Outline

1. Java Memory Model
2. Example Programs
Java does **not** guarantee linearizability or sequential consistency
Java does **not** guarantee linearizability or sequential consistency.

Sequential consistency inhibits widely used compiler optimizations that reorder memory reads and writes:
- register allocation
- common subexpression elimination
- redundant read elimination
Relaxed Memory Model

**Fundamental Property**  If a program’s sequentially consistent executions follow certain rules, then every execution of that program in the relaxed model will still be sequentially consistent.
Objects reside in shared memory
Each thread has a local cached copy of fields it has read or written
A write to a field may not propagate to shared memory
A read of a field may see a cached value instead of the one in shared memory
Own, local reads and writes happen in order
private Singleton instance = null;

public static Singleton getInstance() {
    if (instance == null) {
        synchronized (Singleton.class) {
            if (instance == null)
                instance = new Singleton();
        }
    }
    return instance;
}
Double-Checked Locking is Incorrect!

- The assignment to `instance` in line 6 may be written to memory before the constructor is finished initializing the object.
- Another thread’s invocation of `getInstance` would find that `instance != null` in line 3 and return the reference to the unfinished object.
Synchronization Events

- Certain statements are synchronization events
- Not necessarily atomicity or mutual exclusion
- Reconciliation of local cache with shared memory
  - flushing local writes
  - invalidating cached reads
- Synchronization events are **linearizable**: totally ordered and all threads agree
Mutual exclusion by
- entering a synchronized block or method
- acquiring an explicit lock

If all accesses to a field protected by the same lock, then reads-writes to that field are linearizable:
- unlock() writes back changed fields to shared memory
- lock() invalidates the cache, forcing a reread from shared memory
Fields declared `volatile` are linearizable

Reading from volatile is like acquiring a lock
invalidates the cache, read from shared memory

Writing to volatile is like releasing a lock
writes through to shared memory
Volatile Fields

- Fields declared `volatile` are linearizable
- Reading from volatile is like acquiring a lock, invalidates the cache, read from shared memory
- Writing to volatile is like releasing a lock, writes through to shared memory

**Attention**

- Multiple read-writes are not atomic
- Typical usage pattern: single writer / multiple readers
Volatile Fields

- Fields declared `volatile` are linearizable
- Reading from volatile is like acquiring a lock invalidates the cache, read from shared memory
- Writing to volatile is like releasing a lock writes through to shared memory

Attention

- Multiple read-writes are not atomic
- Typical usage pattern: single writer / multiple readers

Linearizable Memory

- `AtomicReference<T>, AtomicInteger`
- Methods `compareAndSet(), set()` like writes
- Method `get()` like read
Final Fields

- A final field cannot be modified once it has been initialized.
- Initialized in the constructor
- Under simple rules, the correct value of a final field will be visible to other threads without synchronization.
Example: Constructor with Final Field

class FinalFieldExample {
    final int x; int y;
    static FinalFieldExample f;
    public FinalFieldExample() {
        x = 3;
        y = 4;
    }
    static void writer() {
        f = new FinalFieldExample();
    }
    static void readers() {
        if (f != null) {
            int i = f.x; int j = f.y;
            // i == 3 is guaranteed
            // no guarantee about y’s value
        }
    }
}
Incorrect EventListener Class

```java
public class EventListener {
    final int x;
    public EventListener (EventSource eventSource) {
        eventSource.registerListener(this);
    }
    public onEvent(Event e) {
        // handle the event
    }
}
```

- **onEvent** may be invoked
  - after the listener is registered
  - but before the constructor is completed, i.e., before the value of `x` is flushed to shared memory
Outline

1. Java Memory Model
2. Example Programs
Example Programs

- Example programs taken from “Java Concurrency in Practice” by Brian Goetz and others
Thread Safety

Stateless objects are always thread-safe

```java
@ThreadSafe
public class StatelessFactorizer extends GenericServlet implements Servlet {
    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }
}
public class UnsafeCountingFactorizer extends GenericServlet implements Servlet {

    private long count = 0;

    public long getCount() {
        return count;
    }

    public void service(ServletRequest req, ServletResponse resp)
    {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        ++count;
        encodeIntoResponse(resp, factors);
    }
}
Lazy Initialization

Don’t do this

```java
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }
```
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }

    static class Resource {
    }
}
```java
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() {
        return resource;
    }

    static class Resource {
    }
}
```
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceFactory.ResourceHolder.resource;
    }

    static class Resource {
    }
}
```java
public class CountingFactorizer extends GenericServlet implements Servlet {

    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet();
        encodeIntoResponse(resp, factors);
    }
}
```
public class UnsafeCachingFactorizer extends GenericServlet implements Servlet {

    private final AtomicReference<BigInteger> lastNumber = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }
}
public class SynchronizedFactorizer extends GenericServlet implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }
}
public class Widget {
    public synchronized void doSomething() {
        ...
    }
}

public class LoggingWidget extends Widget {
    public synchronized void doSomething() {
        System.out.println(toString() + ": calling doSomething");
        super.doSomething();
    }
}
Locking: More than one state variable

Working example

```java
public class CachedFactorizer extends GenericServlet implements Servlet {

    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() {
        return hits;
    }

    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }

    void encodeIntoResponse(ServletResponse resp, BigInteger[] factors) {
    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        // Doesn't really factor
        return new BigInteger[]{i};
    }
}
```
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;
    }
}
Unsafe publication
Don’t do this

class UnsafeStates {
    private String[] states = new String[]{
        "AK", "AL" /*...*/
    };

    public String[] getStates() {
        return states;
    }
}
Unsafe publication II
Don’t do this

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

    void doSomething(Event e) {
    }

    interface EventSource {
        void registerListener(EventListener e);
    }

    interface EventListener {
        void onEvent(Event e);
    }

    interface Event {
    }
}
```
public class SafeListener {
    private final EventListener listener;

    private SafeListener() {
        listener = new EventListener() {
            public void onEvent(Event e) {
                doSomething(e);
            }
        };
    }

    public static SafeListener newInstance(EventSource source) {
        SafeListener safe = new SafeListener();
        source.registerListener(safe.listener);
        return safe;
    }

    void doSomething(Event e) {
    }

    interface EventSource {
        void registerListener(EventListener e);
    }

    interface EventListener {
        void onEvent(Event e);
    }

    interface Event {
    }
}