Thread safety

A class is thread safe if it **behaves correctly when accessed from multiple threads**, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Where is the problem?

Informally, **an object's state** is its data, stored in state variables such as instance or static fields. An object's state may include fields from other, dependent objects; HashMap's state is partially stored in the HashMap object itself, but also in many Map.Entry objects. An object's state encompasses any data hat can affect its externally visible behavior.

- Atomicity
  Operations **A and B are atomic** with respect to each other if, from the perspective of a thread executing A, when another thread executes B,either all of B has executed or none of it has. An **atomic operation** is one that is atomic with respect to all operations, including itself, that operate on the same state.
  Race conditions:
  Read-modify-write
  Check-then-act
  Compound actions
  Non-atomic 64-bit operations
- Visibility
  Stale data (like "read_uncommited", caches?)
  Reordering
  Non-atomic 64-bit operations

Where is the problem?

Is an object thread safe?

State vs Sharing	Not shared	Shared
Stateless	Yes	Yes
Immutable	Yes	Yes
Mutable	Yes	**No**

What should I do?

- Make stateless
- Stop sharing, any of:
  Not sharing at all
  Use confinement, any of:
  * Thread
  * Stack
  * Ad-hoc

- If shared
  3 types of objects
  * immutable
  * effectively immutable
  * mutable
  What must be done
  * proper construction
  * safe publication
  * syncrhonization (mutable)

Some definitions

Immutable object:
* state can't be modified after construction
* all fields are final (guarantees **safe publication** — consider it's special JMM semantics)
* **properly constructed**

Effectively immutable object — read-only

Avoding the **state escape**:

- **Properly constructing** — an object is properly constructed if *this* reference does not escape during construction. Examples:
  this reference explicitly escapes — used to register instance as a listener
  this reference implicitly escapes — inner class instance using *this* class methods is passed
- **Publishing safely**: both the reference to the object and the object's state are made visible to other threads at the same time. A properly constructed object can be safely published by:
  initializing an object reference from a static initializer
  storing a reference to it into a *volatile field*
  storing a reference to it into an *AtomicReference*
  storing a reference to it into a *final field of a properly constructed object*
  storing a reference to it into a *field that is properly guarded by a lock*

Tools at your disposal

- synchronization, locks — atomicity, visibility
- final — reference to the owning object is guaranteed to be visible **after** the field is initialized
- volatile
  not reordered operations, not cached, visible to every thread
  extended sematntics: write — enter synchronized block, reading — exiting synchronized block
  can points listed above be grouped as "volatile is visibility semantics of syncrhonized block"?