Introduction to Java Concurrency

Anand S. Pandey
September 6, 2017

Agenda

Processes vs. Threads. Why threads ?

Java threads. Thread creation approaches.

What is Thread Safety ? Sharing Objects between threads

Synchronization approaches. Volatile and Final variables

Deadlock examples.

ExecutorService + Future + Callable example

Java Memory Model (JMM). Happens-Before relationship.

Process vs. Threads

- **Process** is a program in execution.
- **Thread** is a path of execution ***within*** a process. It is a basic unit of CPU utilization. It comprises of:
    - thread Id
    - a program counter
    - a register set
    - a stack
- **Thread** shares with other threads belonging to the **same** process, its *code* section, *data* section and other operating system resources.
    - See picture in next slide

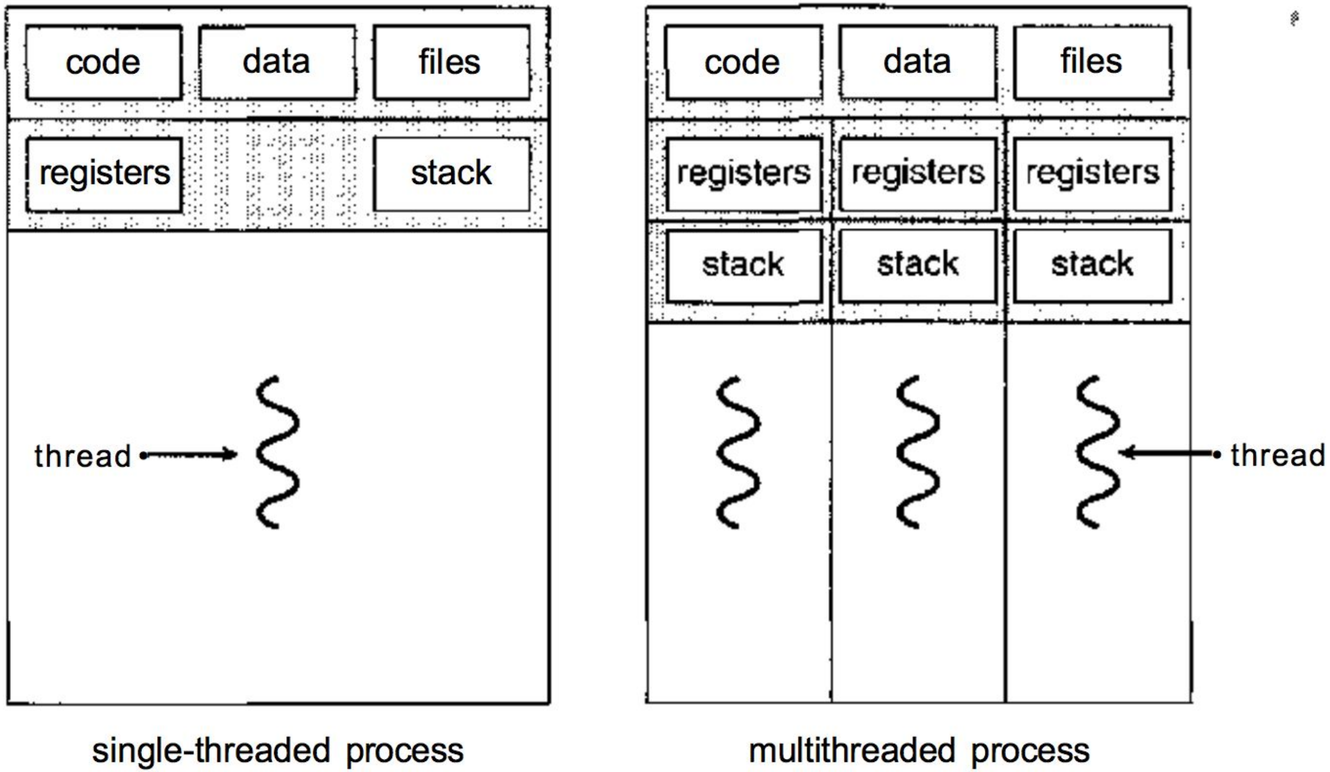


Figure 4.1 Single-threaded and multithreaded processes.

Benefits of Multithreaded programming

- Responsiveness
  - Allows a program to continue running even if parts of it is “waiting”
- Resource sharing
  - Threads share memory and resources of the process to which they belong
  - All threads run within same address space
- Economy
  - They can communicate through shared data and eliminate the overhead of system calls
- Utilization of multiprocessor architectures
  - They allow you to get parallel performance on a shared memory multiprocessor.

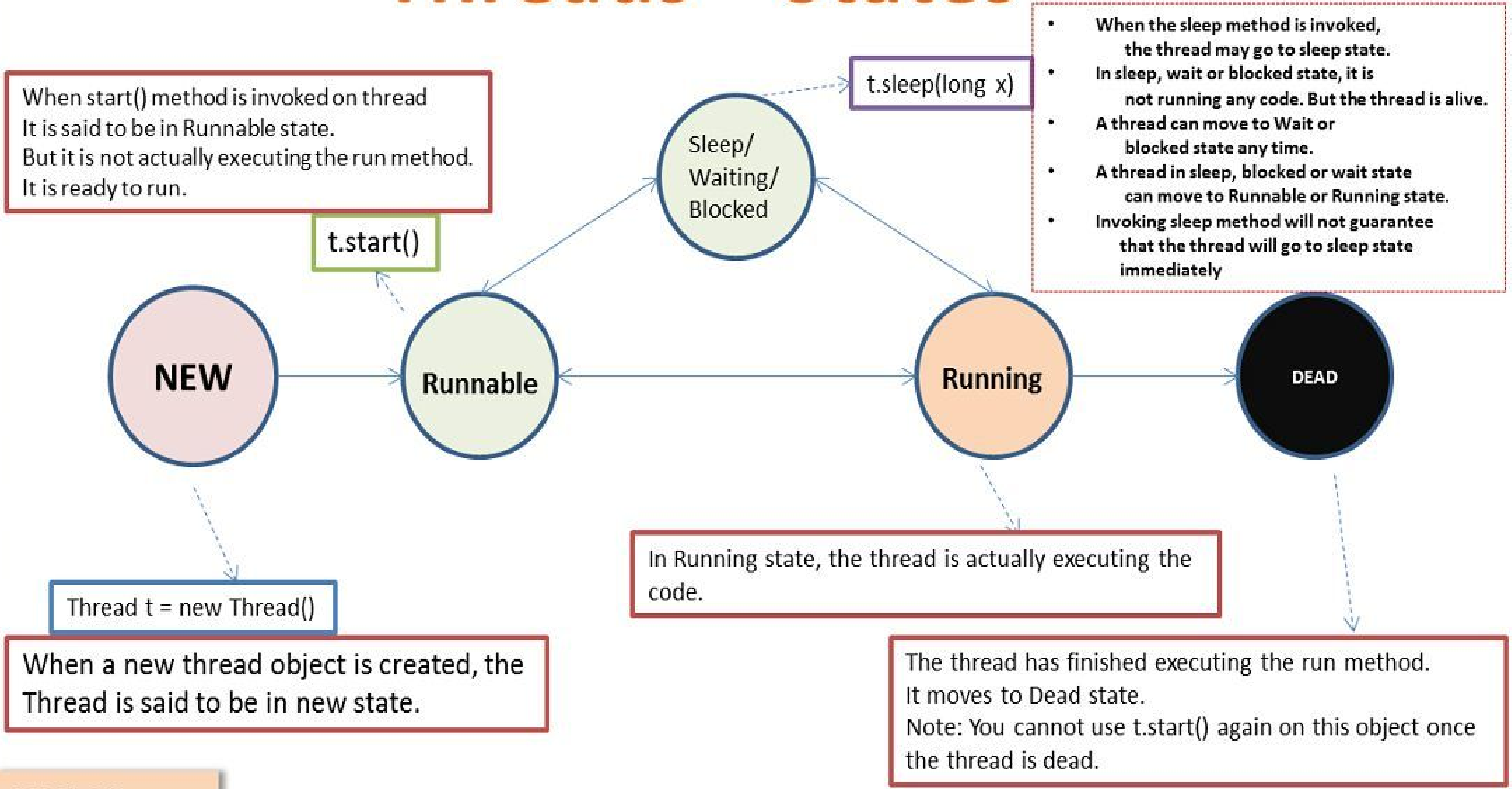
Java Threads

- **Threads** are the fundamental model of program execution in a Java program.
- Java language and its APIs provide a rich set of features for the creation and management of threads
- Various approaches of creating threads**:**
  - By extending **Thread** class and overriding **run()** method
  - Using **Runnable** interface
  - Using **ExecutorService**

Approach 1 (extending Thread class)

```java
package main.demo;

class MyThread extends Thread {

    public void run() {
        System.out.println("Inside my thread");
    }
}

public class ThreadDemo {

    public static void main(String[] args) {
        MyThread myThread = new MyThread();
        myThread.start();
    }
}
```

Threads – States



Approach 1 continued ...

- Creating a **Thread** object does not specifically create the new thread.
- It is the **start()** method that actually creates the new thread.
- Calling the **start()** method for the new object does two things:
  - It allocates memory and initializes a new thread in the JVM.
  - It calls the **run()** method, making the thread eligible to be run by the JVM.
    - We never call the **run()** method directly. Rather, we call the **start()** method, and it calls the **run()** method on our behalf.

Approach 2 (Using Runnable interface) ...

```
package main.demo;

class MyRunnable implements Runnable {

    @Override
    public void run() {
        for (int i = 0; i < 6; i++) {
            System.out.println("The counter value is " + i);
        }
    }
}

public class ThreadRunnableDemo {

    public static void main(String[] args) {
        MyRunnable r = new MyRunnable();
        Thread foo = new Thread(r);
        Thread bar = new Thread(r);
        foo.start();
        bar.start();
    }
}
```

Aprroach 3

Using **ExecutorService**.

Discussed later in this presentation with example code.

Thread Safety

- Writing thread safe programs, at its core, is all about managing access to ***state*** and in particular to ***shared*** , ***mutable*** state.
- An object's ***state*** is its ***data***, stored in state variables *viz.* instance/static variables
  - Shared → Variable is accessed by multiple threads at a time.
  - Mutable → State of variable can change over period of its lifetime
- We may talk about Thread Safety as if it were about **code,** but it's all about protecting **data** from uncontrolled concurrent access.

- Whenever more than one thread accesses a given state variable, and one of them might **write** to it, they all **must** coordinate their access to it using ***synchronization***.
- The primary mechanism for ***synchronization*** in Java is the *synchronized* keyword, which provides *exclusive locking* and *memory visibility*, but the term “synchronization” also ***includes the use of volatile variables, explicit locks, and atomic variables.***

Synchronization and Locks

- Java provides two types of synchronization:
    - Method level synchronization
    - Block level synchronization

- **synchronized** methods prevent more than one thread from accessing an object's critical method code *simultaneously*.

- To synchronize a **block** of code, we must specify an argument that is the object whose lock you want to synchronize on.

- While only *one* thread can be accessing **synchronized** code of a particular instance, multiple threads can still access the same object's ***unsynchronized*** code.

- When a thread goes to **sleep**, its locks will be *unavailable* to other threads.
- *static* methods can be synchronized, using the lock from the *java.lang.Class* instance representing that class.

Defining Thread Safety

- A class is *thread-safe* when it continues to behave correctly when accessed from multiple threads.
    - Correctness means that a class conforms to its specification.
    - A good specification defines **invariants** constraining an object's state and **postconditions** describing the effects of its operations.
- A class is ***thread-safe*** if it behaves *correctly* when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and ***with no additional synchronization or other coordination on the part of the calling code***.

Synchronizing Code

- Account withdrawal example (IDE)

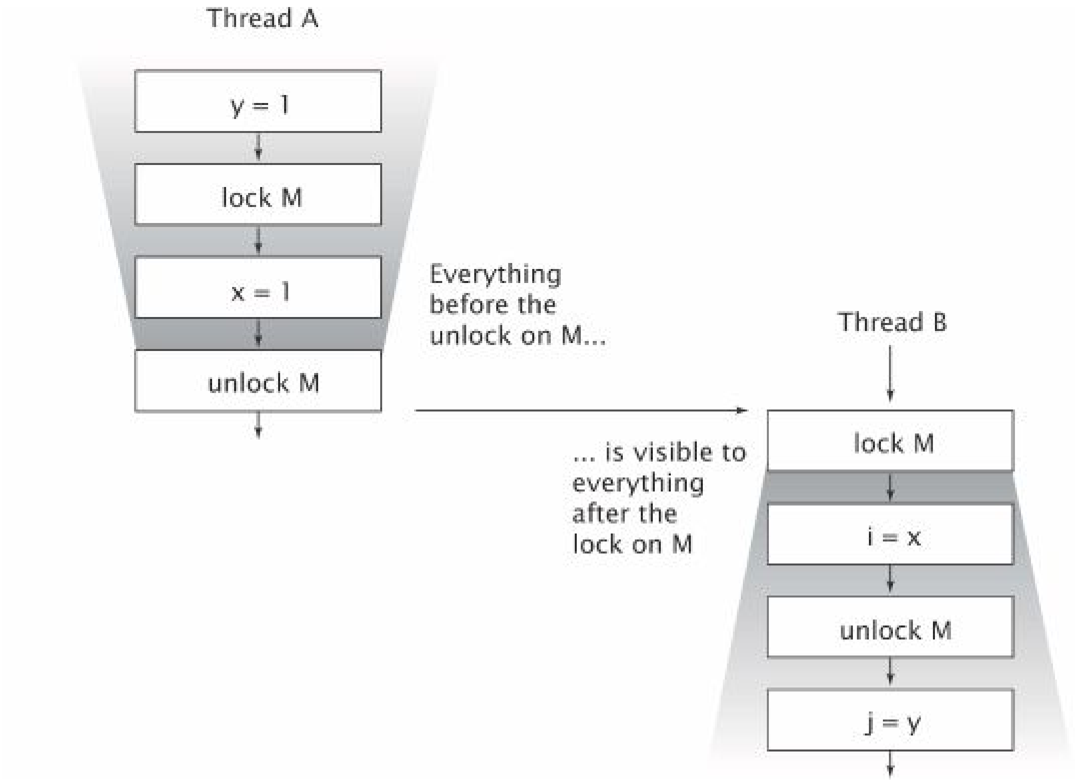
Sharing Objects

```
public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}
```

Some possibilities ...

- NoVisibility could loop forever because the value of ready might never become visible to the *Reader* thread.
- NoVisibility could print **zero** because the write to ready might be made visible to the *Reader* thread before the write to number, a phenomenon known as ***reordering***.
- When the *Main* thread writes first to number and then to ready ***without synchronization***, the *Reader* thread could see those writes happen **in the opposite order** -- **or not at all**.

- In the absence of **synchronization**, the compiler, processor, and runtime can do some downright weird things to the order in which operations appear to execute.
- Compilers are allowed to *reorder* the instructions in either thread, when this **does not** affect the execution of that thread in *isolation*.
- Attempts to reason about the order in which memory actions “must” happen in insufficiently synchronized multithreaded programs will almost certainly be incorrect.
- **Always use the proper synchronization whenever data is shared across threads.**

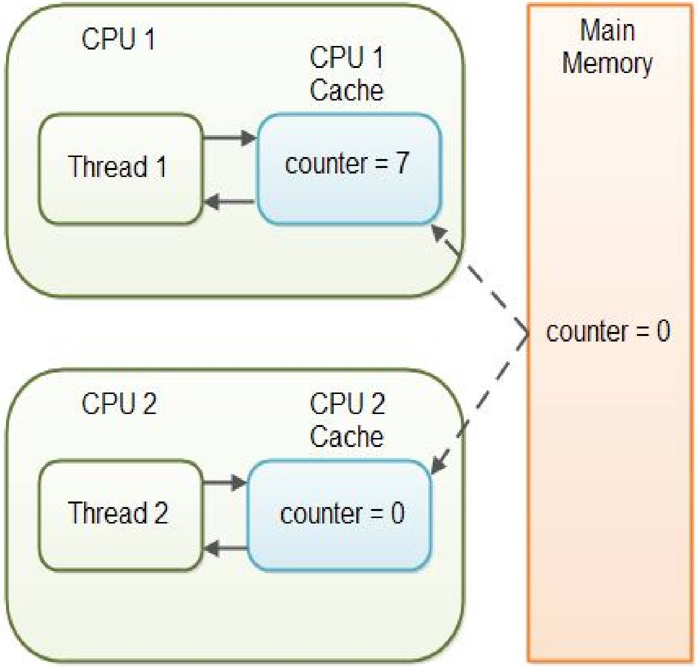


- Intrinsic locking can be used to **guarantee** that one thread sees the effects of another in a ***predictable*** manner

- When thread A executes a synchronized block, and subsequently thread B enters a synchronized block ***guarded by the same lock***, the values of variables that were visible to A prior to releasing the lock are guaranteed to be visible to B upon acquiring the lock.
- In other words, everything A did in or prior to a synchronized block is visible to B when it executes a synchronized block ***guarded by the same lock.***
- ***Without synchronization, there is no such guarantee.***
- IDE mode (more on @GuardedBy)

Volatile variables

- When a field is declared ***volatile***, the compiler and runtime are put on notice that this variable is shared and *that operations on it should not be reordered with other memory operations.*
- Volatile variables are *not cached* in registers or in caches where they are hidden from other processors.
- Read of a volatile variable always returns the most recent write by any thread.




```
Thread A:
    sharedObject.nonVolatile = 123;
    sharedObject.counter     = sharedObject.counter + 1;

Thread B:
    int counter     = sharedObject.counter;
    int nonVolatile = sharedObject.nonVolatile;
```

- Example code (IDE mode). Counter Class

```
class VolatileExample {
  int x = 0;
  volatile boolean v = false;
  public void writer() {
    x = 42;
    v = true;
  }

  public void reader() {
    if (v == true) {
      ??
    }
  }
}
```

Final fields in Java

- Declaring the variable that is initialized to the object as *final* prevents the object from being *partially initialised.*

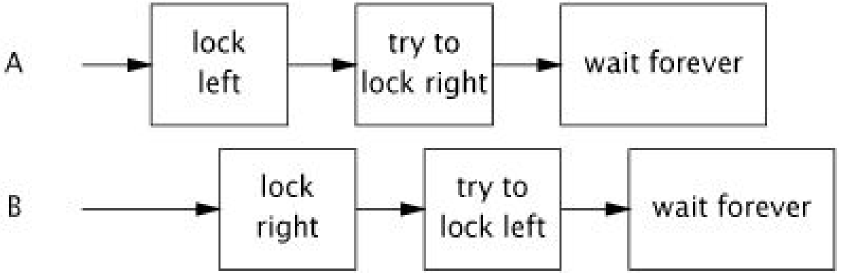
- An object is considered to be **completely** initialized when its constructor finishes.
  - A thread that can only see a *reference* to an object after that object has been completely initialized is **guaranteed** to see the *correctly initialized* values for that object's *final* fields.

- Variables that are declared *static* and initialized at declaration OR from a static initializer are **guaranteed** to be **fully constructed** before being made visible to other threads. However, this solution forgoes the benefits of lazy initialization.

```
class FinalFieldExample {

  final int x;
  int y;
  static FinalFieldExample f;
  public FinalFieldExample() {
    x = 3;
    y = 4;
  }

  static void writer() {
    f = new FinalFieldExample();
  }

  static void reader() {
    if (f != null) {
      int i = f.x;
      int j = f.y;
    }
  }
}
```

Deadlock Example 1

```
// Warning: deadlock-prone!
public class LeftRightDeadlock {
    private final Object left = new Object();
    private final Object right = new Object();

    public void leftRight() {
        synchronized (left) {
            synchronized (right) {
                doSomething();
            }
        }
    }

    public void rightLeft() {
        synchronized (right) {
            synchronized (left) {
                doSomethingElse();
            }
        }
    }
}
```



Deadlock Example 2

```
// Warning: deadlock-prone!
public void transferMoney(Account fromAccount,
                          Account toAccount,
                          DollarAmount amount)
        throws InsufficientFundsException {
    synchronized (fromAccount) {
        synchronized (toAccount) {
            if (fromAccount.getBalance().compareTo(amount) < 0)
                throw new InsufficientFundsException();
            else {
                fromAccount.debit(amount);
                toAccount.credit(amount);
            }
        }
    }
}
```

```
A: transferMoney(myAccount, yourAccount, 10);
B: transferMoney(yourAccount, myAccount, 20);
```

How to avoid deadlock ?

Impose ordering on lock acquisition (see next slide)

```java
public class MoneyTransfer {

    private static final Object tieLock = new Object();

    public void transferMoney(final Account fromAcct, final Account toAcct, double amount) {

        class Helper {
            private void transfer() throws InsufficientFundsException {
                if (fromAcct.getBalance().compareTo(amount) < 0) {
                    throw new InsufficientFundsException("Insufficient funds to perform transfer");
                } else {
                    fromAcct.debitAmount(amount);
                    toAcct.creditAmount(amount);
                }
            }
        }

        int fromHash = System.identityHashCode(fromAcct);
        int toHash = System.identityHashCode(toAcct);
        if (fromHash < toHash) {
            synchronized (fromAcct) {
                synchronized (toAcct) {
                    new Helper().transfer();
                }
            }
        } else if (toHash < fromHash) {
            synchronized (toAcct) {
                synchronized (fromAcct) {
                    new Helper().transfer();
                }
            }
        } else {
            synchronized (tieLock) {
                synchronized (fromAcct) {
                    synchronized (toAcct) {
                        new Helper().transfer();
                    }
                }
            }
        }
    }
```

Limitations of intrinsic locking

- 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.
  - this simplifies coding and interacts nicely with exception handling, but makes *non-block structured* locking disciplines impossible.

Solution: Explicit Locks

```
public interface Lock {
    void lock();
    void lockInterruptibly() throws InterruptedException;
    boolean tryLock();
    boolean tryLock(long timeout, TimeUnit unit)
        throws InterruptedException;
    void unlock();
    Condition newCondition();
}
```

ReentrantLock implements **Lock** interface providing same **mutual exclusion** and **memory-visibility** guarantees as **synchronized**.

Example Code (IDE mode)

Template for using Lock

```
Lock lock = new ReentrantLock();
...
lock.lock();
try {
    // update object state
    // catch exceptions and restore invariants if necessary
} finally {
    lock.unlock();
}
```

Example (IDE mode) (AccountReentrantLock)

Another example of Lock ...

```
public boolean trySendOnSharedLine(String message,
                                   long timeout, TimeUnit unit)
                                   throws InterruptedException {
    long nanosToLock = unit.toNanos(timeout)
                     - estimatedNanosToSend(message);
    if (!lock.tryLock(nanosToLock, NANOSECONDS))
        return false;
    try {
        return sendOnSharedLine(message);
    } finally {
        lock.unlock();
    }
}
```

```java
public boolean transferMoney(Account fromAcct,
                             Account toAcct,
                             DollarAmount amount,
                             long timeout,
                             TimeUnit unit)
        throws InsufficientFundsException, InterruptedException {
    long fixedDelay = getFixedDelayComponentNanos(timeout, unit);
    long randMod = getRandomDelayModulusNanos(timeout, unit);
    long stopTime = System.nanoTime() + unit.toNanos(timeout);

    while (true) {
        if (fromAcct.lock.tryLock()) {
            try {
                if (toAcct.lock.tryLock()) {
                    try {
                        if (fromAcct.getBalance().compareTo(amount)
                                < 0)
                            throw new InsufficientFundsException();
                        else {
                            fromAcct.debit(amount);
                            toAcct.credit(amount);
                            return true;
                        }
                    } finally {
                        toAcct.lock.unlock();
                    }
                }
            } finally {
                fromAcct.lock.unlock();
            }
        }
        if (System.nanoTime() > stopTime)
            return false;
        NANOSECONDS.sleep(fixedDelay + rnd.nextLong() % randMod);
    }
}
```

Code example

ExecutorService + Future + Callable (IDE mode)

Java Memory Model (JMM)

- Java Memory Model (JMM) describes what behaviors are legal in multithreaded code, and how threads may interact through memory (RAM).
- JMM is specified in terms of **actions**, which include *reads* and writes to variables, *locks* and *unlocks* of monitors and *starting* and *joining* with threads.
- JMM defines the behavior of *volatile* and *synchronized*, and, more importantly, ensures that a correctly synchronized Java program runs **correctly** on **all** processor architectures.

Happens Before relationship

- The JMM defines a partial ordering called ***happens before*** on all actions within the program.
- To guarantee that the thread executing action B can see the results of action A (whether or not A and B occur in different threads), there **must be** a ***happens before*** relationship between A and B.
    - In the absence of a *happens-before* ordering between two operations, the JVM is free to *reorder* them as it pleases.
- It should be noted that the presence of a *happens before* relationship between two actions does not necessarily imply that they have to take place in that order in an implementation.
    - If the reordering produces results consistent with a legal execution, it is not illegal

Examples of *Happens Before* relationship

- **Monitor lock rule**. An *unlock* on a monitor lock ***happens before*** every subsequent *lock* on that same monitor lock.
- **Volatile variable rule**. A *write* to a volatile field ***happens before*** every subsequent *read* of that same field.
- **Thread start rule**. A call to *Thread.start()* on a thread ***happens before*** every action in the started thread.
- **Thread termination rule**. Any action in a thread ***happens before*** any other thread detects that thread has terminated, either by successfully return from *Thread.join()* or by *Thread.isAlive()* returning false.

- **Interruption rule**. A thread calling interrupt on another thread ***happens before*** the interrupted thread detects the interrupt (either by having *InterruptedException* thrown, or invoking *isInterrupted* or *interrupted*).
- **Finalizer rule**. The end of a constructor for an object ***happens before*** the start of the finalizer for that object.
- **Transitivity**. If A ***happens before*** B, and B ***happens before*** C, then A ***happens before*** C.

Some topics for future brown bag sessions…

- Concurrent data structures
  - ConcurrentHashMap, Blocking queues, AtomicX classes
  - Synchronizers *viz.* semaphores, countDownLatch, cyclic barriers, exchanger, phaser.
- ThreadPools. Executors *vs.* ExecutorService. Various implementations.
- Different Lock variations
  - Reentrant locks, ReentrantReadWrite locks, Stamped locks (introduced in Java 8)
- Thread interruption mechanisms
- Non-blocking synchronization algorithms
  - Use of CAS (CompareAndSwap) hardware instruction

References

Java Concurrency in Practice by *Joshua Bloch*, *Doug Lea et.al.*

Operating System Principles, Seventh Edition by *Galvin, Gagne et.al*

https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=18581044

Java Language Specification *https://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html*

http://www.cs.umd.edu/~pugh/java/memoryModel/jsr-133-faq.html

https://stackoverflow.com/questions/16213443/instruction-reordering-happens-before-relationship-in-java

http://tutorials.jenkov.com/java-concurrency/volatile.html

Questions ?