



SWEN 601 Software Construction

Threads & Concurrency





Activity: Getting Started

1. Begin by accepting the GitHub Classroom invitation for today's homework.
 - a. *The project may already contain some code!*
2. Create a session package. This is where you will write your solutions to today's activities.
3. Create a homework package. This is where you will implement your solution to the homework.

When you submit your homework, you will include your activities. You may earn up to a 10% bonus on the homework if you have completed all of the activities.

Do not submit code that **does not compile**. Comment it out if necessary.

Next Two Weeks

WEEK 14	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #20				
LECTURE			Thread Cooperation		No Class (Thanksgiving)		
HOMEWORK	Hwk 20 Due (<u>11:30pm</u>)		Hwk 21 Assigned				

WEEK 15	SUN	MON	TUES	WEDS	THURS	FRI	SAT
QUIZ			Quiz #21		Quiz #22		
LECTURE			Networking		Final Exam Review		
HOMEWORK	Hwk 18 Due (<u>11:30PM</u>)		Hwk 22 Assigned			Hwk 22 Due (<u>11:30pm</u>)	

Shared Resources

- When two or more threads both use the same data (e.g. an instance of a class), the data is referred to as a **shared resource**.
- Consider the code to the right, which is an excerpt from a simple, array-based implementation of the List ADT.
- If two or more threads are using the same instance of the `MyArrayList` class at exactly the same time, what potential problems might this cause?

```
// from MyArrayList.java

@Override
public void add(E element) {
    if(size == elements.length) {
        elements = Arrays.copyOf(elements, size * 2);
    }
    elements[size] = element;
    size++;
}

@Override
public E get(int index) {
    return (E)elements[index];
}

@Override
public int size() {
    return size;
}
```

Activity: Threading an Array I

You will write a program that creates 10 threads that all attempt to add 1000 values to the same list. When the program is finished, the list *should* have 10,000 values in it. Right?



What happened? Did all of the values make it into each list?

Does using a `MyNodeList` instead make a difference?

1. Begin by creating a new class, `ListAdder`.
 - a. Make this class a thread by extending `Thread` or implementing `Runnable`.
 - b. It should have three fields:
 - i. `MyList<Integer> list` - the list.
 - ii. `int start` - the number at which to start.
 - iii. `int stop` - the number at which to stop.
 - c. The `run()` method should add the integers between `start` and `stop` to the list.
2. Write a main method that:
 - a. Creates a `MyArrayList` and 10 of your `ListAdders`, each of which adds 1000 values to the list. Wait for the threads to finish (hint: `join()`) and print the size of the list.
 - b. Creates a `MyNodeList` and does the same.

Activity: Threading an Array II



What happened? Did Java's list implementations work better?

That little experiment showed that the `MyList` implementations have problems. But surely Java's lists behave better, right?

1. Write a class called `JavaListAdder`.
 - a. Copy and modify your `ListAdder` code to work with Java lists (`java.util.List`).
2. Modify the main method to
 - a. Create 10 of your `JavaListAdders` with a `java.util.ArrayList`.
 - b. Create 10 more with a `java.util.LinkedList`.

Resource Contention

- **Resource contention** occurs when two or more threads attempt to **manipulate** the value of a **shared resource** at the same time.
 - The two threads come into **conflict** with each other.
- One thread may make changes that the other thread **duplicates** or **ignores**.
 - `Arrays.copyOf(elements, size*2)` - both threads **copy the array**.
 - `elements[size] = value` - both threads **add a value to the same index**.
 - `size++` - both threads **increment size**.
- In each case, which thread **“wins?”**
 - It is **unpredictable**, and sometimes nothing goes wrong.
- This is called a **race condition**.



Code that prevents race conditions from happening is **thread safe**.

Thankfully, Java provides mechanisms for making code **thread safe**.

Critical Regions

- Any block of code that may not *behave* correctly if executed by *two or more threads* at the same time is referred to as a *critical region*.
- A critical region must be *protected* so that two threads *cannot* access the code in the region at the *same time*.
- Sometimes the critical region comprises *an entire method*.
- Other times it is only a *subset* of code in a method.



Much (indeed most) of the code that you write need not be concerned with thread safety because it will never be executed by more than one thread at a time.

But code that will be executed by multiple threads needs to be made *thread safe*.

The first step in making code *thread safe* is identifying the *critical regions* in the code.



Activity: Critical Regions

Given the code examples below, identify the critical regions that need to be protected from being accessed by multiple threads at once.

```
// from MyArrayList.java
```

```
@Override
public void add(E element) {
    if(size == elements.length) {
        elements = Arrays.copyOf(elements, size * 2);
    }
    elements[size] = element;
    size++;
}

@Override
public E get(int index) {
    return (E)elements[index];
}

@Override
public int size() {
    return size;
}
```

```
public class Racer implements Runnable {
    private final List<Integer> numbers;

    public Racer(List<Integer> numbers) {
        this.numbers = numbers;
    }

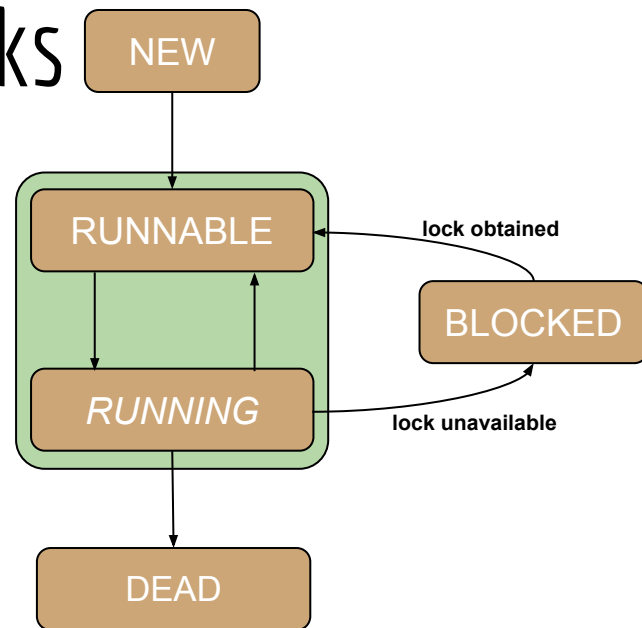
    public void run() {
        for(int i=0; i<1000; i++) {
            numbers.add(i);
        }
    }

    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        for(int n=0; n<10; n++) {
            Racer racer = new Racer(numbers);
            Thread thread = new Thread(racer);
            thread.start();
        }
    }
}
```

Thread Safety: Synchronizing Blocks

- The **synchronized** keyword is used to obtain a **mutually exclusive lock** on an object.
- The **lock** is used to define a **critical region** in curly braces ({}).
- A thread must **obtain the lock** before it can **enter** the critical region.
- If a second thread tries to **obtain the same lock**, it will transition to a new state: **BLOCKED**.
- When the first thread **exits** the critical region, it will **give up the lock**, and the thread scheduler will choose one of the **BLOCKED** threads to transition to **RUNNABLE**.

```
synchronized(someSharedObject) {  
    // critical region  
    // only one thread allowed per unique lock  
}
```



In order for a lock to work, all of the threads involved must share the **same exact lock** (e.g. instance of some object).


It is therefore common for the **shared resource** to be used as the lock.

Activity: Synchronizing Blocks I

Let's create a synchronized block of code that makes sure that no two threads try to add values to the same list at the same time. First, take a look at the relevant code, and answer the following questions:

```
@Override
public void run() {
    for(int i=start; i<=stop; i++) {
        list.add(i);
    }
}
```

1. What should be used as the lock that all of the threads share (it can be any object)?
2. Which line(s) of code need to be protected?



The thread scheduler acts like a **bouncer** that only lets one thread in *per lock* access the critical region.

Activity: Synchronizing Blocks II



1. Create a new class, `SafeListAdder`.
2. Copy the code from your `ListAdder` class into it.
3. Synchronize the appropriate lines of code.
 - a. What will you use for the **lock**? Remember, it has to be an **object** (not a **class**) that all of the threads share in common.
 - b. Synchronized code runs **up to 50% slower** than code that is not synchronized. Keeping this in mind, which lines of code will you define as a critical region?
4. Update your `ThreadedAdder` to use the `SafeListAdder`. What happens?

In order for the thread scheduler to protect a critical region, all of the threads need to use **exactly the same lock**.

Thread Safety: Synchronizing Methods

- Sometimes, the **entire body** of the method is a critical region.
- In these cases, the **entire method** can be synchronized by including the **synchronized** keyword in the method declaration.
- This is the equivalent of enclosing the **entire body** of the method in a **synchronized(this) {...}** block.
 - The **object itself** is the **lock**.
- Two threads will not be able to call **synchronized** methods on the same lock at the same time.

```
public void aMethod() {  
    synchronized(this) {  
        // the whole body is a  
        // critical region  
    }  
}
```

Is the equivalent of...

```
public synchronized void aMethod() {  
  
    // the whole body is a  
    // critical region  
  
}
```



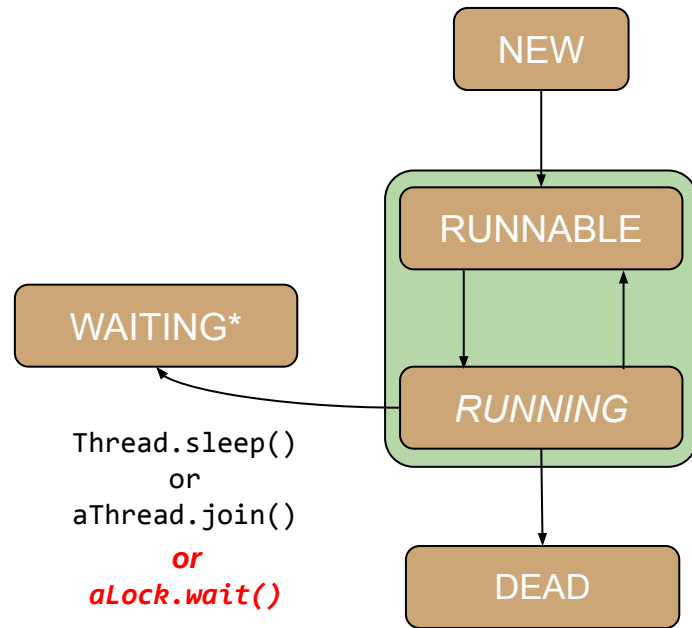
Activity: Synchronizing Methods

1. Create a new class, `SafeArrayList`.
 - a. Copy the code from `MyArrayList` into the `SafeArrayList`.
 - b. Synchronize the `add`, `get`, and `size` methods.
2. Update your `ThreadedAdder` to use a `ListAdder` (**not** `SafeListAdder`) and a `SafeArrayList`.
3. What happens?

So why not just make every data structure thread safe all the time?

Waiting

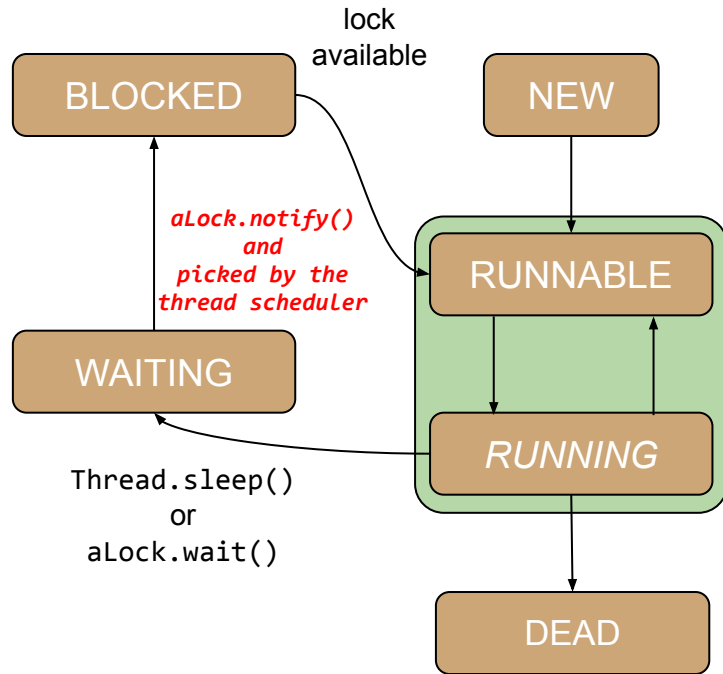
- Sometimes a thread can't do the work it needs to do until some **important event occurs**
 - e.g. an element is **added to a queue** and the thread needs to **use it** in some way.
- In these cases the thread must **wait** for the event.
 - The thread must first **obtain a lock** using **synchronized**, e.g. `synchronized(aLock)`.
 - It must then call the `wait()` method on the lock, e.g. `aLock.wait()`.
 - Calling `wait()` on a lock **releases the lock!**
- A thread may **wait indefinitely** or for some **duration** specified in milliseconds.
- A thread that calls the `wait()` method is moved into the **WAITING** state.
 - In this way it is similar to a thread that is **sleeping** or **joining**.



A thread **must** own the lock before it tries to wait. Trying to wait on a lock before obtaining it causes an **IllegalMonitorStateException**.

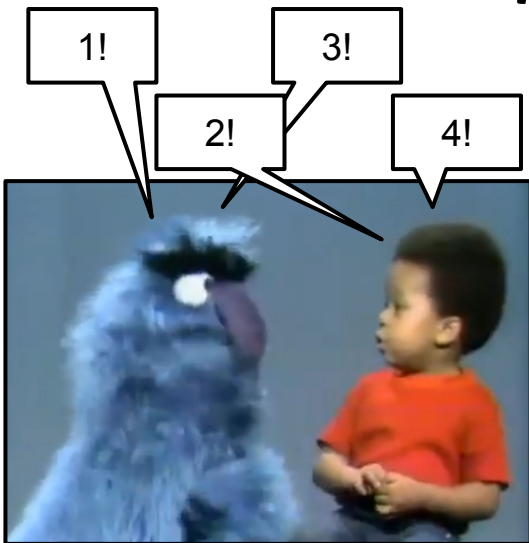
Notify

- A thread that **owns** a lock may use it to **notify** threads that are waiting on the **same lock** by calling the `notify()` method on the lock, e.g. `aLock.notify()`
 - Calling `aLock.notify()` **does not** release the lock!
 - The calling thread will continue to **hold the lock** until it **exits** the **synchronized** block or calls `wait()` on the lock.
- The thread scheduler will choose **one** of the waiting threads and move it to the **BLOCKED** state.
 - All other threads in the **WAITING** state will **remain there** even if the lock is **released**.
 - If **no threads** are waiting on the lock, the call has **no effect**.



If a thread attempts to notify on a lock that it does **not** own, an exception is thrown (illegal monitor state).

Activity: Taking Turns with Wait/Notify

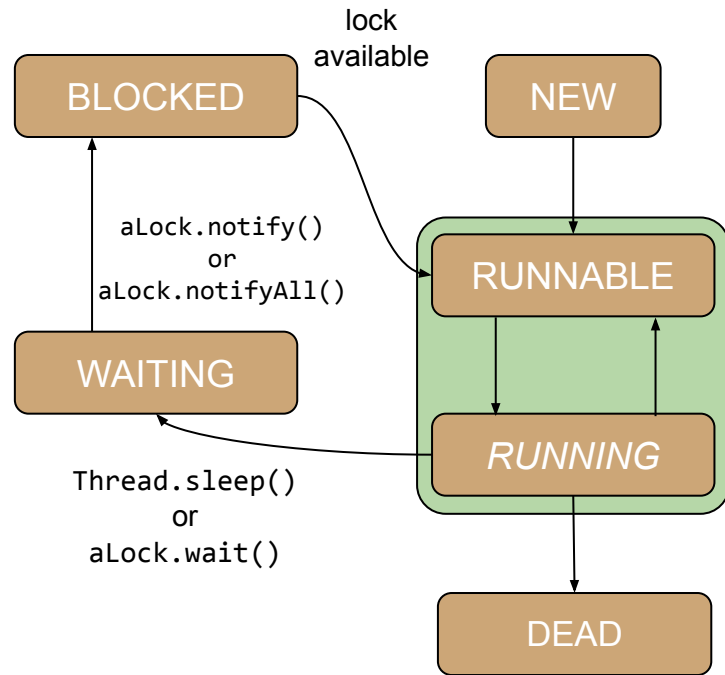


Write a program in which two threads cooperate to count up from zero.

1. One thread should print the odd numbers starting from 1.
2. One thread should print the even numbers starting from 2.
3. Each thread should wait its turn to print.
 - a. How will the even thread know that the odd thread has printed?

Notify All

- The purpose of writing **multithreaded** code is so that a program can do **more than one thing** at the same time.
- Programs often use multiple threads to distribute the **same kind of work** to take care of it more quickly.
- In the event that there is more than one thread waiting on the same lock, it is possible to **notify all of the threads** at the same time.
- This transitions all of the **WAITING** threads into the **BLOCKED** state.
 - Only **one thread** will be able to obtain the lock when it is available.
 - For this reason, cooperating threads **give up the lock** as soon as it is reasonable to do so.



The thread scheduler is still in charge of choosing the thread that obtains the lock. The choice is unpredictable.

Waiting is not Blocked (and Vice Versa)

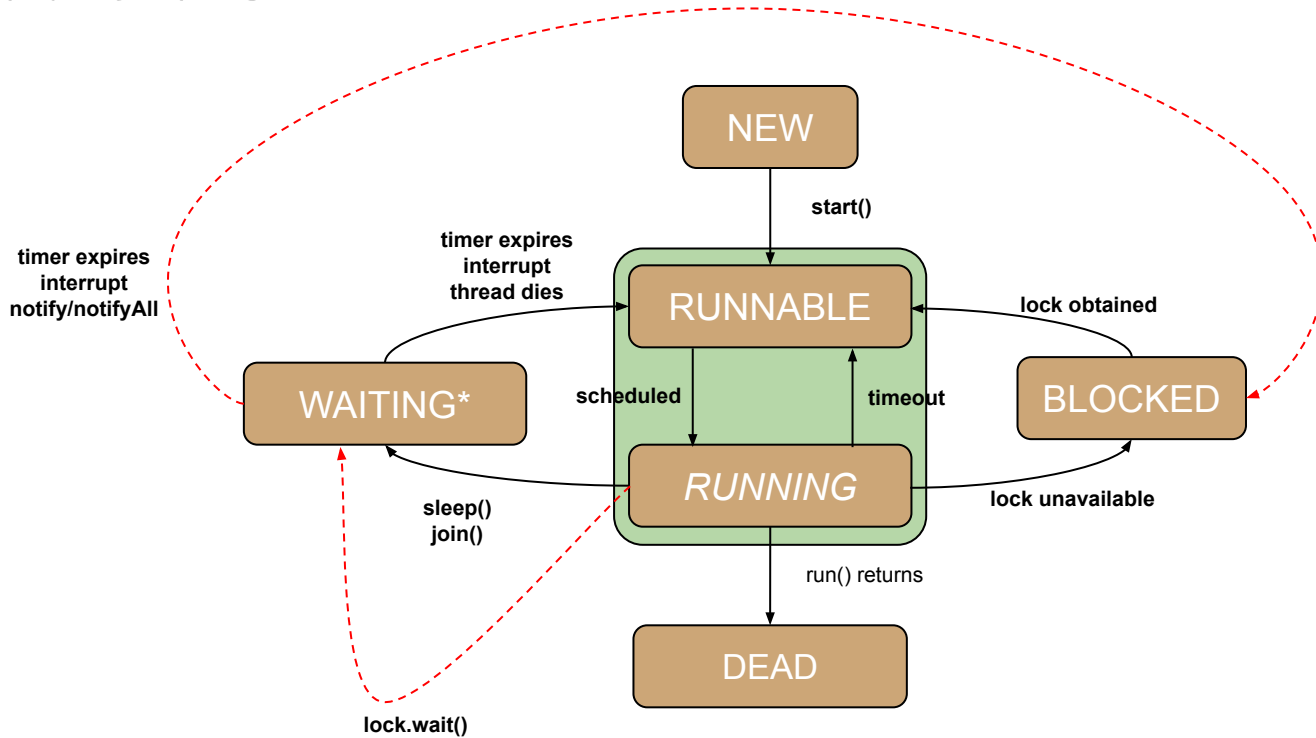


- A **BLOCKED** thread is **not WAITING**.
- A **BLOCKED** thread is **actively** trying to obtain the **synchronized** lock.
- If and when the thread scheduler chooses it, it will **immediately** transition to the **RUNNABLE** state.
- A **BLOCKED** thread **cannot** be interrupted.



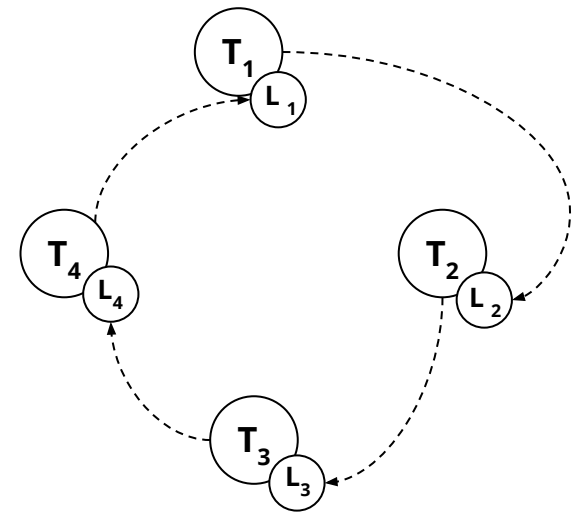
- A **WAITING** thread is waiting for some **important event** to occur.
 - Another thread to **notify** it.
 - A timer to **elapse**.
 - Another thread to **die**.
- A **WAITING** thread will **not** transition to the **RUNNABLE** state, even if a lock becomes **available**.
- A waiting thread **can** be interrupted.

Thread States



Deadlock

- Imagine a scenario:
 - Thread T_1 has obtained lock L_1 .
 - Thread T_2 has obtained lock L_2 .
- What happens if T_1 tries to obtain L_2 and T_2 tries to obtain L_1 ?
 - Each thread already **has one lock** and tries to **get the lock held by the other**.
 - But neither thread **gives up its own lock**.
- Both threads will be **permanently blocked**. This is a condition called **deadlock**.
- Deadlock is really, really **bad**; once deadlocked, threads will **never recover**.
 - There is **no way** in Java to **force** a thread to give up a lock, so a thread **stuck** in deadlock will hold onto its locks **forever**.
- Deadlock is possible whenever a thread tries to obtain **more than one** lock at the same time.



When one thread owns a lock, it holds it until it waits or leaves the synchronized block.

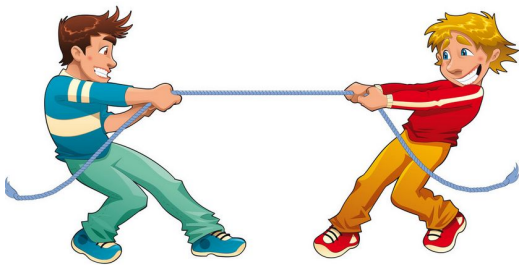
The thread can try to obtain another lock at the same time. But this can cause a condition called **circular hold and wait**.

In a **circular hold and wait**, each thread has one lock and is waiting for another. This causes deadlock.

Activity: Causing Deadlock

Create a new class call Deadlock.

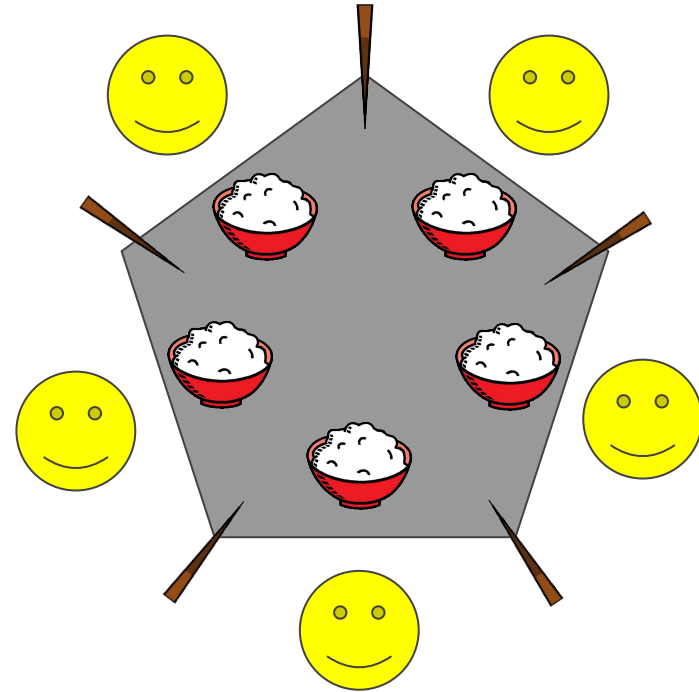
1. Make your class a thread by extending Thread or implementing Runnable.
2. Add three fields:
 - a. String name
 - b. String lock1
 - c. String lock2
3. In the method write code that is likely to produce deadlock.
 - a. Hint: in a loop, try to obtain both locks.
 - b. Print the name and the lock right before trying to obtain it.
4. Add a main method that creates two of your threads and starts them. Does deadlock occur?



Any time two threads try to obtain the **same locks** in **reverse order**, deadlock may occur.

Deadlock & The Dining Philosophers*

- Starvation refers to a thread that gets so little time on the processor that it is unable to do its work.
 - Deadlock is the ultimate form of starvation.
- The Dining Philosopher Problem is a classic problem involving starvation and deadlock.
 - There are 5 philosophers seated at a table.
 - Each has a bowl of rice in front of them.
 - There is a single chopstick between each pair of philosophers.
 - A philosopher must have two chopsticks to eat a grain of rice.
- Design a solution that will allow all of the philosophers to eat without eventually resulting in starvation or deadlock.
- We don't have time to talk about this, but you can check out the supplemental lecture if you are interested.



* Would be a great name for a band.