Threads, Concurrency, and Parallelism

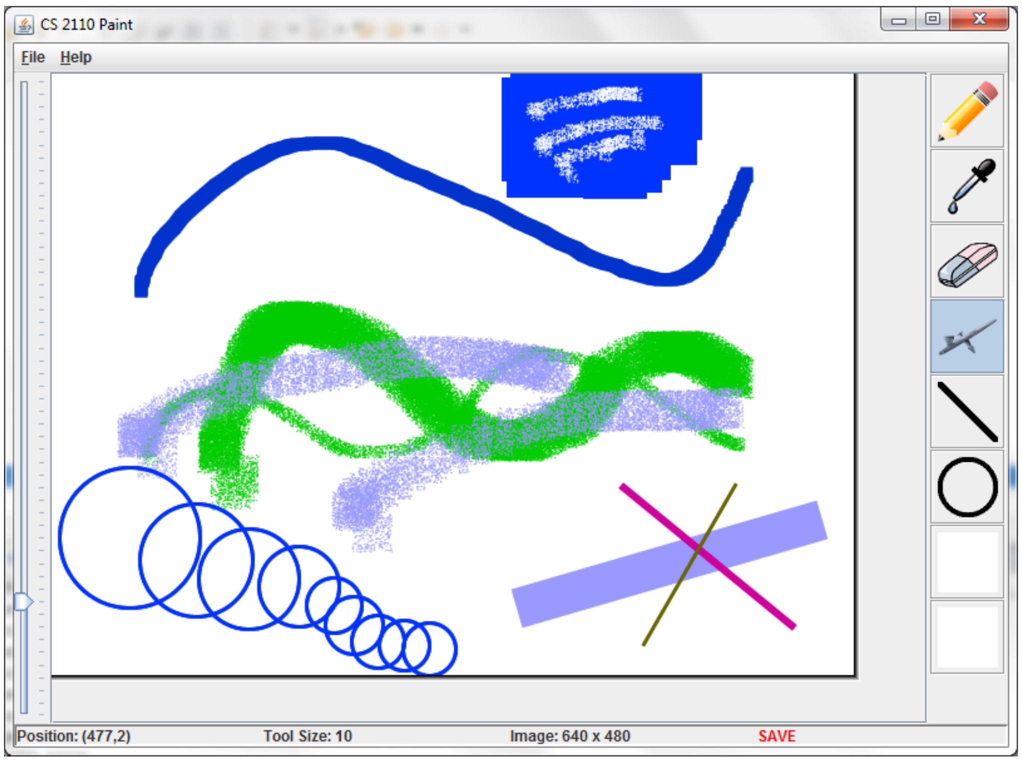
Lecture 24– CS2110 – Spring 2017

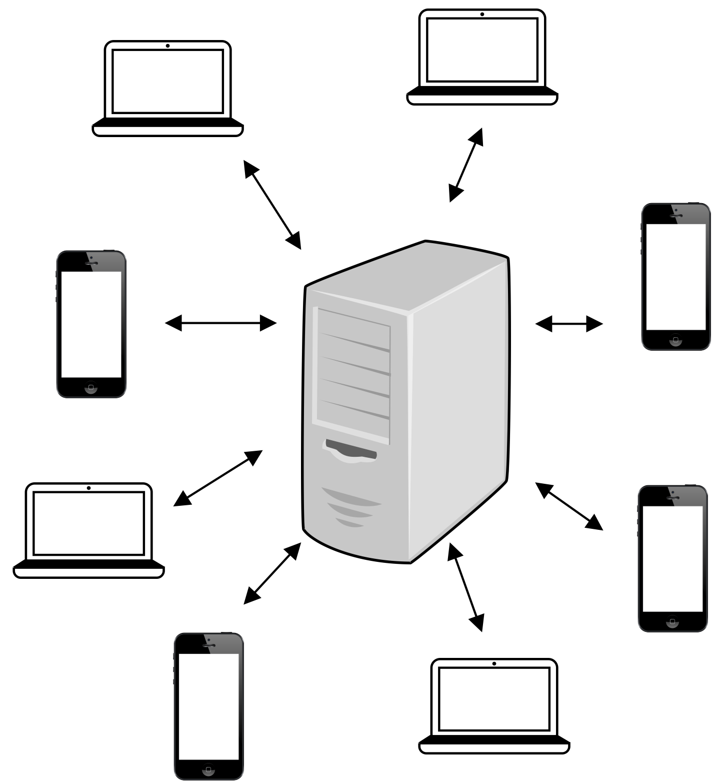
Concurrency & Parallelism

So far, our programs have been *sequential:* they do one thing after another, one thing at a time.

Let's start writing programs that do more than one thing at at a time.

Concurrent Work



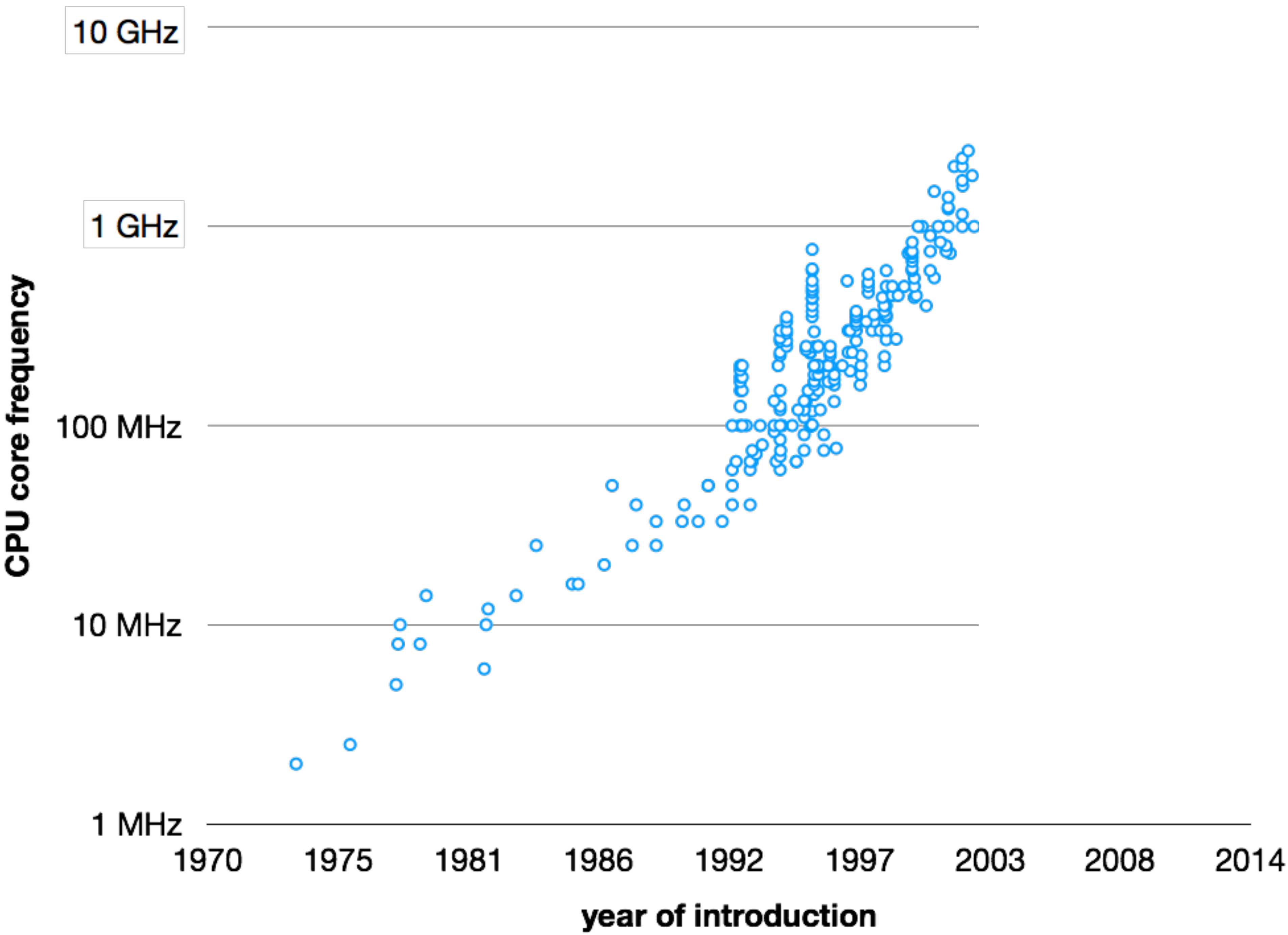


Concurrency in Multiple Machines

- Datacenters and clusters are everywhere:
  - Industrial:
    Google, Microsoft, Amazon, Apple, Facebook…
  - Scientific:
    Several big clusters just in Gates Hall.

Multicore Processors

Every desktop, laptop, tablet, and smartphone you can buy has multiple processors.



10 GHz
1 GHz
100 MHz
10 MHz
1 MHz
CPU core frequency
1970
1975
1981
1986
1992
1997
2003
2008
2014
year of introduction

Concurrency & Parallelism

Parallelism is about using additional computational resources to produce an answer faster.

Concurrency is about controlling access by multiple *threads* to shared resources.

A *thread* or *thread of execution* is a sequential stream of computational work.

Java: What is a Thread?

- *A separate "execution" that runs within a single program and can perform a computational task independently and concurrently with other threads*
- Many applications do their work in just a single thread: the one that called main() at startup
  - But there may still be extra threads...
  - ... Garbage collection runs in a "background" thread
  - GUIs have a separate thread that listens for events and "dispatches" calls to methods to process them
- Today: learn to create new threads of our own in Java

Thread

- A thread is an object that "independently computes"
  - Needs to be created, like any object
  - Then "started" --causes some method to be called. It runs side by side with other threads in the same program; they see the same global data
- The actual executions could occur on different CPU cores, but but don't have to
  - We can also simulate threads by *multiplexing* a smaller number of cores over a larger number of threads

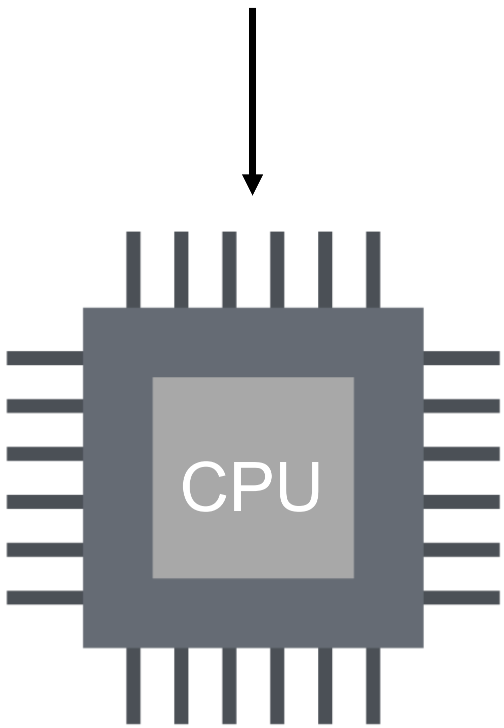
Java class Thread

- threads are instances of class Thread
  - Can create many, but they do consume space & time
- The Java Virtual Machine creates the thread that executes your main method.
- Threads have a priority
  - Higher priority threads are executed preferentially
  - By default, newly created threads have initial priority equal to the thread that created it (but priority can be changed)

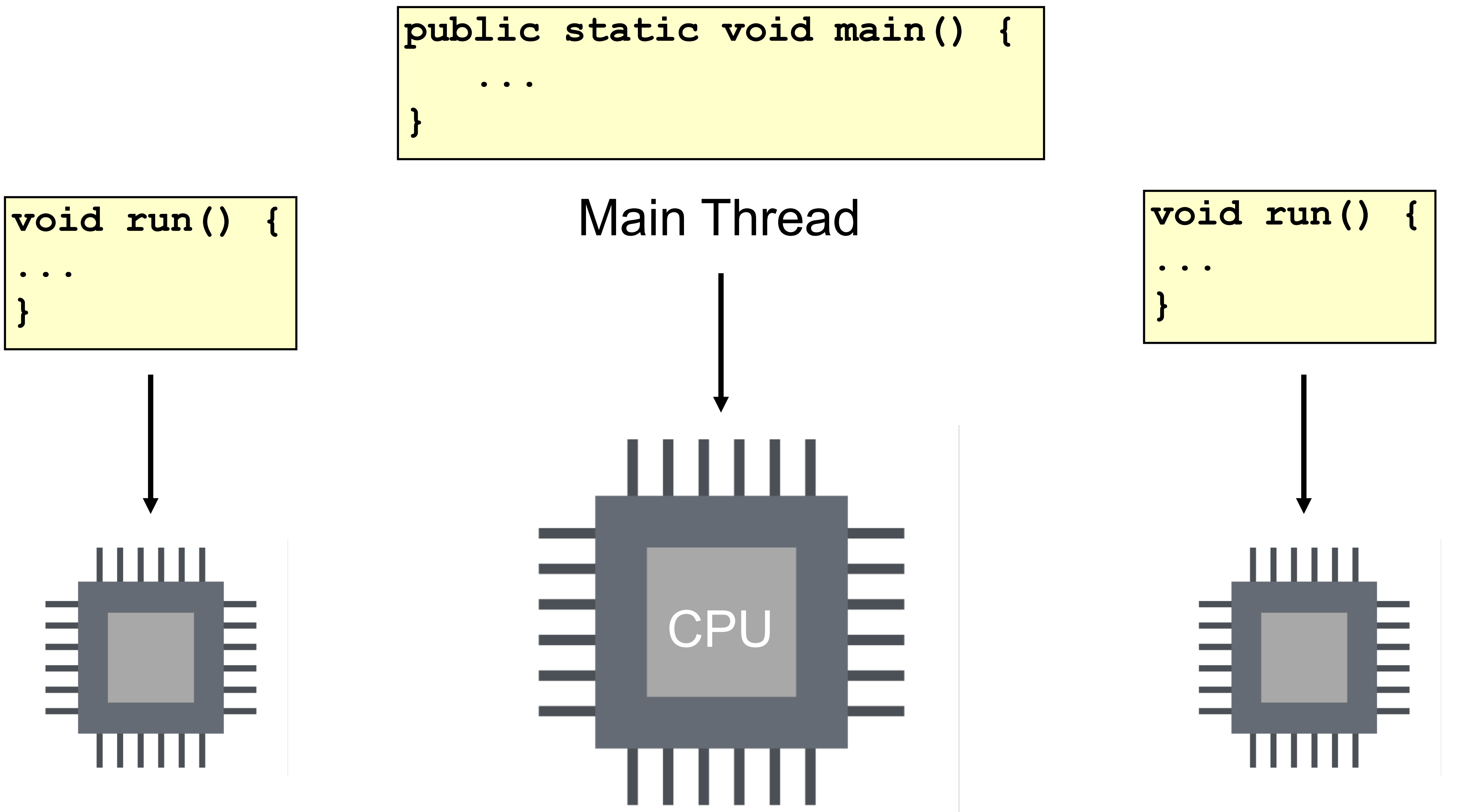
Threads in Java

```
public static void main() {
    ...
}
```

Main Thread



Threads in Java



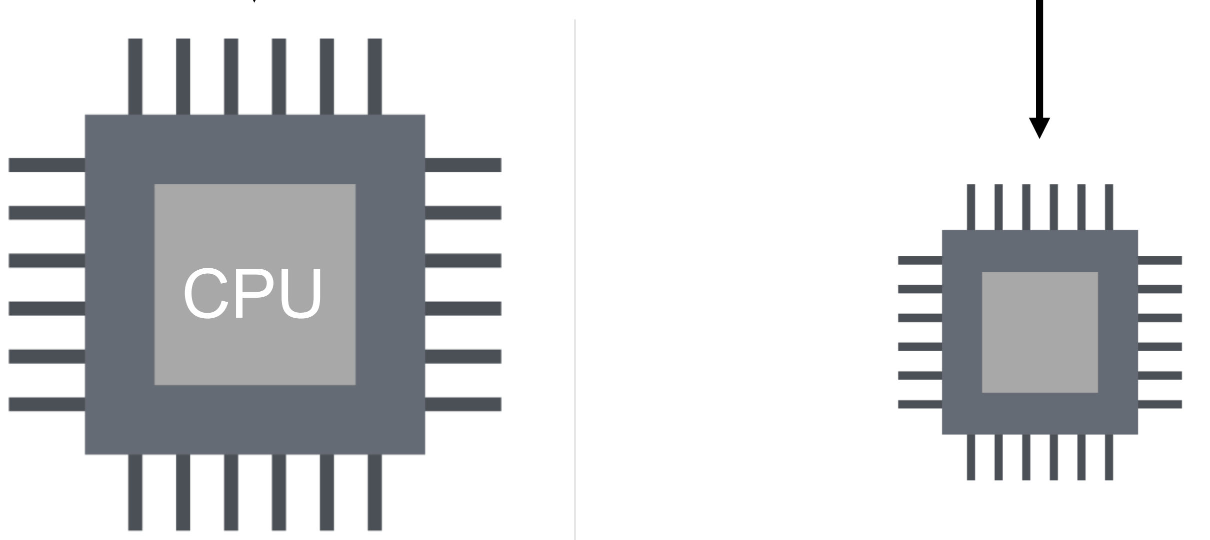
Threads in Java

```
public static void main() {
    ...
}
```

Main Thread

```
void run() {
...
}
```

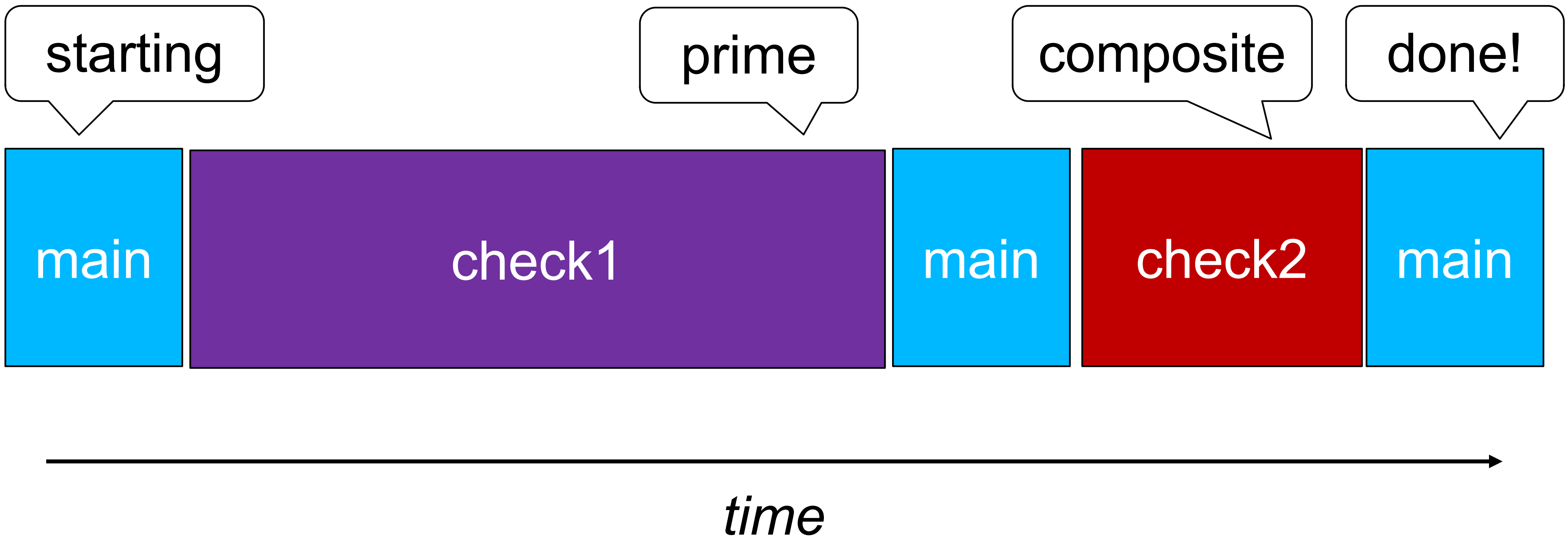
```
void run() {
...
}
```



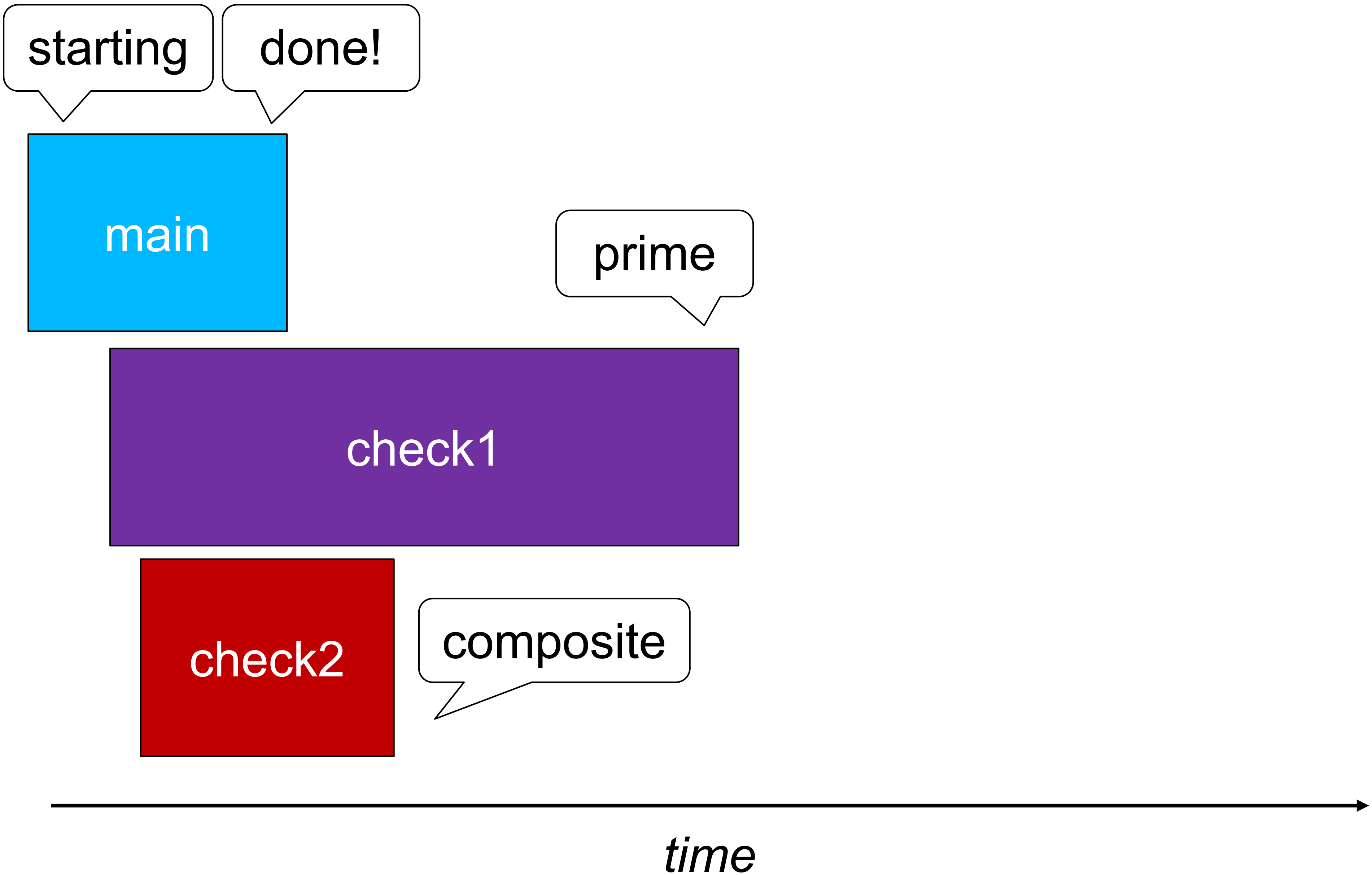
Starting a new Java thread

1. Make a new class that implements Runnable.
2. Put your code for the thread in the run() method.
   Don't call the run method directly!
3. Construct a new Thread with the Runnable:
   ```
   SomeRunnable r = new SomeRunnable(...);
   Thread t = new Thread(r);
   ```
4. Call the Thread's start() method.

Sequential version:



Parallel version:



Creating a new Thread (Method 1)


```java
class PrimeThread extends Thread {
    long a, b;

    PrimeThread(long a, long b) {
        this.a= a; this.b= b;
    }

    @Override public void run() {
        //compute primes between a and b
        ...
    }
}
```

overrides **Thread.run()**

Call **run()** directly?
no new thread is used:
Calling p.start() will run it

```java
PrimeThread p= new PrimeThread(143, 195);
p.start();
```

Do this and
Java invokes **run()** in new thread

Creating a n (Method 1)

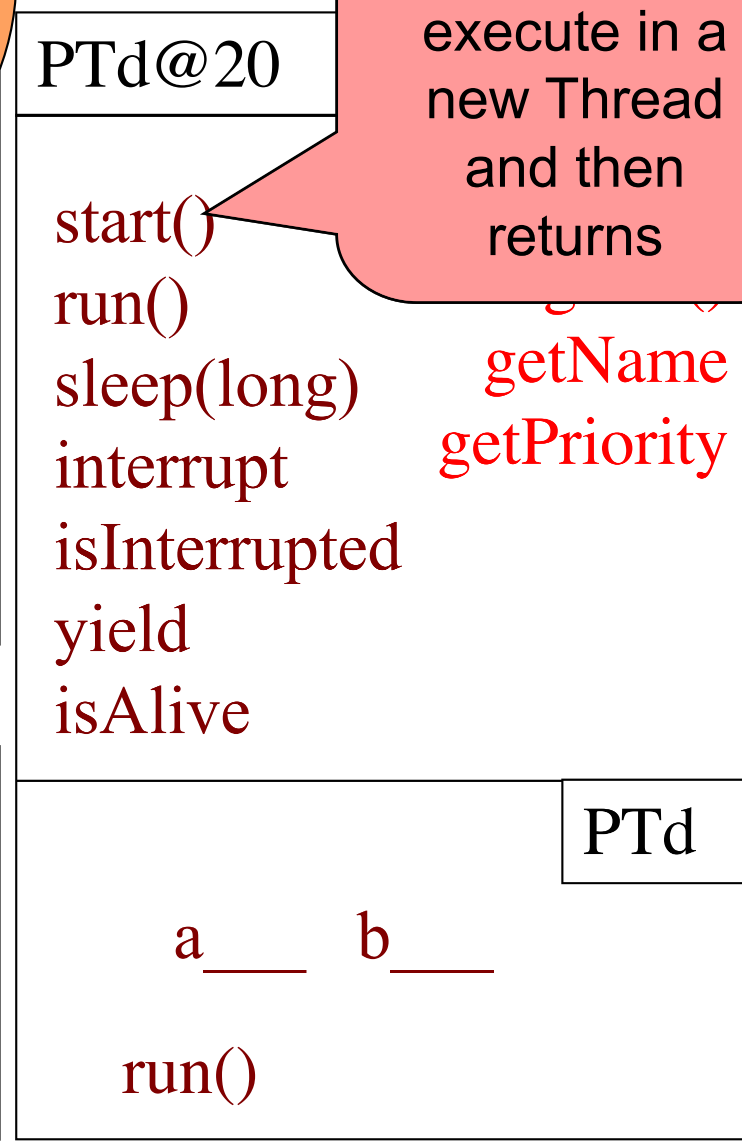
```
class PTd extends
    long a, b;
    PTd (long a, long
        this.a= a; this.b
    }
    @Override public void run() {
        //compute primes between a, b
        ...
    }
}
```

method run() executes in one thread while main program coninues to execute

```
PTd p= new PTd
p.start();

… continue doing other stuff …
```

Calls start() in Thread partition



Calls run() to execute in a new Thread and then returns

Creating a new Thread (Method 2)


```
class PrimeRun implements Runnable {
    long a, b;

    PrimeRun(long a, long b) {
        this.a= a; this.b= b;
    }

    public void run() {
        //compute primes between a and b
        ...
    }
}
```

```
PrimeRun p= new PrimeRun(143, 195);
new Thread(p).start();
```

Example


```
public class ThreadTest extends Thread {
  int M= 1000;       int R= 600;
 public static void main(String[] args) {
    new ThreadTest().start();
    for (int h= 0; true; h= h+1) {
         sleep(M);
         System.out.format("%s %d\n", Thread.currentThread(), h);
    }
}

@Override public void run() {
    for (int k= 0; true; k= k+1) {
         sleep(R);
         System.out.format("%s %d\n", Thread.currentThread(), k);
    }
}
```

We'll demo this with different values of M and R. Code will be on course website

sleep(…) requires a throws clause —or else catch it

Example

Thread name, priority, thread group

```
public class ThreadTest extends Thread {
  int M= 1000;       int R= 600;
 public static void main(String[] args) {
    new ThreadTest().start();
    for (int h= 0; true; h= h+1) {
         sleep(M);
         …format("%s %d\n", Thread.curre
    }
 }

 @Override public void run() {
    for (int k= 0; true; k= k+1) {
         sleep(R);
         …format("%s %d\n", Thread.currentThread(), k);
     }
 }
```

```
Thread[Thread-0,5,main] 0
Thread[main,5,main] 0
Thread[Thread-0,5,main] 1
Thread[Thread-0,5,main] 2
Thread[main,5,main] 1
Thread[Thread-0,5,main] 3
Thread[main,5,main] 2
Thread[Thread-0,5,main] 4
Thread[Thread-0,5,main] 5
Thread[main,5,main] 3
…
```

Example


```
public class ThreadTest extends Thread {
   static boolean ok = true;

   public static void main(String[] args) {
      new ThreadTest().start();
      for (int i = 0; i < 10; i++) {
         System.out.println("waiting...");
         yield();
      }
      ok = false;
   }

   public void run() {
      while (ok) {
         System.out.println("running...");
         yield();
      }
      System.out.println("done");
   }
}
```

If threads happen to be sharing a CPU, yield allows other waiting threads to run.

```
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
running...
waiting...
waiting...
running...
waiting...
running...
done
```

Terminating Threads is Tricky

- The safe way: return from the run() method.
  - Use a *flag* field to tell the thread when to exit.
- Avoid old and dangerous APIs: stop(), interrupt(), suspend(), destroy()…
  - These can leave the thread in a “broken” state.

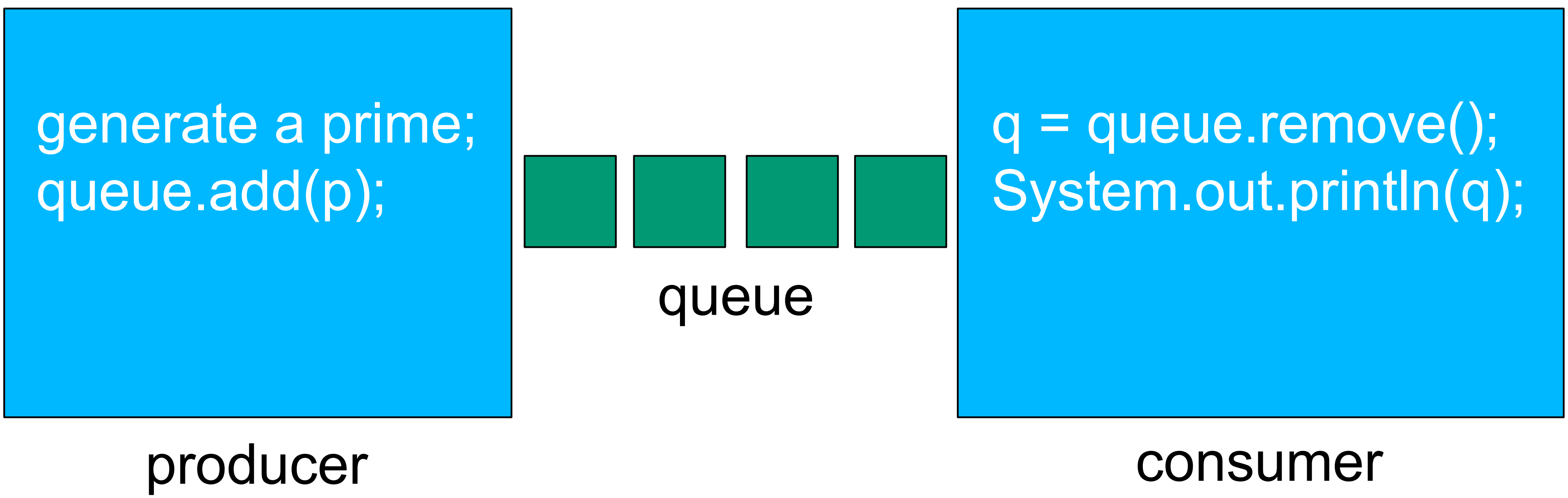
Background (daemon) Threads

- In many applications we have a notion of "foreground" and "background" (daemon) threads
  - Foreground threads are doing visible work, like interacting with the user or updating the display
  - Background threads do things like maintaining data structures (rebalancing trees, garbage collection, etc.) A daemon can continue even when the thread that created it stops.
- On your computer, the same notion of background workers explains why so many things are always running in the task manager.

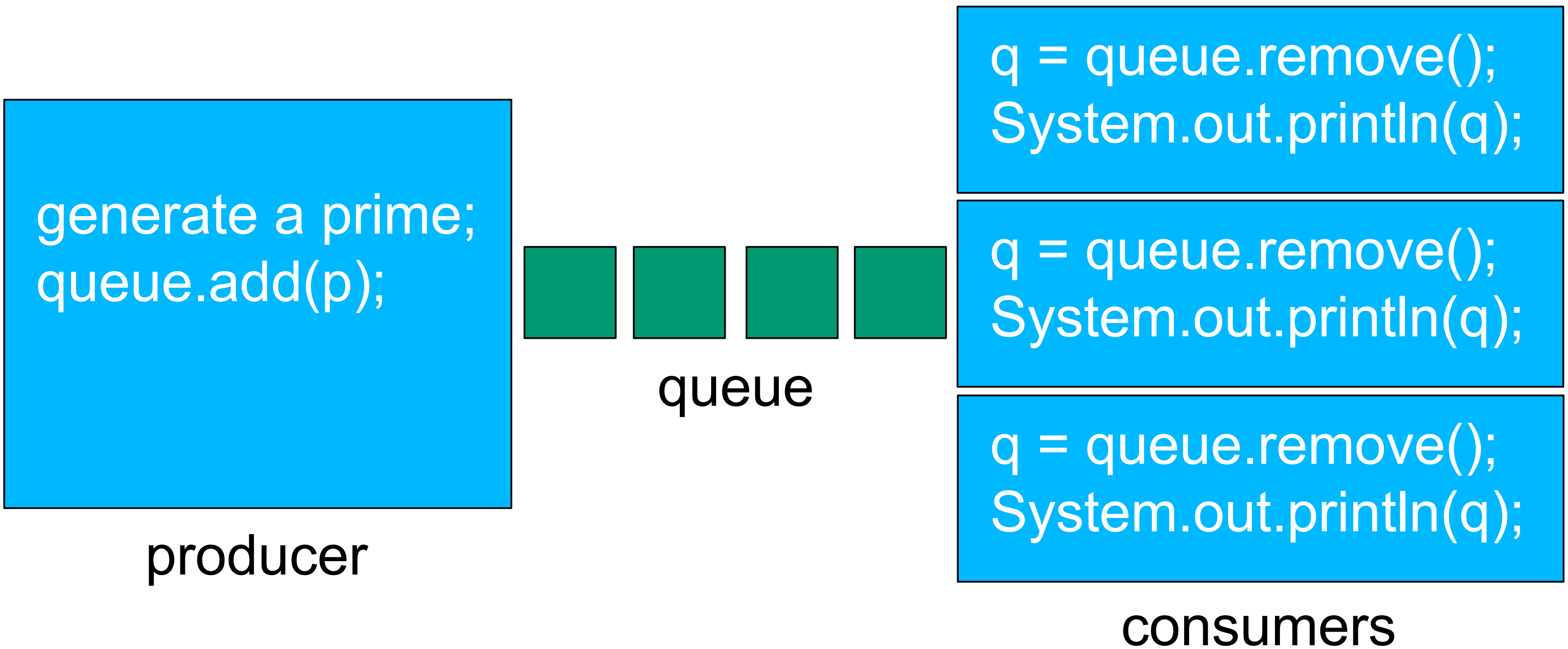
Background (daemon) Threads

- demon: an evil spirit
- daemon. Fernando Corbato, 1963, first to use term. Inspired by Maxwell’s daemon, an imaginary agent in physics and thermodynamics that helped to sort molecules.
- from the Greek δαίμων. Unix System Administration Handbook, page 403: … “Daemons have no particular bias toward good or evil but rather serve to help define a person's character or personality. The ancient Greeks' concept of a "personal daemon" was similar to the modern concept of a "guardian angel"—eudaemonia is the state of being helped or protected by a kindly spirit. As a rule, UNIX systems seem to be infested with both daemons and demons.

Producer & Consumer



Producer & Consumers



Timing is Everything

Thread 1

```
if (!q.isEmpty()) {
  long p = q.remove();
```

Thread 2

```
if (!q.isEmpty()) {
  long p = q.remove();
```

A Fortunate Interleaving

Thread 1 | Thread 2

queue length: 1

Thread 1:

```
if (!q.isEmpty()) {
  long p = q.remove();
```

Condition is true!
Remove an element.

queue length: 0

Thread 2:

```
if (!q.isEmpty()) {
  long p = q.remove();
```

Condition is false.
Do nothing.

queue length: 0

time

Another Fortunate Interleaving

Thread 1 | Thread 2

queue length: 1

Thread 2:

```
if (!q.isEmpty()) {
  long p = q.remove();
```

queue length: 0

Thread 1:

```
if (!q.isEmpty()) {
  long p = q.remove();
```

queue length: 0

time

An Unfortunate Interleaving

Thread 1 | Thread 2

queue length: 1

Condition is true. (Thread 2)

```
if (!q.isEmpty()) {
```

queue length: 1

(Thread 1)

```
if (!q.isEmpty()) {
```

Condition is **still** true.

queue length: 1

Remove an element. (Thread 2)

```
  long p = q.remove();
```

queue length: 0

(Thread 1)

```
  long p = q.remove();
```

NoSuchElementException!

time

Beginning to think about avoiding race conditions

You know that race conditions can create problems:

Basic idea of race condition: Two different threads access the same variable in a way that destroys correctness.

- Process t1

  …

  x= x + 1;

  Process t2

  …

  x= x + 1;

But x= x+1; is not an "atomic action": it takes several step

Two threads may want to use the same stack, or Hash table, or linked list, or … at the same time.

Synchronization

- Java has one primary tool for preventing race conditions. you must use it by carefully and explicitly – it isn't automatic.
  - Called a synchronization barrier
  - Think of it as a kind of lock
    - Even if several threads try to acquire the lock at once, only one can succeed at a time, while others wait
    - When it releases the lock, another thread can acquire it
    - Can't predict the order in which contending threads get the lock but it should be "fair" if priorities are the same

Synchronized Blocks

a.k.a. *locks* or *mutual exclusion*

```
synchronized (obj) {
    ...
}
```

At most one thread can be in a synchronized (obj) block for the same obj at any given time.

Synchronized Blocks

a.k.a. *locks* or *mutual exclusion*

```
synchronized (q) {
  if (!q.isEmpty()) {
    q.remove();
  }
}
```

At most one consumer thread can be trying to remove something from the queue at a time.

Solution: use with synchronization


```
private Stack<String> stack= new Stack<String>();

public void doSomething() {
    synchronized (stack) {
      if (stack.isEmpty()) return;
      String s= stack.pop();
    }
    //do something with s...
}
```

synchronized block

- Put critical operations in a **synchronized** block
- Can't be interrupted by other **synchronized blocks on the same object**
- Can run concurrently with non-synchronized code
- Or code synchronized on a different object!

Example: a lucky scenario


```
private Stack<String> stack= new Stack<String>();

public void doSomething() {
   if (stack.isEmpty()) return;
   String s= stack.pop();
   //do something with s...
}
```

Suppose threads A and B want to call `doSomething()`, and there is one element on the stack

1. thread A tests `stack.isEmpty()` false
2. thread A pops ⇒ stack is now empty
3. thread B tests `stack.isEmpty()` ⇒ true
4. thread B just returns – nothing to do

What Can i Be at the End?

Thread 1

Thread 2

Initially, i = 0

```
i += 1;
```

```
i += 1;
```

Finally, i = ?

What Can i Be at the End?

Thread 1

Thread 2

Initially, i = 0

```
i += 1;
```

```
i += 1;
```

Finally, i = 2 **or 1**!

What Can i Be at the End?

Thread 1

Thread 2

Initially, i = 0

```
tmp = load i;
tmp = tmp + 1;
store tmp to i;
```

```
tmp = load i;
tmp = tmp + 1;
store tmp to i;
```

What Can i Be at the End?

Thread 1 | Thread 2

time

Initially, i = 0

```
tmp = load i;
```

Load 0 from memory

Load 0 from memory

```
tmp = load i;
```

```
tmp = tmp + 1;
store tmp to i;
```

Store 1 to memory

Store 1 to memory

```
tmp = tmp + 1;
store tmp to i;
```

Finally, i = 1

A Pretty Good Rule

Whenever you read or write variables that multiple threads might access, *always* wrap the code in a synchronized block.

(Following this rule will not magically make your code correct, and it is not always strictly necessary to write correct code. But it is usually a good idea.)

Race Conditions

When the result of running two (or more) threads depends on the relative timing of the executions.

- Can cause extremely subtle bugs!
- Bugs that seem to disappear when you look for them!

Race conditions

- Typical race condition: two processes wanting to change a stack at the same time. Or make conflicting changes to a database at the same time.
- Race conditions are bad news
  - Race conditions can cause many kinds of bugs, not just the example we see here!
  - Common cause for "blue screens": null pointer exceptions, damaged data structures
  - Concurrency makes proving programs correct much harder!

Deadlock

Use synchronized blocks to avoid race conditions.

But *locks* are shared resources that can create their own problems. Like other resources: files, network sockets, etc.

If thread A holds a resource that thread B needs to continue, and thread B holds a different resource that thread A needs to continue, you have **deadlock**.

Dining philosopher problem

Five philosophers sitting at a table.

Each repeatedly does this:

1. think
2. eat

What do they eat? spaghetti.

Need TWO forks to eat spaghetti!

Dining philosopher problem

Each does repeatedly :

1. think
2. eat (2 forks)

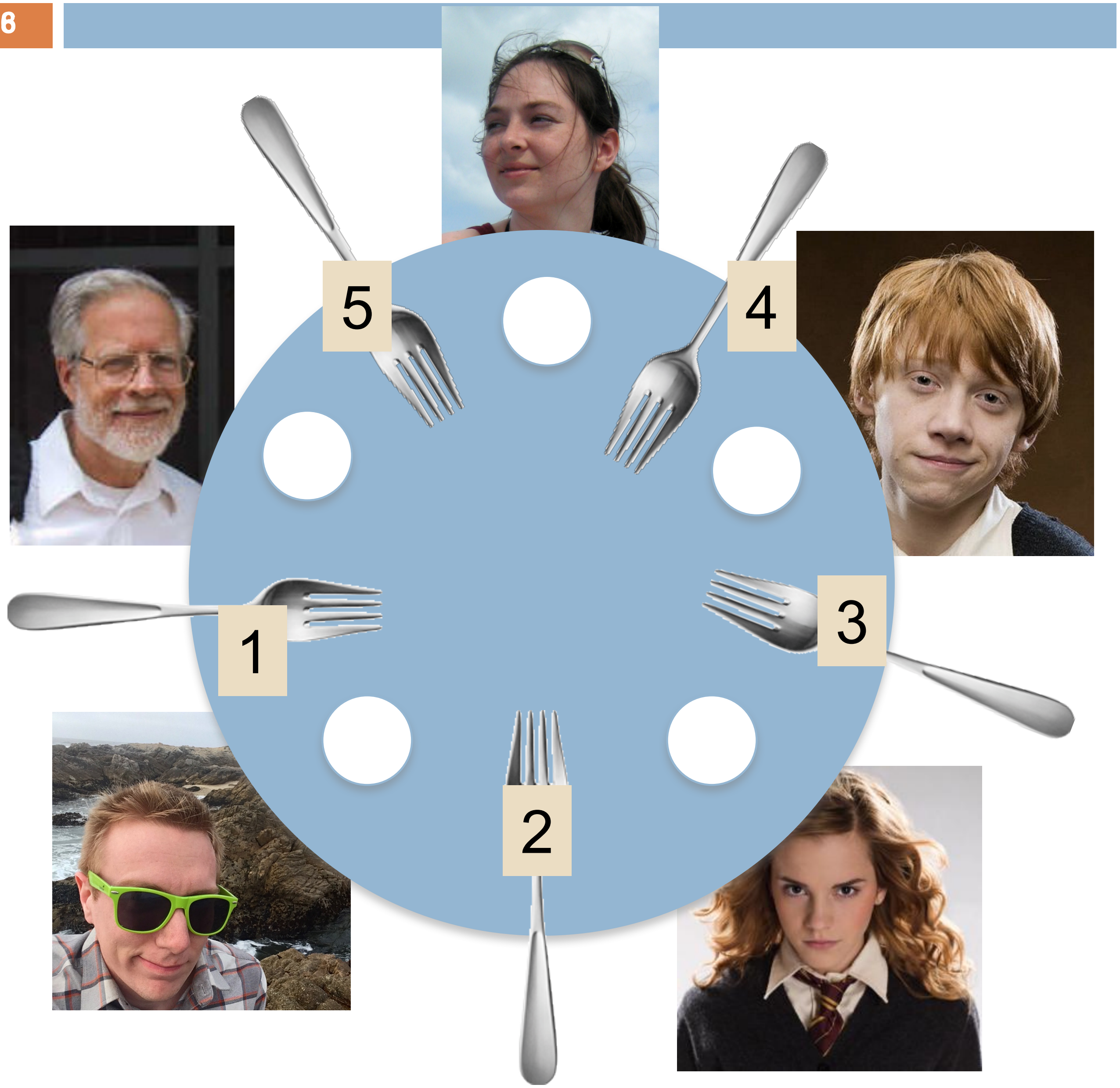
eat is then:

pick up left fork
pick up right fork
eat spaghetti
put down left fork
put down right fork

At one point, they all pick up their left forks

DEADLOCK!

Dining philosopher problem



Simple solution to deadlock:

Number the forks. Pick up smaller one first

1. think
2. eat (2 forks)

eat is then:

pick up smaller fork
pick up bigger fork
eat spaghetti
put down bigger fork
put down smaller fork