Thread Interruption

Thread Interruption

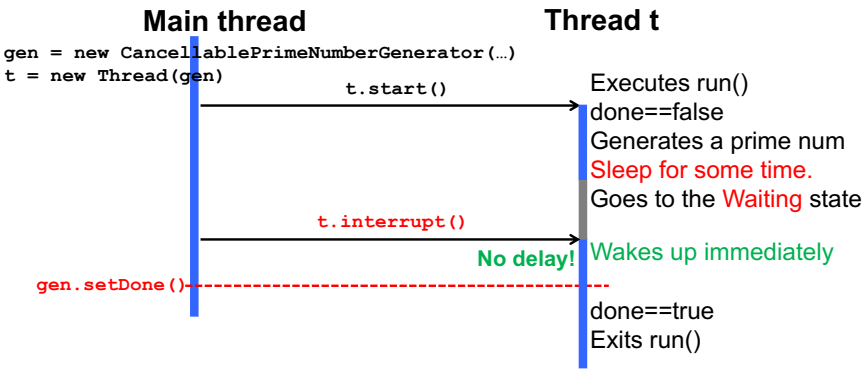
- Often used to stop/cancel a task being executed by a thread and help terminate the thread.
  - Used in one of two approaches for thread termination
    - Flag-based and interruption-based approaches
  - Used in 2-step thread termination
    - Hybrid of Flag-based and interruption-based approaches

Explicit Thread Termination

- Flag-based
  - Pros:
    - Uses **1 lock (faster)**
  - Cons:
    - Need to define and maintain a flag by yourself.
    - Program responsiveness may be lower.
      - if a flag-flipping (e.g. done==false → true) happens when a thread to be terminated is in the Waiting or Blocked state.
- Interruption-based
  - Pros
    - No need to define and maintain a flag
    - Higher program responsiveness
      - interrupt() can immediately wake up a thread that is in the Waiting or Blocked state
  - Cons
    - Uses **2 locks (slower)**

2-Step Thread Termination

- Hybridization of Flag-based and interruption-based approaches
  - Designed for responsive thread termination that uses only 1 lock




```
public void run(){
    for(long n = from; n <= to; n++){
        aLock.lock();
        if(done){
            System.out.println("Stopped generating prime nums.");
            this.primes.clear();
            break; }
        aLock.unlock();
        if( isPrime(n) ){ this.primes.add(n); }
        Thread.sleep(3000); }  } }
```

interrupt(), isInterrupted() and interrupted()

-
```
public class Thread{
  public void interrupt();
  public boolean isInterrupted();
  public static boolean interrupted();
  …
}
```

- **`interrupt()`**
  - Interrupt `this` thread and change its "interrupted" (boolean) state.
    -
```
aThread = new Thread(...);
aThread.start();
aThread.interrupt();
```

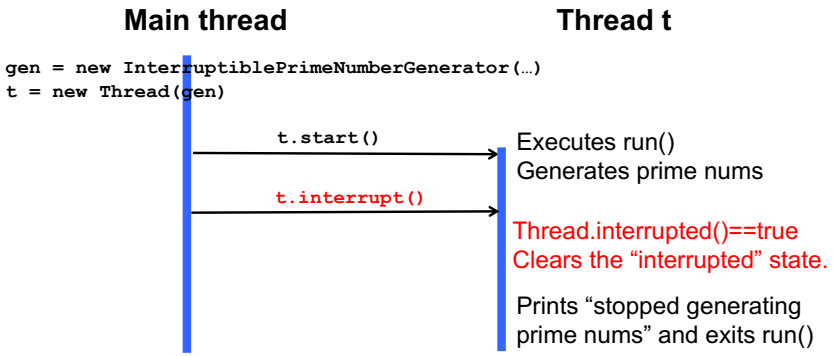
- **`isInterrupted()`**
  - Returns true if `this` thread has been interrupted.
    -
```
aThread = new Thread(…);
aThread.start();
...
if( aThread.isInterrupted() ){...}
```
  - Does not change the "interrupted" state of the thread.

- **`interrupted()`**
  - Static method
  - Returns true if the ***currently-executed*** thread has been interrupted.
  - Clears the "interrupted" state (true → false) if true is returned.

InterruptiblePrimeNumberGenerator

```
class InterruptiblePrimeNumberGenerator extends PrimeNumberGenerator{

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




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

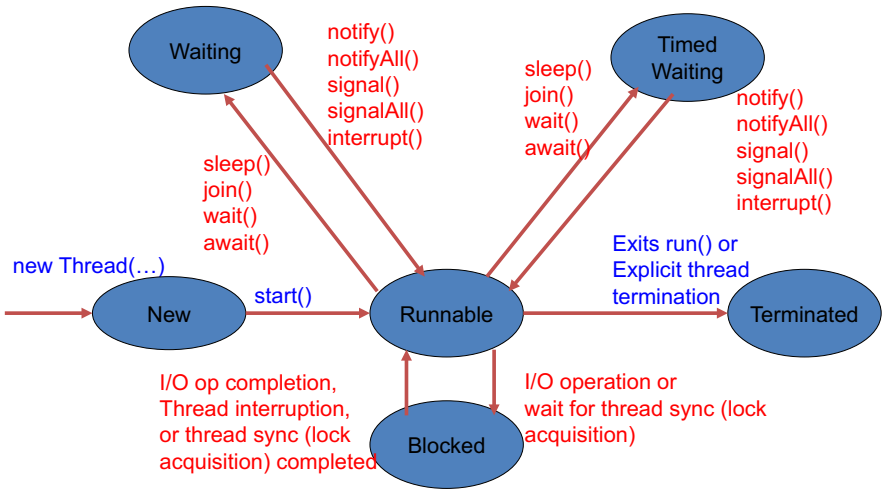

Thread Interruption DOES NOT Mean Thread Termination

- `interrupt()` NEVER terminate a thread.
  - It simply helps/triggers a thread termination.

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

- If the target thread is in the Runnable state, it changes its "interrupted" state to be true.
- If the target thread is in the *Waiting* or *Blocked* state, it raises an `InterruptedException`.

States of a Thread

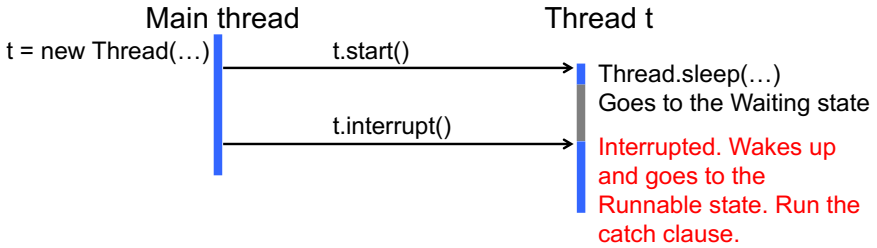


InterruptedException

- Some methods in Java API throws `InterruptedException`.
  - `Thread.sleep()`
  - `Thread.join()`
  - `ReentrantLock.lockInterruptibly()`
  - `BlockingQueue.put()/take()`
  - `Condition.await()`
  - I/O operations
  - These methods can be long-running and interruptible.

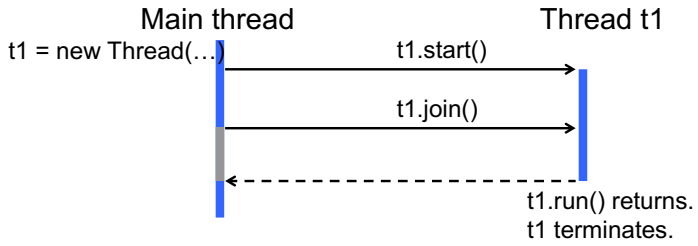
Thread.sleep()

- `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.
}
```



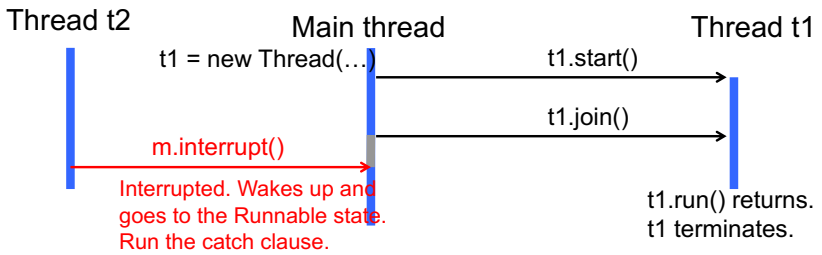
Thread.join()

- `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.join()

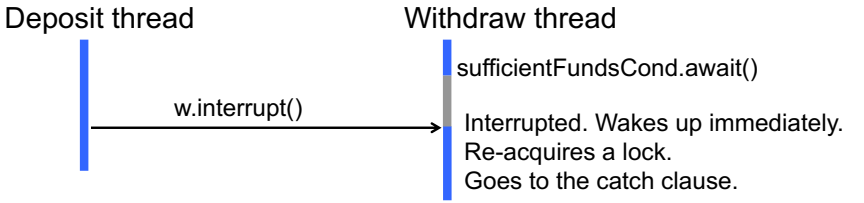
- `join()` lets the *currently-executed thread* to wait/sleep until another thread terminates (i.e., until another thread returns `run()`).
- `interrupt()` can interrupt a waiting/sleeping thread.
  - Force join() to throw an `InterruptedException`.



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(); }
```

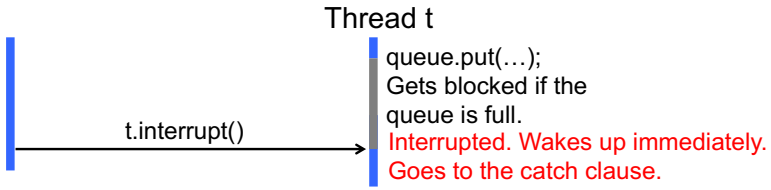



```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something
        }
    }
    ...}
```

- A "D" thread does not need to acquire a lock at the "W" side for calling interrupt().

BlockingQueue

- `interface BlockingQueue<E> extends Queue<E>`
  - Adds A Queue that additionally supports operations that
    - wait for the queue to become non-empty when retrieving an element
    - wait for space to become available in the queue when storing an element.
- Several impls: `ArrayBlockingQueue`, `LinkedBlockingQueue`, etc.
  - `put()` and `take()` are *blocking* methods.
    - `put()`: Add an element to a queue as the last element.
    - `take()`: Get the first element in the queue.
  - They can respond to an interruption by throwing an `InterruptedException`.



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 the 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.

Recap: InterruptiblePrimeNumberGenerator

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

```
class InterruptiblePrimeNumberGenerator extends PrimeNumberGenerator{

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

Thread.interrupt()

```
public void interrupt(){
  ...
  synchronized(...){
    ...
    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 (e.g. grepcode.com)
- However, ***client code*** of `interrupted()` is not guaranteed to be thread-safe.

- InterruptiblePrimeNumberGenerator is not thread-safe. Race conditions can occur.

```
class InterruptiblePrimeNumberGenerator extends PrimeNumberGenerator{

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

Solution: Use a Lock

```
lock.lock();
aThread.interrupt();
lock.unlock();
```

```
while(true){
    lock.lock();
    if(Thread.interrupted()) break; // balking
    // do something
    lock.unlock();
}
```

- This code uses two locks.
  - One in `Thread` for `interrupt()` and `interrupted()`
  - One for client code of those methods.

Explicit Thread Termination

- Flag-based
  - Pros:
    - Uses **1 lock (faster)**
  - Cons:
    - Need to define and maintain a flag by yourself.
    - Program responsiveness may be lower.
      - if a flag-flipping (e.g. done==false → true) happens when a thread to be terminated is in the Waiting or Blocked state.
- Interruption-based
  - Pros
    - No need to define and maintain a flag
    - Higher program responsiveness
      - interrupt() can immediately wake up a thread that is in the Waiting or Blocked state
  - Cons
    - Uses **2 locks (slower)**