PRINCIPLES OF *JAVA* MULTICORE PROGRAMMING

TECHNIQUES AND CAVEATS

Created by Ray Imber

Use the arrow keys to navigate

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

PARALLELISM VS. CONCURRENCY

- Concurrency: The composition of independently executing processes
- Concurrency is an algorithmic and software design concern; like memoziation, or recursion
- Parallelism: The simultaneous execution of (possibly related) computations
- Parallelism is an implementation concern; Threads, SIMD, cluster/network computing

Rob Pike on this topic

C.A.R Hoare CSP Paper

A PROCESS VS. A THREAD

- Process: Fork and Join style parallelism
  - A Child program is spawned
  - No Shared memory
  - limited communication options
  - Entire new memory space
  - OS must execute and manage entire new process

- Thread
  - Shared Memory
  - Extensive communication options
  - Share memory space with parent
  - OS has to manage much less (Just thread state)

JAVA 8 LAMBDA EXPRESSIONS

- Functional Interface or Single Abstract Method interfaces
- The Old Syntax:
  - Create an anonymous inner class with just one method
  - Very cumbersome.

- New way:
  - Single expression
  - Converted into a class by the compiler

```
() -> ;
```

Official Java Docs

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()*.
- This can be represented as a "Lambda Expression" in Java 8.
- 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();
    }

}
```

JAVA 8 EQUIVALENT

```
public class HelloLambdaRunnable {

    public static void main(String args[]) {
        (new Thread(() -> System.out.println("Hello from a thread!");)).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();
        }
    }
```

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

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.

EXECUTOR INTERFACE

- Executor: Launches a runnable. Simplest interface.
- ExecutorService: Subinterface of Executor. Adds features for managing the lifecycle of tasks and of the executor.
- ScheduledExecutorService: Subinterface of ExecutorService. Adds support for future and/or periodic execution of tasks.

Java Docs

TYPES OF EXECUTORS

- Standard Thread Pool
- Fork/Join Pool

FIXED THREAD POOL

- Cached Thread Pool
- Single Thread Pool
- ScheduledExecutorService
  - Executes a Runnable or Callable task after a specified delay or at an interval.

FORK/JOIN POOL

- Uses a work-stealing algorithm.
- Worker threads that run out of things to do can steal tasks from other threads that are blocked.
- Must subclass the computation with *ForkJoinTask*
- Typically Recursive

```
if (my portion of the work is small enough)
do the work directly
else
split my work into two pieces
invoke the two pieces and wait for the results
```

Java Docs

JAVA MEMORY MODELS

IT'S ALL ABOUT SHARING

THREAD STATE

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

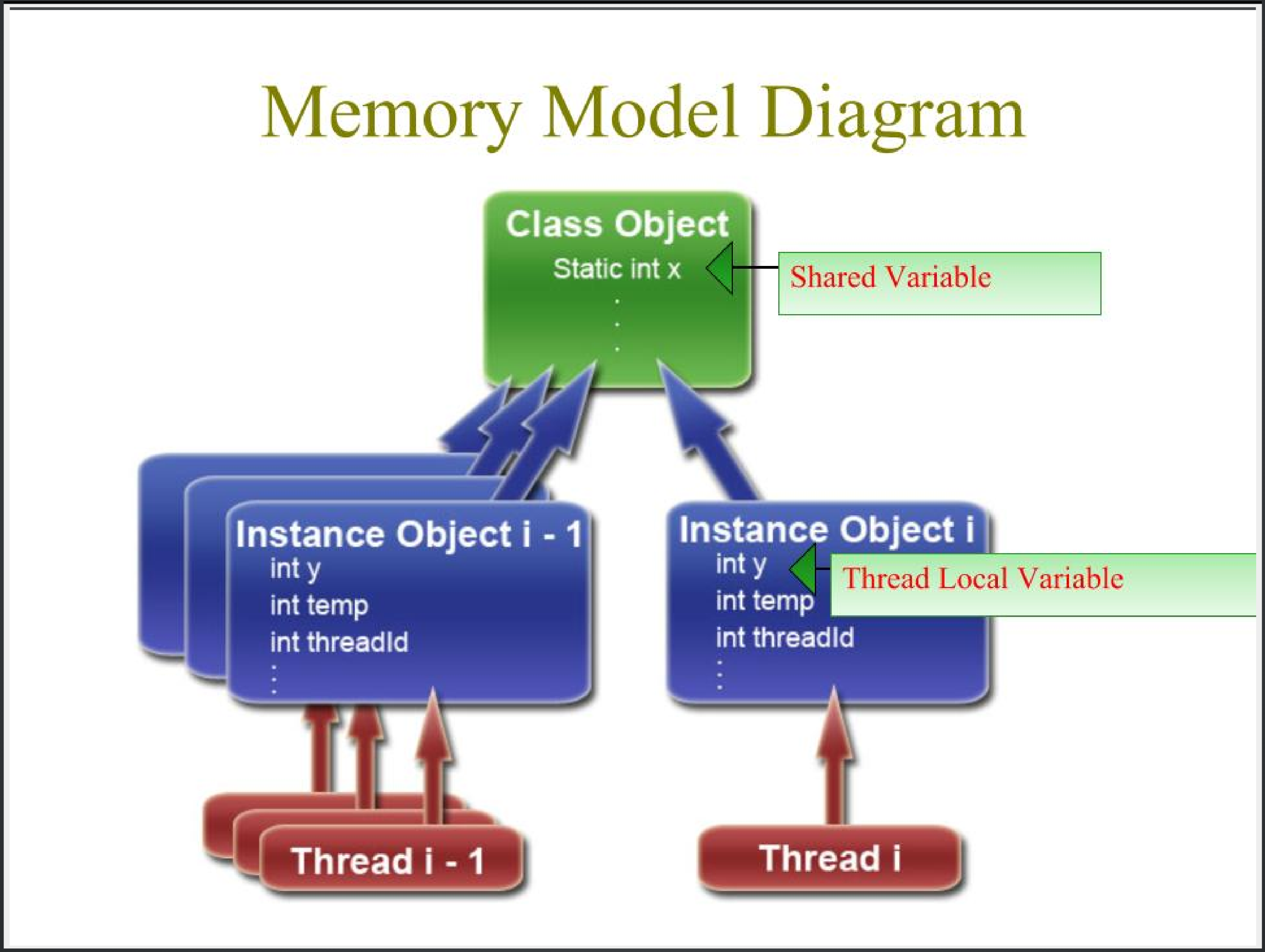
SHARED VS LOCAL VARIABLES

- 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

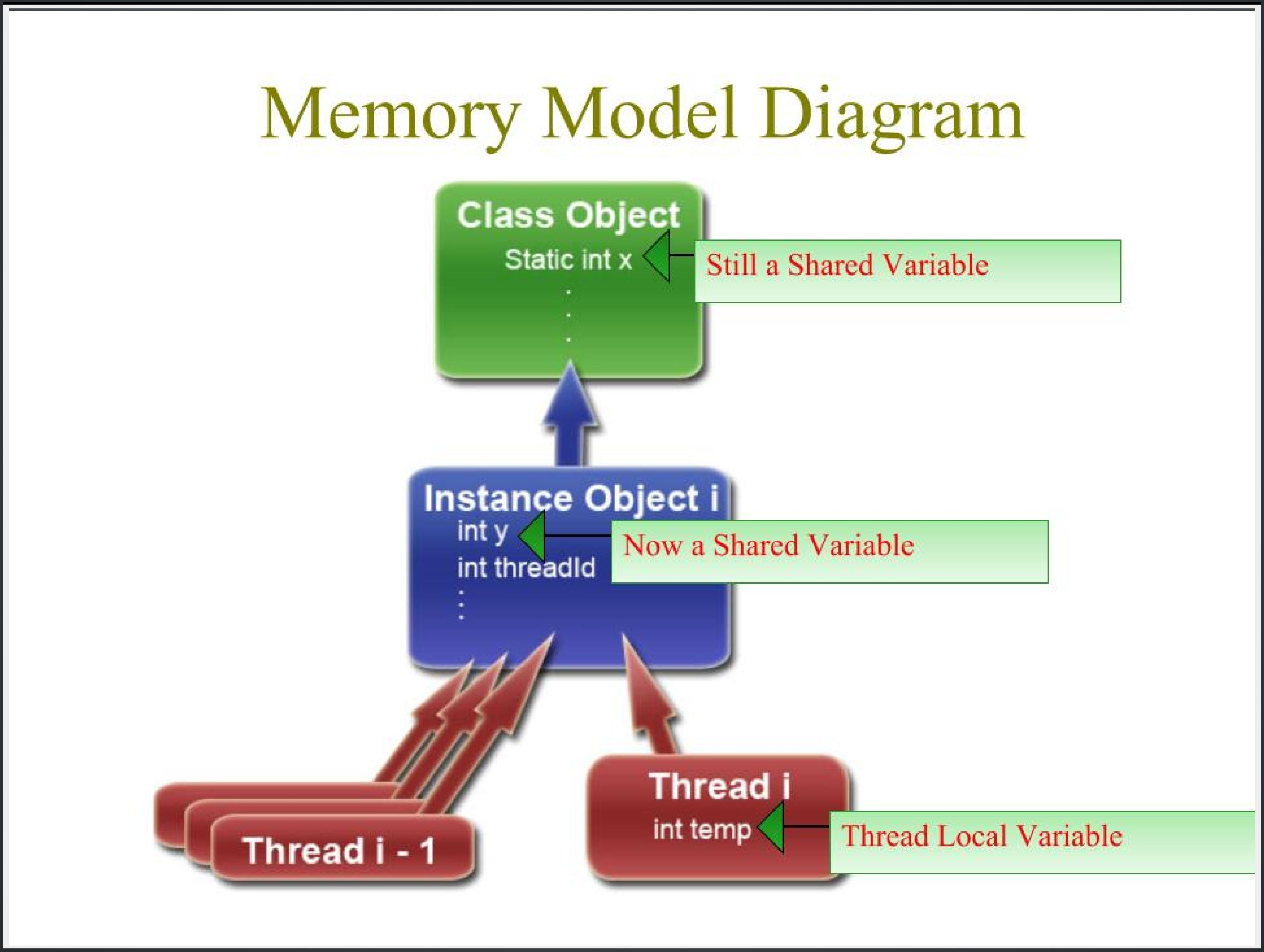
ONE THREAD / ONE OBJECT



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

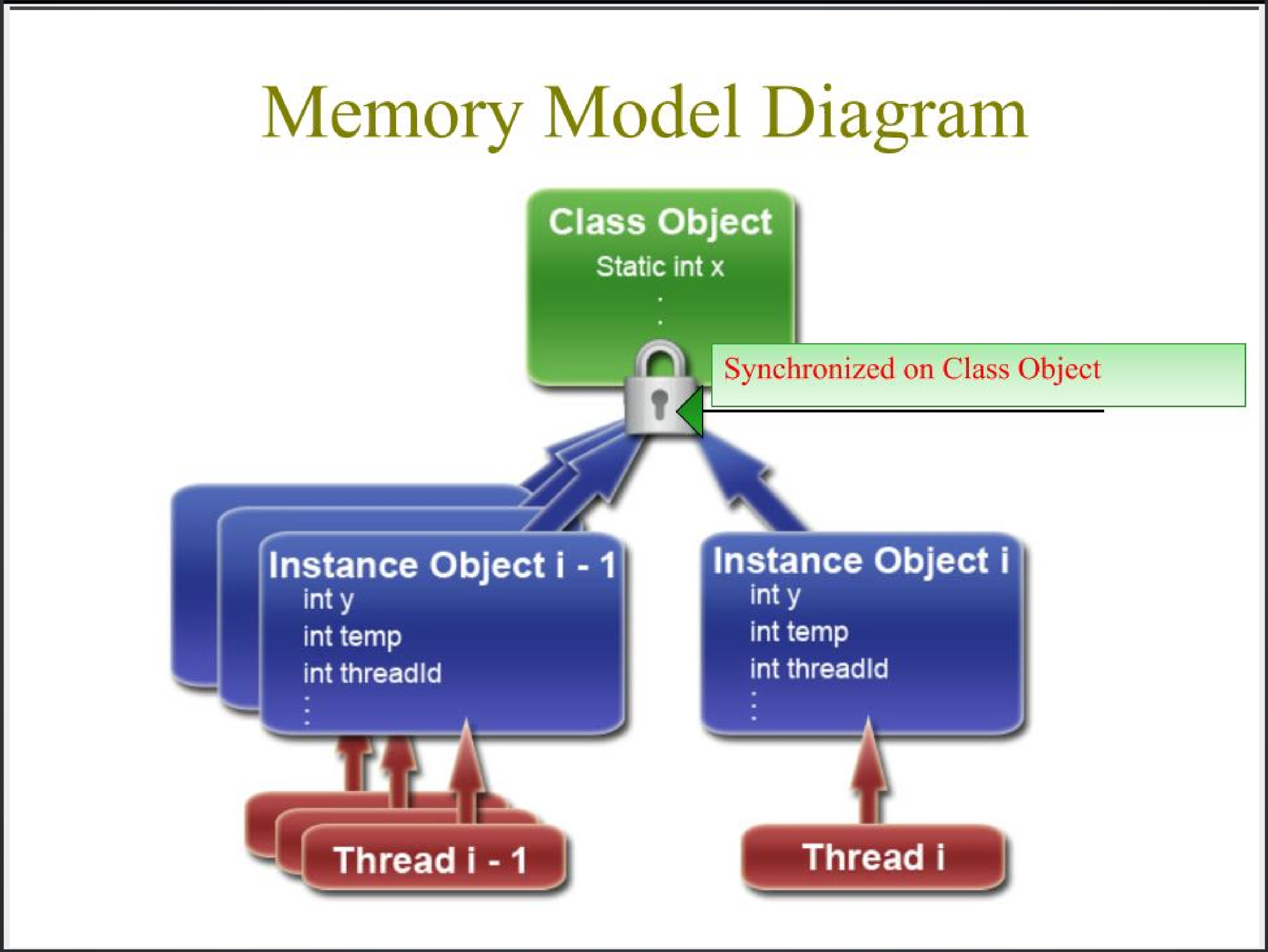
SYNCHRONIZED METHODS

- 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



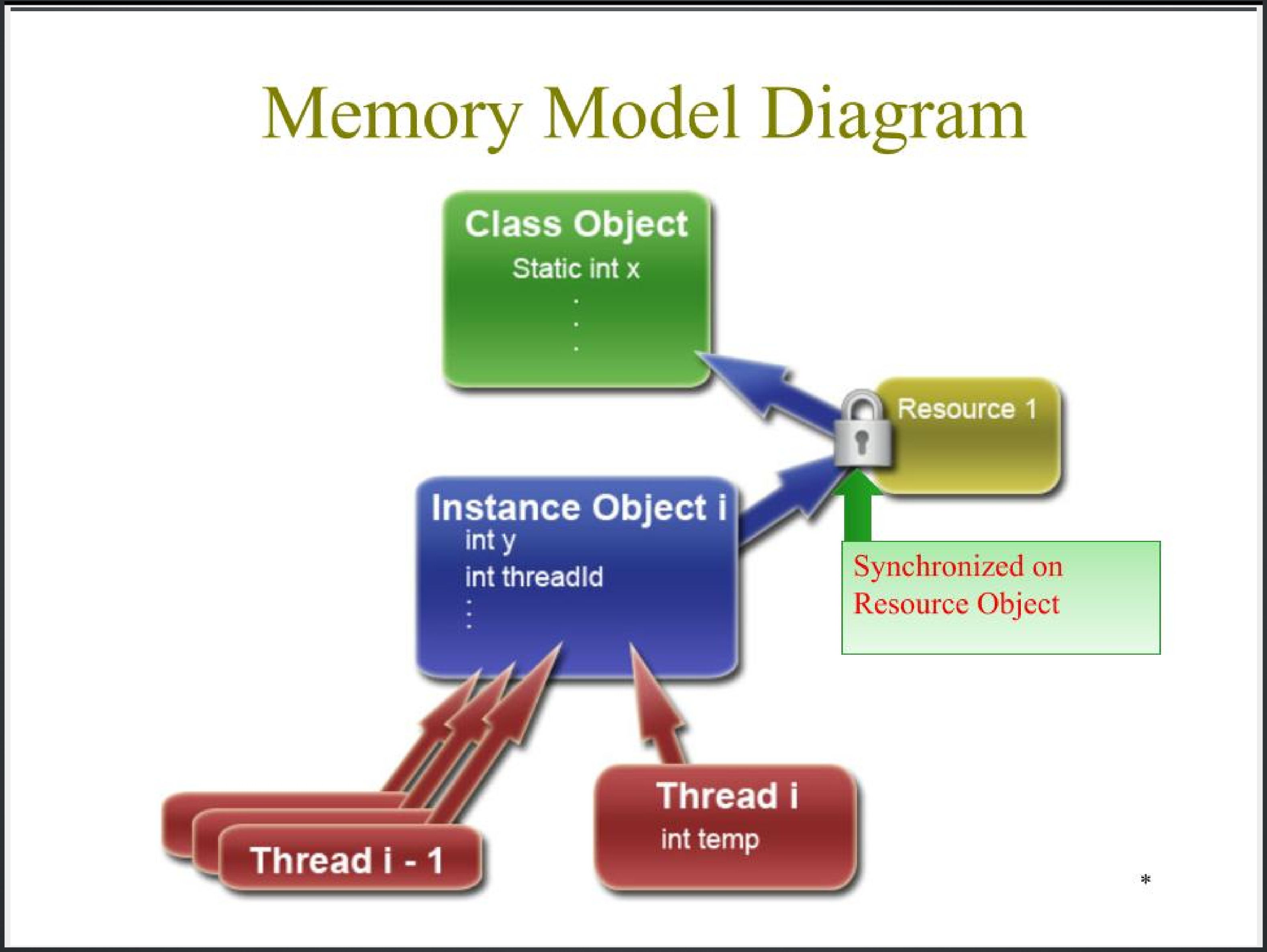
SYNCHRONIZED STATEMENTS

- 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++;
        }
    }
}
```

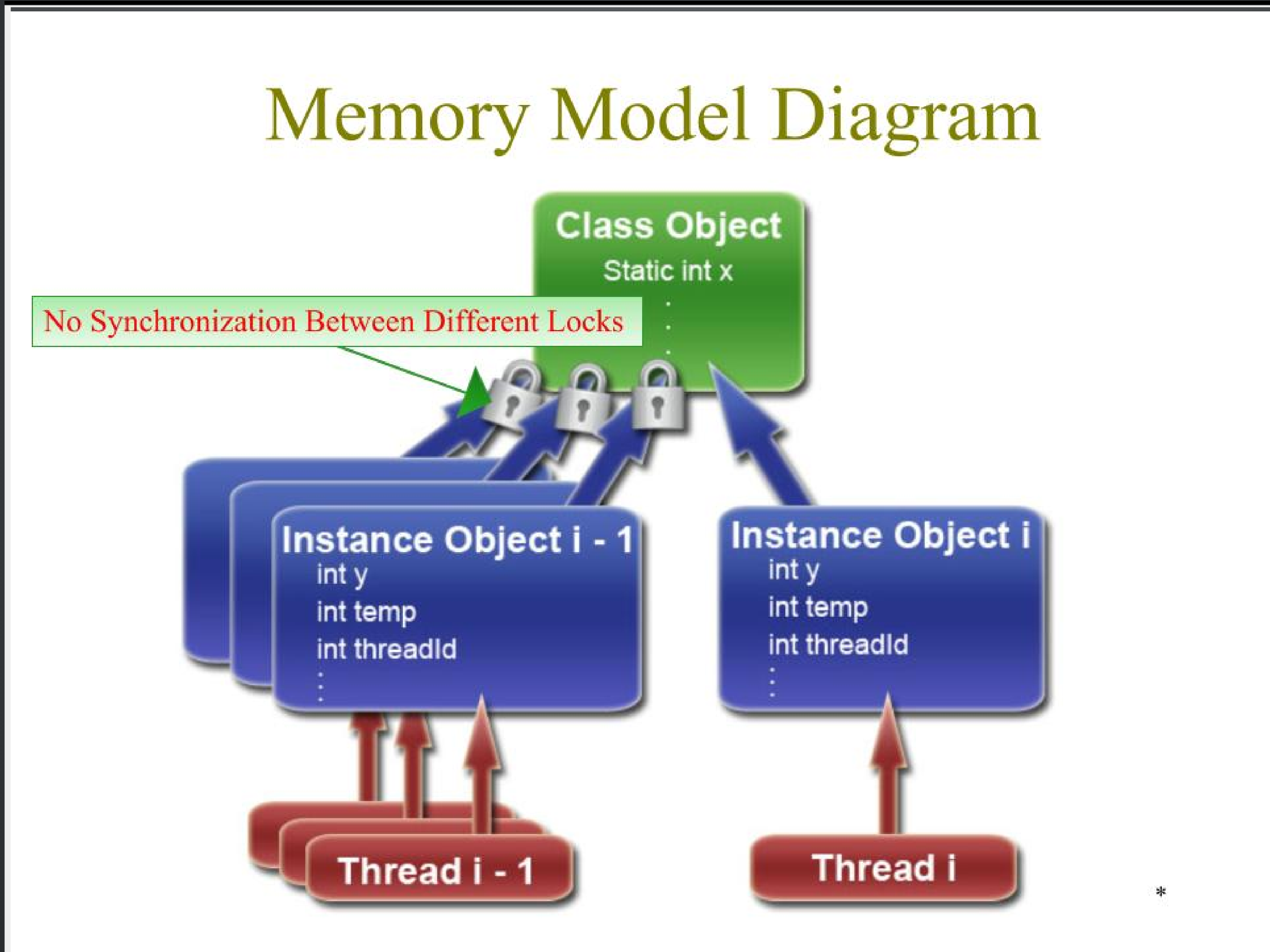
MEMORY MODEL



CODE EXAMPLE

```
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()

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

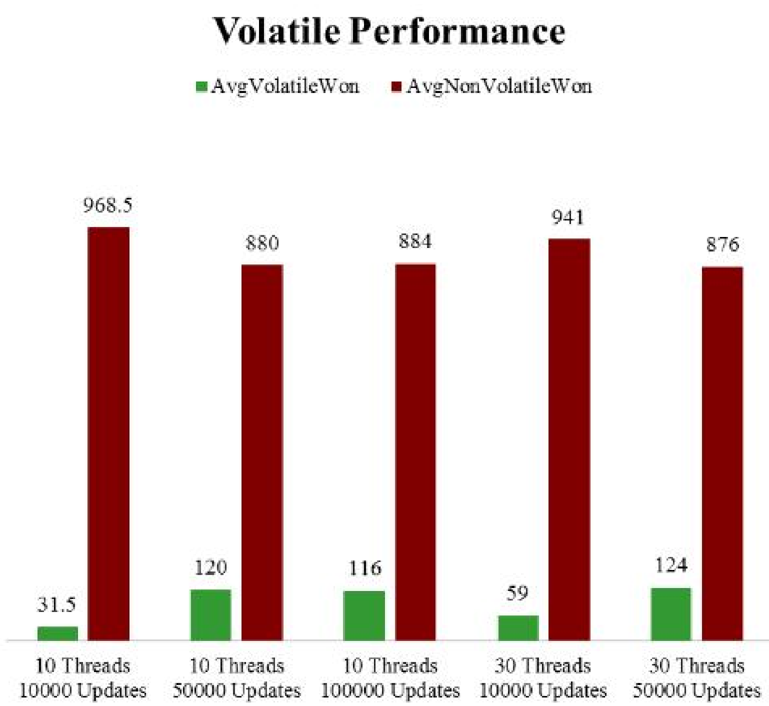
EXAMPLE

```
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

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…

*

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!

ATOMIC PRIMATIVES

- Library of special Atomic Classes provided by Oracle in the Standard Library
- Internally, the atomic classes use compare-and-swap (CAS)

ATOMICINTEGER, ATOMICBOOLEAN, ATOMICLONG, AND ATOMICREFERENCE

- incrementAndGet()
- updateAndGet(lambdaExpression)
- accumulateAndGet(lambdaExpression)

LONGADDER, LONGACCUMULATOR

- Just like the equivalent Atomic Class but
- Maintains a set of variables internally to reduce contention over threads
- Preferable when updates from multiple threads are more common than reads
- The drawback is higher memory consumption from the extra variables

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
        }
    }
}
```

FUTURES (JAVA 8)

- Callable interface: Lke the Runnable interface *but*
- Returns a value
- Callables can be submitted to executor services just like runnables
- Executor.submit() does not wait until the task completes
- The executor service cannot return the result of the callable directly
- Instead the executor returns a special result of type *Future*

FUTURES CONTINUED

- Future can be used to retrieve the actual result at a later point in time
- Future.isDone() : check if the result is ready yet
- Future.get() : blocks the current thread and waits until the callable completes before returning the actual result

BATCHING FUTURES

- Executors support batch submitting of multiple callables at once via Executor.invokeAll()
- Accepts a collection of callables and returns a list of futures

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

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.

STRING ALTERNATIVES COMPARISION

	String	StringBuilder	StringBuffer
Thread Safe	Yes	Yes	No
Synchronization Overhead	Low	High	None
Memory Usage	High	Low	Low

REFERENCES AND FURTHER READING

- The Official Java Tutorials on Concurrency
- Java Concurrency Package API
- Java Memory Model
- Java Concurrency in Practice (Book)
- Java Performance Tuning
- Java Specification on Final Fields
- Comparison of String Class Alternatives
- Five things you didn't know about Java Multithreading
- Thread Pool Performance Tuning
- Implementing a Monitor Lock in Java
- Prisoner Problem Case Study

THANK YOU