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.
- Be able to recognize when to and when not to use concurrency in application design;

Thread Safety

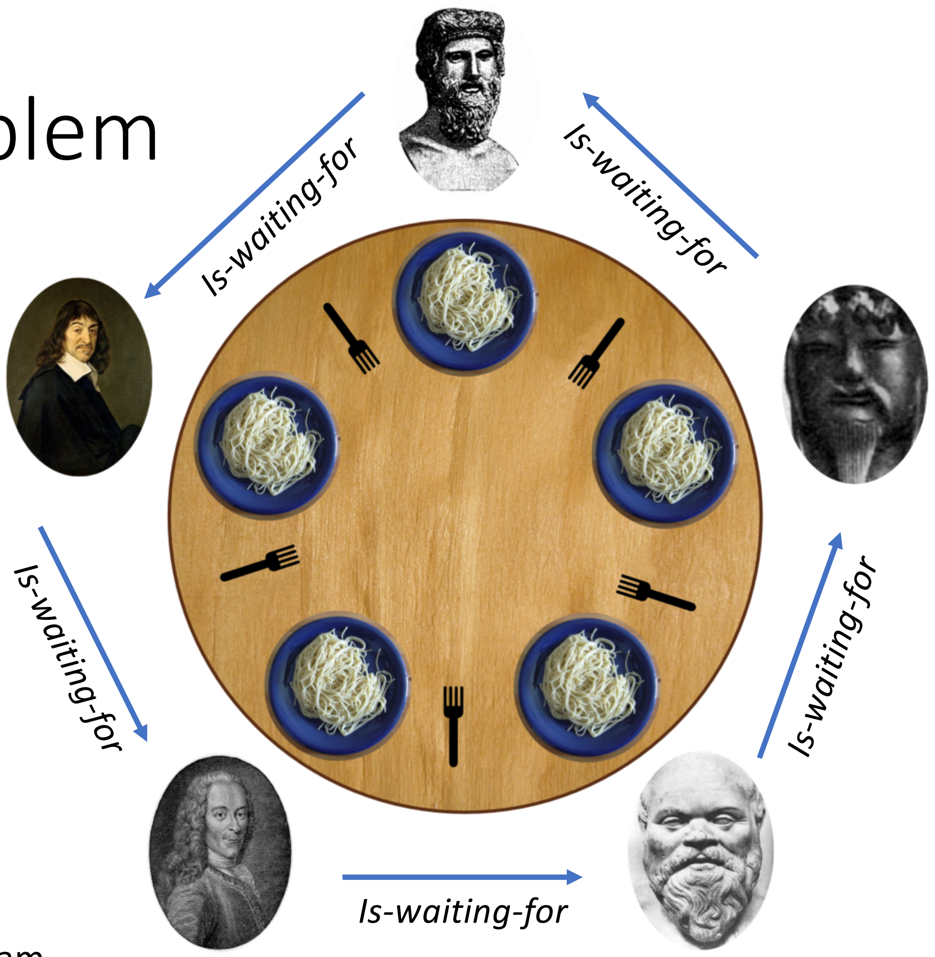
- A class is *thread-safe* if it behaves correctly when accessed from multiple threads, regardless of the scheduling or interleaving of the execution of those threads by the runtime environment, and with no additional synchronization or other coordination on the part of the calling code.

Risks of threads

- Safety Hazard
  - System behave incorrectly
- Liveness Hazard
  - System fails to make forward progress (deadlock, starvation, lovelock)
- 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.



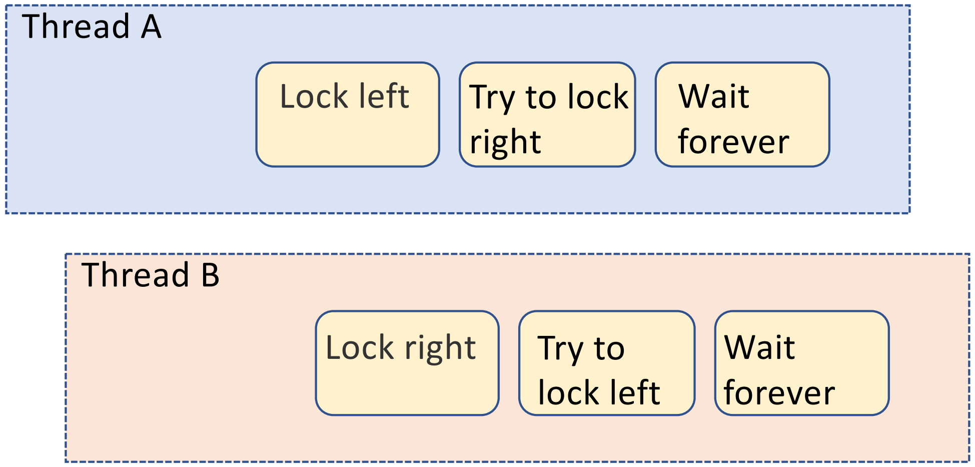
Side note: it is invented by E. W. Dijkstra for a student exam.

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();
         }
      }
   }
}
```



```java
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());
    }
}
```

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

Other Liveness Hazard

- Starvation

  When a thread is perpetually denied access to resources (such as CPU time) it needs in order to make progress

- Livelock

  when a thread, while not blocked, still cannot make progress because it keeps retrying an operation that will always fail.

Java Lock Interface

A more flexible locking mechanism offers better liveness or performance.

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.

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

```
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

Using Lock object with Condition

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

$$\texttt{object.wait();} \longrightarrow \texttt{condition.await();}$$

$$\texttt{object.notifyAll();} \longrightarrow \texttt{condition.signalAll();}$$

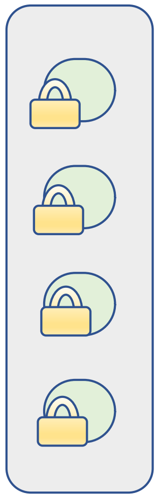
WaitToBeReady Demo with Lock and Condition

```java
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();
    }
  }
```

```java
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

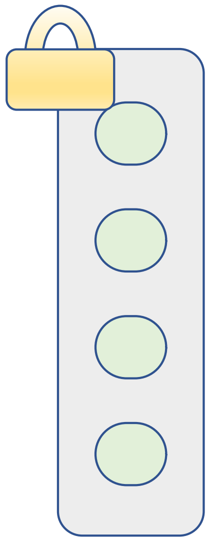
Encapsulating collection state and synchronizing every public method so that only one thread at a time can access the collection state.

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

List<String> wrappredList =
   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.

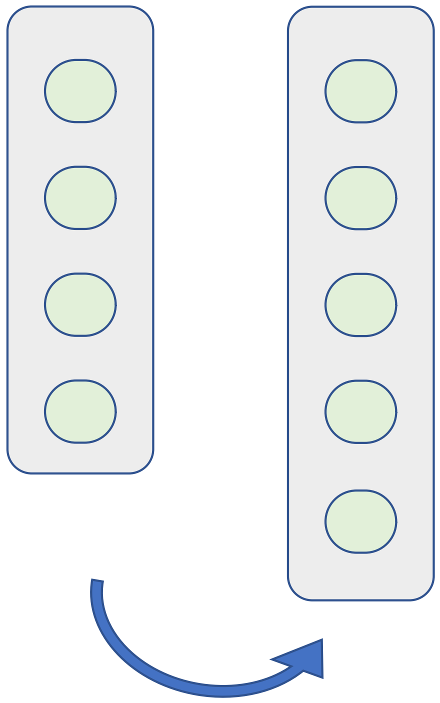
```
List<String> strings = new ArrayList<>();

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

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

Iterating List Demo

Java Concurrent Collection



Classes CopyOnWriteArrayList

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

CopyOnWriteArrayList Demo