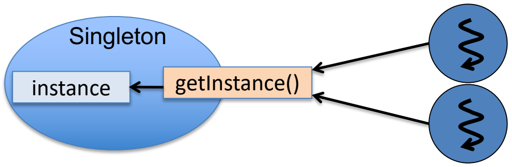
Recap: Singleton Design Pattern

- Guarantee that a class has only one instance.
  - c.f. CS680 lecture note

-
```
public class Singleton{
  private Singleton(){};
  private static Singleton instance = null;

  // Factory method to return the singleton instance
  public static Singleton getInstance(){
     if(instance==null)
       instance = new Singleton();
     return instance;
 }
}
```

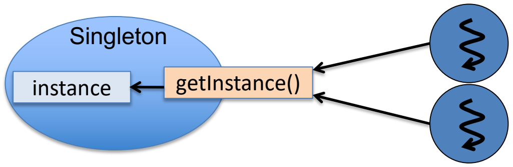
- This code is NOT thread-safe; race conditions can occur.



- When multiple threads call `getInstance()` concurrently, they share `instance`.

-
```
public class Singleton{
  private Singleton(){};
  private static Singleton instance = null;

  // Factory method to return the singleton instance
  public static Singleton getInstance(){
     if(instance==null)                 // Read    3 steps
       instance = new Singleton();      // Write   2+ steps
     return instance;                   // Read    2 steps
 }
}
```



-
```
public class Singleton{
  private Singleton(){};
  private static Singleton instance = null;
                                   // Write 2 steps
                                   // Can assume this to be
                                   // performed atomically   }
```

- JVM completes all initial value assignments on all static data fields BEFORE using a class or creating class instances.
  - `instance` has been initialized before a thread(s) call `getInstance()`
  - You can assume this write logic is performed atomically.
    - No need to worry about race conditions here.

Concurrent Singleton Design Pattern

- Guarantee that a class has only one instance.

-
```
public class ConcurrentSingleton{
 private Singleton(){};
 private static Singleton instance = null;
 private static ReentrantLock lock = new ReentrantLock();

 // Factory method to create or return the singleton instance
 public static Singleton getInstance(){
    lock.lock();
    try{
      if(instance==null){ instance = new Singleton(); }
      return instance;
    }finally{
      lock.unlock();
    }
 }
}
```

HW 10

- The Singleton class is not thread-safe.
  - Race conditions can occur if you do not guard the `instance` variable with a lock. Explain a potential race condition in which more than one instances are created.
    - Use a diagram like in a previous slide.
- Submit a thread-safe version of it (`ConcurrentSingleton`)
  - Define a lock in Singleton. Use the lock in `getInstance()`
    - Use try-finally blocks: <u>Always do this in all subsequent HWs.</u>
  - Create multiple extra threads and have them call `getInstance()`
    - Make sure that only one instance is created.
      - Use `System.out.println(Singleton.getInstance())`

- Deadline: Oct 25 (Thu) midnight

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

Exercise: Regular and Static Locks

```
• 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 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
  - x.syncA() and Foo.sA(): No synchronization
  - Foo.sA() and Foo.sA(): No synchronization
  - Foo.sA() and Foo.sB(): No synchronization
  - x.syncA() and Foo.sSyncA() No synchronization
  - Foo.sSyncA() and Foo.sSyncA(): Synchronization
  - Foo.sSyncA() and Foo.sSyncB(): Synchronization
  - x.sSyncA() and y.sSynchB(): Synchronization
    - This is not grammatically wrong, but write Foo.sSyncA() instead of x.sSyncA()

Thread.sleep()

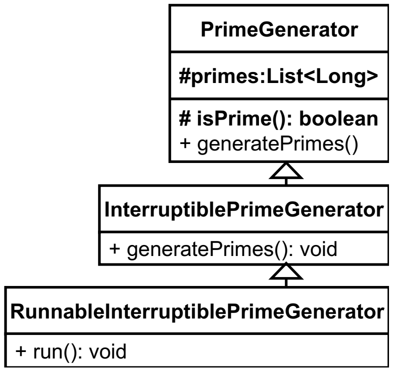
-
  ```
  Thread t = new Thread( new FooRunnable() );
  t.start();
  try{
       t.sleep(1000);
  }catch(InterruptedException e){...}
  ```
- It looks like an extra thread (*t*) will sleep.
- However, the main thread will actually sleep
  - because `sleep()` is a ***static method*** of Thread.
    - `Thread.sleep()`: Causes the *currently executing 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(…).`
  - Make sure to do this in HW 9.

RunnableInterruptiblePrimeGenerator

```
class InterruptiblePrimeGenerator extends PrimeGenerator {
  public void generatePrimes(){
    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); } }}

class RunnableInterruptiblePrimeGenerator
  extends InterruptiblePrimeGenerator
  implements Runnable {

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



PrimeGenerator

#primes:List<Long>

# isPrime(): boolean
+ generatePrimes()

InterruptiblePrimeGenerator

+ generatePrimes(): void

RunnableInterruptiblePrimeGenerator

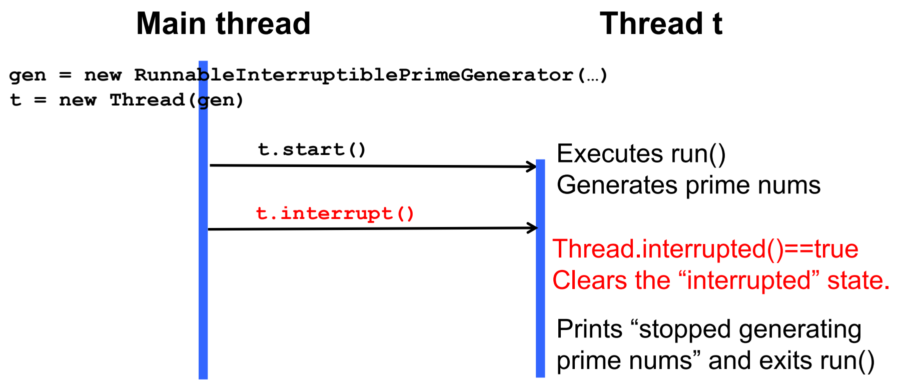
+ run(): void

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) state.
- `interrupt()`
  - Interrupts `this` thread and changes its "interrupted" 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()`
  - 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. 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); } } }
```

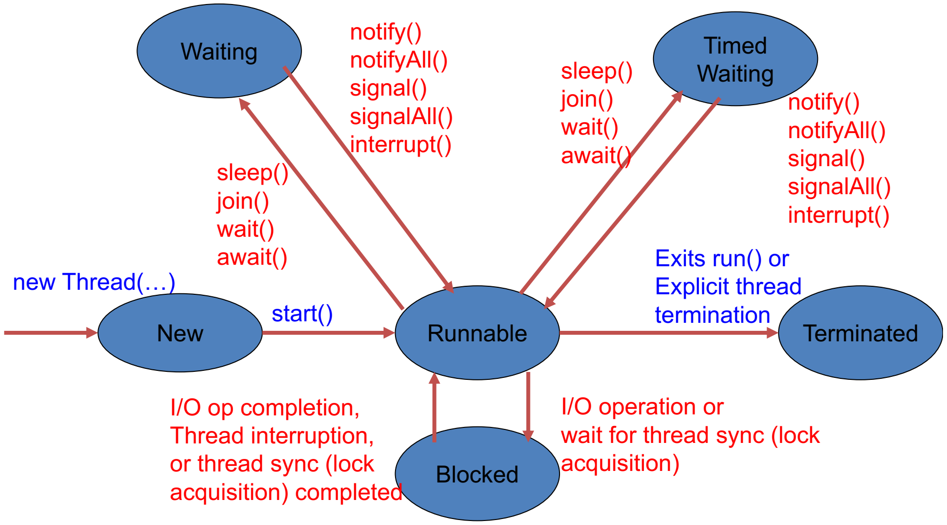
Thread Interruption != Thread Termination

- `interrupt()` NEVER terminate a thread.
  - It simply change the “interrupted” state
    - to help/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`.

States of a Thread



RunnableInterruptiblePrimeGenerator

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

```
class InterruptiblePrimeGenerator extends PrimeGenerator {
  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); } }}

class RunnableInterruptiblePrimeGenerator
  extends InterruptiblePrimeGenerator
  implements Runnable {

  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 (e.g. grepcode.com)

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

Solution: Locking and Balking

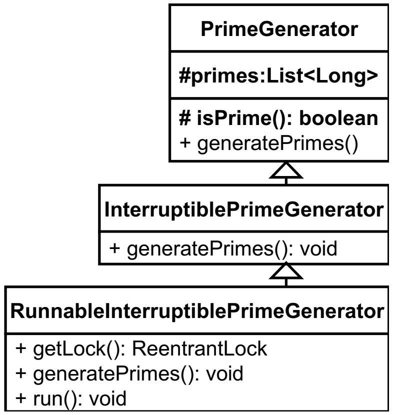
```
class RunnableInterruptiblePrimeGenerator
  extends InterruptiblePrimeGenerator
  implements Runnable {

  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() ){
        System.out.println("Stopped");
        this.primes.clear();
        break;
      }
      lock.unlock();
      if(isPrime(n)){this.primes.add(n);}}}

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



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

InterruptiblePrimeGenerator
+ generatePrimes(): void

RunnableInterruptiblePrimeGenerator
+ getLock(): ReentrantLock
+ generatePrimes(): void
+ run(): void

- Main thread (client of `RunnableInterruptiblePrimeGenerator`)
  -
```
RunnableInterruptiblePrimeGenerator gen =
    new RunnableInterruptiblePrimeGenerator();
Thread aThread = new Thread(gen); aThread.start();

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

- This code uses two locks.
  - One in `Thread`
  - One in `RunnableInterruptiblePrimeGenerator`

HW 11

- Revise `RunnableInterruptiblePrimeGenerator.java` to be thread-safe.
  - c.f. HW 10, in which you work on a thread-safe version of `RunnableCancelablePrimeGenerator.java`
- Deadline: Oct 25 (Thu) midnight

Explicit Thread Termination

- Flag-based
  - Pros:
    - Uses **1 lock (faster)**
  - Cons:
    - 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
    - Higher program responsiveness
      - interrupt() can immediately wake up a thread-to-be-terminated that is in the Waiting or Blocked state
  - Cons
    - Uses **2 locks (slower)**

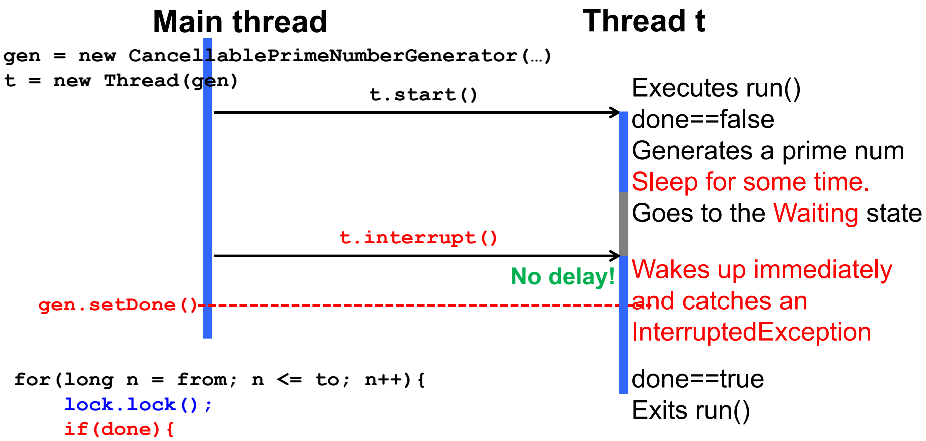
Hybridization of the Two Approaches?

- Can we implement a responsive thread termination that uses only 1 lock?

2-Step Thread Termination (“Graceful” Thread Termination)

2-Step Thread Termination

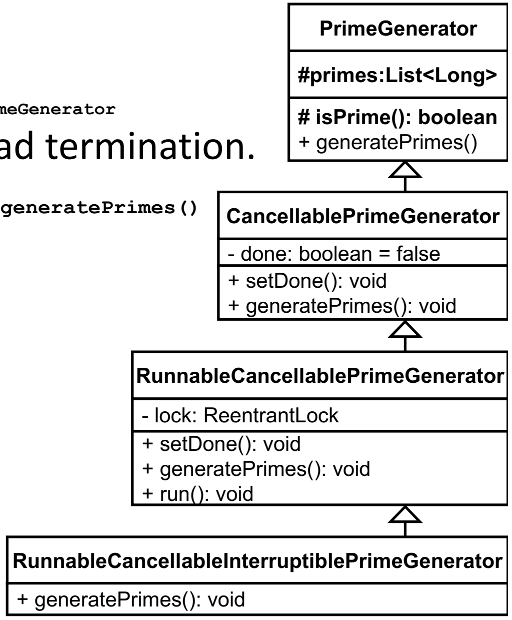
- Primarily takes the flag-based approach.
  - A thread-to-be-terminated periodically checks a flag.
- Let the "terminator" thread call `interrupt()` before flipping the flag's state (i.e., before calling `setDone()`)



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

HW 12

- Define `RunnableCancellableInterruptiblePrimeGenerator` to perform 2-step thread termination.
  - Re-define (or override) `generatePrimes()`
- Deadline: Oct 29 (Tue) midnight



2-Step Thread Termination is Effective if…

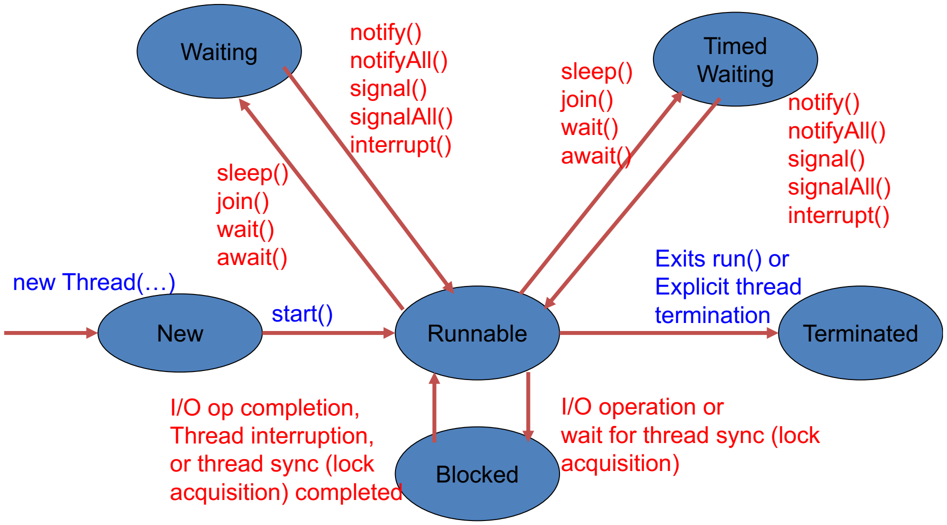
- A thread-to-be-terminated may be in the Waiting or Blocked state when a "terminator" thread tries to terminate it.
  - Performing an I/O operation.
    - e.g., reading/writing data from/to a file, waiting for an incoming data on a socket, sending data to a remote app.
  - Waiting for a lock acquisition
    - 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

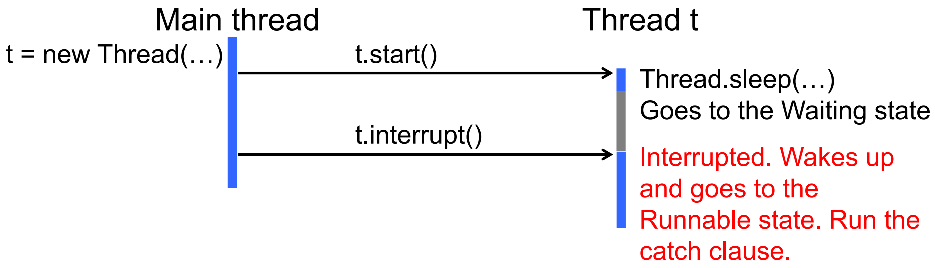


`InterruptedException`

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

`Thread.sleep()`

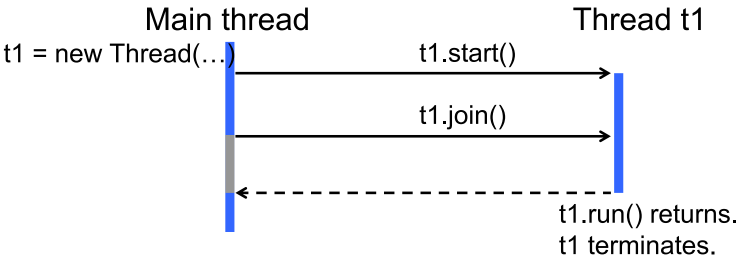
- `sleep()` lets the *currently-executed thread* to sleep for a specified time period.
- `interrupt()` interrupts a sleeping thread.
  - Wakes up the thread and force `sleep()` to throw an `InterruptedException`.
-
```
try{
     Thread.sleep(60000);
}catch(InterruptedException e){
     // Write thread termination (shutdown) logic here.
}
```



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

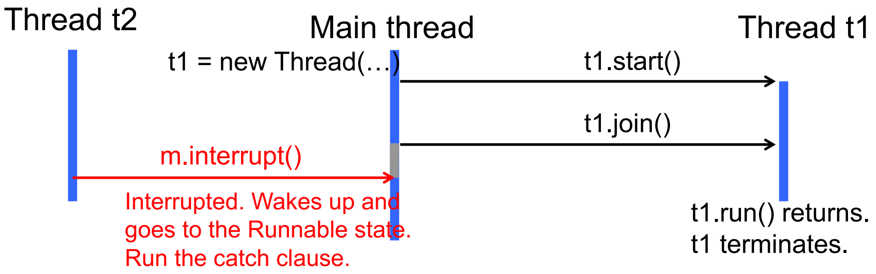
Thread.join()

- `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.join()

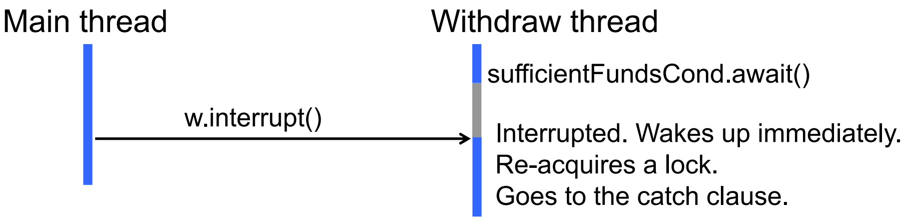
- `join()` lets the *currently-executed thread* to wait/sleep until another thread terminates (i.e., until another thread returns `run()`).
- `interrupt()` can interrupt a waiting/sleeping thread.
  - Force join() to throw an `InterruptedException`.



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`

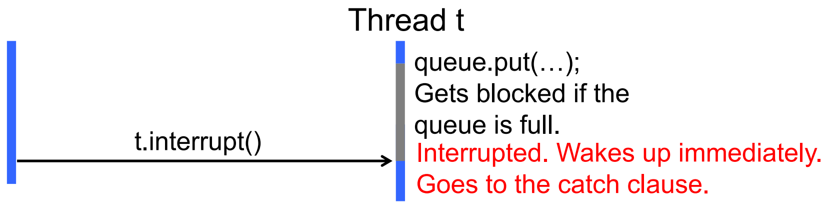
```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            //Do something
        }
    }
    belowUpperLimitFundsCondition.signalAll();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


Main thread
Withdraw thread
sufficientFundsCond.await()
w.interrupt()
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.
        }
    }
    ...}
```

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 a thread interruption by throwing an `InterruptedException`.



Thread t

t.interrupt()

queue.put(…);
Gets blocked if the
queue is full.
Interrupted. Wakes up immediately.
Goes to the catch clause.

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.

Where did the `Synchronized` Keyword go?

- Java still has the `synchronized` keyword.

  -
    ```
    public synchronized void foo(){
              // The entire method body is atomic.
    }
    ```

  -
    ```
    public void foo(){
              // non-atomic code here
              synchronized(this){
                        // atomic code here
              }
              // non-atomic code here }
    ```
  - Implicit locking
    - No need to create a `ReentrantLock` and call `lock()` and `unlock()`.
  - When a thread enters a synchronized method/block, it tries to acquire the (implicit) lock that `this` instance maintains.
    - Instance-by-instance locking
  - Code gets tricky/dirty to use multiple locks in a single class.

- Explicit locking

  -
    ```
    ReentrantLock aLock = new ReentrantLock()
    public void foo(){
         aLock.lock();
         // atomic code
         aLock.unlock(); }
    ```
  - Arbitrary locking scope.
  - Clean code even if a class uses multiple locks.
  - Extra functionalities
    - e.g., getQueueLength(): returns the # of waiting threads.
    - tryLock(): acquires a lock only if it is not held by another thread.
  - The catch is… it's VERY easy to forget calling unlock().
    - Must call unlock() in a finally clause.

- Implicit locking with the “synchronized” keyword
  - A thread can call notify() and notifyAll() even if it has not acquired a lock.
    - An IllegalMonitorStateException is thrown.

- Explicit locking
  - This error/bug never occurs.

    -
      ```
      ReentrantLock lock = new ReentrantLock();
      Condition cond = lock.newCondition();
      lock.lock();
      …
      cond.SignalAll();
      ```