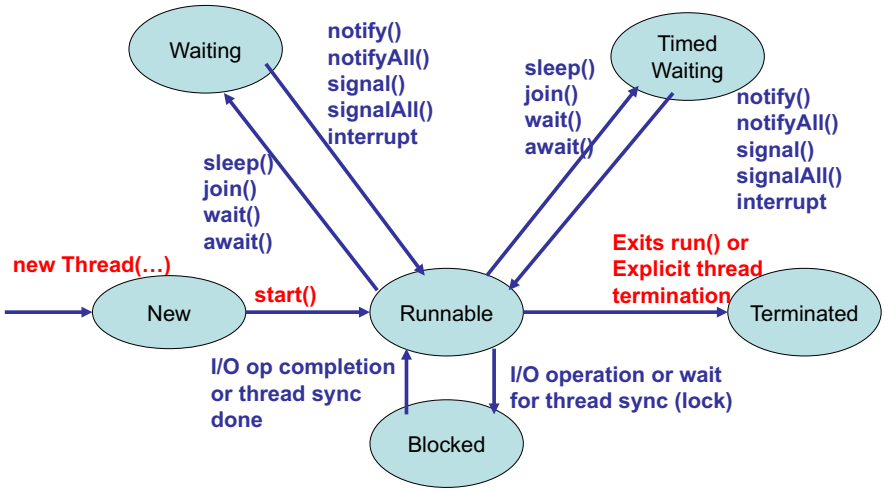
States of a Thread

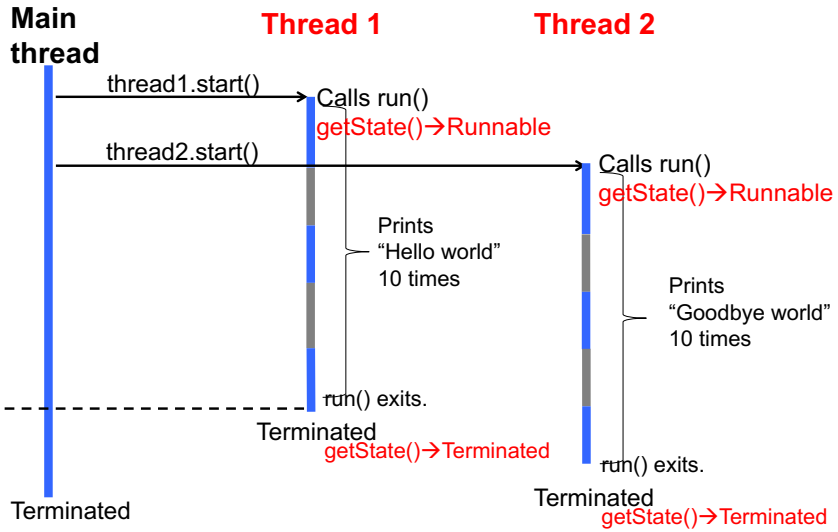


- New
  - A Thread object is created. start() has not been called on the object yet.
- Runnable
  - Java does not distinguish ready-to-run and running. A running thread is still in the Runnable state.
- Terminated/dead
  - A thread automatically dies when run() returns.


```
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: PrimeNumberGenerator

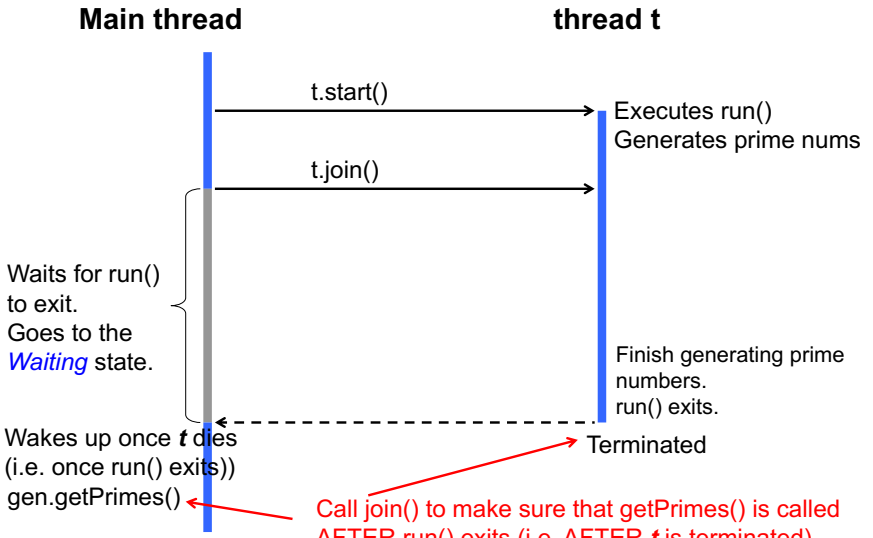
- A Runnable class that generates all prime numbers in between two input numbers.

```
Class PrimeNumberGenerator implements Runnable{
  protected long from, to;
  protected List<Long> primes;

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

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

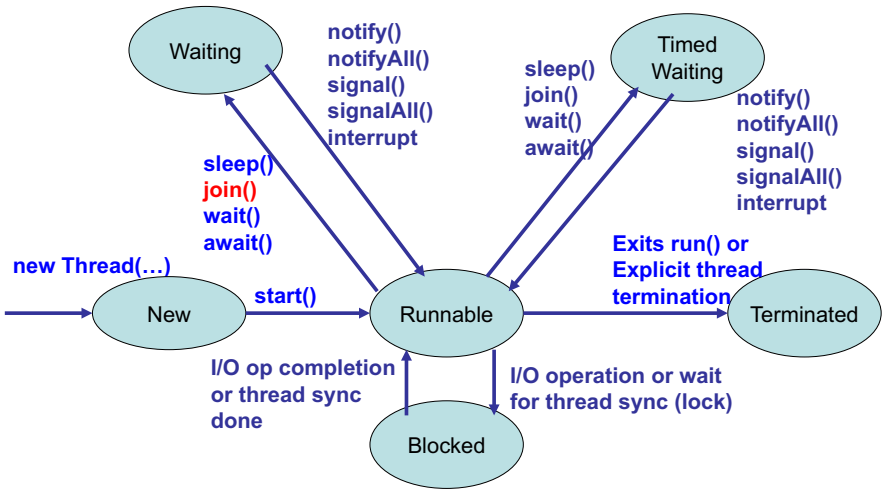
- Client code

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

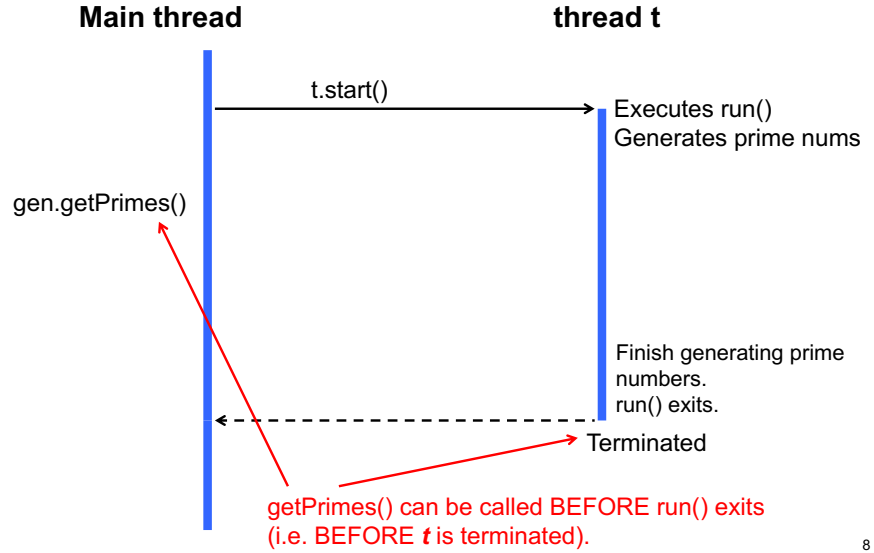

Thread.join()



States of a Thread



This is NOT What You Want



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.

HW 4-1

- Understand PrimeNumberGenerator
- Write a piece of code to
  - Execute its run() on an extra thread.
  - Collect all generated prime numbers after run() exits.
    - Use forEach() on a collection. Pass a lambda expression to forEach()

```
PrimeNumberGenerator gen = new PrimeNumberGenerator(1L, 1000000L);
Thread thread = new Thread(gen);
thread.start();
thread.join();
gen.getPrimes().forEach(...);
```


HW 4-2

- Write a piece of code to
  - Create two extra threads to generate prime numbers
    - in the range from 1 to 1,000,000 on one thread
    - in the range from 1,000,000 to 2,000,000 on the other thread
  - Collect all generated prime numbers after the two threads are terminated.
    - Use forEach() on a collection. Pass a lambda expression to forEach()

```
PrimeNumberGenerator gen1 = new PrimeNumberGenerator(1L, 1000000L);
PrimeNumberGenerator gen2 = new PrimeNumberGenerator(1000001L,...);
Thread t1 = new Thread(gen1);
Thread t2 = new Thread (gen2)
t1.start(); t2.start();
t1.join(); t2.join();
gen1.getPrimes().forEach(...);
gen2.getPrimes().forEach(...);
```

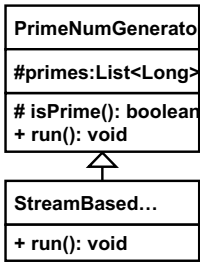

Some Context to this HW

- 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 prime 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 4-3

- Generate prime numbers with a stream.
  - Define `StreamBasedPrimeNumberGenerator` as a subclass of `PrimeNumberGenerator`.



  - Override (re-define) run()
    - PrimeNumberGenerator's run():

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

    - StreamBasedPrimeNumberGenerator's run():

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

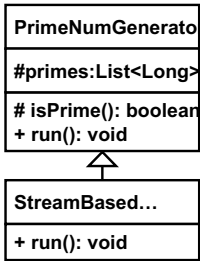
    - c.f. `boxed()` and `collect()`

- **`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: March 27 (Tue) midnight

Free Variables

- Note that a lambda expression can access data fields and methods in its *enclosing* class.
  - c.f. `primes` and `isPrime()` in `PrimeNumberGenerator`.
- Free variables
  - The data fields that a lambda expression accesses



A Note on Free Variables

- 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.
- A LE cannot refer to variables that are mutable.
- Traditionally, immutable variables are defined as `final`; free variables can be defined as `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.