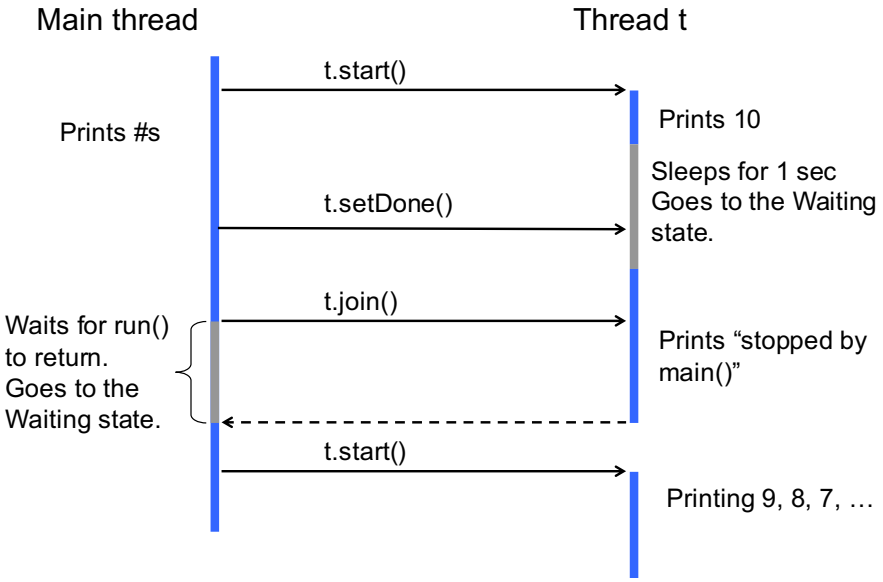
Race Conditions and Locking

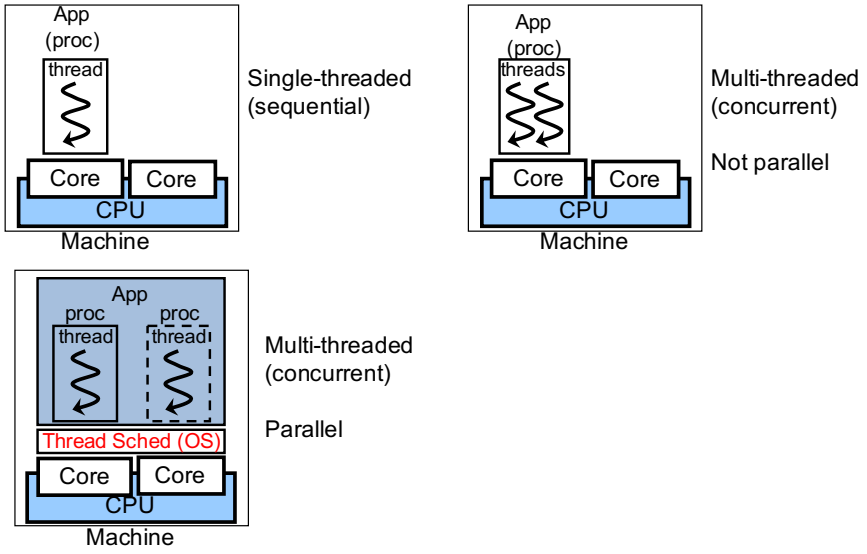
Race Conditions

- Threads run *independently*.
  - No coordination among threads by default.
    - c.f. join()
- They can share object/data in the same process.
- They can mess up the consistency of shared object/data.
  - A thread can write data into a variable when another thread is trying to read data from the variable.
  - A thread can write data to a variable when another thread is trying to write different data to the variable.

join()

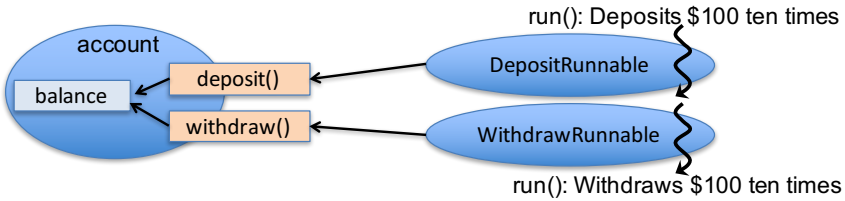


Single- and Multi-threaded Programs



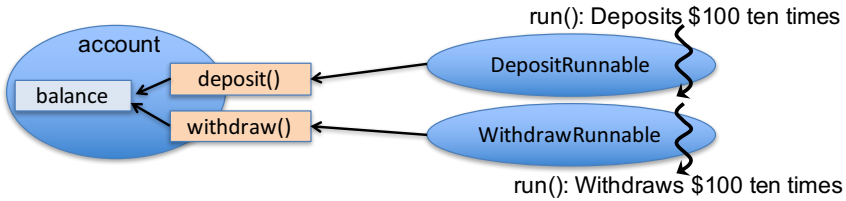
An Example Race Condition:

ThreadUnsafeBank.java



- The variable "balance" is shared by three threads.
- Two of them access it independently.

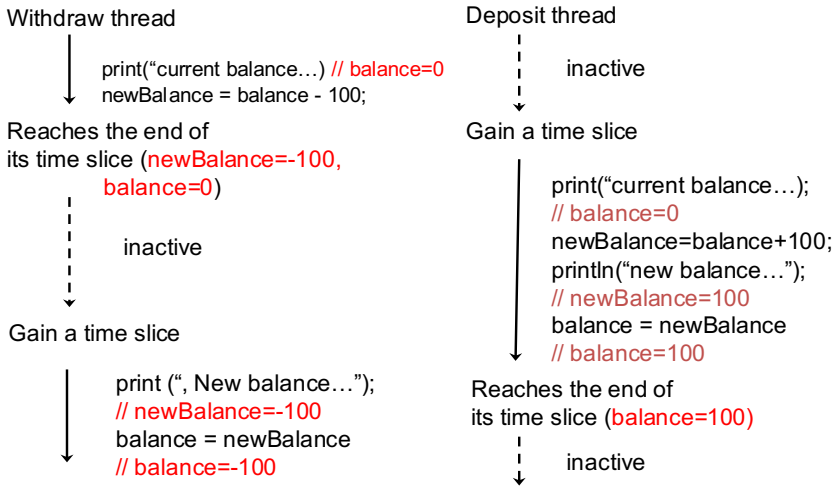
```
public void deposit(double amount){
     System.out.print("Current balance (d): " + balance);
     double newBalance = balance + amount;
     System.out.println(", New balance (d): " + newBalance);
     balance = newBalance;
}
public void withdraw(double amount){
     System.out.print("Current balance (w): " + balance);
     double newBalance = balance - amount;
     System.out.println(", New balance (w): " + newBalance);
     balance = newBalance;
}
```

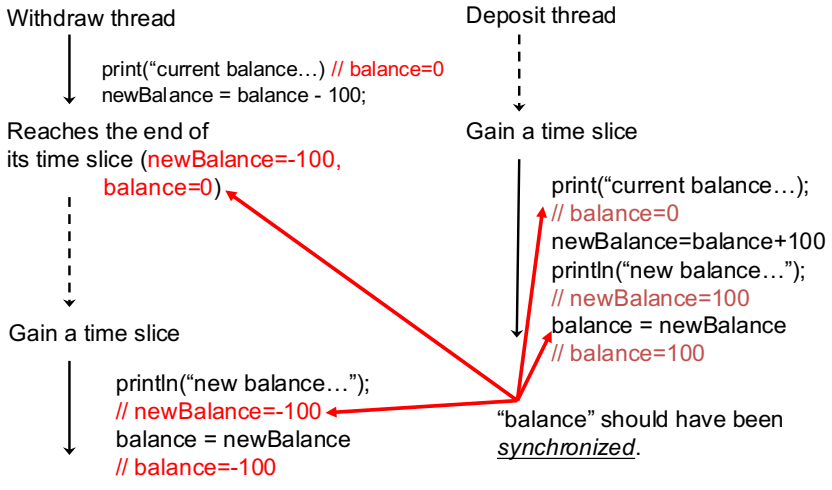
- Desirable output:
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (d): 100.0, New balance (d): 200.0
  - Current balance (w): 200.0, New balance (w): 100.0
  - ……
- In reality:
  - Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0
  - , New balance (w): -100.0

How Can This Occur?

- Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0
- , New balance (w): -100.0



- Current balance (w): 0.0Current balance (d): 0.0, New balance (d): 100.0
- , New balance (w): -100.0



"Visibility" Issue

- The current (most up-to-date) value of the shared variable "balance" is not *visible* (synchronized) for all threads.

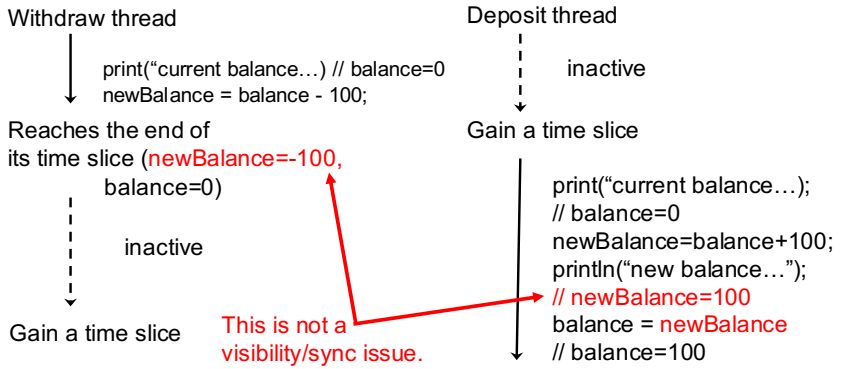
- In other words, the visibility issue never occur from a local variable.
- Focus on other variables regarding the visibility issue.

- Two other example local variables

```
– public void deposit(double amount){
        System.out.print("Current balance (d): " + balance);
        double newBalance = balance + amount;
        System.out.println(", New balance (d): " + newBalance);
        balance = newBalance;
  }
– public void withdraw(double amount){
        System.out.print("Current balance (w): " + balance);
        double newBalance = balance - amount;
        System.out.println(", New balance (w): " + newBalance);
        balance = newBalance;
  }
```

Note: Local Variables

- A local variable is NOT shared by multiple threads.
  - It is maintained on a thread-by-thread manner.
    - The "withdraw" thread has no access to a value of newBalance that the "deposit" thread has created.
    - The "deposit" thread has no access to a value of newBalance that the "withdraw" thread has created.



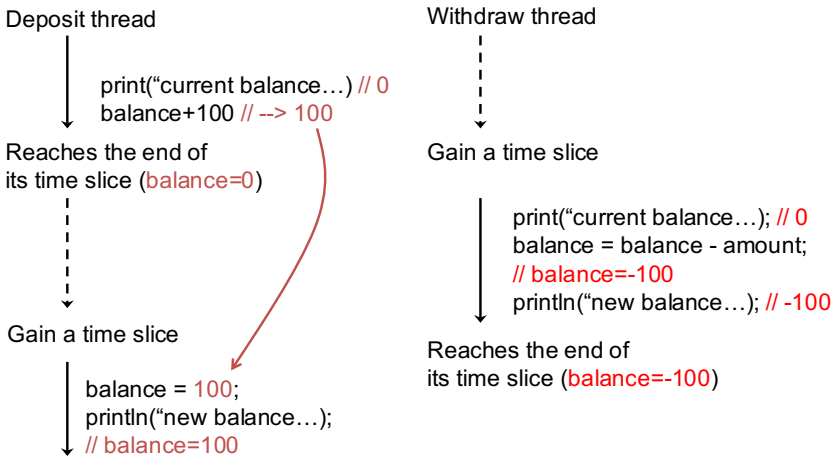
Another Example:

ThreadUnsafeBankAccount2

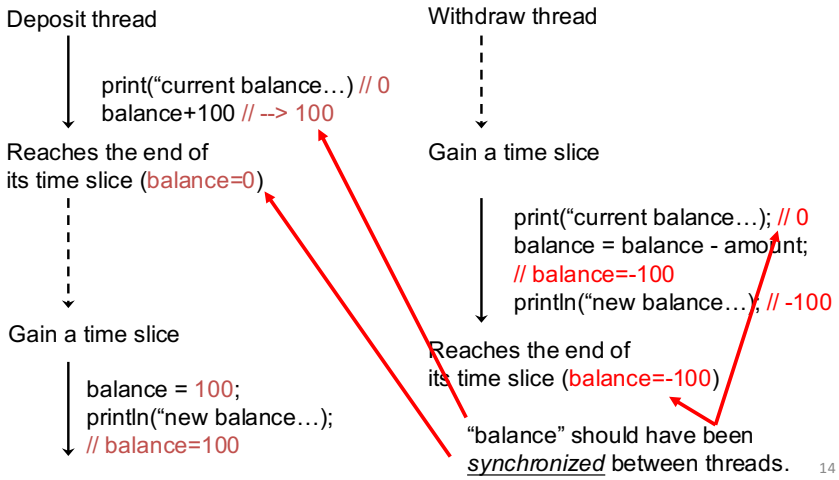
- Local variables (newBalance) are removed from ThreadUnsafeBankAccount
  - `double newBalance = balance - amount;`
    `balance = newBalance;`
  - `balance = balance - amount;`

- Output
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0, New balance (d): 100.0
  - Current balance (w): 100.0, New balance (w): 0.0
  - Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0
  - , New balance (d): 100.0

- Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0
- , New balance (d): 100.0



- Current balance (d): 0.0Current balance (w): 0.0, New balance (w): -100.0
- , New balance (d): 100.0



- This is not a solution: `balance -= amount;`
  - Just a syntactic sugar for `balance = balance - amount;`
- All threads
  - Run in their *race* to complete their tasks.
  - Manipulate a shared object/data independently.
- The end result depends on which of them happens to win the race.
  - No guarantees on the order of thread execution.
  - No guarantees on how many tasks a thread can perform in a single CPU time slice.
  - No guarantees on the end result on shared data.

Thread Synchronization

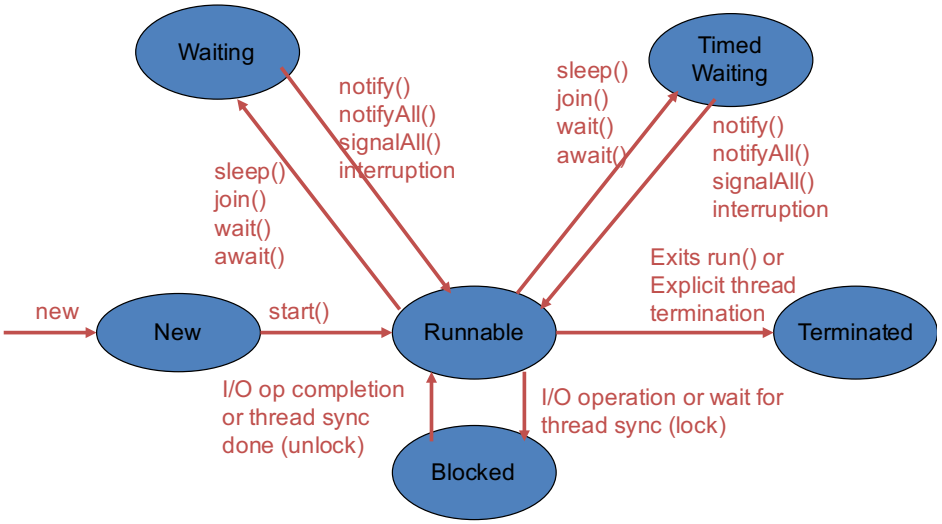
- Need to *synchronize* threads
  - i.e., Need to *serialize* their concurrent access to shared data
  - Serialized/atomic access to shared data
    - Atom: the smallest possible unit of matter; unable to be broken into separate parts
    - *Atomic code*: When a thread is running it, no other threads can run it. No intermediate result/state can be revealed/exposed to other threads.
      - atomic {
        a = 0;
        a = a + 3
        }
      - a.k.a. critical section

Lock

- Used to synchronize/serialize the threads that want to manipulate shared data.
- `java.util.concurrent.locks.Lock` interface
  - `ReentrantLock` class: the most commonly-used class for locking
    - Defines methods to access shared data in a synchronized/serialized way.
- Atomic code is surrounded by `lock()` and `unlock()` method calls.

```
ReentrantLock aLock = new ReentrantLock();
aLock.lock();
atomic code
aLock.unlock();
```


- When a thread calls `lock()`,
  - it acquires and owns a lock until it calls `unlock()`.
  - No other threads can acquire the lock until it is released with `unlock()`.
    - No other threads can execute atomic code until the lock is released with `unlock()`.
- If a thread calls `lock()` when another thread already owns the lock,
  - it goes to the *blocked* state and *gets blocked* (cannot do anything) until the lock is released.



How Can a Blocked Thread Run Again?

- JVM's thread scheduler
  - Periodically reactivates every blocked thread so that it can acquire a target lock.
    - If the lock is still unavailable, the thread is again blocked.
  - Notifies the completion of atomic code to blocked threads.
    - Chooses one of the blocked threads to acquire the lock.
- Eventually, when the target lock is available, a blocked thread can acquire the lock.

A Locking Idiom

- Call unlock() in a finally clause.
  - ReentrantLock aLock = new ReentrantLock();

```
aLock.lock();
try {
  atomic code
}
finally {
  aLock.unlock();
}
```

- unlock() is never invoked
  - if a method returns from atomic code
  - if atomic code throws an exception
  - A deadlock occurs.
    - Atomic code is locked forever, and no other threads can acquire the lock to execute the atomic code.

-
```
aLock.lock();
try{
 atomic code
}
finally{
 aLock.unlock();
}
```

-
```
try{
  aLock.lock();
  atomic code
}
finally {
 aLock.unlock();
}
```

- Call lock() before a "try" block.
- If a thread throws an exception in lock(), it will not acquire the lock. However, it will call unlock().
  - lock() can throw an InterruptedExcepetion when another thread call interrupt().

When Could a Context Switch Occur?

- Across running different lines/statements.
  -
```
public void deposit(double amount){
      System.out.print("Current balance (d): " + balance);
      double newBalance = balance + amount;
      System.out.println(", New balance (d): " + newBalance);
      balance = newBalance;
}
```

- In compound operations

Atomicity of Operations for Primitive Types

- The read and write operations for primitive data types, except double and long (64-bit) types, are atomic.
  - An "atomic" operation is transformed to a single bytecode instruction for a JVM.
  - While a thread works on an atomic operation, no other threads can work on it.
  - `int x;`
    - Thread A does: `x=1;` Thread B does: `x=2;`
    - An assignment of an int value is atomic.
    - x contains 1 or 2 depending on which thread performs assignment earlier.
      - x never contains other values (e.g., 0 and 3) or corrupted data.
      - An example of corrupted data
        - » Some part of x (e.g., the first 16-bit of x) comes from Thread A and the remaining part (e.g., the other 16bit of x) comes from Thread B.

Compound Operations

- A compound of atomic operations is NOT atomic.
  - int i; boolean done;
  - done = true; // 2 steps
  - i = 1; // 2 steps
  - if(done) // 2 steps
  - i = j; // 2 steps
  - j = i +1; // 5 steps
    - Reading the value of i, reading/loading the value of 1, doing i+1, storing the result of i+1 to a certain memory space, and assigning the result to j.
  - i = i + 1; // 5 steps
  - i++ // 5 steps

  - A race condition can occur due to a context switch *in between* different steps.

What about 64-bit Types?

- The read and write operations for double and long variables are NOT atomic.
  - long x;
    - Thread A does: x = 1L;
    - Thread B does: x = 2L;
    - No guarantee that x contains 1L or 2L.
    - x can contain other values (e.g., 0L and 3L) or corrupted data.

  - aLongVar = 100L; // 2+ bytecode instructions
  - If(aLongVar) // 2+ bytecode instructions
  - aLongVar ++ // 5+ bytecode instructions

Atomicity of Operations for Reference Types

- The read and write operations reference types are atomic.
  - A compound of atomic operations
    - e.g., Foo foo = temp; // requires multiple steps

  - A race condition can occur due to a context switch *in between* different steps.

Other Examples

```
public void deposit(double amount){
     System.out.print("Current balance (d): " + balance);
     double newBalance = balance + amount;
     System.out.println(", New balance (d): " + newBalance);
     balance = newBalance;
}
```

ThreadSafeBankAccount

- Output
  - Lock obtained
  - Current balance (d): 0.0, New balance (d): 100.0
  - Lock released
  - Lock obtained
  - Current balance (w): 100.0, New balance (w): 0.0
  - Lock released
  - Lock obtained
  - Current balance (d): 0.0, New balance (d): 100.0
  - Lock released
  - Lock obtained
  - Current balance (w): 100.0, New balance (w): 0.0
  - Lock released

SummationRunnable

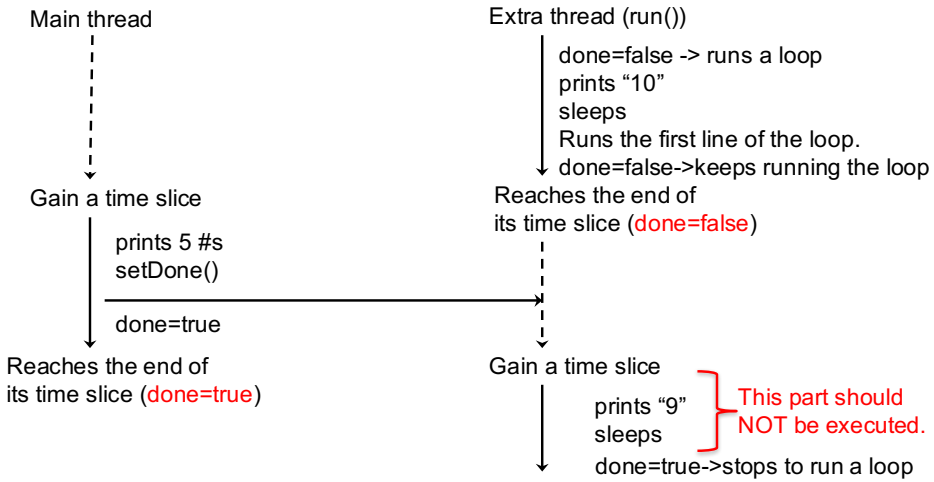
- Read SummationRunnable again. It is NOT thread safe. A race condition can occur.
  - Thread safety:
    - No race conditions
    - No deadlocks

```
- class SummationRunnable implements Runnable{
    private int upperBound = 10;
    private boolean done = false;

    public void run(){
      while( !done ){
        System.out.println(upperBound);
        upperBound--;
      }

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

A Potential Race Condition



"Visibility" Issue

- The current (most up-to-date) value of the shared variable "counter" is not ***visible*** (synchronized) for all threads.

- Locking preserves ***atomicity AND visibility***.

Be VERY Careful

- When multiple threads share and access a variable concurrently.
  - Make sure that a shared variable is protected with a lock.
    - Surround reading and writing parts with lock() and unlock().
      - Reading/writing parts = atomic code
- 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

Solution: Use a Lock and Balking

- Use the "balking" idiom

```
done = false;
ReentrantLock lock = new ReentrantLock();
……
while(true){
  lock.lock();
  try{
    if(done) break; // balking
  }finally{
    lock.unlock();
  }
  System.out.println(...);
}
```

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

- Threads must use the same instance of ReentrantLock.

Try NOT to Surround the Entire Loop with lock() and unlock()

```
public void run(){
   lock.lock();
   while( !done ){   // reading part
      System.out.println(upperBound);
      upperBound--;
      if( upperBound<0 ){ return; }
   }
   lock.unlock(); }
```

```
public void setDone(){
   lock.lock();
   done = true;      // writing part
   lock.unlock(); }
```

- This code is thread-safe, but it does not enjoy concurrency.
  - If the 2nd thread acquires the lock, it complete running the loop.
    - The main thread has a chance to flip the flag.
    - However, the flip occurs a lot later than the 2nd thread wants.
  - If the main thread acquires the lock, it flips the flag.
    - The 2nd thread never run the loop.

```
public void run(){
   lock.lock();
   while( !done ){   // reading part
      System.out.println(upperBound);
      upperBound--;
//    if( upperBound<0 ){ return; }
   }
   lock.unlock(); }
```

```
public void setDone(){
   lock.lock();
   done = true;      // writing part
   lock.unlock(); }
```

- This code is NOT thread-safe.
  - If the 2nd thread acquires the lock, it runs the loop forever.
    - The main thread has no chances to flip the flag.
  - If the main thread acquires the lock, it flips the flag.
    - The 2nd thread never run the loop.

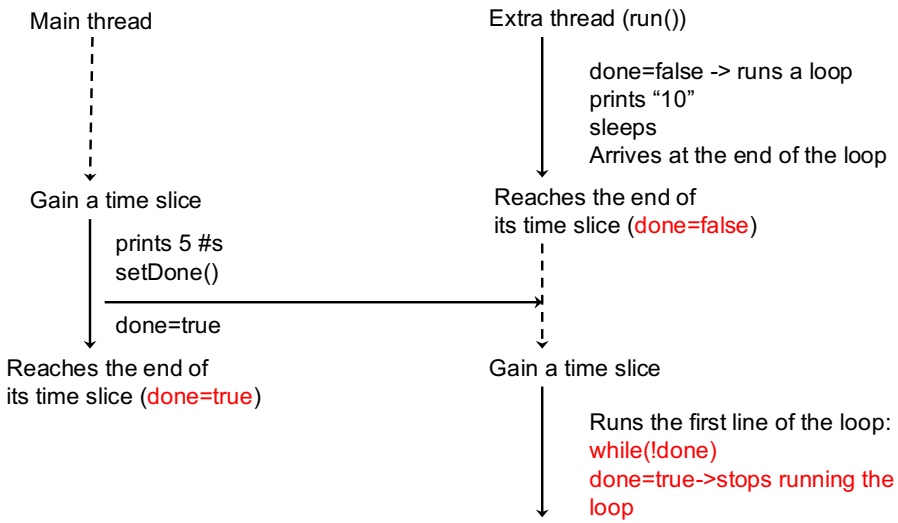
HW 7

- Revise SummationRunnable.java to make it thread safe
  - Use ReentrantLock
  - Use balking
  - Use try-finally blocks.
    - Call unlock() in a finally block. Always do this in all subsequent HWs.

What's Tricky in Thread Programming

- Your test code may or may not be able to detect race conditions.
  - It may not be able to detect race conditions even if you run it a few hundred times.

Consider this "Lucky" Case



Nested Locking

```
class BankAccount {
 private double balance;
 private ReentrantLock lock;

 public void deposit(double amount) {
     lock.lock();
     balance += amount;
     if (balance < MIN_BALANCE)
         subractPenaltyFee();
     lock.unlock(); }

 private void subractPenaltyFee() {
     balance -= PENALTY; //←NO NEED TO SURROUND THIS LINE BY LOCK()
and UNLOCK()
     }
}
```

Thread Reentrancy

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    b = new B();
    lock.lock();
    b.b1(this);
    lock.unlock();
  }
  public void a2(){
    lock.lock();
    do something.
    lock.unlock();
  }
}
```

```
Class B{
  public void b1(A a){
    a.a2();
  }
}

main(){
  A a = new A();
  a.a1();
  a.a2();
```

- This code does not have a deadlock problem.
- A thread can ***re-enter*** the same lock.

```
GreetingRunnable runnable =
    new GreetingRunnable("Hello World");
Thread thread = new Thread(runnable);
thread.start();
```

```
class HelloWorldTest{
    private String greeting;
    public static void main(…){
        greeting = …;
        Thread thread = new Thread( ()->{
            System.out.println(this.greeting)} );
    }
}
```

HW 8

- 4 things to start an extra thread:
  - Define a class implementing the `java.lang.Runnable` interface
    - `public abstract void run();`
  - Write a threaded/concurrent task in `run()` in the class
  - Instantiate `java.lang.Thread` and associate a Runnable object with the thread
  - Start (call `start()` on) the instantiated thread.
    - `run()` is implicitly called on the thread.
- `java.lang.Runnable`: functional interface
  - Can pass a lambda expression to `java.lang.Thread`'s constructor
- Revise GreetingRunnable and HelloWorldTest by replacing GreetingRunnable with a lambda expression