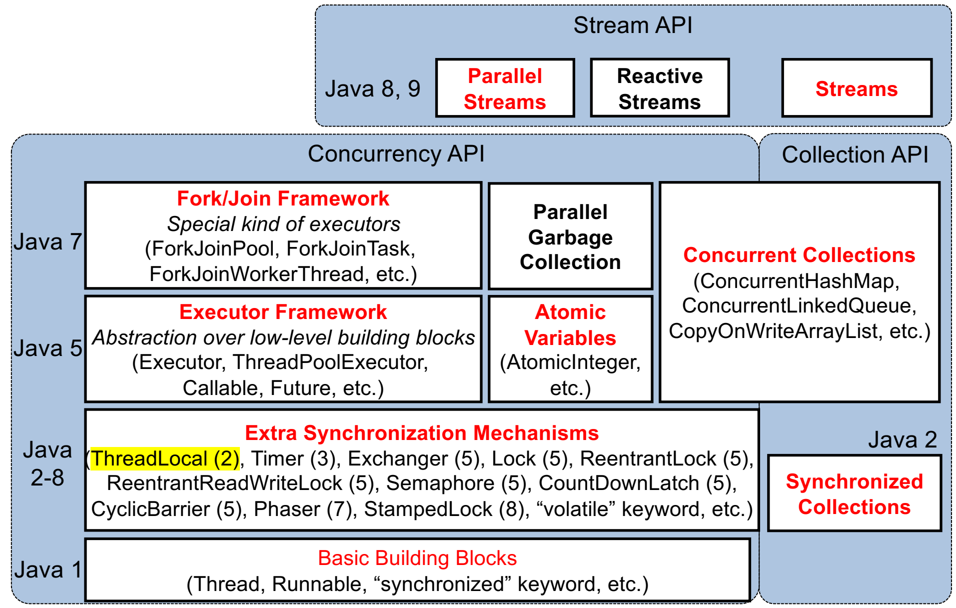
Thread-Specific Storage (TSS)

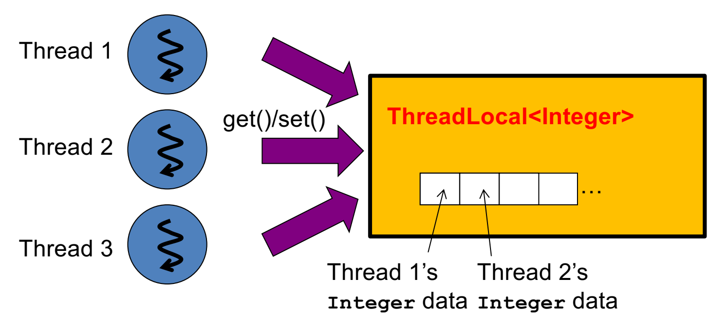
Concurrency API in Java



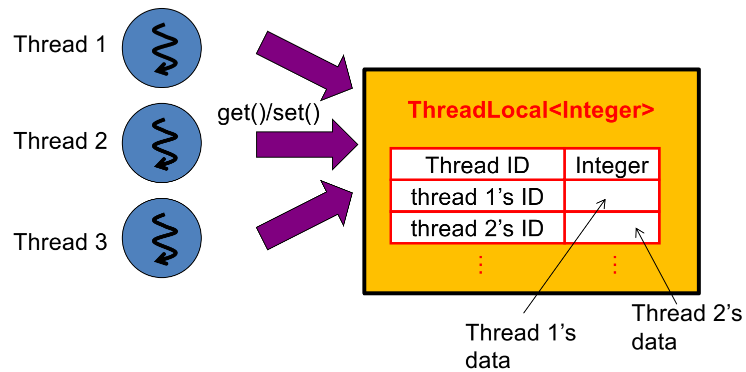
Thread-Specific Storage (TSS)

- Storage (memory space) that is allocated and reserved for a particular thread.
- Once a TSS is allocated to a thread, no other threads can access it.
  - It is dedicated for a single thread.
    - It is NEVER shared among multiple threads.
  - You don't have to worry about potential race conditions on it because of no sharing.
- Implemented in `java.lang.ThreadLocal<T>`

Common Use Case of `ThreadLocal<T>`



- Different threads generate different data of the same type (`T`).
- Each of them
  - stores its own data into `ThreadLocal<T>` with `set()`
  - read its own data from the `ThreadLocal<T>` with `get()`



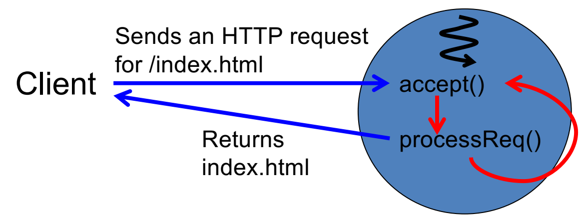
- **`ThreadLocal`**
  - allows threads to access data through their (thread) IDs.
    - A thread cannot access any data generated and maintained by the other threads.
  - Internally performs thread sync to guard the table.

- Client code `ThreadLocal` does not have to do thread sync.

Exercise: Web Server Development

- Suppose you are developing a web server.
  - Receives an HTTP request that a client (browser) transmits to request an HTML file.
  - Returns the requested file to the client.
- What if the server receives multiple requests from multiple clients simultaneously?
  - If the server is single-threaded, it processes requests *sequentially*.

Single-threaded server

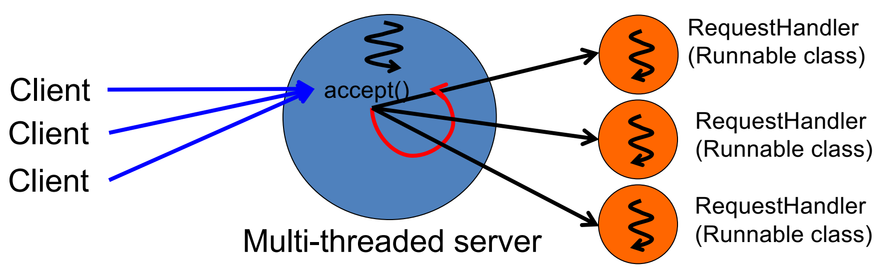


accept() waits/blocks for an incoming request. It returns once a request arrives at the server.

processReq() parses the request, reads the requested file and returns the file.

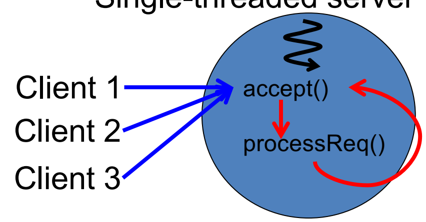
Thread-per-Request Concurrency

- Once the web server's main thread receives a request from a client, it creates a new thread.
  - The new thread parses the incoming request, reads the requested file and returns it.
  - The thread terminates once the requested file is returned to the client.



Concurrent (Multi-threaded) Web Server

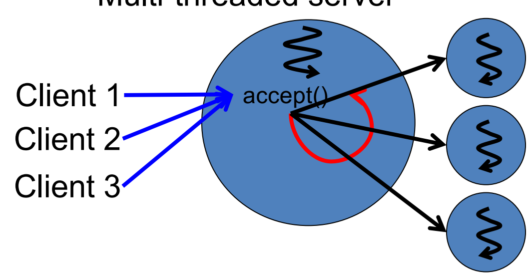
Single-threaded server



The server does the following tasks for each request
(1) Parse a request
(2) Read a requested file
(3) Returns the file

Process requests sequentially.

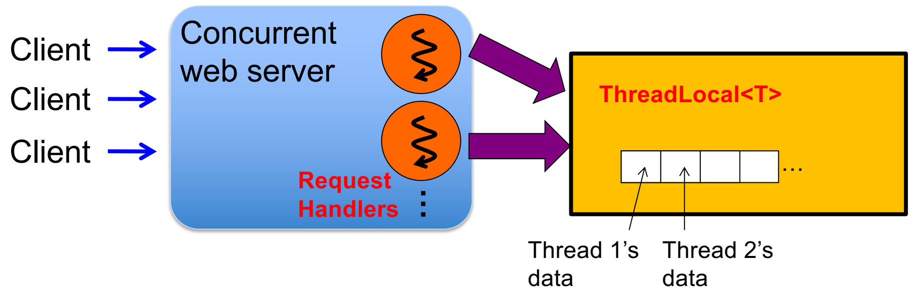
Multi-threaded server



Each thread does the following tasks for a request.
(1) Parse a request
(2) Read a requested file
(3) Returns the file

Requests are processed concurrently.

TSS in a Concurrent Web Server



- Each request handler (thread):
  - May need customer info (e.g. customer ID) from a browser cookie to display some personalized content (e.g. shopping cart items)
  - May need client-specific information (e.g., client OS name and browser name) to display some client-specific content.

Race Conditions: A Summary

3 Cases where Race Conditions can Occur

- **CASE 1:** Threads share and access a variable (data field)
  - Solution 1: Define the variable as a local variable, rather than a data field.
    - Local variables are never shared among threads.
  - Solution 2: Define a lock as a data field of the variable's enclosing class and use it to access the variable
    - Surround every read/write logic on the variable with the lock() and unlock() calls on the lock
  - Solution 3: Use thread-safe tools
    - e.g. volatile variable, atomic variable, immutable class, TSS, etc.
    - Can skip thread synch

- **CASE 2:** Threads call an API method that is NOT thread-safe.
  - You cannot define a lock in the method's enclosing class (i.e., API class).
  - You need "client-side locking," which means performing thread sync (or "locking") in your client code that uses the API method.

- Example: **LinkedList** is NOT thread-safe; its methods never perform thread sync.

```
– // This class is NOT thread-safe
  public class LinkHolder {
    private LinkedList<Integer> list;

    public void addValue(Integer i){
      list.add(i);
    }

    public LinkedList<Integer> getList(){
      return list; } }
```


- Example: **LinkedList** is NOT thread-safe; its methods never perform thread sync.

```
– // This class is NOT thread-safe
  public class LinkHolder {
    private LinkedList<Integer> list;

    public void addValue(Integer i){
      list.add(i);
    }

    public LinkedList<Integer> getList(){
      return list; } }

– // This class is thread-safe, thanks to client-side locking.
  public class ListHolder {
    private LinkedList<integer> list;
    private ReentrantLock lock = new ReentrantLock();

    public void addValue(Integer i){
      lock.lock();
      list.add(i);
      lock.unlock();
    }

    public LinkedList<Integer> getList(){
      lock.lock();
      return list;
      lock.unlock(); } }
```


- **CASE 3:** Threads run client code of a thread-safe method.
  - You may (or may not) need “client-side locking,” which means performing thread sync in the client code.
  - Example: **Integer** is thread-safe, but the following client code is not.

```
– // This class is NOT thread-safe.
  public class Person {
   private Integer age;

   public void setAge(Integer age){
     this.age = age; }

   public Integer getAge(){
      return this.age; }

  public boolean isKindergartener(){ // Multi-steps. Not thread-safe
     if(this.age < 6){return true;}  // Equivalent to:
     else{return false; }}           // if(this.age.intValue()<6)...
```



```
– // This class is thread-safe.
  public class Person {
   private Integer age;                    // Shared variable
   private ReentrantLock lock = new ReentrantLock();

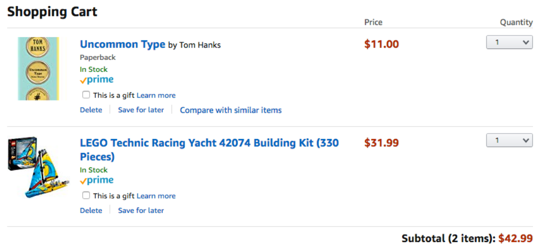
   public void setAge(Integer age){
     lock.lock();
     this.age = age;
     lock.unlock(); }

   public Integer getAge(){
     lock.lock();
     return this.age;
     lock.unlock(); }

  public boolean isKindergartener(){
     lock.lock();
     if(this.age < 6){return true;}
     else{return false;}
     lock.unlock(); }}
```

Exercise

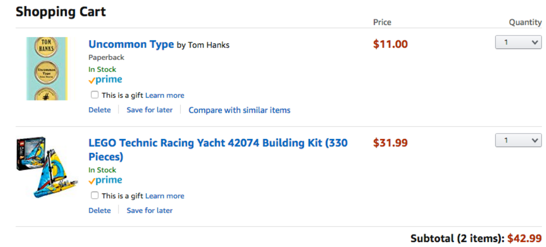
- Suppose you are implementing shopping carts for an online retailer.

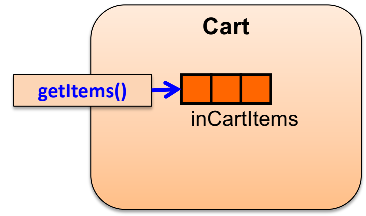


Customer — **Cart** — **Product**

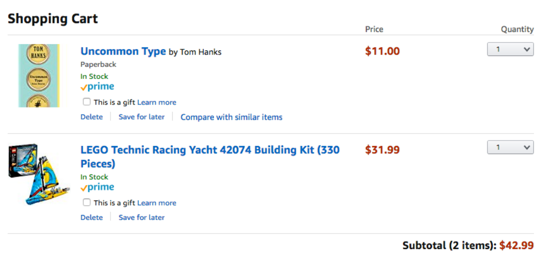
Cart
- inCartItems: LinkedList<Product> **…**
+getItems(): LinkedList<Product> **… other getters** **+addItem(product: Product): void** **+removeItem(productIndex: int): void** **+checkOut(): Order** **…**

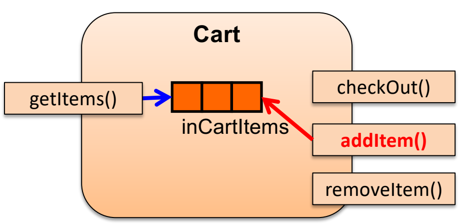
Cart
- inCartItems: LinkedList<Product> **…**
+getItems(): LinkedList<Product> **… other getters** **+addItem(product: Product): void** **+removeItem(productIndex: int): void** **+checkOut(): Order** **…**

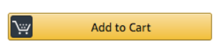




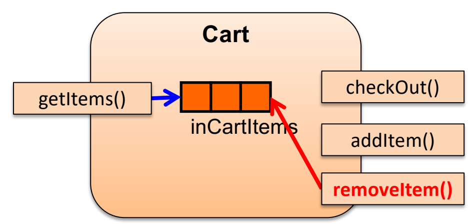
Cart
- inCartItems: LinkedList<Product> **…**
+getItems(): LinkedList<Product> **… other getters** **+addItem(product: Product): void** **+removeItem(productIndex: int): void** **+checkOut(): Order** **…**



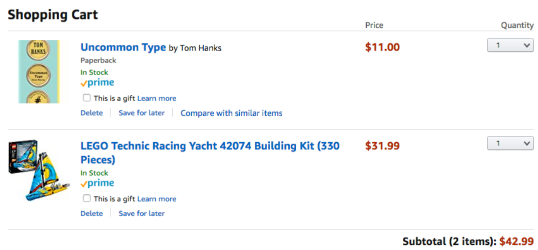


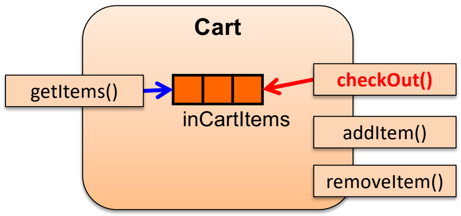


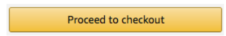
Cart
- inCartItems: LinkedList<Product> **…**
+getItems(): LinkedList<Product> **… other getters** **+addItem(product: Product): void** **+removeItem(productIndex: int): void** **+checkOut(): Order** **…**

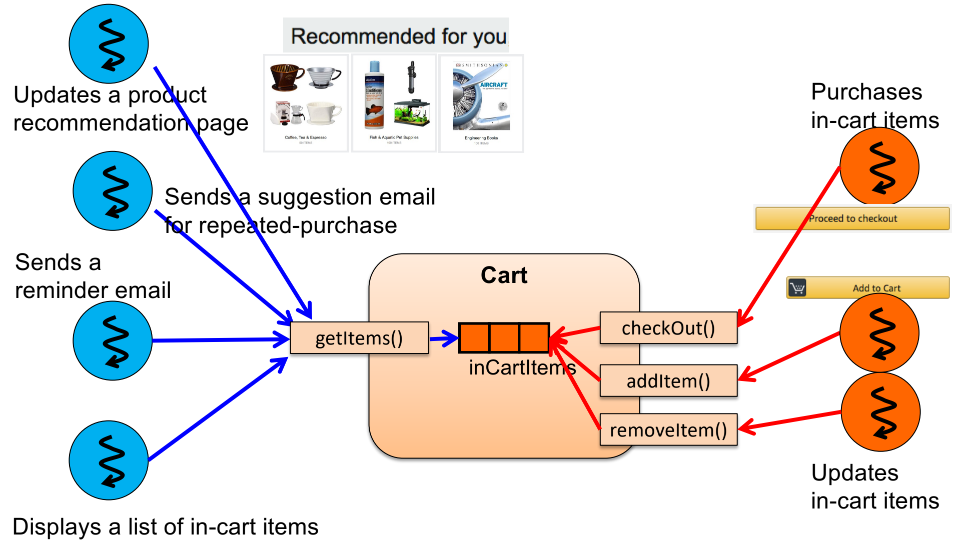


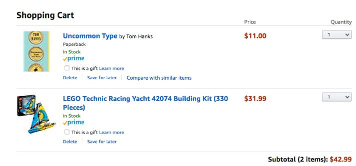
Cart
- inCartItems: LinkedList<Product> ...
+getItems(): LinkedList<Product> ... other getters +addItem(product: Product): void +removeItem(productIndex: int): void +checkOut(): Order ...



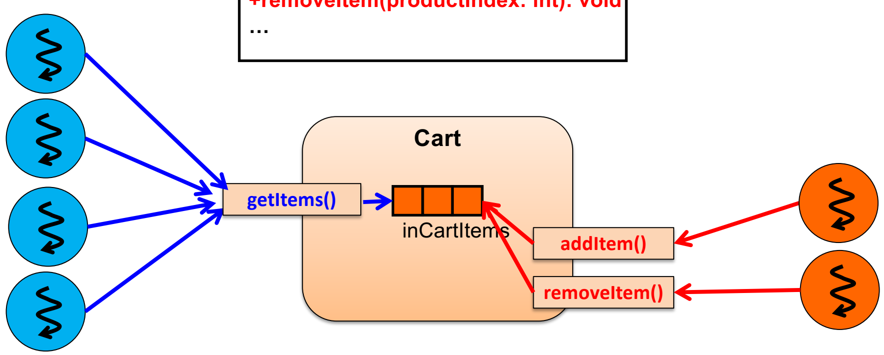


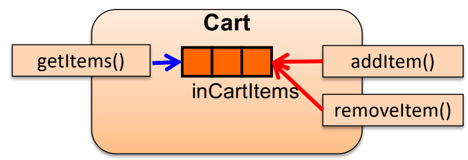






Cart
- inCartItems: LinkedList<Product> ...
+getItems(): LinkedList<Product> +addItem(product: Product): void +removeItem(productIndex: int): void ...






```
class Cart{
 private LinkedList<Product> inCartItems
   = new LinkedList<>();

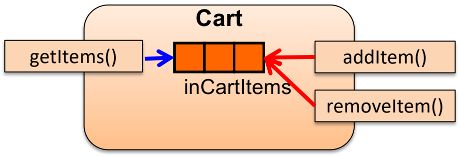
 public LinkedList<Product> getItems(){
   return inCartItems; }              // READ

 public void addItem(Product item){
   inCartItems.add(item); }           // WRITE

 public void removeItem(int productIndex){
   inCarItems.remove(productIndex); // WRITE
} }
```

- inCarItems is shared by multiple threads.
- Need to guard it against concurrent access.
  - Identify every single *read* and *write* operation on it.
  - Surround it with lock() and unlock() on a lock

- **Cart is not thread-safe. Explain why and how to make it thread-safe.**
  - **Note: LinkedList is not thread-safe. (All of its public methods such as add(), remove() and get() never perform thread synch.)**

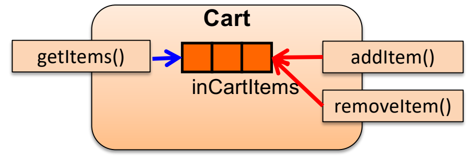



```
class Cart{
 private LinkedList<Product> inCartItems
   = new LinkedList<Product>();

 public LinkedList<Product> getItems(){
   return inCartItems; }     // READ (2 steps)

 public void addItem(Product item){
   inCartItems.add(item); } // WRITE
                            // (multiple steps)
 public void removeItem(int productIndex){
   inCarItems.remove(productIndex);
} }                         // WRITE
                            // (multiple steps)
```

- Each write op is performed with a method of `LinkedList`.
  - `get()` and `remove()`
  - Not thread-safe
- If a thread calls `addItems()` and another calls `removeItem()` concurrently…
- If multiple threads call `removeItem()` concurrently…
- If multiple threads call `addItem()` concurrently…




```
class Cart{
 private LinkedList<Product> inCartItems
   = new LinkedList<Product>();

 public LinkedList<Product> getItems(){
   return inCartItems; }     // READ (2 steps)

 public void addItem(Product item){
   inCartItems.add(item); } // WRITE
                            // (multiple steps)
 public void removeItem(int productIndex){
   inCarItems.remove(productIndex);
} }                         // WRITE
                            // (multiple steps)
```

- Each write op is performed with a method of `LinkedList`.
  - `get()` and `remove()`
  - Not thread-safe.
- We cannot change the methods of `LinkedList`.
  - Cannot introduce thread sync into those API methods.
- We should do thread sync in `Cart` (i.e. client code of `LinkedList`).

"Case 2" where race conditions occur.
Do client-side locking.

Thread-safe Shopping Cart

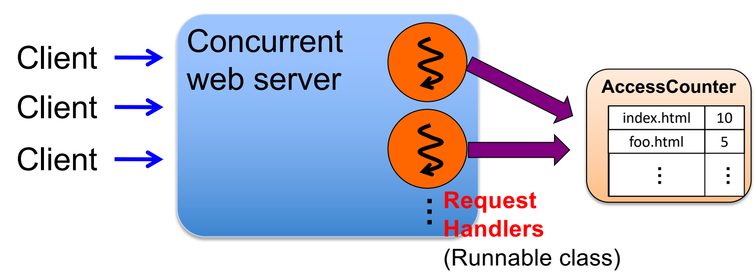
```
class Cart{
 private LinkedList<Product> inCartItems = new...
                              // LinkedList is not thread-safe
 private ReentrantLock lock = new ...;

 public LinkedList<Product> getItems(){
   lock.lock();
   return inCartItems;             // READ
   lock.unlock(); }

public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);          // WRITE
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   inCarItems.remove(productIndex); // WRITE
   lock.unlock(); } }
```

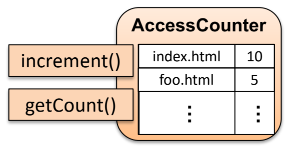
Another Exercise: Access Counter in a Concurrent Web Server



run(): Parses a request, reads a requested file, returns the file and increments the file's access count.

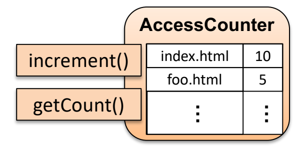
- Each request handler (thread):
  - Increments the access count of a file that is being accessed.

- **AccessCounter**
  - Contains a map that pairs a file path and its access count.
    - Assume `java.util.HashMap<java.nio.file.Path, Integer>`
  - `void increment(Path path)`
    - accepts a file path and increments its access count.
    - if(A requested path is in AC){
      increment the path's access count. }
      else{
      add the path and the access count of 1 to AC. }
  - `int getCount(Path path)`
    - accepts a file path and returns its access count.
    - if(A requested path is in AC){
      get the path's access count and return it. }
      else{
      return 0. }

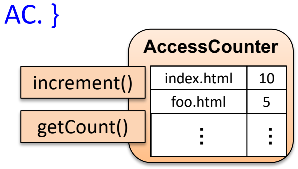


Concurrent Access Counter

- **HashMap** is NOT thread-safe.
  - Its public methods never perform thread synchronization.
    - `containsKey()`, `put()`, `get()`, `putIfAbsent()`, `replace()`, etc.
    - C.f. API doc: "Note that this implementation is not synchronized."
  - Race conditions can occur in those public methods.
- Client code of those public methods needs to perform thread sync (client-side locking; c.f. Case 2).
  - `AccessCounter`'s `increment()` and `getCount()`
    - `increment()`
      - if(A requested path is in AC){
        increment the path's access count. }
        else{
        add the path and the access count of 1 to AC. }
    - `getCount()`
      - if(A requested path is in AC){
        get the path's access count and return it. }
        else{
        return 0. }

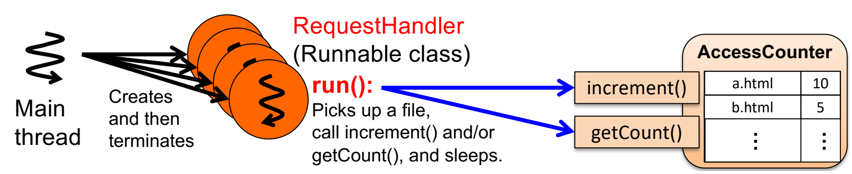


- `AccessCounter`'s `increment()` and `getCount()` need to perform thread sync.
  - `increment()`
    - lock.lock();
      if(A requested path is in AC){
      increment the path's access count. }
      else{
      add the path and the access count of 1 to AC. }
      lock.unlock();
  - `getCount()`
    - lock.lock();
      if(A requested path is in AC){
      get the path's access count and return it. }
      else{
      return 0. }
      lock.unlock();

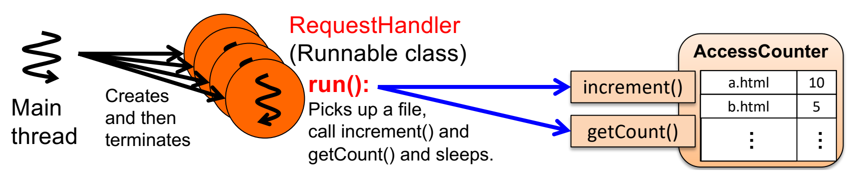


HW 11

- Implement `AccessCounter` as a thread-safe *Singleton* class.
  - Define a `HashMap<java.nio.file.Path, Integer>`
  - Define a regular (non-static) lock and use the lock to guard the `HashMap` in `increment()` and `getCount()`
  - Define another (static) lock and use the lock to guard the instance of `AccessCounter` in `getInstance()`
- Place some test/dummy files to be accessed (a.html, b.html, etc.)
- `RequestHandler` (`Runnable` class)
  - `run()`: Picks up one of the files at random, calls `increment()` and/or `getCount()` for that file, and sleeps for a few seconds. Repeats this forever with an infinite loop.
- Have `main()` create 10+ instances of `RequestHandler` and use 10+ threads to execute `RequestHandler`'s `run()`.



- Do 2-step thread termination in **`RequestHandler`**.
  - Have the main thread terminate those 10+ extra threads in 2 steps.
  - Define a volatile flag in **`RequestHandler`**
    - Or, use an **`AtomicBoolean`**
  - Have the main thread flip the flag on each **`RequestHandler`** and call **`interrupt()`** on each request-handling threads.

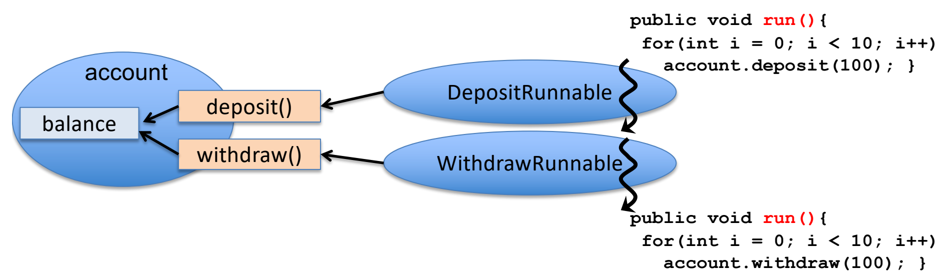


Thread Safety Issues

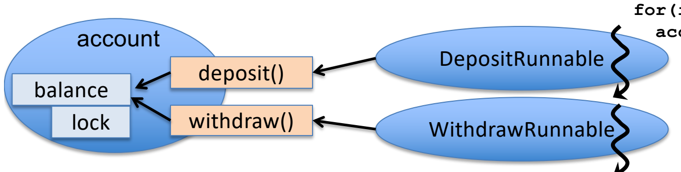
- Race conditions
  - Data inconsistency
- Deadlocks
  - Threads get stuck and cannot proceed even if they intend/want to do so.
- Thread-safe code is free from both race conditions and deadlocks.

Deadlock

DeadlockedBankAccount.java



DeadlockedBankAccount.java




```
public void run(){
 for(int i = 0; i < 10; i++){
   account.deposit(100); }
```

```
public void run(){
 for(int i = 0; i < 10; i++){
   account.withdraw(100); }
```

```
•  withdraw(double amount){
     lock.lock();
     while(balance <= 0){
         System.out.print("W");
         // waiting for the balance to exceed 0
         Thread.sleep(...);
     }
     balance -= amount;
     lock.unlock(); }

•  deposit(double amount){
     lock.lock();
     while(balance >= 300){
         System.out.print("D");
         // waiting for the balance to go below 300
         Thread.sleep(...)
     }
     balance += amount;
     lock.unlock(); }
```

How Can a Deadlock Occur?

- Suppose the withdrawal thread goes ahead.

Withdrawal thread

Acquires the lock.
If(balance <= 0) --> true
Prints "W"
Sleeps

Reaches the end of its time slice

Gain a time slice

If(balance <= 0) --> true
Prints "W"
Sleeps

Deposit thread

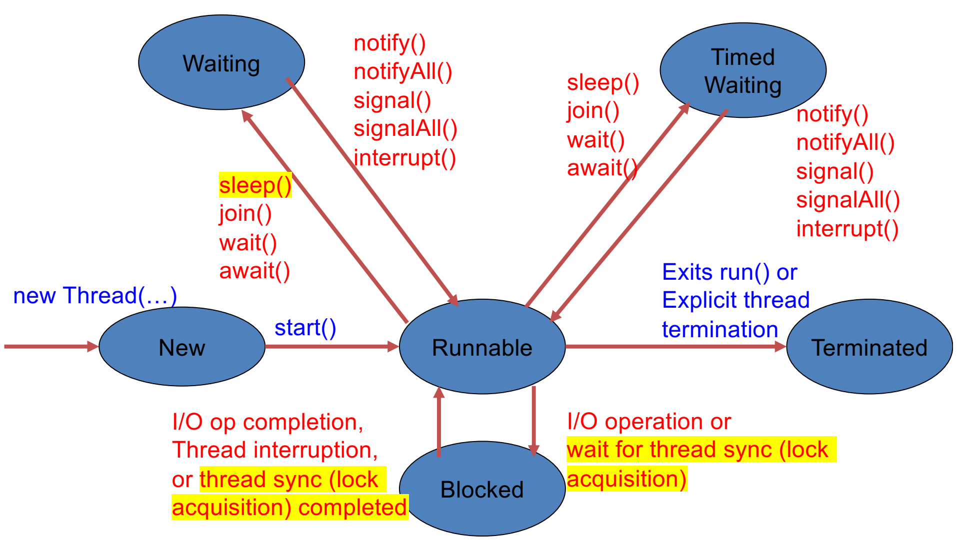
Gain a time slice

Tries to acquire the lock, but it is not available. Goes to the Blocked state.

Reaches the end of its time slice

The withdrawal thread never release the lock. Both threads never perform withdrawal nor deposit.

States of a Thread



Note

- A JVM can perform context switches even when a thread runs atomic code.
  - A lock guarantees that only one thread exclusively runs atomic code at a time.
    - It does NOT control when to (or when not to) perform context switches.
  - Some resources explicitly/implicitly say that context switches never occur when a thread runs atomic code.
    - It is WRONG!

DeadlockedBankAccount2.java

- Previous version

```
– withdraw(double amount){
    lock.lock();
    while( balance <= 0 ){
        System.out.print("W");
        Thread.sleep(...);
    }
    balance -= amount;
    lock.unlock();
  }

– deposit(double amount){
    lock.lock();
    while( balance >= 300 ){
        System.out.print("W");
        Thread.sleep(...)
    }
     balance += amount;
    lock.unlock();
  }
```

- New version

```
– withdraw(double amount){
    while( balance <= 0 ){
        System.out.print("W");
        Thread.sleep(...);
    }
    lock.lock();
    balance -= amount;
    lock.unlock();
  }

– deposit(double amount){
    while( balance >= 300 ){
        System.out.print("W");
        Thread.sleep(...);
    }
    lock.lock();
    balance += amount;
    lock.unlock();
  }
```


- Has no deadlock problems.
- Can generate race conditions.

A Potential Race Condition in DeadlockedBankAccount2

Withdrawal thread

If(balance <= 0) --> true

Reaches the end of
its time slice

Gain a time slice

print out “W”
sleep()
If(balance <=0) --> false
Withdraw money

Deposit thread

Gain a time slice

Acquire the lock
print(“current balance…); // 0
balance += amount;
// balance == 100
println(“new balance…); // 100
Release the lock

Reaches the end of
its time slice (balance==100)

These two lines should not be executed
because balance > 0 at this moment.

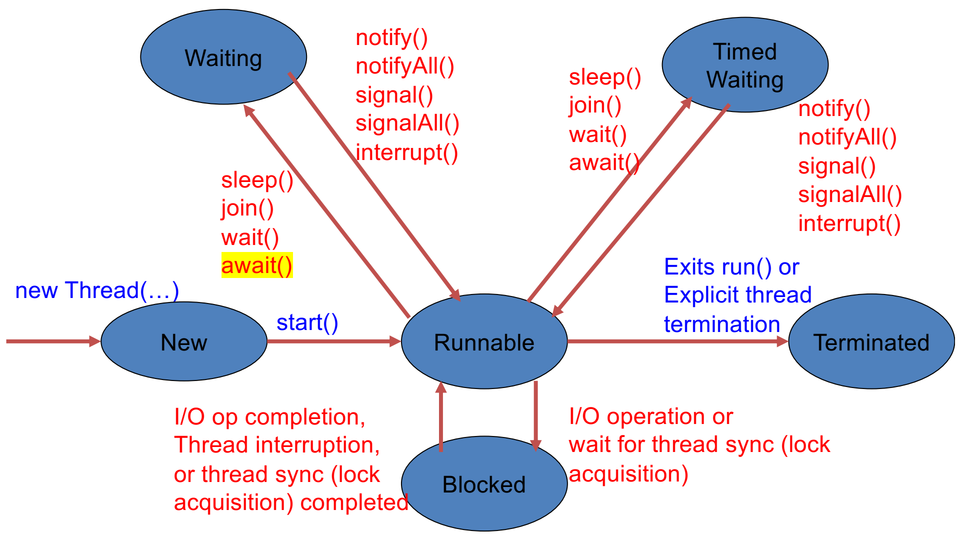
Avoiding Deadlocks and Race Conditions

- Use a `Condition` object.
  - `java.util.concurrent.locks.Condition`
  - Allows a thread to
    - Temporarily release a lock, so another thread can acquire it and proceed.
    - Re-acquire the lock later.

- Use a lock to obtain a `Condition` object

```
– ReentrantLock lock = new ReentrantLock();
  Condition condition = lock.newCondition(); // Factory method
  lock.lock();
  condition.await();// Temporarily releases the lock.
                    // Goes to the Waiting state until getting
                    // signaled by another thread.
```

States of a Thread



ThreadSafeBankAccount2.java

```
• Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();

• withdraw(double amount){
    lock.lock();
    while(balance <= 0){
      // Wait for the balance to exceed 0
      sufficientFundsCondition.await(); }
    balance -= amount;
    belowUpperLimitFundsCondition.signalAll();
    lock.unlock(); }

• deposit(double amount){
    lock.lock();
    while(balance >= 300){
      // Wait for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
    balance += amount;
    sufficientFundsCondition.signalAll();
    lock.unlock(); }
```


If a "Withdrawal" Thread Runs First and balance<=0…

```
• Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();

• withdraw(double amount){
   lock.lock();
   while(balance <= 0){
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await();}// temporarily releases
                                       // the lock and goes to
                                       // the Waiting state til
                                       // getting signaled by
                                       // another thread
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
```



```
• Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();

• withdraw(double amount){
   lock.lock();
   while(balance <= 0){
     // Wait for the balance to exceed 0
     sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }

• deposit(double amount){
   lock.lock();
   while(balance >= 300){
     // Wait for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
```

A "deposit" thread calls signalAll() to wake up a withdrawal thread(s) that is waiting until balance > 0.

It then releases the lock.

```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();

withdraw(double amount){
  lock.lock();
  while(balance <= 0){
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await(); }
  balance -= amount;
  belowUpperLimitFundsCondition.signalAll();
  lock.unlock(); }

deposit(double amount){
  lock.lock();
  while(balance >= 300){
     // Wait for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
  balance += amount;
  sufficientFundsCondition.signalAll();
  lock.unlock(); }
```

The “withdrawal” thread acquires the lock.

It then withdraws some money.

If a “Deposit” Thread Runs First and balance>=300

```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();

withdraw(double amount){
  lock.lock();
  while(balance <= 0){
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await(); }
  balance -= amount;
  belowUpperLimitFundsCondition.signalAll();
  lock.unlock(); }

deposit(double amount){
  lock.lock();
  while(balance >= 300){
     // Wait for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
  balance += amount;
  sufficientFundsCondition.signalAll();
  lock.unlock(); }
```

Temporarily releases the lock and goes to the Waiting state until balance < 300


```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();

withdraw(double amount){
  lock.lock();
  while(balance <= 0){
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await(); }
  balance -= amount;
  belowUpperLimitFundsCondition.signalAll();
  lock.unlock(); }

deposit(double amount){
  lock.lock();
  while(balance >= 300){
     // Wait for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
  balance += amount;
  sufficientFundsCondition.signalAll();
  lock.unlock(); }
```

A “withdrawal” thread calls signalAll() to wake up a “deposit” thread(s) that are waiting til balance < 300.

It then releases the lock.


```
Condition sufficientFundsCondition = lock.newCondition();
Condition belowUpperLimitFundsCondition = lock.newCondition();

withdraw(double amount){
  lock.lock();
  while(balance <= 0){
      // Wait for the balance to exceed 0
     sufficientFundsCondition.await(); }
  balance -= amount;
  belowUpperLimitFundsCondition.signalAll();
  lock.unlock(); }

deposit(double amount){
  lock.lock();
  while(balance >= 300){
     // Wait for the balance to go below 300.
     belowUpperLimitFundsCondition.await(); }
  balance += amount;
  sufficientFundsCondition.signalAll();
  lock.unlock(); }
```

The “withdrawal” thread acquires the lock.

It then deposit some money.

Two-Way (Bidirectional) Signaling b/w 2 Types of Threads

- ```
  Condition sufficientFundsCondition = lock.newCondition();
  Condition belowUpperLimitFundsCondition = lock.newCondition();
  ```

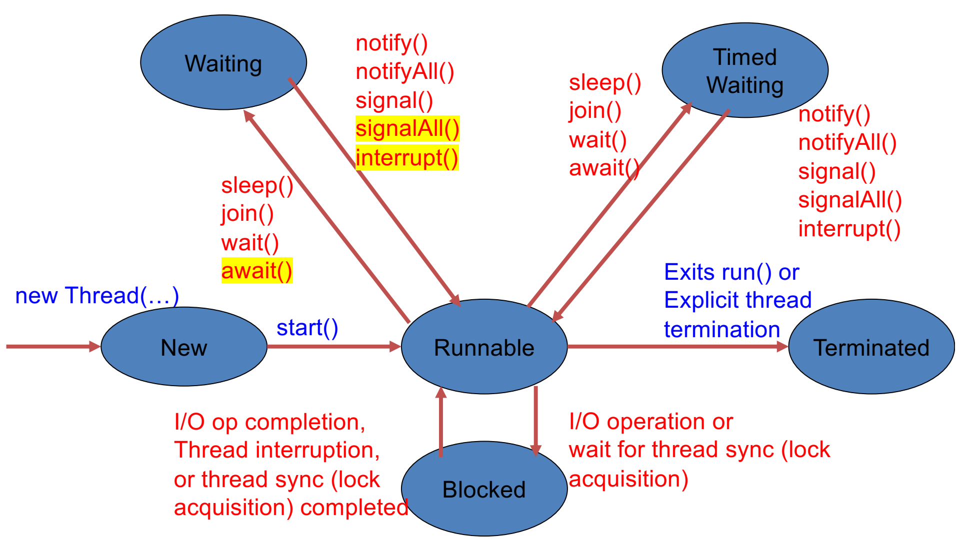
- ```
  withdraw(double amount){
   lock.lock();
   while(balance <= 0){
       // Wait for the balance to exceed 0
      sufficientFundsCondition.await();
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

- ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // Wait for the balance to go below 300.
      belowUpperLimitFundsCondition.await();
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```


Condition

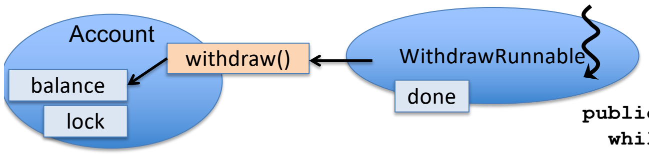
- **`await()`**
  - Blocks (i.e., gets stuck, or does not return) until getting signaled or interrupted
  - Blocks until getting signaled or interrupted, or until a specified waiting time (relative time) has elapsed.
  - Blocks until getting signaled or interrupted, or until a specified deadline (absolute time).
  - Throws an **`InterruptedException`**, if getting interrupted.
    - c.f. A previous lecture note on thread interruption
- **`signalAll()`**
  - Wakes up all the threads that wait on a condition object.
    - All of them go to the “Runnable” state and race to re-acquire a lock.
    - One of them will successfully re-acquire the lock.
      - All others will go to the “Blocked” state upon lock re-acquisition failures.
      - There are no ways to control which one of them to re-acquire the lock.

States of a Thread



- When a thread calls **`await()`** or **`signalAll()`** on a **`Condition`** object,
  - the thread is assumed to hold a lock associated with the **`Condition`** object.
  - If the thread does not, an **`IllegalMonitorStateException`** is thrown.
  ```

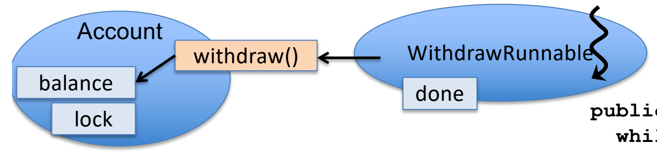
Introducing 2-Step Thread Termination in `ThreadSafeBankAccount2.java`




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

public void run(){
  while(true){
    if(done){break;}
    account.withdraw(100);}}
```

- Define a volatile flag (**`done`**) in **`WithdrawRunnable`**
  - Or, use an **`AtomicBoolean`**
- Have the main thread terminate each "withdrawal" thread by flipping the flag with **`setDone()`**.



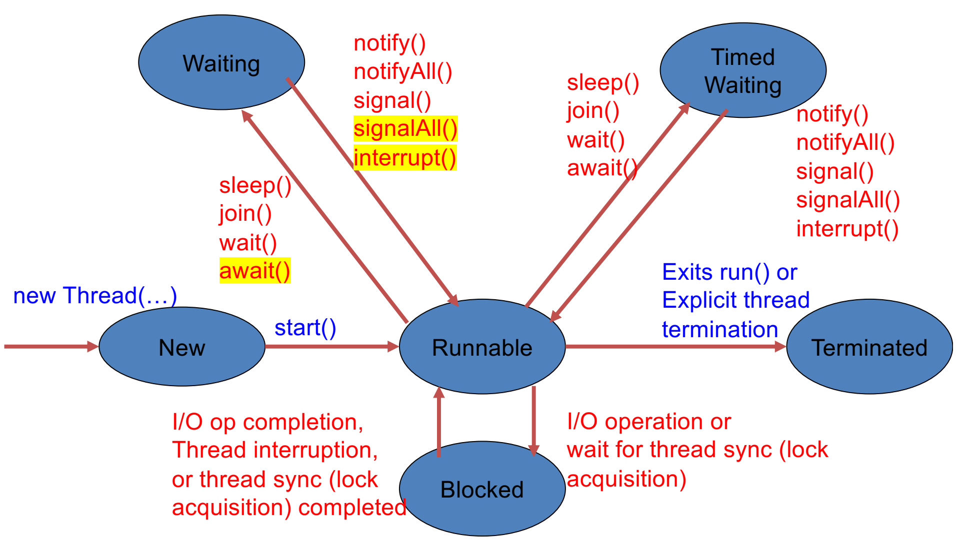

```
public void setDone(){
  done=true; } }

public void run(){
  while(true){
    if(done){break;}
    account.withdraw(100);}}
```

```
withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       try{
          // waiting for the balance to exceed 0
          sufficientFundsCondition.await();
       }catch(InterruptedException e){
          return;   // Returning withdraw()
       }
   }
    balance -= amount;
    belowUpperLimitFundsCondition.signalAll();
    lock.unlock();  }
```

- Have the main thread call **`interrupt()`**, after calling **`setDone()`**, on each "withdrawal" thread.
  - To let "withdrawal" threads wake up and catch an **`InterruptedException`**, if they are in the Waiting state due to **`await()`**.

States of a Thread



- If you don't do 2-step thread termination but do a simple flag-based thread termination (i.e., if the main thread doesn't call **`interrupt()`**)…
  - The main thread may not be able to terminate "withdrawal" threads forever.
    - If they are in the Waiting state and no other threads deposit sufficient money.
  - The bank app will never shut down.

HW 12

- Complete 2-step thread termination in **`ThreadSafeBankAccount2.java`**
  - in **`WithdrawRunnable`** and **`DepositRunnable`**
  - in **`withdraw()`** and **`deposit()`** threads.
  - Run multiple “withdrawal” and “deposit” threads
  - Have the main thread terminate both “withdrawal” and “deposit” threads.

`signalAll()` *Before* or *After* a State Change?

- ```
  withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

- ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```

- What if you call **`signalAll()`** first and then update the balance? Does it cause any problems?

- ```
  withdraw(double amount){
   lock.lock();
   while(balance =< 0){
       // waiting for the balance to exceed 0
      sufficientFundsCondition.await(); }
   belowUpperLimitFundsCondition.signalAll();
   balance -= amount;
   lock.unlock(); }
  ```

- ```
  deposit(double amount){
   lock.lock();
   while(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   sufficientFundsCondition.signalAll();
   balance += amount;
   lock.unlock(); }
  ```

- For example, do you have to worry about potential race conditions in this case? The answer is NO.

Suppose balance==0.
(1) W thread:
Calls await().
Releases the lock temporarily and goes to the “waiting” state.
Then, assume a context switch here.

```
withdraw(double amount){
 lock.lock();
 while(balance <= 0){
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 belowUpperLimitFundsCondition.signalAll();
 balance -= amount;
 lock.unlock(); }
```

D thread: wakes up W thread, which is waiting for the balance to be >0

(2) D thread:
Calls signalAll().
Then, assume a context switch.

```
deposit(double amount){
 lock.lock();
 while(balance >= 300){
    // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
 sufficientFundsCondition.signalAll();
 balance += amount;
 lock.unlock(); }
```

Can the “W” thread withdraw money before the “D” thread deposits money? (If this is possible, it’s a problem.) The answer is NO.

(1) W thread: Calls await(). Releases the lock temporarily and goes to the "waiting" state. Then, assume a context switch here.

(3) W thread: Wakes up and goes to the "runnable" state. Tries to acquire the lock again and **fails**. Goes to the "blocked" state.

```
withdraw(double amount){
  lock.lock();
  while(balance <= 0){
    // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
  belowUpperLimitFundsCondition.signalAll();
  balance -= amount;
  lock.unlock(); }
```

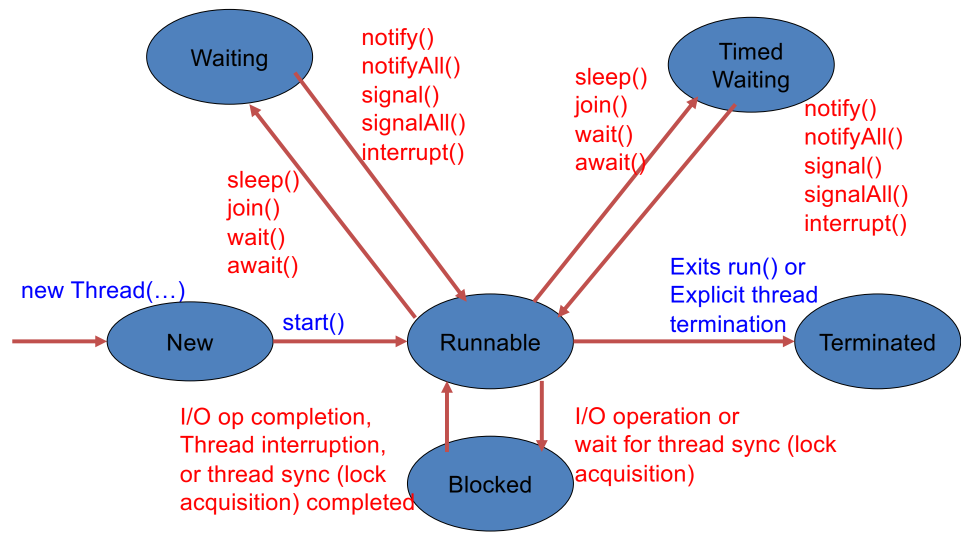
(2) D thread: Calls signalAll(). Then, assume a context switch.

D thread: wakes up W thread

```
deposit(double amount){
  lock.lock();
  while(balance >= 300){
    // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
  sufficientFundsCondition.signalAll();
  balance += amount;
  lock.unlock(); }
```

- "W" thread CANNOT withdraw money before "D" thread deposits money.

States of a Thread



Two Important Things (1)

- You can safely change the value of a shared variable <u>even after</u> calling **signalAll()**.
  - AS FAR AS the value changes in atomic code (i.e. before calling **unlock()**)
- That said, common practice is:
  - A state change first, followed by **signalAll()**.

Two Important Things (2)

- A JVM can perform context switches even when a thread runs atomic code.
  - A lock guarantees that only one thread exclusively runs atomic code at a time.
    - It does NOT control when to (or when not to) perform context switches.
  - Some resources explicitly/implicitly say that context switches never occur when a thread runs atomic code.
    - It is WRONG!

signal() and signalAll()

- signalAll()
  - Wakes up all “waiting” threads on a condition object.
    - All of them go to the “runnable” state.
    - They race to re-acquire a lock. Only one of them will actually re-acquire it. The others will go to the “blocked” state.
- signal()
  - Wakes up one of “waiting” threads on a condition object.
    - The selected thread goes to the “runnable” state. The others stay at the “waiting” state.
    - JVM’s thread scheduler selects one of them. Assume a random selection.
      - Not predictable which waiting thread to be selected.

signal() and signalAll()?

- Either one works well.
- **`signalAll()`** is favored in many cases/projects.
  - I prefer **`signalAll()`** in my personal taste.

“while” or “if” to Surround await()?

```
withdraw(double amount){
 lock.lock();
 while(balance <= 0){
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 balance -= amount;
 belowUpperLimitFundsCondition.signalAll();
 lock.unlock(); }
```

```
deposit(double amount){
 lock.lock();
 while(balance >= 300){
    // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
 balance += amount;
 sufficientFundsCondition.signalAll();
 lock.unlock(); }
```

- “while” should be used rather than “if” when multiple threads call **`withdraw()`**/**`deposit()`** concurrently. Why?

A Potential Problem with “if”

(1) Suppose balance==0.
Two W threads call await(). They go to the “waiting” state and release the lock temporarily.
Then, assume a context switch.

```
withdraw(double amount){
 lock.lock();
 if(balance =< 0){
     // waiting for ...
    sufficientFundsCondition.await(); }
 balance -= amount;
 belowUpperLimitFundsCondition.signalAll();
 lock.unlock(); }
```

(2) D thread:
Deposits $100.
Calls signalAll().
Then, assume a context switch.

```
deposit(double amount){
 lock.lock();
 if(balance >= 300){
    // waiting for the balance to go below 300.
    belowUpperLimitFundsCondition.await(); }
 balance += amount;
 sufficientFundsCondition.signalAll();
 lock.unlock(); }
```

(3) Two W threads go to the “runnable” state.
One of them acquires the lock again. The other one goes to the “blocked” state.

The 1st W thread withdraws $100 (balance==0).

(1) Suppose balance==0.
Two W threads call await(). They go to the "waiting" state and release the lock temporarily.
Then, assume a context switch.

(3) Two W threads go to the "runnable" state.
One of them acquires the lock again. The other one goes to the "blocked" state.

The 1st W thread withdraws $100 (balance==0).

- ```
  withdraw(double amount){
   lock.lock();
   if(balance =< 0){
       // waiting for ...
       sufficientFundsCondition.await();
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

(4) The 2nd W thread acquires the lock after the 1st one releases it, and then withdraws $100 (balance==-100).

(2) D thread:
Deposits $100.
Calls signalAll().
Then, assume a context switch.

- ```
  deposit(double amount){
   lock.lock();
   if(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```

(1) Suppose balance==0.
Two W threads call await(). They go to the "waiting" state and release the lock temporarily.
Then, assume a context switch.

(3) Two W threads go to the "runnable" state.
One of them acquires the lock again. The other one goes to the "blocked" state.

The 1st W thread withdraw $100 (balance==0).

- ```
  withdraw(double amount){
   lock.lock();
   if(balance =< 0){
       // waiting for ...
       sufficientFundsCondition.await();
   balance -= amount;
   belowUpperLimitFundsCondition.signalAll();
   lock.unlock(); }
  ```

(4) The 2nd W thread acquires the lock after the 1st one releases it, and then withdraws $100 (balance==-100).

(2) D thread:
Deposits $100.
Calls signalAll().
Then, assume a context switch.

- ```
  deposit(double amount){
   lock.lock();
   if(balance >= 300){
      // waiting for the balance to go below 300.
      belowUpperLimitFundsCondition.await(); }
   balance += amount;
   sufficientFundsCondition.signalAll();
   lock.unlock(); }
  ```

- The 2nd "W" thread should have made sure if balance>0 in (4).
- If only one "W" thread runs, this problem does not occur.
- Just always use a while loop regardless of the number of threads you use.

"if" in Atomic Code?

- If you want, you can use "if", rather than "while," for the conditional in atomic code
  - if you use **`signal()`**, not **`signalAll()`**.
- However, in practice, the **while-signalAll** pair is more common than the **if-signal** pair.