Composing Objects

Designing a Thread-Safe Class

- To design a thread-safe class:
  1. Identify **state variables**
  2. Identify **invariants**
  3. Establish a **synchronization policy**
- Example: Listing 4-1 link

Synchronization Requirements

- Smaller state space makes reasoning easier
- Invariants introduce **atomicity requirements**
- Related variables must be updated atomically
- Violating invariants breaks correctness

State-dependent Operations

- Examples of operations with preconditions:
  - Removing from an empty queue
  - Acquiring a permit when none are available
- Concurrency enables **waiting** until the precondition becomes true.
- Use: Blocking queues, semaphores, locks.
- Encapsulation prevents unwanted sharing.
  - Listing 4-2 link

Java Monitor Pattern

- Wrap all mutable state in an object and guard access with its intrinsic lock.
- Listing 4-3 link
- Listing 4-4 link

Delegation (Thread-safe Components)

- Immutable types remove sharing concerns.
  - Listing 4-6 link
  - Listing 4-7 link
- Delegation fails when invariants span multiple variables.
  - Listing 4-10 link

Publishing Mutable State Safely

- Using thread-safe mutable objects.
  - Listing 4-11 link
  - Listing 4-12 link

Extending Thread-safe Classes

- Extending vector: Listing 4-13
- Client side locking: Listing 4-15
- Composition: Listing 4-16

Documenting Synchronization Policies

- Document:
  - Which variables are guarded
  - Which locks guard them
  - Atomicity guarantees
  - Whether client-side locking is allowed
- Annotations: `@ThreadSafe`, `@NotThreadSafe`, `@GuardedBy("lock")`

Key Lessons

- Encapsulation simplifies concurrency reasoning
- Invariants dictate atomicity
- Confinement prevents unsafe sharing
- Monitor pattern is simple and safe
- Delegation works only when components are independent
- Publishing mutable state requires caution
- Composition is safer than inheritance
- Documentation is essential