Thread Interruption

Thread Interruption

- Often used for stopping/cancelling the tasks being executed by a thread and terminating the thread.
  - One of two approaches for thread termination

interrupt(), isInterrupt() and interrupted()

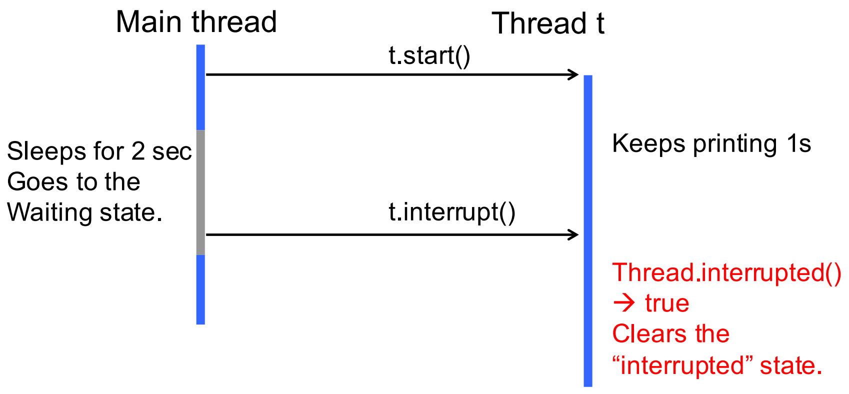
-
```
public class Thread{
  public void interrupt();
  public boolean isInterrupted();
  public static boolean interrupted();
  …
}
```

isInterrupted() and interrupted()

- isInterrupted()
  - Regular method
  - Returns true if ***"this"*** thread has been interrupted.
    -
    ```
    aThread = new Thread(…);
    aThread.start();
    aThread.isInterrupted();
    ```
  - Does not change the "interrupted" state.
- interrupted()
  - Static method (class method)
  - Returns true if the ***currently-executed*** thread has been interrupted.
  - Clears the "interrupted" state (true → false) if true is returned.

InterruptableTask2.java (c.f. lec note #3)

```
class InterruptableTask2
  implements Runnable{
  public void run(){
    while(!Thread.interrupted()){
      System.out.println(1);
    }
  }}
```

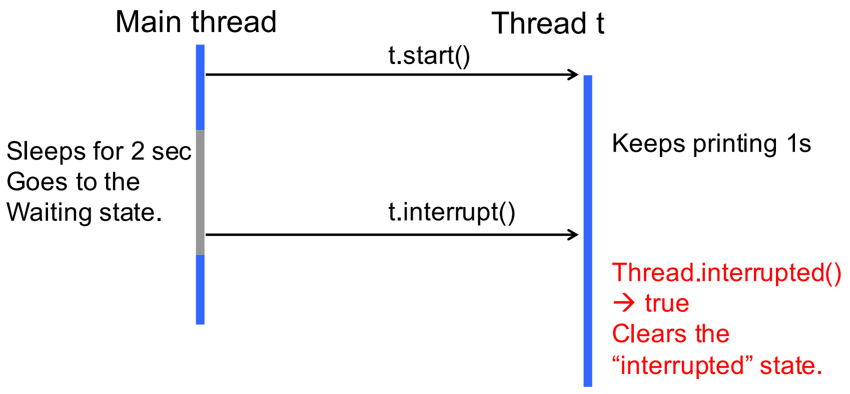


Main thread
Thread t
t.start()
Sleeps for 2 sec Goes to the Waiting state.
Keeps printing 1s
t.interrupt()
Thread.interrupted() → true Clears the "interrupted" state.

Note: DO Thread.interrupted(). DO NOT t.interrupted(). interrupted() is a static method.
Note 2: Understand the difference b/w Thread.interrupted() and t.isInterrupted().

InterruptableTask2.java (c.f. lec note #3)

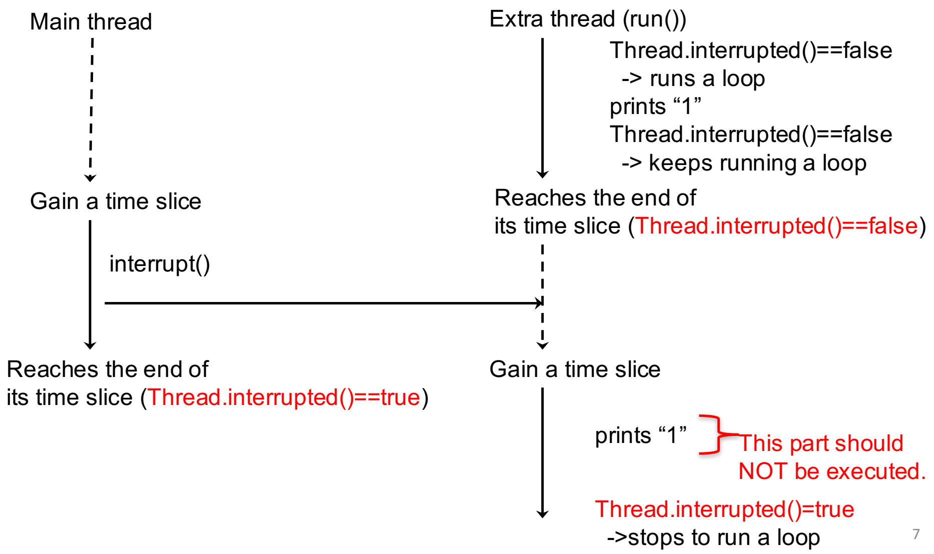
```
class InterruptableTask2
  implements Runnable{
  public void run(){
    while(!Thread.interrupted()){
      System.out.println(1);
    }
  }}
```



Main thread
Thread t
t.start()
Sleeps for 2 sec Goes to the Waiting state.
Keeps printing 1s
t.interrupt()
Thread.interrupted() → true Clears the "interrupted" state.

In fact, this code is not thread-safe… A race condition can occur.

A Potential Race Condition



Main thread
Extra thread (run())
Thread.interrupted()==false -> runs a loop
prints "1"
Thread.interrupted()==false -> keeps running a loop
Gain a time slice
Reaches the end of its time slice (Thread.interrupted()==false)
interrupt()
Reaches the end of its time slice (Thread.interrupted()==true)
Gain a time slice
prints "1"
This part should NOT be executed.
Thread.interrupted()=true ->stops to run a loop

interrupt() and interrupted()

- Thread

```
public void interrupt(){
    ...
    synchronized(...){
        ...
        interrupt0(); //native method (atomically executed)
        ...
    }
}
public static boolean interrupted(){
    return currentThread().isInterrupted(true);// native
method (atomic)
}
```

- interrupt() and interrupted() are thread-safe.
  - isInterrupted() is thread-safe as well.
- However, ***client code*** of interrupted() is not guaranteed to be thread-safe.

How to make *client code* of interrupted() thread-safe?

- Use a lock, as usual:

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

  -
    ```
    while(true){
         lock.lock();
         if(Thread.interrupted()) break; // balking
         // do something
         lock.unlock();
    }
    ```
- In a sense, there is a performance loss by using two locks.
  - One for interrupt() and interrupted()
  - One for client code of those methods.
- This is necessary if you need thread-safe code.

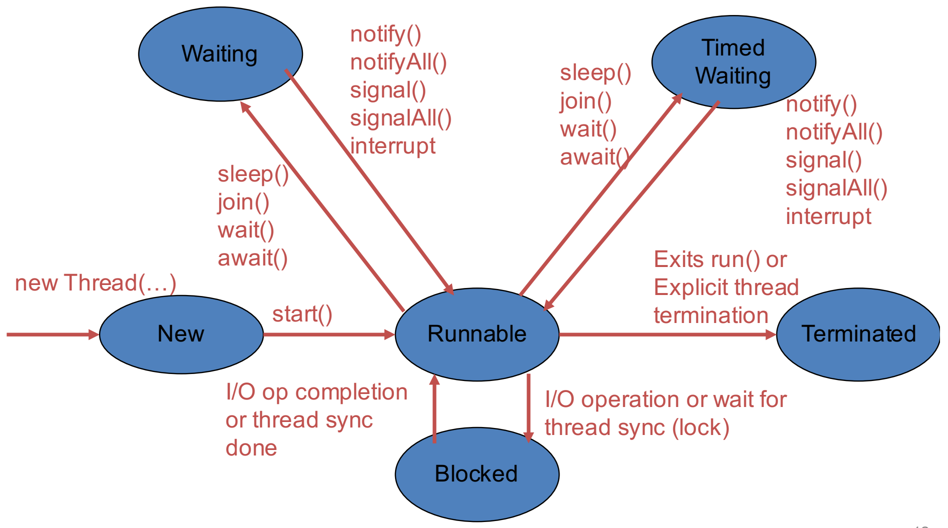
HW 15

- Revise InterruptableTask2.java to make it thread-safe.
  - Keep using thread interruption.
  - Use a ReentrantLock and balking.

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

- If the target thread is running, its "interrupted" state changes.
- If the target thread is in the *Waiting* or *Blocked* state, it raises an `InterruptedException`.
  - c.f. SummationRunnableInterruptable.java (lecture note #3)
    - interrupt() is called for a thread that is in the Waiting state due to join().

States of a Thread

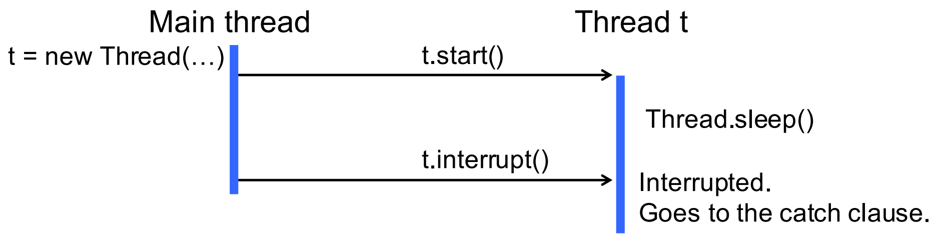


InterruptedException

- Some methods in Java API can throw InterruptedException.
  - Thread.sleep()
  - Thread.join()
  - Condition.await()
  - ReentrantLock.tryLock()
  - BlockingQueue.put()/take()
  - These methods can be long-running and cancellable.
  - Clears the "interrupted" state.

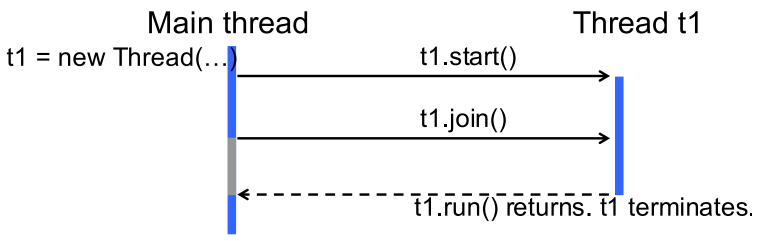
Thread.sleep()

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



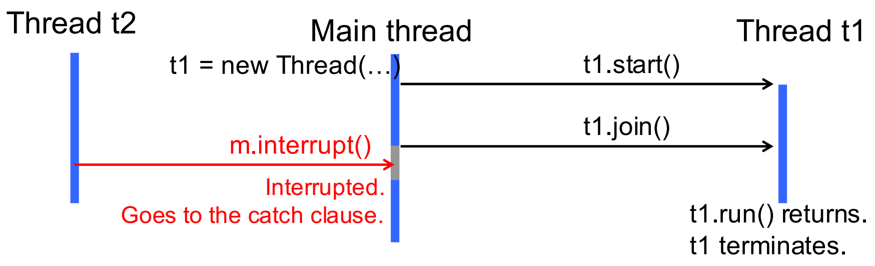
Thread.join()

- 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()

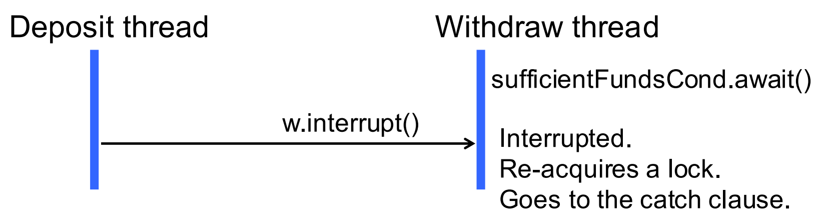
- join() lets the currently-executed thread to wait/sleep until another thread terminates (i.e., returns run()).
- interrupt() can interrupt a waiting/sleeping thread.
  - The waiting/sleeping throws an InterruptedException



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

- put() and take() are blocking methods.
  - put(): Add an element to a queue.
  - take(): get the first element in the queue.
- They can respond to an interruption by throwing an InterruptedException.
  - BlockingQueue aQueue = new BlockingQueue();
  - aQueue.put(generatePrimeNumber());

interrupt() Never Terminate a Thread.

-
  ```
  class TestRunnable implements Runnable{
      public void run(){
          while(true){
              System.out.println(“running”);
          }
      } }
  ```

-
  ```
  main(){
      Thread t = new Thread(new TestRunnable());
      t.start();
      t.interrupt(); }
  ```
- run() never use blocking methods. interrupt() is called on *t* when it is in the Running state.
  - interrupt() just changes *t*’s “interrupted” state from false to true.
  - The main thread never kill/terminate *t*.
- interrupt() never kill/terminate a thread. It can be used to do that though.

Thread Termination can be Tricky.

- Thread creation is a no brainer.
- Thread termination requires your careful attention.
  - No methods available in Thread to terminate threads.
    - Do:
      - Flag-based approach
      - Interruption-based approach
  - 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?
      - How to handle the current data/state maintained by a thread being terminated?
    - Java allows you to flexibly craft your own termination logic.

Deprecated Methods for Thread Termination

- Thread.stop() and Thread.suspend()
  - Not thread-safe. Never use them.
  - http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html