

QA

1. Blocking Queue | Bounded Buffer | Consumer Producer

Classical synchronization problem involving a limited size buffer which can have items added to it or removed from it by different producer and consumer threads. This problem is known by different names: consumer producer problem, bounded buffer problem or blocking queue problem.

Problem Statement

A blocking queue is defined as a queue which blocks the caller of the enqueue method if there's no more capacity to add the new item being enqueued. Similarly, the queue blocks the dequeue caller if there are no items in the queue. Also, the queue notifies a blocked enqueueing thread when space becomes available and a blocked dequeuing thread when an item becomes available in the queue.

▼ Solution1:

```
class Demonstration {
    public static void main( String args[] ) throws Exception{
        final BlockingQueue<Integer> q = new BlockingQueue<Integer>(5);

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    for (int i = 0; i < 50; i++) {
                        q.enqueue(new Integer(i));
                        System.out.println("enqueued " + i);
                    }
                } catch (InterruptedException ie) {

                }
            }
        })
    }
}
```

```

    }
});

Thread t2 = new Thread(new Runnable() {

    @Override
    public void run() {
        try {
            for (int i = 0; i < 25; i++) {
                System.out.println("Thread 2 dequeued: " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});

Thread t3 = new Thread(new Runnable() {

    @Override
    public void run() {
        try {
            for (int i = 0; i < 25; i++) {
                System.out.println("Thread 3 dequeued: " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});

t1.start();
Thread.sleep(4000);
t2.start();

t2.join();

```

```

        t3.start();
        t1.join();
        t3.join();
    }
}

```

// The blocking queue class

```
class BlockingQueue<T> {
```

```

    T[] array;
    Object lock = new Object();
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;

```

```

    @SuppressWarnings("unchecked")
    public BlockingQueue(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
    }

```

```
    public void enqueue(T item) throws InterruptedException {
```

```

        synchronized (lock) {

```

```

            while (size == capacity) {
                lock.wait();
            }

```

```

            if (tail == capacity) {
                tail = 0;
            }

```

```

        array[tail] = item;
        size++;
        tail++;
        lock.notifyAll();
    }
}

public T dequeue() throws InterruptedException {

    T item = null;
    synchronized (lock) {

        while (size == 0) {
            lock.wait();
        }

        if (head == capacity) {
            head = 0;
        }

        item = array[head];
        array[head] = null;
        head++;
        size--;

        lock.notifyAll();
    }

    return item;
}
}

```

▼ Solution2: **Busy wait solution using Lock#**

```

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        final BlockingQueueWithMutex<Integer> q = new BlockingQueueWithMu

Thread producer1 = new Thread(new Runnable() {
    public void run() {
        try {
            int i = 1;
            while (true) {
                q.enqueue(i);
                System.out.println("Producer thread 1 enqueued " + i);
                i++;
            }
        } catch (InterruptedException ie) {
        }
    }
});

Thread producer2 = new Thread(new Runnable() {
    public void run() {
        try {
            int i = 5000;
            while (true) {
                q.enqueue(i);
                System.out.println("Producer thread 2 enqueued " + i);
                i++;
            }
        } catch (InterruptedException ie) {
        }
    }
});

Thread producer3 = new Thread(new Runnable() {
    public void run() {

```

```

        try {
            int i = 100000;
            while (true) {
                q.enqueue(i);
                System.out.println("Producer thread 3 enqueued " + i);
                i++;
            }
        } catch (InterruptedException ie) {

        }
    }
});

Thread consumer1 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 1 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});

Thread consumer2 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 2 dequeued " + q.dequeue());
            }
        } catch (InterruptedException ie) {

        }
    }
});

```

```

Thread consumer3 = new Thread(new Runnable() {
    public void run() {
        try {
            while (true) {
                System.out.println("Consumer thread 3 dequeued " + q.deque
            }
        } catch (InterruptedException ie) {

        }
    }
});

producer1.setDaemon(true);
producer2.setDaemon(true);
producer3.setDaemon(true);
consumer1.setDaemon(true);
consumer2.setDaemon(true);
consumer3.setDaemon(true);

producer1.start();
producer2.start();
producer3.start();

consumer1.start();
consumer2.start();
consumer3.start();

Thread.sleep(1000);
}
}

```

▼ Solution3: **Solution using semaphores#**

```

public class BlockingQueueWithSemaphore<T> {
    T[] array;
    int size = 0;
    int capacity;
    int head = 0;
    int tail = 0;
    CountingSemaphore semLock = new CountingSemaphore(1, 1);
    CountingSemaphore semProducer;
    CountingSemaphore semConsumer;

    @SuppressWarnings("unchecked")
    public BlockingQueueWithSemaphore(int capacity) {
        // The casting results in a warning
        array = (T[]) new Object[capacity];
        this.capacity = capacity;
        this.semProducer = new CountingSemaphore(capacity, capacity);
        this.semConsumer = new CountingSemaphore(capacity, 0);
    }

    public T dequeue() throws InterruptedException {

        T item = null;

        semConsumer.acquire();
        semLock.acquire();

        if (head == capacity) {
            head = 0;
        }

        item = array[head];
        array[head] = null;
        head++;
        size--;
    }
}

```



```

        semLock.release();
        semProducer.release();

        return item;
    }

    public void enqueue(T item) throws InterruptedException {

        semProducer.acquire();
        semLock.acquire();

        if (tail == capacity) {
            tail = 0;
        }

        array[tail] = item;
        size++;
        tail++;

        semLock.release();
        semConsumer.release();
    }
}

```

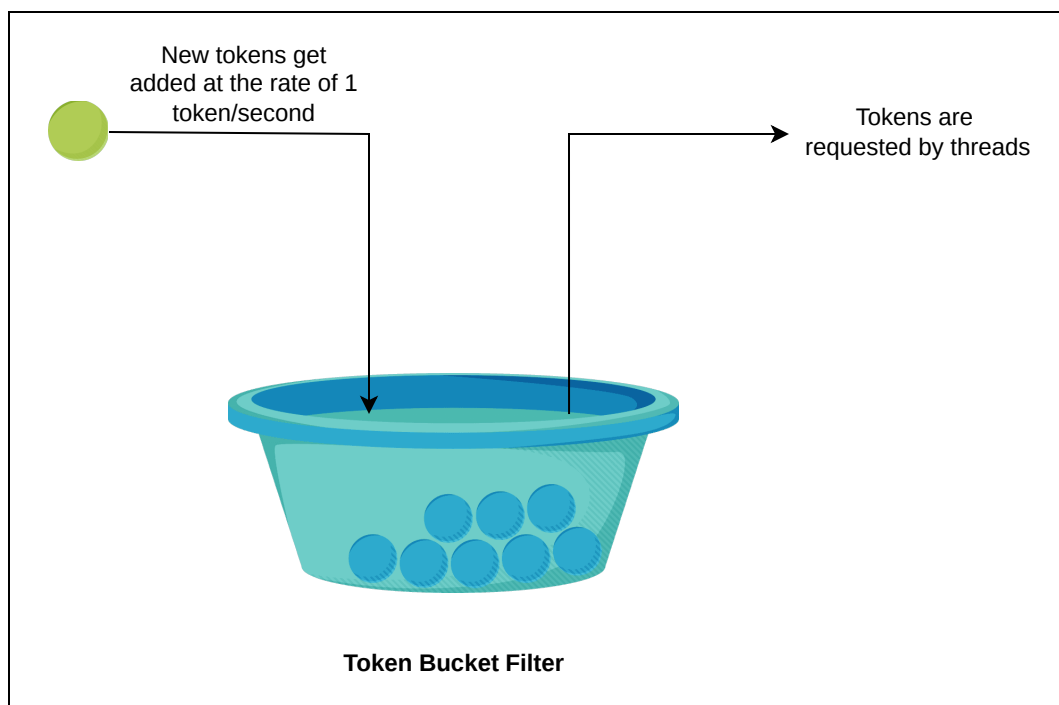
2. Rate Limiting Using Token Bucket Filter

Implementing rate limiting using a naive token bucket filter algorithm.

Problem Statement

This is an actual interview question asked at Uber and Oracle. Imagine you have a bucket that gets filled with tokens at the rate of 1 token per second. The bucket can hold a maximum of N tokens. Implement a thread-safe class that lets threads get a token when one is available. If no token is available, then the token-requesting threads should block. The class should expose an API called `getToken` that various threads can call to get a token.

Press+to interact



▼ Solution1:

```

import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        TokenBucketFilter.runTestMaxTokensIs1();
    }
}

class TokenBucketFilter {

    private int MAX_TOKENS;
    private long lastRequestTime = System.currentTimeMillis();
    long possibleTokens = 0;

    public TokenBucketFilter(int maxTokens) {
        MAX_TOKENS = maxTokens;
    }

    synchronized void getToken() throws InterruptedException {

        possibleTokens += (System.currentTimeMillis() - lastRequestTime) / 100;

        if (possibleTokens > MAX_TOKENS) {
            possibleTokens = MAX_TOKENS;
        }

        if (possibleTokens == 0) {
            Thread.sleep(1000);
        } else {
            possibleTokens--;
        }
        lastRequestTime = System.currentTimeMillis();

        System.out.println("Granting " + Thread.currentThread().getName() + " 1 token");
    }
}

```

```

    }

    public static void runTestMaxTokens1() throws InterruptedException {

        Set<Thread> allThreads = new HashSet<Thread>();
        final TokenBucketFilter tokenBucketFilter = new TokenBucketFilter(1);

        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new Runnable() {
                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }
    }
}

```

▼ Solution2: **Solution using a background thread#**

```

import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        final MultithreadedTokenBucketFilter tokenBucketFilter = new Multithrea

        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            });
            thread.setName("Thread_" + (i + 1));
            allThreads.add(thread);
        }

        for (Thread t : allThreads) {
            t.start();
        }

        for (Thread t : allThreads) {
            t.join();
        }

    }
}

```

```

class MultithreadedTokenBucketFilter {
    private long possibleTokens = 0;
    private final int MAX_TOKENS;
    private final int ONE_SECOND = 1000;

    public MultithreadedTokenBucketFilter(int maxTokens) {

        MAX_TOKENS = maxTokens;

        // Never start a thread in a constructor
        Thread dt = new Thread(() → {
            daemonThread();
        });
        dt.setDaemon(true);
        dt.start();
    }

    private void daemonThread() {

        while (true) {

            synchronized (this) {
                if (possibleTokens < MAX_TOKENS) {
                    possibleTokens++;
                }
                this.notify();
            }

            try {
                Thread.sleep(ONE_SECOND);
            } catch (InterruptedException ie) {
                // swallow exception
            }
        }
    }
}

```

```

void getToken() throws InterruptedException {

    synchronized (this) {
        while (possibleTokens == 0) {
            this.wait();
        }
        possibleTokens--;
    }

    System.out.println(
        "Granting " + Thread.currentThread().getName() + " token at " + Sy
    }
}

```

▼ Solution3: **Using a factory**

```

import java.util.HashSet;
import java.util.Set;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        Set<Thread> allThreads = new HashSet<Thread>();
        TokenBucketFilter tokenBucketFilter = TokenBucketFilterFactory.makeTc

        for (int i = 0; i < 10; i++) {

            Thread thread = new Thread(new Runnable() {

                public void run() {
                    try {
                        tokenBucketFilter.getToken();
                    } catch (InterruptedException ie) {
                        System.out.println("We have a problem");
                    }
                }
            }
        }
    }
}

```

```

    });
    thread.setName("Thread_" + (i + 1));
    allThreads.add(thread);
}

for (Thread t : allThreads) {
    t.start();
}

for (Thread t : allThreads) {
    t.join();
}

}
}

```

3. Thread Safe Deferred Callback

Asynchronous programming involves being able to execute functions at a future occurrence of some event. Designing a thread-safe deferred callback class becomes a challenging interview question.

Problem Statement

Design and implement a thread-safe class that allows registration of callback methods that are executed after a user specified time interval in seconds has elapsed.

▼ Solution

```

import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {

```



```

    public static void main( String args[] ) throws InterruptedException {
        DeferredCallbackExecutor.runTestTenCallbacks();
    }
}

class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparat
    public int compare(CallBack o1, CallBack o2) {
        return (int) (o1.executeAt - o2.executeAt);
    }
});
ReentrantLock lock = new ReentrantLock();
Condition newCallbackArrived = lock.newCondition();

private long findSleepDuration() {
    long currentTime = System.currentTimeMillis();
    return q.peek().executeAt - currentTime;
}

public void start() throws InterruptedException {
    long sleepFor = 0;

    while (true) {

        lock.lock();

        while (q.size() == 0) {
            newCallbackArrived.await();
        }

        while (q.size() != 0) {
            sleepFor = findSleepDuration();

            if(sleepFor <=0)
                break;
        }
    }
}

```

```

        newCallbackArrived.await(sleepFor, TimeUnit.MILLISECONDS);
    }

    Callback cb = q.poll();
    System.out.println(
        "Executed at " + System.currentTimeMillis()/1000 + " required at
        + ": message:" + cb.message);

    lock.unlock();
}
}

public void registerCallback(Callback callBack) {
    lock.lock();
    q.add(callBack);
    newCallbackArrived.signal();
    lock.unlock();
}

static class Callback {
    long executeAt;
    String message;

    public Callback(long executeAfter, String message) {
        this.executeAt = System.currentTimeMillis() + (executeAfter * 1000);
        this.message = message;
    }
}

public static void runTestTenCallbacks() throws InterruptedException {
    Set<Thread> allThreads = new HashSet<Thread>();
    final DeferredCallbackExecutor deferredCallbackExecutor = new DeferredCallbackExecutor();

    Thread service = new Thread(new Runnable() {
        public void run() {

```

```

        try {
            deferredCallbackExecutor.start();
        } catch (InterruptedException ie) {

        }
    }
});

service.start();

for (int i = 0; i < 10; i++) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            Callback cb = new Callback(1, "Hello this is " + Thread.currentThread().getName());
            deferredCallbackExecutor.registerCallback(cb);
        }
    });
    thread.setName("Thread_" + (i + 1));
    thread.start();
    allThreads.add(thread);
    Thread.sleep(1000);
}

for (Thread t : allThreads) {
    t.join();
}
}
}

```

▼ Solution: Part2

```

import java.util.*;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.ReentrantLock;

```

```

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        DeferredCallbackExecutor.runTestTenCallbacks();
    }
}

class DeferredCallbackExecutor {

    PriorityQueue<CallBack> q = new PriorityQueue<CallBack>(new Comparat
    public int compare(CallBack o1, CallBack o2) {
        return (int) (o1.executeAt - o2.executeAt);
    }
});
ReentrantLock lock = new ReentrantLock();
Condition newCallbackArrived = lock.newCondition();

private long findSleepDuration() {
    long currentTime = System.currentTimeMillis();
    return q.peek().executeAt - currentTime;
}

public void start() throws InterruptedException {
    long sleepFor = 0;

    while (true) {

        lock.lock();

        while (q.size() == 0) {
            newCallbackArrived.await();
        }

        while (q.size() != 0) {
            sleepFor = findSleepDuration();

```

```

        if(sleepFor <=0)
            break;

        newCallbackArrived.await(sleepFor, TimeUnit.MILLISECONDS);
    }

    Callback cb = q.poll();
    System.out.println(
        "Executed at " + System.currentTimeMillis()/1000 + " required at
        + ": message:" + cb.message);

    lock.unlock();
}
}

public void registerCallback(Callback callBack) {
    lock.lock();
    q.add(callBack);
    newCallbackArrived.signal();
    lock.unlock();
}

static class Callback {
    long executeAt;
    String message;

    public Callback(long executeAfter, String message) {
        this.executeAt = System.currentTimeMillis() + (executeAfter * 1000);
        this.message = message;
    }
}

public static void runTestTenCallbacks() throws InterruptedException {
    Set<Thread> allThreads = new HashSet<Thread>();
    final DeferredCallbackExecutor deferredCallbackExecutor = new DeferredCallbackExecutor();

```

```

Thread service = new Thread(new Runnable() {
    public void run() {
        try {
            deferredCallbackExecutor.start();
        } catch (InterruptedException ie) {

        }
    }
});

service.start();

for (int i = 0; i < 10; i++) {
    Thread thread = new Thread(new Runnable() {
        public void run() {
            Callback cb = new Callback(1, "Hello this is " + Thread.currentThread().getName() + " ");
            deferredCallbackExecutor.registerCallback(cb);
        }
    });
    thread.setName("Thread_" + (i + 1));
    thread.start();
    allThreads.add(thread);
    Thread.sleep(1000);
}

for (Thread t : allThreads) {
    t.join();
}
}
}

```

4. Implementing Semaphore

Learn how to design and implement a simple semaphore class in Java.

Problem Statement

Java does provide its own implementation of Semaphore, however, Java's semaphore is initialized with an initial number of permits, rather than the maximum possible permits and the developer is expected to take care of always releasing the intended number of maximum permits.

Briefly, a semaphore is a construct that allows some threads to access a fixed set of resources in parallel. Always think of a semaphore as having a fixed number of permits to give out. Once all the permits are given out, requesting threads, need to wait for a permit to be returned before proceeding forward.

Your task is to implement a semaphore which takes in its constructor the maximum number of permits allowed and is also initialized with the same number of permits.

▼ Solution:

```
class Demonstration {
    public static void main( String args[] ) throws InterruptedException {

        final CountingSemaphore cs = new CountingSemaphore(1);

        Thread t1 = new Thread(new Runnable() {

            @Override
            public void run() {
                try {
                    for (int i = 0; i < 5; i++) {
                        cs.acquire();
                        System.out.println("Ping " + i);
                    }
                } catch (InterruptedException ie) {

                }
            }
        });
    }
}
```

```

Thread t2 = new Thread(new Runnable() {

    @Override
    public void run() {
        for (int i = 0; i < 5; i++) {
            try {
                cs.release();
                System.out.println("Pong " + i);
            } catch (InterruptedException ie) {

            }
        }
    }
});

t2.start();
t1.start();
t1.join();
t2.join();
}
}

class CountingSemaphore {

    int usedPermits = 0;
    int maxCount;

    public CountingSemaphore(int count) {
        this.maxCount = count;
    }

    public synchronized void acquire() throws InterruptedException {

        while (usedPermits == maxCount)
            wait();
    }
}

```



```

        notify();
        usedPermits++;
    }

    public synchronized void release() throws InterruptedException {

        while (usedPermits == 0)
            wait();

        usedPermits--;
        notify();
    }
}

```

5. ReadWrite Lock

We discuss a common interview question involving synchronization of multiple reader threads and a single writer thread.

Problem Statement

Imagine you have an application where you have multiple readers and multiple writers. You are asked to design a lock which lets multiple readers read at the same time, but only one writer write at a time.

▼ Solution

```

class Demonstration {

    public static void main(String args[]) throws Exception {

        final ReadWriteLock rwl = new ReadWriteLock();

        Thread t1 = new Thread(new Runnable() {

```

```

@Override
public void run() {
    try {

        System.out.println("Attempting to acquire write lock in t1: " + Sys
        rwl.acquireWriteLock();
        System.out.println("write lock acquired t1: " + +System.currentTim

        // Simulates write lock being held indefinitely
        for (; ; ) {
            Thread.sleep(500);
        }

    } catch (InterruptedException ie) {

    }
}
});

```

```

Thread t2 = new Thread(new Runnable() {

```

```

@Override
public void run() {
    try {

        System.out.println("Attempting to acquire write lock in t2: " + Sys
        rwl.acquireWriteLock();
        System.out.println("write lock acquired t2: " + System.currentTim

    } catch (InterruptedException ie) {

    }
}
});

```

```

Thread tReader1 = new Thread(new Runnable() {

    @Override
    public void run() {
        try {

            rwl.acquireReadLock();
            System.out.println("Read lock acquired: " + System.currentTimeMillis());

        } catch (InterruptedException ie) {

        }

    }
});

Thread tReader2 = new Thread(new Runnable() {

    @Override
    public void run() {
        System.out.println("Read lock about to release: " + System.currentTimeMillis());
        rwl.releaseReadLock();
        System.out.println("Read lock released: " + System.currentTimeMillis());
    }
});

tReader1.start();
t1.start();
Thread.sleep(3000);
tReader2.start();
Thread.sleep(1000);
t2.start();
tReader1.join();
tReader2.join();
t2.join();
}
}

```

```
class ReadWriteLock {

    boolean isWriteLocked = false;
    int readers = 0;

    public synchronized void acquireReadLock() throws InterruptedException {

        while (isWriteLocked) {
            wait();
        }

        readers++;
    }

    public synchronized void releaseReadLock() {
        readers--;
        notify();
    }

    public synchronized void acquireWriteLock() throws InterruptedException {

        while (isWriteLocked || readers != 0) {
            wait();
        }

        isWriteLocked = true;
    }

    public synchronized void releaseWriteLock() {
        isWriteLocked = false;
        notify();
    }
}
```

6.Unisex Bathroom Problem

A synchronization practice problem requiring us to synchronize the usage of a single bathroom by both the genders.

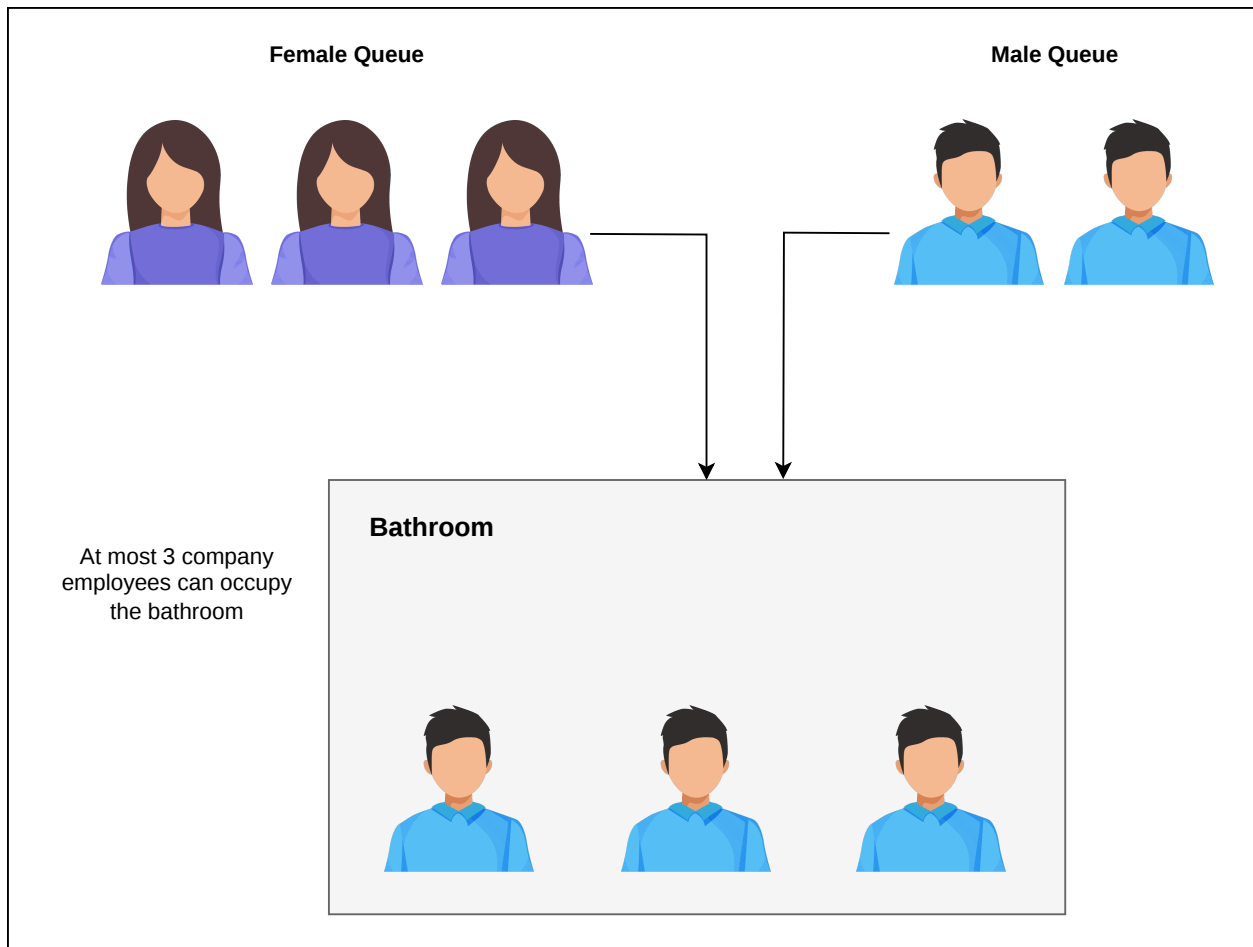
Problem Statement

A bathroom is being designed for the use of both males and females in an office but requires the following constraints to be maintained:

- There cannot be men and women in the bathroom at the same time.
- There should never be more than three employees in the bathroom simultaneously.

The solution should avoid deadlocks. For now, though, don't worry about starvation.

Press+to interact



▼ Solution

```
import java.util.concurrent.Semaphore;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        UnisexBathroom.runTest();
    }
}

class UnisexBathroom {

    static String WOMEN = "women";
```

```

static String MEN = "men";
static String NONE = "none";

String inUseBy = NONE;
int empsInBathroom = 0;
Semaphore maxEmps = new Semaphore(3);

void useBathroom(String name) throws InterruptedException {
    System.out.println("\n" + name + " using bathroom. Current employees " + empsInBathroom);
    Thread.sleep(3000);
    System.out.println("\n" + name + " done using bathroom " + System.currentTimeMillis());
}

void maleUseBathroom(String name) throws InterruptedException {

    synchronized (this) {
        while (inUseBy.equals(WOMEN)) {
            this.wait();
        }
        maxEmps.acquire();
        empsInBathroom++;
        inUseBy = MEN;
    }

    useBathroom(name);
    maxEmps.release();

    synchronized (this) {
        empsInBathroom--;

        if (empsInBathroom == 0) inUseBy = NONE;
        this.notifyAll();
    }
}

void femaleUseBathroom(String name) throws InterruptedException {

```



```

synchronized (this) {
    while (inUseBy.equals(MEN)) {
        this.wait();
    }
    maxEmps.acquire();
    empsInBathroom++;
    inUseBy = WOMEN;
}

useBathroom(name);
maxEmps.release();

synchronized (this) {
    empsInBathroom--;

    if (empsInBathroom == 0) inUseBy = NONE;
    this.notifyAll();
}
}

public static void runTest() throws InterruptedException {

    final UnisexBathroom unisexBathroom = new UnisexBathroom();

    Thread female1 = new Thread(new Runnable() {
        public void run() {
            try {
                unisexBathroom.femaleUseBathroom("Lisa");
            } catch (InterruptedException ie) {

            }
        }
    });

    Thread male1 = new Thread(new Runnable() {

```

```

        public void run() {
            try {
                unisexBathroom.maleUseBathroom("John");
            } catch (InterruptedException ie) {

            }
        }
    });

    Thread male2 = new Thread(new Runnable() {
        public void run() {
            try {
                unisexBathroom.maleUseBathroom("Bob");
            } catch (InterruptedException ie) {

            }
        }
    });

    Thread male3 = new Thread(new Runnable() {
        public void run() {
            try {
                unisexBathroom.maleUseBathroom("Anil");
            } catch (InterruptedException ie) {

            }
        }
    });

    Thread male4 = new Thread(new Runnable() {
        public void run() {
            try {
                unisexBathroom.maleUseBathroom("Wentao");
            } catch (InterruptedException ie) {

            }
        }
    });

```

```
    }  
    });  
  
    female1.start();  
    male1.start();  
    male2.start();  
    male3.start();  
    male4.start();  
  
    female1.join();  
    male1.join();  
    male2.join();  
    male3.join();  
    male4.join();  
  
    }  
}
```

7. Implementing a Barrier

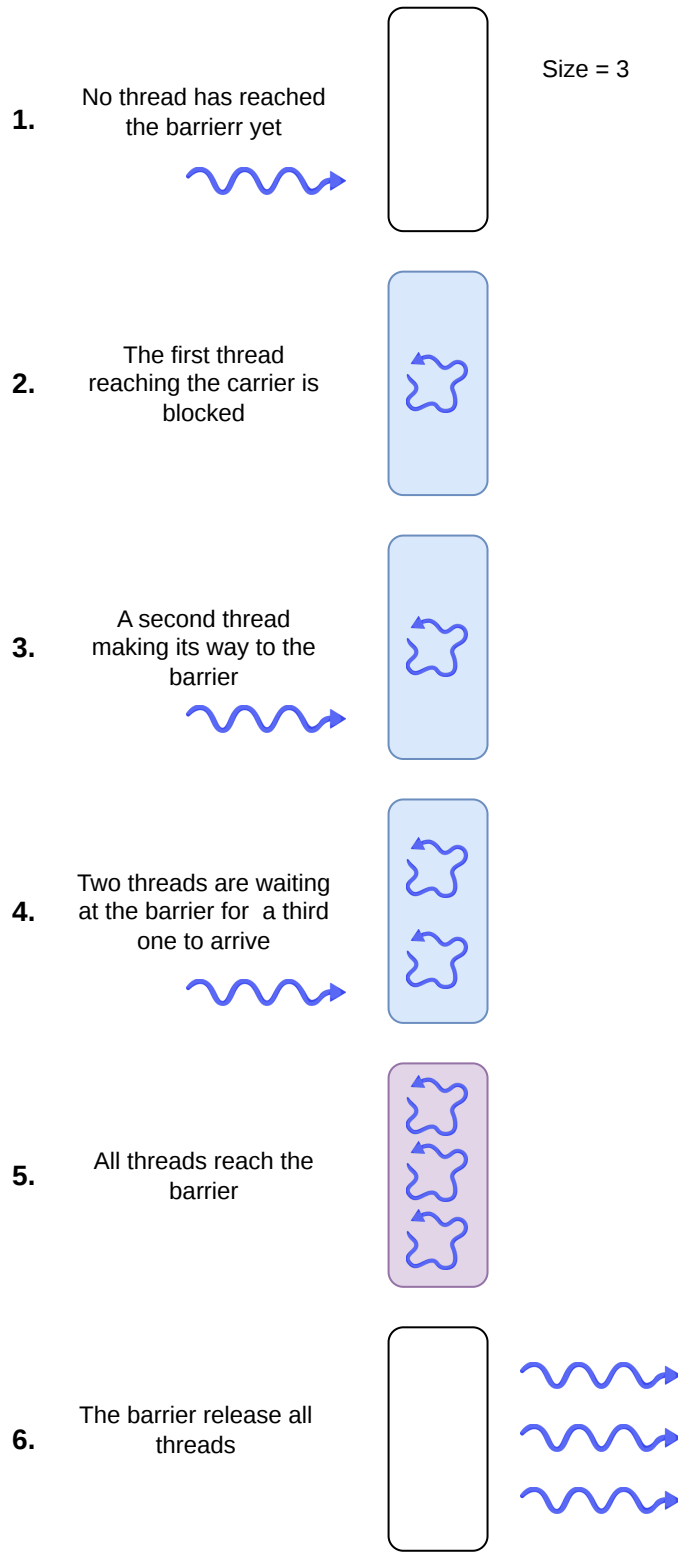
This lesson discusses how a barrier can be implemented in Java.

Problem Statment

A barrier can be thought of as a point in the program code, which all or some of the threads need to reach at before any one of them is allowed to proceed further.

Press+to interact

Working of a Barrier



▼ Solution

```
class Demonstration {
    public static void main( String args[] ) throws Exception{
        Barrier.runTest();
    }
}

class Barrier {

    int count = 0;
    int released = 0;
    int totalThreads;

    public Barrier(int totalThreads) {
        this.totalThreads = totalThreads;
    }

    public static void runTest() throws InterruptedException {
        final Barrier barrier = new Barrier(3);

        Thread p1 = new Thread(new Runnable() {
            public void run() {
                try {
                    System.out.println("Thread 1");
                    barrier.await();
                    System.out.println("Thread 1");
                    barrier.await();
                    System.out.println("Thread 1");
                    barrier.await();
                } catch (InterruptedException ie) {
                }
            }
        });
    }
};
```

```

Thread p2 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
            Thread.sleep(500);
            System.out.println("Thread 2");
            barrier.await();
        } catch (InterruptedException ie) {
        }
    }
});

```

```

Thread p3 = new Thread(new Runnable() {
    public void run() {
        try {
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
            Thread.sleep(1500);
            System.out.println("Thread 3");
            barrier.await();
        } catch (InterruptedException ie) {
        }
    }
});

```

```

p1.start();

```

```

        p2.start();
        p3.start();

        p1.join();
        p2.join();
        p3.join();
    }

    public synchronized void await() throws InterruptedException {

        while (count == totalThreads)
            wait();

        count++;

        if (count == totalThreads) {
            notifyAll();
            released = totalThreads;
        } else {

            while (count < totalThreads)
                wait();
        }

        released--;
        if (released == 0) {
            count = 0;
            // remember to wakeup any threads
            // waiting on line#81
            notifyAll();
        }
    }
}

```


8. Uber Ride Problem

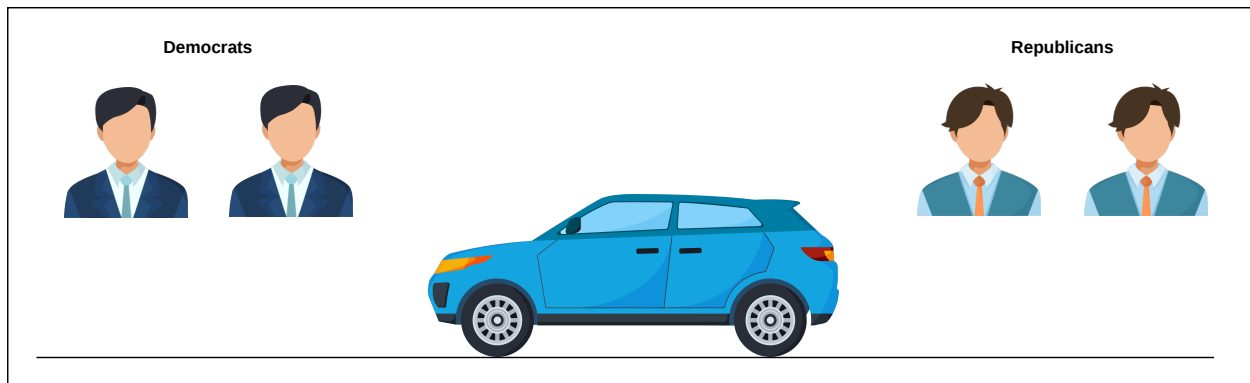
This lesson solves the constraints of an imaginary Uber ride problem where Republicans and Democrats can't be seated as a minority in a four passenger car.

Problem Statement

Imagine at the end of a political conference, republicans and democrats are trying to leave the venue and ordering Uber rides at the same time. However, to make sure no fight breaks out in an Uber ride, the software developers at Uber come up with an algorithm whereby either an Uber ride can have all democrats or republicans or two Democrats and two Republicans. All other combinations can result in a fist-fight.

Your task as the Uber developer is to model the ride requestors as threads. Once an acceptable combination of riders is possible, threads are allowed to proceed to ride. Each thread invokes the method `seated()` when selected by the system for the next ride. When all the threads are seated, any one of the four threads can invoke the method `drive()` to inform the driver to start the ride.

Press+to interact



Uber Seating Problem

▼ Solution

```
import java.util.HashSet;
import java.util.Set;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        UberSeatingProblem.runTest();
    }
}

class UberSeatingProblem {

    private int republicans = 0;
    private int democrats = 0;

    private Semaphore demsWaiting = new Semaphore(0);
    private Semaphore repubsWaiting = new Semaphore(0);

    CyclicBarrier barrier = new CyclicBarrier(4);
    ReentrantLock lock = new ReentrantLock();
```

```

void drive() {
    System.out.println("Uber Ride on Its wayyyy... with ride leader " + Threa
    System.out.flush();
}

```

void seatDemocrat() throws InterruptedException, BrokenBarrierException

```

    boolean rideLeader = false;
    lock.lock();

    democrats++;

    if (democrats == 4) {
        // Seat all the democrats in the Uber ride.
        demsWaiting.release(3);
        democrats -= 4;
        rideLeader = true;
    } else if (democrats == 2 && republicans >= 2) {
        // Seat 2 democrats & 2 republicans
        demsWaiting.release(1);
        repubsWaiting.release(2);
        rideLeader = true;
        democrats -= 2;
        republicans -= 2;
    } else {
        lock.unlock();
        demsWaiting.acquire();
    }

    seated();
    barrier.await();

    if (rideLeader == true) {
        drive();
        lock.unlock();
    }
}

```

```

    }
}

void seated() {
    System.out.println(Thread.currentThread().getName() + " seated");
    System.out.flush();
}

void seatRepublican() throws InterruptedException, BrokenBarrierException

    boolean rideLeader = false;
    lock.lock();

    republicans++;

    if (republicans == 4) {
        // Seat all the republicans in the Uber ride.
        repubsWaiting.release(3);
        rideLeader = true;
        republicans -= 4;
    } else if (republicans == 2 && democrats >= 2) {
        // Seat 2 democrats & 2 republicans
        repubsWaiting.release(1);
        demsWaiting.release(2);
        rideLeader = true;
        republicans -= 2;
        democrats -= 2;
    } else {
        lock.unlock();
        repubsWaiting.acquire();
    }

    seated();
    barrier.await();

    if (rideLeader) {

```

```

        drive();
        lock.unlock();
    }
}

public static void runTest() throws InterruptedException {

    final UberSeatingProblem uberSeatingProblem = new UberSeatingProblem();
    Set<Thread> allThreads = new HashSet<Thread>();

    for (int i = 0; i < 10; i++) {

        Thread thread = new Thread(new Runnable() {
            public void run() {
                try {
                    uberSeatingProblem.seatDemocrat();
                } catch (InterruptedException ie) {
                    System.out.println("We have a problem");
                }

                } catch (BrokenBarrierException bbe) {
                    System.out.println("We have a problem");
                }
            }
        });
        thread.setName("Democrat_" + (i + 1));
        allThreads.add(thread);

        Thread.sleep(50);
    }

    for (int i = 0; i < 14; i++) {
        Thread thread = new Thread(new Runnable() {
            public void run() {
                try {

```

```

        uberSeatingProblem.seatRepublican();
    } catch (InterruptedException ie) {
        System.out.println("We have a problem");

        } catch (BrokenBarrierException bbe) {
            System.out.println("We have a problem");
        }
    }
});
thread.setName("Republican_" + (i + 1));
allThreads.add(thread);
Thread.sleep(20);
}

for (Thread t : allThreads) {
    t.start();
}

for (Thread t : allThreads) {
    t.join();
}
}
}

```

9. Dining Philosophers

This chapter discusses the famous Dijkstra's Dining Philosopher's problem. Two different solutions are explained at length.

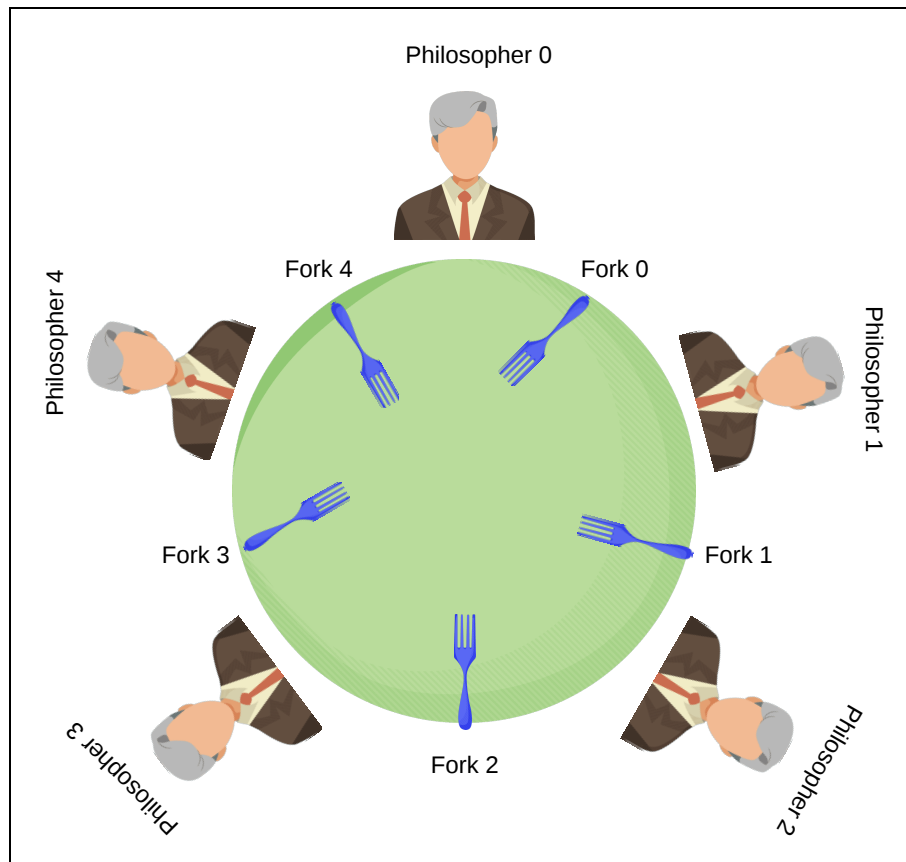
Problem Statement

This is a classical synchronization problem proposed by Dijkstra.

Imagine you have five philosopher's sitting on a roundtable. The philosopher's do only two kinds of activities. One they contemplate, and two they eat. However, they have only five forks between themselves to eat their food with. Each

philosopher requires both the fork to his left and the fork to his right to eat his food. The arrangement of the philosophers and the forks are shown in the diagram. Design a solution where each philosopher gets a chance to eat his food without causing a deadlock.

Press+to interact



▼ Solution

```
import java.util.Random;
import java.util.concurrent.Semaphore;

class Demonstration {

    public static void main(String args[]) throws InterruptedException {
        DiningPhilosophers.runTest();
    }
}

class DiningPhilosophers {

    private static Random random = new Random(System.currentTimeMillis());

    private Semaphore[] forks = new Semaphore[5];
```



```

private Semaphore maxDiners = new Semaphore(4);

public DiningPhilosophers() {
    forks[0] = new Semaphore(1);
    forks[1] = new Semaphore(1);
    forks[2] = new Semaphore(1);
    forks[3] = new Semaphore(1);
    forks[4] = new Semaphore(1);
}

public void lifecycleOfPhilosopher(int id) throws InterruptedException {

    while (true) {
        contemplate();
        eat(id);
    }
}

void contemplate() throws InterruptedException {
    Thread.sleep(random.nextInt(50));
}

void eat(int id) throws InterruptedException {
    maxDiners.acquire();

    forks[id].acquire();
    forks[(id + 1) % 5].acquire();
    System.out.println("Philosopher " + id + " is eating");
    forks[id].release();
    forks[(id + 1) % 5].release();

    maxDiners.release();
}

static void startPhilosopher(DiningPhilosophers dp, int id) {
    try {

```

```

        dp.lifecycleOfPhilosopher(id);
    } catch (InterruptedException ie) {

    }
}

public static void runTest() throws InterruptedException {
    final DiningPhilosophers dp = new DiningPhilosophers();

    Thread p1 = new Thread(new Runnable() {

        public void run() {
            startPhilosoper(dp, 0);
        }
    });

    Thread p2 = new Thread(new Runnable() {

        public void run() {
            startPhilosoper(dp, 1);
        }
    });

    Thread p3 = new Thread(new Runnable() {

        public void run() {
            startPhilosoper(dp, 2);
        }
    });

    Thread p4 = new Thread(new Runnable() {

        public void run() {
            startPhilosoper(dp, 3);
        }
    });
}

```

```

Thread p5 = new Thread(new Runnable() {

    public void run() {
        startPhilosoper(dp, 4);
    }
});

p1.start();
p2.start();
p3.start();
p4.start();
p5.start();

p1.join();
p2.join();
p3.join();
p4.join();
p5.join();
}
}

```

10. Barber Shop

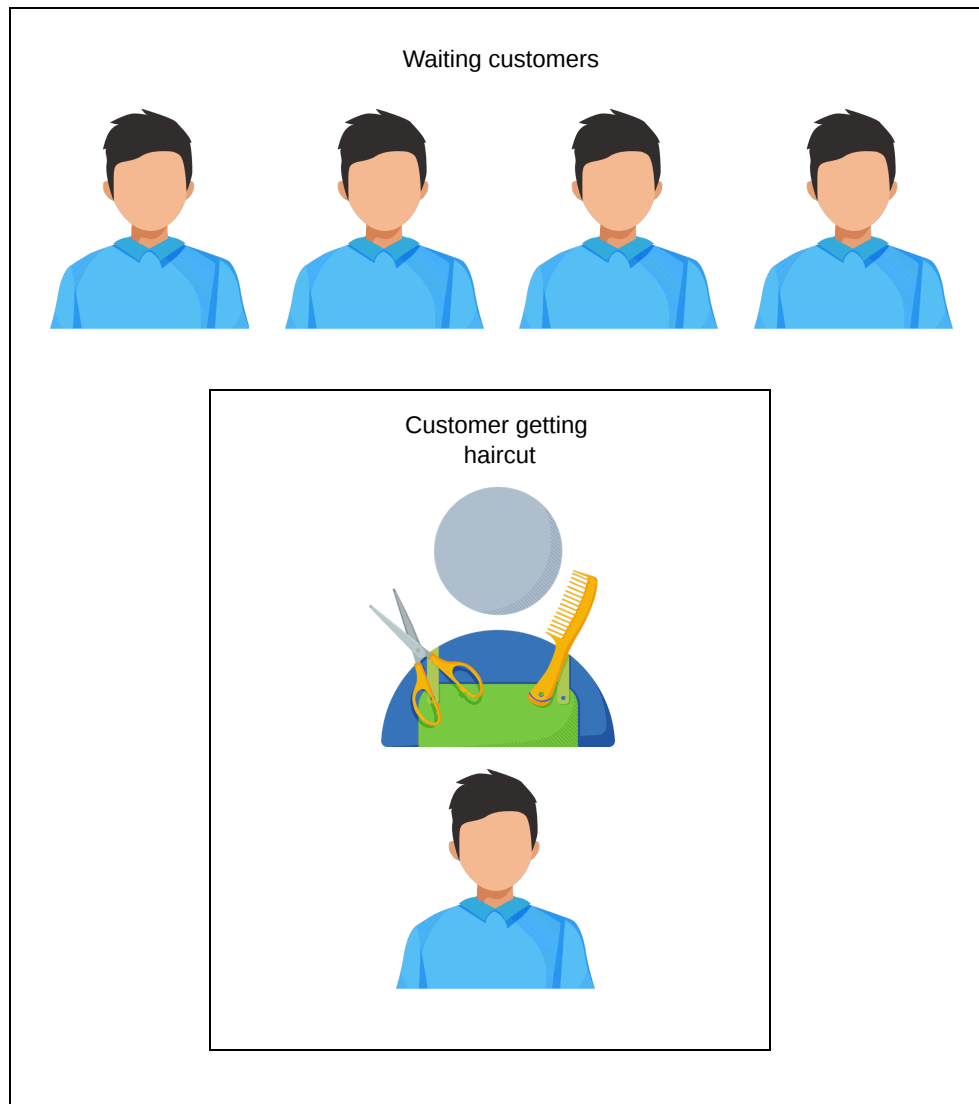
This lesson visits the synchronization issues when programmatically modeling a hypothetical barber shop and how they are solved using Java's concurrency primitives.

Problem Statement

A similar problem appears in Silberschatz and Galvin's OS book, and variations of this problem exist in the wild. A barbershop consists of a waiting room with n chairs, and a barber chair for giving haircuts. If there are no customers to be served, the barber goes to sleep. If a customer enters the barbershop and all chairs are occupied, then the customer leaves the shop. If the barber is busy, but chairs are available, then the customer sits in one of the free chairs. If the barber

is asleep, the customer wakes up the barber. Write a program to coordinate the interaction between the barber and the customers.

Press+to interact



Barber Shop Problem

▼ Solution: Part1

```
import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        BarberShopProblem.runTest();
    }
}
```

```

}

class BarberShopProblem {

    final int CHAIRS = 3;
    Semaphore waitForCustomerToEnter = new Semaphore(0);
    Semaphore waitForBarberToGetReady = new Semaphore(0);
    Semaphore waitForCustomerToLeave = new Semaphore(0);
    Semaphore waitForBarberToCutHair = new Semaphore(0);
    int waitingCustomers = 0;
    ReentrantLock lock = new ReentrantLock();
    int hairCutsGiven = 0;

    void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupied");
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        waitForCustomerToEnter.release();
        waitForBarberToGetReady.acquire();

        waitForBarberToCutHair.acquire();
        waitForCustomerToLeave.release();

        lock.lock();
        waitingCustomers--;
        lock.unlock();
    }

    void barber() throws InterruptedException {

```

```

while (true) {
    waitForCustomerToEnter.acquire();
    waitForBarberToGetReady.release();
    hairCutsGiven++;
    System.out.println("Barber cutting hair..." + hairCutsGiven);
    Thread.sleep(50);
    waitForBarberToCutHair.release();
    waitForCustomerToLeave.acquire();
}
}

public static void runTest() throws InterruptedException {

    HashSet<Thread> set = new HashSet<Thread>();
    final BarberShopProblem barberShopProblem = new BarberShopProblem();

    Thread barberThread = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.barber();
            } catch (InterruptedException ie) {

            }
        }
    });
    barberThread.start();

    for (int i = 0; i < 10; i++) {
        Thread t = new Thread(new Runnable() {
            public void run() {
                try {
                    barberShopProblem.customerWalksIn();
                } catch (InterruptedException ie) {

                }
            }
        });
    }
}

```

```

        }
    });
    set.add(t);
}

for (Thread t : set) {
    t.start();
}

for (Thread t : set) {
    t.join();
}

set.clear();
Thread.sleep(800);

for (int i = 0; i < 5; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.customerWalksIn();
            } catch (InterruptedException ie) {

            }
        }
    });
    set.add(t);
}

for (Thread t : set) {
    t.start();
}

barberThread.join();
}
}

```


▼ Solution: Part2

```
import java.util.HashSet;
import java.util.concurrent.Semaphore;
import java.util.concurrent.locks.ReentrantLock;

class Demonstration {
    public static void main( String args[] ) throws InterruptedException {
        BarberShopProblem.runTest();
    }
}

class BarberShopProblem {

    final int CHAIRS = 3;
    Semaphore waitForCustomerToEnter = new Semaphore(0);
    Semaphore waitForBarberToGetReady = new Semaphore(0);
    Semaphore waitForCustomerToLeave = new Semaphore(0);
    Semaphore waitForBarberToCutHair = new Semaphore(0);
    int waitingCustomers = 0;
    ReentrantLock lock = new ReentrantLock();
    int hairCutsGiven = 0;

    void customerWalksIn() throws InterruptedException {

        lock.lock();
        if (waitingCustomers == CHAIRS) {
            System.out.println("Customer walks out, all chairs occupied");
            lock.unlock();
            return;
        }
        waitingCustomers++;
        lock.unlock();

        waitForCustomerToEnter.release();
        waitForBarberToGetReady.acquire();
    }
}
```

```

        // The chair in the waiting area becomes available
        lock.lock();
        waitingCustomers--;
        lock.unlock();

        waitForBarberToCutHair.acquire();
        waitForCustomerToLeave.release();
    }

    void barber() throws InterruptedException {

        while (true) {
            waitForCustomerToEnter.acquire();
            waitForBarberToGetReady.release();
            hairCutsGiven++;
            System.out.println("Barber cutting hair..." + hairCutsGiven);
            Thread.sleep(50);
            waitForBarberToCutHair.release();
            waitForCustomerToLeave.acquire();
        }
    }

    public static void runTest() throws InterruptedException {

        HashSet<Thread> set = new HashSet<Thread>();
        final BarberShopProblem barberShopProblem = new BarberShopProblem();

        Thread barberThread = new Thread(new Runnable() {
            public void run() {
                try {
                    barberShopProblem.barber();
                } catch (InterruptedException ie) {

                }
            }
        })
    }

```

```

});
barberThread.start();

for (int i = 0; i < 10; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.customerWalksIn();
            } catch (InterruptedException ie) {

            }
        }
    });
    set.add(t);
}

for (Thread t : set) {
    t.start();
}

for (Thread t : set) {
    t.join();
}

set.clear();
Thread.sleep(500);

for (int i = 0; i < 5; i++) {
    Thread t = new Thread(new Runnable() {
        public void run() {
            try {
                barberShopProblem.customerWalksIn();
            } catch (InterruptedException ie) {

            }
        }
    });
}

```

```
    });  
    set.add(t);  
  }  
  for (Thread t : set) {  
    t.start();  
    Thread.sleep(5);  
  }  
  
  barberThread.join();  
}  
}
```

11. Superman Problem

This lesson is about correctly implementing a singleton pattern in Java

Problem Statement

You are designing a library of superheroes for a video game that your fellow developers will consume. Your library should always create a single instance of any of the superheroes and return the same instance to all the requesting consumers.

Say, you start with the class `Superman`. Your task is to make sure that other developers using your class can never instantiate multiple copies of superman. After all, there is only one superman!

Press+to interact



▼ Solution: Part1

```
class Demonstration {  
    public static void main( String args[] ) {  
        Superman superman = Superman.getInstance();  
        superman.fly();  
    }  
}  
  
class Superman {  
    private static Superman superman = new Superman();  
  
    private Superman() {  
    }  
}
```

```

    public static Superman getInstance() {
        return superman;
    }

    public void fly() {
        System.out.println("I am flyyyyyingggggg ...");
    }
}

```

▼ Solution: Part2

```

class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();
    }
}

class Superman {
    private static Superman superman;

    private Superman() {
    }

    public synchronized static Superman getInstance() {

        if (superman == null) {
            superman = new Superman();
        }

        return superman;
    }
}

```

```

    public void fly() {
        System.out.println("I am flyyyyyinggggg ...");
    }
}

```

▼ Solution: Part3

```

class Demonstration {
    public static void main( String args[] ) {
        Superman superman = Superman.getInstance();
        superman.fly();
    }
}

class Superman {

    private Superman() {
    }

    private static class Holder {
        private static final Superman superman = new Superman();
    }

    public static Superman getInstance() {
        return Holder.superman;
    }

    public void fly() {
        System.out.println("I am flyyyyyinggggg ...");
    }
}

```

12. Multithreaded Merge Sort

Learn how to perform Merge sort using threads.

Merge Sort

Merge sort is a typical text-book example of a recursive algorithm and the poster-child of the divide and conquer strategy. The idea is very simple, we divide the array into two equal parts, sort them recursively and then combine the two sorted arrays. The base case for recursion occurs when the size of the array reaches a single element. An array consisting of a single element is already sorted.

The running time for a recursive solution is expressed as a *recurrence equation*. An equation or inequality that describes a function in terms of its own value on smaller inputs is called a recurrence equation. The running time for a recursive algorithm is the solution to the recurrence equation. The recurrence equation for recursive algorithms usually takes on the following form:

Running Time = Cost to divide into n subproblems + n * Cost to solve each of the n problems + Cost to merge all n problems

In the case of merge sort, we divide the given array into two arrays of equal size, i.e. we divide the original problem into sub-problems to be solved recursively.

▼ Solution

```
import java.util.Random;

class Demonstration {

    private static int SIZE = 25;
    private static Random random = new Random(System.currentTimeMillis());
    private static int[] input = new int[SIZE];

    static private void createTestData() {
        for (int i = 0; i < SIZE; i++) {
            input[i] = random.nextInt(10000);
        }
    }

    static private void printArray(int[] input) {
        System.out.println();
        for (int i = 0; i < input.length; i++)
```



```

        System.out.print(" " + input[i] + " ");
        System.out.println();
    }

    public static void main( String args[] ) {
        createTestData();

        System.out.println("Unsorted Array");
        printArray(input);
        long start = System.currentTimeMillis();
        (new MultiThreadedMergeSort()).mergeSort(0, input.length - 1, input);
        long end = System.currentTimeMillis();
        System.out.println("\n\nTime taken to sort = " + (end - start) + " millisec");
        System.out.println("Sorted Array");
        printArray(input);
    }
}

class MultiThreadedMergeSort {

    private static int SIZE = 25;
    private int[] input = new int[SIZE];
    private int[] scratch = new int[SIZE];

    void mergeSort(final int start, final int end, final int[] input) {

        if (start == end) {
            return;
        }

        final int mid = start + ((end - start) / 2);

        // sort first half
        Thread worker1 = new Thread(new Runnable() {

            public void run() {

```

```

        mergeSort(start, mid, input);
    }
});

// sort second half
Thread worker2 = new Thread(new Runnable() {

    public void run() {
        mergeSort(mid + 1, end, input);
    }
});

// start the threads
worker1.start();
worker2.start();

try {

    worker1.join();
    worker2.join();
} catch (InterruptedException ie) {
    // swallow
}

// merge the two sorted arrays
int i = start;
int j = mid + 1;
int k;

for (k = start; k <= end; k++) {
    scratch[k] = input[k];
}

k = start;
while (k <= end) {

```

```

    if (i <= mid && j <= end) {
        input[k] = Math.min(scratch[i], scratch[j]);

        if (input[k] == scratch[i]) {
            i++;
        } else {
            j++;
        }
    } else if (i <= mid && j > end) {
        input[k] = scratch[i];
        i++;
    } else {
        input[k] = scratch[j];
        j++;
    }
    k++;
}
}
}

```

13. Asynchronous to Synchronous Problem

A real-life interview question asking to convert asynchronous execution to synchronous execution.

Problem Statement

This is an actual interview question asked at Netflix.

Imagine we have an `Executor` class that performs some useful task asynchronously via the method `asynchronousExecution()`. In addition the method accepts a callback object which implements the `Callback` interface. the object's `done()` gets invoked when the asynchronous execution is done. The definition for the involved classes is below:

Executor Class

```
public class Executor {  
  
    public void asynchronousExecution(Callback callback) throws Exception {  
  
        Thread t = new Thread(() -> {  
            // Do some useful work  
            try {  
                // Simulate useful work by sleeping for 5 seconds  
                Thread.sleep(5000);  
            } catch (InterruptedException ie) {  
            }  
            callback.done();  
        });  
        t.start();  
    }  
}
```

Callback Interface

```
public interface Callback {  
  
    public void done();  
}
```

An example run would be as follows:

```
class Demonstration {  
    public static void main( String args[] ) throws Exception{  
        Executor executor = new Executor();  
        executor.asynchronousExecution(() -> {  
            System.out.println("I am done");  
        });  
  
        System.out.println("main thread exiting...");  
    }  
}  
  
interface Callback {  
  
    public void done();  
}
```

```

}

class Executor {

    public void asynchronousExecution(Callback callback) throws Exception {

        Thread t = new Thread(() → {
            // Do some useful work
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
            }
            callback.done();
        });
        t.start();
    }
}

```

Note how the main thread exits before the asynchronous execution is completed.

Your task is to make the execution synchronous without changing the original classes (imagine, you are given the binaries and not the source code) so that main thread waits till asynchronous execution is complete. In other words, the highlighted **line#8** only executes once the asynchronous task is complete.

14. Nonblocking Stack

Problem

Design a stack that doesn't use locks or `synchronized` and is thread-safe. You may assume that you are provided with an application-level API that mocks the hardware instruction compare-and-swap, to atomically compare and swap values at a memory location.

▼ Solution1 : solution without locking.

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {

        SynchronizedStack<Integer> stackOfInts = new SynchronizedStack<>()
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        int numThreads = 2;
        CyclicBarrier barrier = new CyclicBarrier(numThreads);

        Integer testValue = new Integer(51);

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 10000; i++) {
                            stackOfInts.push(testValue);
                        }

                        try {
                            barrier.await();
                        } catch (InterruptedException | BrokenBarrierException ex) {
                            System.out.println("ignoring exception");
                            //ignore both exceptions
                        }

                        for (int i = 0; i < 10000; i++) {
                            stackOfInts.pop();
                        }
                    }
                });
            }
        }
    }
}

```

```

    } finally {
        executorService.shutdown();
        executorService.awaitTermination(1, TimeUnit.HOURS);
    }
}
}
}

```

▼ Solution2 : **CAS-based solution#**

```

import java.util.concurrent.*;

class Demonstration {

    public static void main( String args[] ) throws Exception {

        ExecutorService executorService = Executors.newFixedThreadPool(2);
        int numThreads = 7;
        // create an instance of simulated CAS instruction, that holds an integer
        // simultaneously manipulated by 7 threads.
        final SimulatedCompareAndSwap<Integer> sharedInteger = new SimulatedCompareAndSwap<Integer>();

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {

                        int expectedValue;
                        int newValue;
                        int updateAttempt = 0;

                        do {
                            expectedValue = sharedInteger.getValue();
                            newValue = expectedValue + 1;
                            System.out.println(Thread.currentThread().getName() + "

```

```

        updateAttempt++;
    } while (!sharedInteger.compareAndSet(expectedValue, new
    }
    });
}
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

    System.out.println("Final integer value = " + sharedInteger.getValue())
}
}

```

▼ Solution: 3

```

import java.util.concurrent.*;

class Demonstration {
    public static void main( String args[] ) throws Exception {

        NonblockingStack<Integer> stack = new NonblockingStack<>();
        ExecutorService executorService = Executors.newFixedThreadPool(20);
        int numThreads = 2;
        CyclicBarrier barrier = new CyclicBarrier(numThreads);

        long start = System.currentTimeMillis();
        Integer testValue = new Integer(51);

        try {
            for (int i = 0; i < numThreads; i++) {
                executorService.submit(new Runnable() {
                    @Override
                    public void run() {
                        for (int i = 0; i < 10000; i++) {

```



```

        stack.push(testValue);
    }

    try {
        barrier.await();
    } catch (InterruptedException | BrokenBarrierException ex) {
        System.out.println("ignoring exception");
        //ignore both exceptions
    }

    for (int i = 0; i < 10000; i++) {
        stack.pop();
    }
    });
}
} finally {
    executorService.shutdown();
    executorService.awaitTermination(1, TimeUnit.HOURS);
}

System.out.println("Number of elements in the stack = " + stack.size());
}
}

```

Bonus Question

1. Ordered Printing

This problem is about imposing an order on thread execution.

Problem Statement

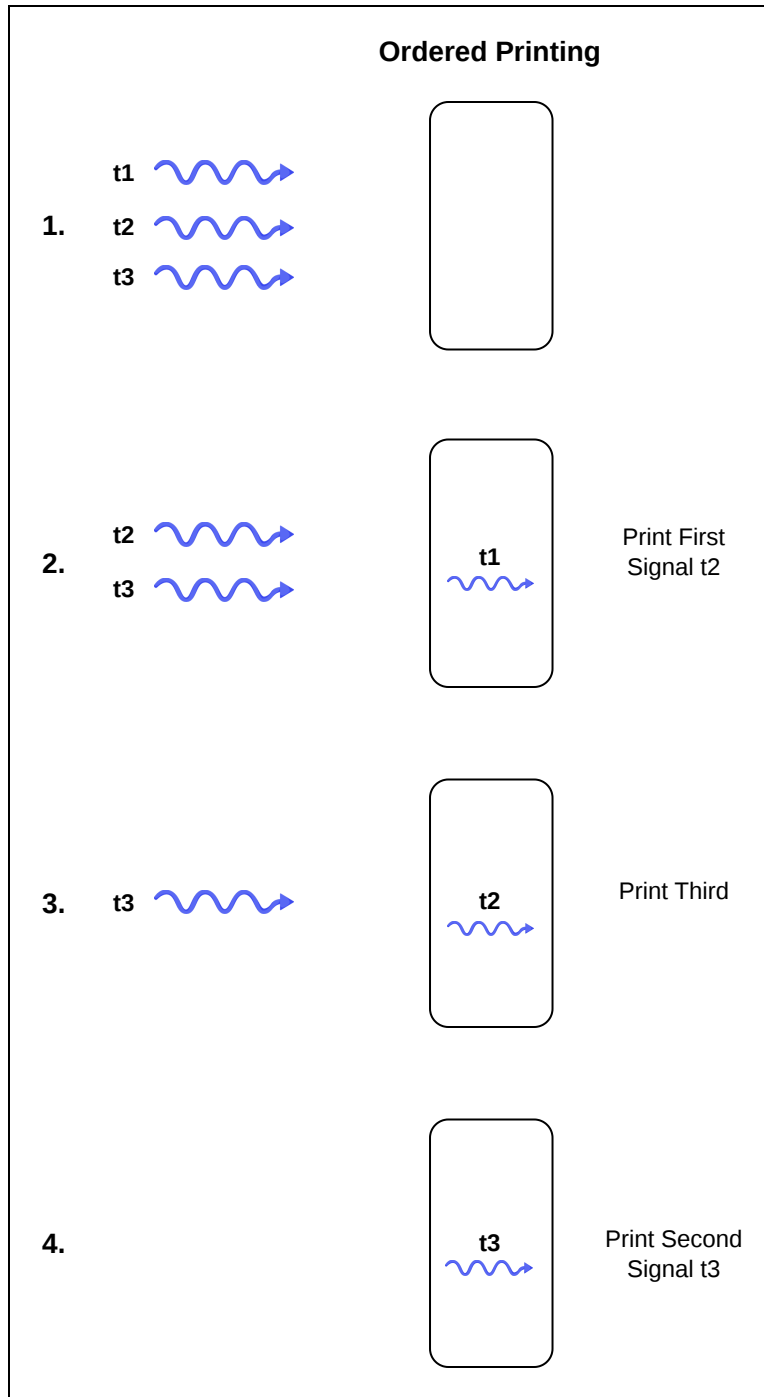
Suppose there are three threads t1, t2 and t3. t1 prints **First**, t2 prints **Second** and t3 prints **Third**. The code for the class is as follows:

```
public class OrderedPrinting {  
  
    public void printFirst() {  
  
        System.out.print("First");  
  
    }  
  
    public void printSecond() {  
  
        System.out.print("Second");  
  
    }  
  
    public void printThird() {  
  
        System.out.print("Third");  
  
    }  
  
}
```

Thread t1 calls printFirst(), thread t2 calls printSecond(), and thread t3 calls printThird(). The threads can run in any order. You have to synchronize the threads so that the functions **printFirst()**, **printSecond()** and **printThird()** are executed in order.

The workflow of the program is shown below:

Press+to interact



▼ Solution

```
class OrderedPrinting {  
  
    int count;
```

```

public OrderedPrinting() {
    count = 1;
}

public void printFirst() throws InterruptedException {

    synchronized(this){
        System.out.println("First");
        count++;
        this.notifyAll();
    }
}

public void printSecond() throws InterruptedException {

    synchronized(this){
        while(count != 2){
            this.wait();
        }
        System.out.println("Second");
        count++;
        this.notifyAll();
    }

}

public void printThird() throws InterruptedException {

    synchronized(this){
        while(count != 3){
            this.wait();
        }
        System.out.println("Third");
    }
}

```

```

    }
}

class OrderedPrintingThread extends Thread
{
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
    {
        this.method = method;
        this.obj = obj;
    }

    public void run()
    {
        //for printing "First"
        if ("first".equals(method))
        {
            try
            {
                obj.printFirst();
            }
            catch (InterruptedException e)
            {
            }
        }
        //for printing "Second"
        else if ("second".equals(method))
        {
            try
            {
                obj.printSecond();
            }
            catch (InterruptedException e)

```

```

        {

        }
    }
    //for printing "Third"
    else if ("third".equals(method))
    {
        try
        {
            obj.printThird();
        }
        catch (InterruptedException e)
        {

        }
    }
}

public class Main
{
    public static void main(String[] args)
    {
        OrderedPrinting obj = new OrderedPrinting();

        OrderedPrintingThread t1 = new OrderedPrintingThread(obj, "first");
        OrderedPrintingThread t2 = new OrderedPrintingThread(obj, "second");
        OrderedPrintingThread t3 = new OrderedPrintingThread(obj, "third");

        t2.start();
        t3.start();
        t1.start();

    }
}

```

▼ Solution 2 : **Solution using CountdownLatch#**

```
import java.util.concurrent.CountDownLatch;

class OrderedPrinting
{
    CountdownLatch latch1;
    CountdownLatch latch2;

    public OrderedPrinting()
    {
        latch1 = new CountdownLatch(1);
        latch2 = new CountdownLatch(1);
    }

    public void printFirst() throws InterruptedException
    {
        System.out.println("First");
        latch1.countDown();
    }

    public void printSecond() throws InterruptedException
    {
        latch1.await();
        System.out.println("Second");
        latch2.countDown();
    }

    public void printThird() throws InterruptedException
    {
        latch2.await();
        System.out.println("Third");
    }
}

class OrderedPrintingThread extends Thread
```



```

{
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method)
    {
        this.method = method;
        this.obj = obj;
    }

    public void run()
    {
        if ("first".equals(method))
        {
            try
            {
                obj.printFirst();
            }
            catch (InterruptedException e)
            {
            }
        }
        else if ("second".equals(method))
        {
            try
            {
                obj.printSecond();
            }
            catch (InterruptedException e)
            {
            }
        }
        else if ("third".equals(method))
        {

```

```

        try
        {
            obj.printThird();
        }
        catch (InterruptedException e)
        {

        }
    }
}

public class Main
{
    public static void main(String[] args)
    {
        OrderedPrinting obj = new OrderedPrinting();

        OrderedPrintingThread t1 = new OrderedPrintingThread(obj, "first");
        OrderedPrintingThread t2 = new OrderedPrintingThread(obj, "second");
        OrderedPrintingThread t3 = new OrderedPrintingThread(obj, "third");

        t3.start();
        t2.start();
        t1.start();

    }
}

```

2. Printing Foo Bar n Times

Learn how to execute threads in a specific order for a user specified number of iterations.

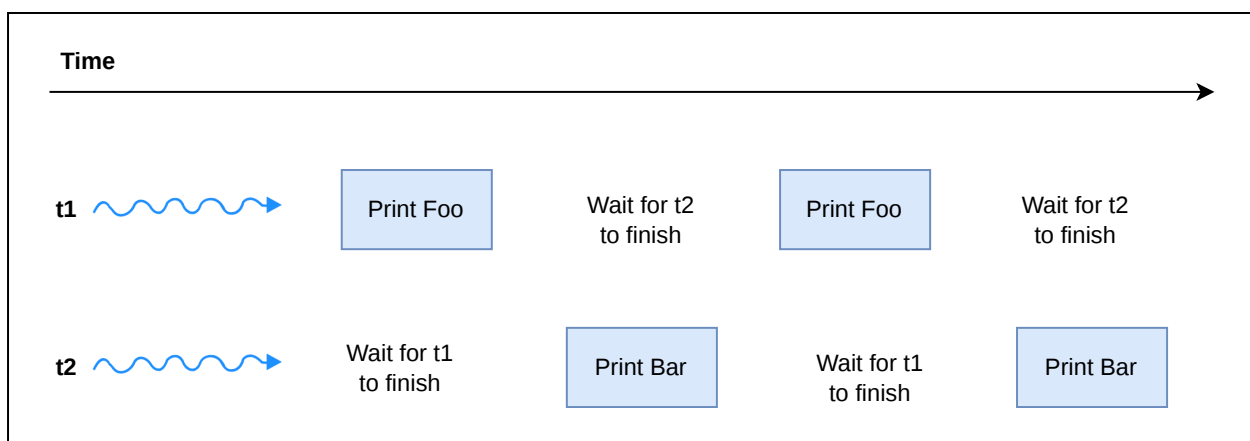
Problem Statement

Suppose there are two threads t1 and t2. t1 prints **Foo** and t2 prints **Bar**. You are required to write a program which takes a user input n. Then the two threads print Foo and Bar alternately n number of times. The code for the class is as follows:

```
class PrintFooBar {  
  
    public void PrintFoo() {  
  
        for (int i = 1; i <= n; i++){  
  
            System.out.print("Foo");  
  
        }  
  
    }  
  
    public void PrintBar() {  
  
        for (int i = 1; i <= n; i++) {  
  
            System.out.print("Bar");  
  
        }  
  
    }  
  
}
```

The two threads will run sequentially. You have to synchronize the two threads so that the functions `PrintFoo()` and `PrintBar()` are executed in an order. The workflow is shown below:

Press+to interact



▼ Solution

```
class FooBar {

    private int n;
    private int flag = 0;

    public FooBar(int n) {
        this.n = n;
    }

    public void foo() {

        for (int i = 1; i <= n; i++) {
            synchronized(this) {
```

```

        while (flag == 1) {
            try {
                this.wait();
            }
            catch (Exception e) {

            }
        }
        System.out.print("Foo");
        flag = 1;
        this.notifyAll();
    }
}

public void bar() {

    for (int i = 1; i <= n; i++) {
        synchronized(this) {
            while (flag == 0) {
                try {
                    this.wait();
                }
                catch (Exception e) {

                }
            }
            System.out.println("Bar");
            flag = 0;
            this.notifyAll();
        }
    }
}
}

```

```

class FooBarThread extends Thread {

    FooBar fooBar;
    String method;

    public FooBarThread(FooBar fooBar, String method){
        this.fooBar = fooBar;
        this.method = method;
    }

    public void run() {
        if ("foo".equals(method)) {
            fooBar.foo();
        }
        else if ("bar".equals(method)) {
            fooBar.bar();
        }
    }
}

public class Main {

    public static void main(String[] args) throws InterruptedException {

        FooBar fooBar = new FooBar(3);

        Thread t1 = new FooBarThread(fooBar,"foo");
        Thread t2 = new FooBarThread(fooBar,"bar");

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

    }
}

```

3. Printing Number Series (Zero, Even, Odd)

This problem is about repeatedly executing threads which print a specific type of number. Another variation of this problem; print even and odd numbers; utilizes two threads instead of three.

Problem Statement

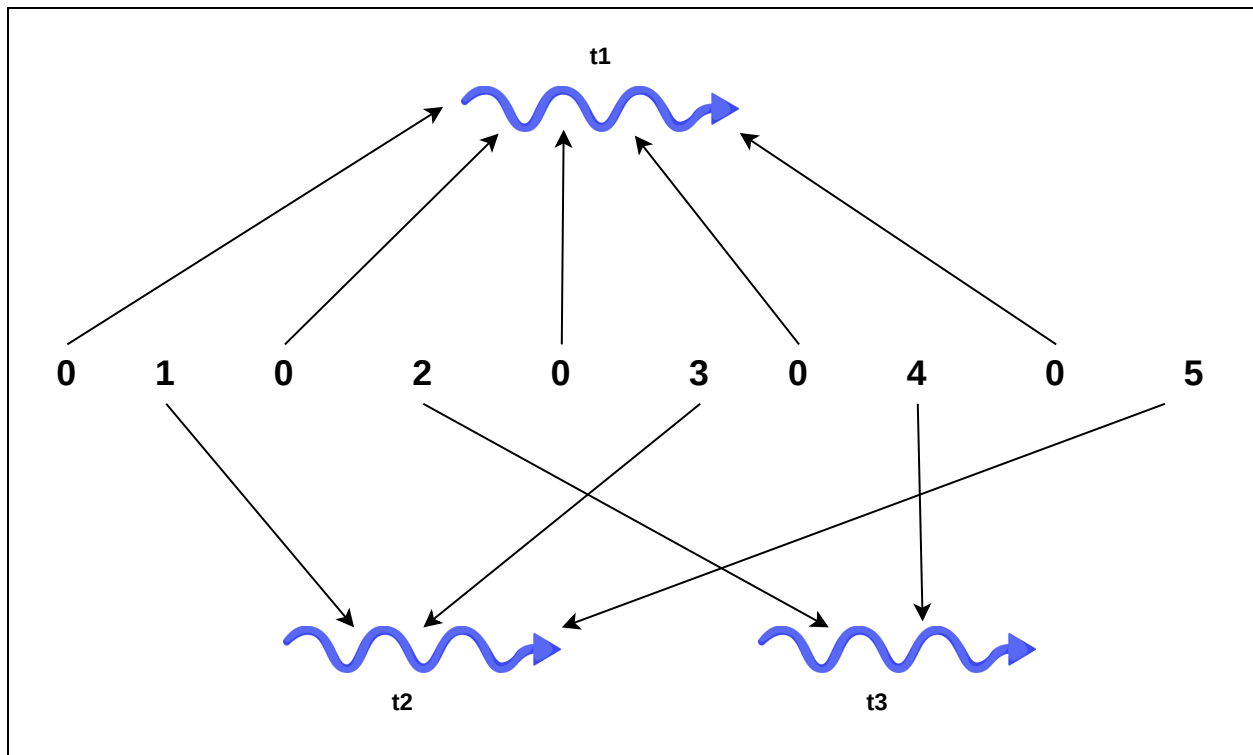
Suppose we are given a number `_n` based on which a program creates the series `010203...0n`. There are three threads `t1`, `t2` and `t3` which print a specific type of number from the series. `t1` only prints zeros, `t2` prints odd numbers and `t3` prints even numbers from the series. The code for the class is given as follows:

```
class PrintNumberSeries {  
  
    public PrintNumberSeries(int n) {  
  
        this.n = n;  
  
    }  
  
    public void PrintZero() {  
  
    }  
  
    public void PrintOdd() {  
  
    }  
  
    public void PrintEven() {  
  
    }  
  
}
```

You are required to write a program which takes a user input n and outputs the number series using three threads. The three threads work together to print zero, even and odd numbers. The threads should be synchronized so that the functions `PrintZero()`, `PrintOdd()` and `PrintEven()` are executed in an order.

The workflow of the program is shown below:

Press+to interact



▼ Solution

```
import java.util.concurrent.*;

class PrintNumberSeries {
    private int n;
    private Semaphore zeroSem, oddSem, evenSem;

    public PrintNumberSeries(int n) {
        this.n = n;
        zeroSem = new Semaphore(1);
        oddSem = new Semaphore(0);
        evenSem = new Semaphore(0);
    }

    public void PrintZero() {
        for (int i = 0; i < n; ++i) {
            try {
                zeroSem.acquire();
            }
        }
    }
}
```

```

    }
    catch (Exception e) {
    }
    System.out.print("0");
    // release oddSem if i is even or else release evenSem if i is odd
    (i % 2 == 0 ? oddSem : evenSem).release();
}
}

public void PrintEven() {
    for (int i = 2; i <= n; i += 2) {
        try {
            evenSem.acquire();
        }
        catch (Exception e) {
        }
        System.out.print(i);
        zeroSem.release();
    }
}

public void PrintOdd() {
    for (int i = 1; i <= n; i += 2) {
        try {
            oddSem.acquire();
        }
        catch (Exception e) {
        }
        System.out.print(i);
        zeroSem.release();
    }
}
}

class PrintNumberSeriesThread extends Thread {

```

```

PrintNumberSeries zeo;
String method;

public PrintNumberSeriesThread(PrintNumberSeries zeo, String method){
    this.zeo = zeo;
    this.method = method;
}

public void run() {
    if ("zero".equals(method)) {
        try {
            zeo.PrintZero();
        }
        catch (Exception e) {
        }
    }
    else if ("even".equals(method)) {
        try {
            zeo.PrintEven();
        }
        catch (Exception e) {
        }
    }
    else if ("odd".equals(method)) {
        try {
            zeo.PrintOdd();
        }
        catch (Exception e) {
        }
    }
}

}

public class Main {

    public static void main(String[] args) {

```

```
PrintNumberSeries zeo = new PrintNumberSeries(100);

Thread t1 = new PrintNumberSeriesThread(zeo,"zero");
Thread t2 = new PrintNumberSeriesThread(zeo,"even");
Thread t3 = new PrintNumberSeriesThread(zeo,"odd");

t2.start();
t1.start();
t3.start();

}
}
```

4. Build a Molecule

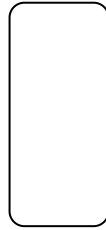
This problem simulated the creation of water molecule by grouping three threads representing Hydrogen and Oxygen atoms.

Problem Statement

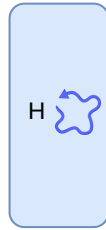
Suppose we have a machine that creates molecules by combining atoms. We are creating water molecules by joining one Oxygen and two Hydrogen atoms. The atoms are represented by threads. The machine will wait for the required atoms (threads), then group one Oxygen and two Hydrogen threads to simulate the creation of a molecule. The molecule then exists the machine. You have to ensure that one molecule is completed before moving onto the next molecule. If more than the required number of threads arrive, they will have to wait. The figure below explains the working of our machine:

Press+to interact

1. H 





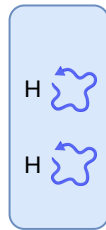
2. H 



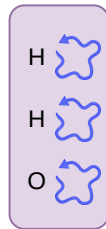
3. H 



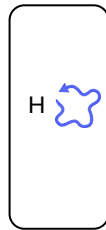
4. H 
O 



5. H 



6.



H 

H 

O 

Two Hydrogen threads are admitted in the machine as they arrive but when the third thread arrives in step 3, it is made to wait. When an Oxygen thread arrives in step 4, it is allowed to enter the machine. A water molecule is formed in step 5 which exists the machine in step 6. That is when the waiting Hydrogen thread is notified and the process of creating more molecules continues. The threads can arrive in any order which means that HHO, OHH and HOH are all valid outputs.

The code for the class is as follows:

```
class H2OMachine {  
  
    public H2OMachine() {  
  
    }  
  
    public void HydrogenAtom() {  
  
    }  
  
    public void OxygenAtom() {  
  
    }  
  
}
```

The input to the machine can be in any order. Your program should enforce a 2:1 ratio for Hydrogen and Oxygen threads, and stop more than the required number of threads from entering the machine.

▼ Solution

```
import java.util.Arrays;  
import java.util.Collections;  
  
class H2OMachine {
```

```

Object sync;
String[] molecule;
int count;

public H2OMachine() {
    molecule = new String[3];
    count = 0;
    sync = new Object();
}

public void HydrogenAtom() {
    synchronized (sync) {

        // if 2 hydrogen atoms already exist
        while (Collections.frequency(Arrays.asList(molecule),"H") == 2) {
            try {
                sync.wait();
            }
            catch (Exception e) {
            }
        }

        molecule[count] = "H";
        count++;

        // if molecule is complete, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}

```



```

public void OxygenAtom() throws InterruptedException {
    synchronized (sync) {

        // if 1 oxygen atom already exists
        while (Collections.frequency(Arrays.asList(molecule),"O") == 1) {
            try {
                sync.wait();
            }
            catch (Exception e) {
            }
        }

        molecule[count] = "O";
        count++;

        // if molecule is complete, then exit.
        if(count == 3) {
            for (String element: molecule) {
                System.out.print(element);
            }
            Arrays.fill(molecule,null);
            count = 0;
        }
        sync.notifyAll();
    }
}

class H2OMachineThread extends Thread {

    H2OMachine molecule;
    String atom;

    public H2OMachineThread(H2OMachine molecule, String atom){
        this.molecule = molecule;
        this.atom = atom;
    }
}

```

```

    }

    public void run() {
        if ("H".equals(atom)) {
            try {
                molecule.HydrogenAtom();
            }
            catch (Exception e) {
            }
        }
        else if ("O".equals(atom)) {
            try {
                molecule.OxygenAtom();
            }
            catch (Exception e) {
            }
        }
    }
}

public class Main
{
    public static void main(String[] args) {

        H2OMachine molecule = new H2OMachine();

        Thread t1 = new H2OMachineThread(molecule,"H");
        Thread t2 = new H2OMachineThread(molecule,"O");
        Thread t3 = new H2OMachineThread(molecule,"H");
        Thread t4 = new H2OMachineThread(molecule,"O");
        Thread t5 = new H2OMachineThread(molecule,"H");
        Thread t6 = new H2OMachineThread(molecule,"H");

        t2.start();
        t1.start();
        t4.start();
    }
}

```

```
t3.start();  
t5.start();  
t6.start();  
}  
}
```

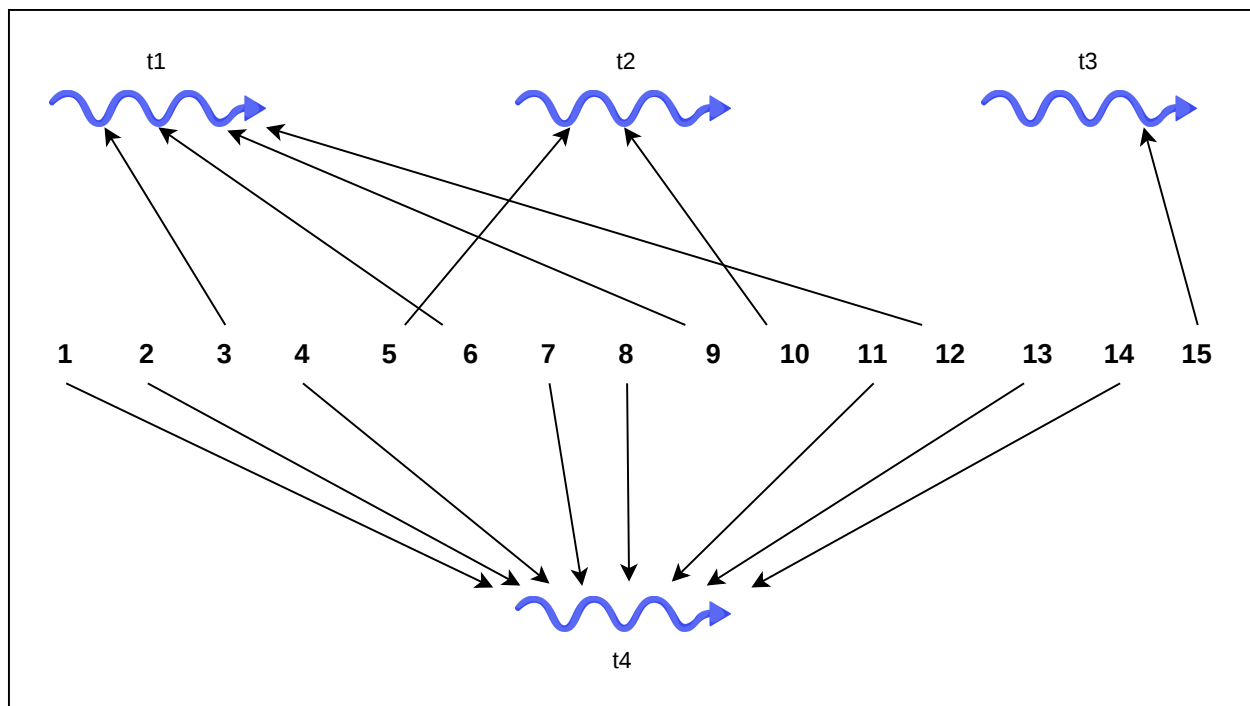
5. Fizz Buzz Problem

This problem explores a multi-threaded solution to the very common Fizz Buzz programming task

Problem Statement

FizzBuzz is a common interview problem in which a program prints a number series from 1 to n such that for every number that is a multiple of 3 it prints "fizz", for every number that is a multiple of 5 it prints "buzz" and for every number that is a multiple of both 3 and 5 it prints "fizzbuzz". We will be creating a multi-threaded solution for this problem. Suppose we have four threads t1, t2, t3 and t4. Thread t1 checks if the number is divisible by 3 and prints **fizz**. Thread t2 checks if the number is divisible by 5 and prints **buzz**. Thread t3 checks if the number is divisible by both 3 and 5 and prints **fizzbuzz**. Thread t4 prints numbers that are not divisible by 3 or 5. The workflow of the program is shown below:

Press+to interact



The code for the class is as follows:

```
class MultithreadedFizzBuzz {
```

```
private int n;

public MultithreadedFizzBuzz(int n) {

    this.n = n;

}

public void fizz() {

    System.out.print("fizz");

}

public void buzz() {

    System.out.print("buzz");

}

public void fizzbuzz() {

    System.out.print("fizzbuzz");

}

public void number(int num) {

    System.out.print(num);

}

}
```

For an input integer n , the program should output a string containing the words fizz, buzz and fizzbuzz representing certain numbers. For example, for $n = 15$, the output should be: 1, 2, fizz, 4, buzz, fizz, 7, 8, fizz, buzz, 11, fizz, 13, 14, fizzbuzz.

▼ Solution

```
class MultithreadedFizzBuzz {
    private int n;
    private int num = 1;

    public MultithreadedFizzBuzz(int n) {
        this.n = n;
    }

    public synchronized void fizz() throws InterruptedException {
        while (num <= n) {
            if (num % 3 == 0 && num % 5 != 0) {
                System.out.println("Fizz");
                num++;
                notifyAll();
            } else {
                wait();
            }
        }
    }

    public synchronized void buzz() throws InterruptedException {
        while (num <= n) {
            if (num % 3 != 0 && num % 5 == 0) {
                System.out.println("Buzz");
                num++;
                notifyAll();
            } else {
                wait();
            }
        }
    }
}
```

```

public synchronized void fizzbuzz() throws InterruptedException {
    while (num <= n) {
        if (num % 15 == 0) {
            System.out.println("FizzBuzz");
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}

public synchronized void number() throws InterruptedException {
    while (num <= n) {
        if (num % 3 != 0 && num % 5 != 0) {
            System.out.println(num);
            num++;
            notifyAll();
        } else {
            wait();
        }
    }
}

class FizzBuzzThread extends Thread {

    MultithreadedFizzBuzz obj;
    String method;

    public FizzBuzzThread(MultithreadedFizzBuzz obj, String method){
        this.obj = obj;
        this.method = method;
    }
}

```

```

public void run() {
    if ("Fizz".equals(method)) {
        try {
            obj.fizz();
        }
        catch (Exception e) {
        }
    }
    else if ("Buzz".equals(method)) {
        try {
            obj.buzz();
        }
        catch (Exception e) {
        }
    }
    else if ("FizzBuzz".equals(method)) {
        try {
            obj.fizzbuzz();
        }
        catch (Exception e) {
        }
    }
    else if ("Number".equals(method)) {
        try {
            obj.number();
        }
        catch (Exception e) {
        }
    }
}

}

class main
{
    public static void main(String[] args) {

```



```
MultithreadedFizzBuzz obj = new MultithreadedFizzBuzz(18);

Thread t1 = new FizzBuzzThread(obj,"Fizz");
Thread t2 = new FizzBuzzThread(obj,"Buzz");
Thread t3 = new FizzBuzzThread(obj,"FizzBuzz");
Thread t4 = new FizzBuzzThread(obj,"Number");

t1.start();
t4.start();
t3.start();
    t2.start();
}
}
```