Principle 1

- If more than one thread accesses a given state variable and one of them might write to it then
- *All* accesses to the variable must be correctly coordinated using some synchronization mechanism
- If this rule is not followed then the program is broken!
- What is “correct” depends on the chosen mechanism

Making Individual Objects Thread-safe

- What is thread-safety?
- Invariants
- Encapsulation
- Immutability

Def. 1: Class thread-safety

- **An class is *thread-safe* if it behaves correctly regardless of scheduling decisions and without any synchronization or coordination by its client code.**

Invariants

- An invariant is a property of an object that holds except when its state is being actively modified
  - Sorted
  - Balanced
  - Exists a path to every element
  - tree.count == # of elements stored in the tree

Invariants (2)

- An invariant is a relationship between the values of state variables

- Stateless objects are always thread-safe (no synchronization is required)
  - What is required to be called “stateless”?

Encapsulation

- The idea that maintaining invariants is the job of an object's code, not the job of its callers
- Thread-safe classes include any required synchronization code

Atomic/Atomicity

- An operation is ***atomic*** iff it is indivisible
  - Concurrent operations see no intermediate states
  - Concurrent operations affect no intermediate states
- Compound actions are the opposite of atomicity
  - Read-modify-write
  - Check-then-act
- Compound actions can be made atomic using synchronization
- java.util.concurrent.atomic classes

Principle 2

- To preserve state consistency (i.e. invariants) update related state variables in a single atomic operation
  - Variables are related if they contribute to operands of a relation in an invariant: <, = , >, element-of, etc.
    - Example invariant: product(lastFactors)==lastNumber (Fig. 2.5)

Achieving atomicity using Java intrinsic locks

- synchronized(object) { … }
- The compound action … above will be atomic with respect to any other action that also uses synchronized(object)
- synchronized foo (…) {…} is equivalent to foo (…) synchronized(this) {…}

Correct use of intrinsic locks

- Every access to a mutable state variable accessed by more than one thread must be guarded by the same lock (document which one) (Principle 1)
- All variables that are related by an invariant must be guarded by the same lock (Principle 2)

Intrinsic lock re-entrancy

- If a thread already holds an intrinsic lock, can it enter another synchronized block protected by that lock?
  - It's a language or library design decision
  - Java – yes; locks are re-entrant
  - Some languages – no
- Implementation: owner (thread) and locked count for each lock

Argument for reentrancy

- Allows overriding synchronized method to call overridden synchronized method:
- synchronized foo(…) { … super.foo(…); … }
- Convenient if part of the work of a synchronized method is interesting in its own right as a synchronized method
- synchronized foo(…) {… bar(…); …}
- synchronized bar(…) {…}
- Argue that it is better to use bar() in foo() than to repeat the code

Argument against re-entrancy

- If the invariant holds whenever the lock is released (which is a basic correctness criterion) and
- Locks are not re-entrant, then
- at the beginning of any synchronized block you can trust that the invariant guarded by that lock holds
- Re-entrancy violates encapsulation

For next time

- Read Chapter 3