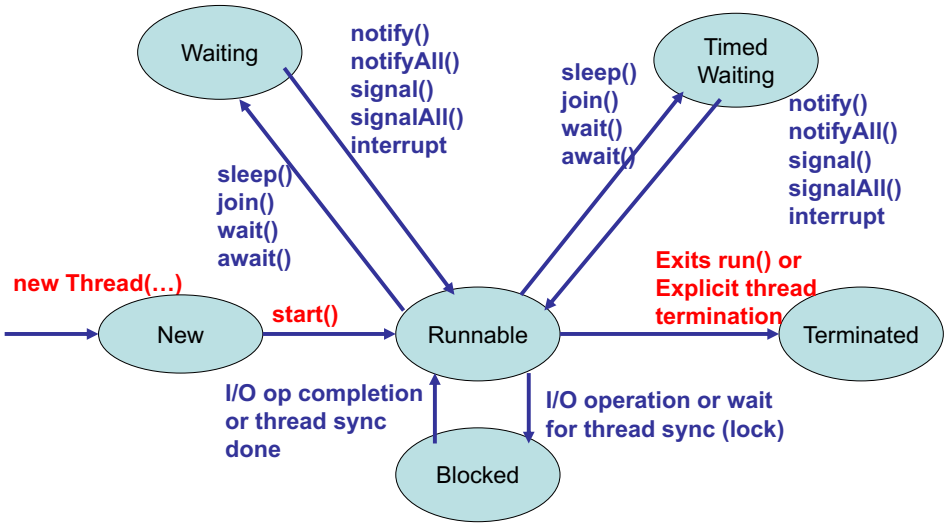
# States of a Thread






- New
  - A Thread object is created. start() has not been called on the object yet.
- Runnable
  - Java does not distinguish ready-to-run and running. A running thread is still 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 }

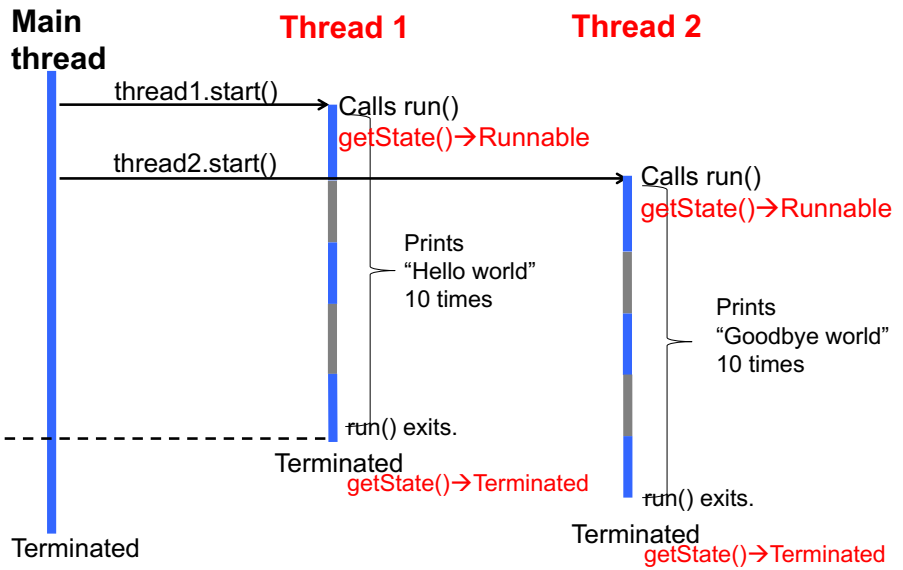
    public Thread.State getState()

    public boolean isAlive()
  ```

- Alive
  - in the Runnable, Blocked, Waiting or Timed Waiting state.
  - isAlive() is usually used to poll/check a thread to see if it is terminated.



# Program Execution w/ HelloWorldTest2


  ```

# Sample Code: PrimeGenerator

- A class that generates all prime numbers in between two input numbers.

```
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));
```



# Sample Code: RunnablePrimeGenerator

- A Runnable class that generates all prime numbers in between two input numbers.

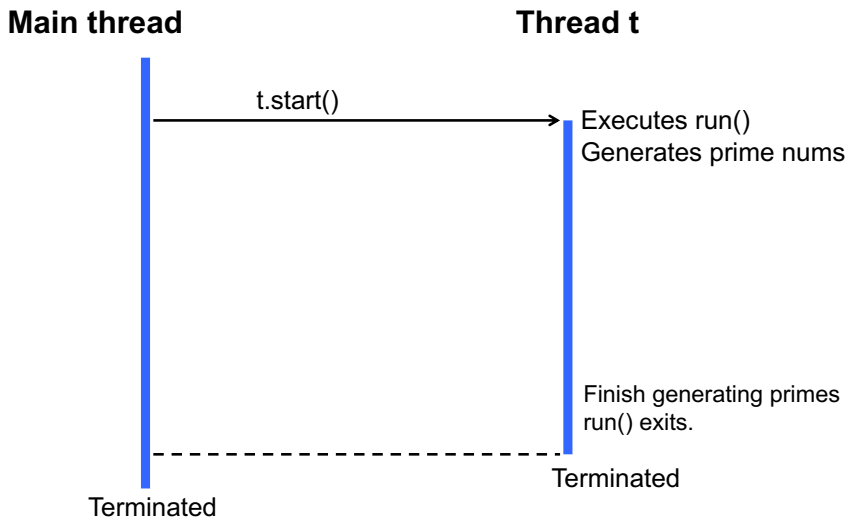
```
Class RunnablePrimeGenerator extends PrimeGenerator
                             implements Runnable {
  public RunnablePrimeGenerator(long from, long to){
    super(from, to); }

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

- Client code (multi-threaded)

```
RunnablePrimeGenerator gen = new RunnablePrimeGenerator(1L, 100L);
Thread t = new Thread(gen);
t.start();
```








# Sample Code: RunnablePrimeGenerator

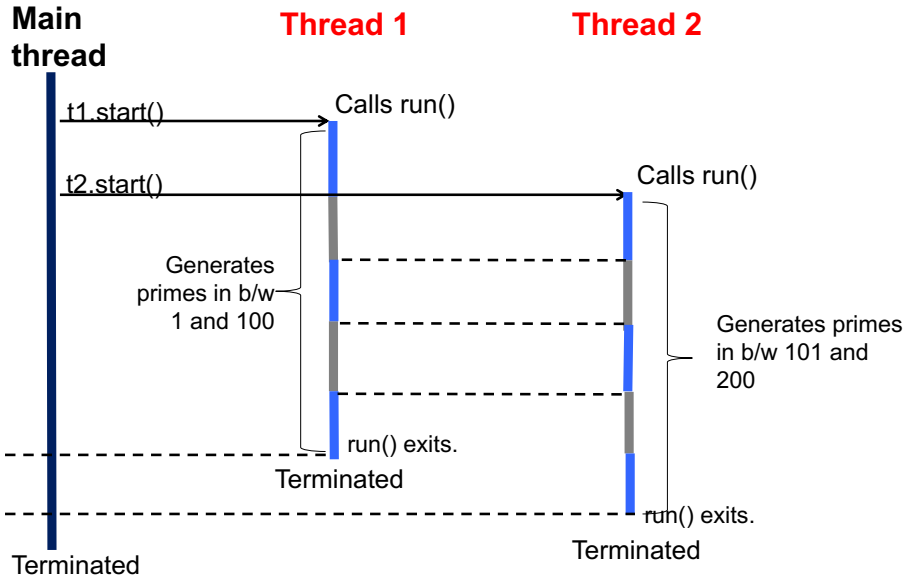
- A Runnable class that generates all prime numbers in between two input numbers.

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

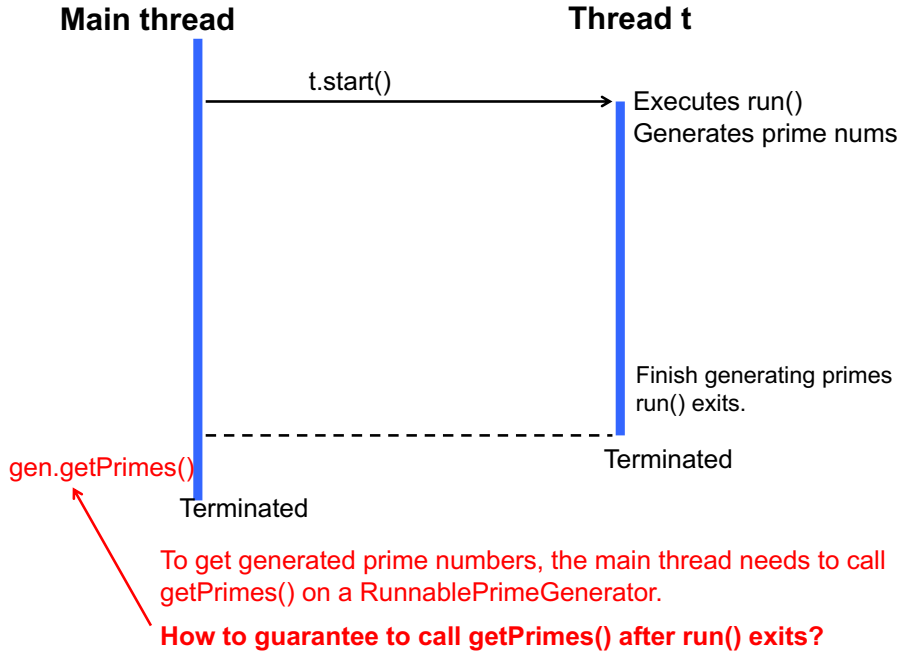
  public void run(){
    generatePrimes(); }
```

- Client code (multi-threaded)

```
RunnablePrimeGenerator g1 = new RunnablePrimeGenerator(1L, 100L);
RunnablePrimeGenerator g2 = new RunnablePrimeGenerator(101L, 200L);
Thread t1 = new Thread(g1);
Thread t2 = new Thread(g2);
t1.start();
t2.start();
```






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

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



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

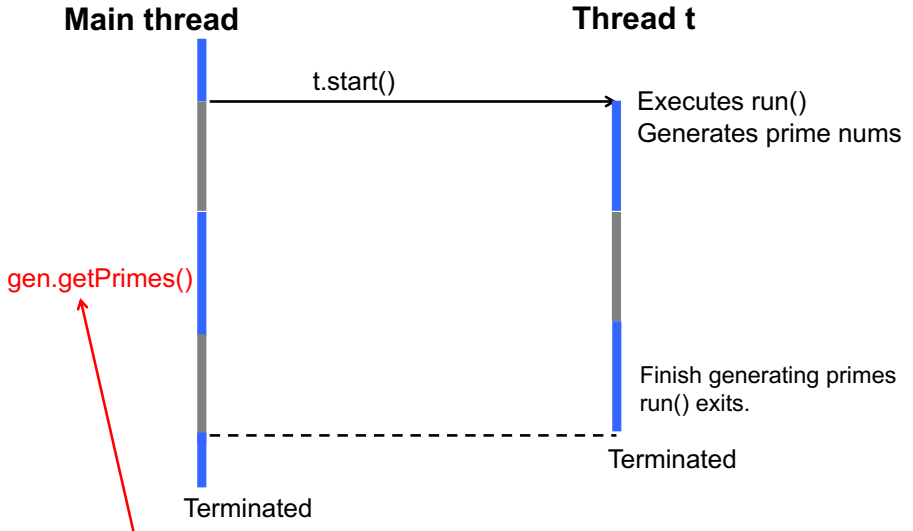
  public void run(){
    generatePrimes(); }
```

- Client code

```
RunnablePrimeGenerator gen = new RunnablePrimeGenerator(1L, 100L);
Thread t = new Thread(gen);
t.start();
gen.getPrimes();
```

  - getPrimes() may be invoked BEFORE run() exits.






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

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

# Sample Code: RunnablePrimeGenerator

- A Runnable class that generates all prime numbers in between two input numbers.

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

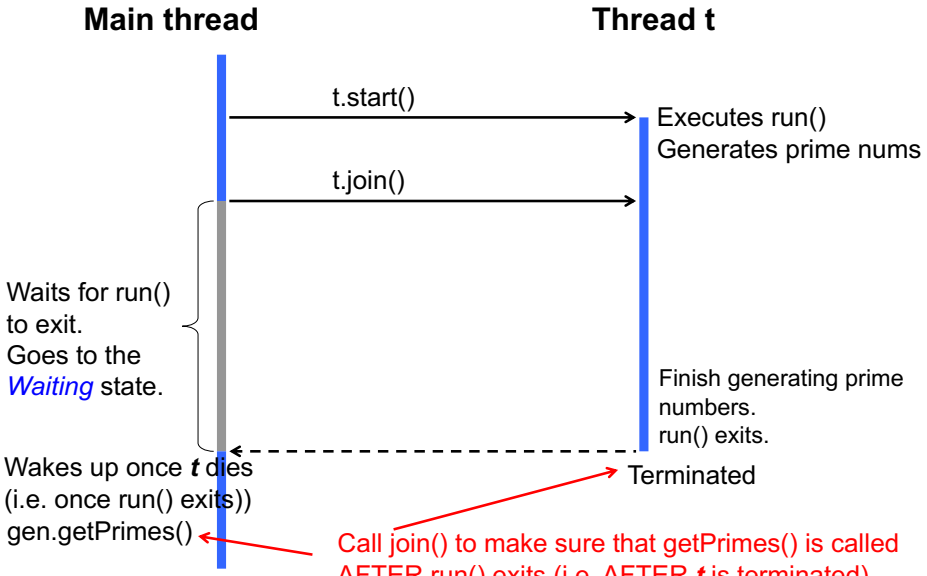
  public void run(){
    generatePrimes(); }
```

- Client code (multi-threaded)

```
RunnablePrimeGenerator gen = new RunnablePrimeGenerator(1L, 100L);
Thread t = new Thread(gen);
t.start();
t.join();
gen.getPrimes().forEach(...);
```

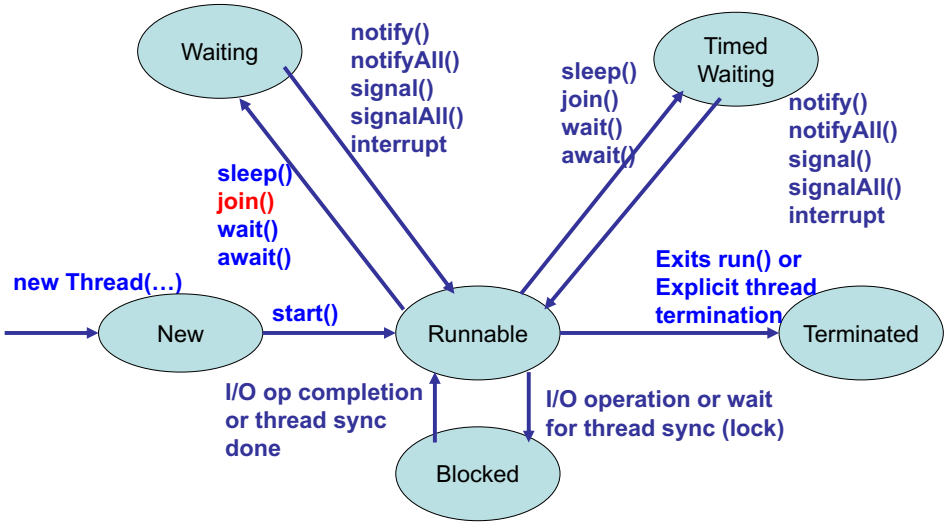


# Thread.join()






# 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().



# Some Context to Prime Number Generation

- Prime factorization is a mathematical key part in RSA encryption algorithm (used in SSL, SSH, etc.)
  - SSL is used to access https://… web sites.
    - https://www.google.com
  - Easy to calculate a product of two prime integers, but hard (time consuming) to factorize a big integer
    - 3 * 11 = 33, 71 * 97 = 6,887, etc.
  - Each public key contains a product of big prime numbers
    - Use openssl to look into a public key (e.g. Google's)
      - A big (e.g. 2,048-bit) prime number is in it
  - Need to factorize the product to break RSA.



  - Prime number in Google's public key (in hex)
    - » 9FA1E1B43B3A570ED0CF54BCCD18D8B2121331A44C373D093EEF
      73DD6423618E951FE46C8D4052626EDE0E82BF4C2ACF86FD413E
      81757484F9603150CFF293899FD4786426D6D2C2E71B01002D82
      AD220B5BBA9830D71F6B25FCD501E152921ABC8861875154776E
      6651640079B1C1C9B1C90B7A050CA45E5EC63647ED88966D55C8
      BF6513DA06B1679198D909B247F9C6A9C74BFD8660532CF5401B
      2205E53C05D5A95D5D3DFAED2EFA4061A7E949C8D0EE42B9AEC
      65352435666CFBACD248114DACEFC96E20D7B8C616D3494F2E37
      52A957ED367C07BE8E642B2AA15C496E5561EC8D160DC0C5C08A
      D25A250415CF62D39835838F712BC63BB6987CB5BC2FF



# HW 4

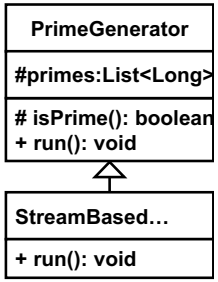
- Generate prime numbers with a stream.
  - Define `StreamBasedRunnablePrimeGenerator` as a subclass of `PrimeGenerator`.
  - Implement run() with Stream API.
    - c.f. PrimeGenerator's run():

```
public void run(){
   for (long n = from; n <= to; n++) {
          if( isPrime(n) ){ this.primes.add(n); }  }}
```

    - StreamBasedRunnablePrimeGenerator's run():

```
public void run(){
   this.primes = LongStream.rangeClosed(this.from, this.to)
                 .filter( ... )
                 ...}
```

    - c.f. `boxed()` and `collect()`



- **`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`**



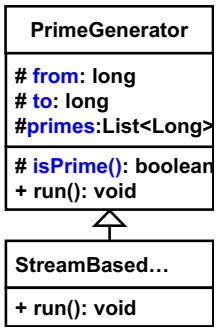
- Deadline: Oct 16 (Tue) midnight



# Free Variables

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

```
public void run(){
  this.primes = LongStream.rangeClosed(
                              this.from, this.to)
                .filter( ... )
                ...}
```




- 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.