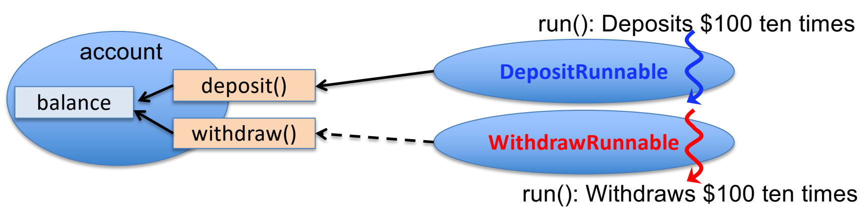
# Thread Synchronization (Locking)

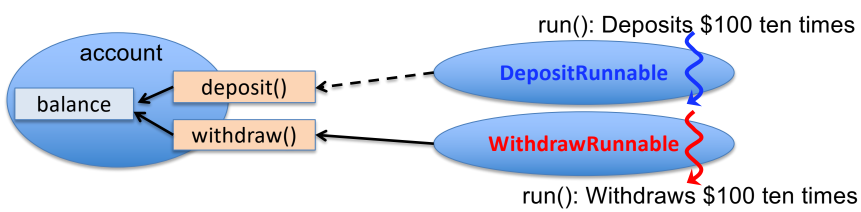
## Solution to Avoid Race Conditions

- Thread synchronization
  – Synchronizes threads
  – Allows only one thread to access a shared variable at a time
    - Forces all other threads to wait and take turn to access it.
  – Enables serialized (or mutually-exclusive) access to a shared variable






- Thread synchronization
  – Prevents the withdrawal thread from withdrawing money when the deposit thread is depositing money.
  – Prevents the deposit thread from depositing money when the withdrawal thread is withdrawing money.




```
• public void deposit(double amount){
      balance = balance + amount;   }
• public void withdraw(double amount){
      balance = balance - amount;   }
```

- Thread synchronization prevents race conditions by eliminating the following cases:
  – Has read `balance`, but hasn't read `amount` yet.
    **Context switch** occurs, followed by a balance change by another thread
  – Has read `balance` and `amount`, but hasn't done addition/subtraction yet.
    **Context switch** occurs, followed by a balance change by another thread
  – Has read `balance` and `amount` and finished addition/subtraction, but hasn't updated `balance` yet
    **Context switch** occurs, followed by a balance change by another thread

# Thread Synchronization with Java

- Java implements thread synchronization by
  - Providing locks
  - Allowing you to write *atomic code* (a.k.a critical section) with locks.
    - Atomic code: A piece of code that can be executed by multiple threads in a serialized (or mutually-exclusive) manner.
      - Only one thread can run it at a time.
        - » When a thread is running it, all other threads have to wait to run it
    - Threads access a shared variable in atomic code.

# Locks in Java

- Used to synchronize (or serialize, or mutually-exclude) multiple threads that access shared data.
- `java.util.concurrent.locks.Lock` interface
  - `ReentrantLock` class: the most commonly-used class for thread synchronization
    - Defines methods for threads to access shared data in a synchronized (or serialized, or mutually-exclusive) manner.
- Atomic code is surrounded by `lock()` and `unlock()` method calls on a lock.

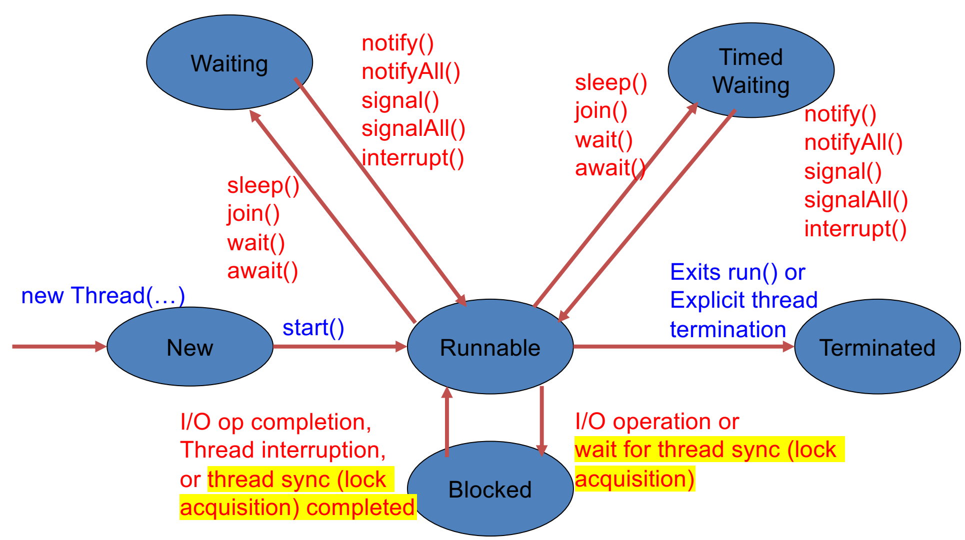
```
– ReentrantLock aLock = new ReentrantLock();
  aLock.lock();
  atomic code (access to shared data)
  aLock.unlock();
```



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



# States of a Thread



# How Can a Blocked Thread Run Again?

- JVM automatically let a *blocked thread* run again when the target lock is released.
  - You (your code) don't have to do anything for that.
- JVM's thread scheduler
  - Periodically reactivates all blocked threads so that they can try to acquire the target lock.
    - If the lock is still unavailable, they get blocked again.
  - Or, detects a release of the target lock.
    - May notify all blocked threads so that one of them can acquire the target lock.
    - May choose one of the blocked threads to acquire the lock.
  - Each blocked thread can eventually acquire the target lock.



# Coding Idiom

- You MUST call `unlock()` to release a lock after running atomic code.
- Call `unlock()` in a finally clause.

```
– ReentrantLock aLock = new ReentrantLock();
  aLock.lock();
  try {
      atomic code (access to a shared variable)
  }finally{
      aLock.unlock(); }
```

- If a `unlock()` call is NOT placed in a finally clause, it may NOT be invoked in the worst cases:
  - if `run()` returns in atomic code
  - if atomic code throws an exception



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

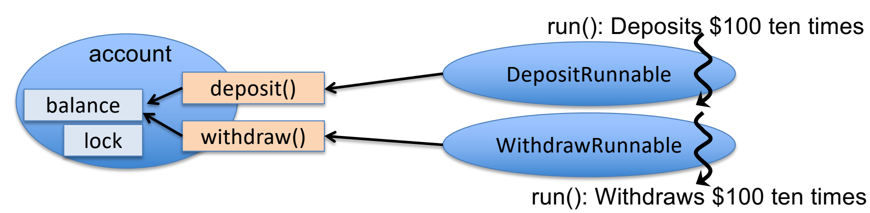
**DO THIS!**

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

**DON'T DO THIS!**

- Make sure to call `lock()` BEFORE the "try" clause.
- If a thread throws an exception in `lock()`, it will not acquire the lock. However, it will call `unlock()`.
  - e.g., `lock()` can throw an `InterruptedException` when another thread call `interrupt()`.

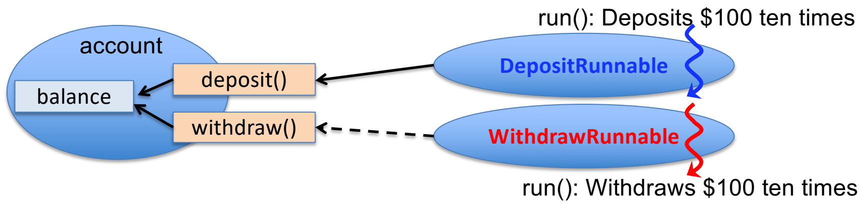
# ThreadSafeBankAccount.java



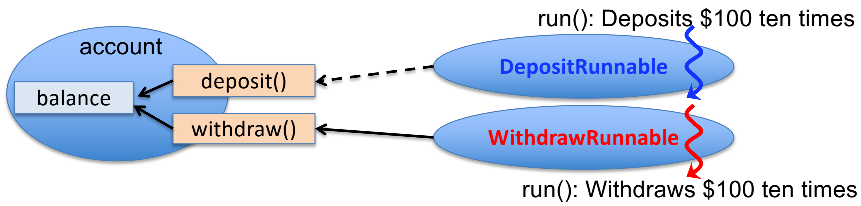

```
• private ReentrantLock lock = new ReentrantLock();

• public void deposit(double amount){
      lock.lock();
       try{
          balance += amount;  // atomic code
      }finally{
          lock.unlock(); } }

• public void withdraw(double amount){
      lock.lock();
      try{
          balance -= amount;  // atomic code
      }finally{
          lock.unlock(); } }
```




- Thread synchronization
  - Prevents the withdrawal thread from withdrawing money when the deposit thread is depositing money.
  - Prevents the deposit thread from depositing money when the withdrawal thread is withdrawing money.




# Summary: How to Avoid Race Conditions?

- When multiple threads share and access a variable concurrently,
  - Make sure to guard the shared variable with a lock.
    - Identify ALL read and write logic to be performed on the variable
    - Surround each of them with `lock()` and `unlock()` to be called on the same lock
  - e.g.,

```
public void deposit(double amount){
    lock.lock();
      try{
        balance = balance+amount; // atomic code (read & write)
    }finally{
        lock.unlock(); }
}
```

  - 

```
public void withdraw(double amount){
    lock.lock();
     try{
        balance = balance-amount; // atomic code (read & write)
    }finally{
        lock.unlock(); }
}
```

- 

```
public void deposit(double amount){
     balance = balance + amount;
}
```

  - A compound of 5 atomic operations.

- Thread synchronization enables serialized (or mutually-exclusive) execution of a compound operation.
  - Allows only one thread to perform the compound operation at a time.

  - 

```
ReentrantLock aLock = new ReentrantLock();
aLock.lock();
try{
   balance = balance + amount; // atomic code
}
finally{
     aLock.unlock();
}
```

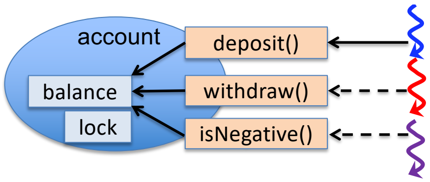
- 

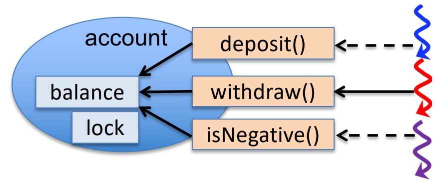
```
public void deposit(double amount){
  ReentrantLock aLock = new ReentrantLock();
  aLock.lock();
  try{
    balance = balance + amount; // atomic code
  }finally{
    aLock.unlock();
  }
}
```

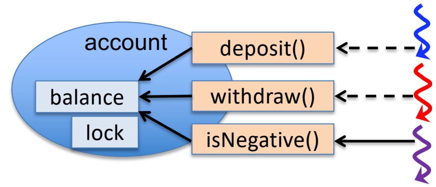
- As a result, the deposit and withdrawal threads update `balance` in either of the following 2 cases:

  - Has't read `balance`.
    **Context switch** occurs, followed by a balance change by another thread

  - Has read `balance` and `amount`, performed addition/sutraction, and updated `balance`
    **Context switch** occurs, followed by a balance change by another thread




```
• private ReentrantLock lock = new ReentrantLock();

• public void deposit(double amount){
      lock.lock();
      balance += amount;
      lock.unlock(); }

• public void withdraw(double amount){
      lock.lock();
      balance -= amount;
      lock.unlock(); }

• Public boolean isNegative(){
      lock.lock();
      if(balance<0){ return true; }
      else if{ return false; }
      lock.unlock(); }
```

- Thread synchronization
  - Mutually excludes those 3 threads, so only one of them can run atomic code associated with “lock” at a time.
  - It is important to use the same lock in ALL read and write logic to be performed on “balance.” Otherwise, threads are NOT mutually excluded.

# Nested Locking

```
• class BankAccount {
    private double balance;
    private double MIN_BALANCE = ...;
    private double PENALTY = ...;
    private ReentrantLock lock = ...;

     public void withdraw(double amount) {
        lock.lock();
        balance -= amount;           // 5+ steps
        if(balance < MIN_BALANCE)    // 3+ steps
           subractPenaltyFee();
        lock.unlock();
     }

     private void subractPenaltyFee() {
        balance -= PENALTY;          // 5+ steps

        // NO NEED TO SURROUND THIS LINE with LOCK() and UNLOCK()
        // because it is called from atomic code.
     }
  }
```



```
• class BankAccount {
    private double balance;
    private double MIN_BALANCE = ...;
    private double PENALTY = ...;
    private ReentrantLock lock = ...;

     public void withdraw(double amount) {
        lock.lock();
        balance -= amount;              // 5+ steps
        if(balance < MIN_BALANCE)       // 3+ steps
           subractPenaltyFee();
        lock.unlock(); }

     private void subractPenaltyFee() {
        balance -= computePenalty();    // many steps

        // NO NEED TO SURROUND THIS LINE with LOCK() and UNLOCK()
        // because it is called from atomic code. }

     private void computePenalty() {
        ...                             // many steps

        // NO NEED TO SURROUND THIS LINE with LOCK() and UNLOCK()
        // because it is called from atomic code. } }
```



# Thread Reentrancy

```
• class BankAccount {
    private double balance;
    private double MIN_BALANCE = ...;
    private double PENALTY = ...;
    private ReentrantLock lock = ...;

   public void withdraw(double amount) {
      lock.lock();
      balance -= amount;          // 5+ steps
      if(balance < MIN_BALANCE)   // 3+ steps
         subractPenaltyFee();
      lock.unlock(); }

   private void subractPenaltyFee() {
      lock.lock();
      balance -= PENALTY;         // 5+ steps
      lock.unlock(); }}
```

- This code does not have a (deadlock) problem.
- A thread can ***re-enter*** *(or re-acquire)* the same lock as far as it already owns the lock.

```
class A{
  private B b = new B();
  private ReentrantLock lock=...;

  public void a1(){
    lock.lock();
    b.b1(this); //nested locking
    lock.unlock();
  }
  public void a2(){
    lock.lock();
    do something.
    lock.unlock();
  } }
```

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

If a thread performs:

```
A a = new A();
a.a1();
```

*it re-enters (or re-acquires)* the same lock that it already owns.

- This code does not have a (deadlock) problem.
- A thread can ***re-enter*** *(or re-acquire)* the same lock as far as it already owns the lock.

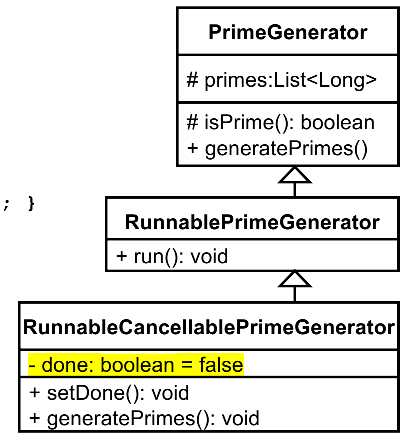
# RunnableCancellablePrimeGenerator

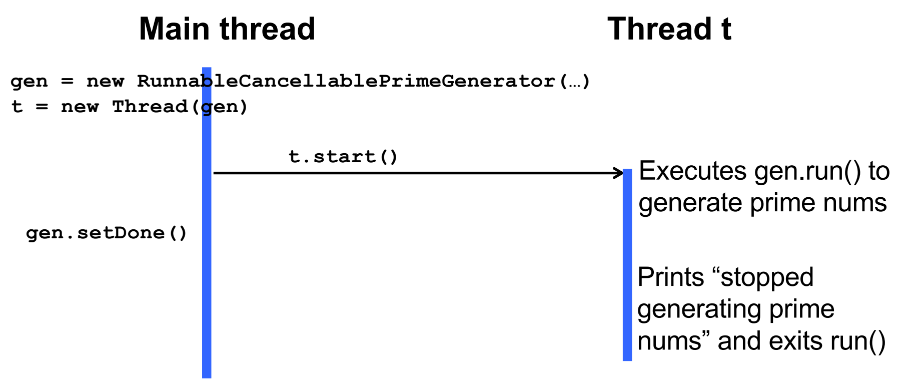
```
class RunnableCancellablePrimeGenerator ...{
  private boolean done = false;

  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); }
  }

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



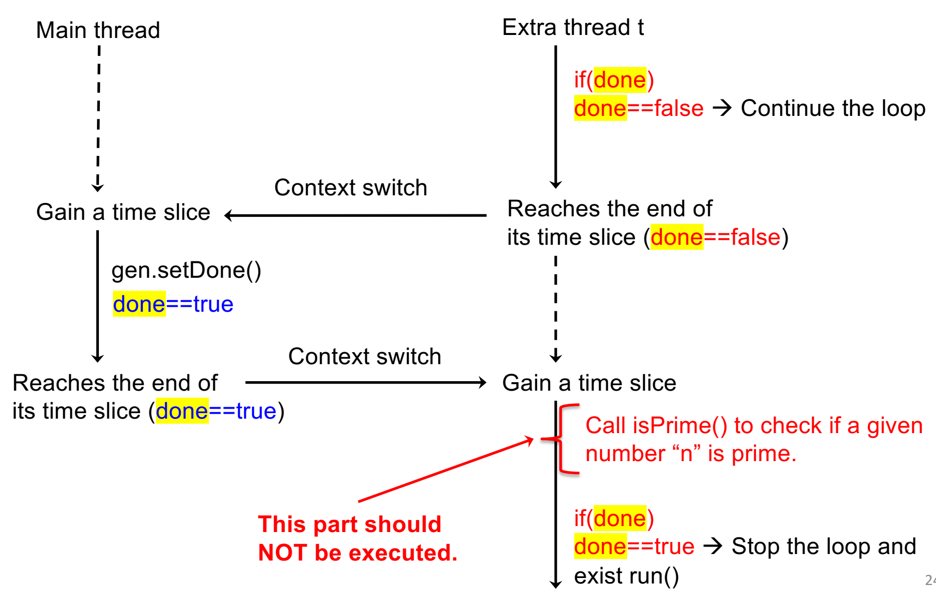




```
for(long n = from; n <= to; n++){
    if(done){
        System.out.println("Stopped generating prime nums.");
        this.primes.clear();
        break;
    }
    if( isPrime(n) ){ this.primes.add(n); }}
```

- This code is NOT thread-safe. Race conditions can occur.



# A Potential Race Condition



```java
class RunnableCancellablePrimeGenerator 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;
  }
}
```

Thread *t*

Context switch



```java
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;
  }
}
```

Thread *t*

Context switch

Main thread

Context switch



```java
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;
  }
}
```

Thread *t*

Context switch

Main thread

Context switch



# Visibility Issue

- The 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
  - Surround each read/write logic with lock() and unlock() calls on the same **ReentrantLock**

# Risk of Race Conditions

- Race conditions generate different risk levels in different applications.
  - It may be acceptable to make an extra loop iteration and generate an extra prime number.
  - It is NOT acceptable to make an extra loop iteration in other applications in
    - Safety/mission critical domains
      - Nuclear plant control, autonomous driving, rocket launch, weapon management, etc.
    - Financial domains
      - Banking, high-frequency trading, stock exchange management, etc.
    - Medical/healthcare domains
  - In any case, you should know that a race condition exists and should be able to fix it.

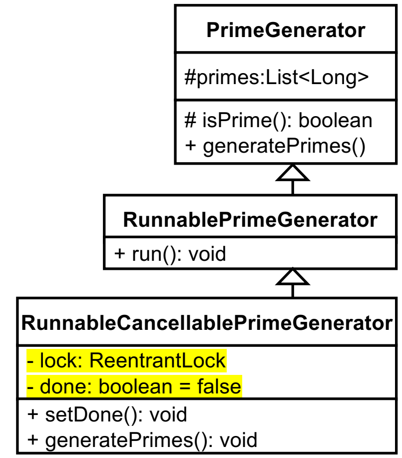
# Solution w/ Thread Synchronization

```
class RunnableCancellablePrimeGenerator ...{
  private ReentrantLock lock = new ReentrantLock();
  private boolean done = false;

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

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

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




# How to Prevent 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 on it with lock() and unlock().
- When a loop checks a condition with a shared variable (e.g., flag).
  - Surround the conditional with lock() and unlock()
  - Try NOT to surround the entire loop with with lock() and unlock()! Why?
    - Does not enjoy concurrency.
    - May result in a deadlock.

# Having the Entire Loop as Atomic Code Does NOT Enjoy Concurrency

- DO **NOT** do this.

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

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

- Do this.

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

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

- If a thread acquires the lock to start generating primes, it will release that lock after *all primes are generated*.
  - No other threads can flip the flag until all primes are generated.
  - No other threads can terminate the prime generation thread when it is generating primes.

*A thread that generates primes*

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

*The main thread, which wants to terminate the prime generation thread, can acquire the lock after all primes have been generated.*

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

# Having the Entire Loop as Atomic Code May Result in a Deadlock

- DO **NOT** do this.

```
lock.lock();
try{
  while(!done){
    System.out.println("#");
  }
}finally{
  lock.unlock();
}
```

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

- Do this.

```
while(true){
  lock.lock();
  try{
    if(done) break;
    System.out.println("#");
  }finally{
    lock.unlock();
  }
}
```

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

- If a thread acquires the lock to start printing #s, it will print #s forever.
  - No other threads can flip the flag forever (deadlock!)

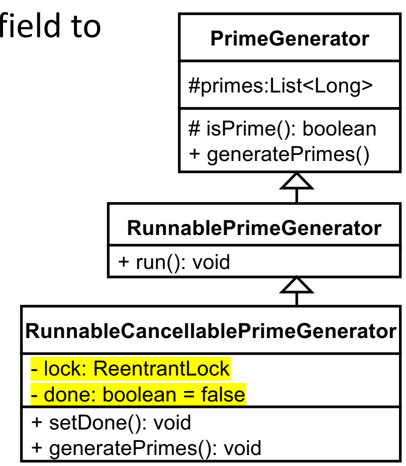
```
lock.lock();
while(!done){                // read logic
System.out.println("#");     // performs some task
}
lock.unlock();
```

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

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

# HW 6-1

- Revise `RunnableCancellablePrimeGenerator` to be thread-safe.
  - Add a `ReentrantLock` as a data field to guard the shared variable `done`
  - Revise `generatePrimes()` and `setDone()` to access `done` with the lock



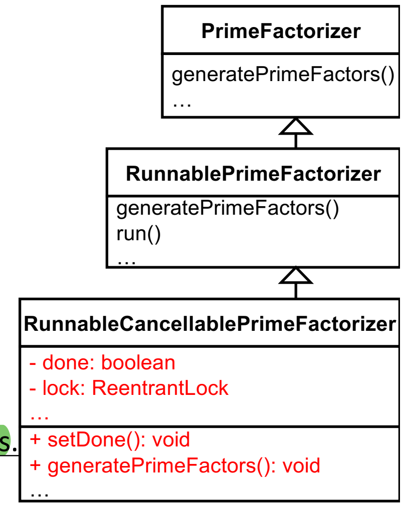
– In **setDone()**,
  - Use try-finally blocks.
    - Call **unlock()** in a finally block. Always do this in all subsequent HWs.

– In **generatePrimes()**
  - Use try-finally blocks; Call **unlock()** in a finally block
  - Do not surround the entire “for” loop with **lock()** and **unlock()**.

# HW 6-2

- Implement **RunnableCancellablePrimeFactorizer** by extending **RunnablePrimeFactorizer**.
  – Implement flag-based thread termination
  – Add a flag variable **done**
  – Add a **ReentrantLock** to guard the shared variable **done**
  – Revise **generatePrimeFactors()** and **setDone()** to access **done** with the lock

  – Call **unlock()** in a finally block. Always do this in all subsequent HWs.




- Place your HW6-1 and HW6-2 solutions in the same package: **edu.umb.cs681.primes**.
- Upload them to the same GitHub repo.

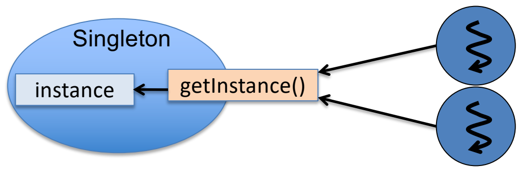
# Recap: Singleton Design Pattern

- 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.

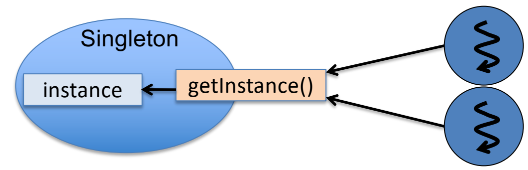



- When multiple threads call `getInstance()` concurrently, they share the data field `instance`.

```
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: (multiple) steps
      instance = new Singleton();     // Write: (multiple) steps
    return instance;                  // Read: (multiple) steps
} }                                   // NOT THREAD-SAFE
```






```
public class Singleton{
  private Singleton(){};
  private static Singleton instance = null;
                            // Write logic. Requires 2 steps
                            // THREAD-SAFE }
```

- JVM completes all initial value assignments on all static data fields BEFORE using a class or creating class instances.
  - `instance` has been initialized before any threads call `getInstance()`
  - This write logic is thread-safe.
    - No need to worry about race conditions here.



# Concurrent Singleton Design Pattern

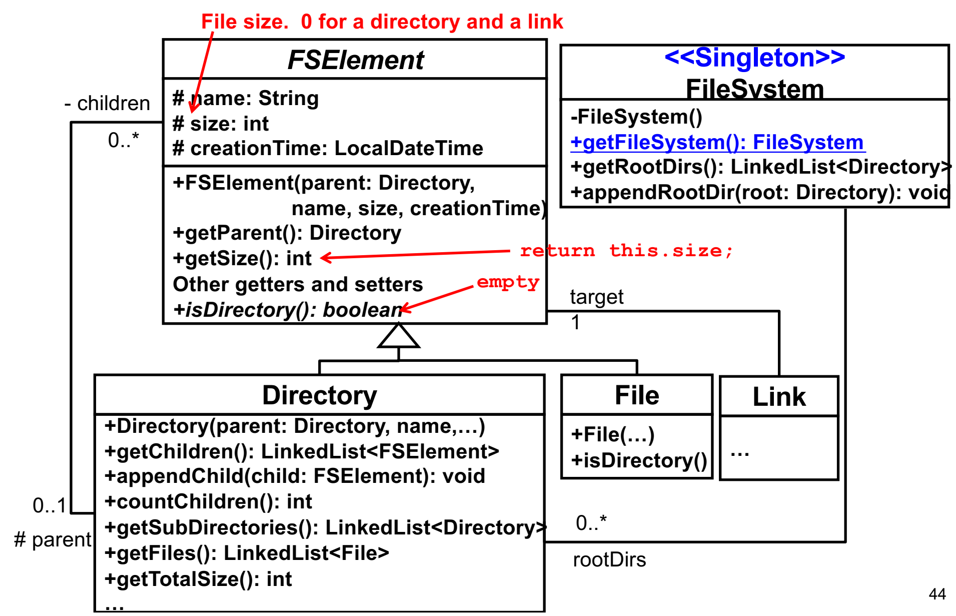
- Guarantee that a class has only one instance.

```
public class ConcurrentSingleton{
 private ConcurrentSingleton(){};
 private static ConcurrentSingleton 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 ConcurrentSingleton(); }
       return instance;
     }finally{
       lock.unlock();
     }
  }
}
```

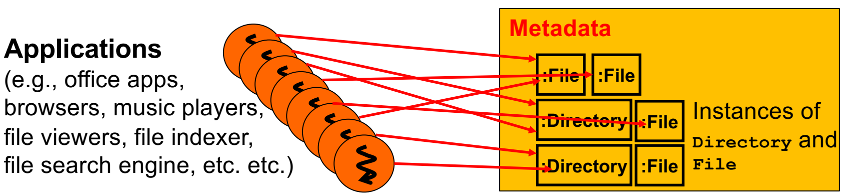


# Recall CS680's HW 8



# Concurrency in a File System

- Modern file systems are all multi-threaded.
  - Any modern OSes want apps and OS services to access a file system concurrently.
    - e.g., reading/updating metadata of file system elements and adding/removing file system elements
    - e.g., saving a file while renaming another file.




# Let's Make your File System Multi-threaded

- In CS680's HW 8, you wrote basic classes to implement a single-threaded file system
  - **FSElement, Directory, File, Link** and **FileSystem**
  - You never considered multiple threads.

- To make your file system multi-threaded, you need to expect multiple threads will use them concurrently.
  - For example, multiple threads will use `FileSystem` concurrently.
    - e.g., They will call `FileSystem.getFileSystem()` concurrently.
  - But, `FileSystem` is NOT thread-safe.

# HW 7

- Revise `FileSystem.getFileSystem()` to be thread-safe
  - Define a lock in `FileSystem`. Use the lock in `getInstance()` to guard the `instance` in data field.
    - Use try-finally blocks: Always do this in all subsequent HWs.
  - Run multiple threads to call `getInstance()`
    - Make sure that only one instance is created.
    - c.f. You worked on identify check in CS680.
- No need to revise other methods of `FileSystem`.
  - e.g., `getRootDirs()`, `appendRootDir()`, etc.

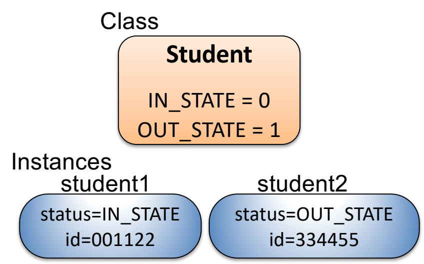
# Just in Case… Static Data Fields and Methods

- A static data field
  - Created and used on a per-class basis.
    - All instances share the data field.
- A regular (non-static) data field
  - Created and used on an instance-by-instance basis.
    - Different instances have different copies of the data field.

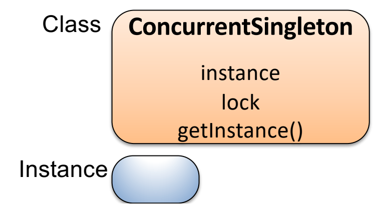
- A static data field
  - Created used on a per-class basis.
    - All instances share the data field.
- A regular (non-static) data field
  - Created and used on an instance-by-instance basis.
    - Different instances have different copies of the data field.

```
• public class Student {
    private static    int IN_STATE = 0;
    private static    int OUT_STATE = 1;
    private           int status;
    private           int id;  }

• Student student1 = new Student(...);
  Student student2 = new Student(...);
```




- A static method
  - Created and used on a per-class basis.
  - Can access static data fields.
  - Can NOT access regular (non-static) data fields.
- A regular (non-static) method
  - Created and used on an instance-by-instance basis.
  - Can access both regular (non-static) and static data fields.




## Just in case... Initialization of Data Fields

```
• public class Foo{
      ReentrantLock lock          = new ReentrantLock();
      static ReentrantLock sLock = new ReentrantLock();
      int foo = 10;
      int boo;
      Object obj;   }
```

- Data fields (both static and non-static ones) are initialized before a constructor is called.
  - Data field initialization is always thread-safe.
  - The default value is used for initialization if the initial value is not explicitly given.
    - e.g., 0 for int, null for a reference type, etc.