Java basics

First learn some basics built into Java via **`java.lang.Thread`**
- Then a better library for parallel programming

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
   - **`start`** sets off a new thread, using **`run`** as its "main"

What if we instead called the **`run`** method of **`C`**?
- This would just be a normal method call, in the current thread

Let's see how to share memory and coordinate via an example…

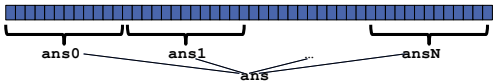
First attempt, part 1

```
class SumThread extends java.lang.Thread {

  int lo; // fields, assigned in the constructor
  int hi; // so threads know what to do.
  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

A Better Approach

The counterintuitive (?) solution to all these problems is to cut up our problem into ***many*** pieces, 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 the "combining of results" part of the code will have `arr.length / 1000` additions
- Linear in size of array (with constant factor 1/1000)
- Previous we had only 4 pieces (Θ(1) to combine)

- In the extreme, suppose we create one thread per element – If we use a for loop to combine the results, we have N iterations
- In either case we get a Θ(N) algorithm with the combining of results as the bottleneck….