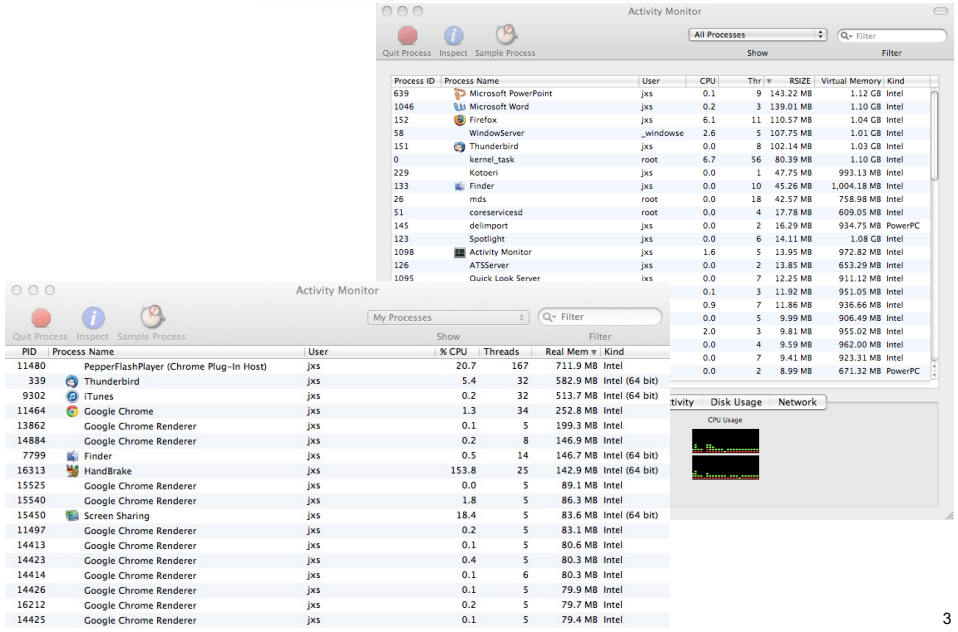
Concurrency with Threads

Processes

- One of the most important concepts in all modern operating systems
- A process is a container (or execution environment) for a program (software) in execution.
- Any software is executed with one or more processes.
  - Java VM (JVM), MS Word, Excel, PPT, Firefox, iTunes, Google Chrome, Kindle, Java IDE, etc.

Example Processes



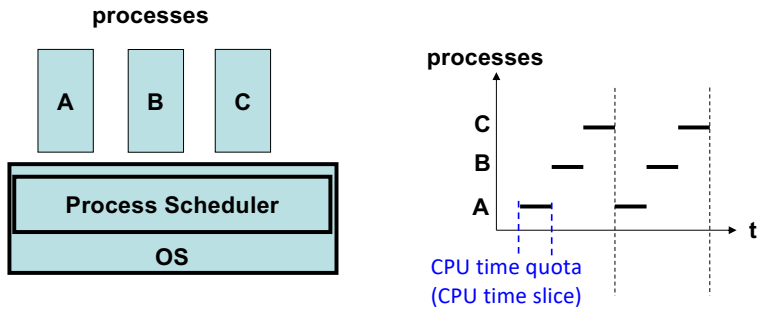
Multi-Tasking (Time Sharing)

- All modern OSes support multi-tasking.
  - Doing multiple things with multiple processes on a single CPU (more precisely, a single CPU core)
    - e.g., Writing a document with an editor while having an online meeting with a conferencing tool.
    - Writing code with an IDE while playing a YouTube video
    - Reading an e-book with an e-book reader while playing music with a music player.
    - Running an activity monitoring (e.g. step counting) app while running other apps
      - e.g., Walking/running on a treadmill while watching a movie

- At any moment, a single CPU core can execute a single program (process).

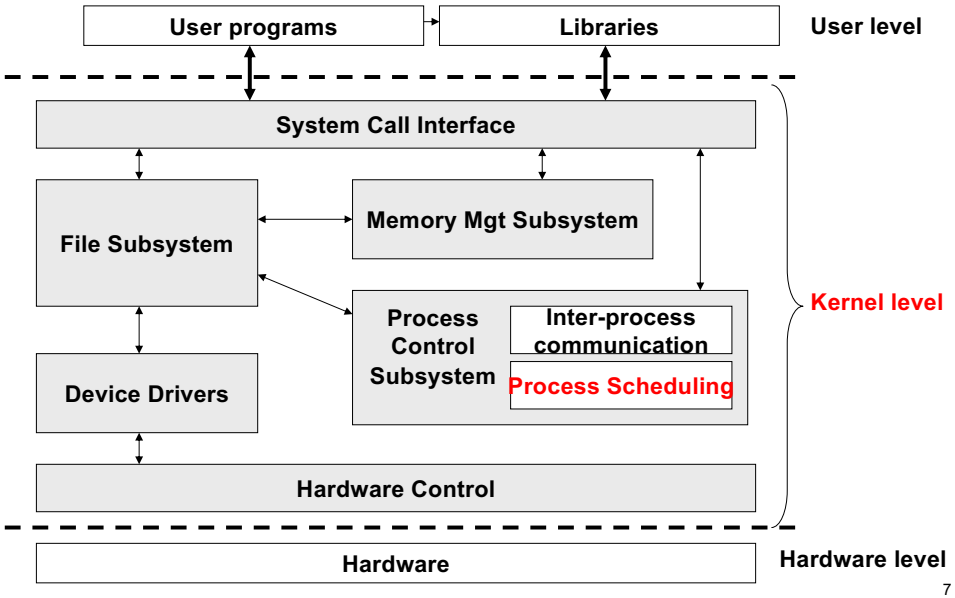
- *Pseudo concurrency/parallelism*: An illusion for human eyes as if multiple processes run at the same time.
  - OS periodically assigns one process to another to the CPU core.
    - every several tens of milliseconds to several hundreds of milliseconds
      - CPU time quota (CPU time slice)
    - A CPU core is multiplexed among processes; processes do NOT run in completely parallel.

Process Scheduling



- Processes run in turn by spending the same amount of CPU time at a time.

An Architectural View of an OS



Why Threads?

- Good old days: Process-based, coarse-grained concurrency was enough
  - e.g., Writing a document with an editor while having an online meeting with a conferencing tool.

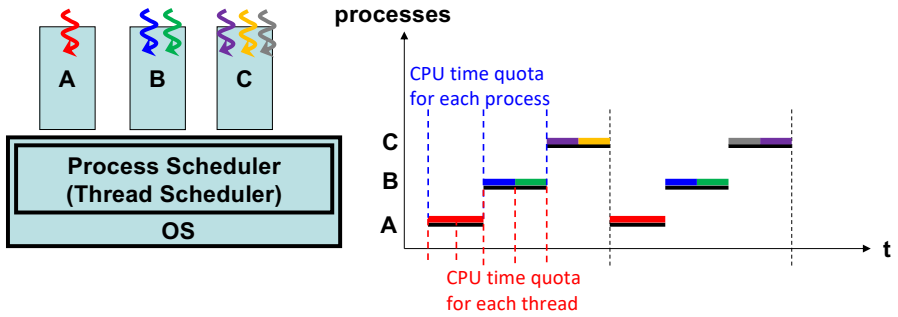
- Now: Finer-grained concurrency is required.
  - Many programs are required to run multiple in-process tasks at the same time.
    - Running different tasks at the same time.
    - Running the same or a similar tasks for different users/clients at the same time.
    - Splitting a computationally heavy (big) task into lightweight (smaller) tasks, and running those smaller tasks at the same time.
  - Assign those in-process tasks to threads.

- Many programs are required to run different in-process tasks at the same time.
  - Document editor
    - Editing a document
    - Performing spelling and grammar check in the background
    - Uploading/downloading (synching) documents to/from a cloud
  - Web browser
    - Running multiple tabs (e.g., displaying a web page on a tab while playing a YouTube video on another tab).
  - Music player
    - Playing music
    - Downloading music and its metadata (lyrics, etc.)
    - Streaming music to
  - Activity monitoring app
    - Running sensors (e.g. accelerometer, gyro, barometric pressure sensor)
    - Calculating step counts, flight/floor climbing, walking/running/swim distance

- Many programs are required to run the same or a similar task for different users/clients at the same time.
  - Web server
    - Does the same sequence of tasks to different clients (remote browsers) at the same time
      - Accepting and parsing an HTTP request
      - Finding a requested file
      - Making a response HTTP message (incl. header, payload, etc.)
      - Returning the requested file's content with the HTTP message
- Each process splits a big task and runs smaller tasks at the same time.
  - e.g., Calculating vax rate for each county in the US

- *Fine-grained* pseudo parallelism: An illusion for human eyes as if multiple threads run on a process at the same time.
  - OS periodically assigns one thread to another to the CPU core.
    - Every tens of milliseconds to approx. 100 milliseconds
      - Usually, shorter CPU time quota for desktop and mobile OSes, and longer CPU time quota for server-side OSes
    - A CPU core is multiplexed among processes and threads; processes/threads do NOT run in a completely parallel manner on a single CPU core.

Thread Scheduling



- Threads run in turn by spending the same amount of CPU time at a time.

Summary: Why Threads?

- Responsiveness/availability
  - If in-process tasks runs sequentially, your app would not be responsive/available for you.
  - Threads allow your app to continue running even if a part of it is blocked for I/O or is performing a long operation.
- Efficiency
  - Threads are more lightweight than processes.
    - Creating a process is approx. 30 times slower than creating a thread.
    - Process switching is approx. 5 times slower than thread switching.
  - In-process tasks can run in parallel on a multi-core CPU

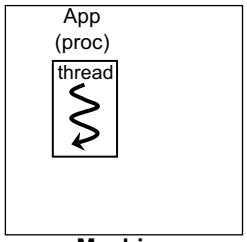
Other Viewpoints to Threads

- CPU clock speed does not increase as it did in the past.
  - Physical material barrier to increase clock speed and # of transistors in a single CPU
  - Excessive power consumption and heat
    - Intuitively, power consumption increases exponentially as clock speed increases.
    - Harder to economically justify power supply cost with increased clock speed in many industry sectors.

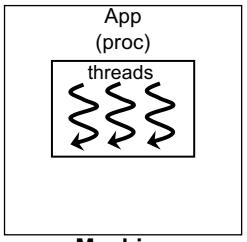
- In mobile computing,
  - Mobile devices often use multi-core CPUs with lower clock speed to reduce their power consumption and extend their battery life.
    - Favor increased density of cores, not increased speed on a each core.
      - Kindle Fire ($50): 4 cores, 1.3 GHz
      - iPhone: 6 cores (2 high-performance and 4 energy-efficient cores)
      - iPad Pro: 8 cores (4 high-performance and 4 energy-efficient cores)
- In cloud computing
  - Servers often use multi-core CPUs with lower clock speed to reduce power supply cost and CO2 emission.
    - Enormous energy consumption
      - Google paid $2M/mo for electricity bills (‘07).
    - Significant CO2 emission
      - The IT industry produces more CO2 emission than the aviation industry.

- You are expected to increase your app’s performance by increasing its concurrency.
  - This will lead you to “green” computing.
- Ultimate goals of multi-threaded applications: Responsiveness and efficiency improvement

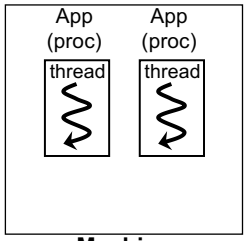
Terminology: Single- and Multi-threaded



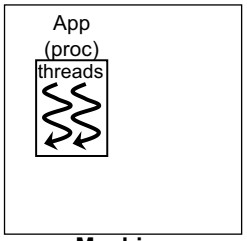
Single-threaded (sequential) app



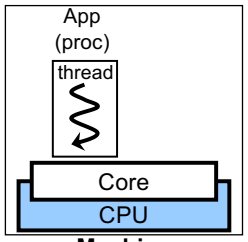
Multi-threaded (concurrent) app



Single-threaded (sequential) apps

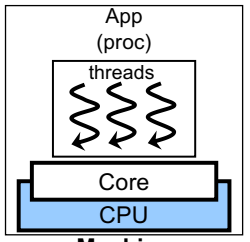


Multi-threaded (concurrent) app



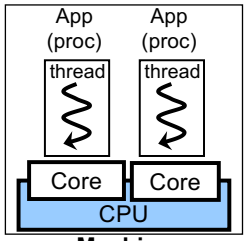
Single-threaded (sequential) app

Executed sequentially



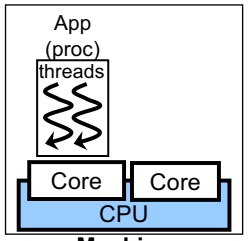
Multi-threaded (concurrent) app

Executed concurrently



Single-threaded (sequential) apps

Two apps runs in parallel. Each app is executed sequentially.



Multi-threaded (concurrent) app

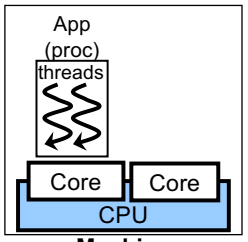
Executed concurrently, not in parallel.

Terminology: Concurrent and Parallel

- Traditionally, the semantics of concurrency and parallelism have been orthogonal (separated clearly).
  - Being "concurrent" means…
    - A program is implemented with multiple threads.
    - Concurrency: a concept (or technique) in program implementation
  - Being "parallel" means…
    - Programs are executed in parallel.
    - Parallelism: a concept (or technique) in program execution
- As a result…
  - Sequential (non-concurrent) programs may be executed in parallel.
  - Concurrent programs may not be executed in parallel.

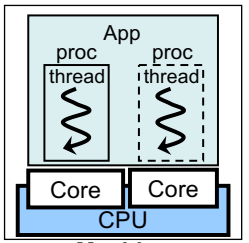
Modern Thread Distribution (Thread Dispatching)

- Many modern OSes automatically distribute in-process threads onto different CPU cores in multi-core CPU environments.
- Concurrent programs often run in parallel (i.e., concurrent programs often means parallel programs)
  - as far as they run on modern OSes in multi-core CPU environments.



Multi-threaded (concurrent) app

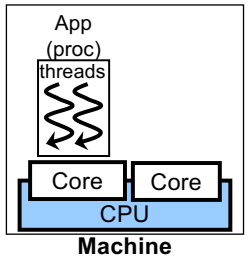
Executed concurrently, not in parallel.



Multi-threaded (concurrent) app

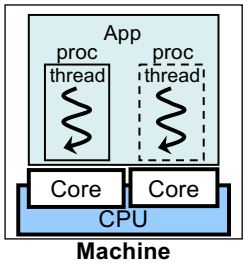
Executed in parallel

Terminology: Concurrent or Parallel?



[1] Multi-threaded (concurrent) app

NOT parallel



[2] Multi-threaded (concurrent) app

Parallel

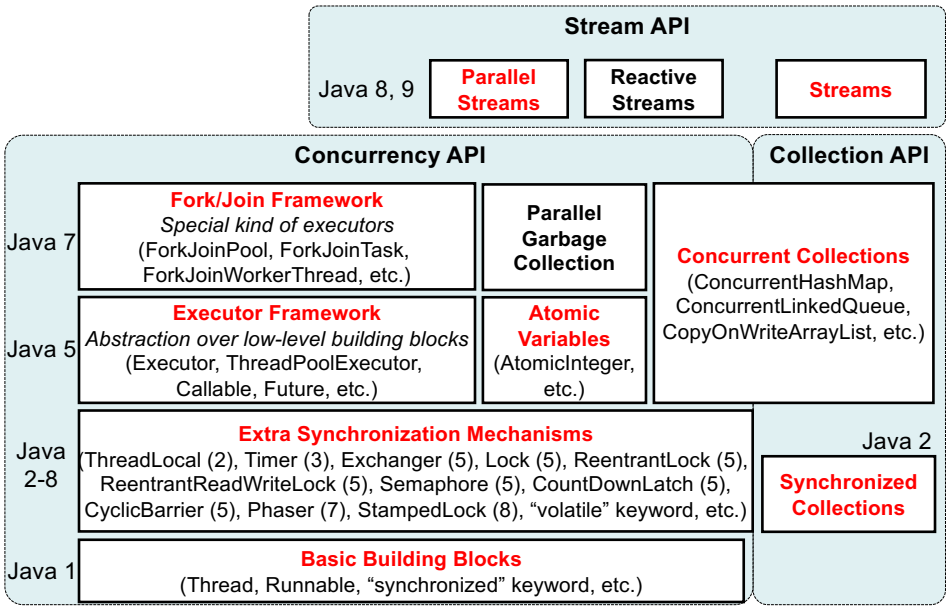
- The terms "concurrent" [1] and "parallel" [2] are NOT equivalent in principle, but they are becoming equivalent more often in practice.
- 2 terminology preferences exist
  - If you think [2] is a special case of [1], you may prefer the term "concurrent."
  - If you think [1] is a special case of [2], you may prefer the term "parallel."

In Java…

- The terms "concurrent" and "parallel" are somewhat mixed up in Java.
  - Java 5 (2004) introduced `java.util.concurrent`, which has been enhanced afterwards until now.
    - "Concurrent" collections such as `ConcurrentHashMap`
    - The Executor framework
      - An extension/abstraction over low-level threads
  - Java 7 (2011)
    - The Fork/Join framework, an extension to the Executor framework
      - The term "parallel" appeared in its documentation, although it was placed in `java.util.concurrent`.
    - "Parallel" garbage collector, which performs garbage collection with multiple threads.
  - Java 8 (2014)
    - "Parallel" streams, which allows for "parallel" (multi-threaded) operations on streams with lambda expressions

Java Threads

Concurrency API in Java



Java Threads

- Every Java program has at least one *thread of control*.
  - **`main()`** runs with a thread (of control) on a JVM.
    - When a JVM starts, it implicitly (or automatically) creates the "main" thread.

```
class HelloWorldApp{
  public static void main(String[] args){
    System.out.println("Hello World!");
  } }
```

```
>java HelloWorldApp
```


- If you need extra threads in addition to the main thread, you need to create them *explicitly.*
- 4 things to do:
  - Define a class implementing the **`java.lang.Runnable`** interface
    - **`public abstract void run();`**
  - Write a threaded/concurrent task in **`run()`** in the class
  - Instantiate **`java.lang.Thread`** and associate a Runnable object with the thread
  - Start (call **`start()`** on) the instantiated thread.
    - **`run()`** is automatically called on the thread.

An Example Code: Creating a Thread

- GreetingRunnable.java

```
class GreetingRunnable implements Runnable{
  private String greetingMsg;

  public GreetingRunnable(String msg){
    this.greetingMsg = msg; }

  public void run(){
    for( int i=0; i<10; i++ ){
      System.out.println( greetingMsg );
    } } }
```

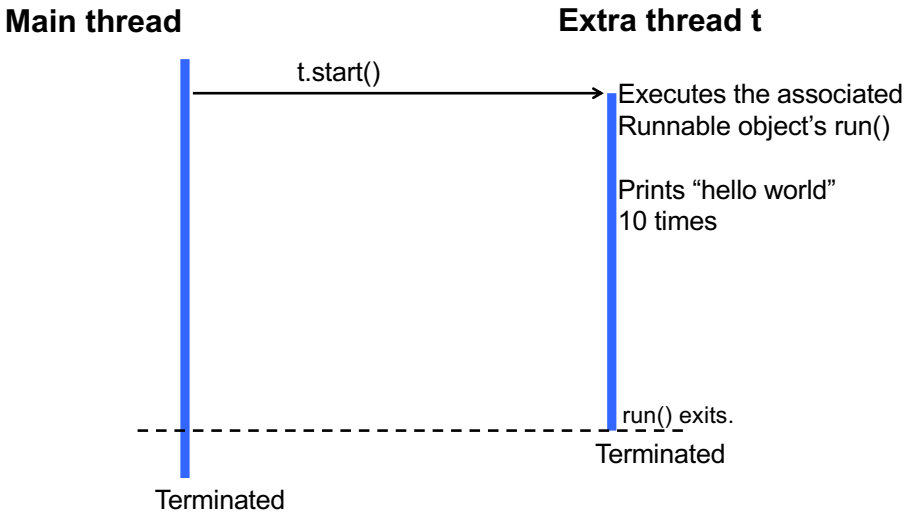
- HelloWorldTest.java

```
main(...){
  GreetingRunnable runnable = new GreetingRunnable("Hello World");
  Thread t = new Thread(runnable);
  t.start(); }
```


Thread.start()

- Creating a **`Thread`** object does not mean creating a new thread on the local JVM and in turn OS.
  - It is **`start()`** that actually creates a thread.
- **`start()`**
  - Allocates memory and initializes a new thread on the a JVM (and its underlying OS).
  - Calls **`run()`** of a specified **`Runnable`** object.
    - DO NOT call **`run()`** directly, but…
      let **`start()`** call **`run()`** on behalf of yourself.

Program Execution



- Output:
  -
```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```


An Example Code: Creating Threads

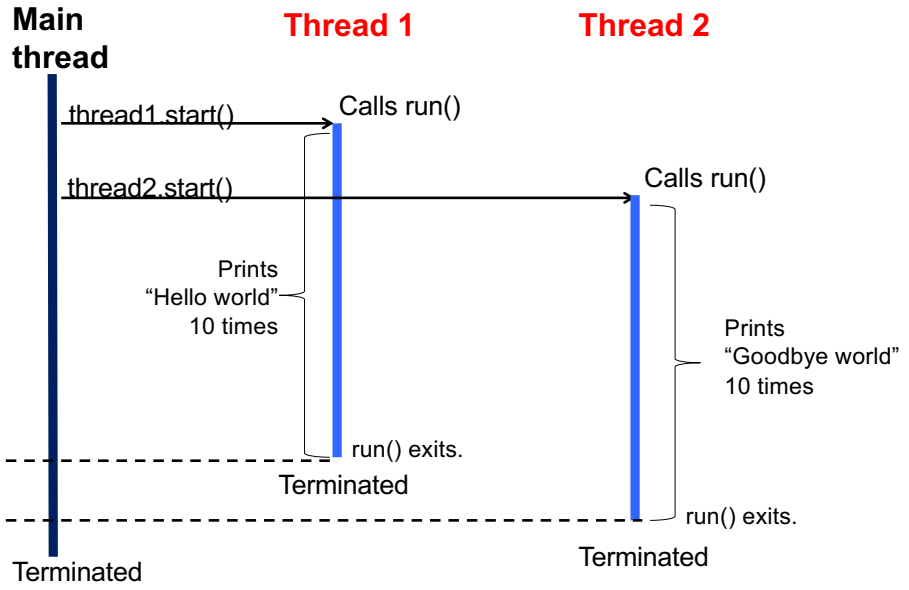
- HelloWorldTest2.java and GreetingRunnable.java

```
main(...){
  GreetingRunnable runnable1 = new GreetingRunnable("Hello World");
  GreetingRunnable runnable2 = new GreetingRunnable("Goodbye World");

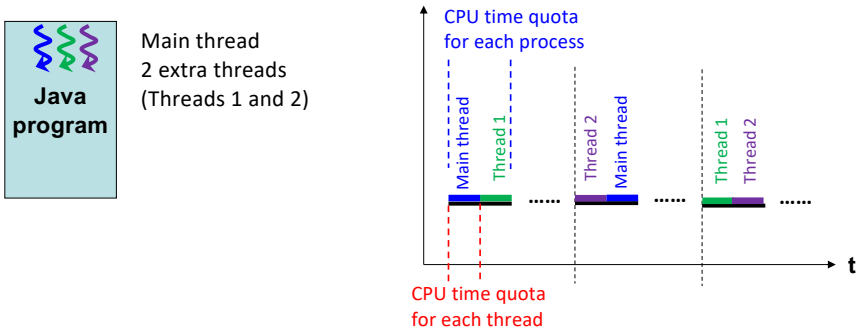
  Thread thread1 = new Thread(runnable1);
  Thread thread2 = new Thread(runnable2);

  thread1.start();
  thread2.start();
}
```


Expected Program Execution



Thread Scheduling



- Other processes (than the Java program) are ignored here.
- 1 per-process CPU time quota is assumed to be equal to 2 per-thread CPU time quotas here.
  - Actual ratio depends on the JVM and OS implementations.
    - May be 1:3, 1:1.5, etc.

A Possible Output

- Output:
  - Goodbye World
  - Hello World
  - Hello World
  - Goodbye World
  - Hello World
  - Goodbye World
  - Goodbye World
  - Hello World
  - Goodbye World
  - Hello World
  - Goodbye World
  - Hello World
  - Goodbye World
  - Hello World
  - Hello World
  - Goodbye World
  - Hello World
  - Goodbye World
  - Hello World
  - Hello World
  - Goodbye World
- Two message sets (Hello and Goodbye) are NOT exactly interleaved.

The Order of Thread Execution

- JVM's thread scheduler gives you NO guarantee about the order of thread execution.
- There are always slight variations in the time to run a concurrent task
  - especially when calling OS system calls (typically I/O related system calls)
- Expect that the order of thread execution is somewhat random.

Exercise

- Modify HelloWorldTest.java;
  - Replace the following highlighted lines
    - ```
      GreetingRunnable runnable1 = new GreetingRunnable("Hello");
      GreetingRunnable runnable2 = new GreetingRunnable("Bye");
      Thread thread1 = new Thread(runnable1);
      Thread thread2 = new Thread(runnable2);
      thread1.start();
      thread2.start();
      ```
  - with the following lines:
    - `runnable1.run();`
    - `runnable2.run();`
- What does the program output?
      ```

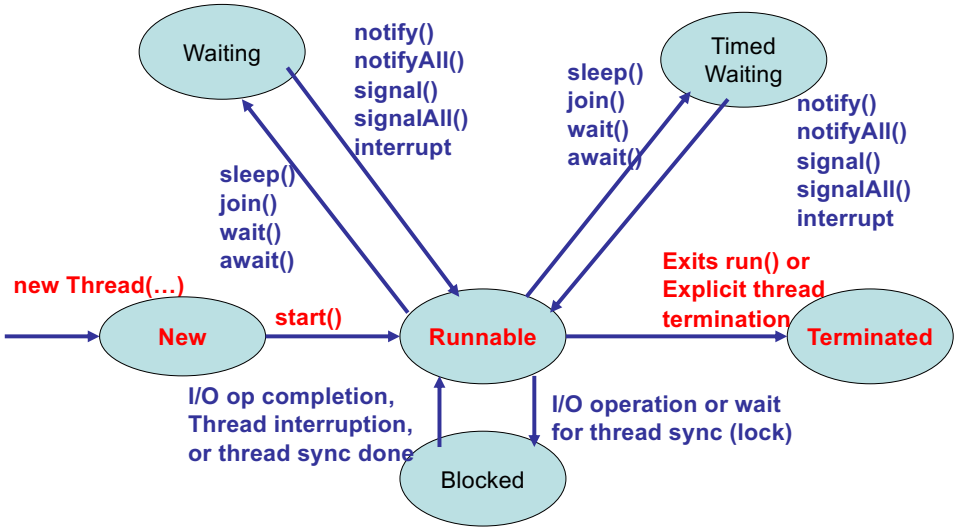
- Modify HelloWorldTest.java;
  - Replace the highlighted lines

```
GreetingRunnable runnable1 = new GreetingRunnable("Hello");
GreetingRunnable runnable2 = new GreetingRunnable("Bye");
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();
thread2.start();
```

  - with the following lines:
    - `runnable1.run();`
    - `runnable2.run();`

- Output: "Hello" in 10 lines, followed by "Bye" in 10 lines.
  - The program runs sequentially, NOT concurrently.

States of a Thread



- New
  - A Thread object is created. `start()` has NOT been called on the object yet.
- Runnable
  - `start()` has been called.
  - Java does not precisely distinguish ready-to-run and running. A running thread is in the Runnable state.
- Terminated (dead)
  - A thread automatically dies when run() returns.

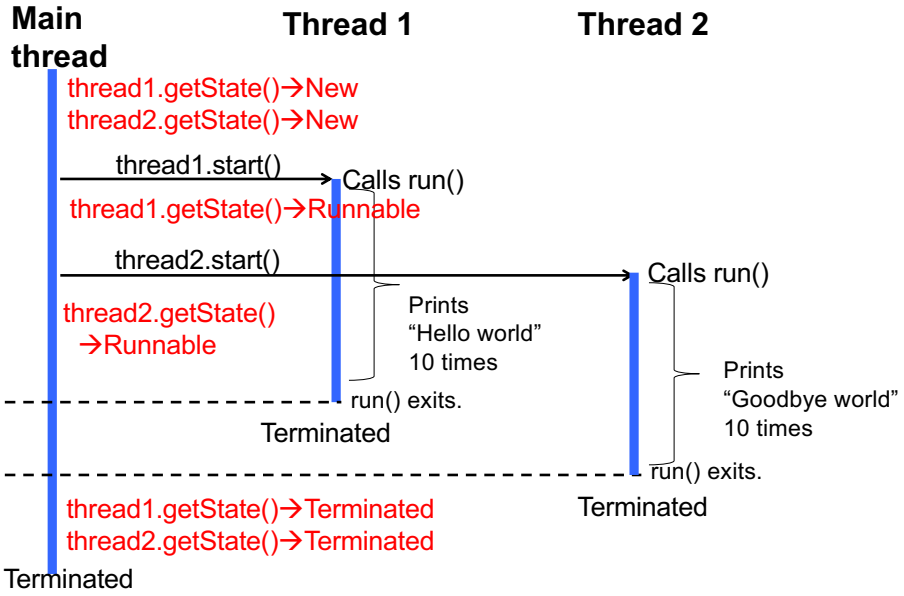

```
public class Thread{
    public enum State{
       NEW, RUNNABLE, BLOCKED, WAITING,
       TIMED_WAITING, TERMINATED }

    public Thread.State getState()

    public boolean isAlive()
```

- Alive
  - in the Runnable, Blocked, Waiting or Timed Waiting state.
    - NOT in New nor Terminated.
  - `isAlive()` can be used to check if a particular thread has been terminated.

Program Execution w/ HelloWorldTest3



Important Note

- Do NOT associate a single **`Runnable`** object with multiple **`Thread`** objects.
- Do (less error-prone):

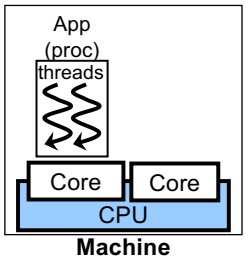
```
GreetingRunnable runnable1 = new GreetingRunnable(...);
GreetingRunnable runnable2 = new GreetingRunnable(...);
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable2);
thread1.start();
thread2.start();
```

- Don't:

```
GreetingRunnable runnable1 = new GreetingRunnable(...);
Thread thread1 = new Thread(runnable1);
Thread thread2 = new Thread(runnable1);
thread1.start();
thread2.start();
```

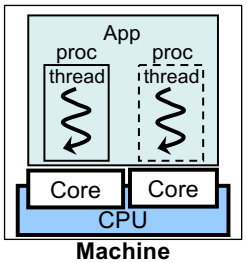

Modern Thread Distribution
(Thread Dispatching)

- Many modern OSes automatically distribute in-process threads onto different CPU cores in multi-core CPU environments.
- Concurrent programs often run in parallel (i.e., concurrent programs often means parallel programs)
  - as far as they run on modern OSes in multi-core CPU environments.



Multi-threaded (concurrent) app

Executed concurrently, not in parallel.



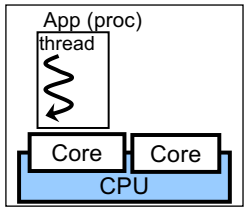
Multi-threaded (concurrent) app

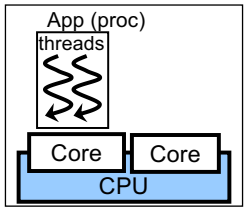
Executed in parallel

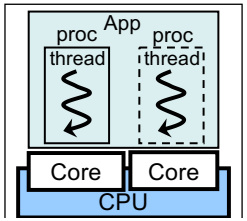
Run MCTest.java

- MCTest.java
  - **`run()`**: calculates 25*25 multiple times.
  - Can vary the number of threads
  - Calculated 25*25 10 billion times with Java 15 on Mac OS X
    - Intel Core i7 2.8 GHz (quad-core) and 16 GB RAM

# of threads	Time (sec)	
1	2.9	
2	2.9	(<< 2.9 * 2. Two threads run in parallel!)
4	3.0	(<< 2.9 * 4. Four threads run in parallel!)
8	6.1	(= 3.0 * 2, roughly)
16	12.3	(= 6.1 * 2, roughly)



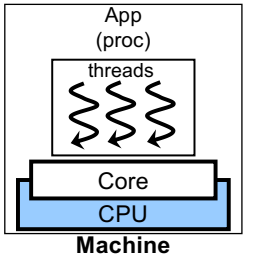


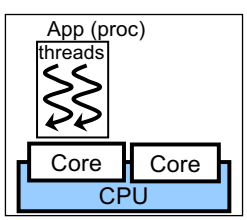


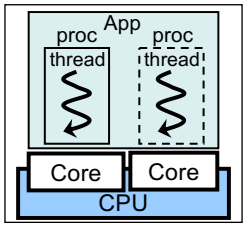
– Run `MCTest.java` with multiple threads on your machine.
  - e.g., `java edu.umb.cs681.basics.MCTest 1000000000 4`
    – First param: # of 25*25 multiplication
    – Second param: # of threads

In CS681…

- I will assume there is ONLY ONE CPU core in the underlying machine
  – when I explain/analyze the safety of multi-threaded programs.
- If a program is thread-safe in the single-core setting, it is always thread-safe as well in multi-core settings.







`Runnable` as a Functional Interface

- `java.lang.Runnable`: functional interface
  – Abstract method: `run()`
- `Thread`'s constructor
  – Takes a `Runnable` object
  – Can receive a lambda expression (LE)
    - Traditional

```
GreetingRunnable runnable =
                          new GreetingRunnable("Hello World");
Thread thread = new Thread(runnable);
thread.start();
```

    - LE-based

```
Thread thread = new Thread( ()->{
                    System.out.println("Goodbye World");} );
thread.start();
```

- It makes less/no sense to use a lambda expression to implement a concurrent task when the task is complex.
  – Lambda expressions are useful/powerful when their code blocks are reasonably short.

Sample Code: `PrimeGenerator`

- Generates prime numbers in between two input numbers (`from` and `to`)

```
Class PrimeGenerator {
  protected long from, to;
  protected LinkedList<Long> primes;

  public void generatePrimes(){ ... }
  public LinkedList<Long> getPrimes(){ return primes };
  protected boolean isPrime(long n){ ... };
```

- Client code (single-threaded)

```
PrimeGenerator gen = new PrimeGenerator(1L, 1000000L);
gen.generatePrimes();
gen.getPrimes().forEach( (prime)-> System.out.println(prime) );
                        //forEach() takes a LE typed with Consumer
```


Notes: `main()` in `PrimeGenerator`

- `PrimeGenerator`'s `main()` implements 2 types of single-threaded client code to generate primes.
- 1st client code
  - Uses `generatePrimes()` to generate primes
    - c.f. previous slide
    - `for(long n = from; n <= to; n++) { if( isPrime(n) ){ primes.add(n); } }`
- 2nd client code
  - Does NOT use `generatePrimes()`
  - Uses Stream API and `isPrime()`
    - `gen.primes = LongStream.rangeClosed(gen.from, gen.to) .filter((long n)->gen.isPrime(n)) ...}`

Sample Code: `RunnablePrimeGenerator`

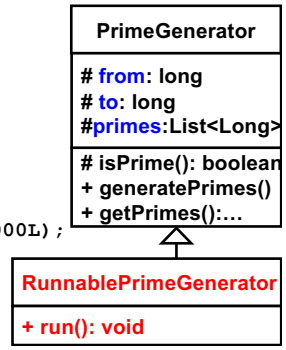
- A Runnable class that generates prime numbers in between two input numbers (`from` and `to`)

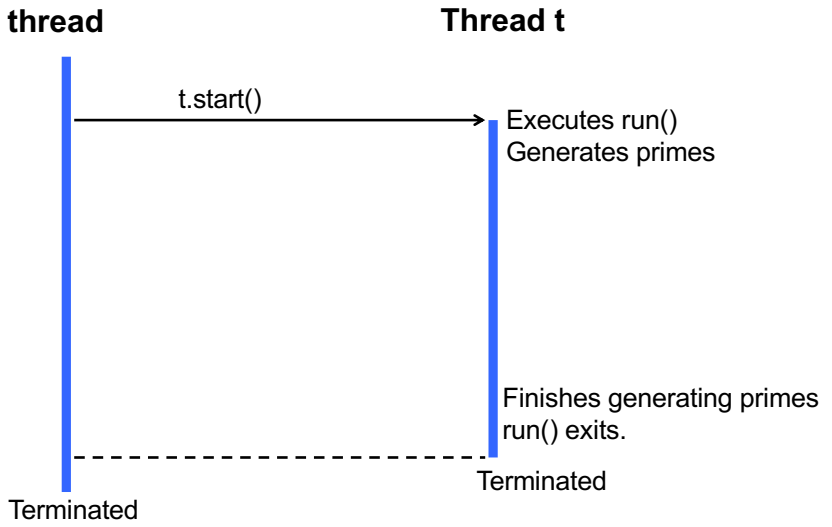
```
class RunnablePrimeGenerator extends PrimeGenerator
                             implements Runnable {
  public RunnablePrimeGenerator(long from, long to){
    super(from, to); }

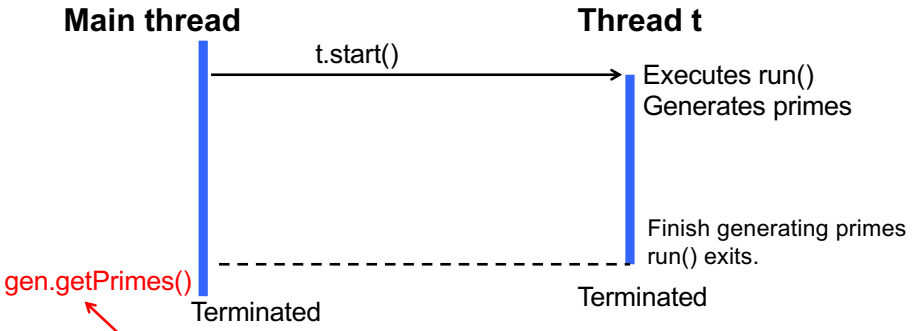
  public void run(){
    generatePrimes(); }
```

- Client code (multi-threaded)

```
RunnablePrimeGenerator gen =
          new RunnablePrimeGenerator(1L, 2000000L);
Thread t = new Thread(gen);
t.start();
```





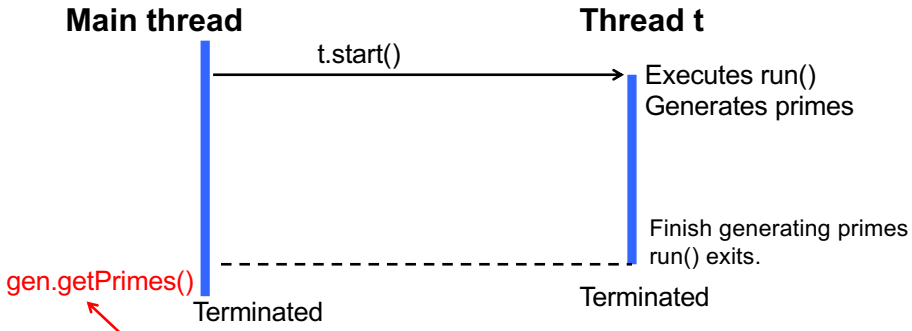


To get generated prime numbers, the main thread needs to call getPrimes() on a RunnablePrimeGenerator.

The main thread can call getPrimes() on the Runnable object ***even after t dies***.

Thread ***t*** dies when run() exits. However, it NEVER kill/delete the Runnable object.

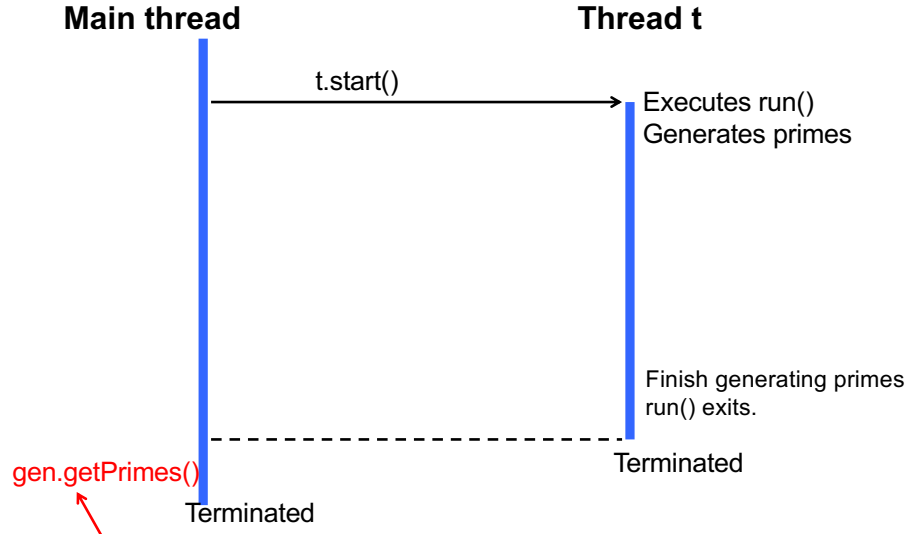
The Runnable object is still available ***even after t dies***.



To get generated prime numbers, the main thread needs to call getPrimes() on a RunnablePrimeGenerator.

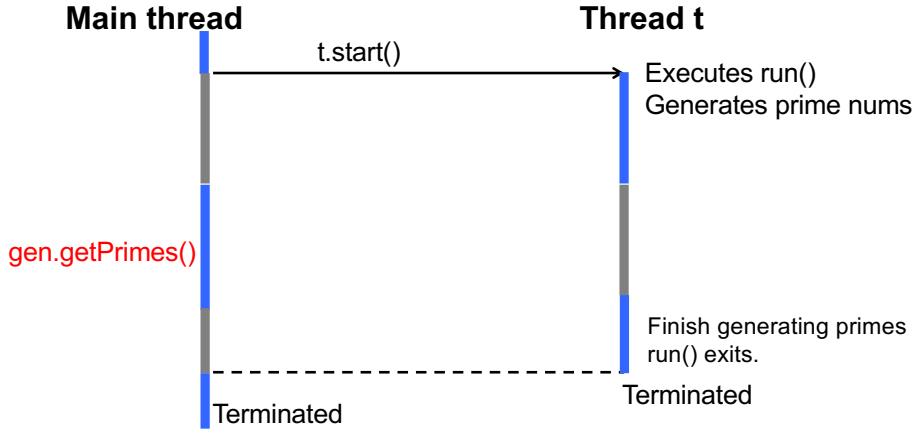
The main thread can call getPrimes() on the Runnable object ***even after t dies***.

- Client code
  - ```
    RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
    Thread t = new Thread(gen);
    t.start();
    gen.getPrimes();
    ```

To get generated prime numbers, the main thread needs to call getPrimes() on a RunnablePrimeGenerator.

How to guarantee to call getPrimes() after run() exits?



- Client code
  - ```
    RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
    Thread t = new Thread(gen);
    t.start();
    gen.getPrimes();
    ```

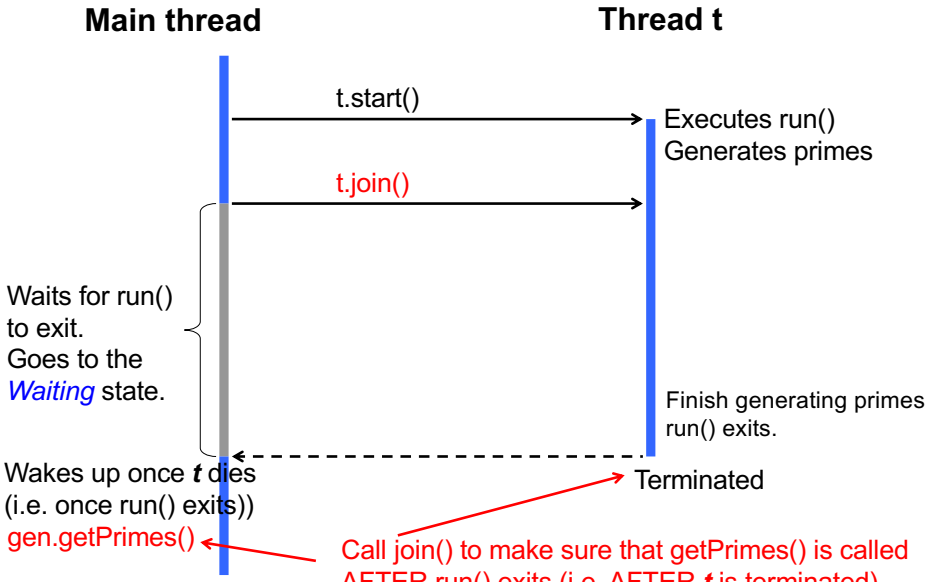
getPrimes() may be called before run() exits.

How to guarantee to call getPrimes() after run() exits?

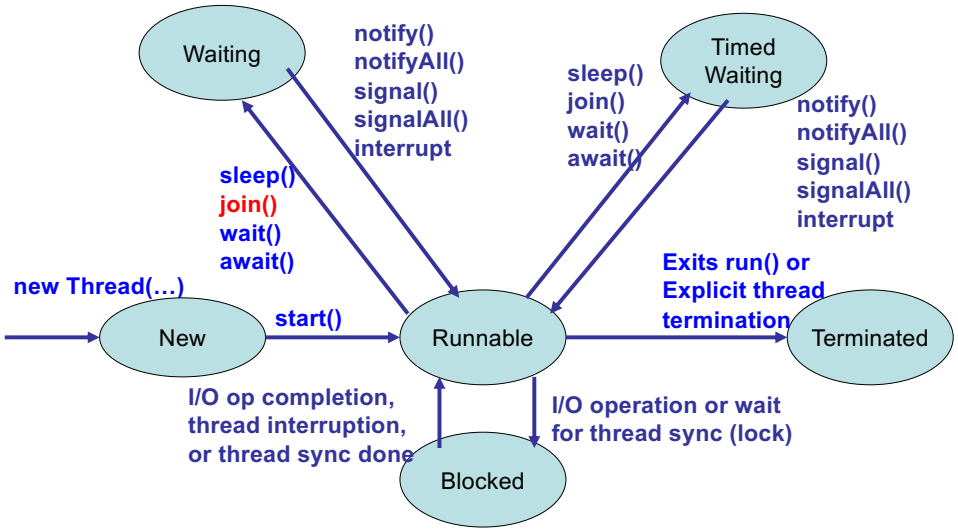
- Revised client code

```
RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);
Thread t = new Thread(gen);
t.start();
t.join();
gen.getPrimes().forEach(...);
```


Thread.join()



States of a Thread



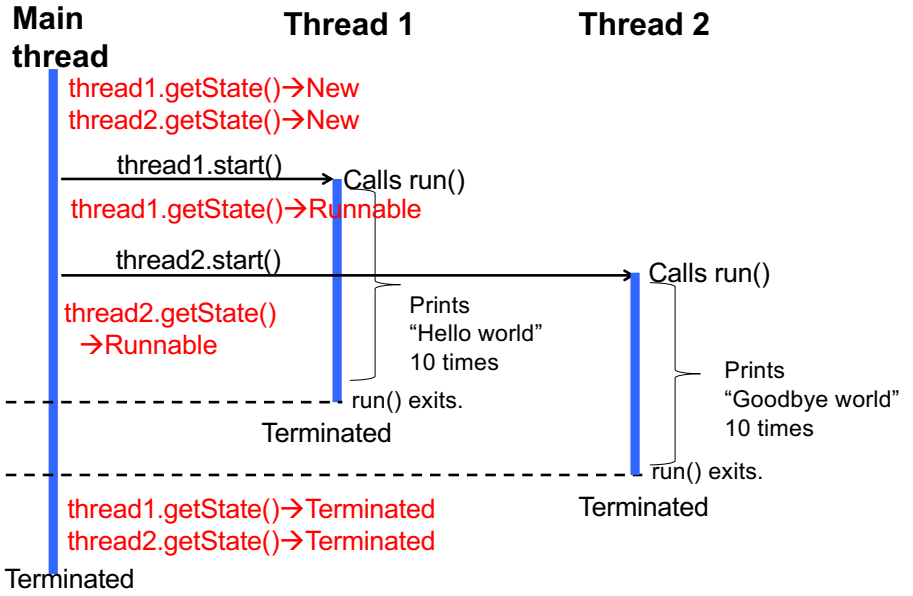
Thread.join()

- By default, there is no guarantee about the order of thread execution
- `join()` allows you to control the order of thread execution to some extent.

Exercise

- See how program behavior changes with and without join().

Note: join() is used in HelloWorldTest3

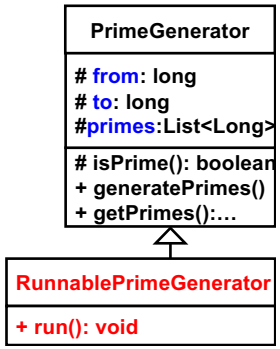


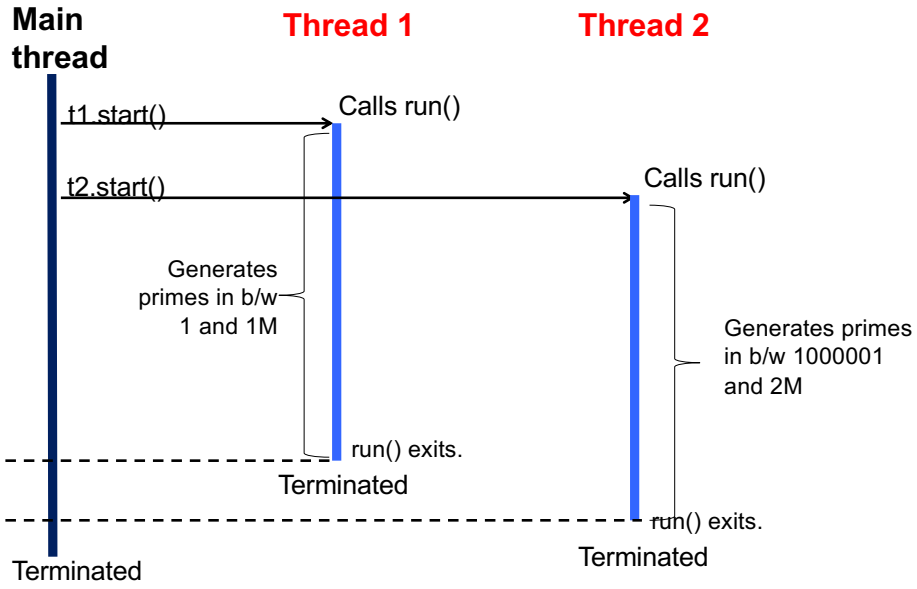
Sample Code: `RunnablePrimeGenerator`

- Client code (multi-threaded), which uses 2 threads

```
RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(
                                                    1L, 1000000L);
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(
                                                1000001L, 2000000L);

Thread t1 = new Thread(g1);
Thread t2 = new Thread(g2);
t1.start();
t2.start();
t1.join();
t2.join();
g1.getPrimes().forEach(...);
g2.getPrimes().forEach(...);
```





Exercise

- Run **`RunnablePrimeGenerator`** to generate primes in b/w 1 and 2,000,000.
  - With 1 thread

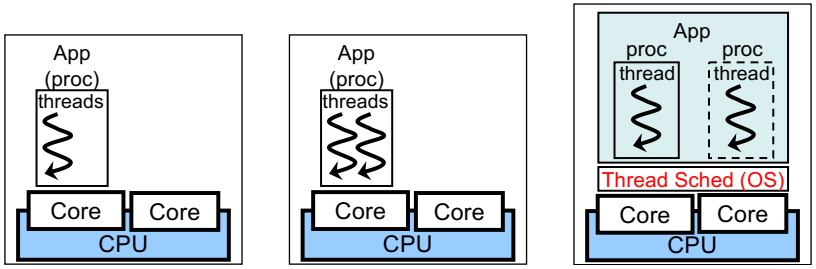
```
RunnablePrimeGenerator g = new RunnablePrimeGenerator(
                                                    1L, 2000000L);
Thread t = new Thread(g);
t.start();
t.join();
g1.getPrimes().forEach(...);
```

  - With 2 threads

```
RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(
                                                    1L, 1000000L);
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(
                                                 1000001L, 2000000L);
Thread t1 = new Thread(g1); // MAKE SURE TO CREATE 2 RUNNABLE
Thread t2 = new Thread(g2); // OBJECTS
t1.start(); t2.start();
t1.join(); t2.join();
g1.getPrimes().forEach(...);
g2.getPrimes().forEach(...);
```

  - With more threads
    - .....

- Measure the overhead of generating primes in b/w 1 and 2M to see if automatic thread distribution/dispatching works.
  - # of threads Time (sec)
  - 1 ???
  - 2 ??? (2 threads run in parallel?)
  - 4 ??? (4 threads run in parallel?)
  - 8 ???
  - 16 ???



HW 4: Data Processing with a Boston Housing Dataset

- Source: http://lib.stat.cmu.edu/datasets/boston
  - Will email you its CSV version.

- Census-based data that was made in 1978
  - Contains 14 data for each of 506 areas/blocks in and around Boston
    - Crime rate (CRIM)
    - Next to Charles river or not (CHAS)
    - # of rooms (RM)
    - House age (AGE)
    - Pupil-teacher ratio (PTRATIO)
    - Accessibility to highways (RAD)
    - TAX rate (TAX)
    - NOX concentration (NOX)
    - House price (MEDV), etc.

- Write a program to
  - Read/parse the CSV file and
  - Process the dataset in 4 different ways.

- Data processing #1:
  - Identify the areas/blocks next to Charles river.
  - Compute the highest, lowest and average price of those houses.

1	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
2	0.00632	18	2.31	0	0.538	6.575	65.2	4.09	1	296	15.3	396.9	4.98	24
3	0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.9	9.14	21.6

- Data processing #2
  - Identify the areas/blocks within the top (lowest) 10% of "low" crime rate and the top (lowest) 10% of pupil-teacher ratio.
  - Compute the max, min and average of:
    - Price
    - NOX concentration
    - # of rooms

- Data processing #3 and #4
  - Come up with your own data processing scenarios.

- Requirements
  - Use Stream API to parse a CSV file and perform data processing.
    - Read the entire dataset. (Use all the 506 data.)

Tips

- Parsing a CSV file to `List<List<String>>`
  - By stripping commas and double quotations.

```
Path path = Paths.get(...);
try( Stream<String> lines = Files.lines(path) ){
  List<List<String>> matrix =
      lines.map( line -> {
        return Stream.of( line.split(",") )
                     .map(value->value.substring(...))
                     .collect( Collectors.toList() ); })
           .collect( Collectors.toList() );
} catch (IOException ex) {}
```



```
// Stream to array conversion
//
String[] arr = stream.toArray(String[]::new);

// Array to stream conversion
//
Stream<String> stream = Arrays.stream(arr);

// Stream to list conversion
//
List<String> list = stream.collect(Collectors.toList());

// List to stream conversion
//
Stream<String> stream = list.stream();

// List to array conversion
//
String[] arr = list.toArray(new String[0]);

// Array to list conversion
//
List<String> list = Arrays.asList(arr);
```