`Runnable` as a Functional Interface

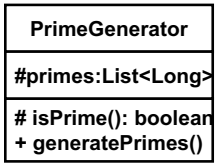
- `java.lang.Runnable`: functional interface
  - Can pass a lambda expression (LE) to `Thread`'s constructor
    - Traditional
      - ```
        GreetingRunnable runnable =
                    new GreetingRunnable("Hello World");
        Thread thread = new Thread(runnable);
        thread.start();
        ```
    - 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

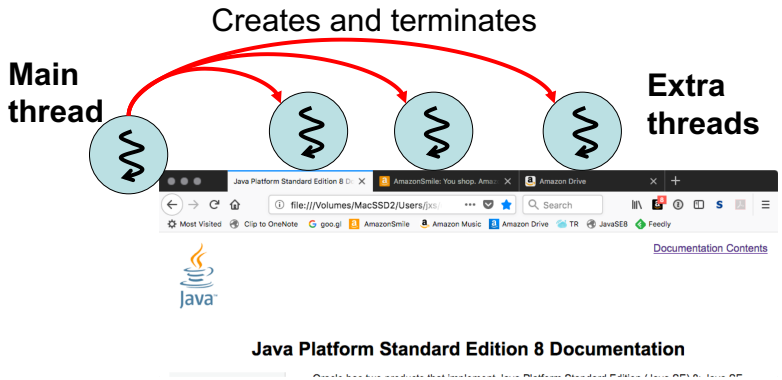
- Define a lambda expression that computes prime numbers in a given range and pass it to `Thread`'s constructor
  - DO NOT use a Runnable class; i.e., DO NOT use `StreamBasedRunnablePrimeGenerator`, which you wrote for HW 4.
  - Write main() of `PrimeGenerator`.
    - ```
      PrimeGenerator gen = new PrimeGenerator(1, 100);
      Thread thread = new Thread( ()->{
           gen.primes = LongStream.rangeClosed(gen.from, gen.to)
                                  .filter( ... )
                                  ... } );
      thread.start();
      ```
  - Deadline: Oct 18 (Thu) midnight



Thread Termination

- ***Implicit*** termination
  - A thread triggers its own death when run() returns.
    - Once a thread starts executing run(), it continues execution until run() returns.
- ***Explicit*** termination
  - A thread terminates another thread.
    - when a certain condition is satisfied.
  - 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.

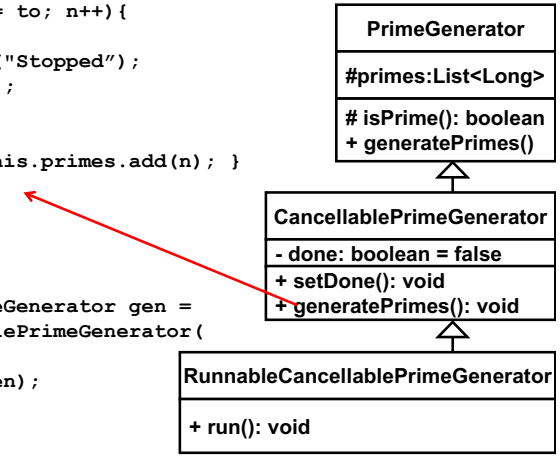
```
public class MyRunnbable implements Runnable{
    private boolean done = false;
    ...
    public void run(){
        while(!done){
            ...
        }
    }
    public setDone(){ done=true; }
}
```

- Have a soon-to-be-killed thread periodically check the flag to determine if it should stop/die.
  - The thread let run() return to die.
- Stop the thread by flipping a flag to inform that the thread should die.

CancellablePrimeGenerator

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

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

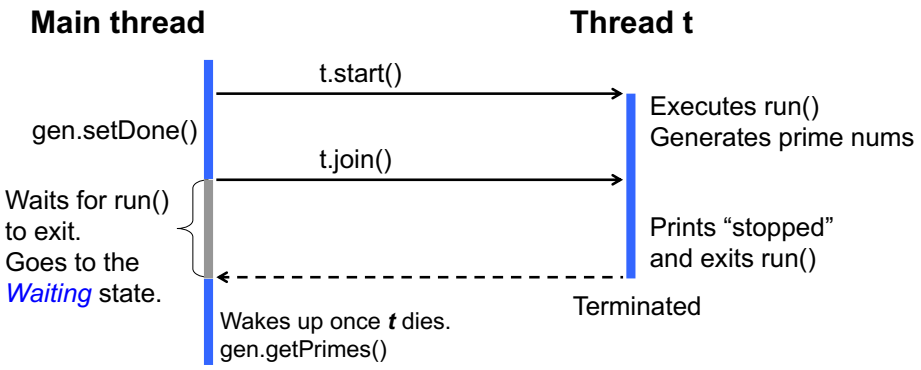


- Client code

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

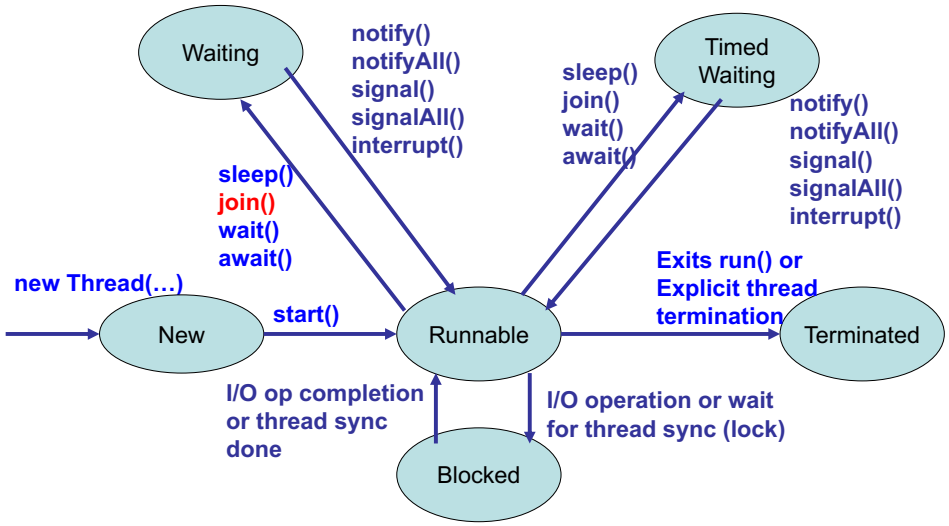

- Alternatively...

```
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(...);
```


States of a Thread



Explicit Thread Termination via Interruption

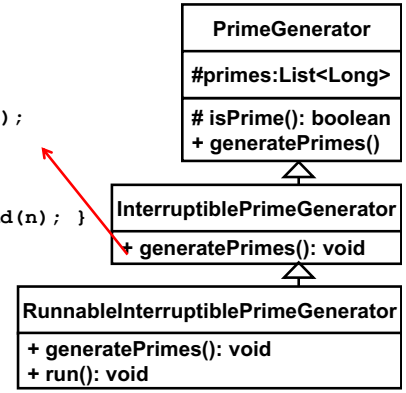
- **`Thread.interrupt()`**
  - Interrupts another thread.
    - `Thread thread = new Thread(aRunnable);`
      `thread.start();`
      `thread.interrupt();`
  - Let that thread know that it is notified via interruption.
- Have a soon-to-be-killed thread periodically detect an interruption to determine if it should stop/die.
  - Let run() return once an interruption is detected.

```
- public class MyRunnbable implements Runnable{
      ...
      public void run(){
          while(!Thread.interrupted()){
              ...
          }
      }
  }
```


InterruptiblePrimeGenerator

- Detect an interruption from another thread to stop generating prime numbers.

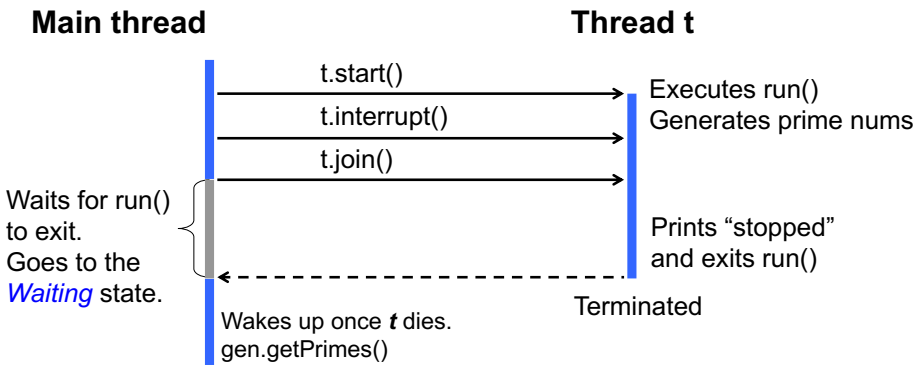
```
- for (long n = from; n <= to; n++){
      if(Thread.interrupted()==true){
          System.out.println("Stopped");
          this.primes.clear();
          break;
      }
      if( isPrime(n) ){ this.primes.add(n); }
  }
```



- 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(...);
```


Exercise

- Write a piece of code to run Cancellable and Interruptible versions of PrimeGenerator
  - The main thread
    - creates an extra thread.
      - The extra thread executes a cancellable/interruptible generator's run().
    - *explicitly terminates* the extra thread while it is generating prime numbers.
      - Flag-based and interruption-based termination
    - call getPrimes() after run() exits.
      - Make sure that getPrimes().size() returns 0.

Which of the 2 Termination Schemes should We Use?

- Flag-based or interruption-based?
- Both work just fine when run() is simple.
  - Both cancellable and interruptible versions of prime number generators work just fine.
- 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 interruptible.
```

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

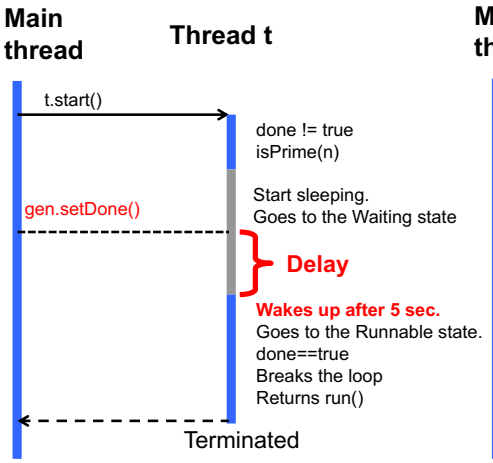
- RunnableCancellablePrimeGenerator

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

- RunnableInterruptiblePrimeGenerator

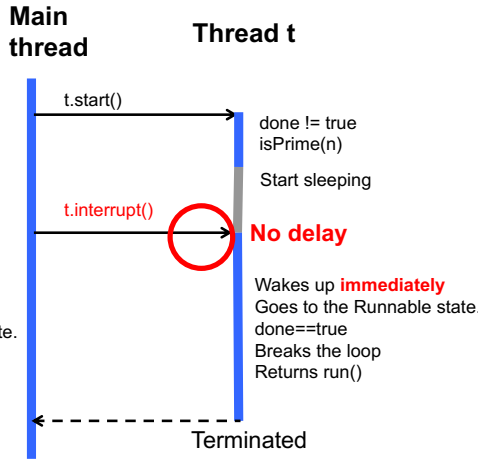
```
for (long n = from; n <= to; n++){
  if(Thread.interrupted()==true){
      System.out.println("Stopped");
      this.primes.clear();
      break;
  }
   if( isPrime(n) ){ this.primes.add(n); }
   Thread.sleep(5000); }
```


RunnableCancellablePrimeGenerator



Sleep period: 5 seconds
Less responsive thread termination

RunnableInterruptiblePrimeGenerator

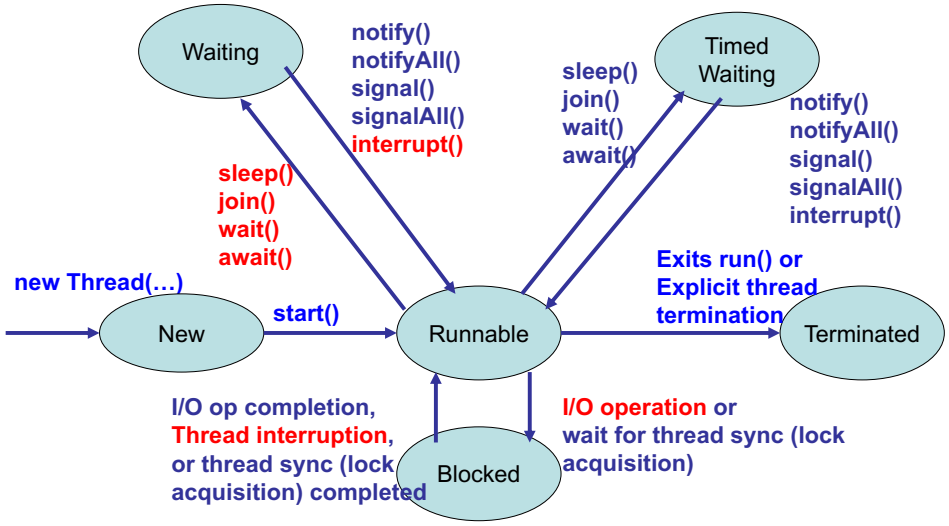


Sleep period: Can be less than 5 seconds
More responsive thread termination

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

States of a Thread



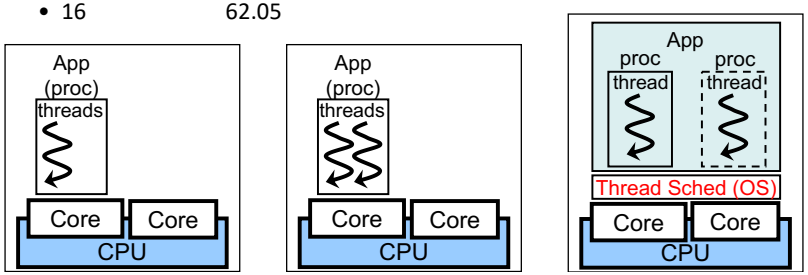
Deprecated Methods for Thread Termination

- `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

Run MCTest.java if You have a Multicore CPU…

- 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



HW 6

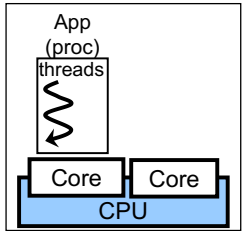
- Modify MCTest.java to use a lambda expression.
  - MCTest.java currently uses an anonymous class to implement run().

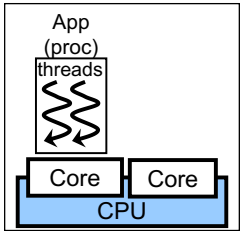
```
Thread t = new Thread(
        new Runnable(){
          public void run(){
            int n = 25;
            for (long j = 0; j < nTimes; j++){
              n *= 25;
            }
          }
        });
```

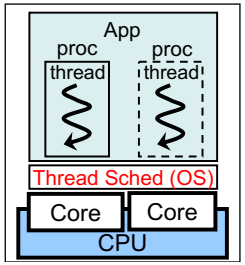
    - Skip implementing a class that implements Runnable.
  - Replace the anonymous class with a lambda expression.
    - Runnable is a functional interface.
  - Run MCTest.java with multiple threads
    - e.g., > java edu.umb.cs.threads.basics.MCTest 1000000000 4
      - First param: # of 25*25 multiplications
      - Second param: # of threads
  - Deadline: Oct 18 (Thu) midnight

HW 7

- Run RunnablePrimeGenerator with multiple threads just like MCTest.
- Measure the overhead for each thread to compute prime numbers to see how threads run.
  - # of threads Time (sec)
  - 1 ???
  - 2 ??? (Two threads run in parallel?)
  - 4 ???
  - 8 ???
  - 16 ???







- Deadline: Oct 23 (Tue) midnight