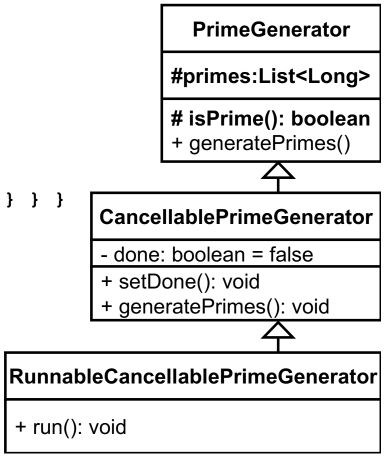
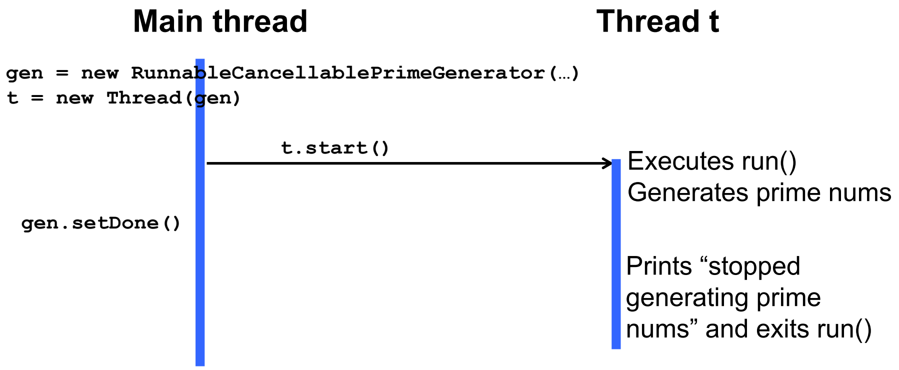
RunnableCancellablePrimeGenerator

```
class CancellablePrimeGenerator extends PrimeGenerator {
  private boolean done = false;

  public void setDone(){
    done = true; }

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

class RunnableCancellablePrimeGenerator
  extends CancellablePrimeGenerator
  implements Runnable {

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



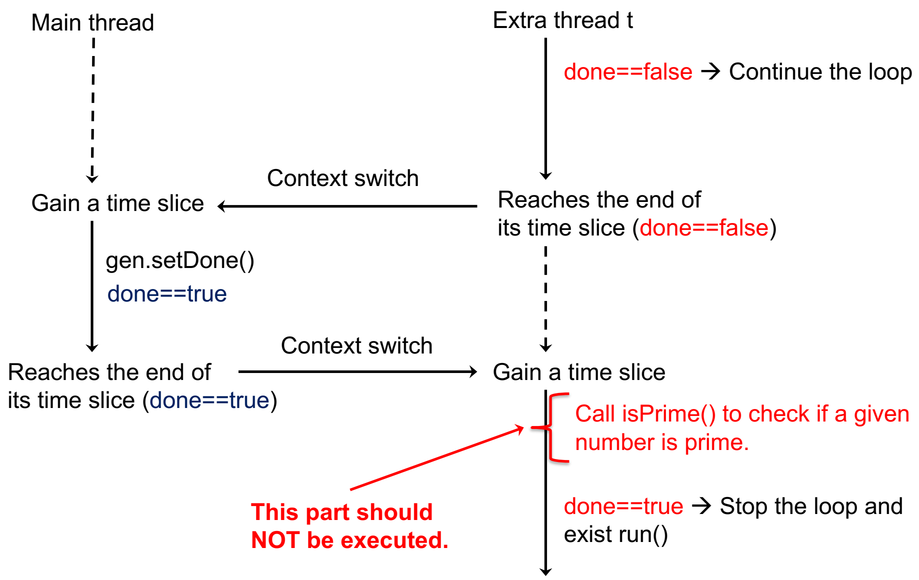


```
gen = new RunnableCancellablePrimeGenerator(…)
t = new Thread(gen)
```

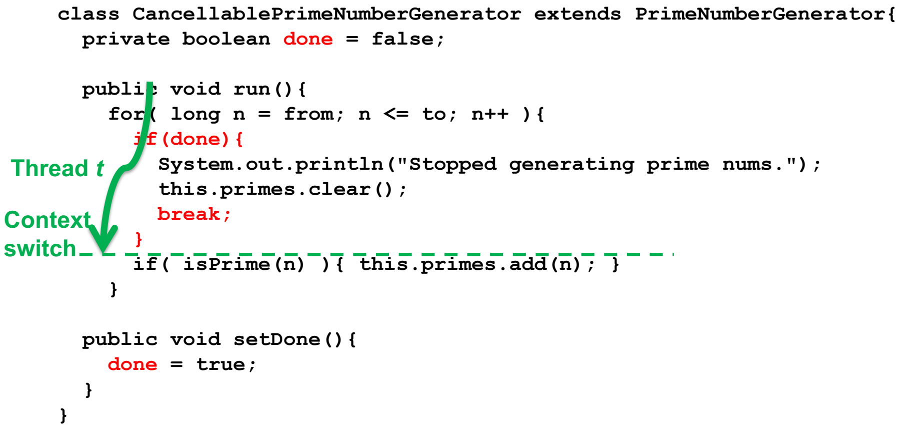
```
for(long n = from; n <= to; n++){
    if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
    }
    if( isPrime(n) ){ this.primes.add(n); }
}
```

- This code is actually NOT thread-safe. Race conditions can occur.
  - Thread-safe code is free from
    - Race conditions
    - Deadlocks

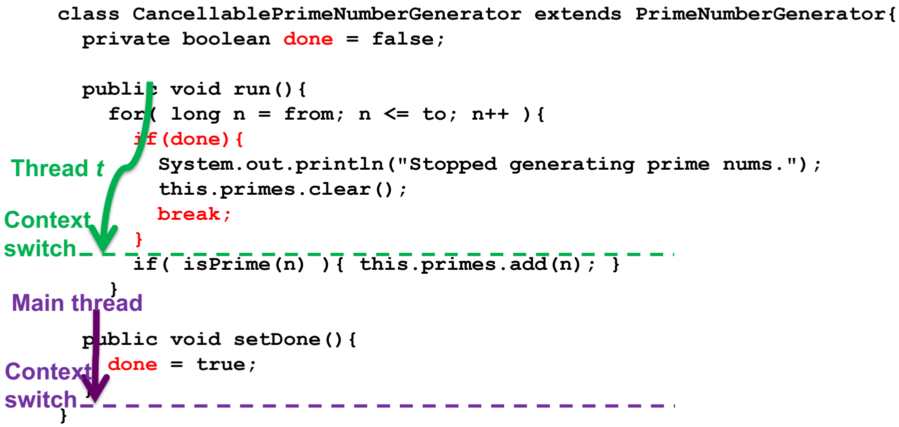
A Potential Race Condition



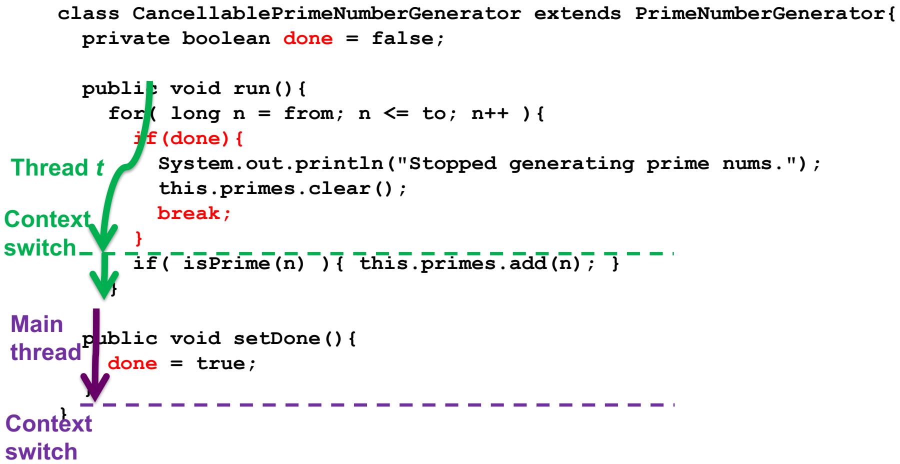
Main thread
Extra thread t
done==false → Continue the loop
Context switch
Gain a time slice
Reaches the end of its time slice (done==false)
gen.setDone()
done==true
Context switch
Reaches the end of its time slice (done==true)
Gain a time slice
Call isPrime() to check if a given number is prime.
This part should NOT be executed.
done==true → Stop the loop and exist run()



```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

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

  public void setDone(){
    done = true;
  }
}
```



```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

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

  public void setDone(){
    done = true;
  }
}
```



```
class CancellablePrimeNumberGenerator extends PrimeNumberGenerator{
  private boolean done = false;

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

  public void setDone(){
    done = true;
  }
}
```

Visibility Issue

- The current (most up-to-date) value of the shared variable “done” is not *visible* for all threads.

- Solution:
  - Identify all read and write logic on the shared variable “done”
  - Surround each read/write logic with lock() and unlock() invocations on the same **ReentrantLock**

Solution: Locking and Balking

- A General form of the “balking” idiom

  -
    ```
    booblean done = false;
    ReentrantLock lock = new ReentrantLock();
    ...
    while(true){
      lock.lock();
      try{
        if(done) break; // Balk
        ...             // Do some task
      }finally{
        lock.unlock();
      }
      ...
    }
    ```

  -
    ```
    void setDone(){
       lock.lock();
      try{
        done = true;
      }finally{
        lock.unlock();
      }
    }
    ```
- Threads must use the same instance of `ReentrantLock`.

Be Careful for Potential Race Conditions

- When multiple threads share and access a variable concurrently.
  - Make sure to guard the shared variable
    - By surrounding each read/write logic with lock() and unlock().
- When a loop performs a conditional check with a shared variable (i.e., flag).
  - Surround read logic (i.e., conditional) and write logic (i.e., flag-flipping statement) with lock() and unlock()
  - Try NOT to surround the entire loop with with lock() and unlock()! Why?
    - May result in a deadlock.
    - Does not enjoy concurrency.

Treating the Entire Loop as Atomic Code May Result in a Deadlock

- DO **NOT** do this.

-
  ```
  try{
    lock.lock();
    while(!done){
      // Do some task
    }
  }finally{
    lock.unlock();
  }
  ```

-
  ```
  lock.lock();
  try{
    done = true;
  }finally{
    lock.unlock();
  }
  ```

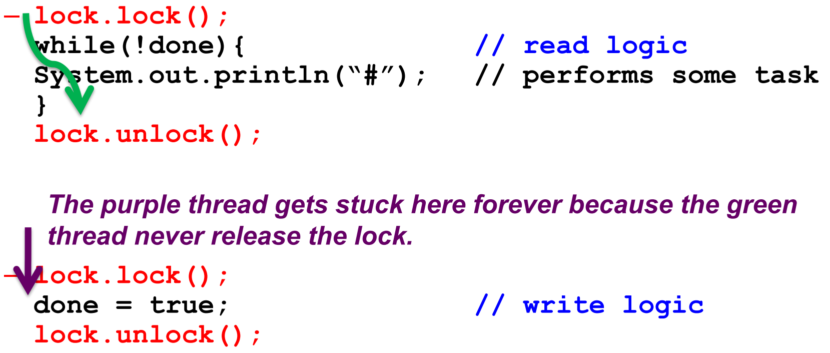
- Do this.

-
  ```
  while(true){
    lock.lock();
    try{
      if(done) break; // Balk
      // Do some task
    }finally{
      lock.unlock();
    }
  }
  ```

-
  ```
  lock.lock();
  try{
    done = true;
  }finally{
    lock.unlock();
  }
  ```

Treating the Entire Loop as Atomic Code May Result in a Deadlock

- If a thread acquires the lock and starts printing #s, it will print #s forever.
  - No other threads cannot flip the flag forever (deadlock!)



```
– lock.lock();
  while(!done){                  // read logic
  System.out.println(“#”);       // performs some task
  }
  lock.unlock();
```

The purple thread gets stuck here forever because the green thread never release the lock.

```
– lock.lock();
  done = true;                   // write logic
  lock.unlock();
```

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- DO **NOT** do this.

```
long n;
lock.lock();
try{
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
  lock.unlock();
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

- Do this.

```
long n;
for(n = from; n <= to; n++){
  lock.lock();
  try{
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }finally{
    lock.unlock(); }
}
```

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- If a thread acquires the lock and starts generating prime numbers, it will release the lock when *n > to*.
  - No other threads cannot flip the flag until the loop ends.
  - No deadlock occurs because the loop ends when *n > to*.

```
try{
  lock.lock();
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
   lock.unlock();
}
```

The purple thread can acquire the lock after all prime numbers have been generated.

```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- This code is thread-safe, but it does not enjoy concurrency.
  - While the green thread generates prime numbers for a given range in between "from" and "to," the purple thread cannot stop the green thread.

```
try{
  lock.lock();
  for(n = from; n <= to; n++){
    if(done==true) break;
    if(isPrime(n)){
      this.primes.add(n); }
  }
}finally{
   lock.unlock();
}
```

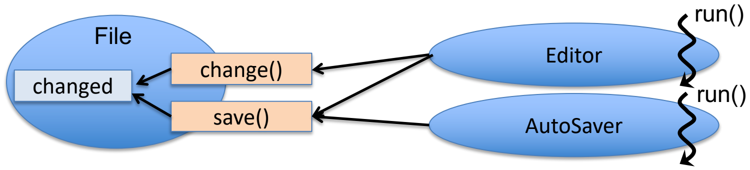
lock() returns when the green thread releases the lock.

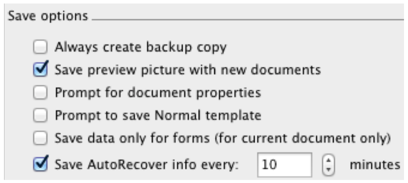
```
lock.lock();
try{
  done = true;
}finally{
  lock.unlock();
}
```

HW 8

- Revise `RunnableCancellablePrimeGenerator.java` to be thread-safe.
  - Use a `ReentrantLock` to guard the shared variable `done`
  - Use try-finally blocks.
    - Call unlock() in a finally block. Always do this in all subsequent HWs.
  - Use *balking* to implement explicit thread termination in a thread-safe manner
  - Do not surround the entire "for" loop with lock() and unlock().
- Deadline: Oct 23 (Tue) midnight

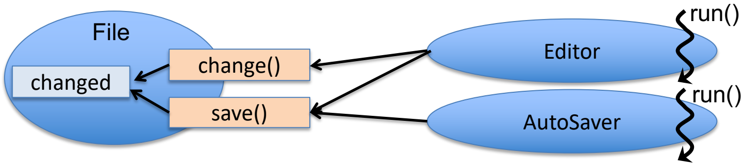
Exercise: Concurrent Access to a File




Save options

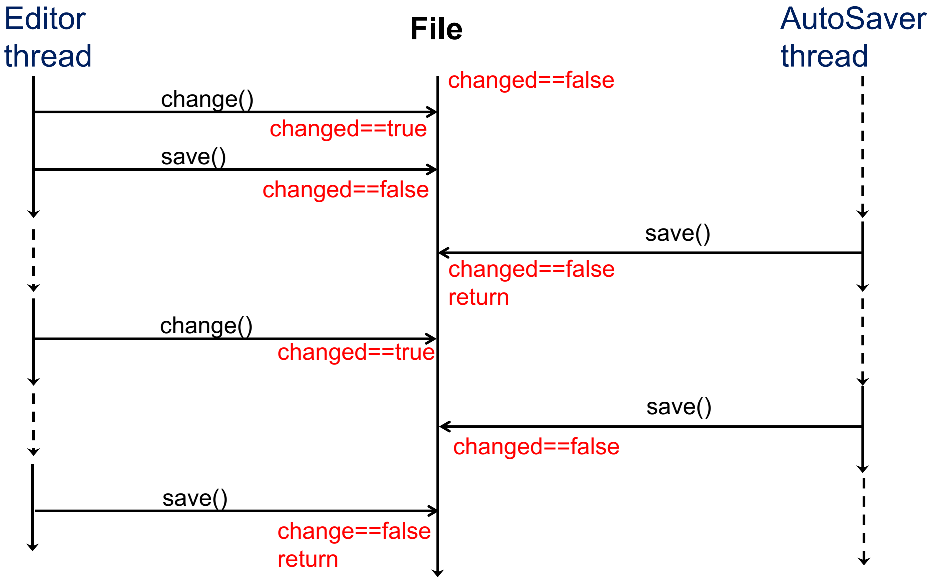
- ☐ Always create backup copy
- ☑ Save preview picture with new documents
- ☐ Prompt for document properties
- ☐ Prompt to save Normal template
- ☐ Save data only for forms (for current document only)
- ☑ Save AutoRecover info every: 10 minutes

- Imagine a word processing app.
- Assume two threads
  - One for editing a file
    - Allows the user to edit a file and save it.
  - One for saving a file automatically
    - Periodically saves an open file at background.
- The 2 threads call `change()` and `save()` on an open file concurrently.



- File
  - Has a boolean variable: "changed"
    - Initialized to be false.
  - change()
    - Changes the file's content.
    - Assigns true to the variable "changed."
  - save()
    - if(!changed) return;
    - if(changed)
      - print out some message (e.g., time stamp, etc.)
      - assigns false to the variable "changed."
- Editor (a Runnable) repeats:
  - Calls change() and save()
  - Sleeps for a second.
- AutoSaver (a Runnable) repeats:
  - Calls save()
  - Sleeps for two seconds.

Desirable Result



HW 9

- Race conditions can occur if you do not guard the variable `changed` with a lock. Explain a potential race condition with a diagram like in the previous slide.
- Implement `File`, `Editor` and `AutoSaver` in a thread-safe manner
  - Define a `ReentrantLock` in File. Use the lock in `change()` and `save()`
    - c.f. `deposit()` and `withdraw()`, which use a lock to access a shared variable in the bank account example
    - Use try-finally blocks: Always do this in all subsequent HWs.
  - Create two extra threads and have them execute `Editor`'s `run()` and `AutoSaver`'s `run()`
    - Those threads acquire and release the lock in `change()` and `save()`

- Implement explicit thread termination in **`Editor`** and **`AutoSaver`** to terminate 2 extra threads.

- Have the main thread sleep for some time while **`Editor`** and **`AutoSaver`** are running.
  - Use **`Thread.sleep()`**

- Have the main thread terminate the two threads.
  - Define a flag variable **`done`** and **`setDone()`** in **`Editor`** and **`AutoSaver`**

```
class Editor implements Runnable{
  private boolean done = false;

  public void run(){
    while(true){
      if(done){
        System.out.println(”…");
        break;
      }
      aFile.change();
      aFile.save();
      Thread.sleep(1000);
    }

  public void setDone(){
    done = true; }  }
```

- Note that this sample code is not thread-safe.
  - Define a **`ReentrantLock`** in each of **`Editor`** and **`AutoSaver`** to guard a flag variable **`done`**.
    - Use try-finally blocks
    - Use balking in run()
      - Do not surround a “while” loop with lock() and unlock().

- Deadline: Oct 23 (Tue) midnight