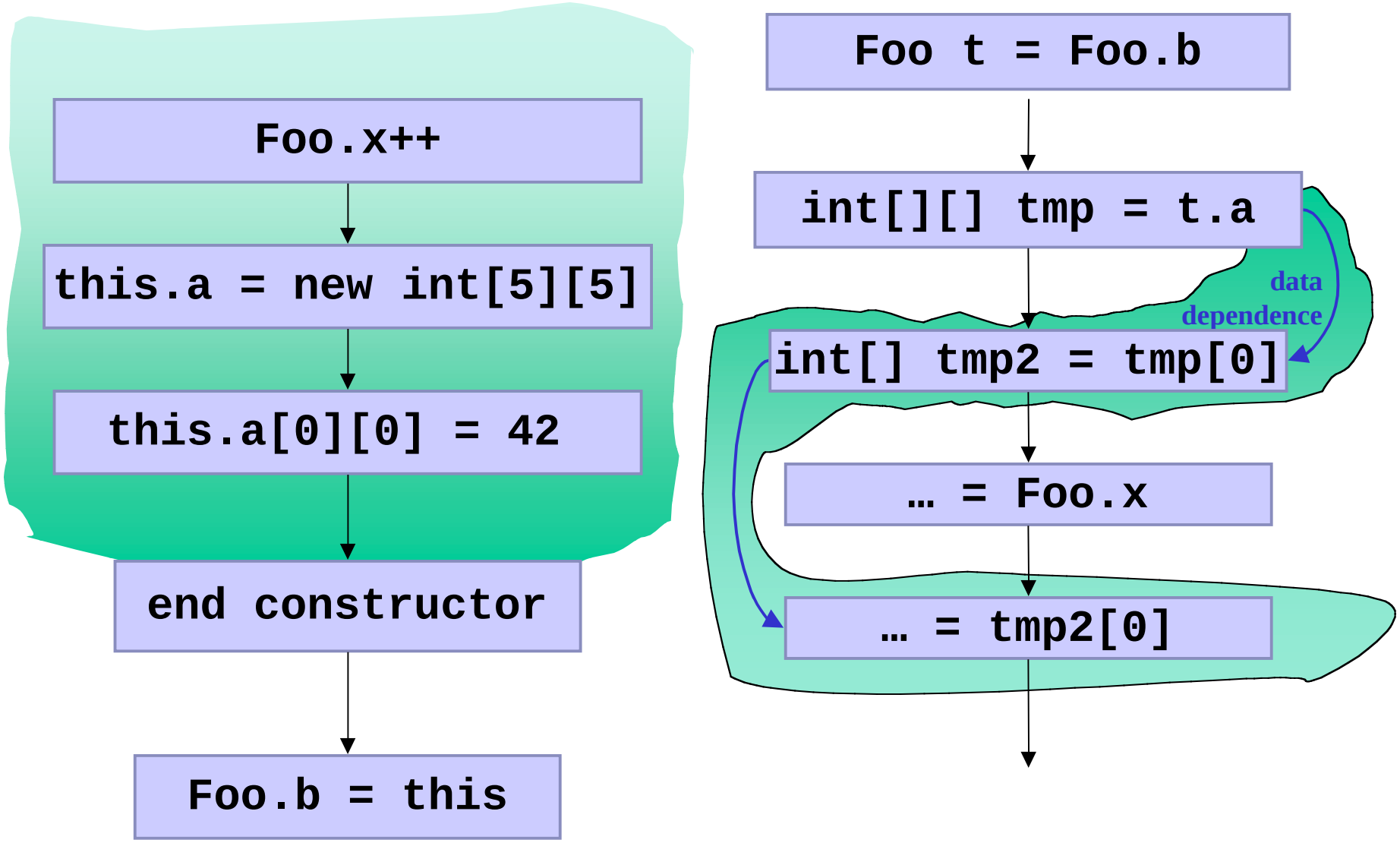


Actions done before assignment to volatile field

must be visible to any action after the read



Data dependence is transitive



Other semantic issues

Thread Communication

- All forms of inter-thread communication force writes to be visible
  - interrupt
  - start/join
  - isAlive
- Sleep and yield have no effect on visibility
  - will cause problems for broken programs
  - but difficult/impossible to specify semantics of visibility for sleep

finalization

- Loosing the last reference to an object is an asynchronous signal to another thread to run the finalizer
  - which writes are visible to the finalizer?
- Proposal: only writes to the object being finalized and writes done during construction
  - need synchronization to see other writes
- Unsynchronized finalizers are dubious

Serialization and Beans

- Serialization constructs objects
  - can’t set final fields in readObject or readExternal