Advanced concurrent programming in Java
Shared objects

Mehmet Ali Arslan

21.10.13

To see(m) or not to see(m)...

There is more to synchronization than just atomicity or critical sessions.

Memory visibility... Updates by one thread to a shared objects state must be visible to the others.

Without proper synchronization, reordering can mess up the view.

- Stale data: out-of-date value

To see(m) or not to see(m)...

There is more to synchronization than just atomicity or critical sessions.
Memory visibility... Updates by one thread to a shared objects state must be visible to the others.
Without proper synchronization, reordering can mess up the view.

- Stale data: out-of-date value

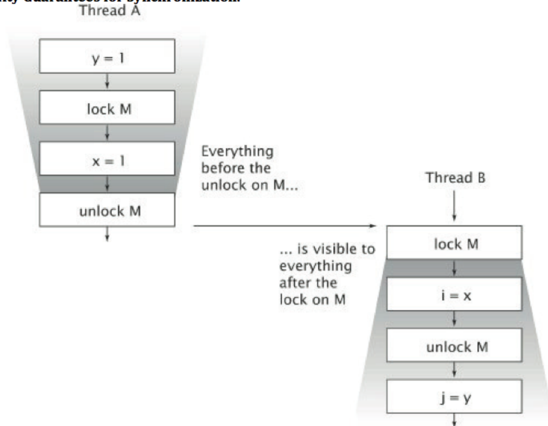
`synchronized` and visibility

We can use intrinsic locks to ensure correct visibility.

`synchronized` and visibility

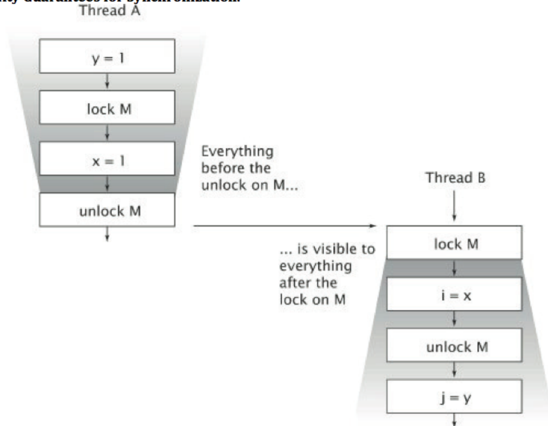
We can use intrinsic locks to ensure correct visibility.

Figure 3.1. Visibility Guarantees for Synchronization.



synchronized and visibility

We can use intrinsic locks to ensure correct visibility.



...acts like a barrier.

`volatile`

- Weaker form of synch.
- To compiler and runtime: "Do not reorder with other memory ops!"
- "...a read of a volatile variable always returns the most recent write by any thread."
- No locking → lighter than `synchronized`
- Does not guarantee atomicity!

Use only when:

- writes don't depend on the current value or only a single thread ever updates.
- the variable does not participate in invariants with other state vars.
- locking is not required for any other reason

`volatile`

- Weaker form of synch.
- To compiler and runtime: ”Do not reorder with other memory ops!”
- ”...a read of a volatile variable always returns the most recent write by any thread.”
- No locking → lighter than `synchronized`
- Does not guarantee atomicity!

Use only when:

- writes don’t depend on the current value or only a single thread ever updates.
- the variable does not participate in invariants with other state vars.
- locking is not required for any other reason

Definitions

Making an object available out of its current scope is called **publishing** it. Examples of publication:

- `public`
- any objects referred to as non-private fields of a published object
- an object passed to an *alien method* i.e. a method whose behavior is not fully specified by the respective object (includes its overrideable methods as well).

An object that is published when it shouldn't have been is **escaped.**

Definitions

Making an object available out of its current scope is called **publishing** it. Examples of publication:

- `public`
- any objects referred to as non-private fields of a published object
- an object passed to an *alien method* i.e. a method whose behavior is not fully specified by the respective object (includes its overrideable methods as well).

An object that is published when it shouldn't have been is **escaped**.

Escape under construction/Safe construction

An object is in a consistent state only after its constructor returns. Publication before that is hazardous.

Some examples that would lead `this` reference to escape:

- starting a thread in the constructor
- calling an overrideable instance method in the constructor that is neither private nor final

Escape under construction/Safe construction

An object is in a consistent state only after its constructor returns.
Publication before that is hazardous.
Some examples that would lead `this` reference to escape:

- starting a thread in the constructor
- calling an overrideable instance method in the constructor that is neither private nor final

To share or not to share... - No publication

When an object is confined to a thread, safety is guaranteed. Even if the object itself is not thread-safe. Programmer is responsible to ensure that the confined objects do not escape from the thread.

- Ad-hoc - no language feature is used. Often used for implementing a single-threaded subsystem.
- Stack - confine objects as local variables
- `ThreadLocal` - every thread gets its own value-holding object, not shared with others.

No mutable, no cry

State cannot be changed after construction = **immutable**
Always thread-safe. No worries about publishing.
Two more conditions for an object to be immutable:

- all fields are `final`
- properly constructed (no escape under construction)

Safe vs. improper publication

- A publication is **safe** when the published object is correctly visible at publication time - regards initialization of the object.
- Both the reference of the object and the object's state must be published at the same time.
- Even if the object itself is thread-safe, if the reference to it is published without sufficient synch., this will cause visibility problems thus, **improper** publication.
- JavaMemory Model guarantees *initialization safety* for immutables.

How to publish a properly constructed object

Properly constructed - no escape in constructor
Some safe publication methods:

- Init the reference from a `static` initializer - safety guaranteed by JVM
- Store a reference into a `volatile` field or `AtomicReference`
- Store a reference to it in a `final` field of another properly constructed object
- Store a reference to it in a field that is guarded by a lock

Sharing objects safely

Safe publication ensures only the visibility of the as-published state $\rightarrow$ synch. is necessary for every access to shared mutable objects.

"Rules of engagement": when publishing an object, document how it can be accessed-regarding mutability, synch. methods, etc.

Some common policies for sharing objects:

- Thread-confined: no thread interaction for the respective object
- Shared read-only: immutable and effectively immutable objects
- Shared thread-safe: object itself is responsible
- Guarded: can be accessed only with a specific lock held

Sharing objects safely

Safe publication ensures only the visibility of the as-published state $\rightarrow$ synch. is necessary for every access to shared mutable objects.

"Rules of engagement": when publishing an object, document how it can be accessed-regarding mutability, synch. methods, etc.

Some common policies for sharing objects:

- Thread-confined: no thread interaction for the respective object
- Shared read-only: immutable and effectively immutable objects
- Shared thread-safe: object itself is responsible
- Guarded: can be accessed only with a specific lock held

Exercise 1

```
//QUESTION 1: Is the class still immutable? If not, why?
//QUESTION 2: Assuming that we don't want to publish any fields of ThreeStooges,
//            is there an escape we should be worried about?
public final class ThreeStooges {
    private final Set<String> stooges;

    public ThreeStooges() {
        stooges = new HashSet<String>();
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public ThreeStooges(String first, String second, String third, HashSet<String> set){
        set.add(first);
        set.add(second);
        set.add(third);
        this.stooges = set;
    }


    public boolean isStooge(String name) {
        return stooges.contains(name);
    }

}
```

Exercise 2

```
//QUESTION : Is it possible to make this class thread safe
//           using the immutable holder class scheme
//           used in section 3.4.2 in the book?
public final class HungryThreeStooges {
    private final String[] stooges = {"Moe", "Larry", "Curly"};
    private int numberOfSteaks=10;
    private int turn=0;
    static HungryThreeStooges instance = new HungryThreeStooges();
    public String feedStooge(){
        if (numberOfSteaks<1)
            return "Damn stooges ate everything!";
        else{
            String stooge = stooges[turn % stooges.length];
            turn++;
            numberOfSteaks--;
            return stooge;
        }
    }
}
```

Thanks for the attention!