PRINCIPLES OF *JAVA* MULTICORE PROGRAMMING

TECHNIQUES AND CAVEATS

Created by Ray Imber

A QUICK OVERVIEW

- Why Java?
- Basic Syntax and Semantics
- Implementation details and Caveats

WHY JAVA?

- Java runs on many devices.
- The VM abstracts many details.
- Java has specific language concepts and structures for dealing with parallelism.
  - These structures have their own *peculiariarities* in behavior
  - Parallelism in an Object Oriented Language

JAVA THREADS

- Java has a *thread Object*.
  - Represents the thread itself, not the executed logic.

- Thread Object executes Objects that implement the *Runnable Interface*.
  - The Runnable object can subclass a class other than Thread.
  - This approach is more flexible
  - Applicable to the high-level thread management APIs covered later

THE RUNNABLE INTERFACE

- The Runnable Interface has one method: *public void run()*.
- run takes no arguments and returns nothing.
  - How is Thread communication achieved?
  - (We will discuss this in much greater detail soon)

A SIMPLE EXAMPLE

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

SLEEP, INTERRUPT, AND JOIN

JOIN ME WHILE I SLEEP AND DON’T INTERRUPT :-P

SLEEP

- `Thread.sleep` causes the current thread to suspend execution for a specified period.
- Two Versions of Sleep:
  - Millisecond Accuracy
  - Nanosecond Accuracy
- Sleep times are not guaranteed to be precise
  - Limited by the OS.

INTERRUPTS

- Indication to a thread that it should stop what it is doing and do something else.
  - That usually means the thread just terminates, but it could do other things.

- Two ways of dealing with Interrupts
  - High Level: catch `InterruptedException`
  - Low Level: check the `Thread.interrupted()` flag

EXAMPLE

USING EXCEPTIONS

```
for (int i = 0; i < importantInfo.length; i++)
{
    // Pause for 4 seconds
    try
    {
        Thread.sleep(4000);
    }
    catch (InterruptedException e)
    {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

EXAMPLE

USING THE INTERRUPT FLAG

```
for (int i = 0; i < inputs.length; i++)
{
    heavyProcessing(inputs[i]);
    if (Thread.interrupted())
    {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

OR

THROW OUR OWN INTERRUPTEDEXCEPTION

```
for (int i = 0; i < inputs.length; i++)
{
    heavyProcessing(inputs[i]);
    if (Thread.interrupted())
    {
        throw new InterruptedException();
    }
}
```

CAUSING INTERRUPTS YOURSELF

- You can interrupt a thread yourself using `Thread.interrupt()`
  - This sets the internal interrupt flag to true for a *particular thread*.
  - No guarantee that the thread will respect your interrupt request!

JOIN

- `Thread.join()` causes one thread to wait for another.
- `Join` throws interruptedExcpetions, just like `sleep`.
- You can pass an optional timeout to limit the amount of time the thread will wait.

EXAMPLE

```
public static void main(String args[])
    throws InterruptedException {
        Thread t = new Thread(new WorkerObject());
        t.start();
        // Wait maximum of 1 second
        // for Thread
        // to finish.
        t.join(1000);

        if ( t.isAlive() ) {
            // Tired of waiting!
            t.interrupt();
            // Shouldn't be long now
            // -- wait indefinitely
            t.join();
        }
    }
```

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

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

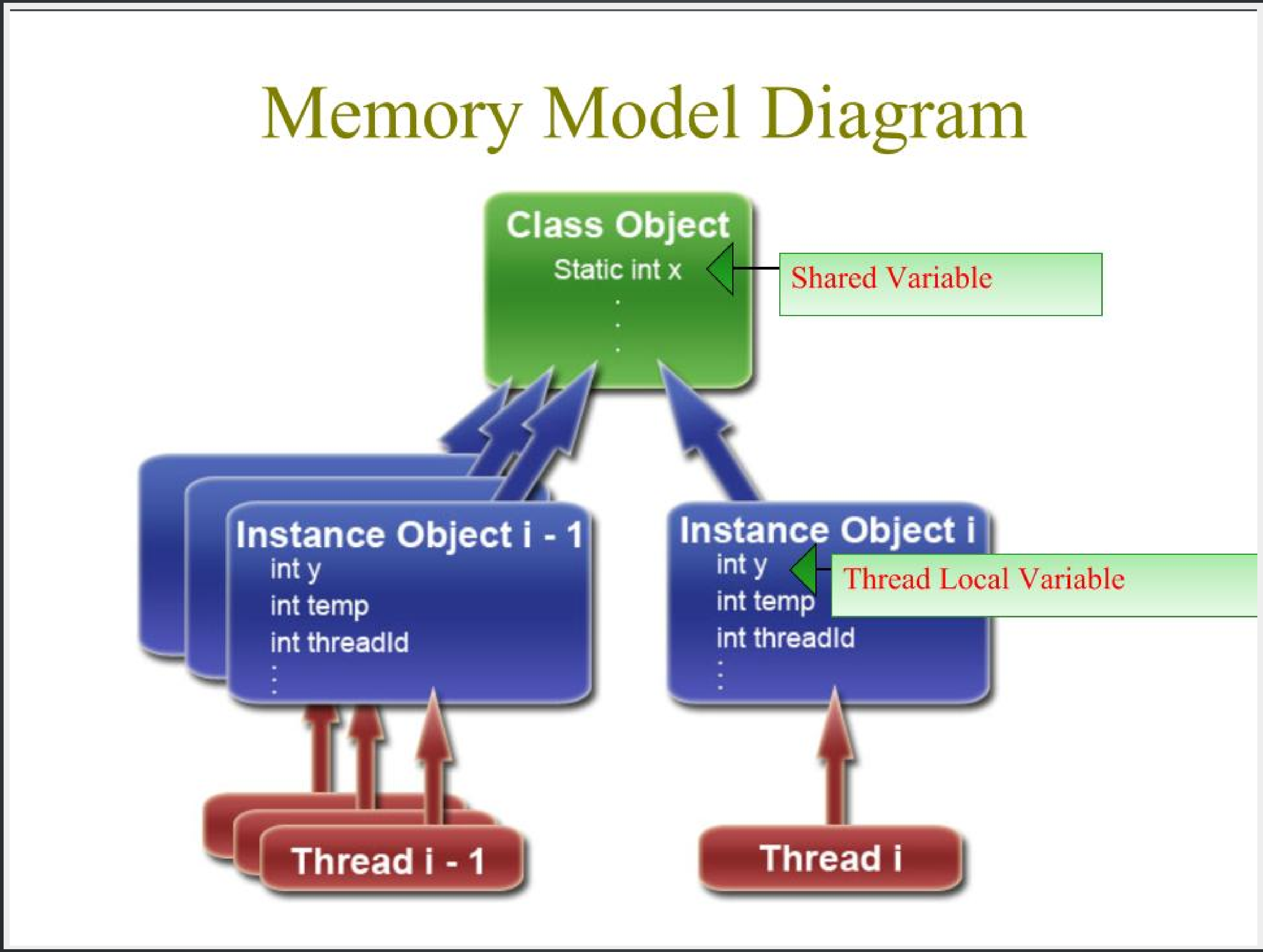
SHARED VS LOCAL VARIABLES

- 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

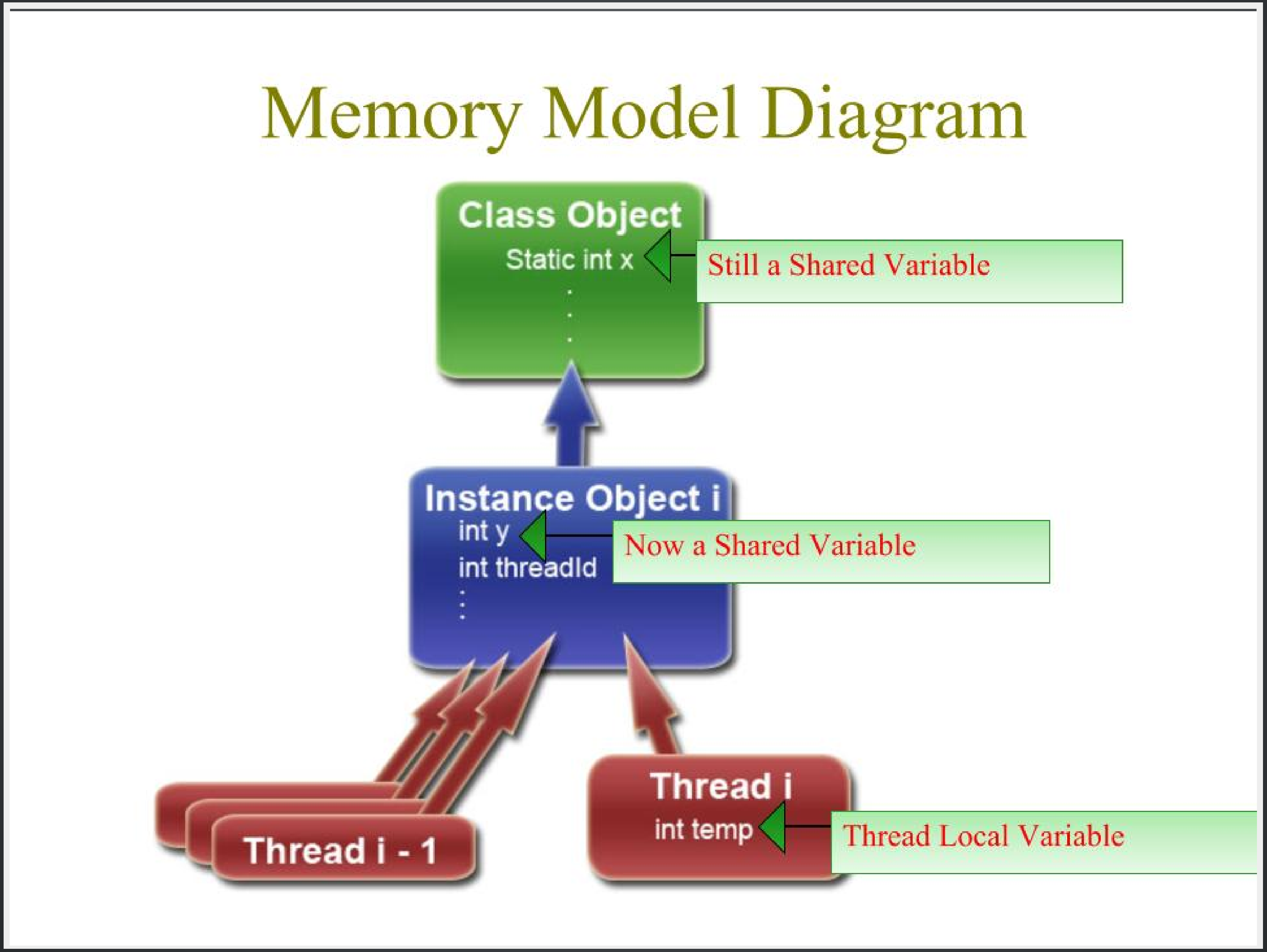
ONE THREAD / ONE OBJECT



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

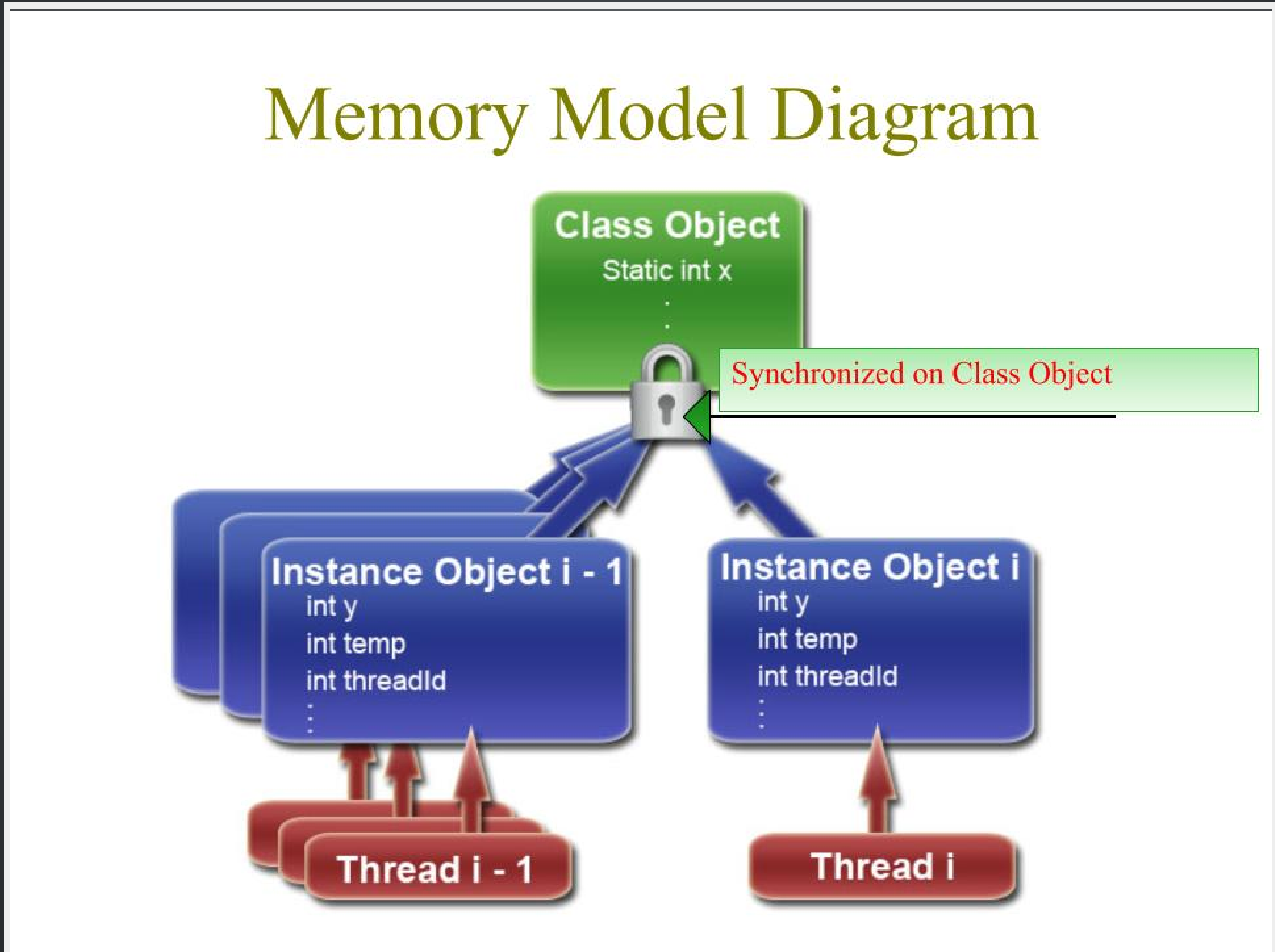
SYNCHRONIZED METHODS

- 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

```java
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



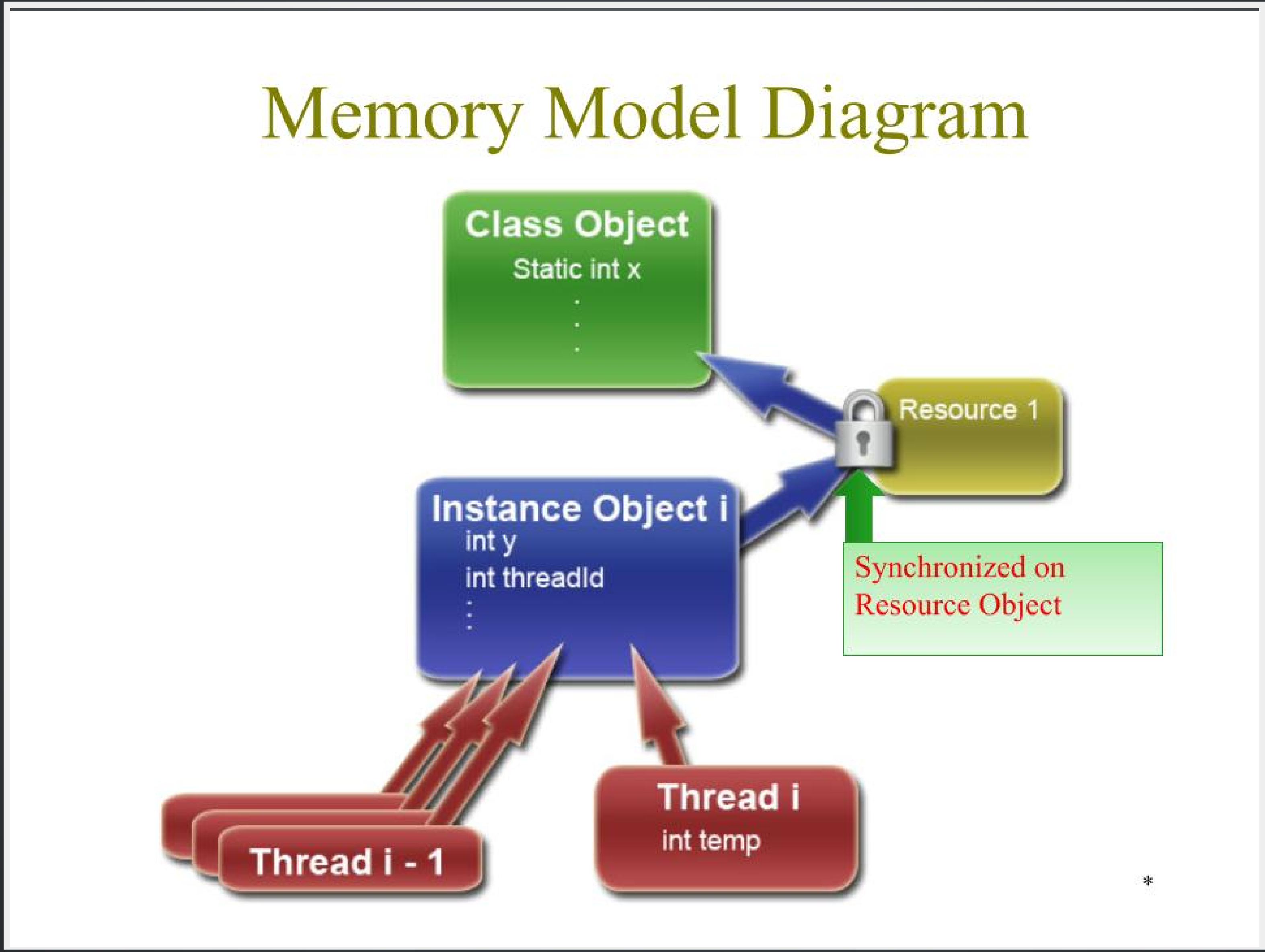
SYNCHRONIZED STATEMENTS

- 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++;
        }
    }
}
```

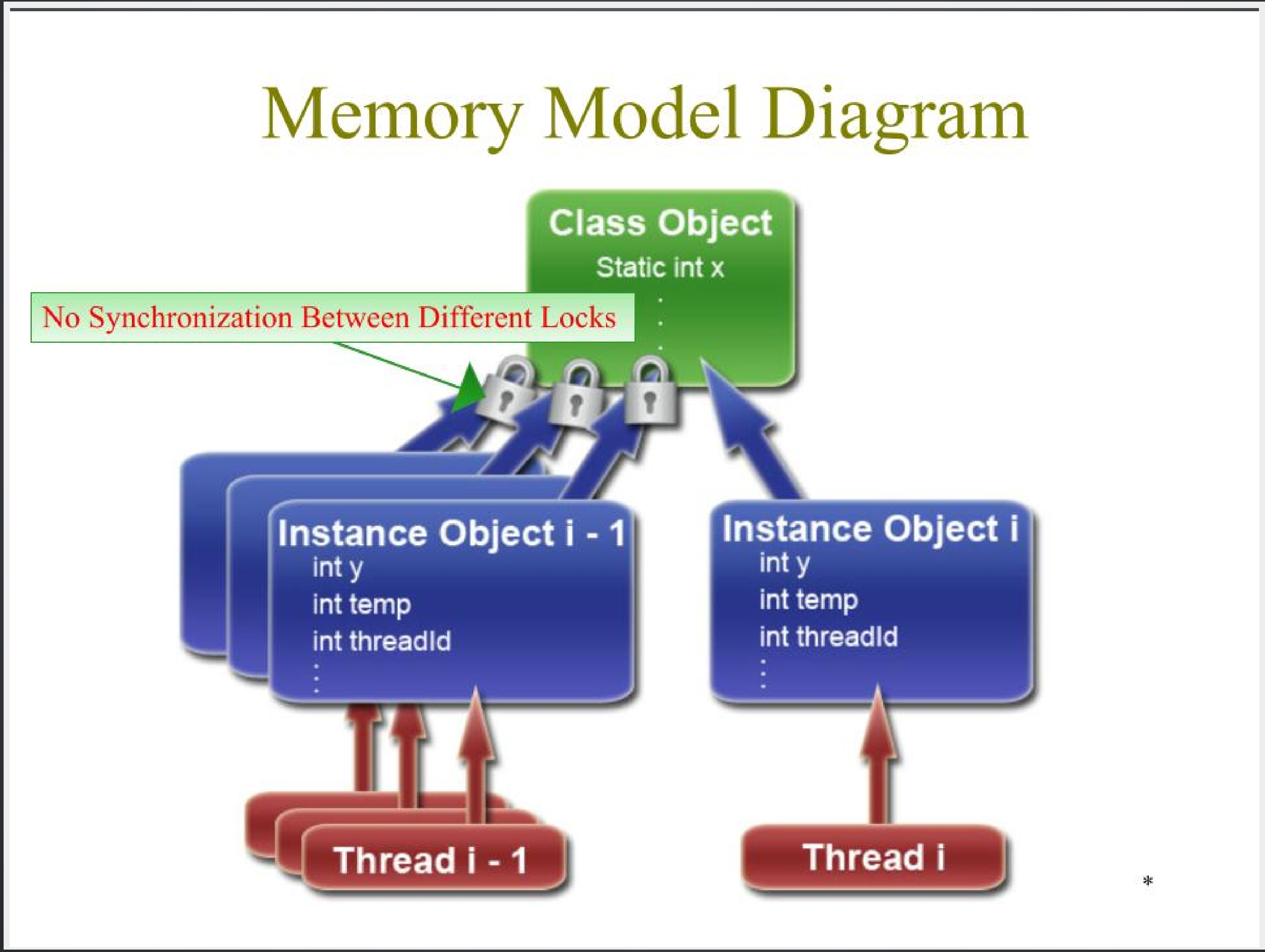
MEMORY MODEL



CODE EXAMPLE

```java
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



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!

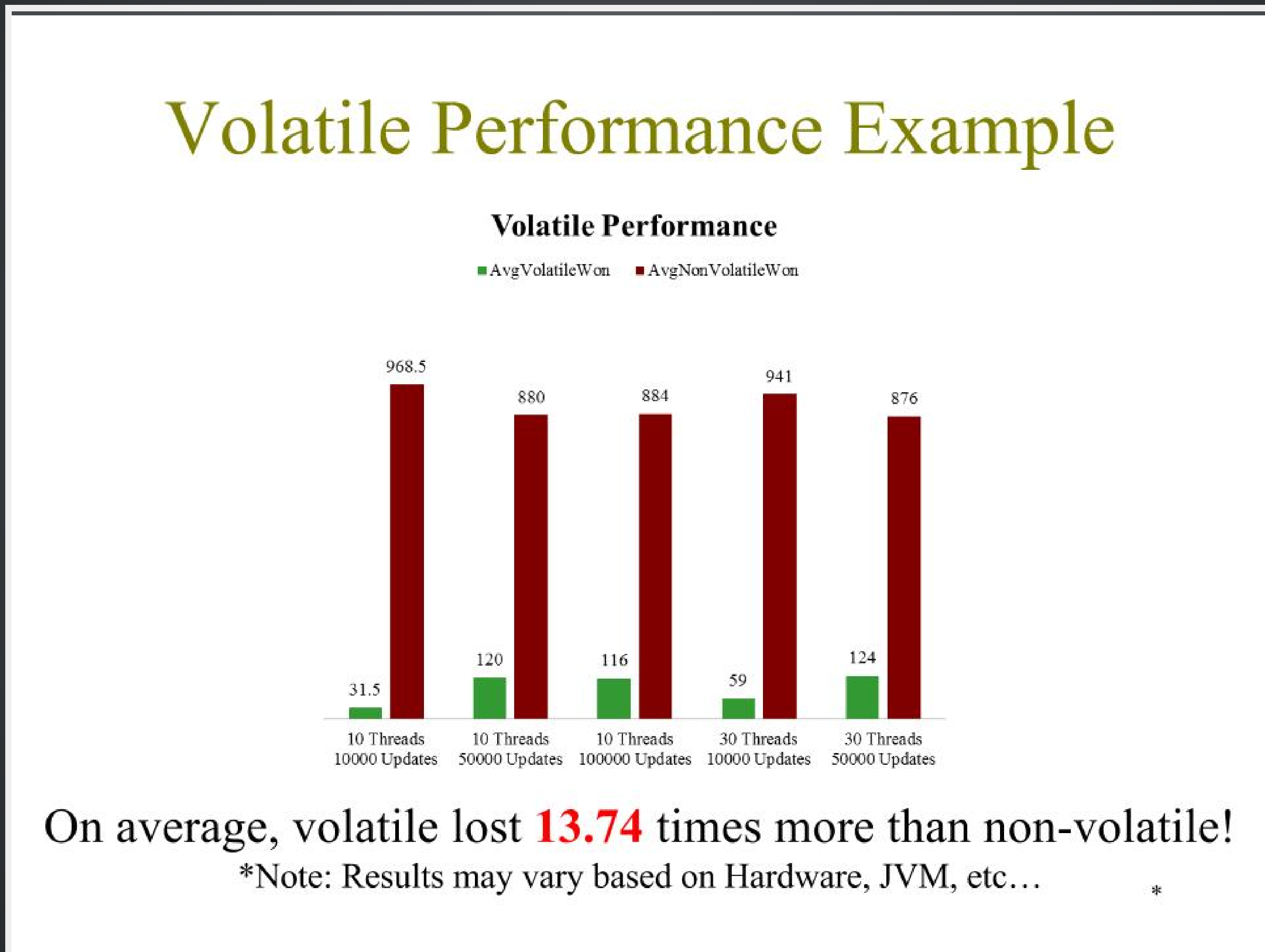
EXAMPLE

```
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



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

- Memory Accesses can be moved into a lock!
- 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

SOFTWARE DESIGN PATTERNS, TIPS, AND TRICKS

THREAD SAFE LAZY LOADING

- Multi-threaded variant of the Singleton Pattern.
- One object shared among all threads.
- Object is not initialized until it is needed.

PROPERTIES THAT MUST HOLD TRUE FOR A MULTI-THREADED SINGLETON

- Must not allow another thread to access the object while the object is still initializing.
  - Initializing Objects takes time!
- Must only initialize the object once.
  - Two threads may start initialing the object at the same time.

DOUBLE CHECK LOCKING

- Check if object has been initialized yet.
- if it has been created, just return a pointer to the already created object.
- if it has not been created: Acquire a lock.
- Check if the object has not been created *again*!
  - Another thread may have already initialized the object before you successfully acquired the lock.
- if it has still not been created, initialize the object, and then return a pointer to the new object.
- Return a pointer to the object.

EXAMPLE

CAN YOU IDENTIFY THE PROBLEM WITH THIS EXAMPLE?

```
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if ( helper == null ) {
            synchronized(this) {
                if (helper == null) { helper = new Helper(); }
            }
        }
        return helper;
    }
 // other functions and members...
}
```

EXAMPLE

DECLARING HELPER VOLATILE GUARANTEES A THREAD WILL NOT READ A PARTIALLY INITIALIZED OBJECT.

```
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if ( helper == null ) {
            synchronized(this) {
                if (helper == null) { helper = new Helper(); }
            }
        }
        return helper;
    }
 // other functions and members...
}
```

MULTI-THREADED LAZY LOADING WITH JAVA LANGUAGE TRICKS

ACHIEVES LAZY LOADING BY USING FEATURES GUARANTEED BY THE JAVA LANGUAGE

```
class Foo {
    private static class HelperHolder {
        public static Helper helper = new Helper();
    }
    public static Helper getHelper() {
         return HelperHolder.helper;
    }
}
```

- Inner classes are not loaded until they are referenced
- The class initialization phase is guaranteed by Java to be serial
- Java uses an implicit lock on all Object Initializations
  Section12.4.2 of the Java Language Specification

IMMUTABILITY

"Choose immutability and see where it takes you" - Rich Hickey (Founder of Clojure)

FINAL FIELDS

- Thread Safe without locking!
- Fields declared final are initialized once, and then never changed under normal circumstances.
- Even if there is a race condition, or a cache inconsistency, a reference to a final field will be valid.
- These guarantees allow the compiler to make many optimizations.

FINAL FIELD COMPILER OPTIMIZATIONS

- Compilers have a great deal of freedom to move reads of final fields across synchronization barriers.
- Compilers can to keep the value of a final field cached in a register and not reload it from memory.

FINAL FIELD INITIALIZATION

- An object is considered to be completely initialized when its constructor finishes.
- Initialize the final fields for an object in the constructor.
- Do not write a reference to the object being constructed in a place where another thread can see it before the object's constructor is finished.
- When the object is seen by a thread, that thread will always see the correct values of the final fields.
- It will also see versions of any object or array referenced by those final fields that are at least as up-to-date as the final fields are.

EXAMPLE

```
class FinalFieldExample {
    final int x;
    int y;
    static FinalFieldExample f;

    public FinalFieldExample() { x = 3; y = 4; } //constructor

    static void writer() { f = new FinalFieldExample(); }
    static void reader() {
        if (f != null) {
            int i = f.x; // guaranteed to see 3
            int j = f.y; // could see 0
        }
    }
}
```

JAVA STRINGS

- Java Strings are implemented with Final.
- Many security features of the Java programming language depend upon String objects being immutable.
- Any time a Java string is modified, a new String object is referenced, the raw data buffer is never changed.
- String Pool saves memory by attempting to re-use old Strings if possible.

DIFFERENT STRING ALTERNATIVES

- String vs. StringBuilder vs. StringBuffer
- String : immutable
  - Memory inefficient but fast and very thread safe.

- String Buffer : mutable, but protected with locks.
  - Memory efficient but slow because of locks.

- String Builder : mutable, no protection.
  - Raw Char buffer like C++ String.
  - Memory efficient and fast, but not thread safe at all.

REFERENCES AND FURTHER READING

- The Official Java Tutorials on Concurrency
- Java Concurrency Package API
- Java Memory Model
- Java Concurrency in Practice (Book)
- Java Perfomance Tuning
- Java Specification on Final Fields
- Comparison of String Class Alternatives

THANK YOU