Sharing Objects

Why Sharing Objects Is Hard

- Concurrent programs center on shared, mutable state.
- Correctness requires managing both atomicity and visibility.
- Synchronization ensures memory visibility.
- Without it, threads may see stale or inconsistent values.

Visibility Problems

- Writes by one thread may never be seen by another.
- Reads may observe stale values.
- Reordering may cause instructions to appear out of order.

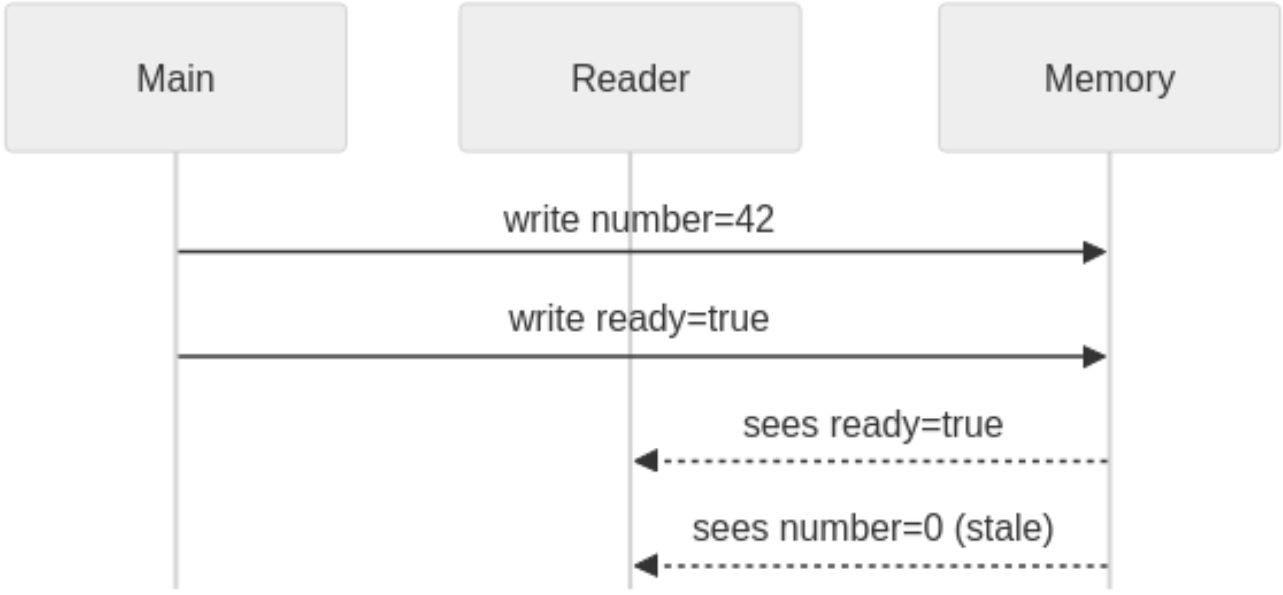
Example: No Visibility

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready) Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Example: No Visibility



Stale Data

- May cause incorrect output.
- May create infinite loops.
- May corrupt data structures.
- Difficult to debug.

Non-Atomic 64-bit Operations

- `long` and `double` may be written as two halves.
- Torn reads possible without `volatile`.

Locking & Visibility

- Acquiring a lock: loads latest values.
- Releasing a lock: flushes writes.

Volatile Variables

- Guarantees visibility and ordering.
- No atomicity for compound actions.
- Good for flags, lifecycle events.

Volatile Example

```
volatile boolean asleep;
while (!asleep) countSomeSheep();
```

Publication & Escape

- Occurs when an object becomes accessible outside intended scope.
- Can lead to observation of partially constructed objects.

Unsafe Publication Example

```
public static Set<Secret> knownSecrets;
public void initialize() {
    knownSecrets = new HashSet<Secret>();
}
```

Internal State Escape

```
class UnsafeStates {
    private String[] states = { "AK", "AL" };
    public String[] getStates() { return states; }
}
```

Escaping `this` in Constructors

```
public class ThisEscape {
    public ThisEscape(EventSource source) {
        source.registerListener(new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        });
    }
}
```

Safe Construction Pattern

```
public class SafeListener {
    private final EventListener listener;
    private SafeListener() {
        listener = e -> doSomething(e);
    }
    public static SafeListener
    newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }
}
```

Thread Confinement

- Avoids sharing entirely.
- Types: ad-hoc, stack confinement, ThreadLocal.

ThreadLocal Example

```
private static ThreadLocal<Connection> connectionHolder =
    new ThreadLocal<Connection>() {
        public Connection initialValue() {
            return DriverManager.getConnection(DB_URL);
        }
    };
```

Immutability

- Immutable objects are always thread-safe.
- Rules: no state changes, all fields final, proper construction.

Immutable Example

```
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges =
        new HashSet<String>();
    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }
    public boolean isStooge(String name) {
        return stooges.contains(name);
    }
}
```

Publishing Immutable Objects

```
@Immutable
class OneValueCache {
    private final BigInteger lastNumber;
    private final BigInteger[] lastFactors;

    public OneValueCache(BigInteger i,
                         BigInteger[] factors) {
        lastNumber = i;
        lastFactors =
            Arrays.copyOf(factors, factors.length);
    }
}
```

Safe Publication Mechanisms

- Static initializers
- Volatile fields
- Final fields
- Locks
- Thread-safe collections

Effectively Immutable Objects

- Mutable but not modified after publication.
- Can be used like immutable objects without locks.

Mutable Objects

- Must be safely published.
- Must be synchronized for all accesses.

Sharing Policies

- Thread-confined
- Shared read-only
- Shared thread-safe
- Guarded

Summary

- Visibility is essential.
- Synchronization ensures visibility + atomicity.
- Avoid escaping `this` during construction.
- Prefer immutability.
- Publish objects safely.