Concurrent Programming: Threads

CS 18000

Sunil Prabhakar

Department of Computer Science

Purdue University

Objectives

- This week we will get introduced to concurrent programming
  - Creating a new thread of execution
  - Waiting for threads to terminate
  - Thread states and scheduling
  - sleep() and yield()
  - Simple synchronization among threads

One-track mind?

- Often, in real life we perform multiple tasks at the same time
  - Doing the laundry
  - Making a pot of coffee
- This is more efficient.
- Our programs thus far have had a single track (thread) of execution
  - at any point in time, there is only one statement being executed
  - not always efficient -- can stall (e.g., GUIs)

Multiple Concurrent Threads

- There are many instances in computing where we can benefit from multiple concurrent threads of execution.
- For example:
  - GUI responsiveness. The GUI should not freeze while performing time-consuming operations.
  - Liveness in games: display shouldn't "lock up"
  - Exploiting available processing: speeding up processing by using all computing cores

Motivation: GUIs

- Consider a GUI event which causes some time-consuming processing to execute, e.g. waiting for the other player to make a move
- While this processing is going on, the GUI will "lock up"
  - This is not desirable from a user experience point of view
- How can we prevent this?
  - *Solution: perform non-GUI processing without locking up GUI thread*

Motivation: Asynchrony

- Some applications are inherently asynchronous
- Consider the Client-Server model:
  - A server needs to listen for connecting clients while serving the currently connected clients
    - a call to `accept()` blocks until a new client connects
  - Each client responds when it is ready
    - a call to `readLine()` for one client blocks until that client sends a message or closes the connection
  - We could use timeouts, but that results in a slow down and useless waiting doing nothing

Motivation: Exploiting Multiple Cores and Processors

- Due to recent hardware trends, modern computers have multiple CPUs (cores or processors)
- If there is only a single thread of execution, only one CPU or core is used by our program.
- How do we exploit these other CPUs?
- Consider
  - the initialization of a large array
  - searching for an item in a large array
- *Solution: Split array into pieces and initialize (search) each piece concurrently.*

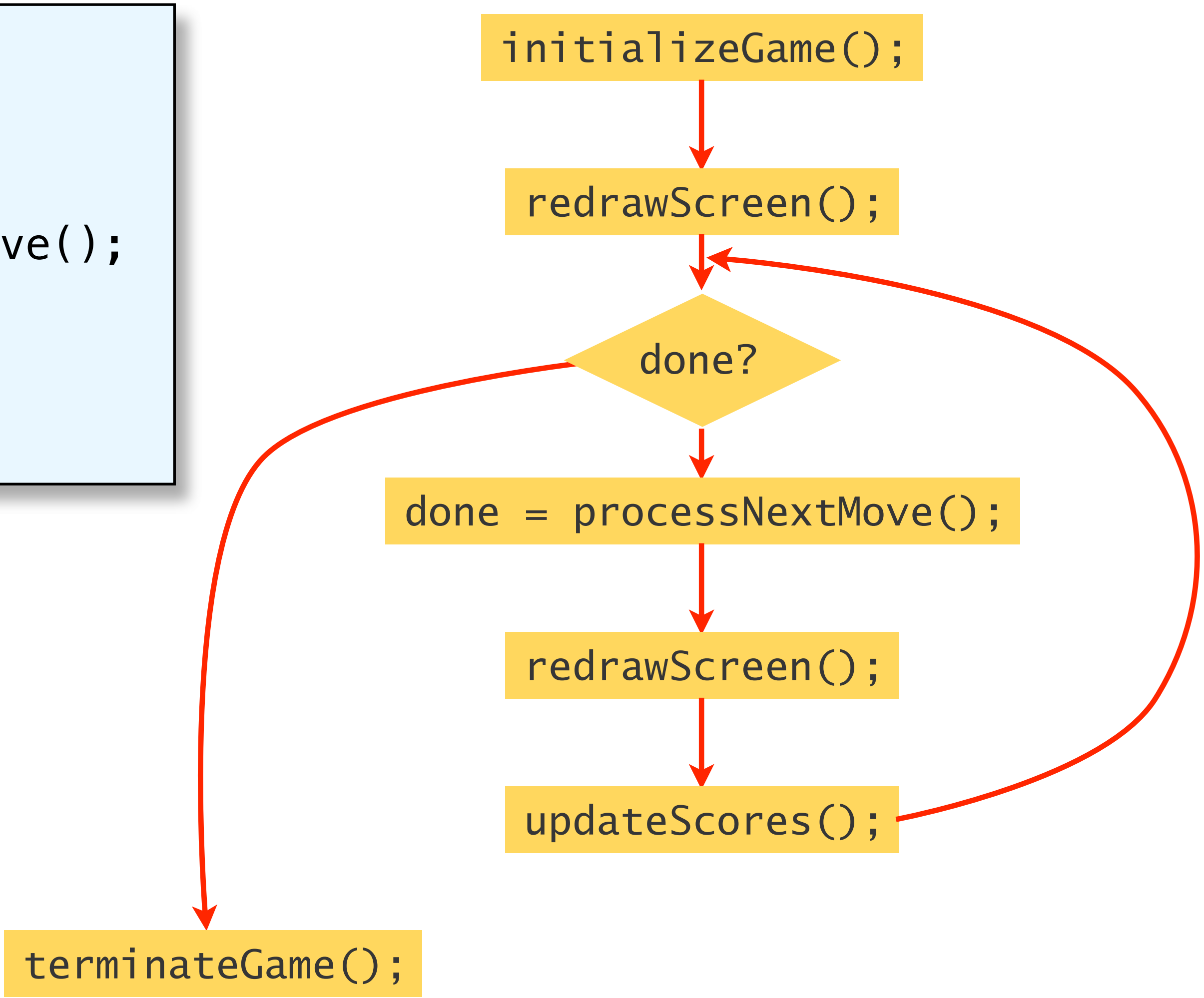
Sequential Processing

- In a non-concurrent (sequential) program there is only one thread: the main thread.
- This thread executes the main method and then terminates.
  - the flow of control is determined by the main method
  - Note that with GUI elements,
    - a separate thread handles events: Event Dispatch Thread

Game: sequential version

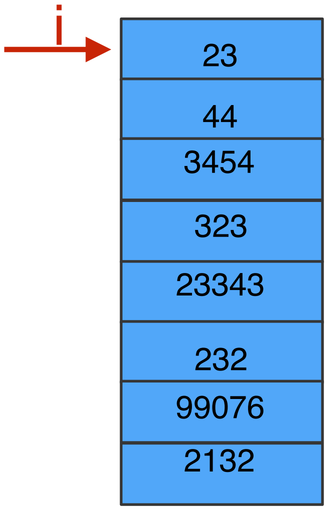
```
initializeGame();
redrawScreen();
boolean done=false;
while(!done) {
    done = processNextMove();
    redrawScreen();
    updateScores();
}
terminateGame();
```

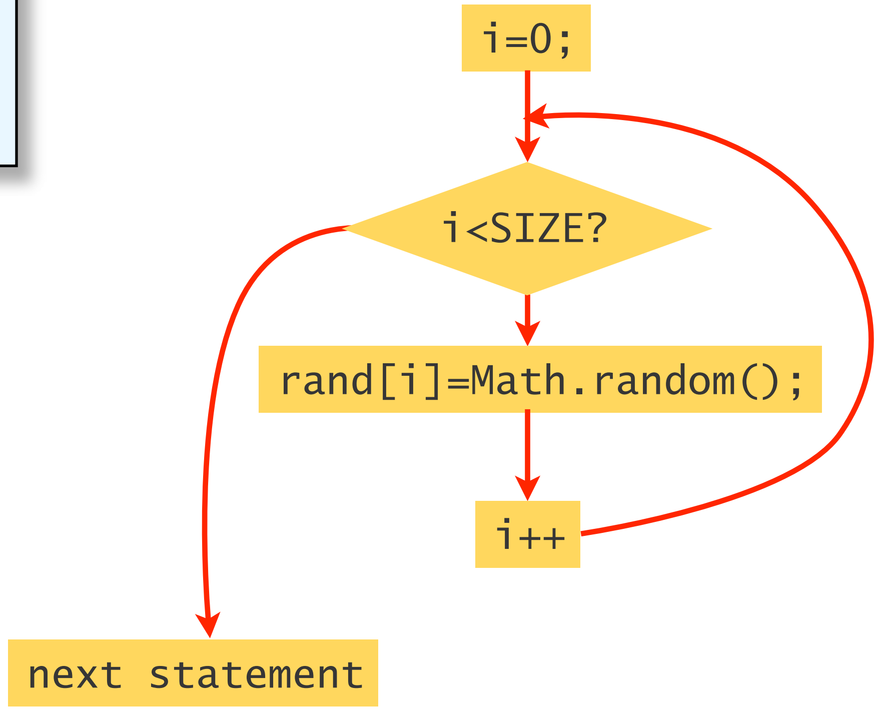
Screen frozen while waiting for user input.



Array: sequential version

```java
final int SIZE = 1000000;
double[] rand = new
double[SIZE];
for(int i=0;i<SIZE;i++)
    rand[i]= Math.random();
```



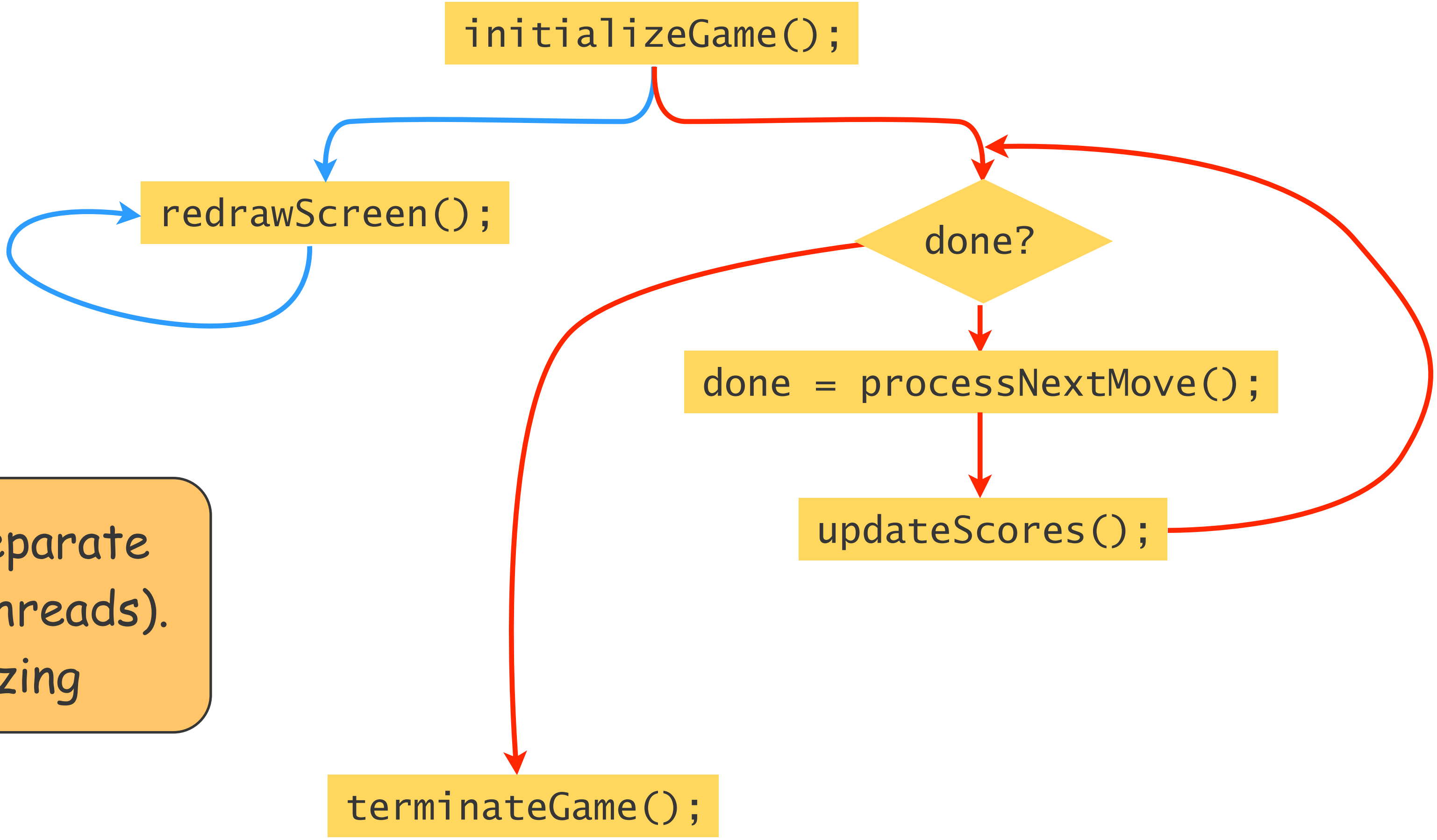


Only one thread -- may take long time even though some CPUs are idle

Concurrent Processing

- How do we create a separate thread of execution?
- The Thread class provides a facility for creating separate threads.
  - Declare a class to be a descendant of Thread
  - Override the run() method to perform the necessary task(s) for the new thread
- When the start() method is called on an object of this class, the thread starts executing concurrently

Game: concurrent version



Note: separate tasks (threads). No freezing

Game: concurrent version

The class extends the Thread class

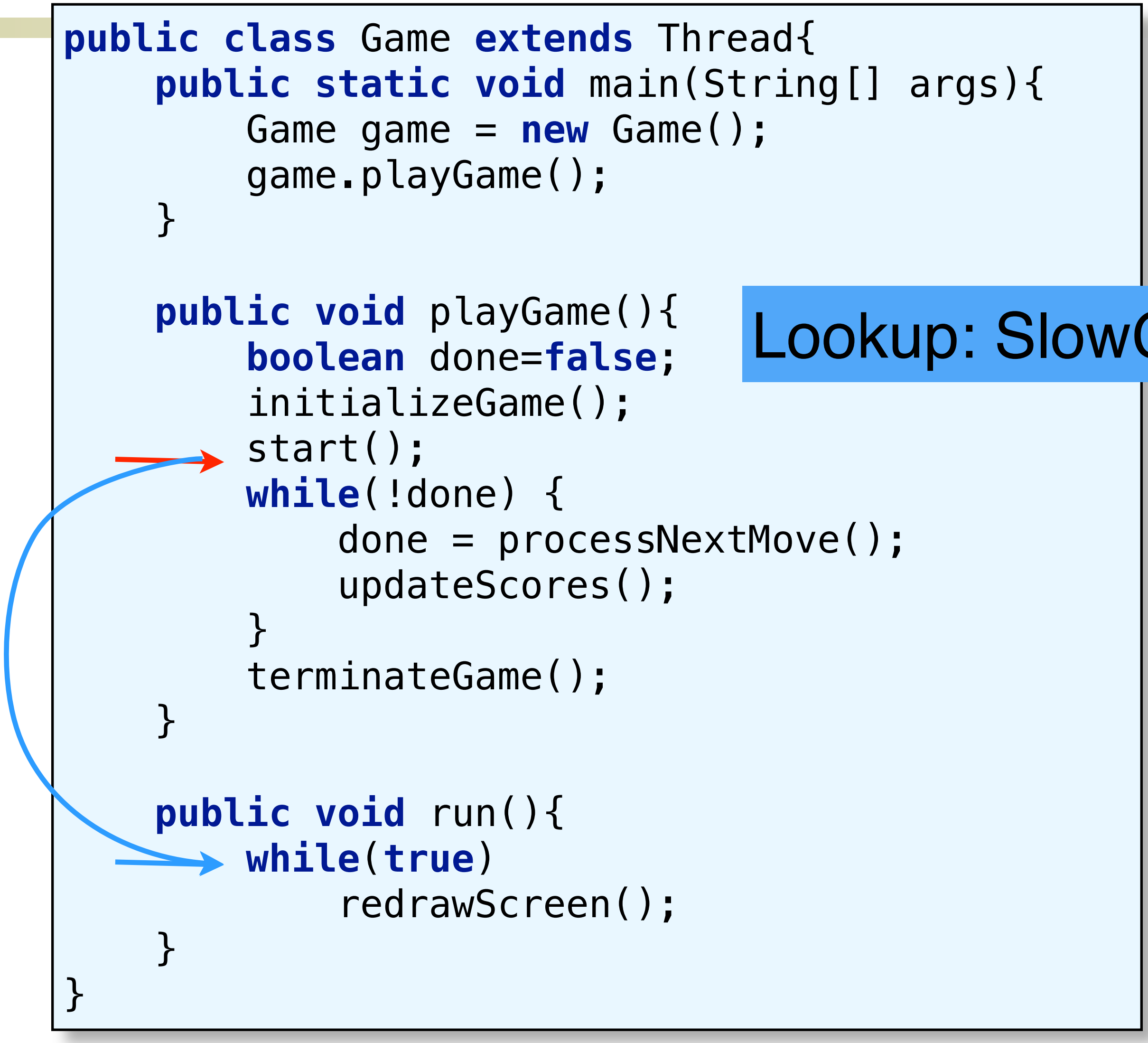
Create a Thread object.

To start a separate thread, we call start() on the thread object

The class overrides the run() method

```
public class Game extends Thread  {
    public static void main(String[] args){
        Game game = new Game();
        game.playGame();
    }

    public void playGame(){
        boolean done=false;
        initializeGame();
        start();
        while(!done) {
            done = processNextMove();
            updateScores();
        }
        terminateGame();
    }

    public void run(){
        while(true)
            redrawScreen();
    }
}
```

Game: Concurrent Version

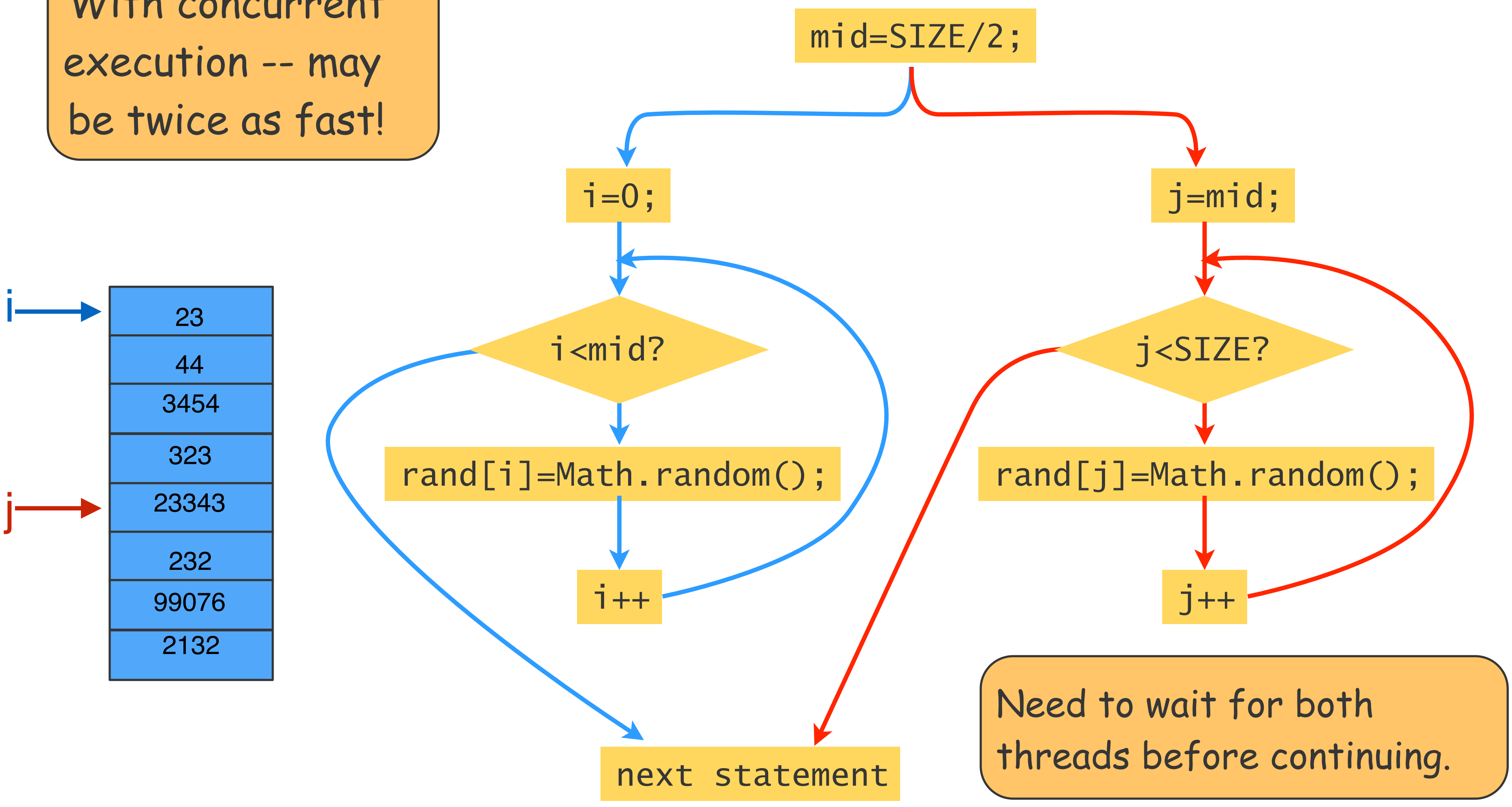


```
public class Game extends Thread{
    public static void main(String[] args){
        Game game = new Game();
        game.playGame();
    }

    public void playGame(){
        boolean done=false;
        initializeGame();
        start();
        while(!done) {
            done = processNextMove();
            updateScores();
        }
        terminateGame();
    }

    public void run(){
        while(true)
            redrawScreen();
    }
}
```

Lookup: SlowGUI.java

Array: concurrent version

With concurrent execution -- may be twice as fast!



i

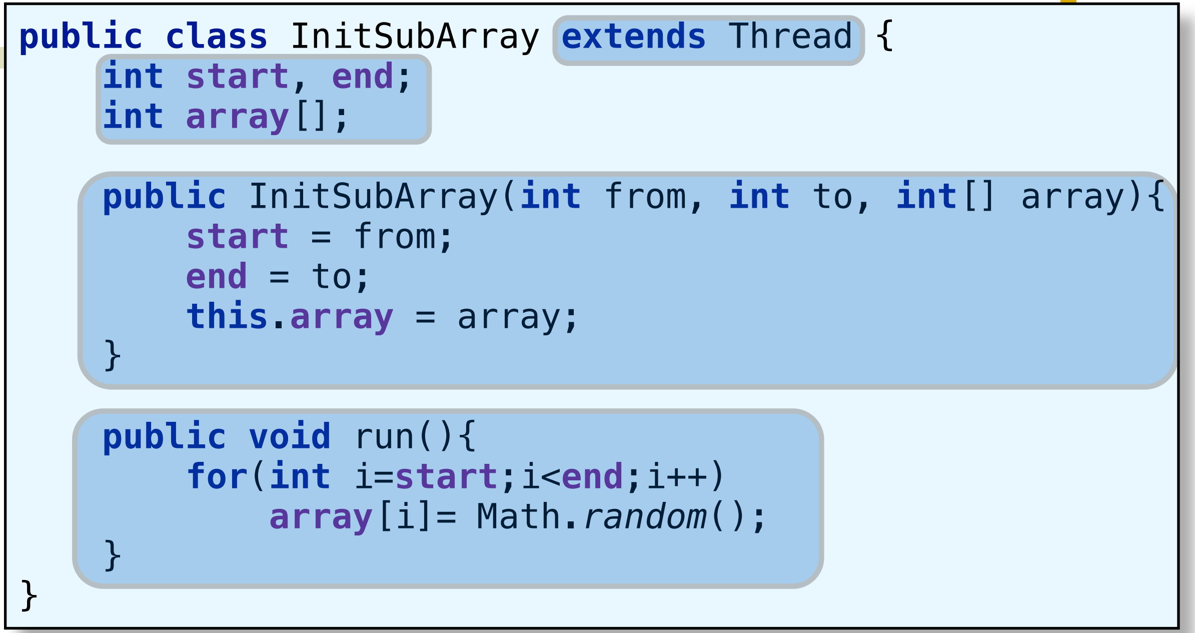
j

23
44
3454
323
23343
232
99076
2132

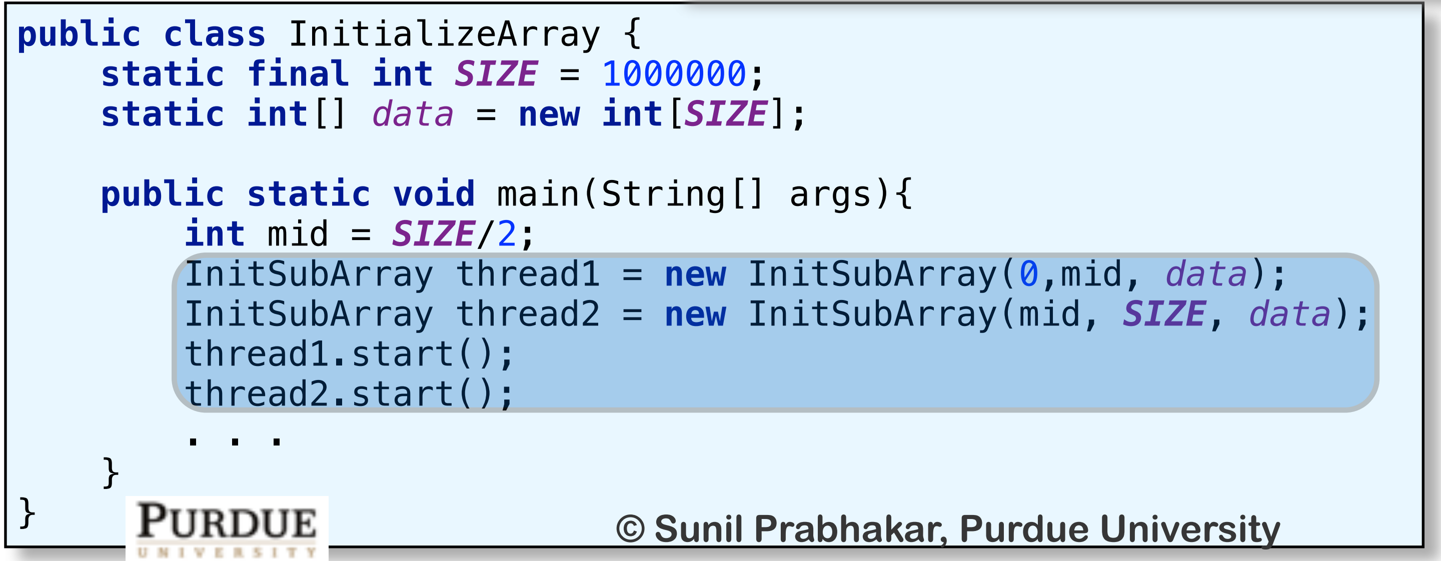
```
mid=SIZE/2;
i=0;
i<mid?
rand[i]=Math.random();
i++
j=mid;
j<SIZE?
rand[j]=Math.random();
j++
next statement
```

Need to wait for both threads before continuing.

Array: Concurrent Version



```java
public class InitSubArray extends Thread {
    int start, end;
    int array[];

    public InitSubArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }

    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
```



```java
public class InitializeArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new InitSubArray(0,mid, data);
        InitSubArray thread2 = new InitSubArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
        . . .
    }
}
```

Array: Concurrent Version

A new thread of execution with its own control flow is created:

```
public class InitSubArray extends Thread {
    int start, end;
    int array[];

    public InitSubArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }

    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
```

The new (blue) thread runs with the context of thread1

```
public class InitializeArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new InitSubArray(0,mid, data);
        InitSubArray thread2 = new InitSubArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
        . . .
    }
}
```

Array: Concurrent Version

A third new thread of execution with its own control flow is created:

```java
public class InitSubArray extends Thread {
    int start, end;
    int array[];

    public InitSubArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }

    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
```

```java
public class InitializeArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new InitSubArray(0,mid, data);
        InitSubArray thread2 = new InitSubArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
        . . .
    }
}
```

The new (magenta) thread runs with the context of thread2

Rejoining Threads

- In the last example, it is necessary to wait for both threads to finish before moving on.
- This is achieved by calling the `join()` method
  - the thread that calls join is suspended until the thread on which it is called terminates.
  - this method can throw the (checked) InterruptedException so we should catch this exception

Array: Concurrent Version 2

```
public class InitializeArray {
    . . .
    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new
                 InitSubArray(0,mid, data);
        InitSubArray thread2 = new
                 InitSubArray(mid, SIZE, data
        thread1.start();
        thread2.start();
        try{
            thread1.join();
            thread2.join();
        } catch (InterruptedException e){
            System.out.println("Error in thread");
        }
        . . .
    }
}
```

```
public class InitSubArray extends
Thread {
    int start, end;
    int array[];
    public InitArray(int from, int
to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }
    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
```

The `join()` Method

- A call to the `join()` method blocks (i.e., does not return) until the thread on which it is called terminates
  - returns from its `run()` method, or
  - propagates an exception from `run()`
- While being blocked, the calling thread may get interrupted which is why the join method throws the exception.
- Do not use the `stop()` method to stop a thread -- deprecated.

Speedup

- Two key reasons for concurrency:
  - liveness (e.g., game keeps redrawing screen)
  - speedup (with more cores, programs run faster)
- Can be measured using the System class methods:
  - `public static long currentTimeMillis()`
    - time elapsed since January 1st, 1970 12:00am, in milliseconds
  - `public static long nanoTime()`
    - current value of computer’s timer in ns.

Creating Sub-Tasks

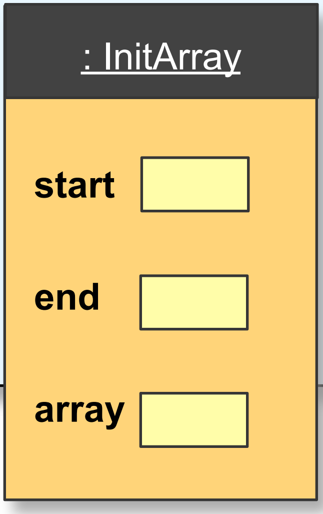
- To achieve concurrent processing, we need to divide a task into multiple pieces that can be assigned to concurrent threads.
- Two main approaches
  - Task decomposition
    - divide the type of work being performed
    - e.g., game example
  - Domain decomposition
    - divide the data on which the same task is performed
    - e.g. matrix initialization

Domain Decomposition

- To achieve domain decomposition, recall that each thread is run in the context of an object of a descendent of the Thread class
  - Each thread object has its own values for its data members
  - We use these data members to divide the domain for each thread
  - Each thread object is usually given its own range of the task to complete
    - e.g., the range of array indexes that it is responsible for initializing

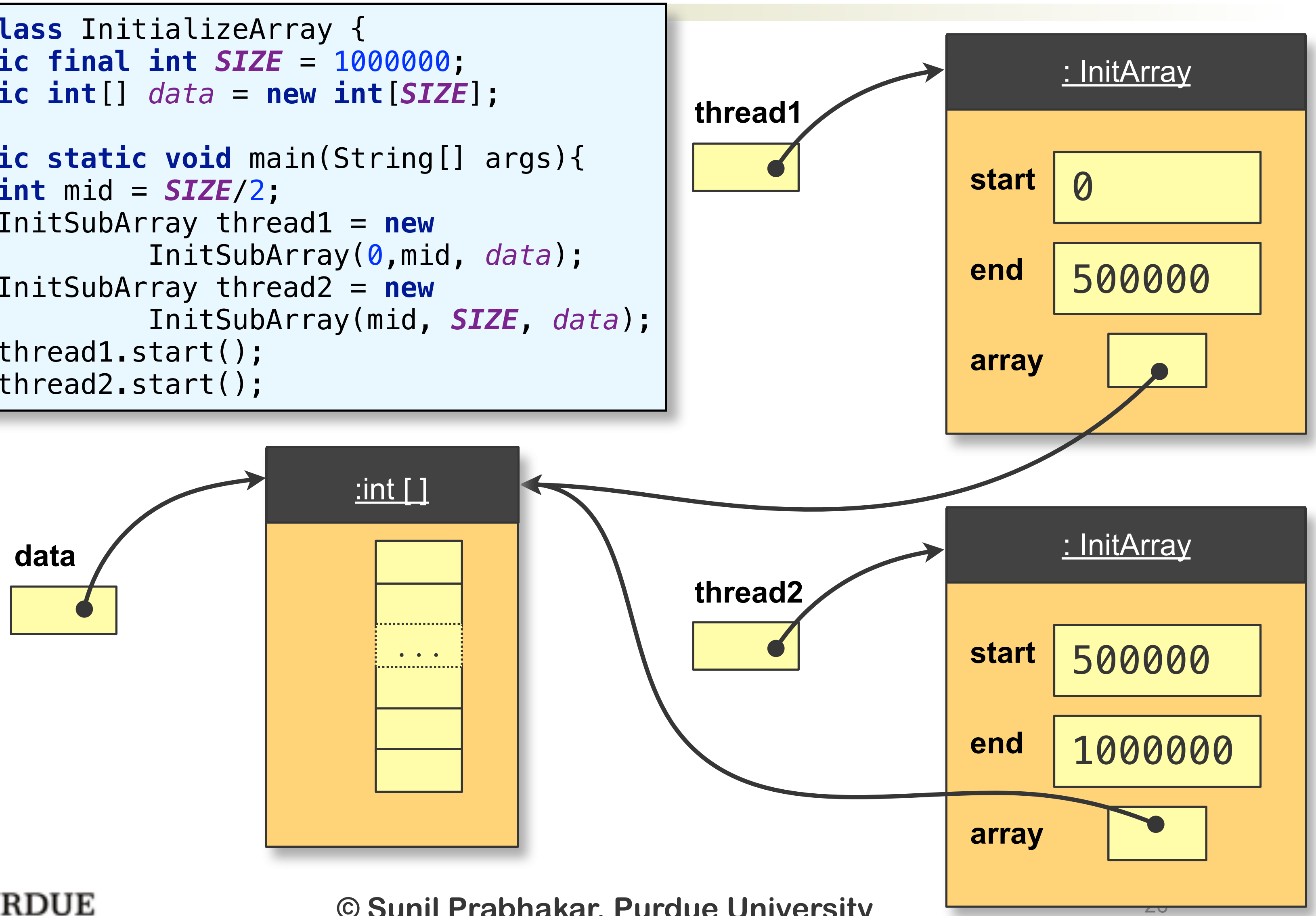
Array: Domain Decomposition

```java
public class InitSubArray extends Thread {
    int start, end;
    int array[];

    public InitSubArray(int from, int to, int[] array){
        start = from;
        end = to;
        this.array = array;
    }

    public void run(){
        for(int i=start;i<end;i++)
            array[i]= Math.random();
    }
}
```



Array: Domain Decomposition

```
public class InitializeArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new
                 InitSubArray(0,mid, data);
        InitSubArray thread2 = new
                 InitSubArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
```

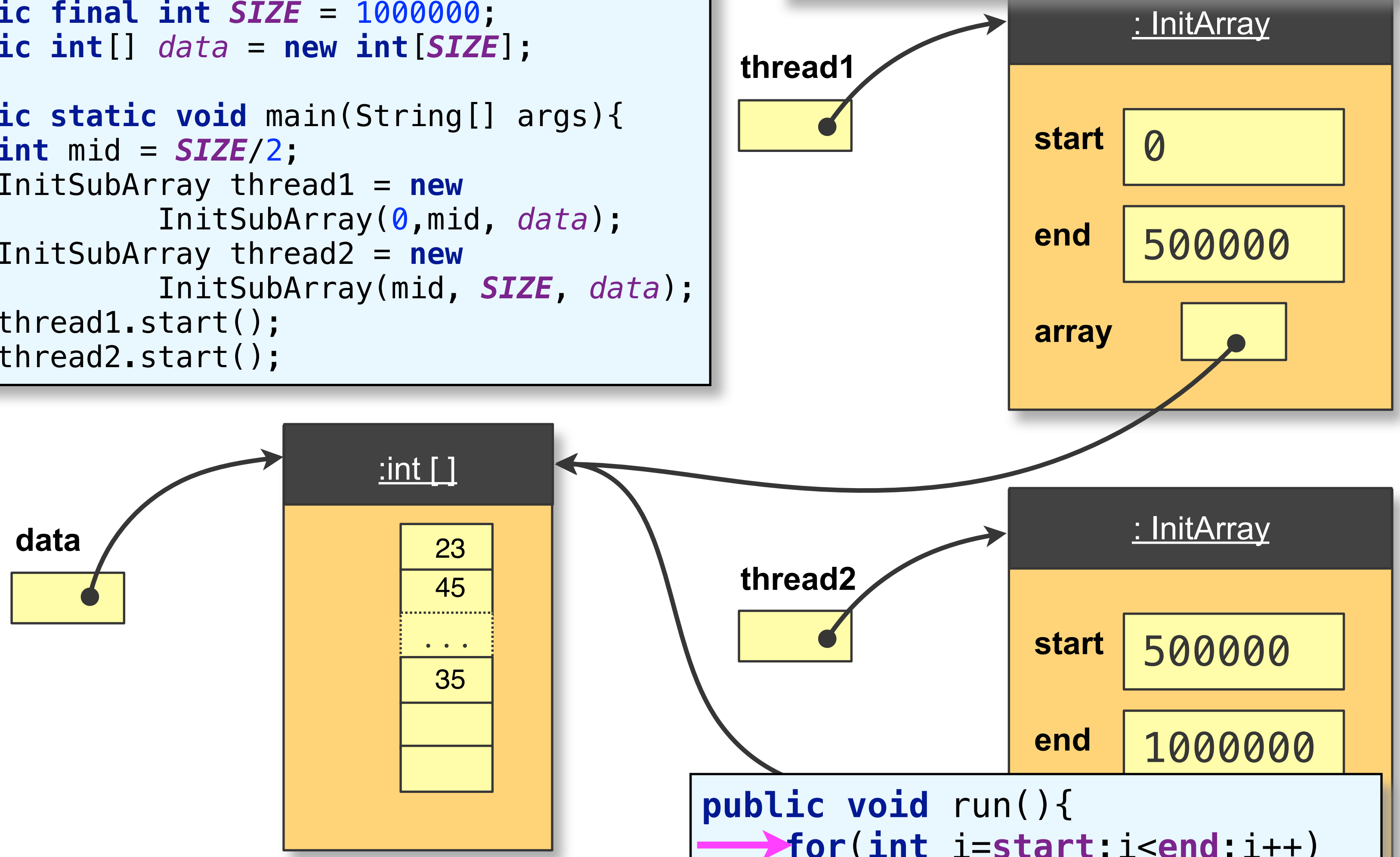


Array: Domain De

```java
public void run(){
    for(int i=start;i<end;i++)
        array[i]= Math.random();
}
```

i 1

```java
public class InitializeArray {
    static final int SIZE = 1000000;
    static int[] data = new int[SIZE];

    public static void main(String[] args){
        int mid = SIZE/2;
        InitSubArray thread1 = new
                 InitSubArray(0,mid, data);
        InitSubArray thread2 = new
                 InitSubArray(mid, SIZE, data);
        thread1.start();
        thread2.start();
```



```java
public void run(){
    for(int i=start;i<end;i++)
        array[i]= Math.random();
}
```

i 500001

Array multiplication

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \text{X} \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

$$a_{11} = \Sigma_{k=1}\, b_{1k}{}^{*}\, c_{k1} = b_{11}{}^{*}c_{11} + b_{12}\, {}^{*}\, c_{21} + b_{13}{}^{*}c_{31} + b_{14}{}^{*}c_{41}$$

4

$$a_{ij} = \Sigma_{k=1}\, b_{ik}{}^{*}c_{kj} = b_{i1}{}^{*}c_{1j} + b_{i2}\, {}^{*}\, c_{2j} + b_{i3}{}^{*}c_{3j} + b_{i4}{}^{*}c_{4j}$$

Task sub-division

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \text{X} \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

Array Multiplication Example

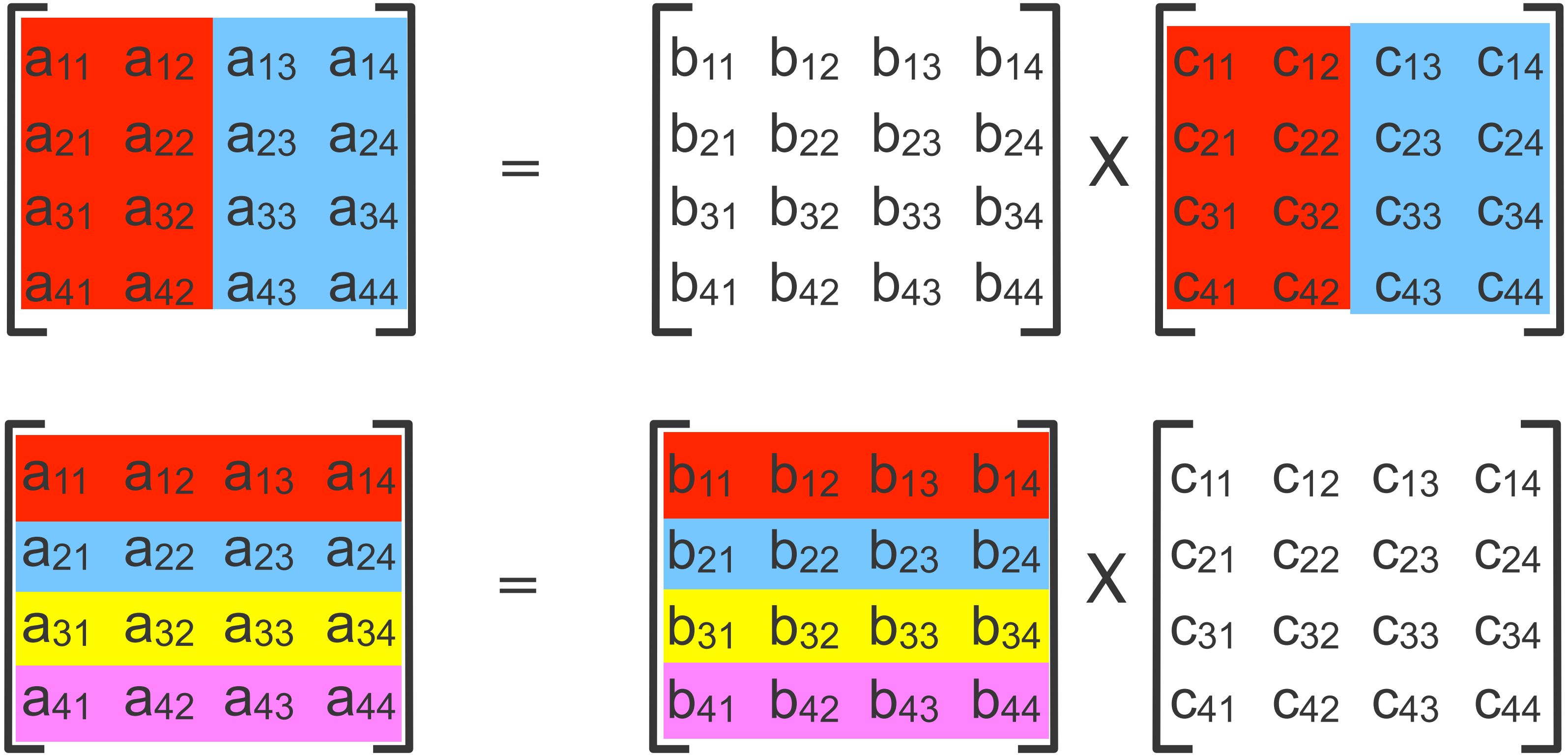
```
public class SubArrayMultiplier extends Thread {
    int start, end;
    int[][] a, b, c;

    public SubArrayMultiplier(
           int from, int to, int[][] a, int[][] b, int[][] c) {
        start = from;
        end = to;
        this.a = a;
        this.b = b;
        this.c = c;
    }

    public void run() {
        for (int i = start; i < end; i++) {
            for (int j = 0; j < a[0].length; j++) {
                a[i][j] = 0;
                for (int k = 0; k < b.length; k++)
                    a[i][j] += b[i][k] * c[k][j];
            }
        }
    }
  //main() on next slide
}
```

Array Multiplication Example

```java
public static void main(String[] args) {
    final int M = 4, N = 4, K = 4;
    int[][] a = new int[M][N];
    int[][] b = new int[M][K];
    int[][] c = new int[K][N];

    initialize(a, M, N);
    initialize(b, M, K);
    initialize(c, K, N);

    SubArrayMultiplier mult1 =
          new SubArrayMultiplier(0, M / 2, a, b, c);
    SubArrayMultiplier mult2 =
          new SubArrayMultiplier(M / 2, M, a, b, c);

    mult1.start();
    mult2.start();
    try {
        mult1.join();
        mult2.join();
    } catch (InterruptedException e) {
        System.out.println("Unexpected Interrupt");
    }
}
```

```java
public static void initialize(int[][]
array, int rows, int cols) {

    for (int i = 0; i < rows; i++)
        for (int j = 0; j < cols; j++)
            array[i][j] = (int)
              (Math.random() * 1000);
}
```

Task sub-division



$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \text{X} \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} = \begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix} \text{X} \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}$$

Multiplication Multiple Threads

```java
public class ArrayMultiplier {
    public static void main(String[] args){
        final int NUM_THREADS=5;
        SubArrayMultiplier[] threads = new SubArrayMultiplier[NUM_THREADS];
        . . .
        int subsetSize = (int) Math.ceil(a.length /(float)NUM_THREADS);
        int startRow = 0;
        for(int i=0;i<NUM_THREADS;i++){
            threads[i]=new SubArrayMultiplier(startRow,
                    Math.min(startRow+subsetSize,a.length), a, b, c);
            threads[i].start();
            startRow += subsetSize;
        }
        try{
            for(int i=0;i<NUM_THREADS;i++)
                threads[i].join();
        } catch (InterruptedException e) {
            System.out.println("Unexpected Interrupt");
        }
    }
}
```

Processes

- Modern operating systems support multi-tasking
  - painting the screen, listening to the keyboard, printing, running several programs, ...
- Even with a single core multiple tasks are concurrently running
- Achieved by sharing the processor among multiple processes
  - the CPU runs a little of each process in turn
  - this is called *process scheduling*

Threads

- A process often corresponds to a program
  - Browser, editor, ...
- Modern processes often have multiple threads of execution.
- Roughly,
  - different processes are largely independent of each other;
  - different threads of the same process often share the same memory space.

Thread Scheduling

- Within a single thread, instructions are processed one at a time.
- However, different threads can run at different times/rates.
- When a thread runs is determined by many factors:
  - Java implementation
  - Operating system
  - Instructions being executed
  - ....

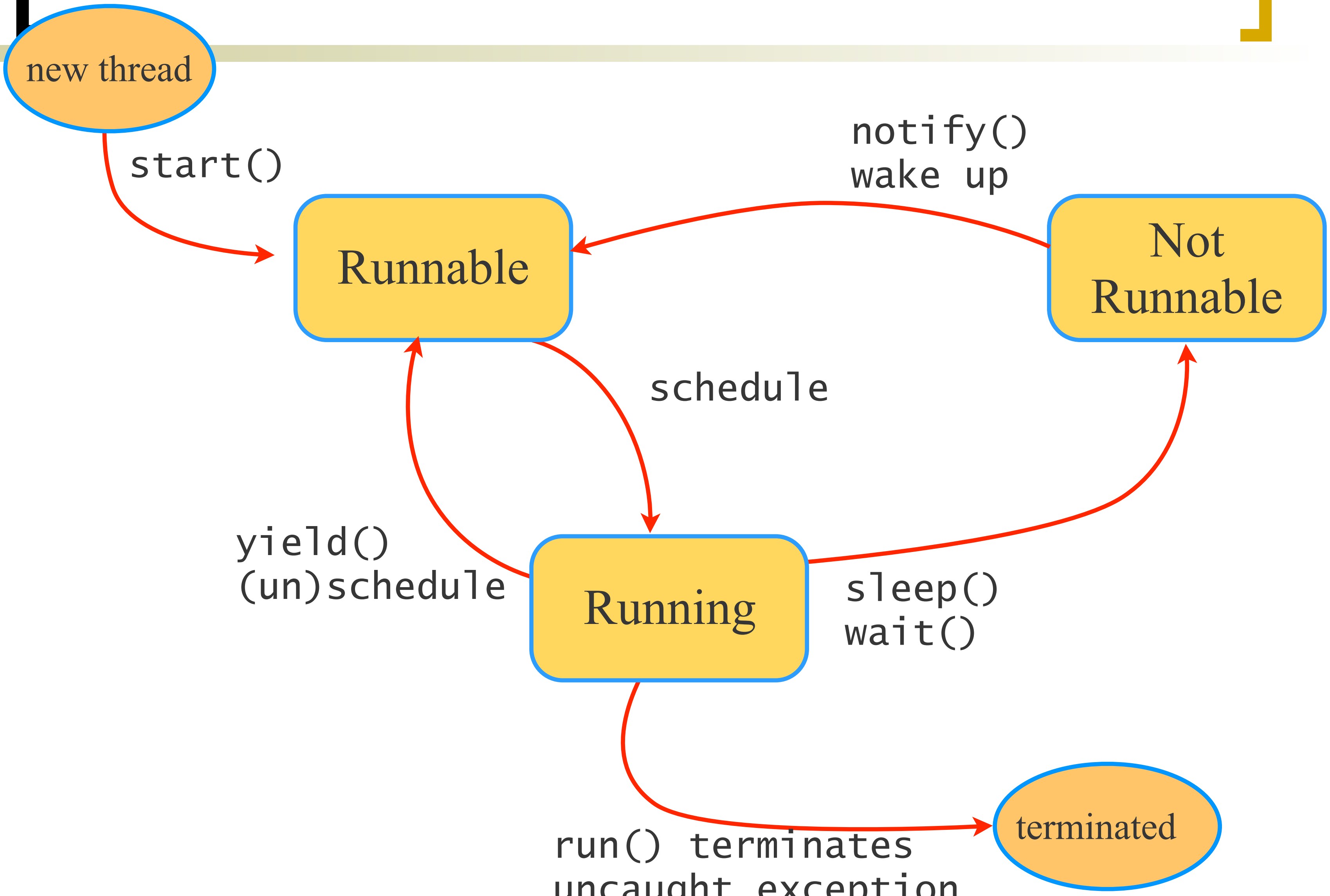
Non-Determinism

- An important property of threads is that it is not possible to know exactly when a given thread will be scheduled
  - cannot assume anything about relative ordering between threads
- Order of concurrent threads (and consequently the result of the output) may change from run to run!
- Programmer must anticipate all possible orderings and protect against possible errors.

Controlling Thread Scheduling

- As a programmer we have several mechanisms available:
  - sleep()
    - thread cannot be scheduled for some time
  - yield()
    - voluntarily give up your turn for the CPU
  - wait()
    - wait for some condition to be true
  - Priority
    - Each thread has a priority. Can set priorities for threads we create (with some limitations).

Thread States



Thread Scheduling

- At any given time there may be a number of threads that are runnable
  - each has a priority
  - usually the same as the creating thread’s priority
- Periodically, the OS schedules one of the threads with the highest priority for some time.

Synchronization example

- Say we want to try to control the relative ordering of two threads:
  - thread1 prints: “Left, Left, Left” then “Left”
  - thread2 prints: “Right”
- Suppose we want to ensure the following output:
  - “Left, Left, Left, Right, Left” multiple times.
  - How can we ensure that the timing of the threads ensures this output?
  - I.e., how to avoid non-determinism?

Synchronization Example

- Say we want to try to control the relative ordering of two threads:
  - thread1 prints: “Left, Left, Left” then “Left”
  - thread2 prints: “Right”
- Suppose we want to ensure the following output:
  - “Left, Left, Left, Right, Left” multiple times.
  - How can we ensure that the timing of the threads ensures this output?
  - I.e., how to avoid non-determinism?

Attempt 0: No Synchronization

```java
public class LeftThread extends Thread {
    int reps;

    public LeftThread(int reps) {
        this.reps = reps;
    }

    public void run() {
        for (int i = 0; i < reps; i++) {
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.println("Left ");
        }
    }
}
```

```java
public class RightThread extends Thread {
    int reps;

    public RightThread(int reps) {
        this.reps = reps;
    }


    public void run() {
       for (int i = 0; i < reps; i++) {
            System.out.print("Right ");
       }
    }

}
```

Attempt 0: Driver Program

```
public class March {
    public static void main(String[] args) {
       int reps = Integer.parseInt(JOptionPane.
            showInputDialog(null, "Enter number of repetitions"));

        LeftThread left = new LeftThread(reps);

        RightThread right = new RightThread(reps);

        left.start();
        right.start();
    }
}
```

Attempt 1: using sleep()

```
public class LeftThread extends Thread {
      . . .

   public void run() {
       for (int i = 0; i < reps; i++) {
           System.out.print("Left ");
           System.out.print(“Left ");
           System.out.print("Left ");
           try {
                Thread.sleep(10);
           } catch (InterruptedException e) {
                e.printStackTrace();
           }
           System.out.println("Left ");
        }
    }
}
```

```
public class RightThread extends Thread {
    . . .
    public void run() {
        try {
            Thread.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (int i = 0; i < reps; i++) {
           System.out.print(“Right ");
           try {
               Thread.sleep(10);
           } catch (InterruptedException e) {
               e.printStackTrace();
           }
        }
    }
}
```

Problem With `sleep()`

- Doesn’t work
  - There is no guarantee that with the sleeping we will get synchronized each time
  - With enough chances, will get out of sync
- There may be unnecessary waiting
- Hard to tune the sleep times

Attempt 2: Using yield()

```java
public class LeftThread extends Thread {
      . . .

   public void run() {
       for (int i = 0; i < reps; i++) {
            System.out.print("Left ");
            System.out.print("Left ");
            System.out.print("Left ");
            try {
                Thread.yield();
           } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("Left ");
        }
    }
}
```

```java
public class RightThread extends Thread {
    . . .
    public void run() {
        try {
            Thread.yield();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        for (int i = 0; i < reps; i++) {
            System.out.print("Right ");
            try {
                Thread.yield();
           } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Problem With `yield()`

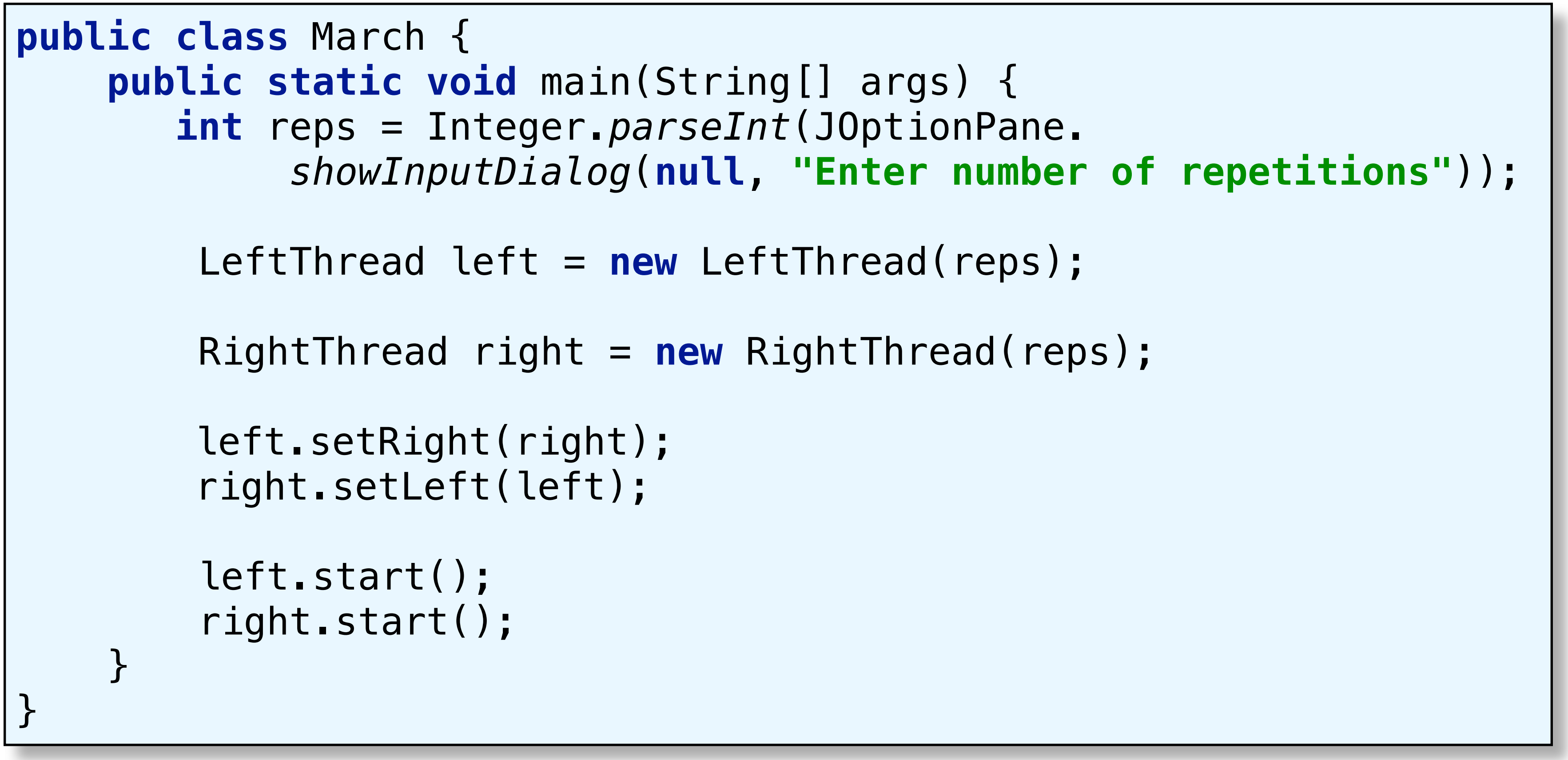
- Assumes that the `yield()` calls will give control to the other thread
  - many threads may be running on the machine
  - can cause unexpected switches between our 2 threads
- With multiple cores, each will be running on a separate core -- yielding does not provide anything!
- Also doesn’t work

Attempt 3: using polling

```java
public class LeftThread extends Thread {
    int reps;
    private RightThread right;
    private volatile boolean done = false;

    public void setRight(RightThread right) {
      this.right = right;
    }

    public void run() {
      for (int i = 0; i < reps; i++) {
        System.out.print("Left ");
        System.out.print("Left ");
        System.out.print("Left ");
        done = true;
        Thread.yield();
        while (!right.isDone()) ;
        right.setDone(false);
        System.out.println("Left ");
        Thread.yield();
      }
    }

   public boolean isDone() { return done; }
   public void setDone(boolean value) {
      done = value;
   }
}
```

```java
public class RightThread extends Thread {
     int reps;

     private LeftThread left;
     private volatile boolean done = false;

     public void setLeft(LeftThread left) {
         this.left = left;
     }

     public void run() {
         for (int i = 0; i < reps; i++) {
             while (!left.isDone()) ;
             left.setDone(false);
             System.out.print("Right ");
             done = true;
             Thread.yield();
         }
     }

     public boolean isDone() {return done; }
     public void setDone(boolean value) {
         done = value;
     }
}
```

Polling Driver Class



```java
public class March {
    public static void main(String[] args) {
       int reps = Integer.parseInt(JOptionPane.
            showInputDialog(null, "Enter number of repetitions"));

        LeftThread left = new LeftThread(reps);

        RightThread right = new RightThread(reps);

        left.setRight(right);
        right.setLeft(left);

        left.start();
        right.start();
    }
}
```

Polling Solution

- This works
  - always produces correct output.
- However,
  - No real concurrency!
  - Only one thread running at a time.
  - Busy waiting (wastes resources)
- Technicality:
  - should ensure that done variables are visible to the other thread immediately: use the `volatile` modifier.

Correct Solution

- We will use wait() and notify() to coordinate the two threads
  - The Right thread waits for a notification before printing its “Right”. Once it is done, it notifies the Left thread.
  - The Left thread prints three “Left”s then notifies the Right thread, and waits for the Right thread.
- This works correctly, as long as every call to wait() is followed by a call to notify() from the other thread
  - Right thread should be started first

Correct Solution

```java
public class March {
    public static void main(String[] args) {
        Object lock = new Object();

        int reps = Integer.parseInt(JOptionPane.showInputDialog(
                  null, "Enter number of repetitions"));

        LeftThread left = new LeftThread(reps, lock);
        RightThread right = new RightThread(reps, lock);

        right.start();
        left.start();

    }
}
```

Correct Solution

```java
public class LeftThread extends Thread {
  int reps;
  Object lockObject;

  public LeftThread(int reps, Object o) {
    this.reps = reps;
    lockObject = o;
  }

  public void run() {
    for (int i = 0; i < reps; i++) {
      System.out.print("Left ");
      System.out.print("Left ");
      System.out.print("Left ");
      synchronized (lockObject) {
        lockObject.notify();
        try {
           lockObject.wait();
        } catch (InterruptedException e) {
        }
      }
      System.out.println("Left ");
    }
  }
}
```

```java
public class RightThread extends Thread {
  int reps;
  Object lockObject;

  public RightThread(int reps, Object o) {
    this.reps = reps;
    lockObject = o;
  }

  public void run() {
    for (int i = 0; i < reps; i++) {
      synchronized (lockObject) {
        try {
          lockObject.wait();
          System.out.print("Right ");
          lockObject.notify();
        } catch (InterruptedException e) {
        }
      }
  }
}
```

Concurrency Is Tricky

- Writing concurrent programs that work as expected can be tricky
- Need to deal with
  - non-determinism of scheduling
  - ensuring access to shared data is correct (see slides on Synchronization)
- Achieving speed up is not always easy

The Runnable Interface

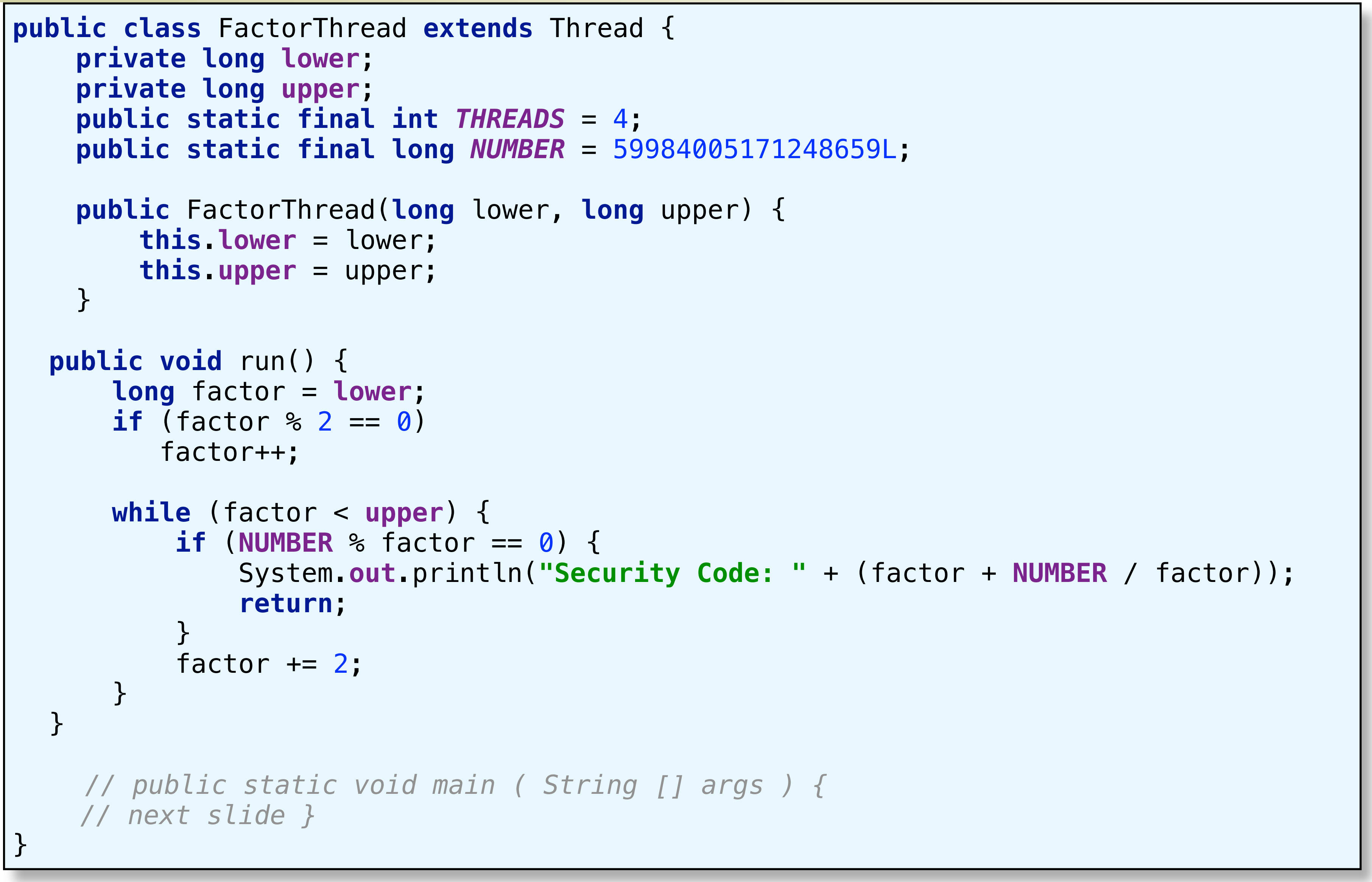
- What if we want to use a class to create threads, but it extends some other class?
- We can use the Runnable interface.
  - First, declare that the class implements the Runnable interface
    - This requires a run() method to be created
  - To start a thread using an object of this class
    - Create a Thread object with this object as an argument to the Thread constructor
    - Call start() of the thread object.

See CountUpRunnable.java

Examples

- Factorization of a large integer
  - need to find the two prime factors of a large integer value
  - divide the task by domain decomposition
- Array summation
  - compute the sum of the sine of all values of a large array
  - divide by domain decomposition
  - need to synchronize after sub-tasks are done

Factorization



```
public class FactorThread extends Thread {
    private long lower;
    private long upper;
    public static final int THREADS = 4;
    public static final long NUMBER = 59984005171248659L;

    public FactorThread(long lower, long upper) {
        this.lower = lower;
        this.upper = upper;
    }

  public void run() {
      long factor = lower;
      if (factor % 2 == 0)
         factor++;

      while (factor < upper) {
          if (NUMBER % factor == 0) {
              System.out.println("Security Code: " + (factor + NUMBER / factor));
              return;
          }
          factor += 2;
      }
  }

    // public static void main ( String [] args ) {
    // next slide }
}
```

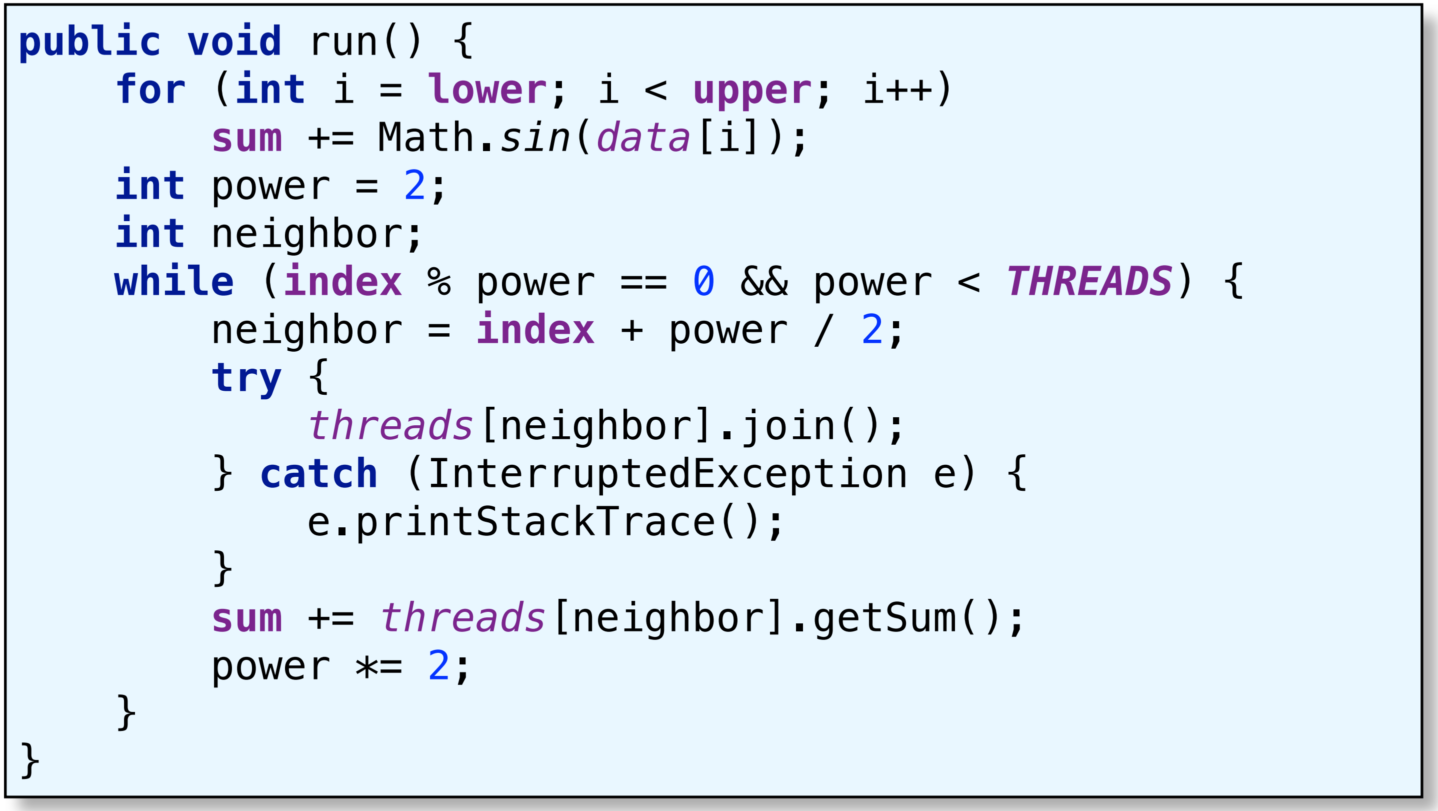
Factorization (Contd.)

```java
public static void main(String[] args) {
    FactorThread[] threads = new FactorThread[THREADS];
    long root = (long) Math.sqrt(NUMBER);
    long start = 3;
    long factorsTestedPerThread
         = (long) Math.ceil((root - 2) / (float) THREADS);

    for (int i = 0; i < THREADS; i++) {
        threads[i] = new FactorThread(start,
                 Math.min(start + factorsTestedPerThread, root + 1));
        threads[i].start();
        start += factorsTestedPerThread;
    }
    try {
        for (int i = 0; i < THREADS; i++)
            threads[i].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
```

MatrixSum

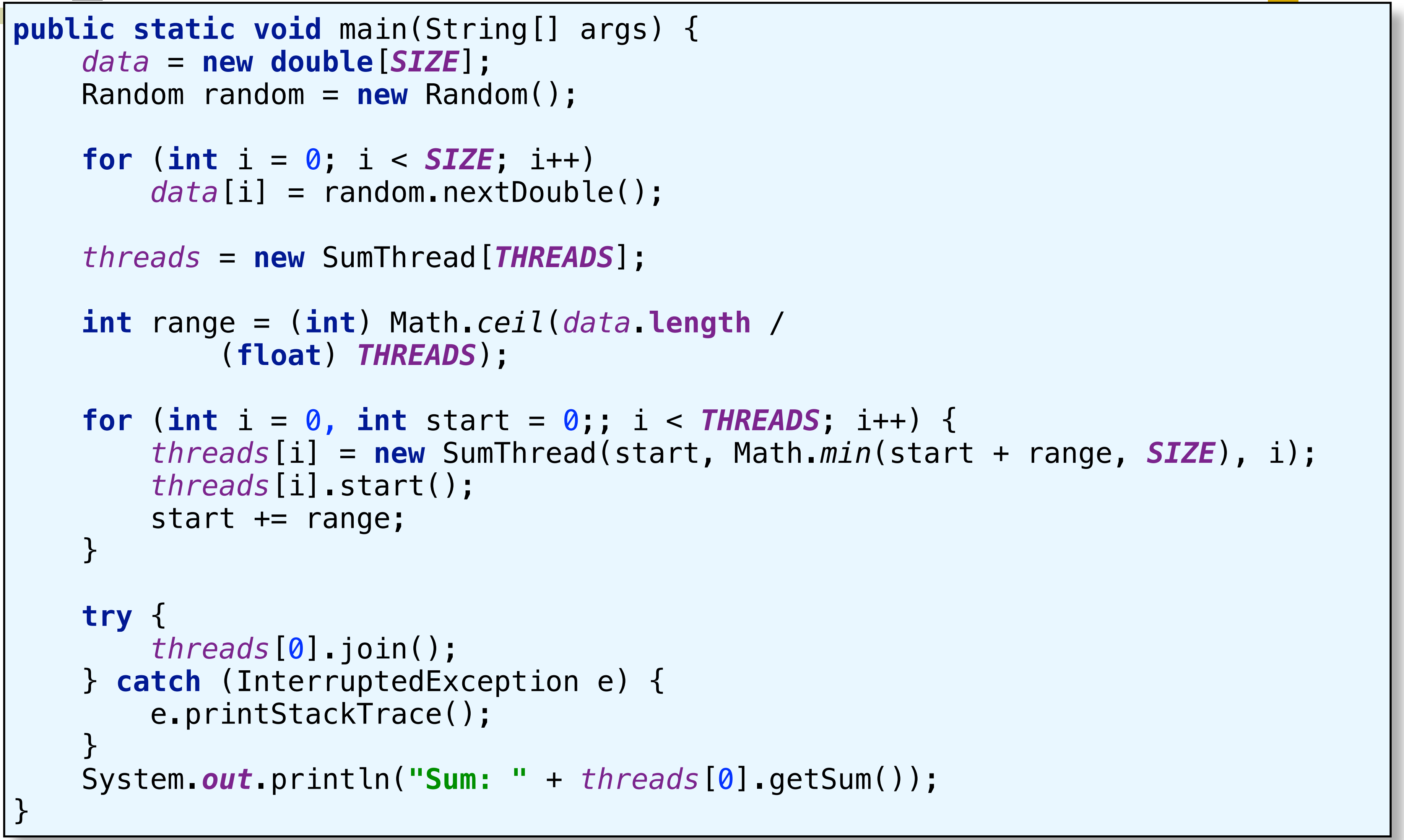
```java
import java.util.Random ;
public class SumThread extends Thread {
    private static double [] data ;
    private static SumThread [] threads ;
    private double sum = 0;
    private int lower, upper, index ;
    public static final int SIZE = 1000000;
    public static final int THREADS = 8;

    public SumThread (int lower, int upper, int index) {
        this.lower = lower;
        this.upper = upper;
        this.index = index;
    }
    public double getSum () { return sum ; }
    //public void run () { //next slide }
    // public static void main ( String [] args ) {
    // next slide }
}
```

MatrixSum (contd.)



```
public void run() {
    for (int i = lower; i < upper; i++)
        sum += Math.sin(data[i]);
    int power = 2;
    int neighbor;
    while (index % power == 0 && power < THREADS) {
        neighbor = index + power / 2;
        try {
            threads[neighbor].join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        sum += threads[neighbor].getSum();
        power *= 2;
    }
}
```

MatrixSum (contd.)



```java
public static void main(String[] args) {
    data = new double[SIZE];
    Random random = new Random();

    for (int i = 0; i < SIZE; i++)
        data[i] = random.nextDouble();

    threads = new SumThread[THREADS];

    int range = (int) Math.ceil(data.length /
                (float) THREADS);

    for (int i = 0, int start = 0;; i < THREADS; i++) {
        threads[i] = new SumThread(start, Math.min(start + range, SIZE), i);
        threads[i].start();
        start += range;
    }

    try {
        threads[0].join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("Sum: " + threads[0].getSum());
}
```