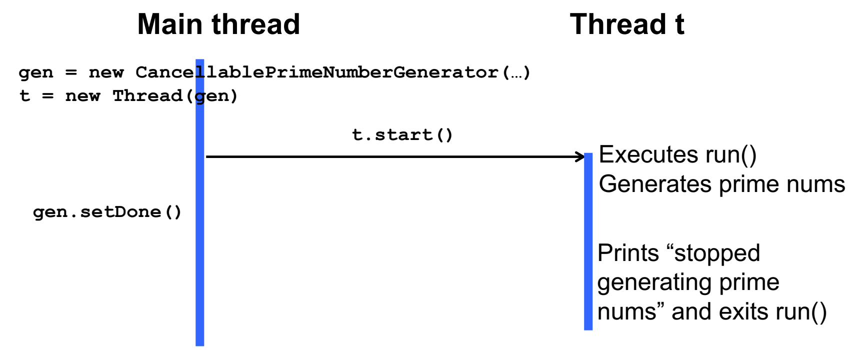
CancellablePrimeNumberGenerator

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

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

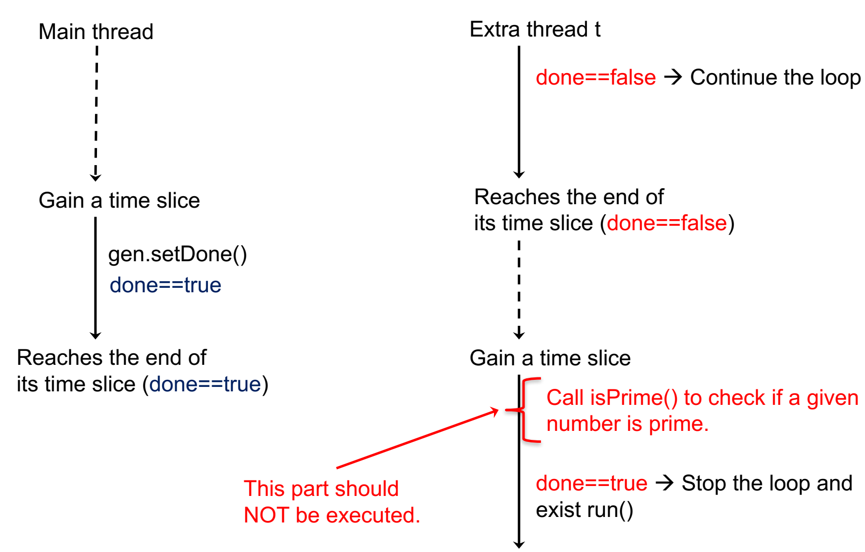
  public void setDone(){
    done = true;
  }
}
```


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


- Not thread-safe. Race conditions can occur.
  - Thread safety:
    - No race conditions
    - No deadlocks

A Potential Race Condition



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

  public void run(){
    for( long n = from; n <= to; n++ ){
      if(done==true){
        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==true){
        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==true){
        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.

- `ReentrantLock` allows your code to preserve ***atomicity AND visibility***.

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 that a shared variable is guarded with a lock.
    - Surround reading and writing parts with lock() and unlock().
      - Reading/writing parts → atomic code
      - c.f. Thread-unsafe and -safe bank accounts
- When a loop performs a conditional check with a shared variable (i.e., flag).
  - Surround reading part (i.e., conditional block) and writing part (i.e., flag-flipping part) with lock() and unlock()
    - Reading/writing parts → atomic code
  - Try NOT to surround the entire loop with 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
      System.out.println("#");
    }
  }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
      System.out.println("#");
    }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){               // reading part
     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;                // writing part
  lock.unlock();
```

Treating the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- DO NOT do this.

```
• long n;
  try{
    lock.lock();
    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++){
    try{
      lock.lock();
      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();
  }
```

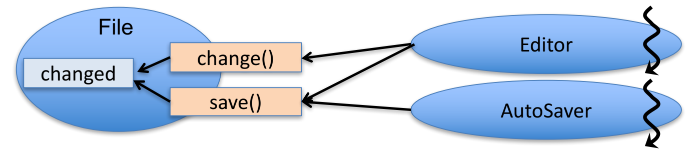
lock() returns when the green thread releases the lock.

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

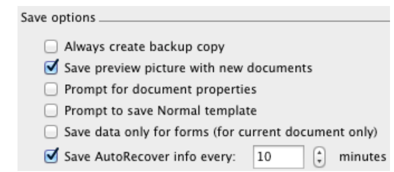
HW 7

- Revise CancellablePrimeNumberGenerator.java to make it thread-safe.
  – Use ReentrantLock
  – Use try-finally blocks.
    - Call unlock() in a finally block. <u>Always do this in all subsequent HWs.</u>
  – Use balking
  – Do not surround the "for" loop with lock() and unlock().

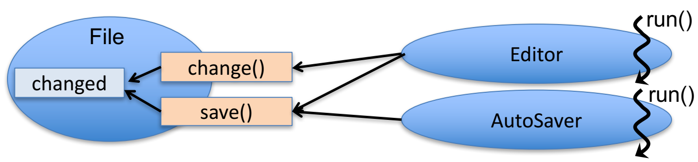
Exercise: Concurrent Access to a File



- Imagine word processing software.

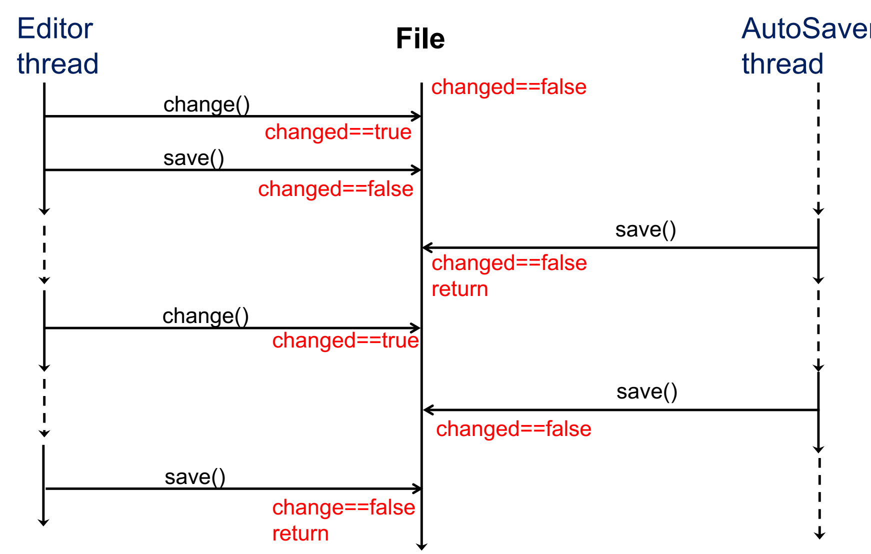


- Assume two threads
  – One for a file editing feature
    - Allows the user to edit a file and save it.
  – One for an automatic file saving feature
    - Periodically saves an open file at background.
- The 2 threads call `change()` and `save()` on an open file concurrently.



- 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==false) return;
    - if(changed==true)
      – 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 8

- Race conditions can occur if you do not guard the "changed" variable with a lock. Explain a potential race condition with a diagram like in the previous slide.
- Submit thread-safe code.
  - Define 3 classes: **`File`**, **`Editor`** and **`AutoSaver`**
  - Define a lock in File. Use the lock in **`change()`** and **`save()`**
    - 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 via `change()` and `save()`.

- Have the main thread sleep for some time while Editor and AutoSaver are running.
  - Use Thread.sleep()
- Have the main thread terminate the other 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==true){
        System.out.println("...");
        break;
      }
      aFile.change();
      aFile.save();
      Thread.sleep(1000);
    }

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

- Note that this code is not thread-safe.
  - c.f. CancellablePrimeNumberGenerator.java
- Use a lock in Editor and AutoSaver to guard flag variables.
  - Use try-finally blocks
  - Use balking in run()
    - Do not surround "while" loops with lock() and unlock().

Recap: Singleton Design Pattern

- Guarantee that a class has only one instance.

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

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

HW 9

- 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 where more than one instances are created.
    - Use a diagram like in a previous slide.
- Submit a thread-safe version of Singleton.
  - Define a lock in Singleton. Use the lock in getInstance()
    - Use try-finally blocks: Always do this in all subsequent HWs.
  - Create multiple extra threads and have them call getInstance()
    - Make sure that only one instance is created.
      - Use System.out.println(Singleton.getInstance())

Concurrent Singleton Design Pattern

- Guarantee that a class has only one instance.

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

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


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();
   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 exclusive execution) for the two threads
  - x.a() and x.b(): no synchronization
  - x.a() and x.syncA(): no synchronization
  - x.syncA() and x.syncA(): Synchronization (exclusive execution)
  - 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();} }
```

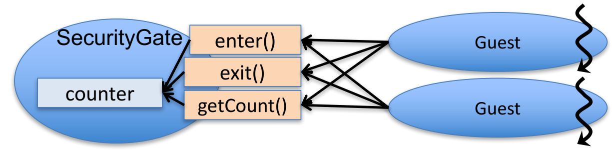
- `x = new Foo(); y = new Foo();`
- Two threads call…
  - x.a() and Foo.sA(): No synchronization for the two threads
  - x.syncA() and Foo.sA(): No synchronization
  - Foo.sA() and Foo.sA(): No synchronization
  - Foo.sA() and Foo.sB(): No synchronization
  - x.syncA() and Foo.sSyncA() No symchronization
  - 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 8.

Another Exercise




```
class SecurityGate{
  private int counter = 0;

  public void enter(){
    counter++;
  }

  public void exit(){
    counter--;
  }

  //Get the # of guests
  //in the gate
  public int getCount(){
    return counter;
  }
}
```

```
class Guest implements Runnable{
  private SecurityGate gate;

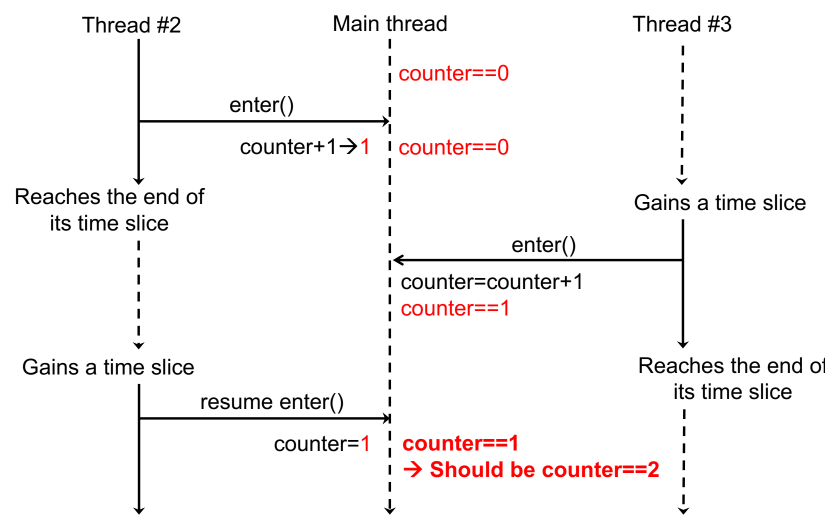
  public Guest(……){
   gate = SecurityGate.getInstance();
  }

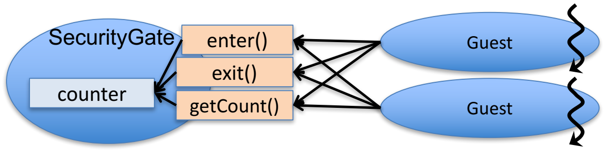
  public void run(){
    gate.enter();
    gate.getCount();
    gate.exit();
  }
}
```

Not Thread Safe!

- `SecurityGate` is not thread-safe due to potential race conditions.
- `couter++` is a compound operation and not atomic.
  - Syntactic sugar for `counter = counter + 1;`
  - Requires multiple atomic operations
    - A context switch can occur *across* different atomic ops.
- The same goes to `counter--`.

A Potential Race Condition






```
class SecurityGate{
  private int counter = 0;
  private ReentrantLock lock;

  public void enter(){
    lock.lock();
    counter++;
    lock.unlock();
  }

  public void exit(){
    lock.lock();
    counter--;
    lock.unlock();
  }

  public int getCount(){
    return counter;
  }}
```

```
class Guest implements Runnable{
  private SecurityGate gate;

  public Guest(……){
   gate = SecurityGate.getInstance();
  }

  public void run(){
      gate.enter();
      gate.exit();
      gate.getCount();
  }
}
```

HW 10-1

- Make `SecurityGate` thread-safe.
  - Revise `SecurityGate` to be a concurrent singleton class.
  - Use two locks in `SecurityGate`
    - One to guard `counter`
      - Use it in `enter()` and `exit()`
    - One to guard `instance`
      - Use it in `getInstance()`
- Create many threads to execute `Guest` instances' `run()` and have them call `enter()`, `exit()` and `getCount()`.

An Alternative Solution: Use `AtomicInteger`

- Offers a series of thread-safe methods to manipulate an integer value ***atomically***.
  - ```
    int i;
    i = i + 1;                    // Not thread-safe
    ```
  - ```
    AtomicInteger atomicInt = new AtomicInteger(0);
    atomicInt.incrementAndGet();  // Thread-safe
    ```
  - `decrementAndGet(), addAndGet(int), set(), get(), getAndSet(int), compareAndSet(int, int)...`
  - `updateAndGet(IntUnaryOperator), accumulateAndGet(int, IntBinaryOperator)`
  - Many of the methods do not use locking; they are faster than lock-based code.

Be Careful!

- ```
  int i = ...
  i = i + 1;                              // Not thread-safe
  ```
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  atomicInt.incrementAndGet();            // Thread-safe
  ```
  - This is atomic.
- ```
  AtomicInteger atomicInt = new AtomicInteger(0);
  int i = atomicInt.incrementAndGet();  // NOT thread-safe
  ```
  - This is compound!
    - `incrementAndGet()` is atomic though.
  ```

AtomicInteger

– **updateAndGet(IntUnaryOperator updateFunction)**
  - `IntUnaryOperator`: a general-purpose functional interface
  - Atomically updates the current integer value with the result of applying the given function, returning the updated value.
  - `updateFunction`: a lambda expression that takes the current int value as a parameter, updates it and returns the updated value.
  -
```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)->++i) );    // 11. Thread safe
atomicInt.incrementAndGet();                // 12. Thread safe
```

	Params	Returns	Example use case
UnaryOperator<T>	T	T	Logical NOT (!)

-
```
AtomicInteger atomicInt = new AtomicInteger(10);
atomicInt.updateAndGet( (int i)->++i) );   // 11. Thread safe
```
- Why ++1? Just in case, note that:
  -
```
int i = 0;
i++;         // i==1
```
  -
```
int i = 0;
int x = i++; // i==1, x==0
```
  -
```
int i = 0;
int y = ++i; // i==1, y==1
```

– **accumulateAndGet(int, IntBinaryOperator)**
  - `IntBinaryOperator`: a general-purpose functional interface
  - Atomically updates the current integer value with the result of applying the given function, returning the updated value.
  -
```
atomicInt.accumulateAndGet(initValue,
                           (result, currentVal)-> ... );
```
  -
```
int result = initValue;
result = accumulate(result, currentVal);
```
  - Takes a lambda expression as the second parameter.
    - The body of `accumulate()` is expressed in the LE.
    - c.f. `Stream.reduce()`

	Params	Returns	Example use case
BinaryOperator<T>	T, T	T	Multiplying two numbers (*)

-
```
AtomicInteger atomicInt = new AtomicInteger(0);
atomicInt.accumulateAndGet(10,
                           (result, currentVal)->
                               (currentVal+result)/2 );
```
-
```
AtomicInteger atomicInt = new AtomicInteger(0);
atomicInt.accumulateAndGet(
    10, (result, currentVal)->{
        if(currentVal >= result) return currentVal;
        else if return result; }));
```
-
```
int currentVal = 0; int result = 10;
result = accumulate(result, currentVal);
```

Again…

- `AtomicInteger atomicInt = new AtomicInteger(0);`
```
atomicInt.accumulateAndGet(10,
                           (result, currentVal)->
                              (currentVal+result)/2 );
```
  – This is atomic.

- `AtomicInteger atomicInt = new AtomicInteger(0);`
```
int average = atomicInt.accumulateAndGet(
    10, (result, currentVal)->(currentVal+result)/2 );
```
  – This is compound!
    - The methods of `AtomicInteger` are atomic though.

`java.util.concurrent.atomic` Package

- `AtomicBoolean`
- `AtomicInteger`
- `AtomicIntegerArray`
- `AtomicLong`
- `AtomicLongArray`
- `AtomicReference<V>`
- `AtomicReferenceArray<E>`
- `DoubleAccumulator`
- `DoubleAdder`
- `LongAccumulator`
- `LongAdder`
- …

HW 10-2

- Define the "counter" variable with AtomicInteger.
- Increment and decrement the "counter" variable with AtomicInteger.updateAndGet()
  – Do NOT use incrementAndGet() and decrementAndGet().
- Create many threads to execute Guest instances' run() and have them call enter(), exit() and getCount().

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();
```