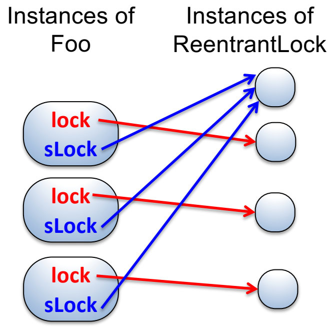
Static Locks

Regular and Static Locks

```
• public class Foo{
       ReentrantLock lock         = new ReentrantLock();
       static ReentrantLock sLock = new ReentrantLock(); }
```

- A regular lock is created and used on an *instance-by-instance* basis.
  - Different instances of `Foo` have different locks (i.e., different instances of `ReentrantLock`).
- A static lock is created and used on a *per-class* basis.
  - All instances of `Foo` share a single lock (`sLock`).



Exercise: Regular and Static Locks

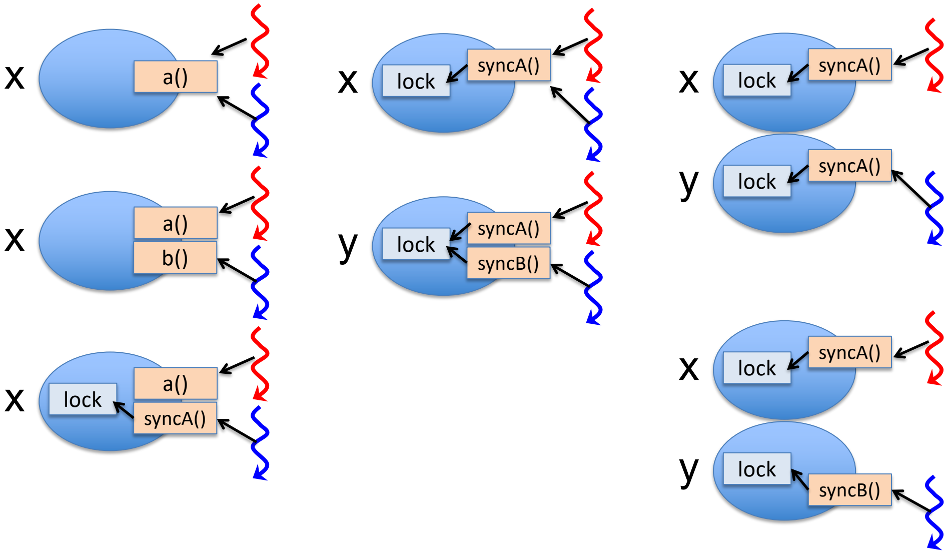
```
• public class Foo{
     private ReentrantLock lock = new ReentrantLock();

     public void a(){…}
     public void b(){…}
     public void syncA(){lock.lock(); … lock.unlock();}
     public void syncB(){lock.lock(); … lock.unlock();}   }
```

```
• x = new Foo(); y = new Foo();
```

- Two threads call…

– x.a() and x.a():	no synchronization (no mutual exclusion) for the two threads
– x.a() and x.b():	no synchronization
– x.a() and x.syncA():	no synchronization
– x.syncA() and x.syncA():	Synchronization (mutual exclusion)
– y.syncA() and y.syncB():	Synchronization
– x.syncA() and y.syncA():	No synchronization
– x.syncA() and y.syncB():	No synchronization



-
  ```
  public class Foo{
     private ReentrantLock          lock = new ReentrantLock();
     private static ReentrantLock   sLock = new ReentrantLock();

     public        void a(){…}
     public        void b(){…}
     public        void syncA(){lock.lock(); … lock.unlock();}
     public        void syncB(){lock.lock(); … lock.unlock();}

     public static void sA(){…}
     public static void sB(){…}
     public static void sSyncA(){sLock.lock(); … sLock.unlock();}
     public static void sSyncB(){sLock.lock(); … sLock.unlock();} }
  ```

-
  ```
  x = new Foo(); y = new Foo();
  ```
- Two threads call…

– x.a() and Foo.sA():	No synchronization for the two threads
– Foo.sA() and Foo.sA():	No synchronization
– Foo.sA() and Foo.sB():	No synchronization
– x.syncA() and Foo.sA():	No synchronization
– x.syncA() and Foo.sSyncA()	No synchronization
– Foo.sSyncA() and Foo.sSyncA():	Synchronization (mutual exclusion)
– Foo.sSyncA() and Foo.sSyncB():	Synchronization (mutual exclusion)
– x.sSyncA() and y.sSynchB():	Synchronization

  - This is not grammatically wrong, but write Foo.sSyncA() instead of x.sSyncA()

Thread.sleep()

-
  ```
  Thread t = new Thread( new FooRunnable() );
  t.start();
  try{
       t.sleep(1000);
  }catch(InterruptedException e){...}
  ```

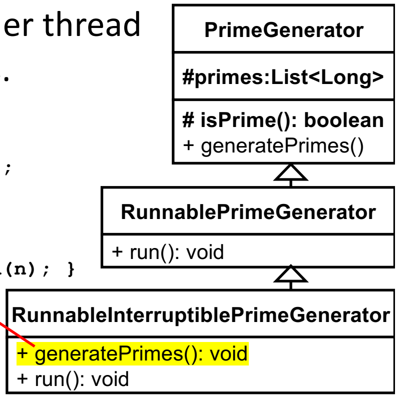
- It looks like an extra thread (*t*) will sleep.
- However, the main thread will actually sleep
  - because `sleep()` is a ***static method*** of Thread.
    - `Thread.sleep()`: Allows the currently executed thread to sleep (temporarily cease execution) for the specified number of milliseconds
- DO NOT write `t.sleep(…)`. It's misleading and error-prone.
- ALWAYS WRITE `Thread.sleep(…).`

Race Conditions (cont'd)

RunnableInterruptiblePrimeGenerator

- 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); }
    }
    ```


PrimeGenerator

#primes:List<Long>

# isPrime(): boolean
+ generatePrimes()

RunnablePrimeGenerator

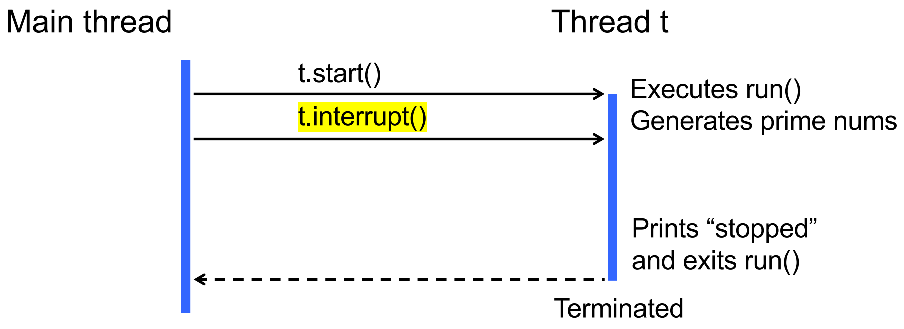
+ run(): void

RunnableInterruptiblePrimeGenerator

+ generatePrimes(): void
+ run(): void

- Client code

  -
    ```
    RunnableInterruptiblePrimeGenerator gen =
        new InterruptiblePrimeNumberGenerator(1L, 1000000L);
    Thread t = new Thread(gen); t.start();
    t.interrupt();
    ```



Main thread

Thread t

t.start()

t.interrupt()

Executes run()
Generates prime nums

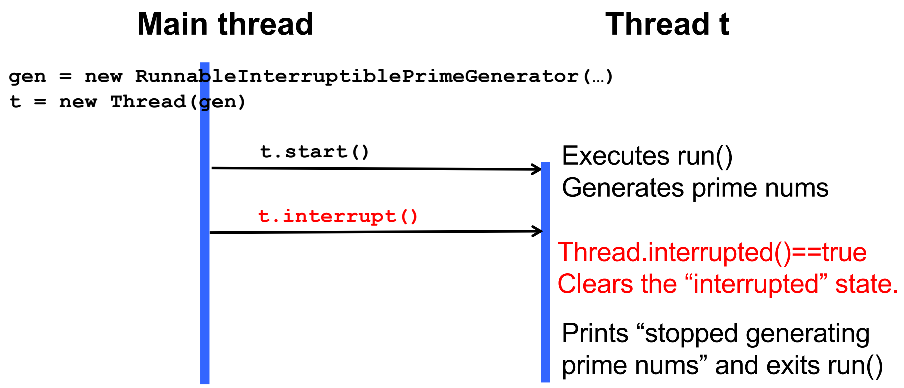
Prints “stopped”
and exits run()

Terminated

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

interrupt(), isInterrupted() and interrupted()

-
  ```
  public class Thread{
    public void interrupt();
    public boolean isInterrupted();
    public static boolean interrupted();
  ```
- Each thread (`Thread` instance) has the “interrupted” (boolean) data field.
- `interrupt()`
  - Interrupts `this` thread and changes its “interrupted” state.
    - `Thread t = new Thread(...); t.start();`
      `t.interrupt();`
- `isInterrupted()`
  - Returns true if `this` thread has been interrupted.
    - `Thread t = new Thread(…); t.start();`
      `if( t.isInterrupted() ){...}`
  - Does not change the “interrupted” state of the thread.
- `interrupted()`
  - Returns true if the *currently-executed* thread has been interrupted.
  - Clears the “interrupted” state (true → false) if true is returned.



Main thread

Thread t

```
gen = new RunnableInterruptiblePrimeGenerator(…)
t = new Thread(gen)
```

t.start()

t.interrupt()

Executes run()
Generates prime nums

Thread.interrupted()==true
Clears the “interrupted” state.

Prints “stopped generating
prime nums” and exits run()

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

Thread Interruption != Thread Termination

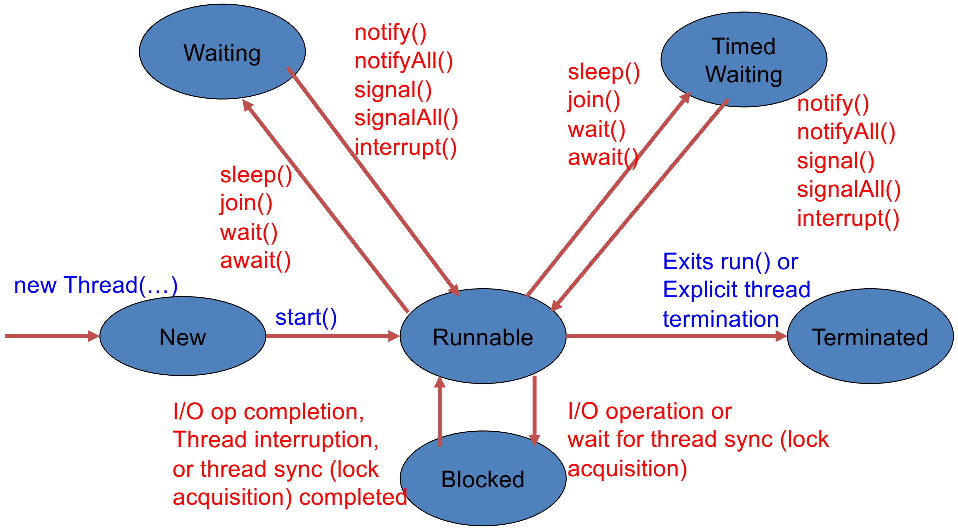
- `interrupt()` NEVER terminate a thread.
  - It simply changes the “interrupted” state
    - to trigger a thread termination.

What Happens When `interrupt()` is Called on a Thread?

- If the soon-to-be-terminated thread is in the Runnable state, `interruput()` changes its "interrupted" state to be true.

- If the soon-to-be-terminated thread is in the *Waiting* or *Blocked* state, it throws an `InterruptedException`.
  - e.g., it called `Thread.sleep()`, and it has been sleeping (waiting).
  - It is reading data from the local disk or the network.
  - It tried to acquire a lock, but it hasn't been available.

States of a Thread



Waiting

notify()
notifyAll()
signal()
signalAll()
interrupt()

sleep()
join()
wait()
await()

Timed Waiting

sleep()
join()
wait()
await()

notify()
notifyAll()
signal()
signalAll()
interrupt()

new Thread(…)

New

start()

Runnable

Exits run() or Explicit thread termination

Terminated

I/O op completion, Thread interruption, or thread sync (lock acquisition) completed

Blocked

I/O operation or wait for thread sync (lock acquisition)

`RunnableInterruptiblePrimeGenerator`

- In fact, this code is NOT thread-safe. Race conditions can occur.

```
class RunnableInterruptiblePrimeGenerator
                                extends RunnablePrimeGenerator {
  public void generatePrimes(){
    for (long n = from; n <= to; n++){
      if( Thread.interrupted() ){         // 2 steps
        System.out.println("Stopped");
        this.primes.clear();
        break;
      }
      if( isPrime(n) ){ this.primes.add(n); } }

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

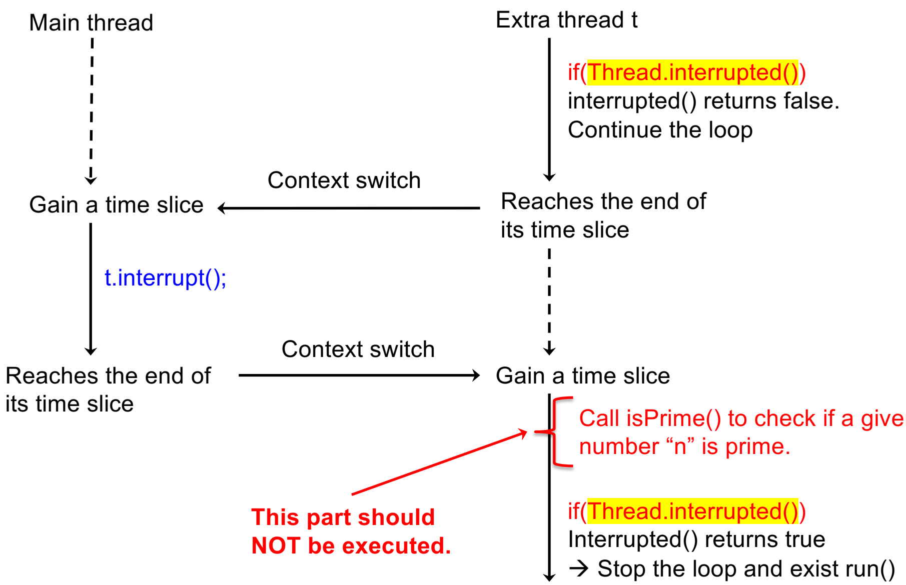
`Thread.interrupt()`

-
```
public void interrupt(){
  ...
  synchronized(...){ // Acquire a lock in Thread
    ...
    interrupt0();    // native method (atomic)
    ...
  }
}
public static boolean interrupted(){
  return currentThread().isInterrupted(true);// native method
                                               //  (atomic)
 }
```

- `interrupt()` and `interrupted()` are thread-safe.
  - `isInterrupted()` is thread-safe as well.
  - c.f. Java source code

- However, ***client code*** of `interrupted()` is NOT thread-safe.

A Potential Race Condition



Main thread

Extra thread t

if(Thread.interrupted())
interrupted() returns false.
Continue the loop

Gain a time slice

Context switch

Reaches the end of
its time slice

t.interrupt();

Reaches the end of
its time slice

Context switch

Gain a time slice

Call isPrime() to check if a given
number “n” is prime.

**This part should
NOT be executed.**

if(Thread.interrupted())
Interrupted() returns true
→ Stop the loop and exist run()

Thread-safe Version
w/ Thread Synchronization

-
  ```
  class RunnableInterruptiblePrimeGenerator
                                extends RunnablePrimeGenerator {

    private final ReentrantLock lock = new ReentrantLock();

    public ReentrantLock getLock(){
      return lock;  }

    public void generatePrimes(){
      for (long n = from; n <= to; n++){
        lock.lock();
        if( Thread.interrupted() ){      // Read logic on “interrupted”
          System.out.println("Stopped");//   data field in Thread
          this.primes.clear();
          break;
        }
        if(isPrime(n)){this.primes.add(n);}
        lock.unlock(); } }

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

- Main thread

  -
    ```
    RunnableInterruptiblePrimeGenerator gen =
                            new RunnableInterruptiblePrimeGenerator();
    Thread aThread = new Thread(gen);
    aThread.start();

    gen.getLock().lock();
    aThread.interrupt();    // Write logic on the “interrupted” data
    gen.getLock().unlock(); // field in Thread
    ```
- This code uses two locks.
  - One in `Thread`
    - `Thread` uses the lock in `interrupt()` and `interrupted()` to guard the “interrupted” state.
  - One in `RunnableInterruptiblePrimeGenerator`

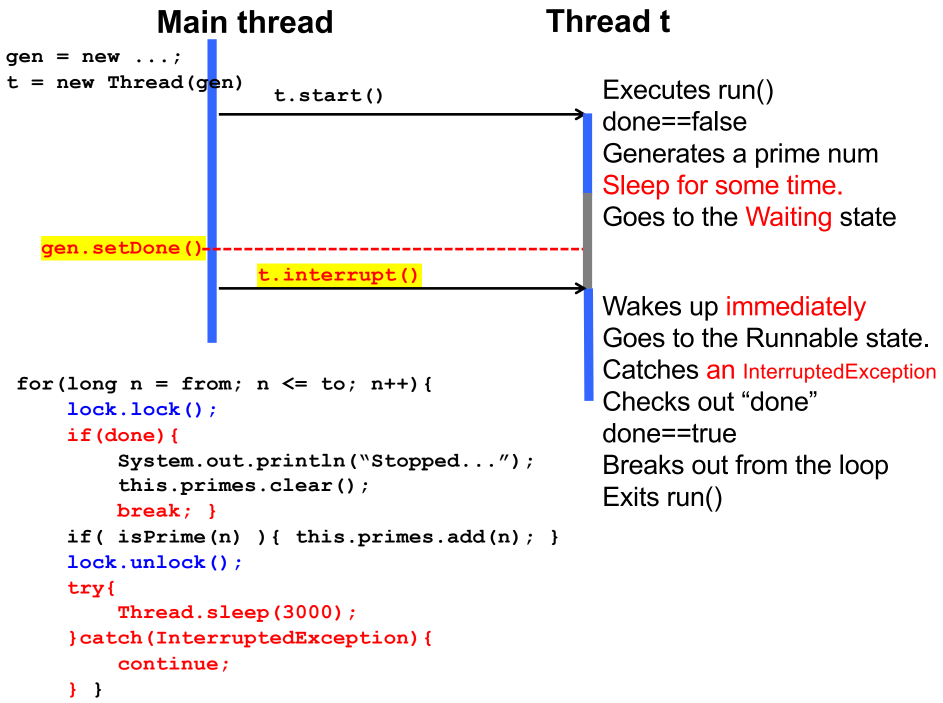
2-Step (“Graceful”)
Thread Termination

Explicit Thread Termination

- Flag-based
  - Pros:
    - Uses **1 lock (computationally less expensive)**
  - Cons:
    - Program responsiveness may be lower.
      - if a flag-flipping (e.g. done==false → true) happens when a soon-to-be-terminated thread is in the Waiting or Blocked state.
- Interruption-based
  - Pros
    - Higher program responsiveness
      - interrupt() can immediately wake up a soon-to-be-terminated thread that is in the Waiting or Blocked state
  - Cons
    - Uses **2 locks (computationally more expensive)**

2-Step Thread Termination

- Hybrid of the 2 approaches
  - Intended to offer a responsive thread termination that uses only 1 lock.
- Primarily takes the flag-based approach.
  - A soon-to-be-terminated thread periodically checks a flag.
- Let a "terminator" thread call `interrupt()` after flipping the flag's state
  - e.g., after calling `setDone()`



Main thread

Thread t

```
gen = new ...;
t = new Thread(gen)
```

t.start()

Executes run()
done==false
Generates a prime num
Sleep for some time.
Goes to the Waiting state

gen.setDone()

t.interrupt()

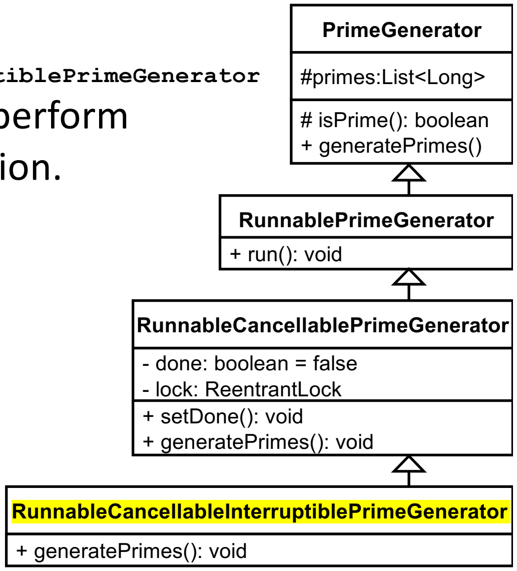
Wakes up immediately
Goes to the Runnable state.
Catches an InterruptedException
Checks out "done"
done==true
Breaks out from the loop
Exits run()

```
for(long n = from; n <= to; n++){
    lock.lock();
    if(done){
        System.out.println("Stopped...");
        this.primes.clear();
        break; }
    if( isPrime(n) ){ this.primes.add(n); }
    lock.unlock();
    try{
        Thread.sleep(3000);
    }catch(InterruptedException){
        continue;
    } }
```

Exercise:

`RunnableCancellableInterruptiblePrimeGenerator`

- Read and run `RunnableCancellableInterruptiblePrimeGenerator` to understand how to perform 2-step thread termination.



PrimeGenerator
#primes:List<Long>
# isPrime(): boolean
+ generatePrimes()

RunnablePrimeGenerator
+ run(): void

RunnableCancellablePrimeGenerator
- done: boolean = false
- lock: ReentrantLock
+ setDone(): void
+ generatePrimes(): void

RunnableCancellableInterruptiblePrimeGenerator
+ generatePrimes(): void

HW 8

- Define `RunnableCancellableInterruptiblePrimeFactorizer` by extending `RunnableCancellablePrimeFactorizer`.
  - Add 2-step thread termination

2-Step Thread Termination is Effective if…

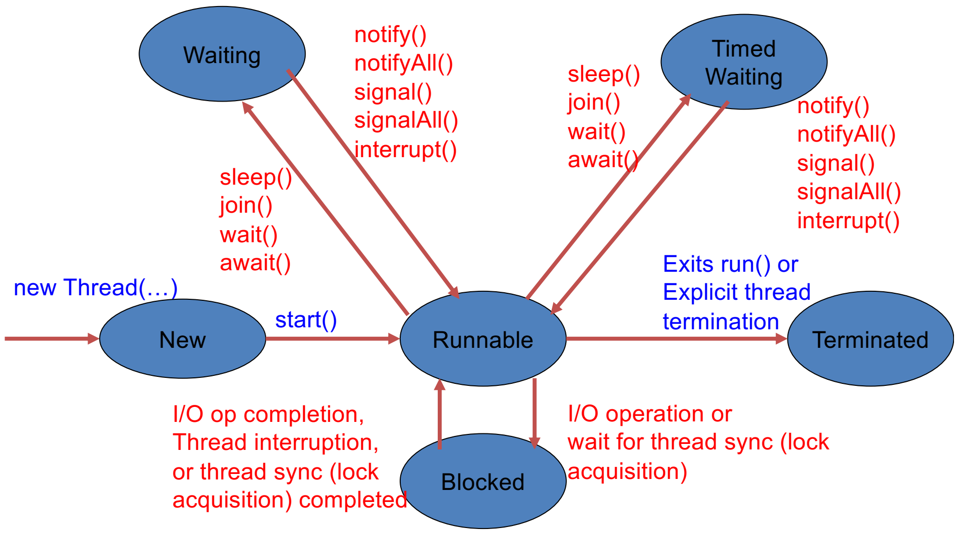
- A “soon-to-be-terminated” thread may be in the Waiting or Blocked state when a “terminator” thread tries to terminate it.
  - Performing an I/O operation.
    - e.g., reading/writing data from/to a file, waiting for an incoming data on a socket, sending data to a remote app.
  - Waiting for a lock acquisition
    - Has called lock() on a lock, but the lock is not available yet.
  - Has called sleep(), join(), etc.

What Happens When `interrupt()` is Called on a Thread?

- If a soon-to-be-terminated thread is in the Runnable state, `interruput()` changes its “interrupted” state to be true.

- If the soon-to-be-terminated thread is in the *Waiting* or *Blocked* state, it raises an `InterruptedException`.

States of a Thread



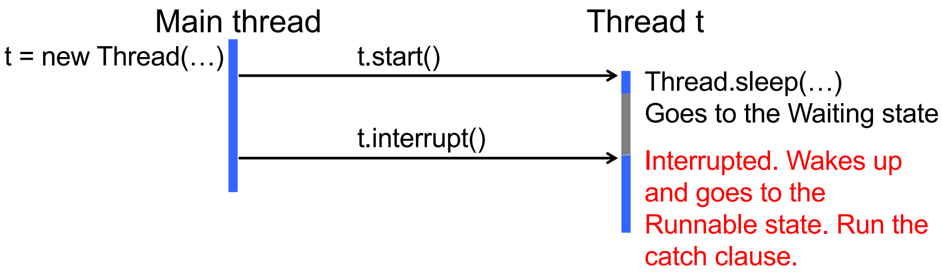
InterruptedException

- Some methods in Java API throws **InterruptedException**.
  - They can respond to a thread interruption by throwing an **InterruptedException**.
  - **Thread.sleep()**
  - **Thread.join()**
  - I/O operations
  - **Condition.await()**
  - **ReentrantLock.lockInterruptibly()**
  - **BlockingQueue.put()/take()**
  - These methods can be long-running and interruptible.

Thread.sleep()

- **sleep()** lets the *currently-executed thread* to sleep for a specified time period.
- **interrupt()** interrupts a sleeping thread.
  - Wakes up the thread and force **sleep()** to throw an **InterruptedException**.

-
  ```
  try{
       Thread.sleep(60000);
  }catch(InterruptedException e){
       // Write thread termination (shutdown) logic here.
  }
  ```


Main thread

Thread t

t = new Thread(…)

t.start()

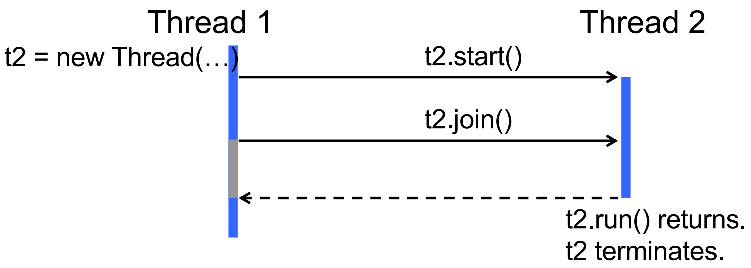
Thread.sleep(…)
Goes to the Waiting state

t.interrupt()

Interrupted. Wakes up and goes to the Runnable state. Run the catch clause.

Thread.join()

- **join()** lets the *currently-executed thread* to wait/sleep until another thread terminates (i.e., until another thread returns **run()**).
- **interrupt()** can interrupt a waiting/sleeping thread.
  - Force join() to throw an **InterruptedException**.


Thread 1

Thread 2

t2 = new Thread(…)

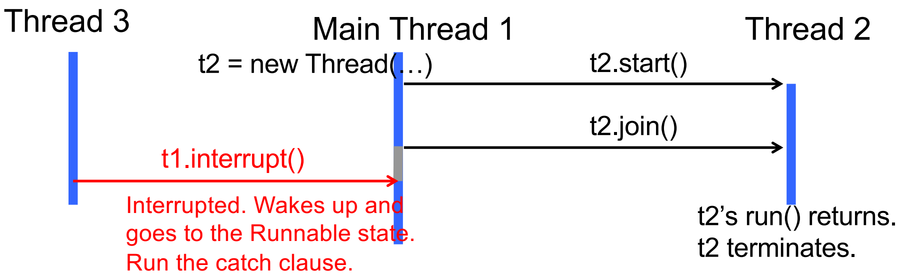
t2.start()

t2.join()

t2.run() returns.
t2 terminates.

Thread.join()

- **join()** lets the *currently-executed thread* to wait/sleep until another thread terminates (i.e., until another thread returns **run()**).
- **interrupt()** can interrupt a waiting/sleeping thread.
  - Force join() to throw an **InterruptedException**.


Thread 3

Main Thread 1

Thread 2

t2 = new Thread(…)

t2.start()

t2.join()

t1.interrupt()

Interrupted. Wakes up and goes to the Runnable state. Run the catch clause.

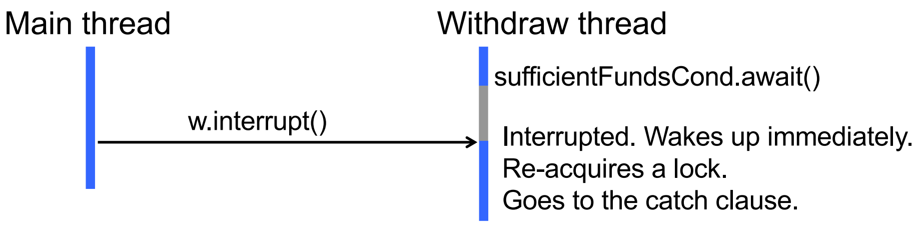
t2's run() returns.
t2 terminates.

Condition.await()

- `await()` lets the currently-executed thread wait/sleep until another thread wakes it up with `signal()/signalAll()`.

- `interrupt()` can interrupt a waiting/sleeping thread.
  - Allows `await()` to acquire a lock and forces it to throw an `InterruptedException`

```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something
        }
    }
    belowUpperLimitFundsCondition.signalAll();
    balance -= amount;
    lock.unlock(); }
```



Interrupted. Wakes up immediately.
Re-acquires a lock.
Goes to the catch clause.

```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something; e.g., balk with a
            //"break" statement.
        }
    }
    ...}
```

Thread Termination

- Thread creation is a no brainer.
- Thread termination requires your careful attention.
  - No methods available in Thread to directly terminate threads like `terminate()`.
    - Do: 2-step termination
  - Why not?
    - Different programmers/apps need different termination policies.
      - Notify on-going thread termination to other threads?
      - Raise exception(s) in addition to `InterruptException`?
      - What to do for the data maintained by a thread being terminated?
    - Java allows you to flexibly craft your own termination policy.