CS 2100: Data Structures & Algorithms 1

Concurrency

Race Conditions and Synchronization
Avoiding Deadlocks
Blocking Queue

Dr. Nada Basit // basit@virginia.edu

Spring 2022

Friendly Reminders

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - Just come up to me at the start of class and ask!
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, please stay home
  - *We will work with you!*
  - At home: eye mask instead! Get some rest ☺

Announcements

- **Final Exam:**
  - Date: Saturday, May 7, 2022
  - Time/Duration: 7:00pm – 9:00pm ET (two hours)
  - Location: Section 002: **RDL G008** *new* [Section 001: McLeod Hall 1020]
- **Make-up Exam:** [Email me if you haven't already]
  - If you have a conflict with the following courses, **email** me:
    - APMA 3100
    - APMA 3140
    - ECON 2020 (sections 001 and 002 only)
  - Make-up Date: Sunday, May 8, 2022
  - *At this time we do not have a time or a location; however, given there are no officially held final exams on this day (May 8) we anticipate the chosen time will suit your schedule*

The Final Exam – Saturday, May 7 (Make-up: May 8)

- Mode: Taken **in-person**
- Duration: **two (2)** hours
- Policies:
  - Closed-book / Closed-notes
  - Closed-Google/Internet (except to access the quiz itself)
  - Closed-Eclipse/other IDE
  - Closed-friend/any other person
  - Closed… everything ☺
  - Can retake **as many quizzes** as you want
  - The work you do must represent your **individual effort**, and involve **no outside assistance** from any one or any resource
- Location of Quizzes: ONLINE AS BEFORE.
  Explicit instructions will be given on the day!
- Students with accommodations with **SDAC**:
  - Please see email that I have sent to you.
  - If you choose to book a testing appointment with SDAC, please do so as soon as possible!
  - You will have your extended time accommodations
- What to bring with you to the final exam:
  - Fully charged **laptop** (+ charging cable)
  - Pen/pencil to write on scratch paper (*not necessary, only if you want*)
  - Student ID card

From Last Time... Try/Catch/Finally

- Yes, but Java's concurrency libraries **throw a lot of exceptions**, so you will often need to use `try/catch statements` to handle those.
- REMEMBER...

```
public void run(){
  for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
    //DO STUFF
  }
  //Clean up if necessary
}
```

- NOW IT WILL BE:

```
public void run(){
  try{
    for(int i=0; i<=REPETITIONS; i++){
      //DO STUFF
      sleep(1000) //thread waits. but might throw exception
    }
  }
  catch(InterruptedException e){
    //Do something if you want here
  }
  finally{
    //clean everything up
  }
}
```

Sleep Method throws an InterruptedException

- The **sleep** method throws an `InterruptedException` when a sleeping thread is **interrupted**

```
public void run(){
  try{
    for(int i=0; i<=REPETITIONS; i++){
      //DO STUFF
      sleep(1000) //thread waits. but might throw exception
    }
  }
  catch(InterruptedException e){
    //Do something if you want here
  }
  finally{
    //clean everything up
  }
}
```

Eclipse DEMO

MyRunnableWithInterrupt.java – *To illustrate try/catch, interrupt() and sleep()*

```java
public void run() {
    // What order will statements [1] to [4] be executed?
    // Will all be executed?
    try {
        System.out.println("[1] Before thread goes to sleep");
        Thread.sleep(DELAY); // sleep for one second
        System.out.println("[2] After thread sleeps for one second");
    }
    catch (InterruptedException exception) {
        // if thread is interrupted it will throw an "InterruptedException"
        // and will be caught here
        System.out.println("[3] Inside catch -- thread was interrupted!");
    }
    System.out.println("[4] Outside of try-catch");
}
```

Eclipse DEMO

WordCount.java && WordCountRunnable.java – *Counting words in parallel*

Race Conditions and Synchronization

(Also an understanding of shared resources)

Race Conditions

- Consider the following program that contains two threads:
  - Variable int **amount** in an account
  - **Thread 1**: repeatedly deposits $100 into the account (n times)
  - **Thread 2**: repeatedly withdraws $100 into the account (n times)

- What would happen? We should end up with $0.00, right??

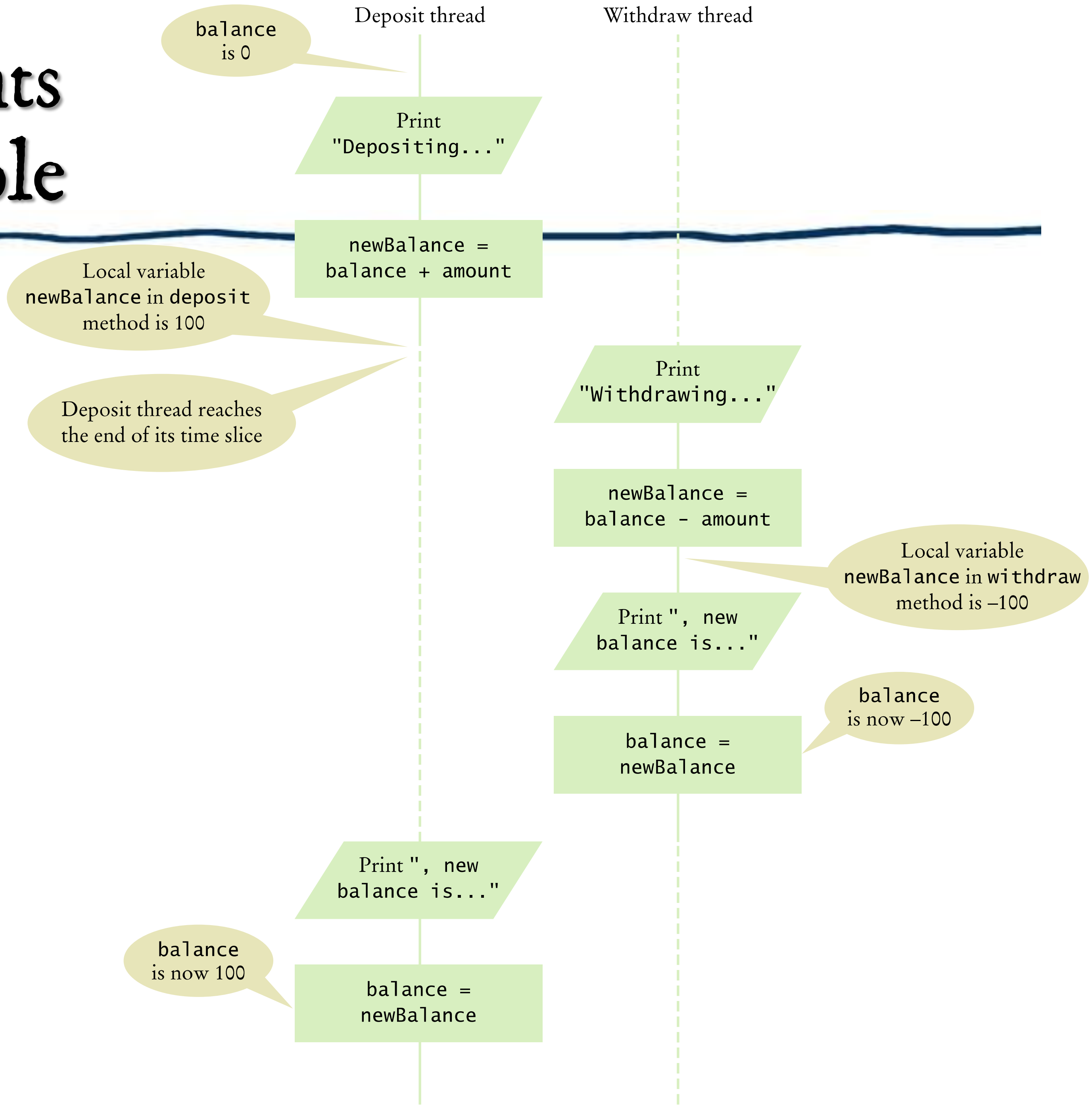
Eclipse DEMO

Watch the following demos presented in class:

Bank Example: **[Thread Example 4 – Bank]**

BankAccount.java
BankAccountThreadRunner.java
DepositRunnable.java
WithdrawRunnable.java

Corrupting the Contents of the balance Variable



Results?

```
Withdrawing 100.0, new balance is 1100.0
Depositing 100.0Depositing 100.0, new balance is 1200.0
Withdrawing 100.0Depositing 100.0, new balance is 1300.0
Withdrawing 100.0, new balance is 1200.0
Depositing 100.0, new balance is 1300.0
Depositing 100.0, new balance is 1400.0
Withdrawing 100.0, new balance is 1300.0
, new balance is 1200.0
Depositing 100.0, new balance is 1300.0
Withdrawing 100.0, new balance is 1200.0
```

- Did we get a balance of 0.0? No…!
- Why? This is called a **race condition**!

Race Conditions

- Occurs if the effect of multiple threads on ***shared data*** *depends on the* ***order*** *in which they are scheduled*
  - i.e., the threads are racing, and the output depends on which one is faster
- It is possible for a thread to reach the end of its time slice in the middle of a statement
- It may evaluate the right-hand side of an equation but **not** be able to store the result until its next turn:

```
public void deposit(double amount)
{
    System.out.print("Depositing " + amount);
    double newBalance = balance + amount;
    System.out.println(", new balance is " + newBalance);
    balance = newBalance;
}
```

- Race condition can still occur:

```
balance = the right-hand-side value does not get assigned
```

Race Conditions – Why Does it Happen??

- Race conditions can occur when there are **shared resources**: variables or objects that multiple threads are interacting with at once.
- This code:

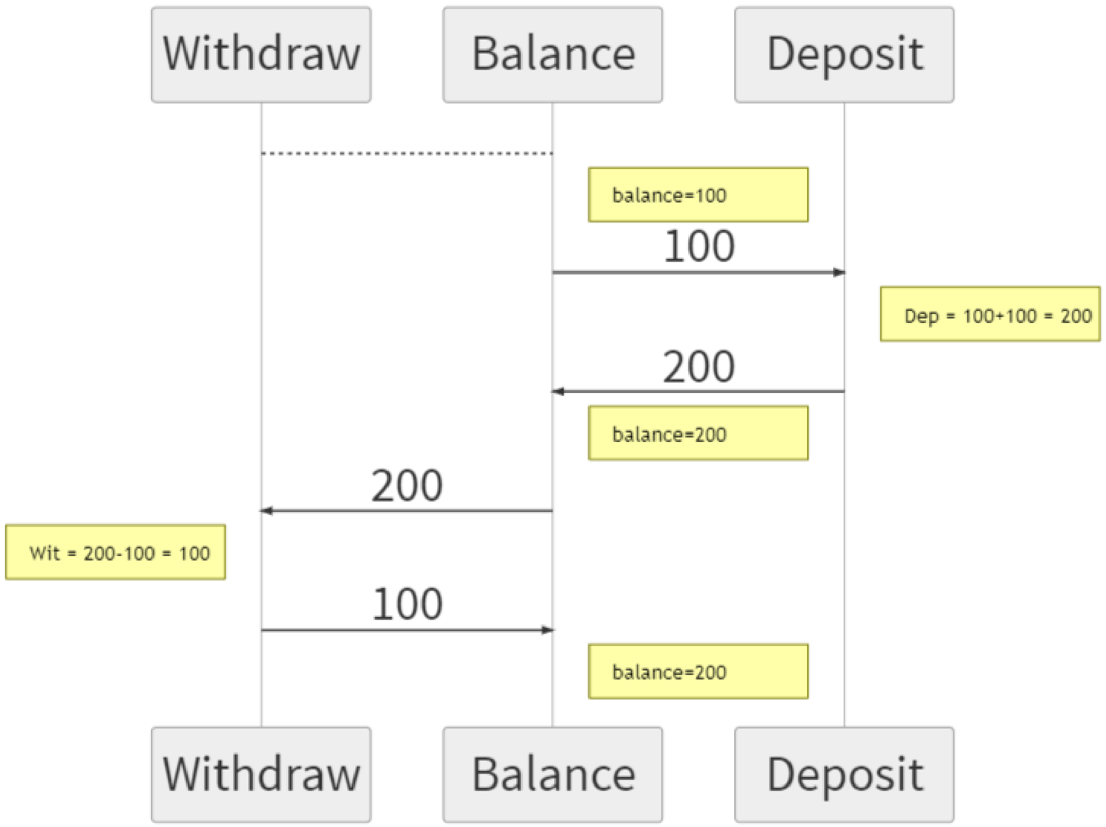
```
double newBalance = balance + amount;
```

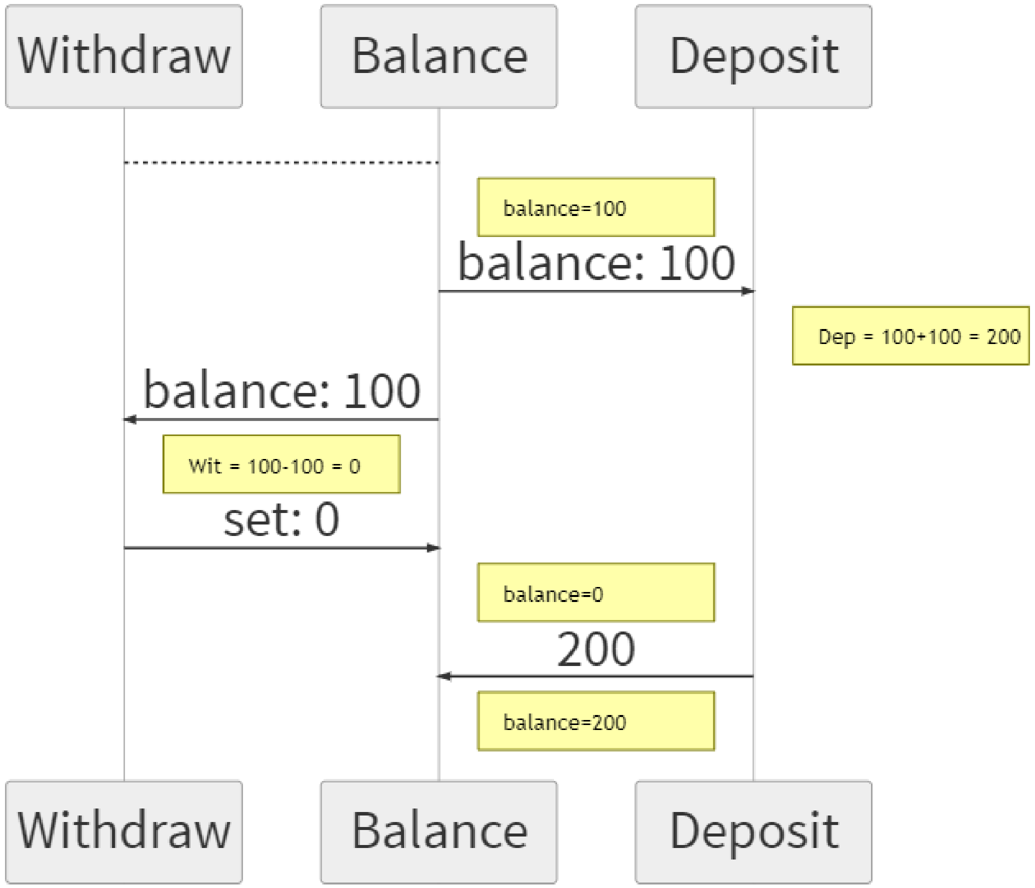
- Is turned into (by the compiler) something like this:

```
#I don't expect you to fully understand this
mov rax, balance
mov rax, amount
mov balance, rax
```

- This means that a thread may have calculated that the new balance is 100+100 = 200 **BEFORE** storing that result (200) back into the variable balance.
- If the thread is interrupted at that point, then ***bad things could happen***.

What We Hope Happens //What Might Happen Instead





Locks

To synchronize object access, we use locks!

Fixing Race Conditions: Use Locks!

- To solve problems such as the one just seen (race condition), use a *lock object*
- **Lock object:** used to control threads that manipulate shared resources
  - It is a resource that only one thread is allowed to "hold" at a single time
    - Forces threads to "take turns"
    - But also usually slows execution down because one thread may have to wait on the other
- Many types of lock are out there, we will use Java's **ReentrantLock** *(most commonly used lock class)*
  - Inherits from the Lock interface

Locks

- When there is a **shared resource**, we usually instantiate a **lock**:

```java
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;  // ** Add a lock **


    /**
      Constructs a bank account with a zero balance
    */
    public BankAccount(){
        balance = 0;
        balanceChangeLock = new ReentrantLock(); // ** lock **
    }
```

Locks

- Code that manipulates a **shared resource** is *surrounded* by calls to **lock** and **unlock**.
- So, when we use the **shared resource**, we grab the lock first:

```
balanceChangeLock.lock();
double newBalance = balance + amount;
balanceChangeLock.unlock();
```

- If lock() is called, and another thread has the lock, this thread will **wait**.

Locks

```
balanceChangeLock.lock();
double newBalance = balance + amount;
balanceChangeLock.unlock();
```

- ***… But there is a problem!***
- If code between calls to lock and unlock throws an ***exception***, call to unlock never happens!

```
balanceChangeLock.lock();
double newBalance = balance + amount;
/* Exception thrown HERE - code afterwards does not get executed */
balanceChangeLock.unlock();
```

- To resolve this, use **try/catch** instead – place a call to unlock into the **finally** clause:

```
balanceChangeLock.lock();
try {
    double newBalance = balance + amount;
    /* Exception thrown HERE */
}
//catch { /* Stuff here */ }
finally {
    balanceChangeLock.unlock();
}
```

Final Deposit Code

```
public void deposit(double amount){

    balanceChangeLock.lock();  // ** lock! **
    try
    {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
    finally
    {
        balanceChangeLock.unlock(); // ** unlock! **
    }
}
```

- … and similar code for the withdraw method!

Final Notes on Locks

- When a thread calls **lock**, **in owns the lock** until **unlock** is called
- Another thread that calls **lock** will be ***deactivated*** by the **scheduler** so that it "waits" for the lock.
  - Occasionally the thread scheduler **reactivates** a thread so it can try to acquire the **lock** (see if the lock is now available)
- Eventually (hopefully) the waiting thread can acquire the **lock**

Eclipse DEMO

Watch the following demos presented in class:

Bank Example: **[Thread Example 5 – Bank Sync]**

BankAccount.java
BankAccountThreadRunner.java
DepositRunnable.java
WithdrawRunnable.java

Avoiding Deadlocks

Avoiding Deadlocks

Let's try to model the real world; if you go to a bank and try to withdraw money, you can only withdraw an amount less than or equal to the size of your balance.

If your balance is $50, you **cannot** withdraw $100! (You don't have a "negative" balance!)

Let's see how to make our code mimic this realistic real-world behavior

Deadlocks

- A **Deadlock** is a problem that occurs when no thread can proceed because each is waiting on another.
  - e.g., thread A is waiting on B which is waiting on C which is waiting on A
  - No progress is made, and the program freezes forever.

a condition needs to be checked (an appropriate test) →

```
public void withdraw(double amount) {

    balanceChangeLock.lock();  // lock!
    try {
        // Check condition:
        while (balance < amount) {
            // WAIT FOR BALANCE TO GROW...
        }
        //... Rest of Withdraw Code ...
    }
    finally {
        balanceChangeLock.unlock(); // unlock!
    }
```

Banking Example

- **How can we wait for the balance to grow?**
  - We cannot just **wait (or sleep)**, because the thread owns the **balanceChangeLock**!
  - In particular, no other thread can successfully execute **deposit**
  - Other threads will call **deposit**, but will be blocked until **withdraw** exits
  - But **withdraw** doesn't exit until it has funds available (**withdraw** will never finish because **deposit** cannot happen…)
  - **DEADLOCK !!**

```
public void withdraw(double amount) {

    balanceChangeLock.lock();  // lock!
    try {
        // Check condition:
        while (balance < amount) {
            // WAIT FOR BALANCE TO GROW...
        }
        //... Rest of Withdraw Code ...
    }
    finally {
        balanceChangeLock.unlock(); // unlock!
    }
```

Overcoming Deadlocks: Condition Objects

- To overcome deadlocks, use Java's **condition object**.
- **Condition objects** allow a thread to temporarily release a lock until a condition is met, and then reacquire the lock
- This is done autonomously, so no race conditions within this acquisition step
- Each condition object belongs to a **specific** lock object

Condition Objects

```
public class BankAccount {
    private double balance;
    private Lock balanceChangeLock;  // lock
    private Condition sufficientFundsCondition; // Add condition object


    /**
      Constructs a bank account with a zero balance
    */
    public BankAccount(){
        balance = 0;
        balanceChangeLock = new ReentrantLock(); // lock
        // condition object associated with specific
        // lock object (balanceChangeLock)
        sufficientFundsCondition = balanceChangeLock.newCondition();
    }
```

Condition object given a name that *describes* the condition

condition object belongs to a *specific* lock object

Condition Objects

```
public void withdraw(double amount) throws InterruptedException {

    balanceChangeLock.lock();  // lock!
    try {
        // Check condition:
        while (balance < amount) {
            // If balance is less than withdrawal amount...
            // Condition object calls "await"
            sufficientFundsCondition.await();   // await!
            // Another thread can now acquire the lock object
        }
        System.out.print("Withdrawing " + amount);
        double newBalance = balance - amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
    }
    finally {
        balanceChangeLock.unlock(); // unlock!
    }
}
```

the condition needs to be checked

the condition object calls "await"

As long as the test is not fulfilled, call ***await*** *on the condition object*

Condition Objects

- Calling **await** makes the current thread **wait** and allows other threads to acquire the lock object
- To **unblock** the waiting thread, another thread must execute **signalAll** *(on the same condition object)*

```
sufficientFundsCondition.signalAll();
```

- **signalAll** unblocks all threads waiting on the condition
  - This lets other threads know that the condition might now be met for the waiting thread. Gives control back to waiting threads

Signaling

```
public void deposit(double amount){

    balanceChangeLock.lock();  // lock!
    try {
        System.out.print("Depositing " + amount);
        double newBalance = balance + amount;
        System.out.println(", new balance is " + newBalance);
        balance = newBalance;
        // Funds added to balance...
        // Unblock other threads waiting on the condition by "signalAll"
        sufficientFundsCondition.signalAll();
    }
    finally {
        balanceChangeLock.unlock(); // unlock!
    }
}
```

Results?

```
Depositing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
Depositing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
...
Withdrawing 100.0, new balance is 100.0
Depositing 100.0, new balance is 200.0
Withdrawing 100.0, new balance is 100.0
Withdrawing 100.0, new balance is 0.0
```

Notice how the balance doesn't drop below zero! This is a more realistic situation and we can achieve this by using locks and condition objects

Eclipse DEMO

Watch the following demos presented in class:

Bank Example: **[Thread Example 6 – Bank Deadlock]**

BankAccount.java
BankAccountThreadRunner.java
DepositRunnable.java
WithdrawRunnable.java

Blocking Queue

Using concurrency: Locks and conditions

Concurrent Queue

- Suppose we have a **linked-list backed queue** and we want to be able to access the queue with **multiple threads.**
- Doesn't seem too bad, should be able to enqueue at front at same time as dequeuing at back.
- This is your assignment this week!

Blocking Concurrent Queue

- **Enqueue** - Lock the queue, then add the element
  - Once an element is added, then **signalAll** to waiting **dequeue** threads *(why? See below)*
- **Dequeue** - Lock the queue, then delete the element
  - If no nodes to delete, then **await** a signal that something (an **enqueue** thread) has been added
  - This is the "blocking" part because the queue will **wait** until it can delete something

- Note that there are more efficient ways to implement this, but this is sufficient for our assignment.