PRINCIPLES OF *JAVA* MULTICORE PROGRAMMING

TECHNIQUES AND CAVEATS

Created by Ray Imber

Use the arrow keys to navigate

HIGH LEVEL API'S

- Thread Pools
- Concurrent Collections

THREAD POOLS

LET'S GO SWIMMING

- In large-scale applications, it makes sense to separate thread management and creation from the rest of the application
- Known as *executors* in Java
- Thread Pools are the most common kind of executor.

WHAT DO THREAD POOLS DO?

- Creating new Threads is expensive in terms of memory and time for creation.
- Instead of creating a new Thread for each new job, ad-hoc.
- Create a pre-defined set of worker Threads at the begining of the program.
- These worker threads never terminate.
  - They wait for a job.
  - They Execute the job.
  - They are then recycled.

See Java Docs for more info

THE OPTIMAL THREAD POOL SIZE

LITTLE'S LAW

$$L = \lambda W$$

Where L is the average number of customers in the system, λ is the arrival rate, and W is the average time it takes to service a customer.

- Start with an arbitrary Pool Size.
- Measure rate at which work unit requests arrive and the average amount of time to service them.
- Calculate the average number of requests in the system.
- If L is less than the pool size, reduce the pool size.
- If L is greater than the pool size, determine if system can support a larger thread pool.

CONCURRENT COLLECTIONS

- Pre-built data-structures and algorithms provided by Oracle that are Thread Safe and Thread Optimized.
- These are *NOT* built into the language. They are libraries, provided for your convenience.
- Many of these libraries are implemenations based on the theories from this course.

See Java Docs for more info

COMMON CONCURRENT COLLECTIONS

- BlockingQueue:
  - A FIFO data structure that blocks or times out on add to a full queue, or retrieve from an empty queue.
- ConcurrentHashMap:
  - A concurrent analog of HashMap.
- ConcurrentSkipListMap:
  - A concurent analog of a Red-Black tree.
- ThreadLocalRandom:
  - A thread optimized random number generator.
  - `Math.random()` is thread safe, but performs badly.

JAVA MEMORY MODELS

IT'S ALL ABOUT SHARING

THREAD STATE

- A Thread State has three parts
  - Program Counter
  - (Thread) Local Variables
  - Shared Variables

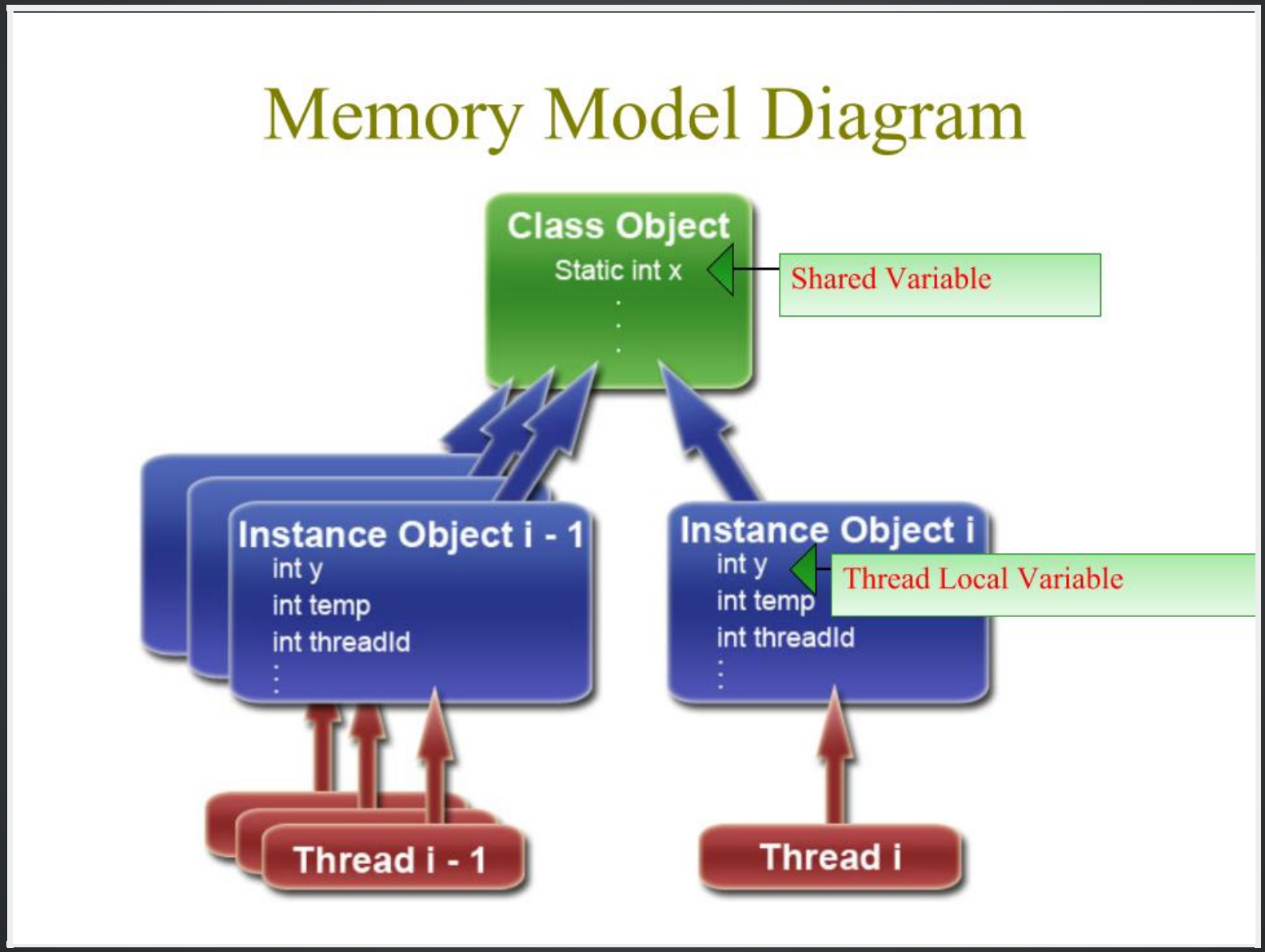
SHARED VS LOCAL VARIABLES

- Thread communication is achieved through Shared Variables.
- All memory in Java must be contained in Objects.
  - Must have a way to seperate Shared memory from Local memory.
  - We must use a model to conceptualize this separation.

JAVA OBJECT MODELS

- Two Object Models for Thread Variables
  - One Thread / One Object
  - Multiple Threads / One Object

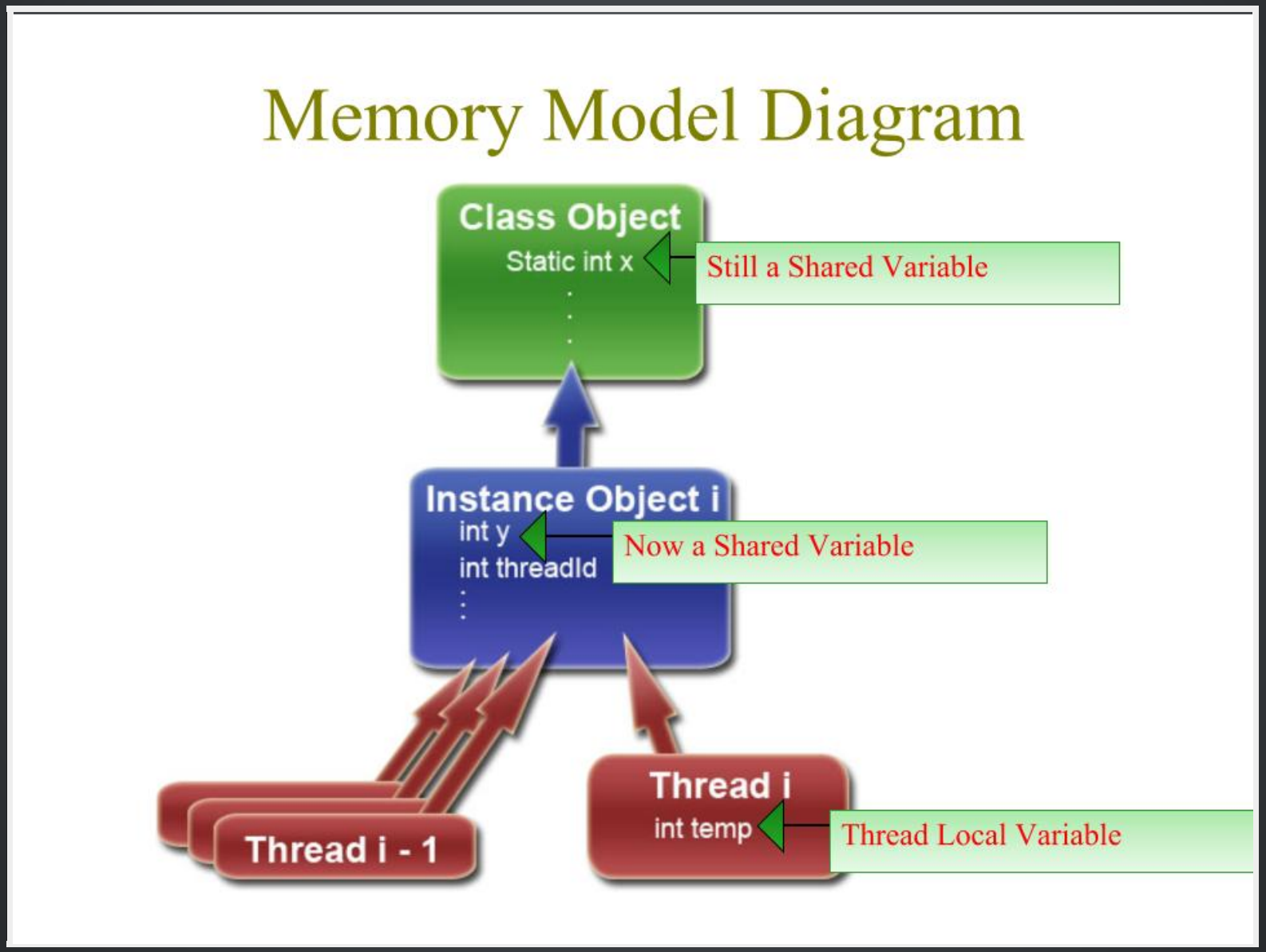
ONE THREAD / ONE OBJECT



EXAMPLE

```
Thread thread[] = new Thread[maxThreads];
for( int i = 0 ; i < maxThreads ; i++ )
{
    thread[i] = new Thread(new Worker());
}
```

MULTIPLE THREADS / ONE OBJECT



EXAMPLE

```
Thread thread[] = new Thread[maxThreads];
Worker singleRunnable = new Worker();
for( int i = 0 ; i < maxThreads ; i++ )
{
    thread[i] = new Thread(singleRunnable);
}
```

LOCKS AND ATOMIC VARIABLES IN JAVA

- Synchronized
- Volatile

SYNCHRONIZED

LOCKING IN JAVA

- Synchronized is implemented with a *Monitor Lock*
  - See Chapter 8 of the textbook (pg.177).
- Two Ways to use
  - Synchronized Methods
  - Synchronized Statements

SYNCHRONIZED METHODS

- Simplest way to use locks in Java.
- The entire method becomes a critical section.
- The lock is attached to `this`.
- `Static Synchronized` lock is attached to the *class* object.

EXAMPLE

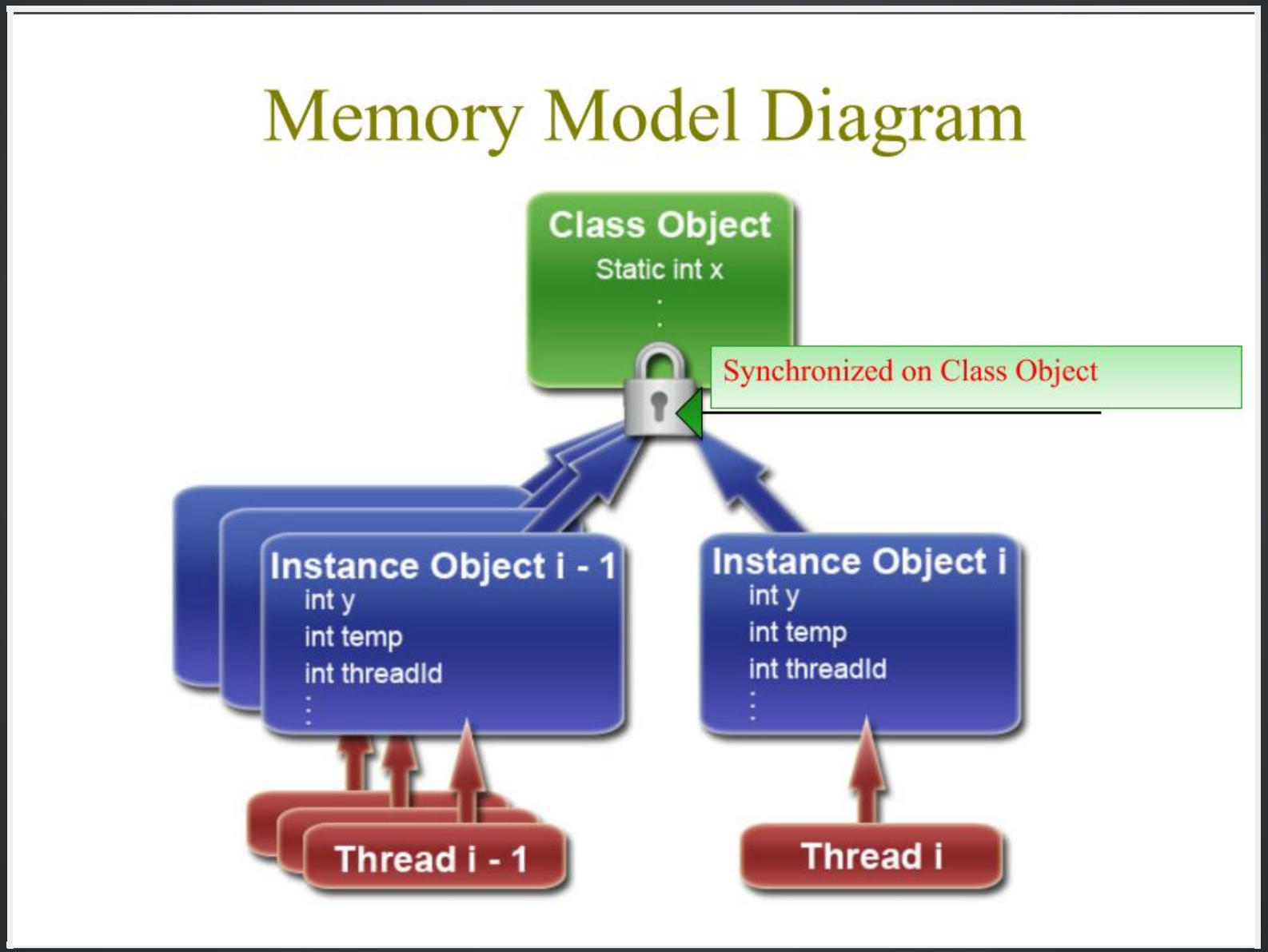
```
public class SynchronizedCounter
{
    private int c = 0;
    private static int s = 1;

    public synchronized void increment() {
        c++;
    }

    public synchronized int value() {
        return c;
    }

    public static synchronized int staticValue() {
        return s;
    }
}
```

MEMORY MODEL



SYNCHRONIZED STATEMENTS

- More control than synchronized methods.
- Define a specific block of statements to be the critical section.
- More efficient than locking the whole method.
- Can control which lock you use.

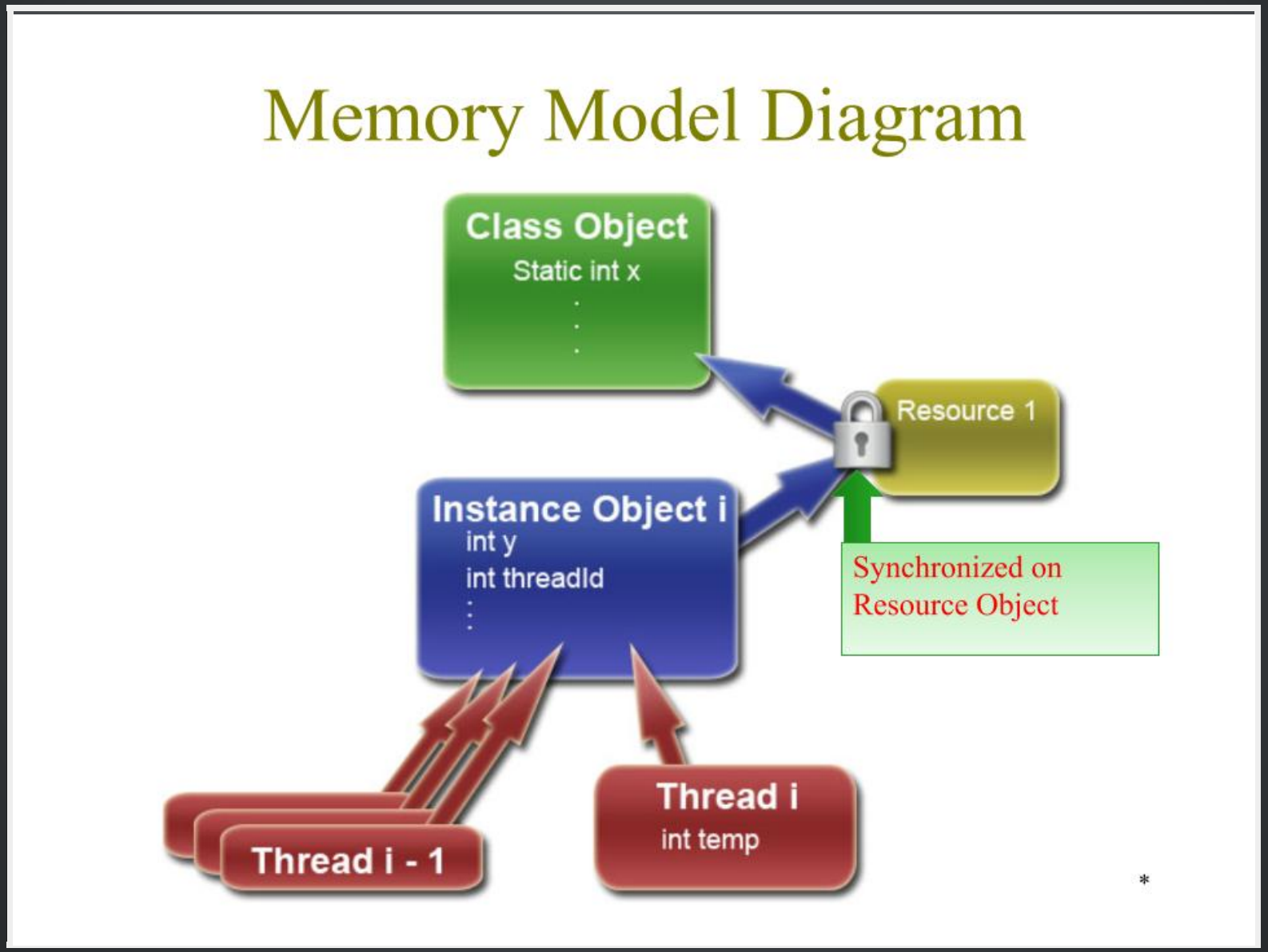
EXAMPLE

```
public class MsLunch
{
    private long c1 = 0;
    private long c2 = 0;
    private Object extLock = new Object();

    public void inc1() {
        synchronized(this) {
            c1++;
        }
    }

    public void inc2() {
        synchronized(extLock) {
            c2++;
        }
    }
}
```

MEMORY MODEL

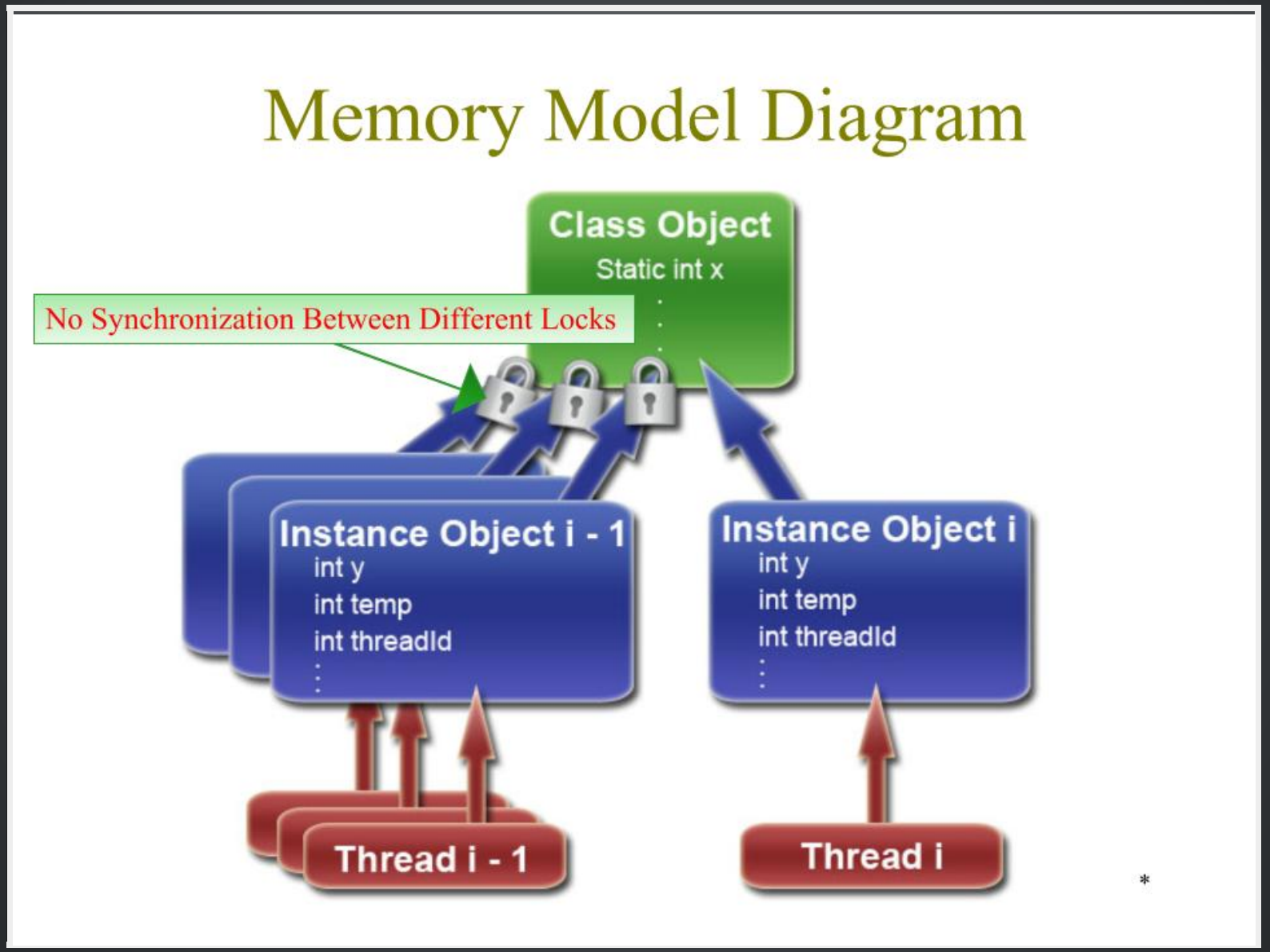


CODE EXAMPLE

```
private static int x;

private void readAndWrite() {
    int temp;
    synchronized(this){
        // Update Shared Class Variable
        temp = x;// Read x
        temp=temp+1;// increment x
        x=temp;// write x
    }
}
...
public static void main(String args[]) {
    Thread thread[] = new Thread[maxThreads];// Create a thread array
    for( int i = 0 ;i < maxThreads ; i++ ) {
        thread[i] = new Thread(new BadSync());
    }
}
```

DANGERS OF IMPROPER LOCKING



FINAL NOTE ABOUT SYNCHRONIZED

- Recall that All Objects in Java implement the Monitor interface
- Synchronized uses this implicit monitor to work
- You can use the monitor directly in your code!
- All of the Monitor Methods are available to you
  - wait()
  - notify()
  - notifyAll()

ATOMIC OPERATIONS

- What operations are Atomic in Java?
- Reads and writes are atomic for reference variables and for *most* primitive variables
  - Except long and double.

- Reads and writes are atomic for all variables declared volatile
  - Including long and double.

- Memory consistency errors are still possible!

VOLATILE

- Makes *any* variable atomic. This includes objects (but not necessarily properties of the object)!
  - a.k.a pointers
  - 64 bit variables : doubles and longs

- A write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable
- Side effects that lead up to the change of the variable are also completely visible.

VOLATILE CONS

- Volatile is awesome, why should we ever use Synchronized?
- Volatile only works for simple reads and writes.
  - No complex logic.
- Only applys to An Object's Reference not it's properties.
  - This applies to arrays and array elements as well.
- Volatile only guarantees order inside a single thread is preserved.
  - This is a special guarantee that can be difficult to reason about.
  - Happens Before does not guarantee total ordering!

EXAMPLE

```
class VolatileTest {
    static volatile int i = 0, j = 0;

    static void threadOne() { i++; j++; }

    static void threadTwo() { System.out.println("i=" + i + " j=" + j); }

    }
```

VOLATILE PERFOMANCE VS. SYNCHRONIZED PERFORMANCE

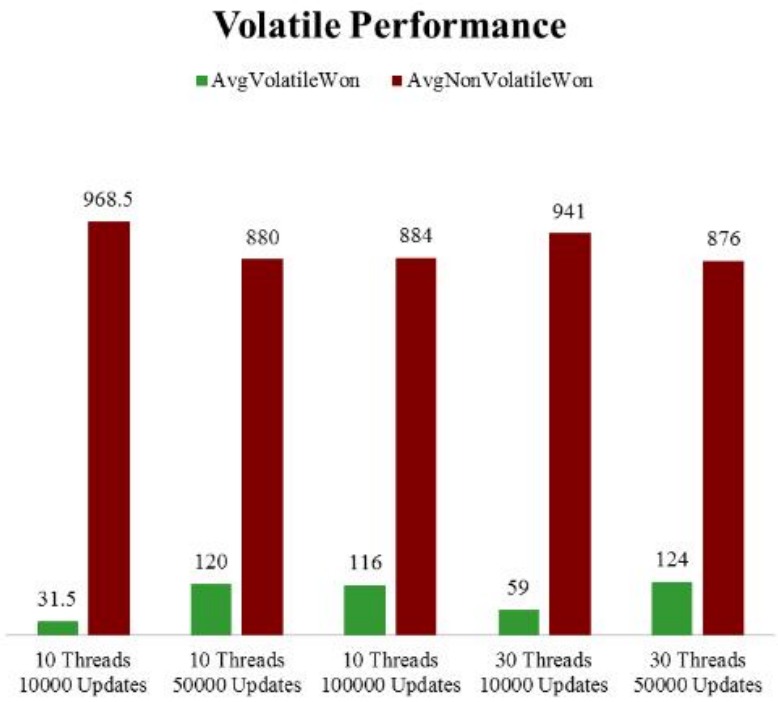
- Volatile and Synchronized force cpu caches to be flushed on a write.
  - Synchronized flushes all caches.
  - Volatile flushes only the cache block containing the written variable.

- Volatile can never have a higher overhead than synchronized.
  - Synchronized has to do locking and unlocking overhead.
  - Volatile does not have any lock overhead.

- Synchronized can have the same overhead if the compiler (JIT) is able to optimize it.

THE COST OF A VOLATILE CACHE FLUSH

Volatile Performance Example



On average, volatile lost **13.74** times more than non-volatile!

*Note: Results may vary based on Hardware, JVM, etc…

IMPLEMENTATION DETAILS

THE DEVIL IS IN THE DETAILS

- CPU Scheduler
- Memory Consistency
- Compiler Transformations
- Semantic Rules

ORDERING OF EVENTS

- Everything before an unlock (release) is Visible to everything after a later lock (acquire) on the same Object
- The release pushes the updates...

HAPPENS BEFORE

- Happens-before is transitive
- Data races are avoided if you use "happens-before"
- Action in VM. Not lines of code (Assembly Instructions)
- There are tools that can check for data races
  - JProbe - tool for java

EXPECTATIONS

- You would expect that the following over simplification would be true:
- Get lock
- Make updates
- Push cache to main memory
- Release lock
- You are NOT guaranteed to get this

COMPILER TRANSFORMATIONS

- One threads reads a variable and continually checks just that variable.
- Continues to read until a different thread changes the variable.
- compiler "may" look at the code and see that all the loop does is read the same variable, which never changes
- It "optimizes" it just by making it into an infinite loop and avoid the reads all together.

EXAMPLE

YOU WRITE THIS:

```
class OptimizeMe {

    static boolean checkMe = true;

    public threadOne() {
        while(checkMe == true) {
            //Do something that does not change the value of checkMe.
        }
    };

    public threadTwo() {
        Thread.sleep(4000);
        checkMe = false;
    }
}
```

EXAMPLE (TRANSFORMED)

A NAIVE COMPILER MAY TRY TO TRANSFORM THE CODE INTO THIS:

```
class OptimizeMe {

    static boolean checkMe = true;

    public threadOne() {
        while(true) {
            //Do something that does not change the value of checkMe.
        }
    };

    public threadTwo() {
        Thread.sleep(4000);
        checkMe = false;
    }
}
```

HOW VOLATILE CAN HELP

- *Volatile* will stop the compiler from being able to optimize this.
- If the JVM analyses the runtime code, and determines that a synchronized() call point is completely unnecessary, it can eliminate the overhead completely.
- Volatile, on the other hand, cannot be elminiated