Principles of Software Development

V. Adamchik

CSCI 201

Lecture 9

USC

More on Threads: Monitors, Locks, Conditions

Reading: chapter 32.

Review

```
public class AddAPenny implements Runnable {
   private static PiggyBank bank = new PiggyBank();
   public void run() {bank.deposit(1);}
   public static void main(String [] args) {
      ExecutorService exec = Executors.newCachedThreadPool();
      for (int i=0; i < 100; i++)
            exec.execute(new AddAPenny());
     exec.shutdown(); // shutdownNow();
     while(!exec.isTerminated()) {Thread.yield(); }
} }
class PiggyBank{
   private int balance = 0;
   public int getBalance(){return balance;}
   public synchronized void deposit(int amount)   {
       int newBalance = balance + amount;
       try {Thread.sleep(2);} catch (InterruptedException ex) {}
       balance = newBalance;
   }
```

Monitor Overview

Prior to Java 5, thread communications were programmed using the object's built-in monitors. If you work with legacy Java code, you may encounter Java's built-in monitor.

A monitor is an object with mutual exclusion and synchronization capabilities. Any object can be a monitor. An object becomes a monitor once a thread locks it. Locking is implemented using the synchronized keyword on a method or a block.

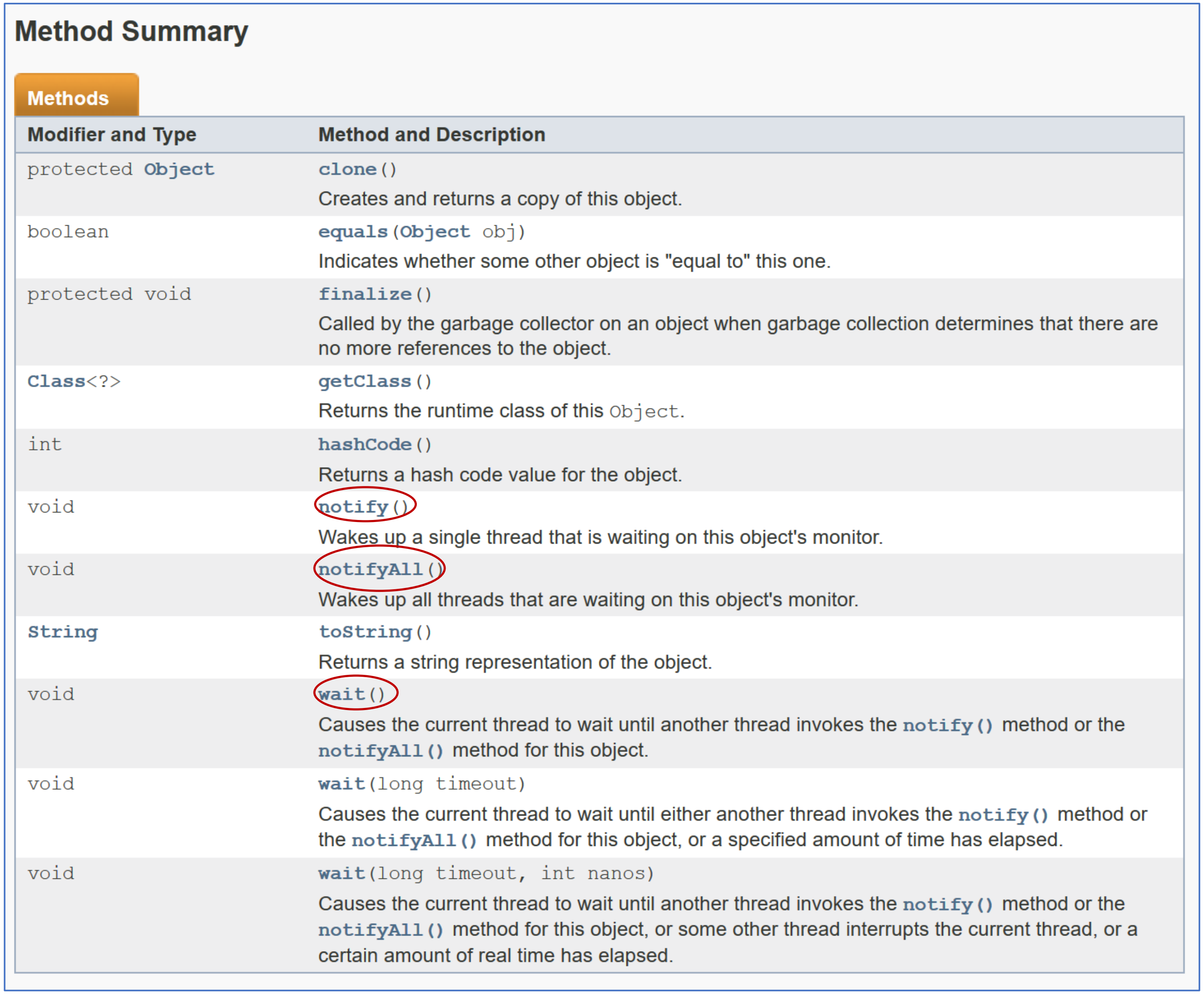
The monitor functionality is implemented in the Object class.

A thread can wait() in a monitor; this will release the lock on the object.

That thread must then be awakened using notify() or notifyAll() from another thread to be moved back into the Ready state.

The wait(), notify() and notifyAll() methods must be called in a synchronized method or a synchronized block.

Object class



Method Summary

Methods

Modifier and Type	Method and Description
protected Object	clone() Creates and returns a copy of this object.
boolean	equals(Object obj) Indicates whether some other object is "equal to" this one.
protected void	finalize() Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
Class<?>	getClass() Returns the runtime class of this Object.
int	hashCode() Returns a hash code value for the object.
void	notify() Wakes up a single thread that is waiting on this object's monitor.
void	notifyAll() Wakes up all threads that are waiting on this object's monitor.
String	toString() Returns a string representation of the object.
void	wait() Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void	wait(long timeout) Causes the current thread to wait until either another thread invokes the notify() method or the notifyAll() method for this object, or a specified amount of time has elapsed.
void	wait(long timeout, int nanos) Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object, or some other thread interrupts the current thread, or a certain amount of real time has elapsed.

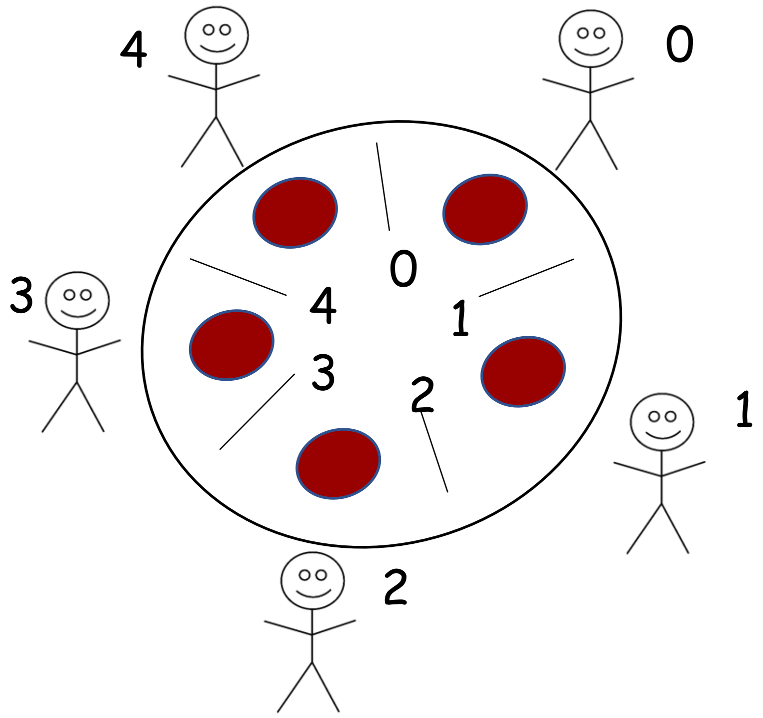
Dining Philosopher Problem

We will implement the Dining Philosopher Problem using the Monitor concept.

Create the Chopsticks class, that has two synchronized method (pickup() and putdown()) for taking a chopstick and then putting it back.

Create the Philosopher class that extends Thread. The run method() is an infinite loop in which a philosopher

1. Thinks
2. Gets hungry
3. Eats



Possible Output

```
Philosopher 0 is thinking
Philosopher 3 is thinking
Philosopher 2 is thinking
Philosopher 1 is thinking
Philosopher 4 is thinking
Philosopher 0 is hungry.
Philosopher 0 takes the chopstick 0
Philosopher 0 takes the chopstick 1
Philosophers 0 is eating
Philosopher 2 is hungry.
Philosopher 2 takes the chopstick 2
Philosopher 2 takes the chopstick 3
Philosophers 2 is eating
Philosopher 3 is hungry.
Philosopher 4 is hungry.
Philosopher 4 takes the chopstick 4
Philosopher 1 is hungry.
Philosopher 0 release the right chopstick
Philosopher 0 release the left chopstick
```

```java
public class DiningProblem
{
      public static void main(String[] args)
      {
            Chopstick[] chopsticks = new Chopstick[5];
            Philosopher[] philosopher=  new Philosopher[5];
            for (int i=0; i<5;i++)
            {
                  chopsticks[i] = new Chopstick(1);
            }
            for (int i=0; i<5;i++)
            {
                  philosopher[i] = new Philosopher(i, chopsticks);
                  philosopher[i].start();
            }
      }
}
```

```
class Chopstick {
        private boolean isAvailable;
        public Chopstick(boolean value){isAvailable = value;}

        public synchronized void pickup() {







        }

        public synchronized void putdown() {




        }
}
```

```
class Philosopher extends Thread {
   private int name;
   private Chopstick chopsticks[];
   public Philosopher(int name, Chopstick chopsticks[]){
   this.name = name; this.chopsticks = chopsticks;
   }
   public void run() {
```

Monitor Overview

Can the wait-and-notify mechanism be used to replace the synchronized mechanism?

The answer is no; wait-and-notify does not solve the *race condition* (two or more threads can access shared data) problem that the synchronized mechanism solves.

The wait-and-notify mechanism is more of a communication mechanism: it allows one thread to communicate to another thread that a particular condition has occurred.

It does not specify what the specific condition is.

Synchronization Using Locks

A synchronized instance method implicitly acquires a lock on the instance before it executes the method.

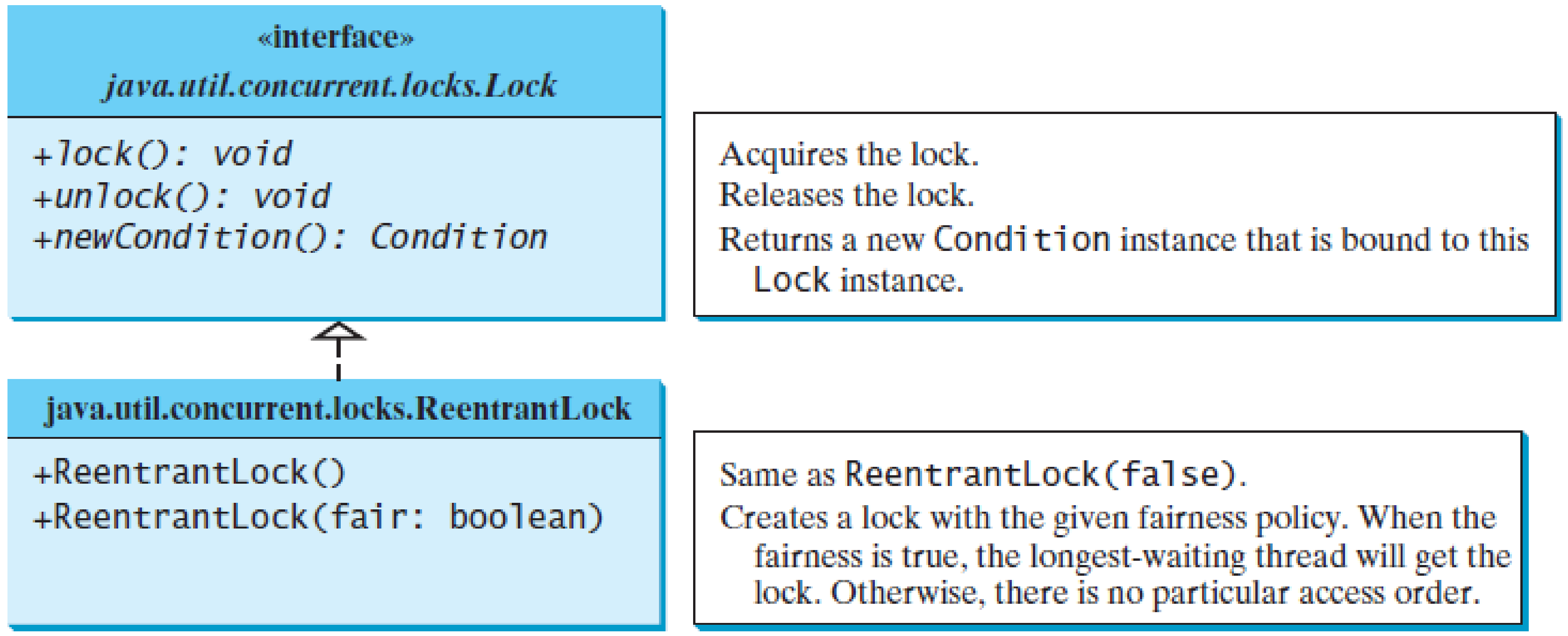
The synchronized keyword is quite rigid in its use. For example, a thread can take a lock only once. Synchronized methods/blocks don't offer any mechanism of a waiting queue and after the exit of one thread, any thread can take the lock. This could lead to *starvation* of resources for some other thread for a very long period of time.

Synchronization Using Locks

Java enables you to acquire locks explicitly, which give you more control for coordinating threads. A lock is an instance of the Lock interface. A lock may also use the newCondition() method to create any number of Condition objects.

Locks and conditions are more powerful and flexible than the built-in monitor, so will not need to use monitors.

Lock and ReentrantLock


«interface»
java.util.concurrent.locks.Lock
+lock(): void
+unlock(): void
+newCondition(): Condition
Acquires the lock.
Releases the lock.
Returns a new Condition instance that is bound to this Lock instance.
java.util.concurrent.locks.ReentrantLock
+ReentrantLock()
+ReentrantLock(fair: boolean)
Same as ReentrantLock(false).
Creates a lock with the given fairness policy. When the fairness is true, the longest-waiting thread will get the lock. Otherwise, there is no particular access order.

ReentrantLock class is a concrete implementation of Lock interface for creating mutually exclusive locks. You can create a lock with the specified *fairness policy*. True fairness policies guarantee that the longest waiting thread will obtain the lock first. False fairness policies grant a lock to a waiting thread arbitrarily.

Reentrant Locks

```
class some_class {
   private Lock lck = new ReentrantLock();

   public void some_method() {
        lck.lock();
        try {
            //Do some work
        }
        catch(Exception e) {
            e.printStackTrace();
        } finally {
            lck.unlock();
        }
}}
```

The unlock statement is always called in the finally block. It is recommended practice.

lock() method increments the hold count by 1.

unlock() method decrements the hold count by 1.

If the resource is not held by any other thread, then call to tryLock() acquires the lock and the hold count is incremented by one.

More on Explicit Locks

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*.

Why create a new locking mechanism that is so similar to synchronized? Intrinsic lock works fine in most situations but has some functional limitation:

- it is not possible to interrupt a thread waiting to acquire a lock,
- it is not possible to acquire a lock without being willing to wait for it forever,
- intrinsic locks also must be released in the same block of code in which they are acquired;

None of these are reasons to abandon synchronized, but in some cases, it is a more flexible locking mechanism.

AddAPennyWithLock Example

```
class PiggyBank {
    private int balance = 0;
    private Lock lock = new ReentrantLock();
    public int getBalance() {
        return balance;
    }
    public void deposit(int amount) {
        lock.lock();
        try {
            int newBalance = balance + amount;
            Thread.sleep(2);
            balance = newBalance;
        } catch (InterruptedException ex) {}
        finally {
            lock.unlock();
        }
}}
```

A thread cannot acquire a lock owned by another thread.

But a thread can acquire a loc that it already owns.

Allowing a thread to acquire t same lock more than once enables Reentrant synchronization.

Thread Conditions

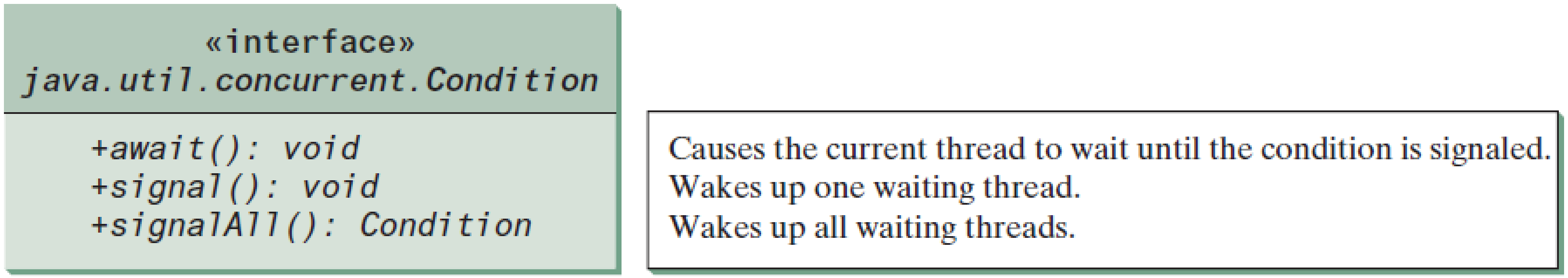
Thread synchronization suffices to avoid race conditions by ensuring the mutual exclusion of multiple threads in the critical region, but sometimes you also need a way for threads to cooperate.

Conditions can be used to facilitate communications among threads. A thread can specify what to do under a certain condition. Conditions are objects created by invoking the newCondition() method on a Lock object.

Once a condition is created, you can use await() , signal() , and signalAll() methods for thread communications.

These methods are similar to Object class' wait and notify methods.

Thread Conditions


«interface»
java.util.concurrent.Condition
+await(): void
+signal(): void
+signalAll(): Condition
Causes the current thread to wait until the condition is signaled.
Wakes up one waiting thread.
Wakes up all waiting threads.

The await() method causes the current thread to wait until the condition is signaled. If signal() or signalAll() is never called on the condition, the thread will wait forever.

You must first obtain the lock on the object before you are able to invoke a method on its Condition. When a thread calls await(), it will release its lock.

AddAndRemove Program

Let's modify our AddAPenny program to also allow threads for depositing and withdrawing.

```
public class AddAndRemove implements Runnable {
   private static PiggyBank piggy = new PiggyBank();
   private boolean isDeposit;

   public AddAndRemove(boolean task) {
              isDeposit = task;
   }
   public void run() {
      if (isDeposit)
         piggy.deposit((int)(Math.random() * 10 + 1));
      else
         piggy.withdraw((int)(Math.random() * 9 + 1));
   }
   public static void main(String [] args ) {
      ExecutorService exec = Executors.newCachedThreadPool();
      for (int i=0; i < 100; i++)
              exec.execute(new AddAndRemove(Math.random() < 0.5));
     exec.shutdown();
     while(!executor.isTerminated()){Thread.yield();}
     System.out.println("Balance = " + piggy.getBalance());
}}
```

```
class PiggyBank {
   private int balance = 10;
   private Lock lck = new ReentrantLock();
   public int getBalance() { return balance; }
   public void withdraw(int amount) {
      lck.lock(); // acquires lock
      try {
          while (balance < amount) {
              System.out.print("\tWaiting to withdraw $" + amount);
              System.out.println(" from balance of $" + balance + "...");
           }
           balance -= amount;
           System.out.println("\t$" + amount + " withdrawn; balance is $" + balance);
      }
      finally {lck.unlock();
}}
   public void deposit(int amount) {
      lck.lock();
      try {
           balance += amount;
           System.out.println("$" + amount + " deposited; balance is $" + balance);
      }
      finally{lck.unlock(); }
```

Output Explanation

Sometimes it works

```
    $9 withdrawn; balance is $1
$3 deposited; balance is $4
$1 deposited; balance is $5
.....
    $5 withdrawn; balance is $125
$5 deposited; balance is $130
Balance = 130
```

Sometimes we end up in an endless loop (deadlock) because we are stuck in the withdraw method's while loop, holding onto the lock, which does not allow any deposits.

```
Waiting to withdraw $4 from balance of $3...
Waiting to withdraw $4 from balance of $3...
Waiting to withdraw $4 from balance of $3...
Waiting to withdraw $4 from balance of $3...
Waiting to withdraw $4 from balance of $3...
Terminate batch job (Y/N)?
```

```
class PiggyBank {
  private int balance = 10;
  private Lock lock = new ReentrantLock();
  private Condition newDeposit = lock.newCondition();

  public int getBalance() { return balance; }
  public void withdraw(int amount) {
    lock.lock(); // acquires lock
    try {
      while (balance < amount) {
        System.out.print("\tWaiting to withdraw $" + amount);
        System.out.println(" from balance of $" + balance + "...");
        newDeposit.await();
      }
      balance -= amount;
      System.out.println("\t$" + amount + " withdrawn; balance is $" + balance);
    }
    finally { lock.unlock(); }
  }
  public void deposit(int amount) {
    lock.lock();
    try {
      balance += amount;
      System.out.println("$" + amount + " deposited; balance is $" + balance);
      newDeposit.signalAll();
    }
    finally{ lock.unlock(); }
} }
```

await()

From Javadoc:

The lock associated with this Condition is atomically released and the current thread becomes disabled for thread scheduling purposes and lies dormant until one of things happens:

- Some other thread invokes the signal() method for this Condition and the current thread happens to be chosen as the thread to be awakened; or
- Some other thread invokes the signalAll() method for this Condition; or
- Some other thread interrupts the current thread, and interruption of thread suspension is supported