



M9(b)-Concurrency | Jin L.C. Guo

Image source: https://cdn.pixabay.com/photo/2016/10/10/00/48/chefs-1727446_960_720.jpg

Objective

- Understand the concept of a Thread and its usefulness for programming;
- Be able to write basic concurrent programs in Java;
- Understand the causes of basic concurrency errors
- Understand the mechanisms that help prevent the basic concurrency errors.

Risks of threads

- Safety Hazard
 - System behave incorrectly
- Liveness Hazard
 - System fails to make forward progress (deadlock, starvation, livelock)
- Performance Hazard
 - Impair service time, responsiveness, throughput, resource consumption, or scalability of the system.

Philosopher dining problem

- The philosophers alternate between thinking and eating
- Each needs to acquire two chopsticks for long enough to eat
- They can put the chopsticks back and return to thinking.

Side note: it is invented by E. W. Dijkstra for a student exam.

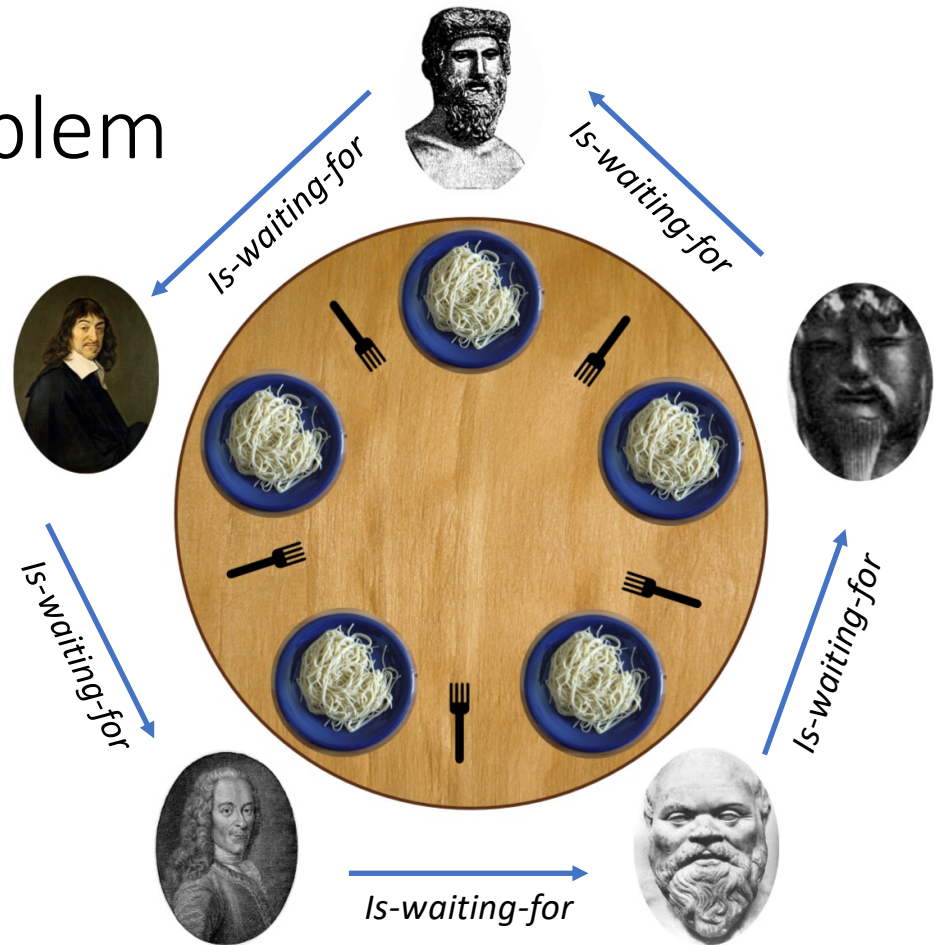


Image Source: https://upload.wikimedia.org/wikipedia/commons/7/7b/An_illustration_of_the_dining_philosophers_problem.png

```

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) {
                doSomething();
            }
        }
    }
}

```

Thread A

Lock left

Try to lock
right

Wait
forever

Thread B

Lock right

Try to
lock left

Wait
forever

```

static class Friend {
    private final String name;
    public Friend(String name) {
        this.name = name;
    }
    public String getName() {
        return this.name;
    }
    public synchronized void bow(Friend bower) {
        System.out.format("%s: %s"
            + " has bowed to me!\n",
            this.name, bower.getName());
        bower.bowBack(this);
    }
    public synchronized void bowBack(Friend bower) {
        System.out.format("%s: %s"
            + " has bowed back to me!\n",
            this.name, bower.getName());
    }
}

```

<https://docs.oracle.com/javase/tutorial/essential/concurrency/deadlock.html>

DeadLock Demo

Deadlock

- A class has a potential deadlock doesn't mean that it ever *will* deadlock, just that it can.

Improvement

- Avoid multiple locking
 - Make your program that never acquires more than one lock at a time.
- Locker-ordering
 - Minimize the number of potential locking interactions, and follow and document a lock-ordering protocol for locks that may be acquired together.
- Documentation
 - Lock-ordering assumptions
 - When a method must acquire a lock to perform its function or must be called with a specific lock held

Java Lock Interface

A more flexible locking mechanism offers better liveness or performance.

```
Lock lock = ...;  
lock.lock();  
try {  
    // access the resource protected by this lock  
} finally {  
    lock.unlock();  
}
```

If the lock is not available then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired.

```
Lock lock = ...;  
lock.lock();  
try {  
    // access the resource protected by this lock  
} finally {  
    lock.unlock();  
}
```

VS

```
synchronized (object) {  
    // access or modify shared state guarded by lock  
}
```

tryLock method

Acquires the lock if it is available and returns immediately with the value true.

```
Lock lock = ...;  
if (lock.tryLock()) {  
    try {  
        // manipulate protected state  
    } finally {  
        lock.unlock();  
    }  
} else {  
    // perform alternative actions  
}
```

If the lock is not available then this method will return immediately with the value false.

DeadLock Fixed Demo

Coordinate threads with wait and notifyAll

```
public void prepare() {  
    while(!ready)  
    {  
        // Do stuff  
    }  
    System.out.println("I am ready!");  
}
```

Executes continuously while waiting

Coordinate threads with wait and notifyAll

```
public synchronized void prepare() {  
    while(!ready) { // must get lock before entering here  
        try {  
            wait(); // releases lock here  
            // must regain the lock to reentering here  
        }  
        catch (InterruptedException e){ }  
    }  
    System.out.println("I am ready!");  
}
```

Coordinate threads with wait and notifyAll

```
public synchronized void setReady() {  
    ready = true;  
    notifyAll();  
}
```

informing all threads waiting on that
lock that something important has
happened:

WaitToBeReady Demo

WaitToBeReady Demo with Lock and Condition

Using Lock object with Condition

```
Condition condition = lock.newCondition();
```

```
object.wait();           ➡ condition.await();
```

```
object.notifyAll();      ➡ condition.signalAll();
```

```
class BoundedBuffer {  
    final Lock lock = new ReentrantLock();  
    final Condition notFull = lock.newCondition();  
    final Condition notEmpty = lock.newCondition();  
    final Object[] items = new Object[100];  
    int putptr, takeptr, count;  
  
    public void put(Object x) throws InterruptedException {  
        lock.lock();  
        try {  
            while (count == items.length)  
                notFull.await();  
            items[putptr] = x;  
            if (++putptr == items.length) putptr = 0;  
            ++count;  
            notEmpty.signal();  
        } finally {  
            lock.unlock();  
        }  
    }  
}
```

```
public Object take() throws InterruptedException {  
    lock.lock();  
    try {  
        while (count == 0)  
            notEmpty.await();  
        Object x = items[takeptr];  
        if (++takeptr == items.length) takeptr = 0;  
        --count;  
        notFull.signal();  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```

Java Synchronized Collection Classes



Java Collections

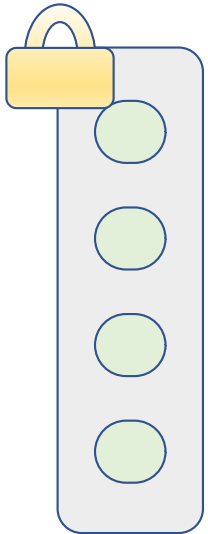
Encapsulating collection state and synchronizing every public method so that only one thread at a time can access the collection state.

```
List<String> strings = new ArrayList<>();
```

```
List<String> wrappedList =  
    Collections.synchronizedList(strings);
```

Not enough for common compound actions on **collections**, such as iteration.

Java Synchronized Collection Classes



Java Collections

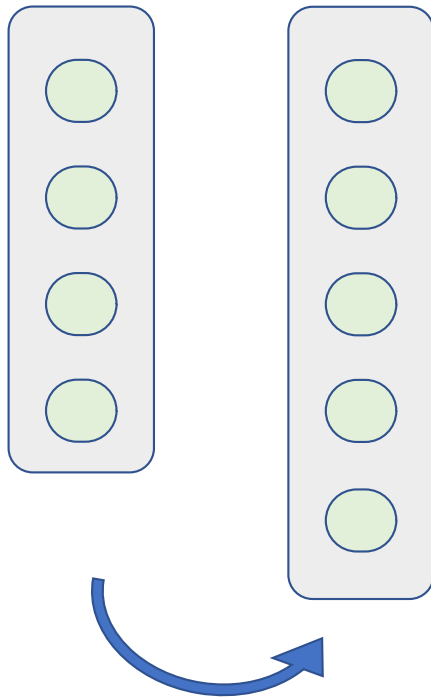
Encapsulating collection state and synchronizing every public method so that only one thread at a time can access the collection state.

```
List<String> strings = new ArrayList<>();
```

```
List<String> wrappedList =  
    Collections.synchronizedList(strings);
```

```
synchronized (wrappedList) {  
    Iterator i = wrappedList.iterator(); //  
    Must be in synchronized block  
    while (i.hasNext())  
        foo(i.next());  
}
```

Java Concurrent Collection



Classes CopyOnWriteArrayList

```
concurrentList.add(new Object());
```


Iterating List Demo

Recap

- Understand the concept of a Thread and its usefulness for programming;
- Be able to write basic concurrent programs in Java;
- Understand the causes of basic concurrency errors
- Understand the mechanisms that help prevent the basic concurrency errors.