CS 351
Design of Large Programs
Java Synchronization

Brooke Chenoweth

University of New Mexico

Fall 2024

Concurrency Issues

- The performance benefits of concurrency come with added programming complexity.
- Without the proper use of *mutual exclusion* when accessing shared resources, some concurrency issues can arise:
  - Thread interference (race conditions)
  - Deadlock
  - Livelock
  - Starvation

Recall: Mutual Exclusion

Mutual exclusion is the requirement that no more than one thread of execution may access a particular *critical section* at once.

$$x = 1$$

Thread A	Thread B
$x = x + 1$	$x = x - 2$

What does x equal?

Without mutual exclusion, we don't know which thread will access, manipulate, and store the result first!

Synchronization: Programming Mutual Exclusion

Synchronization is the programming construct used to ensure mutual exclusion in Java.
Two synchronization idioms are used:

Synchronized methods

```
public synchronized void foo() {
  // ...
}
```

Only one thread at a time can execute in a synchronized method. The lock is on the object providing the methods.

Synchronized blocks

```
public void foo() {
  synchronized(lock) {
    // ...
  }
}
```

Only the thread possessing the lock can execute in a synchronized block.

Example: Counter

Consider a static Counter which allows *two* Farmers to keep track of the number of sheep in the pen.

```
public class Counter {
  private static int n = 0;

  public static void increment() {
    n++;
  }

  public static void decrement() {
    n--;
  }

  public static void printCount() {
    // print sheep count...
  }
}
```

```
public class Farmer
    implements Runnable {

  @Override
  public void run() {
    // two sheep arrive
    Counter.increment();
    Counter.increment();

    // one sheep leaves
    Counter.decrement();
  }
}
```

How many sheep are in the pen?

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

```
2 sheep in the pen.
```

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

```
2 sheep in the pen.
3 sheep in the pen.
```

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

```
2 sheep in the pen.
3 sheep in the pen.
2 sheep in the pen.
```

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

```
2 sheep in the pen.
3 sheep in the pen.
2 sheep in the pen.
1 sheep in the pen.
```

Counter: Console Output

```
public static void main(String[] args) {
  Farmer f1 = new Farmer();
  Thread t1 = new Thread(f1);

  Farmer f2 = new Farmer();
  Thread t2 = new Thread(f2);

  t1.start();
  t2.start();

  Counter.printCount();
}
```

Output

```
2 sheep in the pen.
3 sheep in the pen.
2 sheep in the pen.
1 sheep in the pen.
```

We have a *race condition.*

Access to the critical region (where we manipulate the shared resource n) should be synchronized.

Counter: Synchronized Methods

In this trivial example, synchronizing access to both methods eliminates the race condition.

```
public class Counter {
  private static int n = 0;

  public static synchronized void increment() {
    n++;
  }

  public static synchronized void decrement() {
    n--;
  }
}
```

Output

Counter: Synchronized Methods

In this trivial example, synchronizing access to both methods eliminates the race condition.

```
public class Counter {
  private static int n = 0;

  public static synchronized void increment() {
    n++;
  }

  public static synchronized void decrement() {
    n--;
  }
}
```

Output

```
2 sheep in the pen.
```

Counter: Synchronized Methods

In this trivial example, synchronizing access to both methods eliminates the race condition.

```
public class Counter {
  private static int n = 0;

  public static synchronized void increment() {
    n++;
  }

  public static synchronized void decrement() {
    n--;
  }
}
```

Output

```
2 sheep in the pen.
2 sheep in the pen.
```

Counter: Synchronized Methods

In this trivial example, synchronizing access to both methods eliminates the race condition.

```
public class Counter {
  private static int n = 0;

  public static synchronized void increment() {
    n++;
  }

  public static synchronized void decrement() {
    n--;
  }
}
```

Output

```
2 sheep in the pen.
2 sheep in the pen.
2 sheep in the pen.
```

Counter: Synchronized Methods

In this trivial example, synchronizing access to both methods eliminates the race condition.

```
public class Counter {
  private static int n = 0;

  public static synchronized void increment() {
    n++;
  }

  public static synchronized void decrement() {
    n--;
  }
}
```

Output

```
2 sheep in the pen.
2 sheep in the pen.
2 sheep in the pen.
2 sheep in the pen.
```

Case Study: Producer/Consumer problem

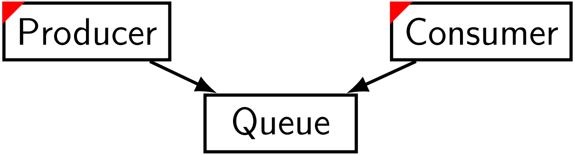
- Consider a *queue* which buffers data provided by a Producer and removed by a Consumer
  - the queue has a maximum size
  - the Producer should not add to a full queue
  - the Consumer should not consume from an empty queue
- How do we avoid...
  - putting objects in a full queue?
  - attempting to remove them from an empty one?

Case Study: Producer/Consumer problem

- Consider a *queue* which buffers data provided by a Producer and removed by a Consumer
  - the queue has a maximum size
  - the Producer should not add to a full queue
  - the Consumer should not consume from an empty queue
- How do we avoid...
  - putting objects in a full queue?
  - attempting to remove them from an empty one?
- We will explore a solution that implements a *singleton* queue.
  - Educational purposes only!
  - BlockingQueue implementations already exist!

Producer/Consumer: Design Overview

- Producer (thread)
  - integers are placed in the Queue
  - waits when the Queue is full
- Queue – a singleton queue
  - holds integer values
  - starts being empty
  - has a maximum size
- Consumer (thread)
  - waits for a non-empty Queue
  - consumes its contents



Queue: recall the singleton pattern...

- A single instance of Queue is referenced globally by the Producer and Consumer
  - thus, we can implement it using the *singleton pattern*.
- Recall that the singleton pattern is *not necessarily thread-safe* unless implemented correctly.
  - lazy vs. eager instantiation
- Let's see why...

Queue: Lazy Instantiation

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance;

  private Queue() {}

  public static Queue getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Queue();
    }
    return uniqueInstance;
  }

  // enqueue(), dequeue()...
}
```

Queue: Lazy Instantiation Pitfall

Consider the case in which the Producer and Consumer running concurrently in separate threads call `Queue.getInstance()` in order to access data in the queue...

```
1 public static Queue getInstance() {
2   if (uniqueInstance == null) {
3     uniqueInstance = new Queue();
4   }
5   return uniqueInstance;
6 }
```

If Consumer reaches line 2 before Producer has instantiated `uniqueInstance` on line 3, `getInstance()` will return two separate, unique instances of Queue!

Queue: Eager Instantiation Fix

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance = new Queue();

  private Queue() {}

  public static Queue getInstance() {
    return uniqueInstance;
  }

  // enqueue(), dequeue()...
}
```

`uniqueInstance` is instantiated eagerly (i.e. before we know we need it). What is another way of addressing this thread-safety issue?

Queue: Synchronized getInstance Fix

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance;

  private Queue() {}

  public static synchronized Queue getInstance() {
    if (uniqueInstance == null) {
      uniqueInstance = new Queue();
    }
    return uniqueInstance;
  }

  // enqueue(), dequeue()...
}
```

Here, we preserve lazy instantiation but *synchronize access* to `getInstance()`, ensuring that only one thread will be active in the method at a time.

Producer/Consumer Concept

- In this example (and often in practice), Producers and Consumers are tasks running concurrently in different threads while exchanging information through a shared data structure.
- Their execution is coordinated using synchronized methods accessed *within the Queue object*.

Producer

```
public class Producer implements Runnable {
  @Override
  public void run() {
    Queue queue = Queue.getInstance();

    while (!Thread.interrupted()) {
      Integer newData = new Random().nextInt();
      queue.enqueue(newData);
    }
  }
}
```

Consumer

```
public class Consumer implements Runnable {
  @Override
  public void run() {
    Queue queue = Queue.getInstance();

    while (!Thread.interrupted()) {
      // consume the last value in the queue
      queue.dequeue();
    }
  }
}
```

Producer/Consumer: Guarded Blocks

- While the Producer and Consumer wait for the Queue to be in an appropriate state (non-full or non-empty, respectively), they must perform *guarded blocks*.
- Guarded blocks allow the execution of threads to be coordinated based upon the state of shared variables.
- There are two types of guarded blocks:
  - **Bad**: Busy waiting
  - **Good**: Wait/notify

Queue: Guarded block with a busy wait

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance;

  // constructor, getInstance() ...

  public void enqueue(Integer data) {
    while (dataQueue.size() >= CAPACITY) {
      // wait...
    }
    dataQueue.add(data);
  }
}
```

Spin in a while loop while we wait for the queue to be non-full.

Busy waits are a waste of processor resources!

Queue: Guarded block with wait/notify

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance;

  // constructor, getInstance() ...

  public synchronized void enqueue(Integer data) {
    while (dataQueue.size() >= CAPACITY) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    dataQueue.add(data);
    notifyAll();
  }
}
```

What's happening here?

- enqueue is synchronized now. *wait calls must be performed by threads currently holding the* lock in a synchronized method or block.
- We still have a while loop: *wait calls must occur in a loop.* Otherwise, our waiting thread might be asleep when it's notified by another thread to wake.
- Despite the while loop, *our call to wait signals the thread scheduler to use processor resources elsewhere*
  No more busy wait!

Queue: Guarded block on dequeue

```
public class Queue {
  private static final int CAPACITY = 5;
  private List<Integer> dataQueue = new ArrayList<>();
  private static Queue uniqueInstance;

  // constructor, getInstance(), enqueue()...

  public synchronized Integer dequeue() {
    while (dataQueue.size() == 0) {
      try {
        wait();
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }
    notifyAll();
    return dataQueue.remove(dataQueue.size() - 1);
  }
}
```

Producer/Consumer: Conclusion

- Using synchronization idioms, the execution of two different threads (in this case, Producer and Consumer) can be coordinated.
- Is synchronization always necessary?
  - no, and in some cases it can be a detriment to performance
  - redundant when applied to threads that possess mutually exclusive, private state