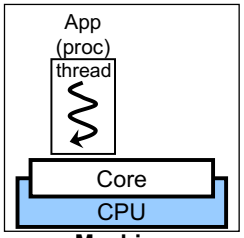
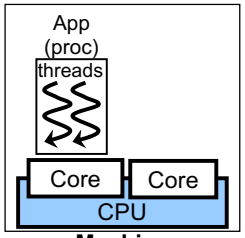
Terminology: Concurrent or Parallel?



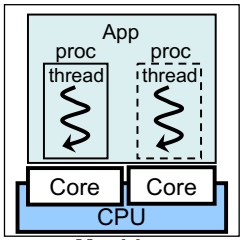
[1] Single-threaded (sequential) app

Not parallel



[2] Multi-threaded (concurrent) app

Not parallel



[3] Multi-threaded (concurrent) app

Parallel

The terms "concurrent" [2] and "parallel" [3] are not always equivalent.

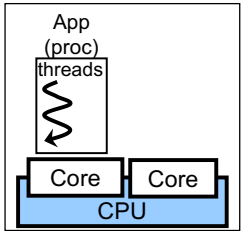
If you think [3] is a special case of [2], you may prefer the term "concurrent."

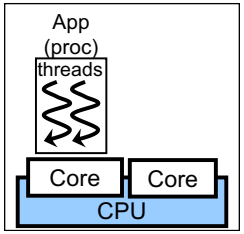
If you think [2] is a special case of [3], you may prefer the term "parallel."

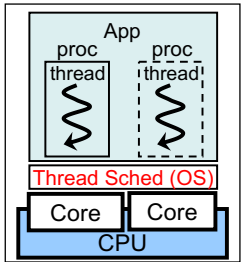
Run MCTest.java

- MCTest.java
  - run() calculates 25*25 10 billion times (on each thread)
  - With JDK 1.8 on Mac OS X (MacBook Pro)
    - Intel Core i7 2.8 GHz (dual core with hyper threading) and 8 GB RAM

# of threads	Time (sec)
1	10.55
2	10.55 (<< 10.55 * 2. Two threads run in parallel!)
4	17.35 (>10.55, but still << 10.55 * 4)
8	36.29
16	62.05







- Run MCTest.java with multiple threads
  - e.g., > java edu.umb.cs.threads.basics.MCTest 1000000000 4
    - First param: # of 25*25 multiplications
    - Second param: # of threads

Sample Code: `PrimeGenerator`

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

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

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

- Client code (single-threaded)

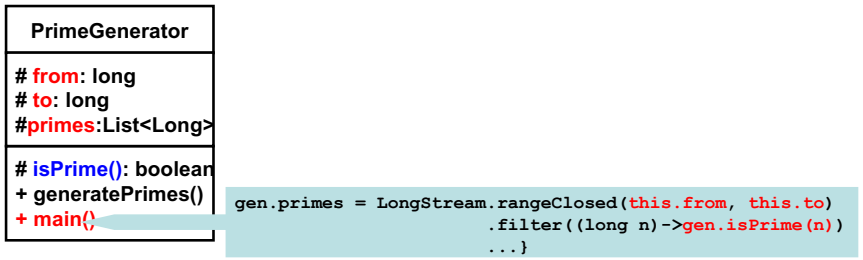
```
PrimeGenerator gen = new PrimeGenerator(1L, 1000000L);
gen.generatePrimes()
gen.getPrimes().forEach( (Long prime)-> System.out.print(prime));
```

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))
                             ...}
      ```


- `LongStream`
  - A stream of primitive `long` values
    - Specialization of Stream to `long`.
  - `range(long startInclusive, long endExclusive)`
    - Create a stream from `startInclusive` (inclusive) to `endExclusive` (exclusive) by an incremental step of 1.
  - `rangeClosed(long startInclusive, long endInclusive)`
    - Create a stream from `startInclusive` (inclusive) to `endInclusive` (inclusive) by an incremental step of 1.
- `DoubleStream`
- `IntStream`

- Note that a lambda expression can access data fields and methods in its *enclosing* class.
  - i.e., `from`, `to`, `primes` and `isPrime()` in `PrimeGenerator`.
  - Free variables: Data fields (variables) that a LE accesses in its enclosing class.

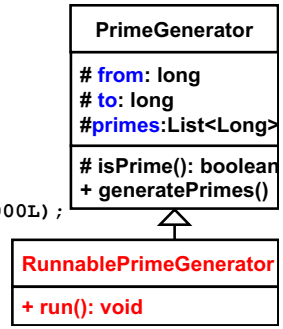


- The value of a free variables must be fixed (or immutable).
  - Once a value is assigned to the variable, no re-assignments (value changes) are allowed.
- Traditionally, immutable variables are defined with `final`; free variables can be defined with `final`.
- In fact, a LE can refer to variables that are not `final`, but they still have to be *effectively* `final`.
  - Even if they are not `final`, they need to be used as `final` if they are to be used in lambda expressions.

Sample Code: `RunnablePrimeGenerator`

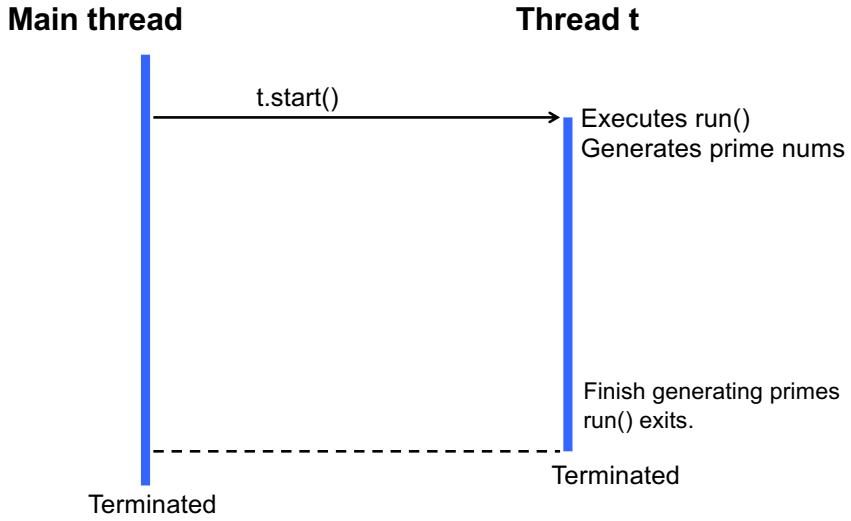
- A Runnable class that generates prime numbers in between two input numbers (`from` and `to`)

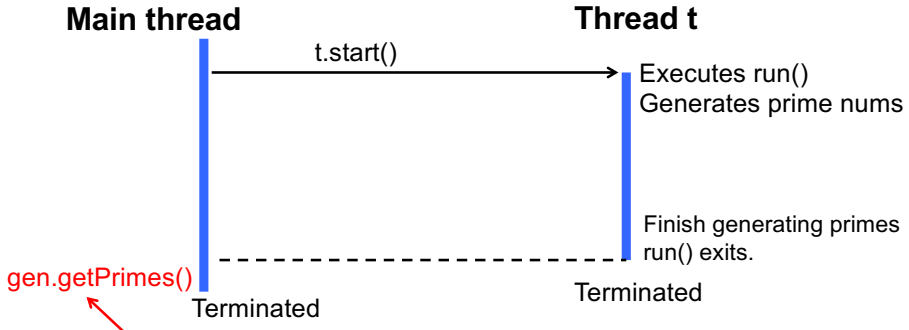
```
Class RunnablePrimeGenerator extends PrimeGenerator
                             implements Runnable {
  public RunnablePrimeGenerator(long from, long to){
    super(from, to); }

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



- Client code (multi-threaded)

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



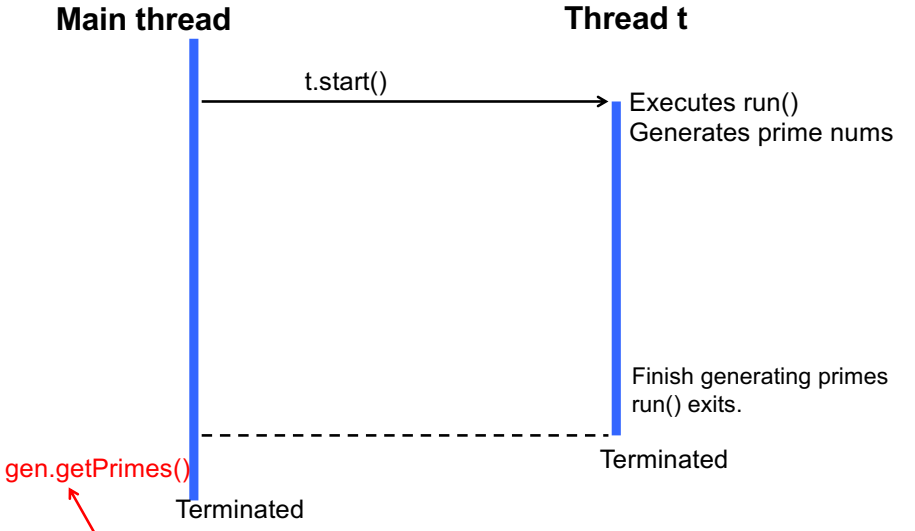


To get generated prime numbers, the main thread needs to call getPrimes() on a RunnablePrimeGenerator.

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

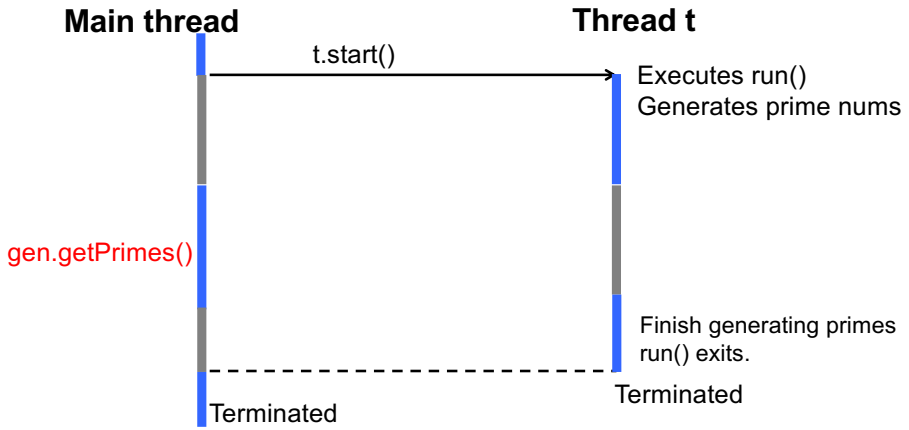
Thread ***t*** dies when run() exits. However, it NEVER kill/delete the Runnable object.

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.

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



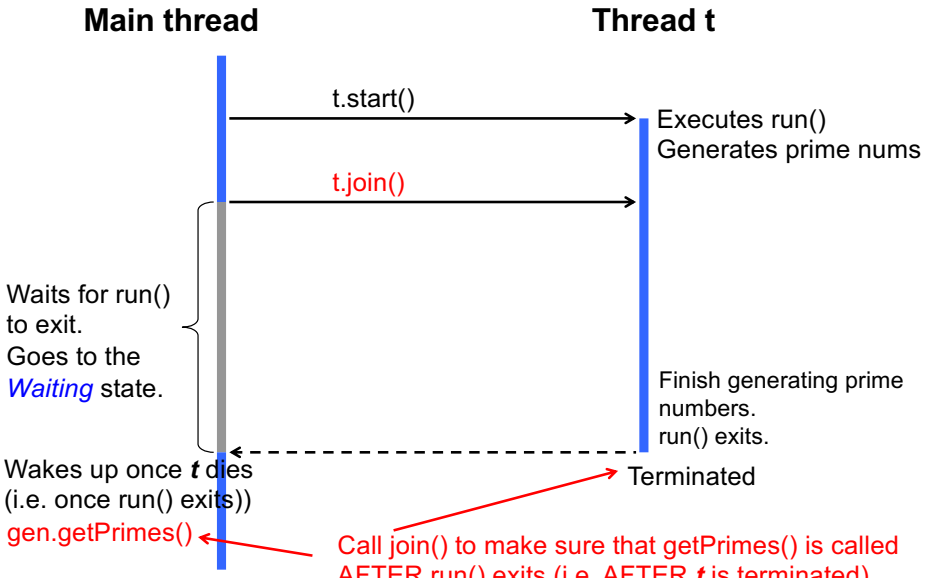
- Client code
  - `RunnablePrimeGenerator gen = new RunnablePrimeGenerator(...);`
    `Thread t = new Thread(gen);`
    `t.start();`
    `gen.getPrimes();`

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

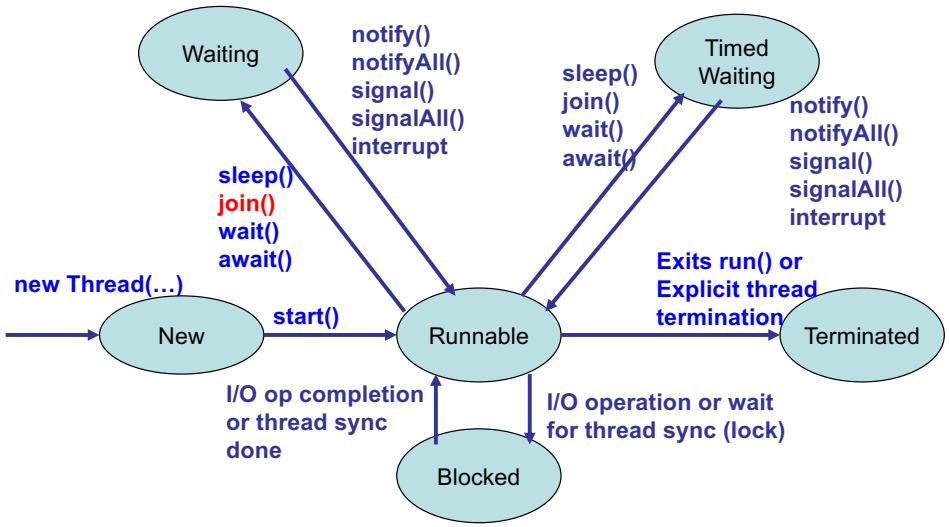
How to guarantee to call getPrimes() after run() exits?

- Revised client code (multi-threaded)
  - `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.

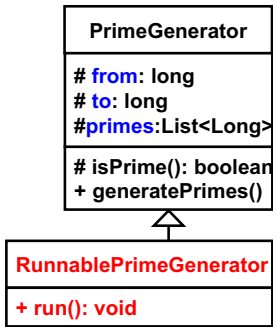
Exercise

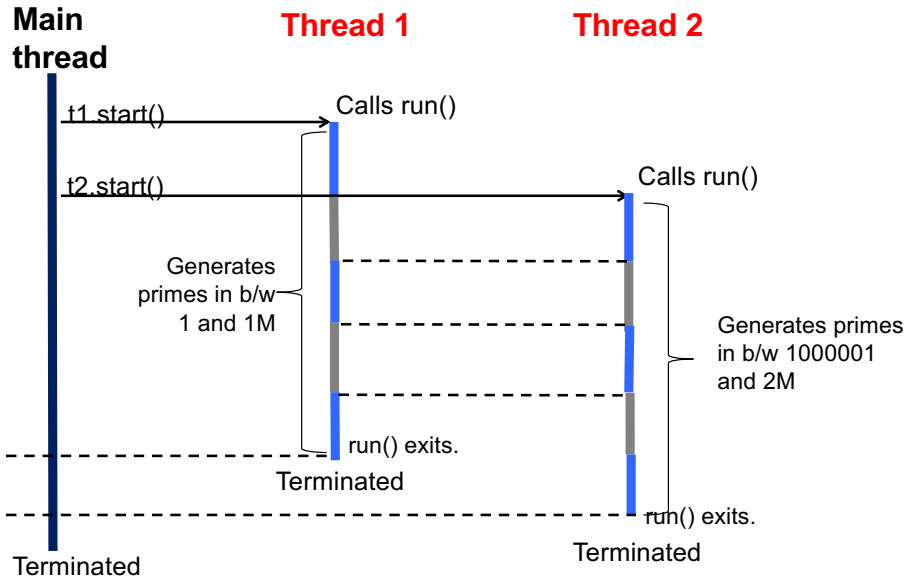
- See how program behavior changes with and without join().

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(...);
```





HW 4

- 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(...);
```

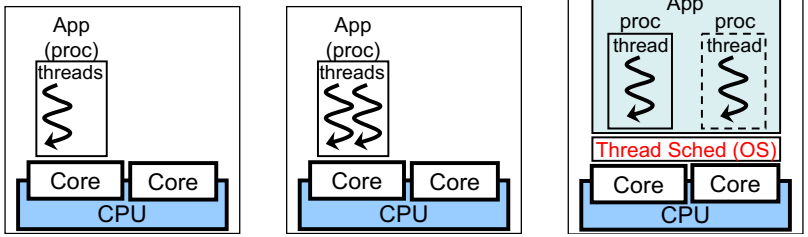
  - With 2 threads

```
RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(
                                                    1L, 1000000L);
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(
                                                 1000001L, 2000000L);
Thread t1 = new Thread(g1);
Thread t2 = new Thread(g2);
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 how threads run.
  - # of threads Time (sec)
  - 1 ???
  - 2 ??? (Two threads run in parallel?)
  - 4 ???
  - 8 ???
  - 16 ???

- Deadline: March 12 (Tue) midnight



`Runnable` as a Functional Interface

- `java.lang.Runnable`: functional interface
  - The only 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();
```

Note that…

- It makes less/no sense to use a lambda expression when the code block is long and complex.
  - Lambda expressions are useful/powerful when their code blocks are reasonably short.

HW 5

- Modify MCTest.java to use a lambda expression.
  - MCTest.java currently uses an anonymous Runnable class to implement run().

```
Thread t = new Thread(
        new Runnable(){
          public void run(){
            int n = 25;
            for (long j = 0; j < nTimes; j++){
              n *= 25;
            }
          }
        });
```

  - Replace the anonymous class with a lambda expression.

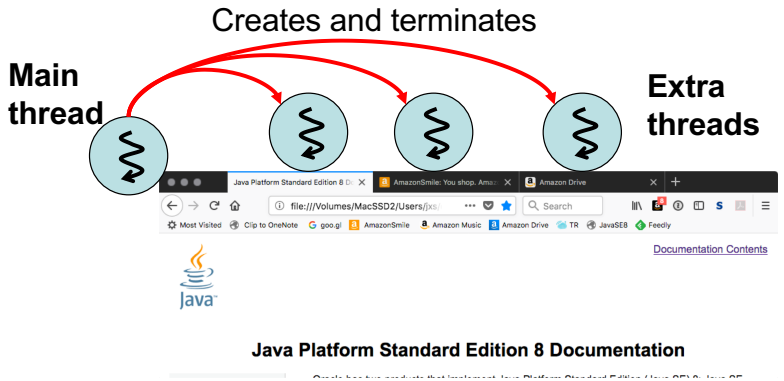
- Deadline: March 12 (Tue) midnight

Thread Termination

Thread Termination

- ***Implicit*** termination
  - A thread automatically triggers its own death when run() returns.
    - Once a thread starts executing run(), it continues execution until run() returns.
- ***Explicit*** termination
  - A thread explicitly terminates another thread.
    - when a certain condition is satisfied.
    - even if an on-going concurrent task is not completed.
  - Two ways
    - With a flag
    - With thread interruption

An Example of Explicit Termination



Explicit Thread Termination with a Flag

- Define a flag in a Runnable class.

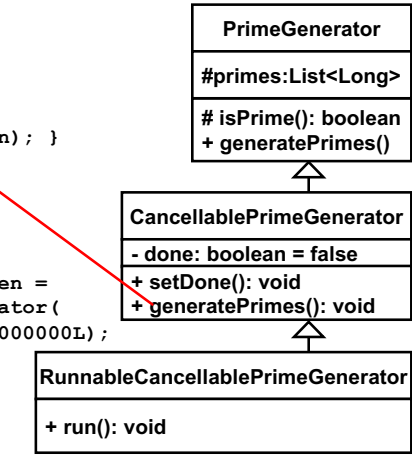
```
public class MyRunnbable implements Runnable{
    private boolean done = false;
    ...
    public void run(){
        while(!done){
            ... // Concurrent task is written here.
        }
    }
    public setDone(){ done=true; }
}
```

- Have a soon-to-be-killed thread periodically check the flag to determine if it should stop/die.
  - Let run() return once "done==true" is detected.
- Stop the thread by flipping the flag to inform that the thread should die.

Sample Code: CancellablePrimeGenerator

- Define and use a flag to stop generating prime numbers.

```
for (long n = from; n <= to; n++){
  if(done){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
}
```

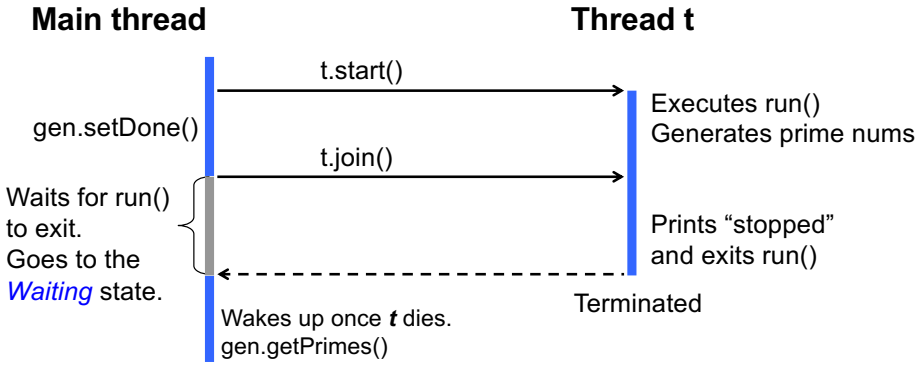


- Client code

```
RunnableCancellablePrimeGenerator gen =
  new RunnableCancellablePrimeGenerator(
                                    1L,1000000L);
Thread t = new Thread(gen);
t.start();
gen.setDone();
t.join();
gen.getPrimes().forEach(...);
```


- Alternatively...

```
long n = from;
while (!done && n <= to){
  if( isPrime(n) ){ this.primes.add(n); }
  n++;
}
System.out.println("Stopped generating prime numbers.");
this.primes.clear();
```

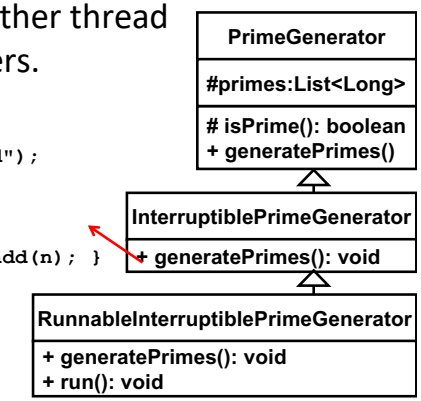

```
RunnableCancellablePrimeGenerator gen =
       new RunnableCancellablePrimeGenerator(1L,1000000L);
Thread t= new Thread(gen);
t.start();
gen.setDone();
t.join();
gen.getPrimes().forEach(...);
```

Explicit Thread Termination via Interruption

- **`Thread.interrupt()`**
  - Used to interrupt another thread.
    - `Thread thread = new Thread(aRunnable);`
      `thread.start();`
      `thread.interrupt();`
  - Let that thread know that it is notified via interruption.
- Have a soon-to-be-killed thread periodically detect an interruption to determine if it should stop/die.
  - Let run() return once an interruption is detected.
  - ```
    public class MyRunnbable implements Runnable{
        ...
        public void run(){
            while(!Thread.interrupted()){
                ...
            }
        }
    }
    ```

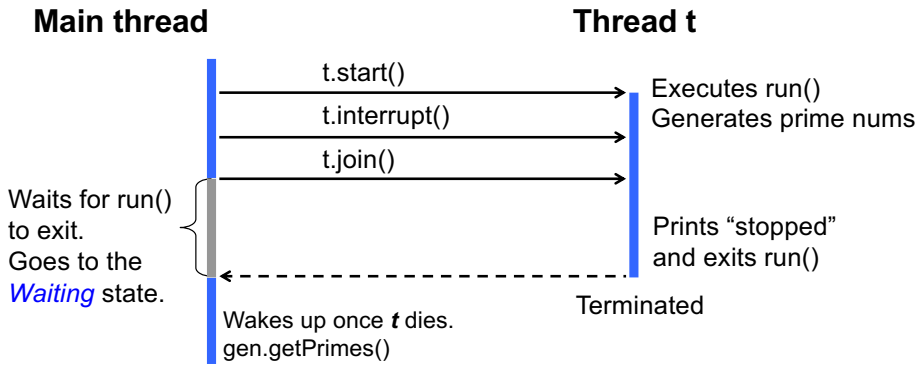

Sample Code: `InterruptiblePrimeGenerator`

- Detect an interruption from another thread to stop generating prime numbers.
  - ```
    for (long n = from; n <= to; n++){
        if(Thread.interrupted()){
            System.out.println("Stopped");
            this.primes.clear();
            break;
        }
        if( isPrime(n) ){ this.primes.add(n); }
    }
    ```



- Client code
  - ```
    RunnableInterruptiblePrimeGenerator gen =
        new InterruptiblePrimeNumberGenerator(1L, 1000000L);
    Thread t = new Thread(gen); t.start();
    t.interrupt();
    t.join();
    gen.getPrimes().forEach(...);
    ```


- Alternatively...
  - ```
    long n = from;
    while(!Thread.interrupted() && n <= to){
      if( isPrime(n) ){ this.primes.add(n); }
      n++;
    }
    System.out.println("Stopped generating prime numbers.");
    this.primes.clear();
    ```

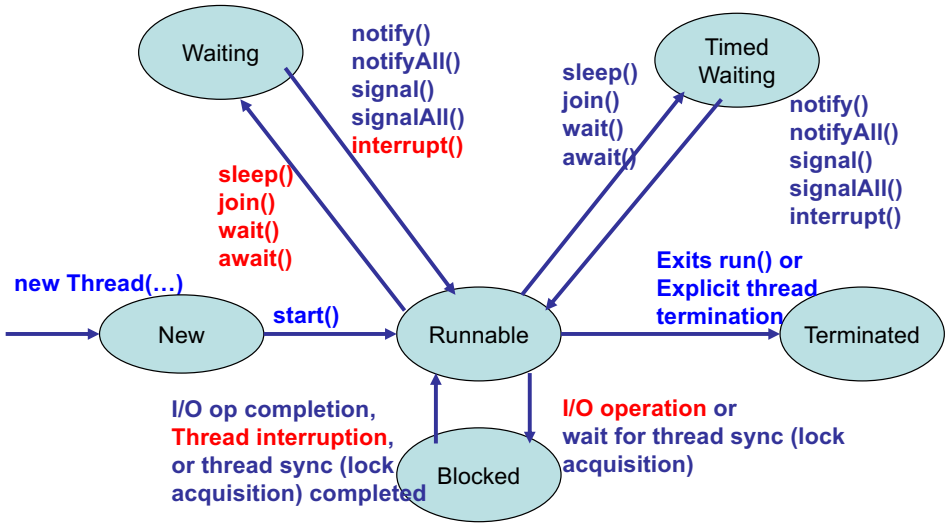




```
InterruptiblePrimeNumberGenerator gen =
      new InterruptiblePrimeNumberGenerator(1L, 1000000L);
Thread t= new Thread(gen);
t.start();
t.interrupt();
t.join();
gen.getPrimes().forEach(...);
```

Which Termination Scheme to Use?

- Flag-based or interruption-based?
- Both work perfectly well if run() is simple.
  - Both cancellable and interruptible versions of prime number generators work well.
- Favor interruption-based scheme if a soon-to-be-killed thread can be in the Waiting or Blocked state.
  - `Thread.sleep()`
  - `Thread.join()`
  - I/O operations
  - These methods can be long-running and interrupted.

States of a Thread



If `Thread.sleep()` is called in `run()`...

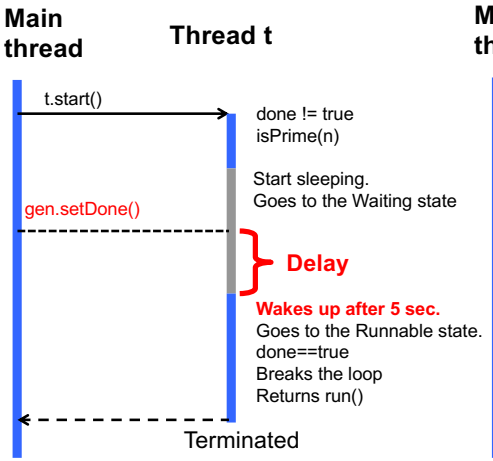
- `RunnableCancellablePrimeGenerator`'s `run()`

```
for (long n = from; n <= to; n++){
  if(done){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
  Thread.sleep(5000); }
```

- `RunnableInterruptiblePrimeGenerator`'s `run()`

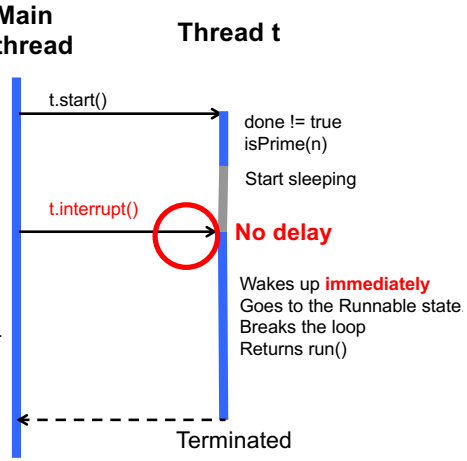
```
for (long n = from; n <= to; n++){
  if(Thread.interrupted()){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
  if( isPrime(n) ){ this.primes.add(n); }
  Thread.sleep(5000); }
```


RunnableCancellablePrimeGenerator



Sleep period: 5 seconds
Less responsive thread termination

RunnableInterruptiblePrimeGenerator

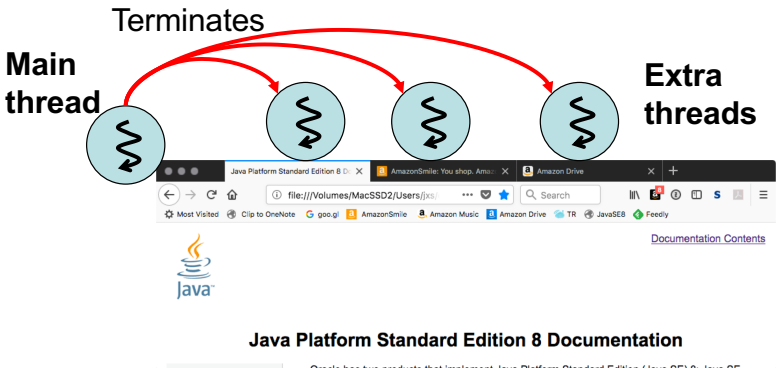


Sleep period: Can be less than 5 seconds
More responsive thread termination

If an I/O operation is performed in `run()`...

```
run(){
  for(element: pageElements){
    if(done){ return; }
    download(element);
}
```

```
run(){
  for(element: pageElement){
    if(Thread.interrupted()){ return; }
    download(element);
}
```



Thread Termination Requires Your Attention

- Thread creation is a no brainer.
- Thread termination requires your attention.
  - No methods available in `Thread` to directly terminate threads like `terminate()`.
    - Use:
      - Flag-based OR interruption-based scheme

Deprecated Methods for Thread Termination

- `Thread.stop()` and `Thread.suspend()`
  - Not thread-safe. **Never use them.**
  - c.f. "Why Are Thread.stop, Thread.suspend,Thread.resume and Runtime.runFinalizersOnExit Deprecated?"
    - http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html