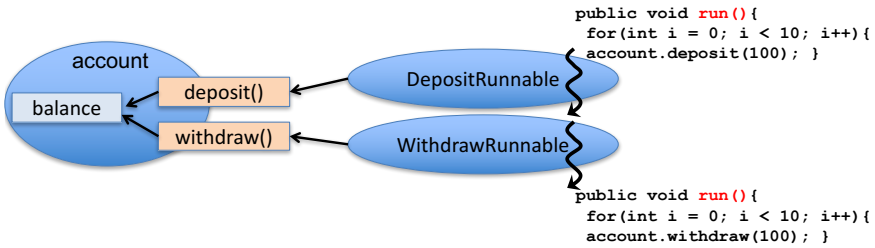
Deadlock

Thread Safety Issues

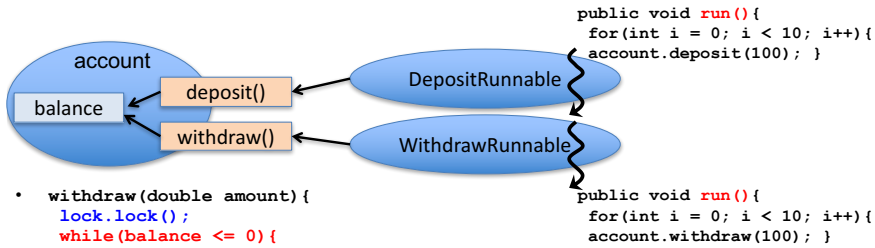
- Race conditions
- Deadlocks

- Thread-safe code is free from race conditions and deadlocks.

DeadlockedBankAccount.java



DeadlockedBankAccount.java

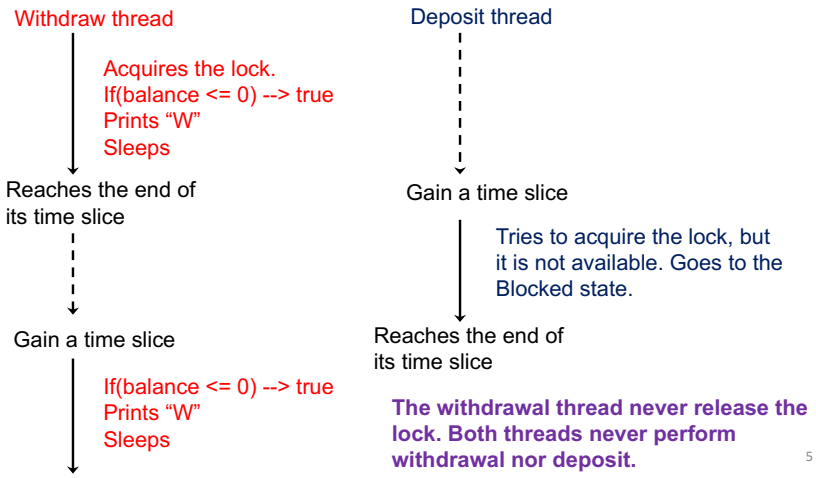



```
withdraw(double amount){
 lock.lock();
 while(balance <= 0){
     System.out.print("W");
     // waiting for the balance to exceed 0
     Thread.sleep(1000);
 }
 balance -= amount;
 lock.unlock(); }
```

```
deposit(double amount){
 lock.lock();
 while(balance >= 300){
     System.out.print("W");
     // waiting for the balance to go below 300
     Thread.sleep(1000)
 }
 balance += amount;
 lock.unlock(); }
```

How Can a Deadlock Occur?

- Suppose the withdrawal thread goes ahead.



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.
- Some resouces explicitly/implicitly say that context switches never occur when a thread runs atomic code.
  - It is WRONG!

DeadlockedBankAccount2.java

- Previous version

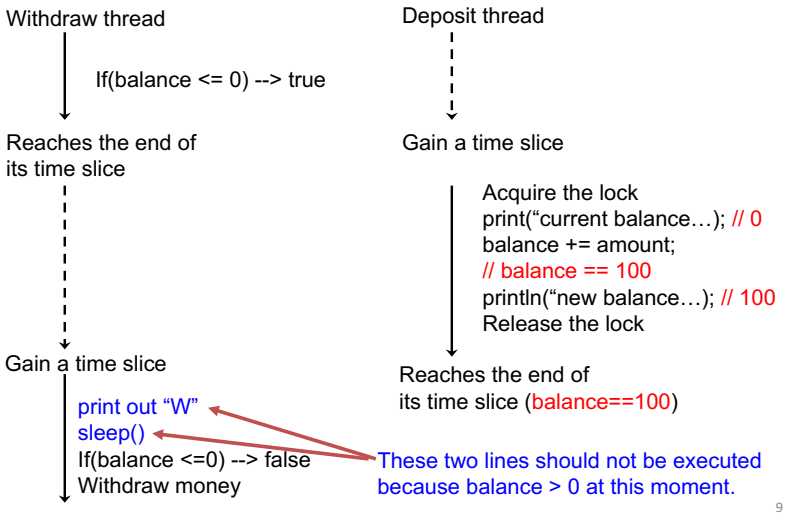
```
withdraw(double amount){
    lock.lock();
    while(balance <= 0){
        System.out.print("W");
        // waiting for the
        // balance to exceed 0
        Thread.sleep(1000);
    }
    balance -= amount;
    lock.unlock();
}

deposit(double amount){
    lock.lock();
    while(balance > 300){
        System.out.print("W");
        // waiting for the balance
        // to go below 300
        Thread.sleep(1000)
    }
     balance += amount;
    lock.unlock(); }
```

- New version

```
withdraw(double amount){
    while( balance <= 0 ){
        System.out.print("W");
        Thread.sleep(2);
    }
    lock.lock();
    balance -= amount;
    lock.unlock();
}

deposit(double amount){
    while( balance > 300 ){
        System.out.print("W");
        Thread.sleep(2);
    }
    lock.lock();
    balance += amount;
    lock.unlock();
}
```

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

A Potential Race Condition in DeadlockedBankAccount2

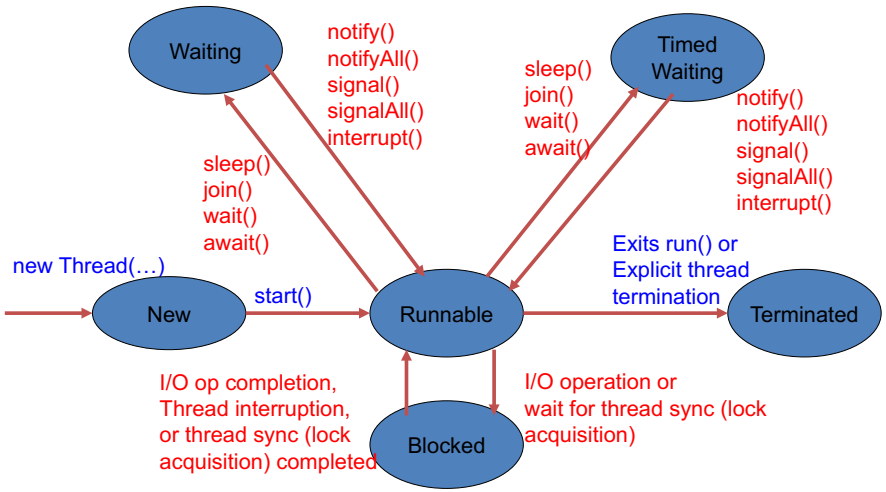


Avoiding Deadlocks and Race Conditions

- Use a `Condition` object(s).
  - Allows a thread to
    - Temporarily release a lock so that another thread can acquire it and proceed.
    - Re-acquire the lock later.
- `java.util.concurrent.locks.Condition`
  - Obtain its instance from a lock object

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition(); // Factory method
condition.await();  // Temporarily releases the lock
                    // Goes to the Waiting state
```


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(); }
```

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(); }
```

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

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(); }
```

A "withdraw" thread calls signalAll() to wake up a thread(s) that is/are waiting until balance < 300.

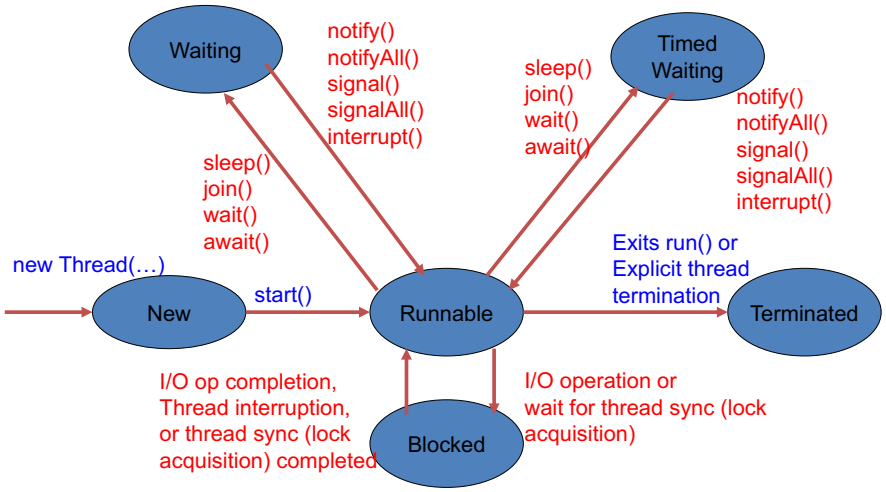
ThreadSafeBankAccount2.java

```
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(); }
```


Condition

- **`await()`**
  - Will wait until it is signaled or interrupted
  - Will wait until it is signaled or interrupted, or until a specified waiting time (relative time) elapsed.
  - Will wait until it is signaled or interrupted, or until a specified deadline (absolute time).
  - If signaled, goes to the Runnable state and re-acquires a lock.
    - Will be "blocked" if the thread re-acquisition fails.
  - Throws an `InterruptedException` if interrupted.
    - c.f. A previous lecture note on thread interruption
- **`signalAll()`**
  - Wakes up all waiting threads on a condition object.
    - All of them go to the "runnable" state.
    - One of them will re-acquire a lock.

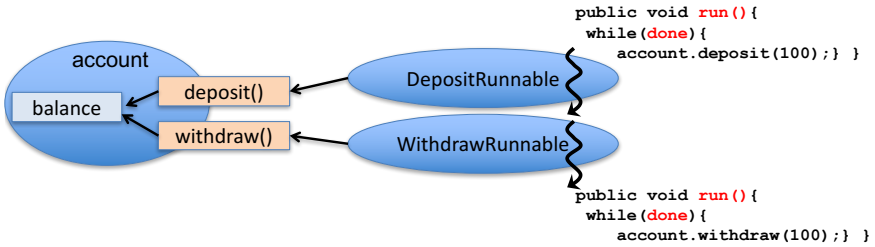
States of a Thread



- When a thread calls await(), signal() 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.

2-Step Thread Termination

(c.f. Lec Note #9)




```
public void run(){
  while(done){
     account.deposit(100);} }
```

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

```
withdraw(double amount){
    lock.lock();
    while(balance =< 0){
        try{
            // waiting for the balance to exceed 0
            sufficientFundsCondition.await();
        }catch(InterruptedException e){
            break; // Balking
        }
    }
    ...}
```

HW 13

- Implement 2-step termination for “deposit” and “withdraw” threads.
  - Implement a flag-based termination scheme in DepositRunnable and WithdrawRunnable
    - To let “deposit” and “withdraw” threads to return run().
  - Have the main thread call interrupt() on “deposit” and “withdraw” threads
    - To let those threads to wake up in case they are in the Waiting state due to await() or sleep().
- Due: April 24 (Tue) midnight

`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? Will any thread safety issues come out?

```
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 need to worry about race conditions in this case?

(1) W thread: "waiting" temporarily releases the lock

(2) D thread: signalAll(). Ctx switch

```
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(); }
```

D thread: wakes up W thread, which is waiting until balance > 0.

- Can the "W" thread withdraw money before the "D" thread deposits money?
  - Can the balance have a negative value?
    - The answer is NO.

(1) W thread: "waiting" temporarily releases the lock

(2) D thread: signalAll(). Ctx switch

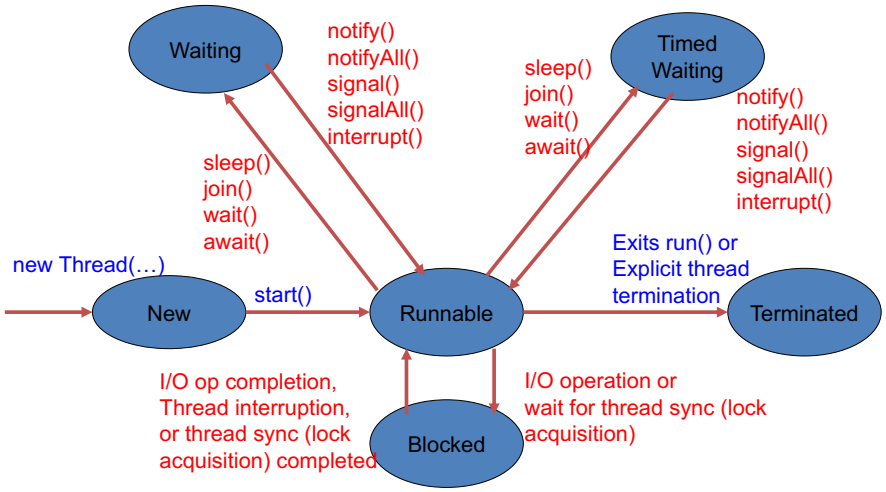
```
withdraw(double amount){
 lock.lock();
 while(balance =< 0){
     // waiting for the balance to exceed 0
    sufficientFundsCondition.await(); }
 belowUpperLimitFundsCondition.signalAll();
 balance -= amount;
 lock.unlock(); }
```

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

(3) W thread: "runnable" Tries to acquire the lock again and fails. Goes to "blocked."

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

States of a Thread



Two Important Things (1)

- You can safely change the state/value of a shared variable after calling signalAll().
  - AS FAR AS the state changes in atomic code
- Common programming convention/practice:
  - 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.
- Some resources (books, online materials, etc.) 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.
    - One of them will re-acquire a lock. The others will go to the “waiting” 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 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.

ThreadSafeBankAccount2.java

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

```
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(); }
```

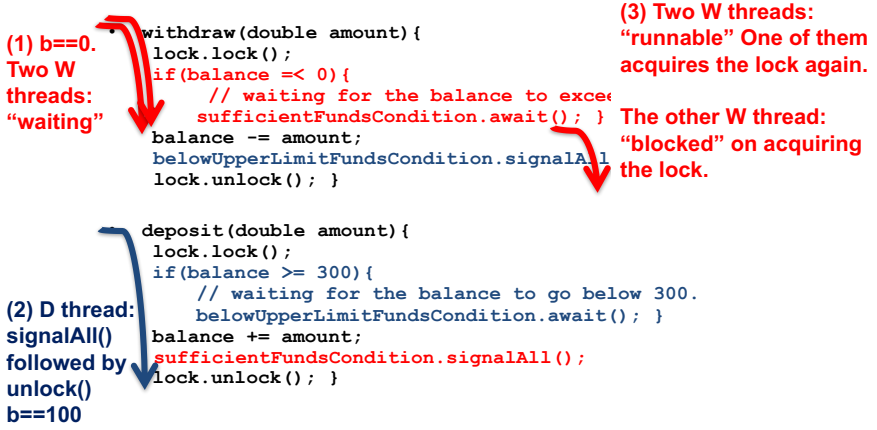

"while" or "if" to Surround await()?

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

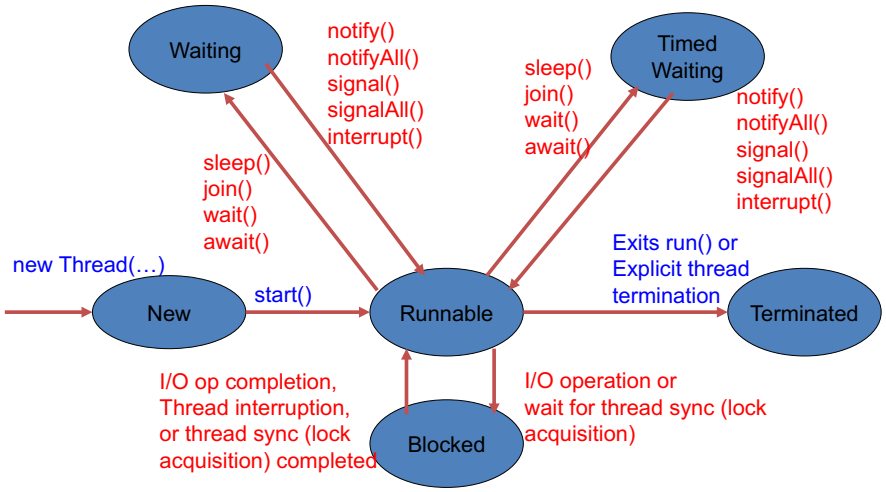
```
withdraw(double amount){
 lock.lock();
 if(balance =< 0){
      // waiting for the balance to exceed 0
     sufficientFundsCondition.await(); }
 balance -= amount;
 belowUpperLimitFundsCondition.signalAll();
 lock.unlock(); }
```

- "while" should be used rather than "if" when multiple threads call withdraw() concurrently. Why?

A Potential Problem



States of a Thread



(1) b==0. Two W threads: "waiting"

(2) D thread: signalAll() followed by unlock() b==100

(3) Two W threads: "runnable" One of them acquires the lock again, withdraws money and releases the lock. b==0.

(4) The 2nd W thread acquires the lock and withdraws money. b==-100.

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

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) b=0. Two W threads: "waiting"

(2) D thread: signalAll() followed by unlock() balance==100

(3) Two W threads: "runnable" One of them acquires the lock again and releases it. balance==0. The other W thread: "blocked" on acquiring the lock.

(4) The 2nd W thread acquires the lock and releases it. b==-100.

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

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.
- 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.

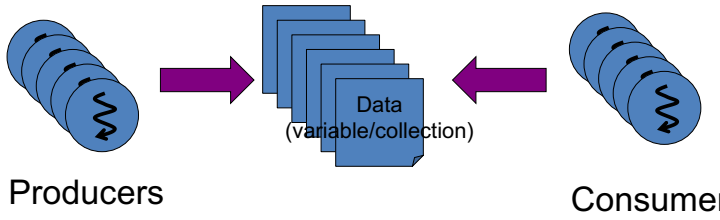
"if" or "while" in Atomic Code?

- You can use "if", rather than "while," for a conditional check
  - if you use signal(), not signalAll().
- However, in practice, the **while-signalAll** pair is more common than the **if-signal** pair.

Producer-Consumer Design Pattern

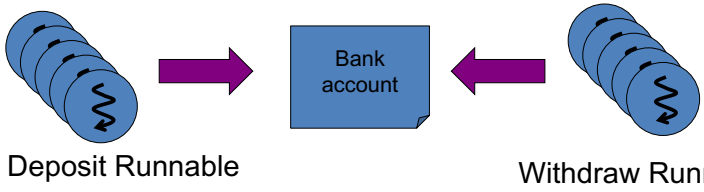
Producer-Consumer Design Pattern

- Producer
  - Generates data to be processed.
- Consumer
  - Take and process those generated data.
  - If no data is available, wait until a producer generates data.



Bank Account Example

- DepositRunnable (Producer)
  - A thread (or a group of threads) that deposits money to a bank account.
  - If the current balance is over the upper limit, the thread(s) wait(s) until the balance goes below the upper limit.
- WithdrawRunnable (Consumer)
  - A thread (or a group of threads) that withdraws money from the account.
  - If the current balance is below the lower limit, the thread(s) wait(s) until the balance exceeds the lower limit.

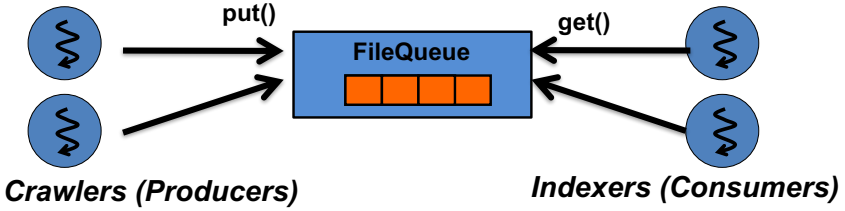


An Exercise: File Indexing Service

- Imagine an indexing service for a file system
  - e.g., Windows indexing service and Mac/iOS's Spotlight
- Key functionalities
  - Scan/crawl files in the local file system
  - Index those files for later file searches.
    - Extract and keep each file's metadata
      - Metadata: file's attributes (e.g., location, name, file type, author) and file's content

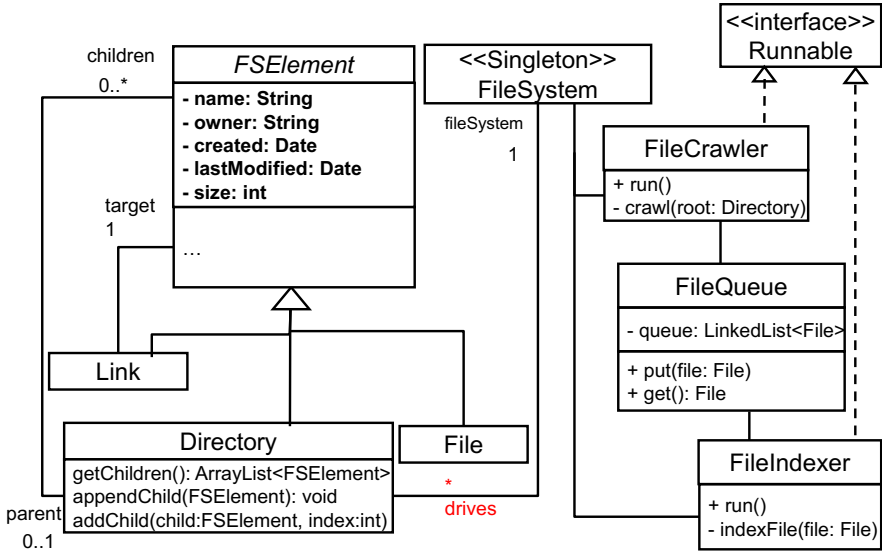
Threads in File Indexing

- Single threaded
  - Use a single thread for both crawling and indexing
- Multi-threaded
  - Use different threads for crawling and indexing
    - One crawling thread and one indexing thread
    - Multiple crawling threads and multiple indexing threads
  - More efficient than the single-threaded version in multi-core environments
  - Crawlers and indexers interact with each other based on the Producer-Consumer design pattern.



- Assume multiple crawler threads and multiple indexer threads.
  - One crawler thread per a drive.
- A crawler thread
  - Traverses a tree structure in a given drive and puts files to the queue.
  - Waits, if the queue's size reaches a certain number (upper limit), until the size becomes below the limit.
  - Dies when it completes to traverse a given tree structure or when the main thread tells it to die.
- An indexer thread
  - Gets a file from the queue and indexes it.
    - Waits, if no files are available in the queue, until a crawler puts a new file.
  - Repeats this forever until the main thread tells it to die.

HW 14: Implement this.



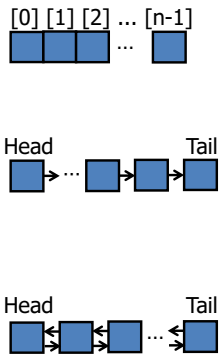

```
class FileCrawler implements Runnable{
  private Directory root; //root dir of a given drive (tree structure)
  private FileQueue queue;
  ...
  public void run(){
    crawl(root);
    ...
  }
  private void crawl(Directory root){
    // Crawl a given drive (tree structure)
    // Put files to a queue. Ignore directories and links.
    queue.put( file );
  }
}
class FileIndexer implements Runnable{
  private FileQueue queue;
  ...
  public void run(){
    while(true){
      indexFile( queue.get() );
    }
  }
  public indexFile(File file){
    // Index a given file.
  }
}
```

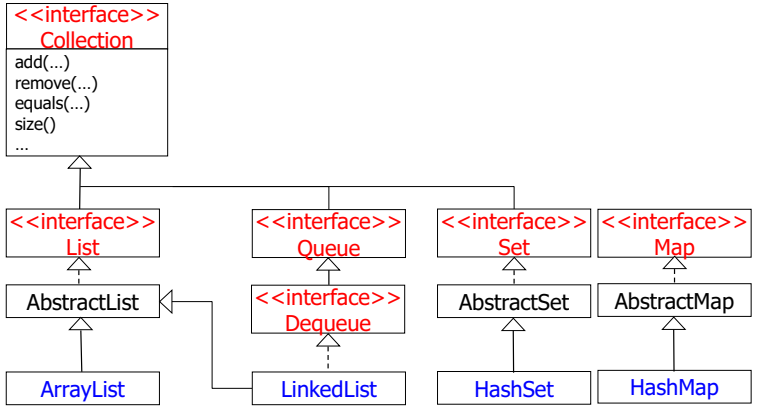
- No need to include a interactive command-line I/F
- No need to filter files in crawl()
  - crawl() can queue all files to the file queue
- No need to implement actual indexing logic.
  - indexFile() can just print out each file's metadata (e.g. file name) on a shell.
- Make multiple drives and assign a crawler thread to each drive.
- Run multiple crawler threads and multiple indexer threads from main().
- Have the main thread terminate crawler and indexer threads.
  - Use 2-step thread termination.

- Use LinkedList<File> in FileQueue.

Just in Case: Major Collection Types in Java

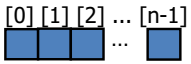
- List
  - Orders elements with their integer index numbers.
  - Offers index-based random access.
  - Can contain duplicate elements.
- Queue
  - Orders elements with links.
  - Offers FIFO (First-In-First-Out) access.
  - Can contain duplicate elements.
- Dequeue
  - Stands for "Double Ended QUEUE" (pronounced "deck").
  - Orders elements with links.
  - Offers both FIFO and LIFO (Last-In-First-Out) access.
  - Can contain duplicate elements.
- Set
  - Contains non-duplicate elements without an order.
- Map
  - Contains key-value pairs (w/ non-duplicate keys) without an order.



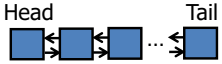


ArrayList v.s. LinkedList

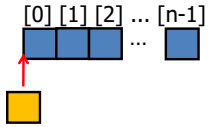
- ArrayList
  - Array-based implementation of the List interface



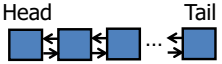
- LinkedList
  - Doubly-linked implementation of the List and Deque interfaces



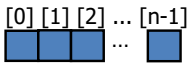
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



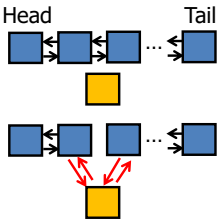
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces



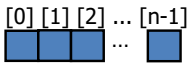
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



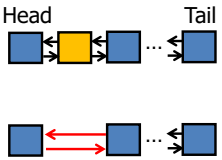
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements



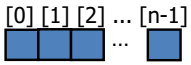
- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



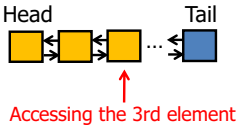
- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements



- ArrayList
  - Array-based impl of the List interface
  - Fast index-based access
  - Slow insertion and removal of non-tail elements
    - Fast insertion and removal of the tail element



- LinkedList
  - Doubly-linked impl of the List and Deque interfaces
  - Fast insertion and removal of elements
  - Slow index-based access for "middle" elements.



- Use ArrayList
  - If you often need to access "middle" elements.
- Use LinkedList
  - If you often need to insert/remove elements.
- Both yield the same performance for element traversal (i.e. sequential element access).

- Due: May 1 (Tue) midnight