

Processes and Threads and how it is done in Java

Michelle Kuttel

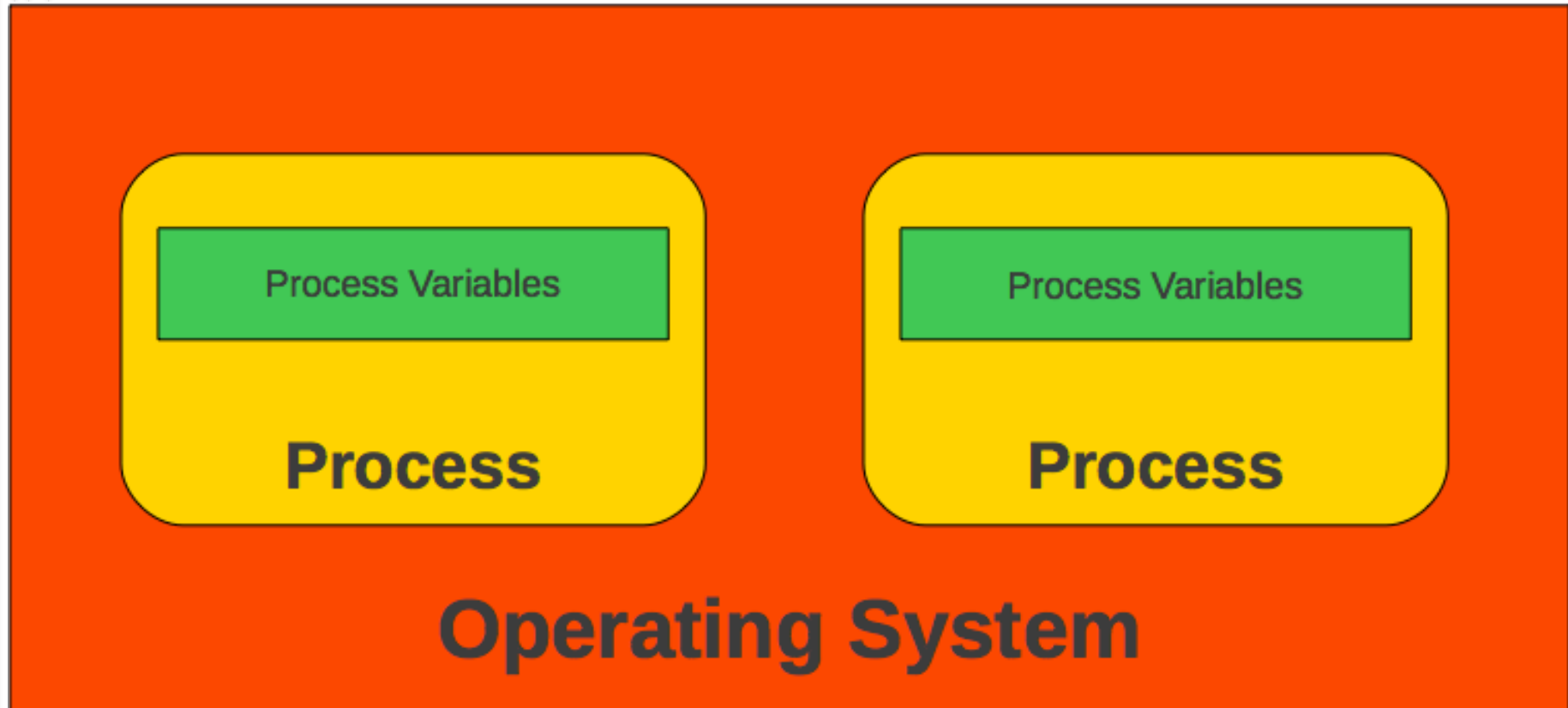
mkuttel@cs.uct.ac.za

Origin of term *process*

originates from operating systems.

- a unit of resource allocation both for CPU time and for memory.
- A process is represented by its **code**, **data** and the **state of the machine registers**.
- The data of the process is divided into global variables and local variables, organized as a stack.
- Generally, each process in an operating system has its own address space and some special action must be taken to allow different processes to access shared data.

Process memory model



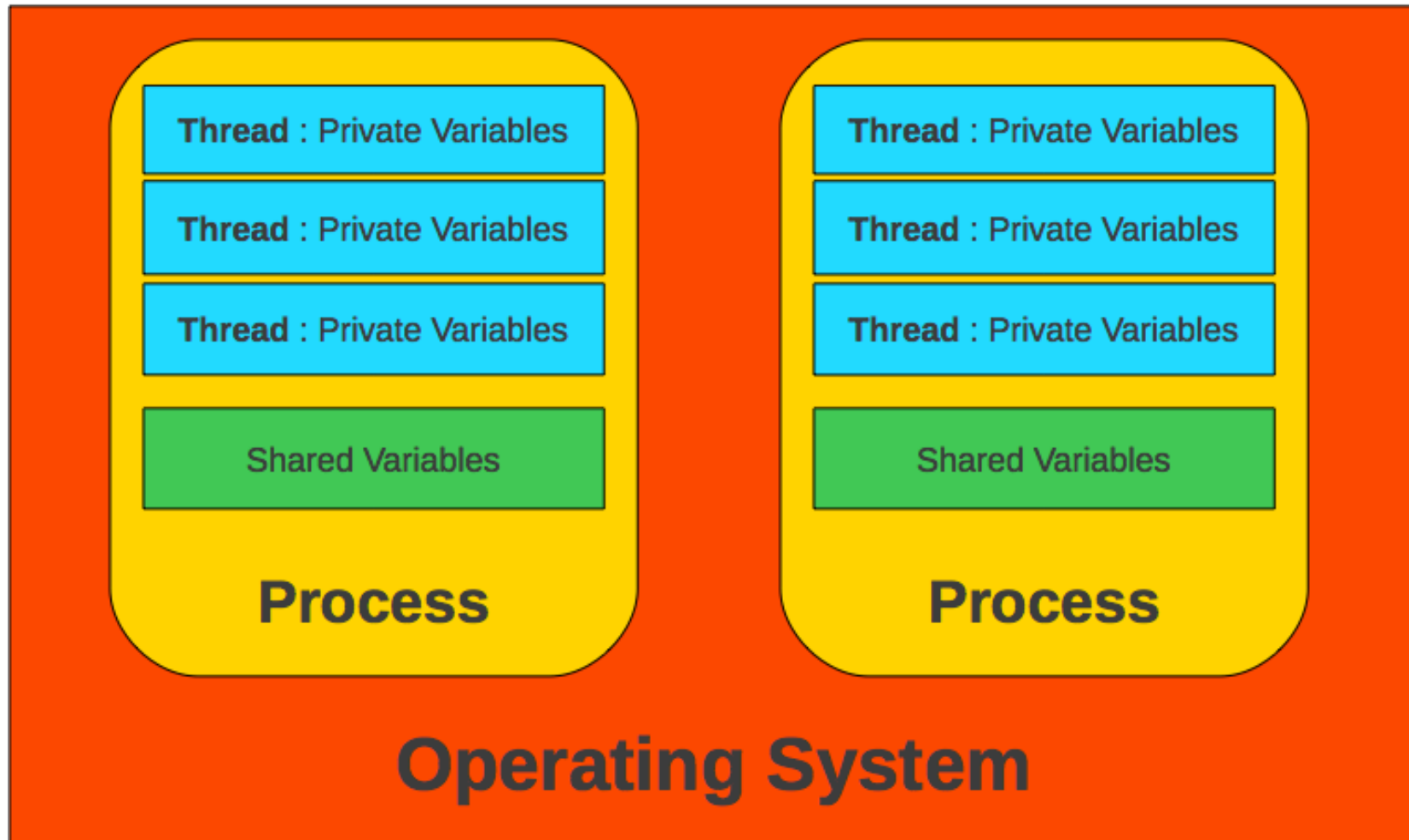
graphic: www.Intel-Software-Academic-Program.com

Origin of term *thread*

The traditional operating system process has a single thread of control – it has **no internal concurrency**.

- With the advent of shared memory multiprocessors, operating system designers catered for the requirement that a process might require internal concurrency by providing *lightweight processes or threads*.
- “*thread of control*”
- Modern operating systems permit an operating system process to have multiple threads of control.
- In order for a process to support multiple (lightweight) threads of control, it has multiple stacks, one for each thread.

Thread memory model



graphic: www.Intel-Software-Academic-Program.com

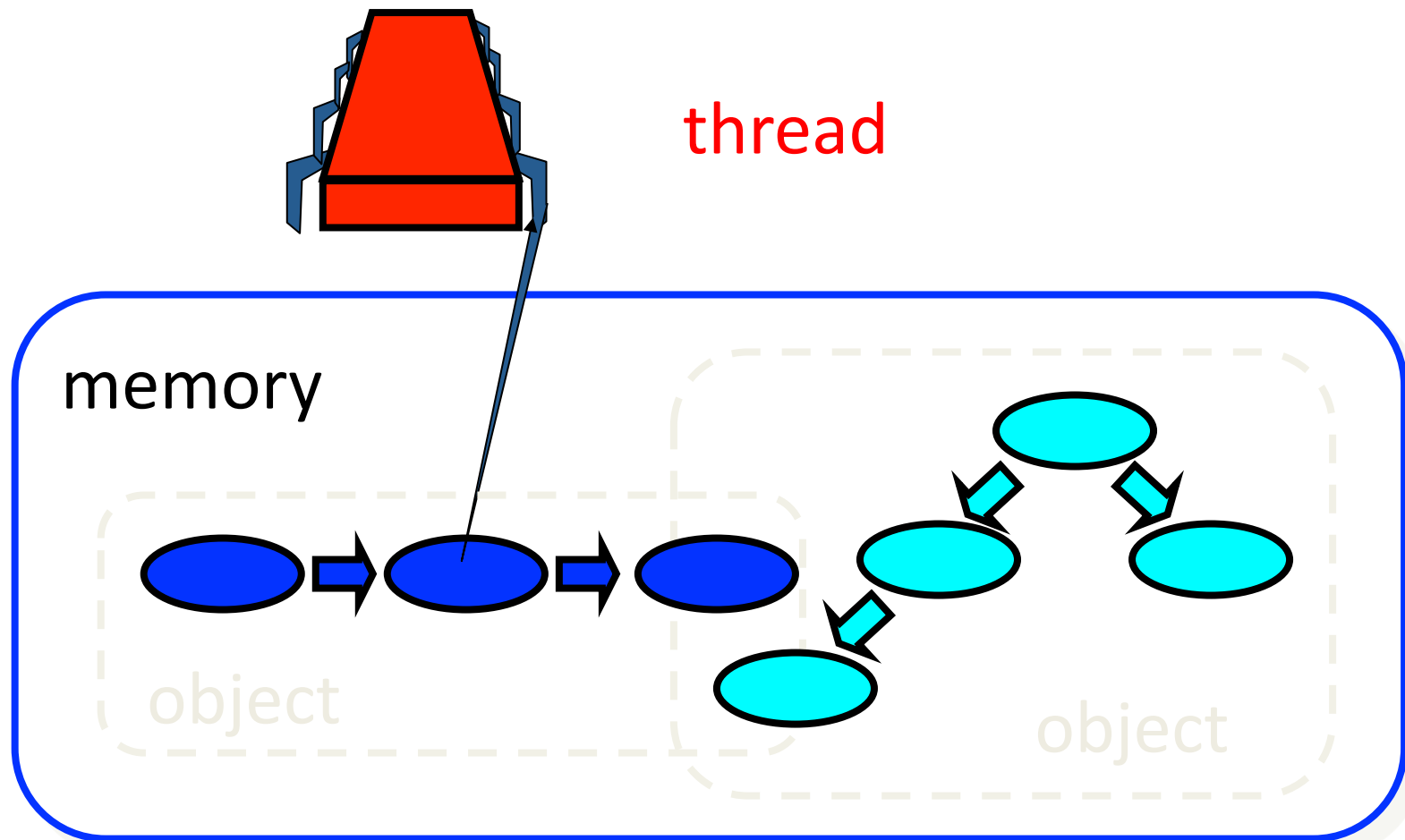
What is a parallel program?

A sequential program has a **single** thread of control

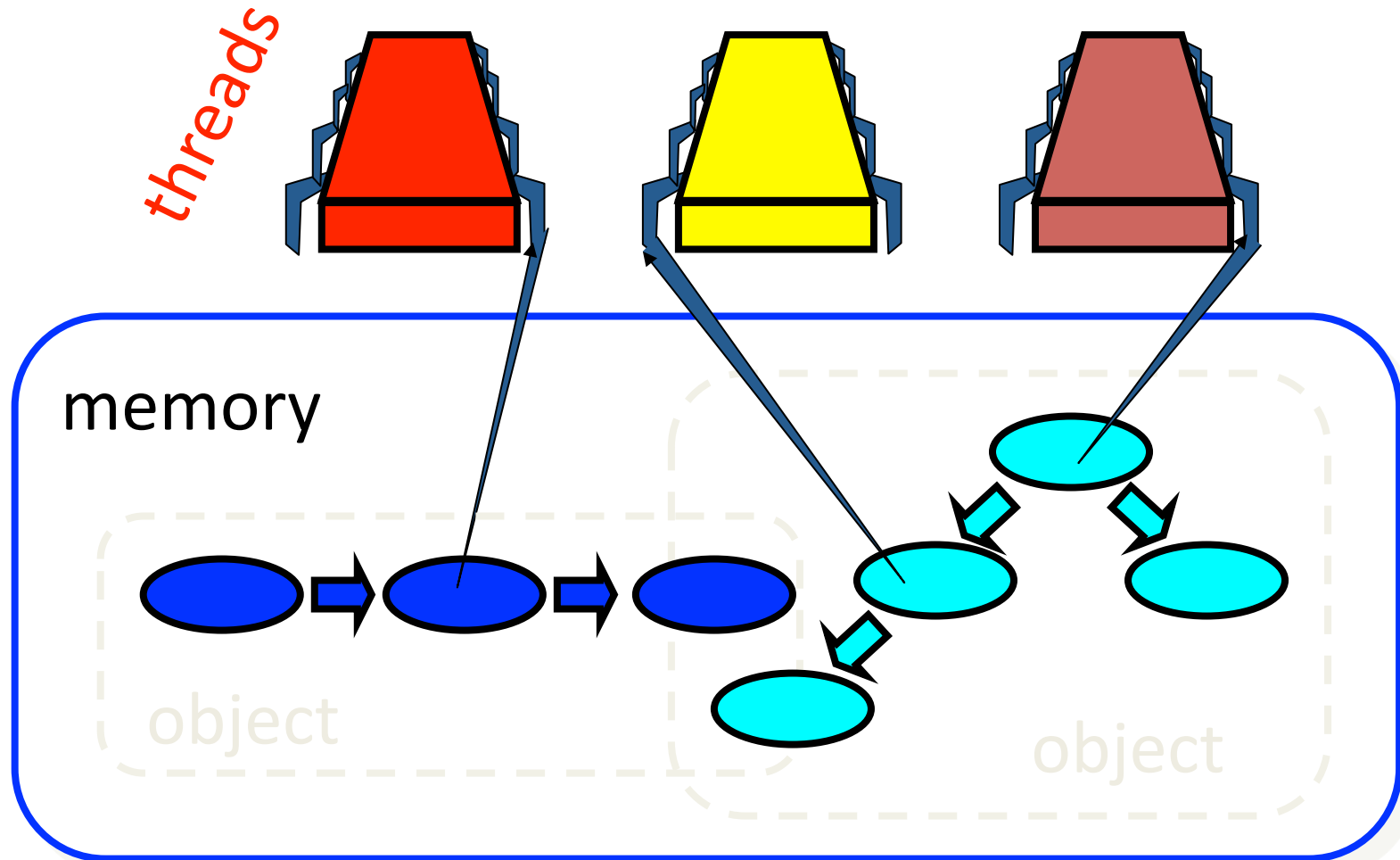
A parallel program has **multiple** threads of control

- can perform multiple computations in parallel
- can control multiple simultaneous external activities
- threads from the same process **share memory** (data and code).
- They can **communicate easily**, but it's dangerous if you don't protect your variables correctly.

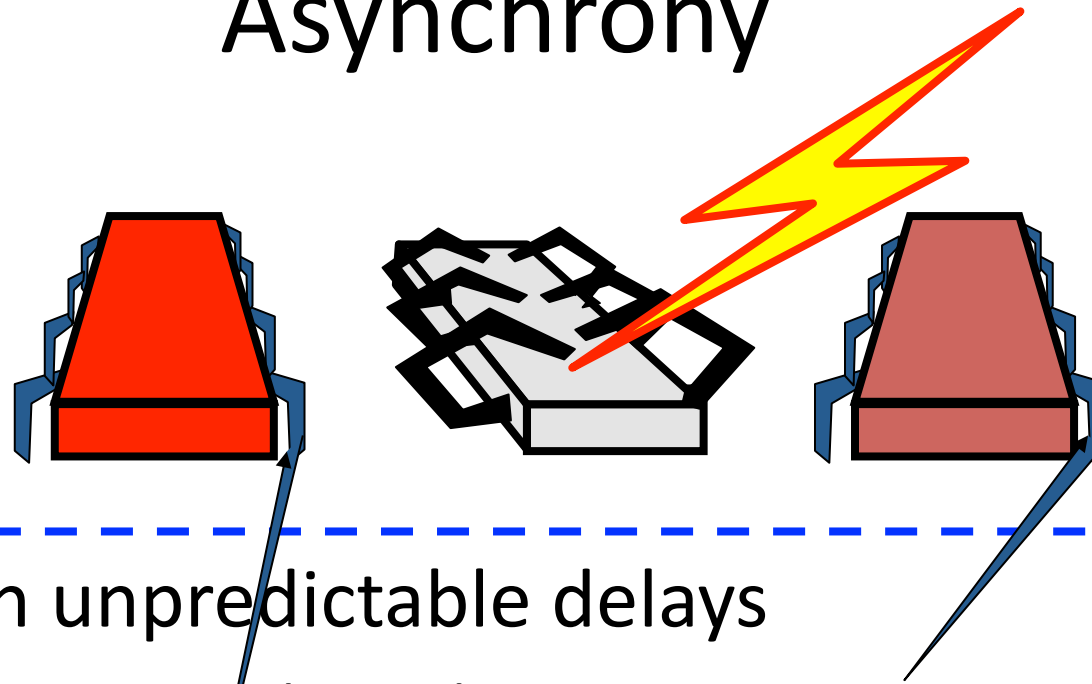
Sequential Computation



Concurrent Computation



Asynchrony



- Sudden unpredictable delays
 - Cache misses (*short*)
 - Page faults (*long*)
 - Scheduling quantum used up (*really long*)

Model Summary

- Multiple *threads*
 - Sometimes called *processes* (!!)
- Single shared *memory*
- *Objects* live in memory
- **Unpredictable asynchronous delays** (arbitrary speed)
- **Interleaving** (arbitrary order)

Concurrency Jargon

- Hardware
 - Processors
- Software
 - Threads, processes
- Sometimes OK to confuse them, sometimes not.

Parallel execution

Parallel execution **does not** require **multiple** processors:

Interleaving the instructions from multiple processes on a single processor can be used to simulate concurrency, giving the illusion of parallel execution.

*called **pseudo-concurrent execution** since instructions from different processes are not executed at the same time, but are interleaved.*

it is usual to have more active processes than processors. In this case, the available processes are switched between processors.

Java Threads -1

Java has had support for threads since its very beginning

at first, low-level approach with `interrupt`, `join` and `sleep` methods

also `notify` and `wait` methods

However, the threading constructs have undergone modification since the start.

In particular, several dangerous constructs have been *deprecated (dropped from the language)* e.g. Thread deprecated `stop` and `suspend` methods.

Java Threads - 2

Java 1.5 provided a higher level framework

- an extensive library of concurrency constructs:
`java.util.concurrent`
- threading simpler, easier and less-error prone way.

Java Threads – 3

Fork/Join framework

Java SE 7 introduced the Fork/Join framework

- designed to make divide-and-conquer algorithms easy to parallelize

Basic Threads in Java

We will first learn some basics built into Java via **`java.lang.Thread`**

- operations to create and initialize basic threads and control their execution

Then move on to a better library for parallel programming.

Java Threads

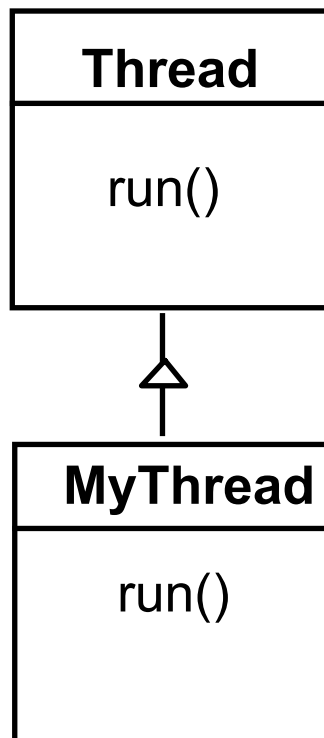
The Java Virtual Machine

- executes as a process under some operating system
- supports multiple threads.

Each Java thread has its **own local variables** organized as a stack and threads can access **shared variables**.

Basic Threads in Java

A Thread class manages a single sequential thread of control. Threads may be **created** and **deleted dynamically**.



Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```
class MyThread extends Thread {
    public void run() {
        //.....
    }
}
```

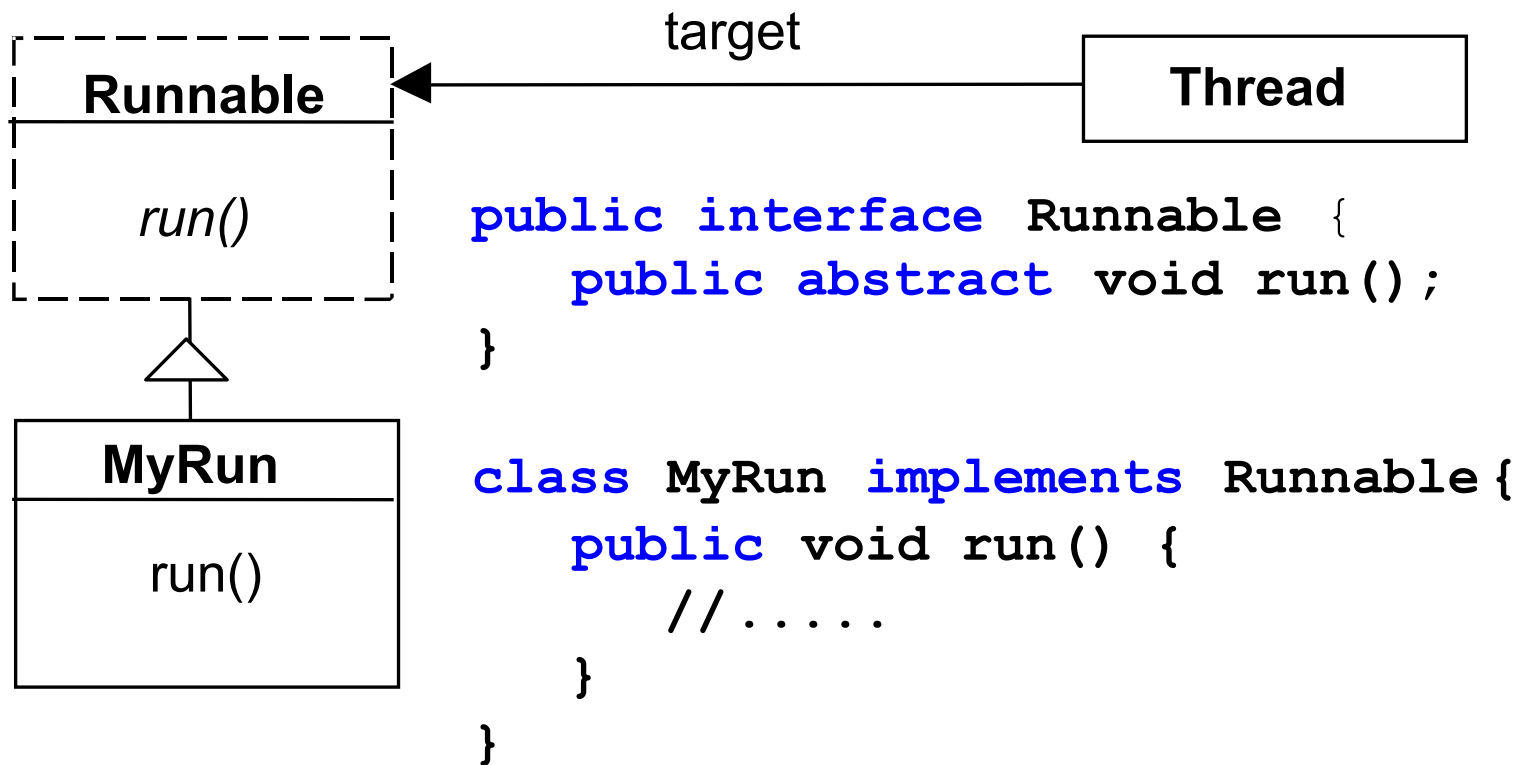
Creating a thread object:

```
Thread a = new MyThread();
```

Basic Threads in Java

Since Java does not permit multiple inheritance, it is sometimes more convenient to implement the `run()` method in a class not derived from `Thread` but from the interface `Runnable`

Basic Threads in Java



Creating a thread object:

```
Thread b = new Thread(new MyRun());
```

Basic Threads in Java

So, there are two ways to create a basic thread in Java:

- Implement the Runnable interface (`java.lang.Runnable`)
- Extend the Thread class (`java.lang.Thread`)

Java Threads

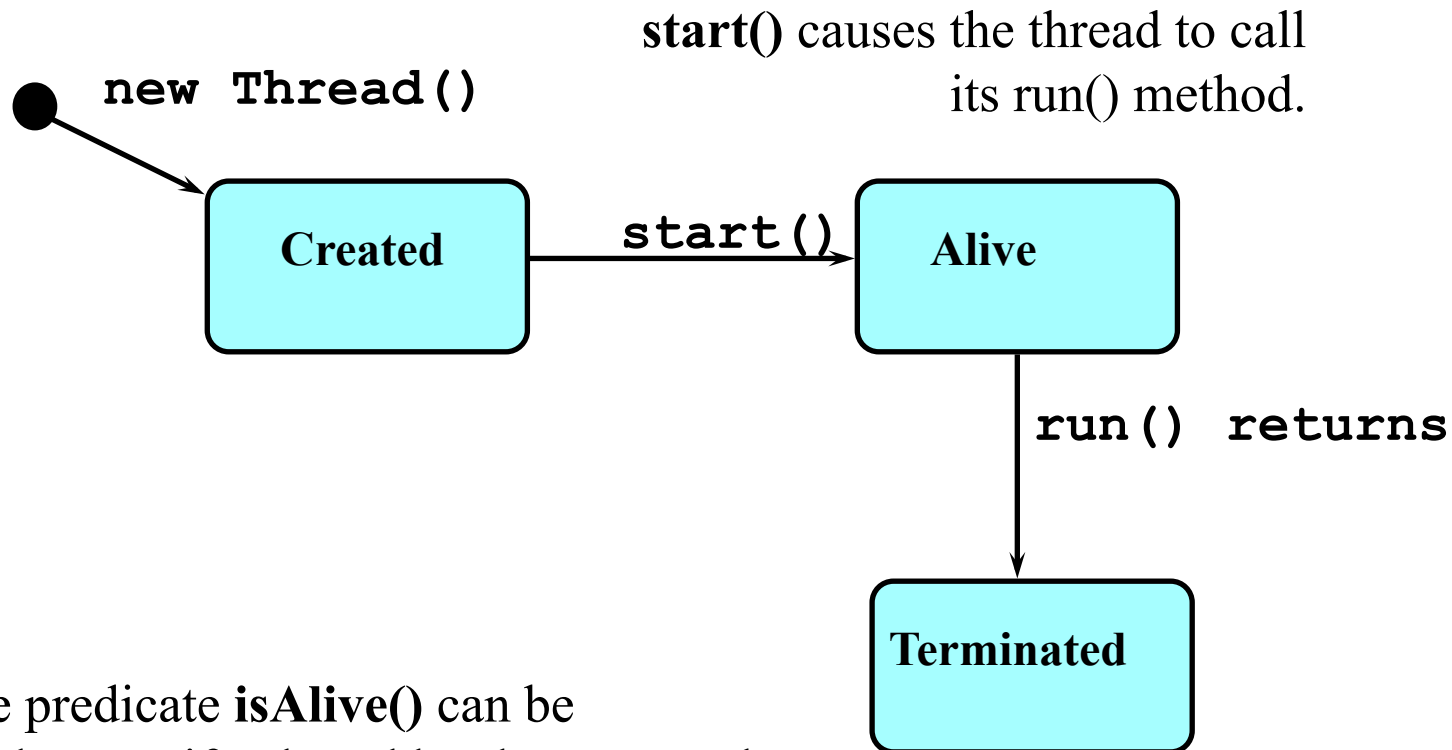
Allocation and construction of a Thread object do not cause the thread to run.

To get a new thread running:

1. Define a subclass **C** of **java.lang.Thread**, overriding **run**
2. Create an object of class **C**
3. Call that object's **start** method
 - Not **run**, which would just be a normal method call
 - **start** sets off a new thread, using **run** as its “main”

thread life-cycle in Java

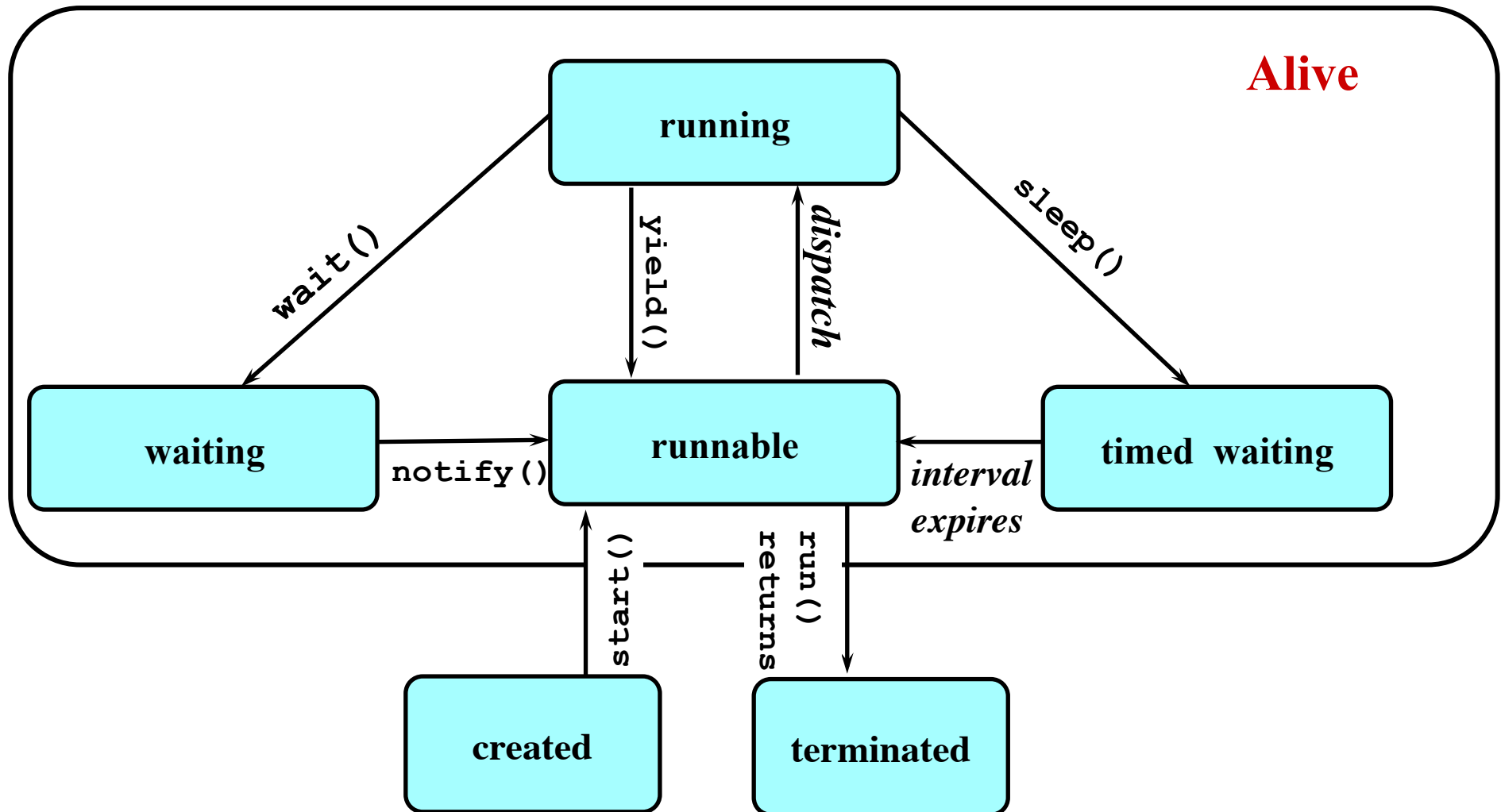
An overview of the life-cycle of a thread as state transitions:



The predicate **isAlive()** can be used to test if a thread has been started but not terminated. Once terminated, it cannot be restarted.

thread **alive** states in Java

Once started, an **alive** thread has a number of substates :



Deprecated thread primitives

Most of the time we allow threads to stop by running to completion

Sometimes we want to stop threads sooner, e.g. when

- user cancels operation
- application needs to shutdown quickly
- Not easy to get threads to stop safely, quickly and reliably
 - Thread.stop and Thread.suspend were an attempt at doing this
 - now deprecated, as too dangerous
 - Java **does not now provide any mechanism for forcing a thread to stop**
 - instead, **ask** the thread to stop what it is doing
- Will discuss this further later, when we talk about safety and deadlock

java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html

Here is a complete example of a useless Java program that starts with one thread and then creates 20 more threads:

```
class C extends java.lang.Thread {
    int i;
    C(int i) { this.i = i; }
    public void run() {
        System.out.println("Thread " + i + " says hi");
        System.out.println("Thread " + i + " says bye");
    }
}
class M {
    public static void main(String[] args) {
        for(int i=1; i <= 20; ++i) {
            C c = new C(i);
            c.start();
        }
    }
}
```

When this program runs, it will print 40 lines of output, one of which is:
Thread 13 says hi

Non-determinism

Concurrent programs are often **non-deterministic**:

it is not possible to tell, by looking at the program, what will happen when it executes.

Concurrent execution

In sequential programs, instructions are executed **in a fixed order** determined by the program and its input. The execution of one procedure does not overlap in time with another. **Deterministic**

In concurrent programs, computational activities may overlap in time and the subprogram executions describing these activities proceed concurrently. **Nondeterministic**

Simple example of a non-deterministic program

Thread A:

```
print "A"  
print "B"
```

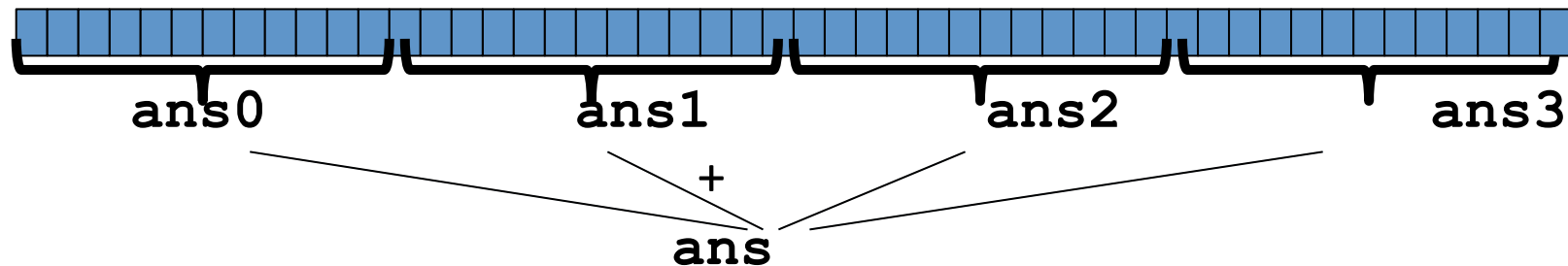
Thread B:

```
print "1"  
print "2"
```

What is the output?

Parallelism idea

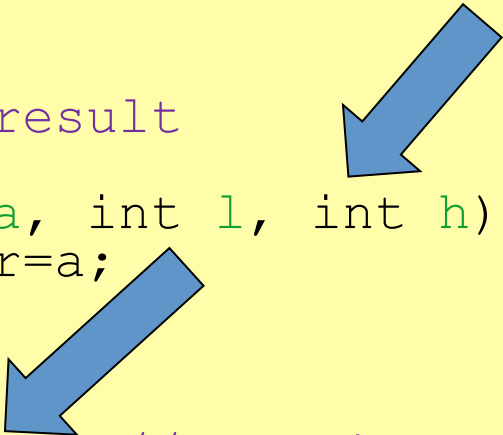
- Example: Sum elements of a large array
- Idea: Have 4 threads simultaneously sum 1/4 of the array
 - Warning: Inferior first approach



- Create 4 *thread objects*, each given a portion of the work
- Call **start()** on each thread object to actually *run* it in parallel
- *Wait* for threads to finish using **join()**
- Add together their 4 answers for the *final result*

First attempt, part 1

```
class SumThread extends java.lang.Thread {  
  
    int lo; // arguments  
    int hi;  
    int[] arr;  
  
    int ans = 0; // result  
  
    SumThread(int[] a, int l, int h) {  
        lo=l; hi=h; arr=a;  
    }  
  
    public void run() { //override must have this type  
        for(int i=lo; i < hi; i++)  
            ans += arr[i];  
    }  
}
```



Because we must override a no-arguments/no-result `run`, we use fields to communicate across threads

First attempt, continued (wrong)

```
class SumThread extends java.lang.Thread {  
    int lo, int hi, int[] arr; // arguments  
    int ans = 0; // result  
    SumThread(int[] a, int l, int h) { ... }  
    public void run() { ... } // override  
}
```

```
int sum(int[] arr) { // can be a static method  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    for(int i=0; i < 4; i++) // do parallel computations  
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);  
    for(int i=0; i < 4; i++) // combine results  
        ans += ts[i].ans;  
    return ans;  
}
```


First attempt, continued (wrong)

```
class SumThread extends java.lang.Thread {  
    int lo, int hi, int[] arr; // arguments  
    int ans = 0; // result  
    SumThread(int[] a, int l, int h) { ... }  
    public void run() { ... } // override  
}
```

WHAT IS WRONG?

```
int sum(int[] arr) { // can be a static method  
    int len = arr.length;  
    int ans = 0;  
    SumThread[] ts = new SumThread[4];  
    for(int i=0; i < 4; i++) // do parallel computations  
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);  
    for(int i=0; i < 4; i++) // combine results  
        ans += ts[i].ans;  
    return ans;  
}
```

Second attempt (still wrong)

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start(); // start not run
    }
    for(int i=0; i < 4; i++) // combine results
        ans += ts[i].ans;
    return ans;
}
```

Basic Fork-Join parallelism

- The only synchronization primitive we will need is `join`, which causes one thread to wait until another thread has terminated.

Third attempt (correct in spirit)

```
class SumThread extends java.lang.Thread {
    int lo, int hi, int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { ... } // override
}
```

```
int sum(int[] arr) { // can be a static method
    int len = arr.length;
    int ans = 0;
    SumThread[] ts = new SumThread[4];
    for(int i=0; i < 4; i++) { // do parallel computations
        ts[i] = new SumThread(arr, i*len/4, (i+1)*len/4);
        ts[i].start();
    }
    for(int i=0; i < 4; i++) { // combine results
        ts[i].join(); // wait for helper to finish!
        ans += ts[i].ans;
    }
    return ans;
}
```

Join (not the most descriptive word)

- The **Thread** class defines various methods you could not implement on your own
 - For example: **start**, which calls **run** in a new thread
- The **join** method is valuable for coordinating this kind of computation
 - Caller blocks until/unless the receiver is done executing (meaning its **run** returns)
 - Else we would have a **race condition** on **ts[i].ans**
- This style of parallel programming is called “fork/join”
- Java detail: code has 1 compile error because **join** may throw **java.lang.InterruptedException**
 - In basic parallel code, should be fine to catch-and-exit

Shared memory?

- Fork-join programs (thankfully) don't require much focus on sharing memory among threads
- But in languages like Java, there is memory being shared.
In our example:
 - **lo**, **hi**, **arr** fields written by “main” thread, read by helper thread
 - **ans** field written by helper thread, read by “main” thread
- When using shared memory, you must avoid race conditions
 - While studying parallelism, we'll stick with **join**
 - With concurrency, we'll learn other ways to synchronize

A better approach

Several reasons why this is a poor parallel algorithm

1. Want code to be reusable and efficient across platforms
 - “Forward-portable” as core count grows
 - So at the *very* least, parameterize by the number of threads

```
int sum(int[] arr, int numThreads) {  
    ... // note: shows idea, but has integer-division bug  
    int subLen = arr.length / numThreads;  
    SumThread[] ts = new SumThread[numThreads];  
    for(int i=0; i < numThreads; i++) {  
        ts[i] = new SumThread(arr, i*subLen, (i+1)*subLen);  
        ts[i].start();  
    }  
    for(int i=0; i < numThreads; i++) {  
        ...  
    }  
    ...  
}
```

A Better Approach

2. Want to use (only) processors “available to you *now*”
 - Not used by other programs or threads in your program
 - Maybe caller is also using parallelism
 - Available cores can change even while your threads run
 - If you have 3 processors available and using 3 threads would take time **X**, then creating 4 threads would take time **1.5X**

```
// numThreads == numProcessors is bad
// if some are needed for other things
int sum(int[] arr, int numThreads) {
    ...
}
```

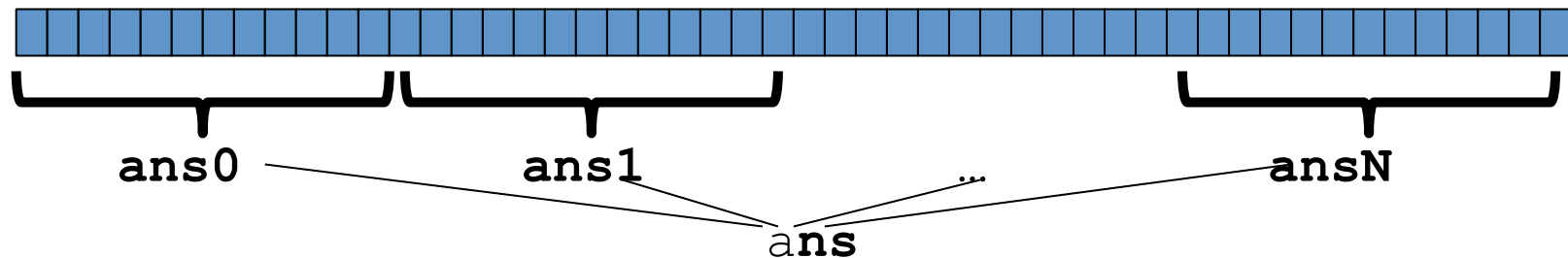

A Better Approach

3. Though unlikely for **sum**, in general subproblems may take significantly different amounts of time
 - Example: Apply method **f** to every array element, but maybe **f** is much slower for some data items
 - Example: Is a large integer prime?
 - If we create 4 threads and all the slow data is processed by 1 of them, we won't get nearly a 4x speedup
 - Example of a **load imbalance**

A Better Approach

The counterintuitive (?) solution to all these problems is to use lots of threads, far more than the number of processors

- But this will require changing our algorithm
- And for constant-factor reasons, abandoning Java's threads



1. Forward-portable: Lots of helpers each doing a small piece
2. Processors available: Hand out “work chunks” as you go
 - If 3 processors available and have 100 threads, then ignoring constant-factor overheads, extra time is < 3%
3. Load imbalance: No problem if slow thread scheduled early enough
 - Variation probably small anyway if pieces of work are small

Naïve algorithm is poor

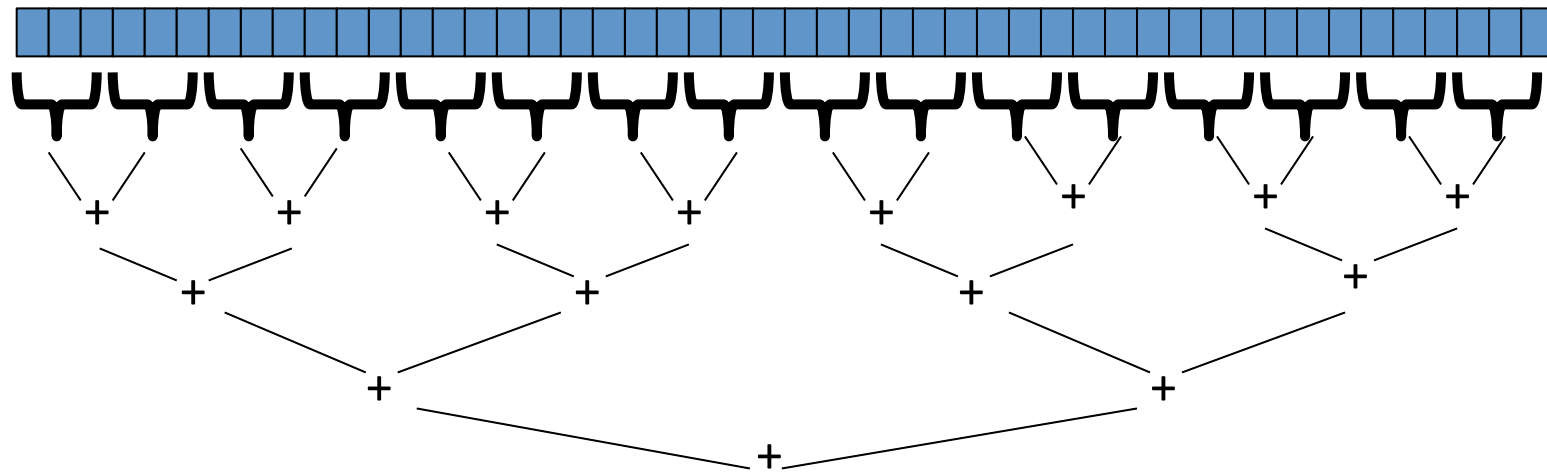
Suppose we create 1 thread to process every 1000 elements

```
int sum(int[] arr) {  
    ...  
    int numThreads = arr.length / 1000;  
    SumThread[] ts = new SumThread[numThreads];  
    ...  
}
```

Then combining results will have `arr.length / 100` additions to do – still linear in size of array

In fact, if we create 1 thread for every 1 element, we recreate a sequential algorithm

A better idea



This is straightforward to implement using divide-and-conquer

- Parallelism for the recursive calls

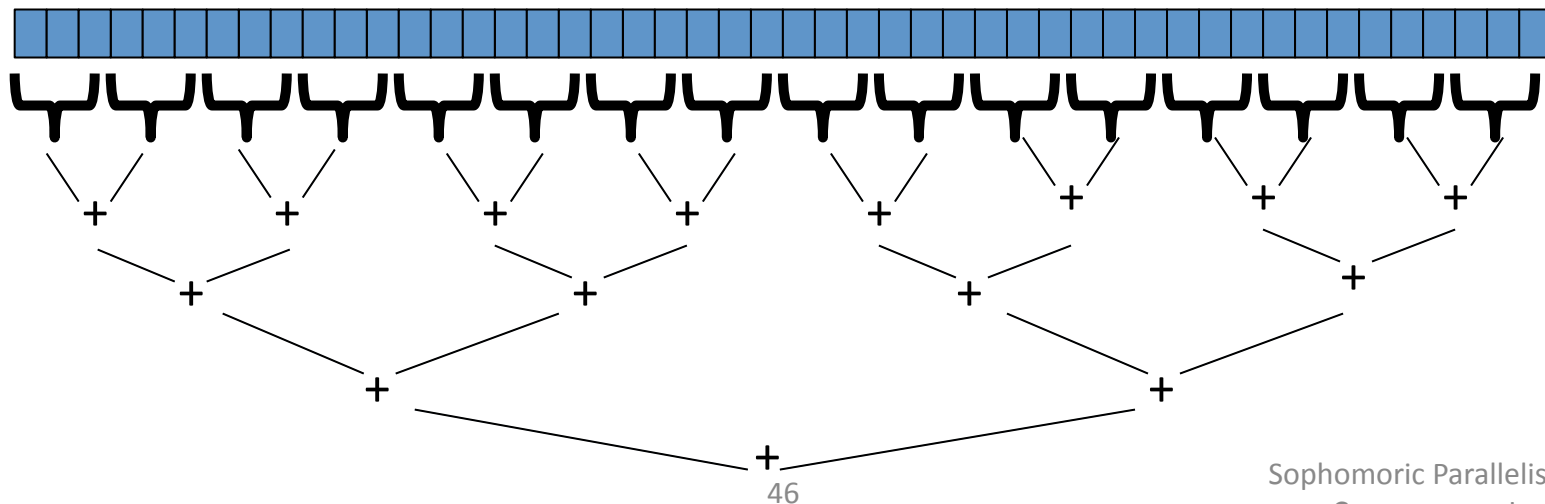
Divide-and-conquer to the rescue!

```
class SumThread extends java.lang.Thread {
    int lo; int hi; int[] arr; // arguments
    int ans = 0; // result
    SumThread(int[] a, int l, int h) { ... }
    public void run() { // override
        if (hi - lo < SEQUENTIAL CUTOFF)
            for (int i = lo; i < hi; i++)
                ans += arr[i];
        else {
            SumThread left = new SumThread(arr, lo, (hi+lo)/2);
            SumThread right = new SumThread(arr, (hi+lo)/2, hi);
            left.start();
            right.start();
            left.join(); // don't move this up a line - why?
            right.join();
            ans = left.ans + right.ans;
        }
    }
}

int sum(int[] arr) {
    SumThread t = new SumThread(arr, 0, arr.length);
    t.run();
    return t.ans;
}
```

Divide-and-conquer really works

- The key is divide-and-conquer parallelizes the result-combining
 - If you have enough processors, total time is height of the tree: $O(\log n)$ (optimal, exponentially faster than sequential $O(n)$)
 - Next lecture: study reality of $P \ll n$ processors
- Will write all our parallel algorithms in this style
 - But using a special library engineered for this style
 - Takes care of scheduling the computation well
 - Often relies on operations being associative (like +)



Being realistic

- In theory, you can divide down to single elements, do all your result-combining in parallel and get optimal speedup
 - Total time $O(n/\text{numProcessors} + \log n)$
- In practice, creating all those threads and communicating swamps the savings, so:
 - Use a *sequential cutoff*, typically around 500-1000
 - Eliminates *almost all* the recursive thread creation (bottom levels of tree)
 - *Exactly* like quicksort switching to insertion sort for small subproblems, but more important here
 - Don't create two recursive threads; create one and do the other "yourself"
 - Cuts the number of threads created by another 2x

Half the threads

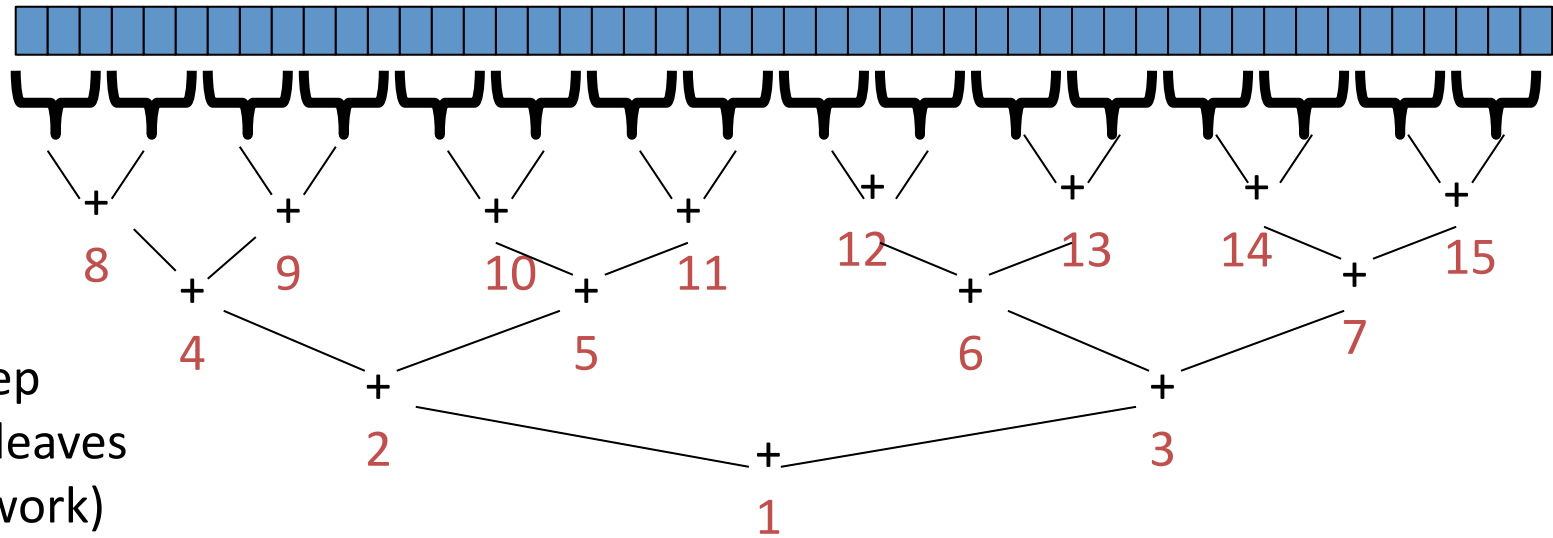
```
// wasteful: don't
SumThread left = ...
SumThread right = ...
left.start();
right.start();
left.join();
right.join();
ans=left.ans+right.ans;
```

```
// better: do
SumThread left = ...
SumThread right = ...
// order of next 4 lines
// essential - why?
left.start();
right.run();
left.join();
ans=left.ans+right.ans;
```

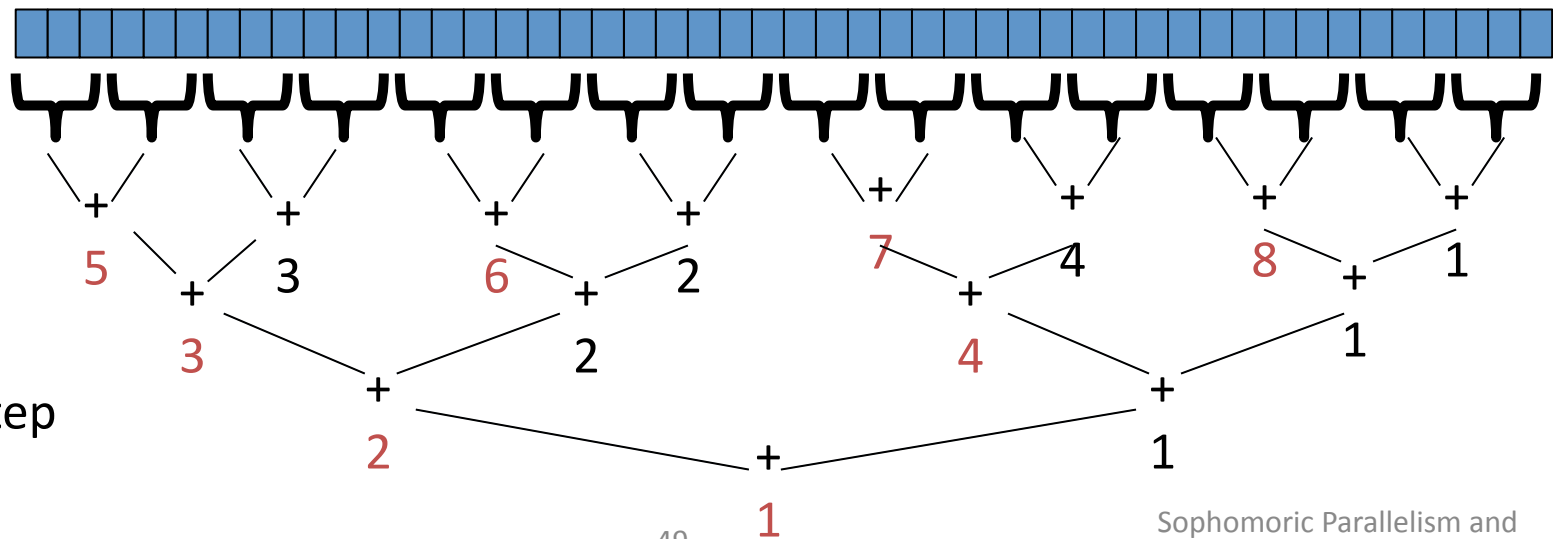
- If a *language* had built-in support for fork-join parallelism, I would expect this hand-optimization to be unnecessary
- But the *library* we are using expects you to do it yourself
 - And the difference is surprisingly substantial
- Again, no difference in theory

Fewer threads pictorially

2 new threads
at each step
(and only leaves
do much work)



1 new thread
at each step



That library, finally

- Even with all this care, Java's threads are too "heavyweight"
 - Constant factors, especially space overhead
 - Creating 20,000 Java threads just a bad idea ☹
- The **ForkJoin Framework** is designed to meet the needs of divide-and-conquer fork-join parallelism
 - In the Java 7 standard libraries
 - (Also available in Java 6 as a downloaded **.jar** file)
 - Section will focus on pragmatics/logistics
 - Similar libraries available for other languages
 - C/C++: Cilk (inventors), Intel's Thread Building Blocks
 - C#: Task Parallel Library
 - ...
 - Library's implementation is a fascinating but advanced topic

Different terms, same basic idea

To use the ForkJoin Framework:

- A little standard set-up code (e.g., create a **ForkJoinPool**)

Don't subclass Thread	Do subclass RecursiveTask<V>
Don't override run	Do override compute
Do not use an ans field	Do return a V from compute
Don't call start	Do call fork
Don't just call join	Do call join which returns answer
Don't call run to hand-optimize	Do call compute to hand-optimize
Don't have a topmost call to run	Do create a pool and call invoke

See the web page for

“A Beginner's Introduction to the ForkJoin Framework”

Example: final version (missing imports)

```
class SumArray extends RecursiveTask<Integer> {
    int lo; int hi; int[] arr; // arguments
    SumArray(int[] a, int l, int h) { ... }
    protected Integer compute() { // return answer
        if(hi - lo < SEQUENTIAL_CUTOFF) {
            int ans = 0;
            for(int i=lo; i < hi; i++)
                ans += arr[i];
            return ans;
        } else {
            SumArray left = new SumArray(arr, lo, (hi+lo)/2);
            SumArray right = new SumArray(arr, (hi+lo)/2, hi);
            left.fork();
            int rightAns = right.compute();
            int leftAns = left.join();
            return leftAns + rightAns;
        }
    }
}

static final ForkJoinPool fjPool = new ForkJoinPool();
int sum(int[] arr) {
    return fjPool.invoke(new SumArray(arr, 0, arr.length));
}
```

Getting good results in practice

- Sequential threshold
 - Library documentation recommends doing approximately 100-5000 basic operations in each “piece” of your algorithm
- Library needs to “warm up”
 - May see slow results before the Java virtual machine re-optimizes the library internals
 - Put your computations in a loop to see the “long-term benefit”
- Wait until your computer has more processors 😊
 - Seriously, overhead may dominate at 4 processors, but parallel programming is likely to become much more important
- Beware memory-hierarchy issues
 - Won’t focus on this, but often crucial for parallel performance