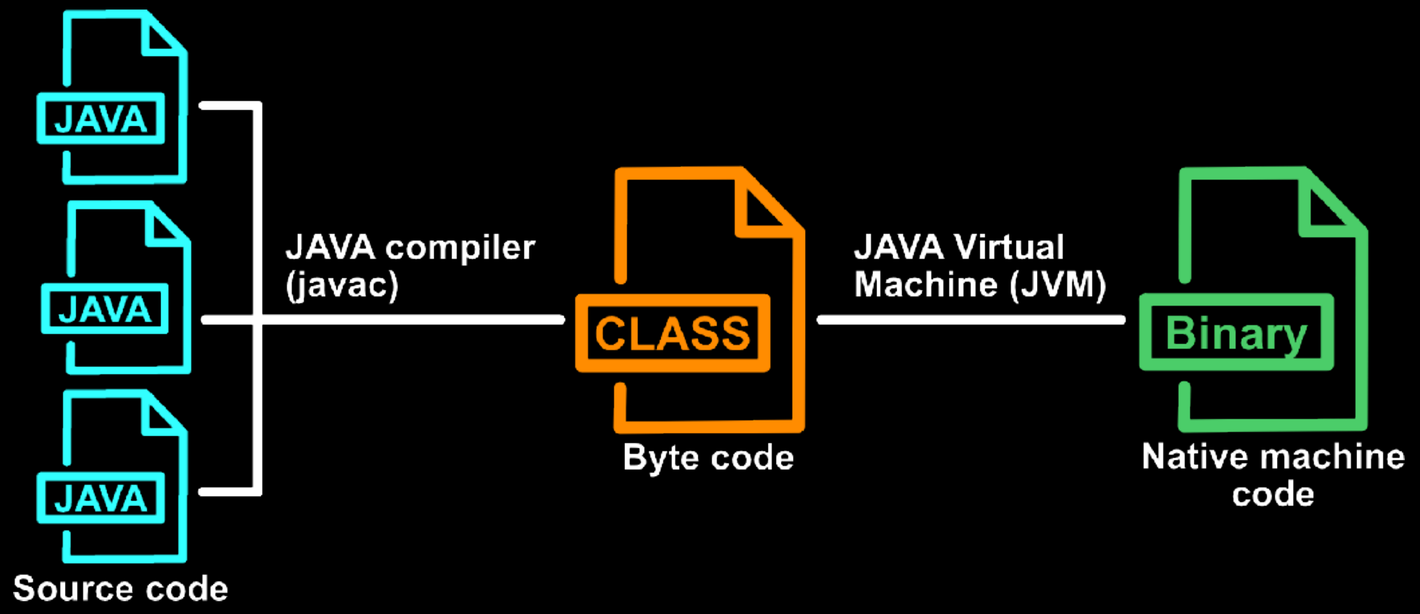
Parallel Programming in Java

“Java is still not dead—and people are starting to figure that out.”

Neco Kriel (n9702806)

Java Programming
The Environment

- Object Orientated Programming (OOP)
- Java compiled code can run on any Java supported platform



Java Programming
Designing Parallel Programs

- The *java.util.concurrent* package provides support for concurrency
- *Locks* are provided to protect shared resources in multithreaded programs

```
public synchronized void critcalSection() {
    // critical code goes here
    //
    //
}
```

```
public int add(int val_1, int val_2) {
    synchronized (this) {
        return val_1 + val_2;
    }
}
```

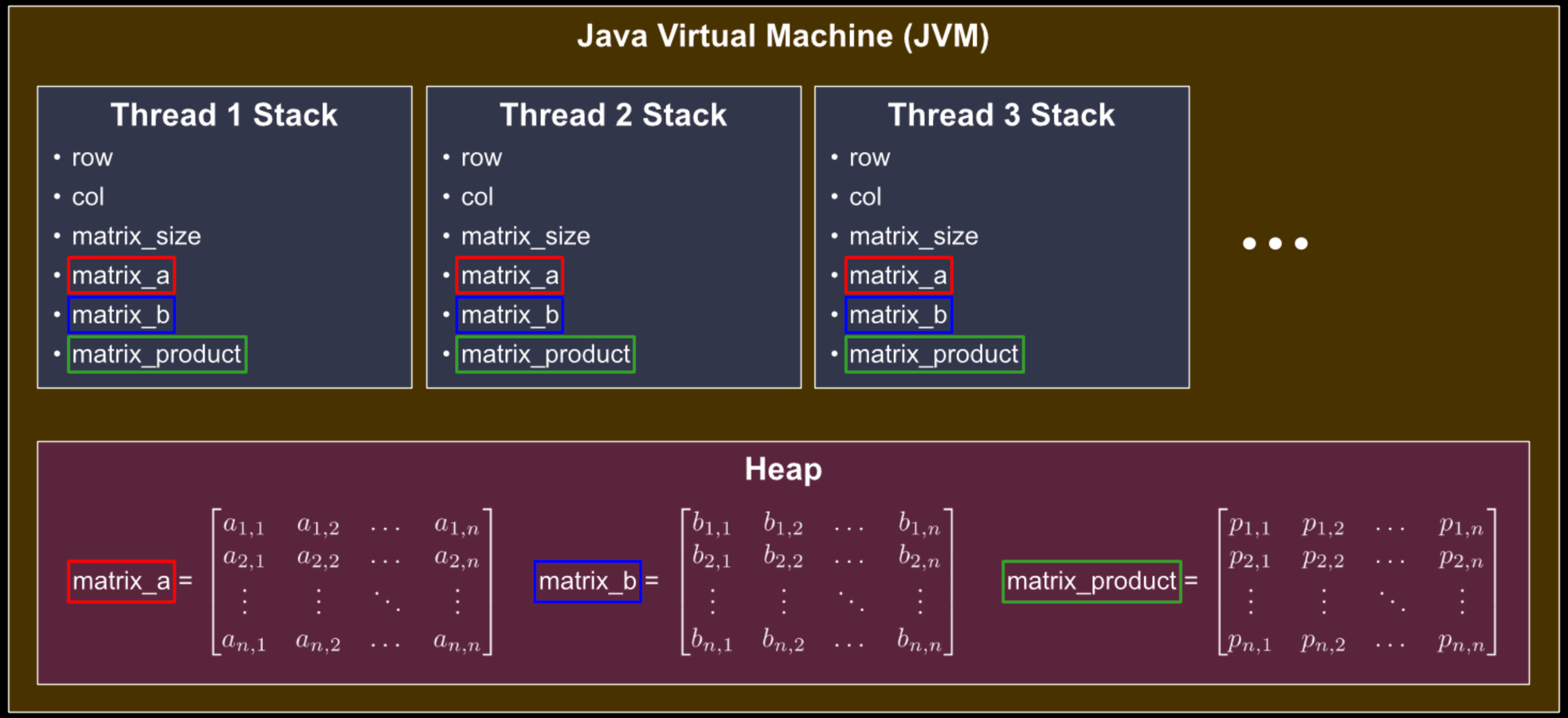
Code Snippets: the keyword *synchronized* makes a code block thread safe.

Java Programming
The Java Memory Model in Multithreaded Programs

- Operations on shared resources should be *`atomic`*
  - *`Atomic operation`*: a single unit of work that can't be interrupted
  - *`AtomicIntegerArray`*, *`AtomicLong`*, ect
  - *`get()`*, `set()`, *`getAndIncrement()`*, *`compareAndSet()`*, ect
- *`Primitive`* variables that are local to a thread are stored on the thread's stack
- *`Static`* variables are stored on the heap
  - Allows threads to share resources
  - Atomic variables ensure that shared resources are thread safe

Java Programming

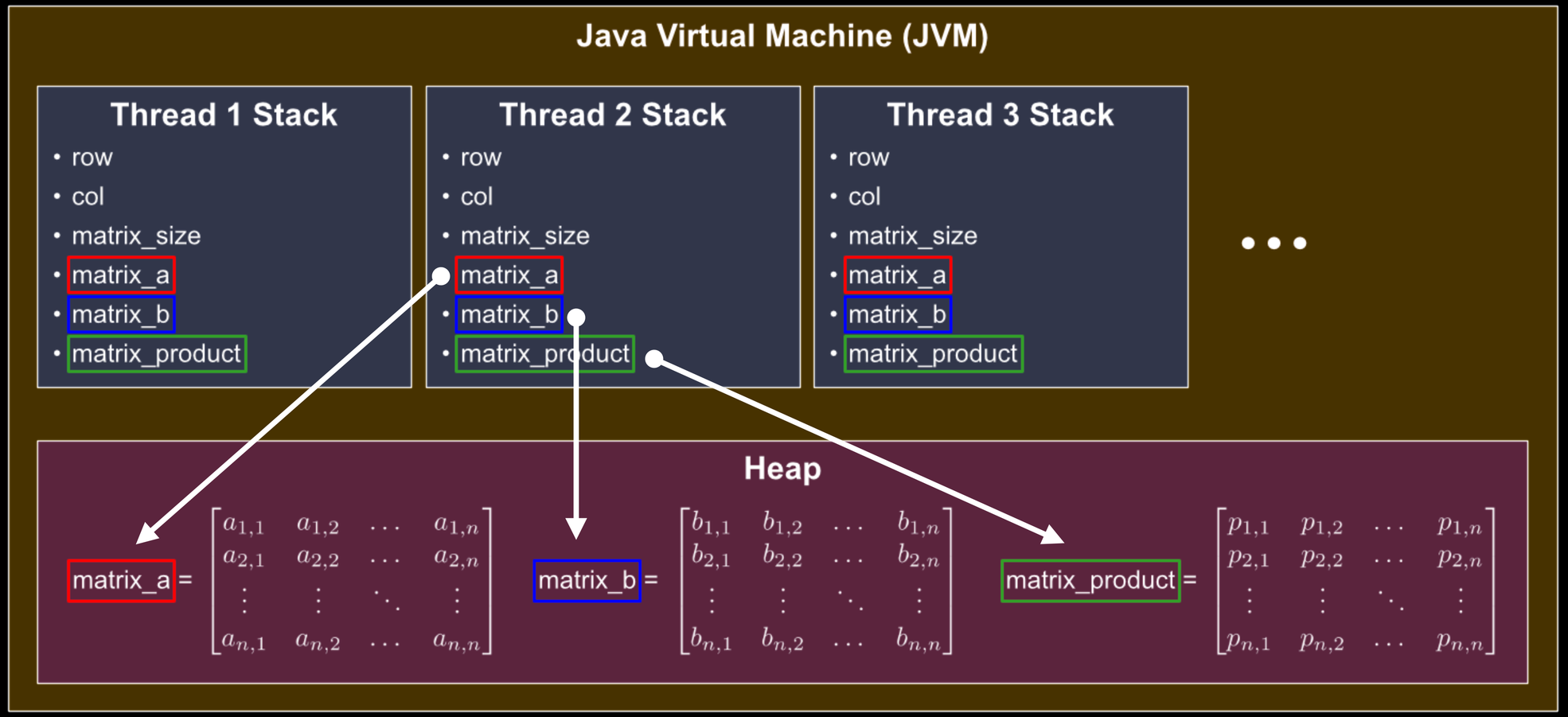
Memory Management



The Java Memory Model

Java Programming

Memory Management



Static variables are stored on the heap.

Java Programming
Parallelism with Multithreading - Defining the Task

- Two main ways of defining a task to be executed by multiple threads
  - Extending the *`Thread`* class
    - Each thread creates a unique copy of each object
  - Implementing the *`Runnable`* interface
    - Threads share the same object instance
- It is good practice to implement the *`Runnable`* interface
  - A class can only extend one other class in Java (i.e. multiple inheritance is not supported)
  - Avoid overhead associated with coupling your class with the Thread class

Java Programming

Parallelism with Multithreading - Executing the Task

- The Java *`ExecutorService`* abstracts away the low-level complexities associated with **scheduling**, **executing** and **managing** tasks
  - Creates a re-usable pool of threads
  - Runs submitted tasks asynchronously
  - Ensures predictability
- Create a pool of threads: *`newFixedThreadPool()`*
- Wait for tasks to finish execution: *`awaitTermination()`*

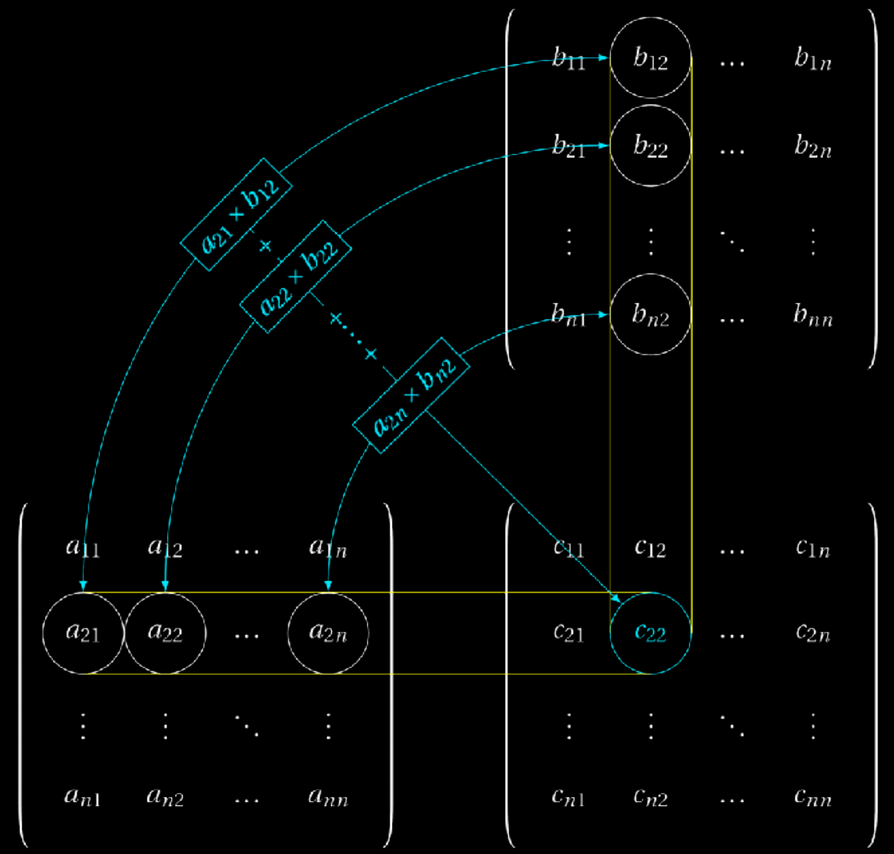
Parallelism with Multithreading

Example: Matrix Multiplication

- Compute each inner-product concurrently

```
for Each matrix A row i do
    for Each matrix B column j do
        Compute the inner-product of matrix A's row i and matrix B's
         column j;
    end
end
```

Algorithm 1: Calculating the product of matrices A and B



Parallelism with Multithreading

Example: Matrix Multiplication

- Compute each inner-product concurrently

```
for Each matrix A row i do
    for Each matrix B column j do
        Compute the inner-product of matrix A's row i and matrix B's
         column j;
    end
end
```

Algorithm 1: Calculating the product of matrices A and B

Parallelism with Multithreading

Example: Matrix Multiplication

- Parallelise the inner-loop: compute each inner-product concurrently

```
// Send the information for matrix multiplication in parallel
public static void ParallelMatrixDistributor(ExecutorService executor, int matrix_size,
        AtomicIntegerArray[] matrix_a, AtomicIntegerArray[] matrix_b,
        AtomicIntegerArray[] matrix_product) {
    // for each matrix row
    for (int i = 0; i < matrix_size; i++) {
        // and column elemnt in the row
        for (int j = 0; j < matrix_size; j++) {
            // calculate the product result for the PRODUCT[row, column] value
            ParallelMultiplier multiplyThread = new ParallelMultiplier(i, j, matrix_size, matrix_a[i], matrix_b, matrix_product);
            // submit the thread to pool for row-wise multiplication
            executor.execute(multiplyThread);
        }
    }
}
```

Parallelism with Multithreading

Example: Matrix Multiplication

- Parallelise the inner-loop: compute each inner-product concurrently

```
// Send the information for matrix multiplication in parallel
public static void ParallelMatrixDistributor(ExecutorService executor, int matrix_size,
        AtomicIntegerArray[] matrix_a, AtomicIntegerArray[] matrix_b,
        AtomicIntegerArray[] matrix_product) {
    // for each matrix row
    for (int i = 0; i < matrix_size; i++) {
        // and column elemnt in the row
        for (int j = 0; j < matrix_size; j++) {
            // calculate the product result for the PRODUCT[row, column] value
            ParallelMultiplier multiplyThread = new ParallelMultiplier(i, j, matrix_size, matrix_a[i], matrix_b, matrix_product);
            // submit the thread to pool for row-wise multiplication
            executor.execute(multiplyThread);
        }
    }
}
```

Execute the job

Create a new thread job

Parallelism with Multithreading

Example: Matrix Multiplication

- Calculating the inner-product

Initialise `total sum`;
for *Each element k in matrix A's ith row and matrix B's jth column* **do**
  Sum the product of elements k and `total sum`;
end

Algorithm 2: Calculating the inner-product of row i in matrices A and column j in matrix B

Parallelism with Multithreading

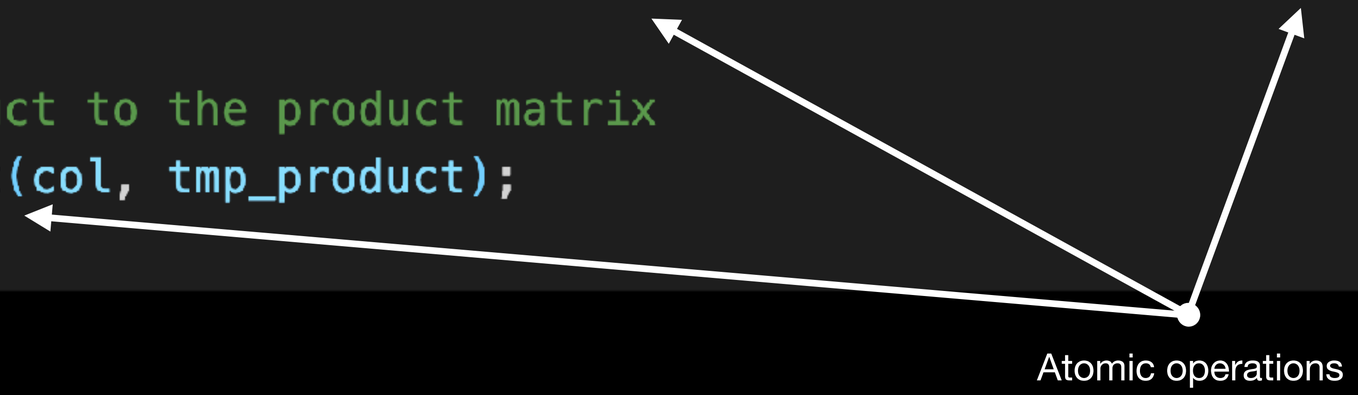
Example: Matrix Multiplication

```
// calculate result stored at matrix_product[row, col]
public void run() {
    // initialise the element total
    int tmp_product = 0;
    // evaluate the dot product of the vectors A[row, :] and B[:, col]
    for (int elem = 0; elem < matrix_size; elem++) {
        // find the total tmp_product
        tmp_product = tmp_product + (matrix_a.get(elem) * matrix_b[elem].get(col));
    }
    // assign the tmp_product to the product matrix
    matrix_product[row].set(col, tmp_product);
}
```

Parallelism with Multithreading

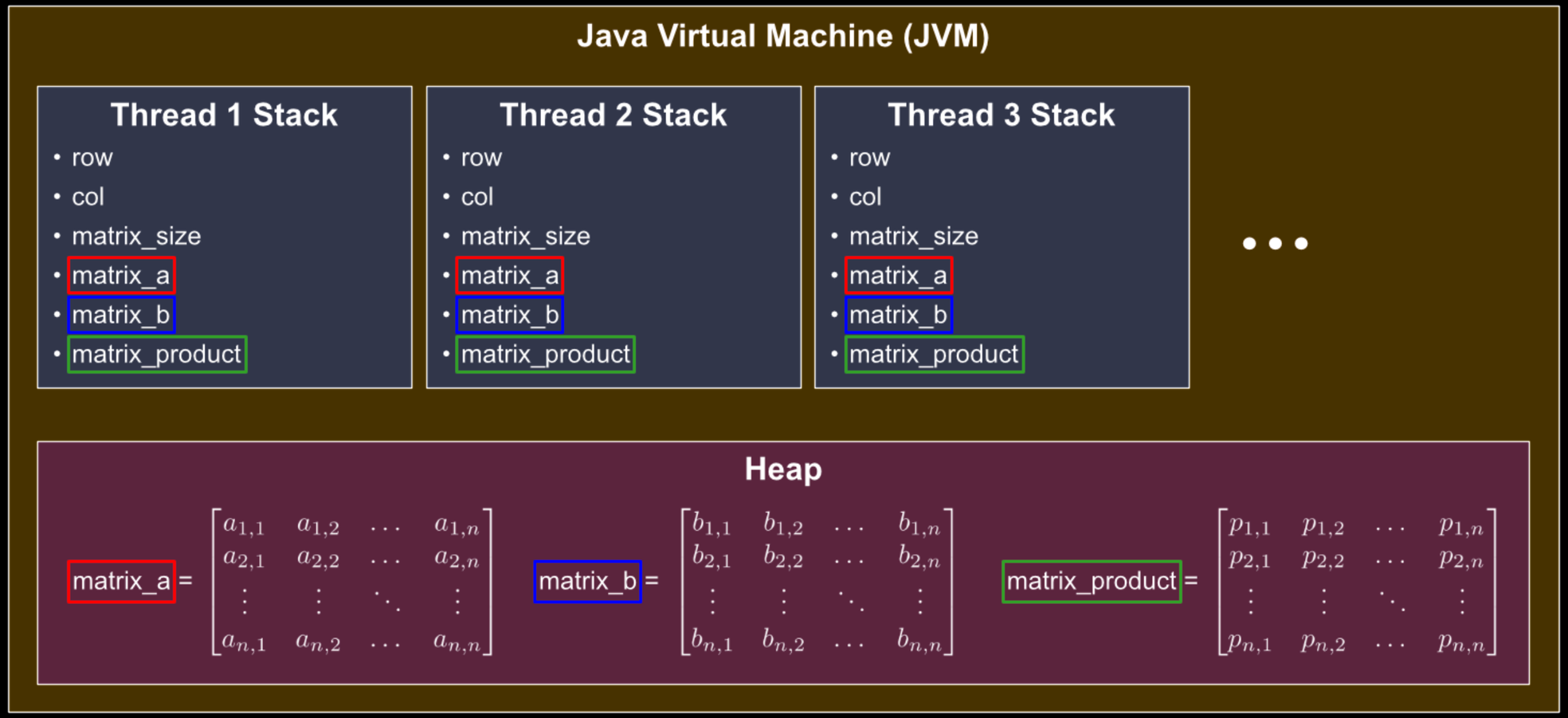
Example: Matrix Multiplication

```
// calculate result stored at matrix_product[row, col]
public void run() {
    // initialise the element total
    int tmp_product = 0;
    // evaluate the dot product of the vectors A[row, :] and B[:, col]
    for (int elem = 0; elem < matrix_size; elem++) {
        // find the total tmp_product
        tmp_product = tmp_product + (matrix_a.get(elem) * matrix_b[elem].get(col));
    }
    // assign the tmp_product to the product matrix
    matrix_product[row].set(col, tmp_product);
}
```



Java Programming

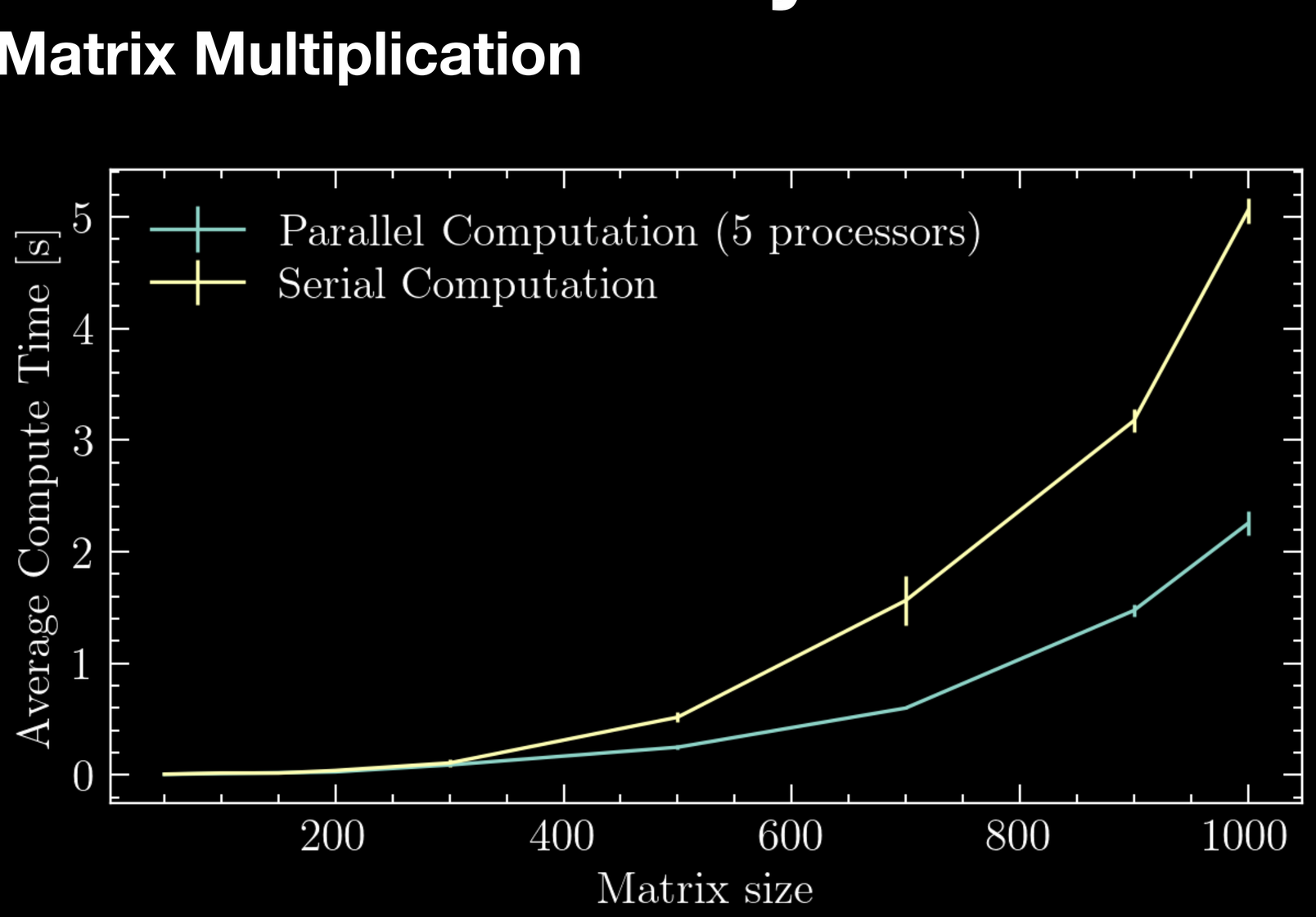
Example: Matrix Multiplication



The Java Memory Model

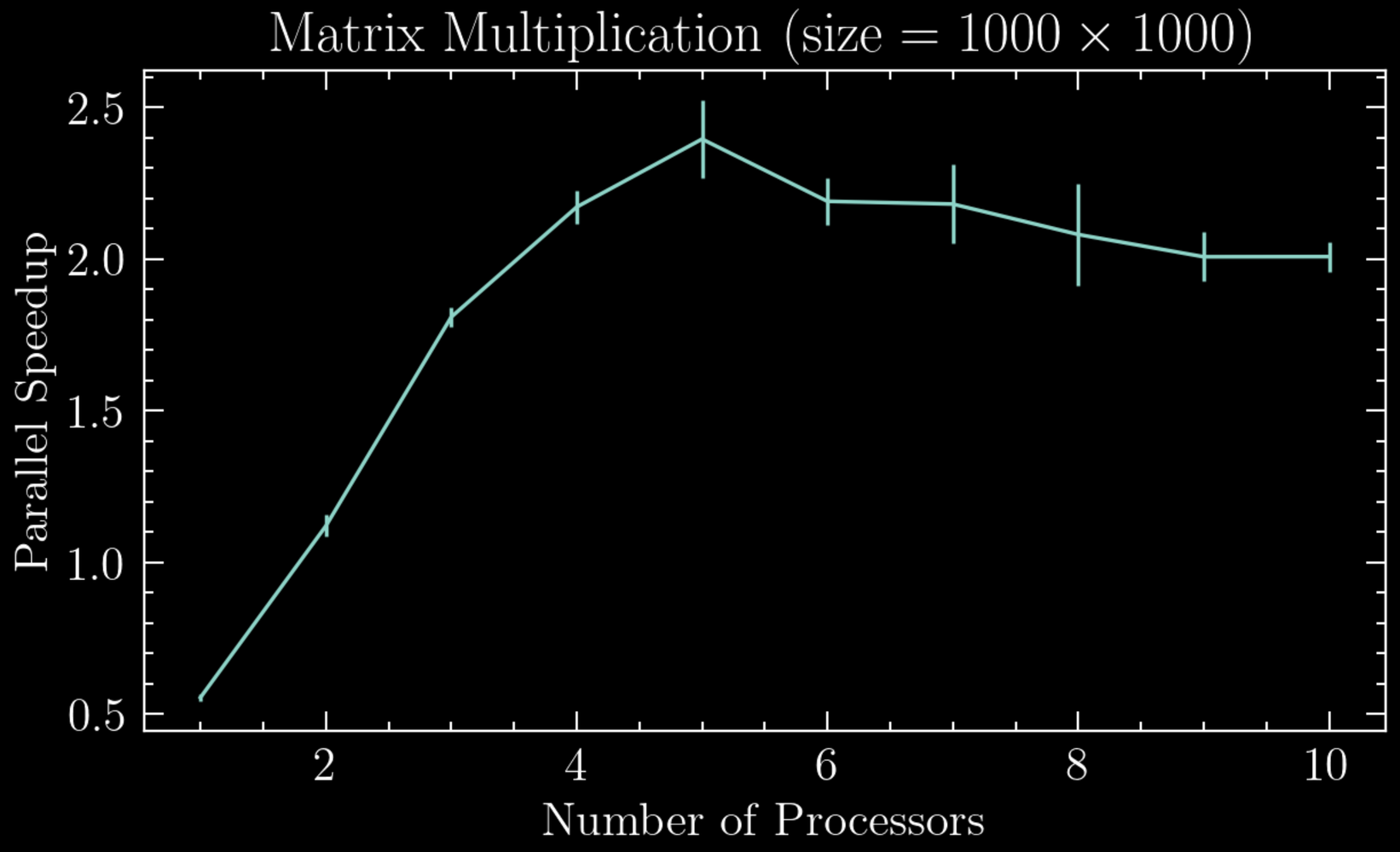
Performance & Scalability

Example: Matrix Multiplication



Performance & Scalability

Example: Matrix Multiplication



Summary

Key Results & Takeaways

- The *java.util.concurrent* package provides support for concurrency
- Resources shared by threads should be protected and the operations on them should be *atomic*
- Implement the *Runnable* interface
    - Extending the *Thread* class has performance overhead associated
- Manage the execution of tasks with the *ExecutorService*

Thank you

Useful Resources

- *Lesson: Concurrency (The JavaTM Tutorials > Essential Classes)*. (2020). Oracle. https://docs.oracle.com/javase/tutorial/essential/concurrency/

- Winterberg, B. (2015, April 30). *Java 8 Concurrency Tutorial: Synchronization and Locks*. Winterbe. https://winterbe.com/posts/2015/04/30/java8-concurrency-tutorial-synchronized-locks-examples/

- GmbH, V. V. L.-. (2016). *Java concurrency (multi-threading) - Tutorial*. Vogella. https://www.vogella.com/tutorials/JavaConcurrency/article.html

- Jenkov, J. (2020). *Java Concurrency and Multithreading Tutorial*. Tutorials. http://tutorials.jenkov.com/java-concurrency/index.html