Java

Креирање и контрола рада нити 1

Први начин је наслеђивањем класе Thread:

```
import java.lang.*;
public class MyThread extends Thread {
    public void run() {
        ....
    }
}
```

Нит се инстанцира са:

```
myThread = new MyThread(...);
```

Нит се покреће са :

```
myThread.start();
```

Креирање и контрола рада нити 2

Други начин за креирање нити је имплементирањем интерфејса Runnable:

```
import java.lang.*;
public class MyThread implements Runnable {
    public void run() {
      ....
   }
}
```

Нит се инстанцира са:

```
Thread myThread = new Thread(new MyThread(...));
```

Нит се покреће са:

```
myThread.start();
```

Креирање и контрола рада нити 3

Чекање да се извршавање нити заврши:

```
myThread.join();
```

Препуштање права на извршавање другој нити

```
myThread.yield();
```

Суспендовање извршавања нити на временски период t:

```
myThread.sleep(t);
```

Методе у класи Thread

Modifier and Type	Method and Description
`void`	`run()` If this thread was constructed using a separate `Runnable` run object, then that `Runnable` object's `run` method is called; otherwise, this method does nothing and returns.
`void`	`start()` Causes this thread to begin execution; the Java Virtual Machine calls the `run` method of this thread.
`static void`	`yield()` A hint to the scheduler that the current thread is willing to yield its current use of a processor.
`static void`	`sleep(long millis)` Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
`void`	`join()` Waits for this thread to die.
`void`	`join(long millis)` Waits at most `millis` milliseconds for this thread to die.
`static Thread`	`currentThread()` Returns a reference to the currently executing thread object.
`void`	`interrupt()` Interrupts this thread.
`static boolean`	`interrupted()` Tests whether the current thread has been interrupted.

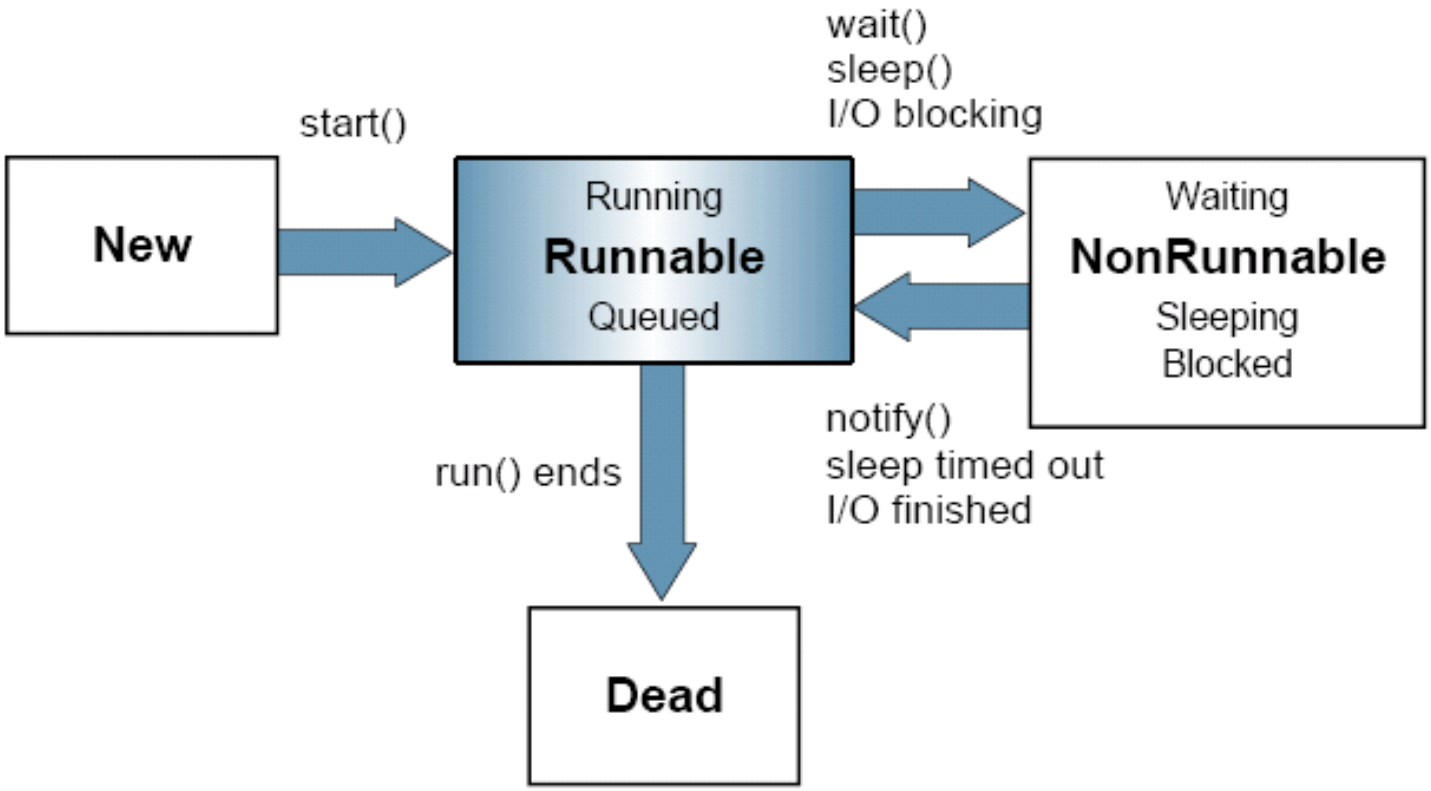
Методе у класи Thread

Modifier and Type	Method and Description
`long`	`getId()` Returns the identifier of this Thread.
`String`	`getName()` Returns this thread's name.
`int`	`getPriority()` Returns this thread's priority.
`ThreadGroup`	`getThreadGroup()` Returns the thread group to which this thread belongs.
`Thread.State`	`getState()` Returns the state of this thread.
`static boolean`	`holdsLock(Object obj)` Returns `true` if and only if the current thread holds the monitor lock on the specified object.
`boolean`	`isAlive()` Tests if this thread is alive.
`boolean`	`isDaemon()` Tests if this thread is a daemon thread.

Методе у класи Thread

Modifier and Type	Method and Description
void	`stop()` **Deprecated.** *This method is inherently unsafe. Stopping a thread with Thread.stop causes it to unlock all of the monitors that it has locked (as a natural consequence of the unchecked `ThreadDeath` exception propagating up the stack). If any of the objects previously protected by these monitors were in an inconsistent state, the damaged objects become visible to other threads, potentially resulting in arbitrary behavior. Many uses of `stop` should be replaced by code that simply modifies some variable to indicate that the target thread should stop running. The target thread should check this variable regularly, and return from its run method in an orderly fashion if the variable indicates that it is to stop running. If the target thread waits for long periods (on a condition variable, for example), the `interrupt` method should be used to interrupt the wait.*
void	`suspend()` **Deprecated.** *This method has been deprecated, as it is inherently deadlock-prone. If the target thread holds a lock on the monitor protecting a critical system resource when it is suspended, no thread can access this resource until the target thread is resumed. If the thread that would resume the target thread attempts to lock this monitor prior to calling `resume`, deadlock results. Such deadlocks typically manifest themselves as "frozen" processes.*
void	`resume()` **Deprecated.** *This method exists solely for use with `suspend()`, which has been deprecated because it is deadlock-prone.*
void	`destroy()` **Deprecated.** *This method was originally designed to destroy this thread without any cleanup. Any monitors it held would have remained locked. However, the method was never implemented. If if were to be implemented, it would be deadlock-prone in much the manner of `suspend()`. If the target thread held a lock protecting a critical system resource when it was destroyed, no thread could ever access this resource again. If another thread ever attempted to lock this resource, deadlock would result. Such deadlocks typically manifest themselves as "frozen" processes.*

Животни циклус нити



Синхронизација 1

Део кода или метод се могу декларисати као критична секција коришћењем речи synchronized.

```
synchronized (someObject) { ... }
```

део кода је критична секција, а објекат одговара условном критичном региону

```
public class SomeClass          {
        public synchronized void f(...) {
                ...
        }
}
```

део кода је критична секција, а објекат одговара мониторима са signal and continue дисциплином

Синхронизација 2

Наредба wait прекида извршавање нити и ставља га у стање чекања; привремено се скида забрана другим нитима да приступају монитору. wait();

wait(tmilisec);

wait(tmilisec,tnanosec);

wait() чека неограничено,.

Наредба notifyAll() шаље нотификацију свим нитима које чекају да приступе циљном објекту да то могу учинити.

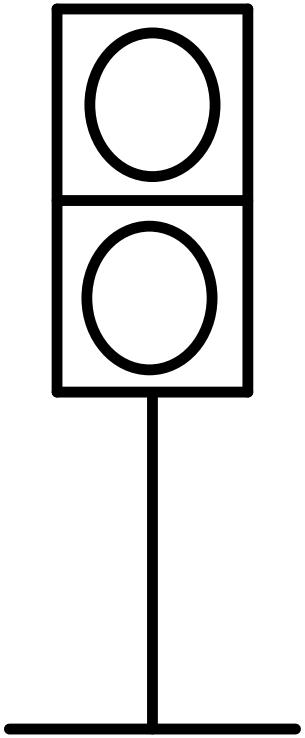
Наредба notify() ради исто што и notifyAll(), али нотификује само једну нит.

Reentrant monitor

```
public class Reentrant {

    public synchronized void a() {
        b();
        System.out.println("here I am, in a()");
    }

    public synchronized void b() {
        System.out.println("here I am, in b()");
    }
}
```

Задаци

Семафор



Семафор

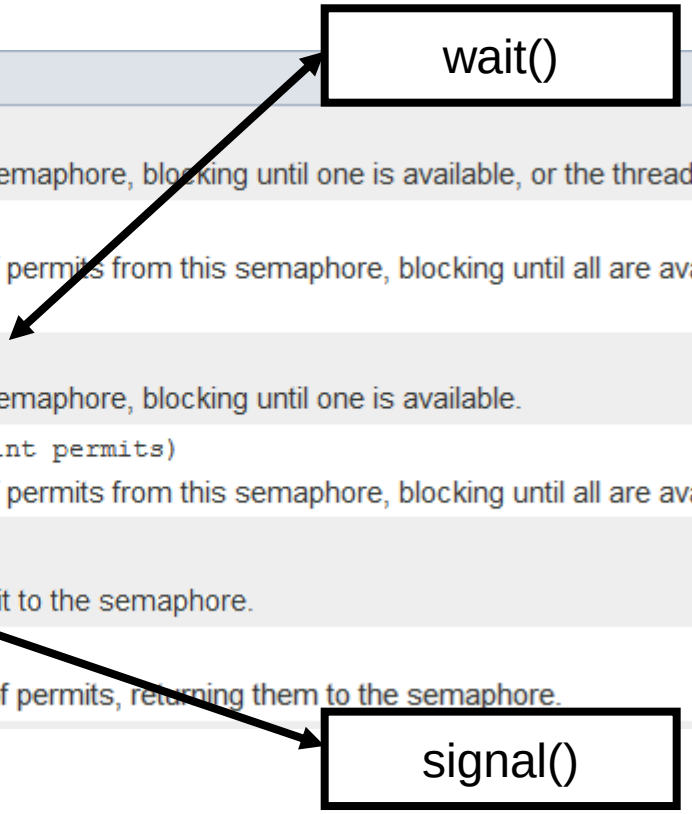
Користећи програмски језик Јава направити класу која одговара семафорима.

Семафор

```
public class Semaphore {
    private int s = 0;
    public synchronized void initS(int i) {
        s = i;
    }
    public synchronized void P() {
        while (s == 0) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        s = s - 1;
        // notifyAll();
    }
    public synchronized void V() {
        s = s + 1;
        notify();
        // if ( s == 1) notify();
        // if ( s == 1) notifyAll();
    }
}
```

Семафор (java.util.concurrent.*) 1

Modifier and Type	Method and Description
void	`acquire()` Acquires a permit from this semaphore, blocking until one is available, or the thread is **interrupted**.
void	`acquire(int permits)` Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is **interrupted**.
void	`acquireUninterruptibly()` Acquires a permit from this semaphore, blocking until one is available.
void	`acquireUninterruptibly(int permits)` Acquires the given number of permits from this semaphore, blocking until all are available.
void	`release()` Releases a permit, returning it to the semaphore.
void	`release(int permits)` Releases the given number of permits, returning them to the semaphore.



Семафор (java.util.concurrent.*) 2

Modifier and Type	Method and Description
int	availablePermits() Returns the current number of permits available in this semaphore.
int	drainPermits() Acquires and returns all permits that are immediately available.
protected Collection<Thread>	getQueuedThreads() Returns a collection containing threads that may be waiting to acquire.
int	getQueueLength() Returns an estimate of the number of threads waiting to acquire.
boolean	hasQueuedThreads() Queries whether any threads are waiting to acquire.
boolean	isFair() Returns true if this semaphore has fairness set true.
protected void	reducePermits(int reduction) Shrinks the number of available permits by the indicated reduction.
boolean	tryAcquire() Acquires a permit from this semaphore, only if one is available at the time of invocation.
boolean	tryAcquire(int permits) Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.
boolean	tryAcquire(int permits, long timeout, TimeUnit unit) Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not been **interrupted**.
boolean	tryAcquire(long timeout, TimeUnit unit) Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been **interrupted**.

Producer – Consumer problem

Producer – Consumer problem V1

```
public class BoundedBuffer<T> {

   public static final int N = 100;
   final T[] items = (T[]) new Object[N];
   int putptr, takeptr, count;
```

Producer – Consumer problem V1

```
public synchronized void put(T x) throws InterruptedException {
    while (count == items.length) {
            wait();
    }
    items[putptr] = x;
    putptr = (putptr + 1) % items.length;
    count++;
    notifyAll();
}
```

Producer – Consumer problem V1

```
public synchronized T get() throws InterruptedException {
    while (count == 0) {
            wait();
    }
    T x = items[takeptr];
    takeptr = (takeptr + 1) % items.length;
    count--;
    notifyAll();
    return x;
 }
```

Методе интерфејса Lock

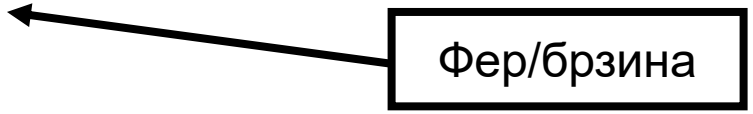
Modifier and Type	Method and Description
`void`	`lock()` Acquires the lock.
`void`	`lockInterruptibly()` Acquires the lock unless the current thread is **interrupted**.
`Condition`	`newCondition()` Returns a new `Condition` instance that is bound to this `Lock` instance.
`boolean`	`tryLock()` Acquires the lock only if it is free at the time of invocation.
`boolean`	`tryLock(long time, TimeUnit unit)` Acquires the lock if it is free within the given waiting time and the current thread has not been **interrupted**.
`void`	`unlock()` Releases the lock.

Методе интерфејса Condition

Modifier and Type	Method and Description
void	`await()` Causes the current thread to wait until it is signalled or **interrupted**.
boolean	`await(long time, TimeUnit unit)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
long	`awaitNanos(long nanosTimeout)` Causes the current thread to wait until it is signalled or interrupted, or the specified waiting time elapses.
void	`awaitUninterruptibly()` Causes the current thread to wait until it is signalled.
boolean	`awaitUntil(Date deadline)` Causes the current thread to wait until it is signalled or interrupted, or the specified deadline elapses.
void	`signal()` Wakes up one waiting thread.
void	`signalAll()` Wakes up all waiting threads.

Коришћење интерфејса Lock

```
import java.util.concurrent.locks.*;
...
Lock lock = new ReentrantLock();
// Lock lock = new ReentrantLock(true);
```




```
...
public boolean criticalSection() {
    lock.lock();
    try {
        work();
    } finally {
        lock.unlock();
    }
}
```

Коришћење интерфејса Lock

```
import java.util.concurrent.locks.*;
...
Lock lock = new ReentrantLock();
...
public boolean criticalSection() {
    boolean locked = false;
    try {
        locked = lock.tryLock();
        if(locked) {
            work();
        }
    } finally {
        if (locked) { lock.unlock(); }
    }
    return locked;
}
```

Producer – Consumer problem V2


```
public class BoundedBuffer<T> {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   public static final int N = 100;
   final T[] items = (T[]) new Object[N];
   int putptr, takeptr, count;
```

Producer – Consumer problem V2

```
public void put(T x) throws InterruptedException {
    lock.lock();
    try {
        while (count == items.length)
            notFull.await();
        items[putptr] = x;
        putptr = (putptr + 1) % items.length;
        count++;
        notEmpty.signal();
    } finally {
        lock.unlock();
    }
}
```

Producer – Consumer problem V2

```
public T get() throws InterruptedException {
    lock.lock();
    try {
        while (count == 0)
            notEmpty.await();
        T x = items[takeptr];
        takeptr = (takeptr + 1) % items.length;
        count--;
        notFull.signal();
        return x;
    } finally {
        lock.unlock();
    }
}
```

Producer – Consumer problem V3

Интерфејс java.util.concurrent.BlockingQueue<E>

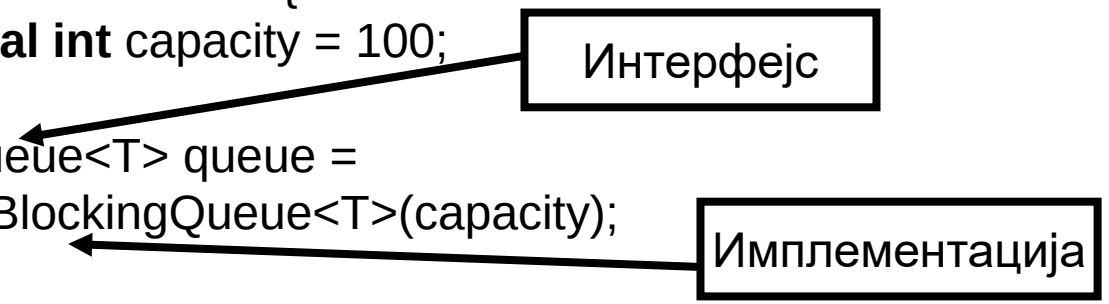
	Throws exception	*Special value*	*Blocks*	*Times out*
Insert	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Remove	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Examine	`element()`	`peek()`	*not applicable*	*not applicable*

Producer – Consumer problem V3

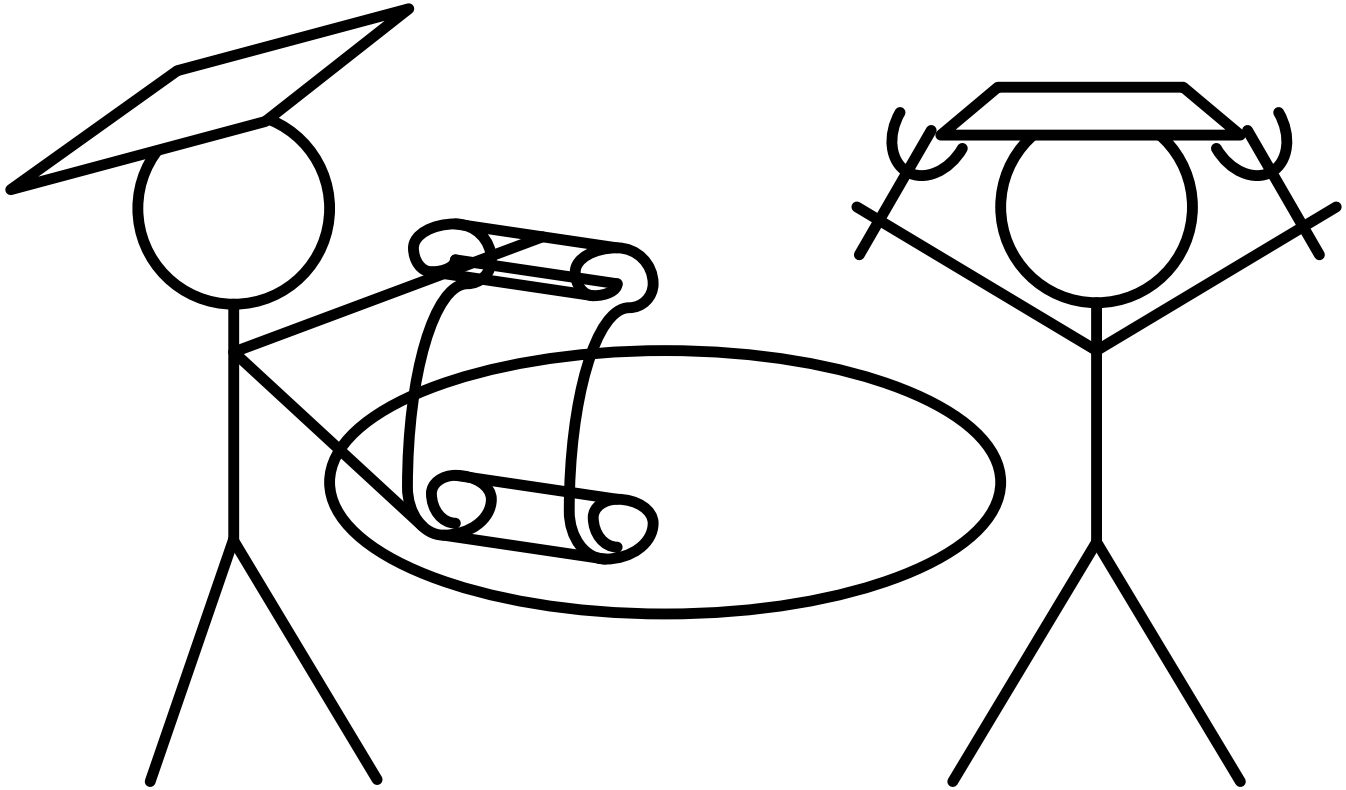
Modifier and Type	Method and Description
boolean	`add(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and throwing an `IllegalStateException` if no space is currently available.
boolean	`contains(Object o)` Returns `true` if this queue contains the specified element.
int	`drainTo(Collection<? super E> c)` Removes all available elements from this queue and adds them to the given collection.
int	`drainTo(Collection<? super E> c, int maxElements)` Removes at most the given number of available elements from this queue and adds them to the given collection.
boolean	`offer(E e)` Inserts the specified element into this queue if it is possible to do so immediately without violating capacity restrictions, returning `true` upon success and `false` if no space is currently available.
boolean	`offer(E e, long timeout, TimeUnit unit)` Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.
E	`poll(long timeout, TimeUnit unit)` Retrieves and removes the head of this queue, waiting up to the specified wait time if necessary for an element to become available.
void	`put(E e)` Inserts the specified element into this queue, waiting if necessary for space to become available.
int	`remainingCapacity()` Returns the number of additional elements that this queue can ideally (in the absence of memory or resource constraints) accept without blocking, or `Integer.MAX_VALUE` if there is no intrinsic limit.
boolean	`remove(Object o)` Removes a single instance of the specified element from this queue, if it is present.
E	`take()` Retrieves and removes the head of this queue, waiting if necessary until an element becomes available.

Producer – Consumer problem V3

```
public class BoundedBuffer<T> {
    public static final int capacity = 100;

    final BlockingQueue<T> queue =
            new ArrayBlockingQueue<T>(capacity);

    public void put(T x) throws InterruptedException {
            queue.put(x);
    }

    public T get() throws InterruptedException {
            return queue.take();
    }
}
```



Dining philosophers problem



Dining philosophers problem

```
public class Philosopher extends Thread {
    int id;
    int firstodd, secondeven;
    Semaphore [] fork;
    public Philosopher (int i, int n , Semaphore [] fork) {
        id = i;
        this.fork = fork;
        if (i % 2 == 1) {
            firstodd = i;
            secondeven = (i + 1) % n;
        }
        else{
            firstodd = (i + 1) % n;
            secondeven = i;
        }
    }
```

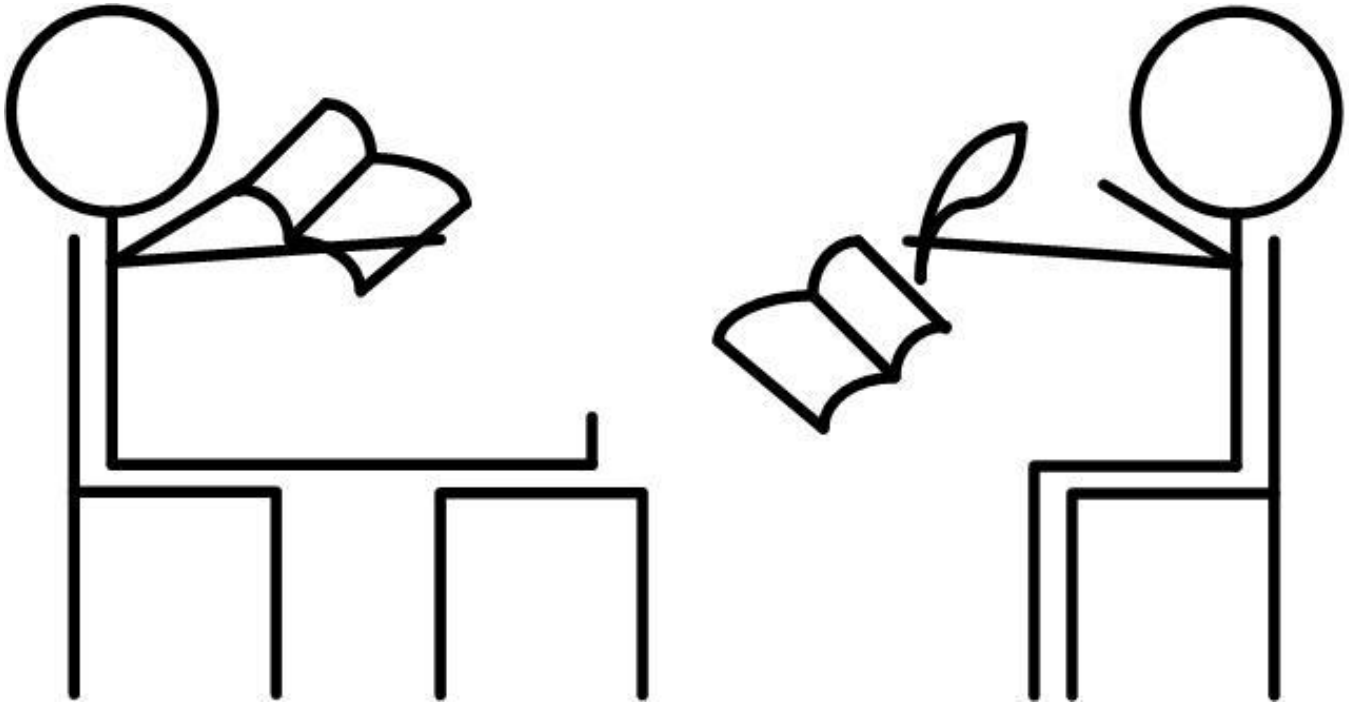
Dining philosophers problem

```
    private void think() {
            try {
                    sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) { }
    }
    private void eat() {
            try {
                    sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) { }
    }
    public void run() {
            while (true){
                    think();
                    fork[firstodd].P();
                    fork[secondeven].P();
                    eat();
                    fork[secondeven].V();
                    fork[firstodd].V();
            }
    }
}
```

Dining philosophers problem

```
public class Dining_philosophers{
public static final int N = 5;

public static void main(String [] vpar){
        int n = N;
        Semaphore [] fork = new Semaphore[n];
        Philosopher [] philosopher = new Philosopher[n];
        int i;
        for( i=0; i < n; i++) fork[i] = new Semaphore();
        for( i=0; i < n; i++) philosopher[i] = new Philosopher(i, n , fork);
        for( i=0; i < n; i++) fork[i].initS(1);
        philosopher[0].start();
        philosopher[1].start();
        philosopher[2].start();
        philosopher[3].start();
        philosopher[4].start();
    }
}
```

Readers – Writers problem



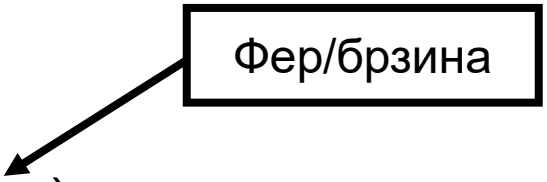
Readers – Writers problem

Интерфејс
java.util.concurrent.locks.ReadWriteLock

Modifier and Type	Method and Description
`Lock`	`readLock()` Returns the lock used for reading.
`Lock`	`writeLock()` Returns the lock used for writing.

Readers – Writers problem

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

public class RW {
    ReadWriteLock rw;
    Lock readLock, writeLock;

    public RW() {
        // rw = new ReentrantReadWriteLock(true);
        rw = new ReentrantReadWriteLock();
        readLock = rw.readLock();
        writeLock = rw.writeLock();
    }
```



Readers – Writers problem

```
public void startRead() {
    readLock.lock(); // rw.readLock().lock();
}

public void endRead() {
    readLock.unlock(); // rw.readLock().unlock();
}

public void startWrite() {
    writeLock.lock(); // rw.writeLock().lock();
}

public void endWrite() {
    writeLock.unlock(); // rw.writeLock().unlock();
}
```

Readers – Writers problem V2

Класа java.util.concurrent.locks.StampedLock

Modifier and Type	Method and Description
long	**readLock**() Non-exclusively acquires the lock, blocking if necessary until available.
long	**writeLock**() Exclusively acquires the lock, blocking if necessary until available.
void	**unlock**(long stamp) If the lock state matches the given stamp, releases the corresponding mode of the lock.
void	**unlockRead**(long stamp) If the lock state matches the given stamp, releases the non-exclusive lock.
void	**unlockWrite**(long stamp) If the lock state matches the given stamp, releases the exclusive lock.
long	**tryReadLock**() Non-exclusively acquires the lock if it is immediately available.
long	**tryWriteLock**() Exclusively acquires the lock if it is immediately available.
boolean	**tryUnlockRead**() Releases one hold of the read lock if it is held, without requiring a stamp value.
boolean	**tryUnlockWrite**() Releases the write lock if it is held, without requiring a stamp value.

Коришћење класе StampedLock

```
import java.util.concurrent.locks.StampedLock;
...
private final StampedLock sl = new StampedLock();


...
public boolean criticalWriteSection() {
    long stamp = sl.writeLock();
    try {
            work();
    } finally {
            sl.unlockWrite(stamp);
    }
}
```

Readers – Writers problem V2

```
import java.util.concurrent.locks.StampedLock;

public class RWStamp {

    private final StampedLock sl;

    public RWStamp() {
            sl = new StampedLock();
    }
```

Readers – Writers problem V2

```
public long startRead() {
    return sl.readLock();
}

public void endRead(long stamp) {
    sl.unlockRead(stamp);
}

public long startWrite() {
    return sl.writeLock();
}

public void endWrite(long stamp) {
    sl.unlockWrite(stamp);
}
```

Readers – Writers problem V3

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


public class ReadersWritersStamp {
    private final StampedLock sl;
    private Map<Thread, Long> stamps;

    public ReadersWritersStamp() {
        sl = new StampedLock();
        stamps = new ConcurrentHashMap<>();
        // stamps = Collections.synchronizedMap(
        //                  new HashMap<Thread, Long>());
    }
```

Readers – Writers problem V3

```
    public void startRead() {
            long stamp = sl.readLock();
            stamps.put(Thread.currentThread(), stamp);
    }
    public void endRead() {
            long stamp = stamps.remove(Thread.currentThread());
            sl.unlockRead(stamp);
    }
    public void startWrite() {
            long stamp = sl.writeLock();
            stamps.put(Thread.currentThread(), stamp);
    }
    public void endWrite() {
            long stamp = stamps.remove(Thread.currentThread());
            sl.unlockWrite(stamp);
    }
}
```

Readers – Writers problem V4

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

public class ReadersWritersStampThreadLocal  {
    private final StampedLock sl;
    private final ThreadLocal<Long> stamps;

    public ReadersWritersStampThreadLocal() {
        sl = new StampedLock();
        stamps = new ThreadLocal<Long>();
    }
```

Readers – Writers problem V4

```
    public void startRead() {
            long stamp = sl.readLock();
            stamps.set(stamp);
    }
    public void endRead() {
            long stamp = stamps.get();
            sl.unlockRead(stamp);
    }
    public void startWrite() {
            long stamp = sl.writeLock();
            stamps.set(stamp);
    }
    public void endWrite() {
            long stamp = stamps.get();
            sl.unlockWrite(stamp);
    }
}
```

Недељив приступ користећи volatile

У Јави постоје акције које се обављају недељиво/атомски:

- Операције читања или уписа у променљиве већине простих типова је атомска операција (изузеци су променљиве типа long и double).
- Операције читања или уписа у **све** променљиве декларисане као **volatile** (укључујући и променљиве типа long и double).
- Приступ за читање и упис **volatile** променљива ствара релацију уређеног приступа (*happens-before relationship*).

Коришћење атомског приступа може да буде ефикасније од приступа користећи синхронизоване блокове, али захтева више пажње како би се избегле грешке.

Недељив приступ користећи volatile

- Ово значи да:
  - Уколико нит A уписује вредност у неку **volatile** променљиву и уколико нит B након тога чита ту променљиву, онда су све променљиве које су биле видљиве нити A пре приступа датој променљивој такође видљиве нити B.
  - JVM неће мењати редослед инструкцијама које приступају **volatile** променљивама. Инструкције које се налазе пре или после приступа **volatile** променљивама се може мењати редослед, али се не могу мешати међусобно.

```
public class Shared {
    public volatile int cnt = 0;
    public int nonVolatile = 0;
}
```

Thread A:

```
shared.nonVolatile = 1;
shared.cnt = shared.cnt + 1;
```

Thread B:

```
int cnt = shared.cnt;
int nonVolatile = shared.nonVolatile;
```

synchronized и volatile

Карактеристика	synchronized	volatile
Тип променљиве	Објекат	Објекат или примитивни тип
Null објекат	Није дозвољено	Дозвољено
Блокирајући приступ	Да	Не
Синхронизован приступ променљивама?	Да	Почев од Јаве 5
Када се обавља синхронизација?	Приликом уласка/изласка из блока који је означен са synchronized	Приликом приступа променљивој означеној са volatile
Да ли може да се користи за креирање атомских операција?	Да	Пре Јаве 5, не. Од Јаве 5 постоји FA и TS!

Заустављање

```
import java.util.*;
import java.text.DateFormat;

public class Clock implements Runnable {
    private volatile Thread thread = null;

    public void start() {
        if (thread == null) {
            thread = new Thread(this, "Clock");
            thread.start();
        }
    }
```

Заустављање

```
public void run() {
        Thread myThread = Thread.currentThread();
        while (thread == myThread) {
                paint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){
                        System.out.println("Interrupted");
                }
        }
}
```

Заустављање

```
    public void paint() {
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        System.out.println(dateFormatter.format(date));
    }

    public void stop() {
        Thread stopThread = thread;
        thread = null;
        stopThread.interrupt();
    }
}
```

Заустављање - V2

```
import java.util.*;
import java.text.DateFormat;

public class Clock implements Runnable {
    private volatile boolean running = false;
    private Thread thread = null;

    public void start() {
        if (!running) {
            thread = new Thread(this, "Clock");
            running = true;
            thread.start();
        }
    }
```

Заустављање - V2

```
public void run() {
        while (running) {
                paint();
                try {
                        Thread.sleep(1000);
                } catch (InterruptedException e){
                        System.out.println("Interrupted");
                }
        }
}
```

Заустављање - V2

```
    public void paint() {
        Calendar cal = Calendar.getInstance();
        Date date = cal.getTime();
        DateFormat dateFormatter = DateFormat.getTimeInstance();
        System.out.println(dateFormatter.format(date));
    }

    public void stop() {
        Thread stopThread = thread;
        thread = null;
        running = false;
        stopThread.interrupt();
    }
}
```

Пакет java.util.concurrent.atomic

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.

Class AtomicInteger

Modifier and Type	Method and Description
int	addAndGet(int delta) Atomically adds the given value to the current value.
boolean	compareAndSet(int expect, int update) Atomically sets the value to the given updated value if the current value == the expected value.
int	decrementAndGet() Atomically decrements by one the current value.
double	doubleValue() Returns the value of the specified number as a double.
float	floatValue() Returns the value of the specified number as a float.
int	get() Gets the current value.
int	getAndAdd(int delta) Atomically adds the given value to the current value.
int	getAndDecrement() Atomically decrements by one the current value.
int	getAndIncrement() Atomically increments by one the current value.
int	getAndSet(int newValue) Atomically sets to the given value and returns the old value.

Fetch and add!

Test and set!

Class AtomicReferenceFieldUpdater<T,V>

Modifier and Type	Method and Description
abstract boolean	compareAndSet(T obj, V expect, V update) Atomically sets the field of the given object managed by this updater to the given updated value if the current value == the expected value.
abstract V	get(T obj) Gets the current value held in the field of the given object managed by this updater.
V	getAndSet(T obj, V newValue) Atomically sets the field of the given object managed by this updater to the given value and returns the old value.
abstract void	lazySet(T obj, V newValue) Eventually sets the field of the given object managed by this updater to the given updated value.
static <U,W> AtomicReferenceFieldUpdater<U,W>	newUpdater(Class<U> tclass, Class<W> vclass, String fieldName) Creates and returns an updater for objects with the given field.
abstract void	set(T obj, V newValue) Sets the field of the given object managed by this updater to the given updated value.
abstract boolean	weakCompareAndSet(T obj, V expect, V update) Atomically sets the field of the given object managed by this updater to the given updated value if the current value == the expected value.

Class AtomicReferenceFieldUpdater<T,V>

```
class Node {
  private volatile Node left;
  private volatile Node right;

  private static final
    AtomicReferenceFieldUpdater<Node, Node> leftUpdater =
    AtomicReferenceFieldUpdater.newUpdater(Node.class, Node.class, "left");

    Node getLeft() { return left;  }

    boolean compareAndSetLeft(Node expect, Node update) {
            return leftUpdater.compareAndSet(this, expect, update);
    }
  // ... and so on
}
```

Име променљиве

Међусобно искључивање – spin locks

- Користећи само приступ променљивама означеним као *volatile* и пакет за рад са атомским променљивама, *java.util.concurrent.atomic.**, обезбедити међусобно искључивање приликом приступа неком ресурсу.

Peterson алгоритам – за две нити

```
public class Peterson extends SimpleLock {
        private volatile boolean in0, in1;
        private volatile int last;

        public Peterson() {
                in0 = false;
                in1 = false;
                last = 0;
        }
```

Peterson алгоритам – за две нити

```
class CS0 extends SimpleLock {
        public void lock() {
                /* entry protocol CS0 */
                in0 = true;
                last = 0;
                while (in1 && last == 0) {
                        Thread.onSpinWait();
                }
        }
        public void unlock() {
                /* exit protocol CS0 */
                in0 = false;
        }
}
```

Peterson алгоритам – за две нити

```
class CS1 extends SimpleLock {
        public void lock() {
                /* entry protocol CS1 */
                in1 = true;
                last = 1;
                while (in0 && last == 1) {
                        Thread.onSpinWait();
                }
        }
        public void unlock() {
                /* exit protocol CS1 */
                in1 = false;
        }
}
```

Peterson алгоритам – за две нити

```
@Override
public SimpleLock create(int id) {
        SimpleLock lock = (id == 0 ? new CS0() : new CS1());
        return lock;
}
```

Ticket алгоритам – лоше решење

```
public class Ticket_Deadlock extends SimpleLock {
        private volatile int number;
        private volatile int next;
        public Ticket_Deadlock() {
                number = 0; next = 0;
        }
        public void lock() {
                int turn = number++;
                while (turn != next) {
                        Thread.onSpinWait();
                }
        }
        public void unlock() {
                next = next + 1;
        }
}
```

Није атомично!

Ticket алгоритам

```
public class Ticket extends SimpleLock {
        private AtomicInteger number;
        private volatile int next;
        public Ticket() {
                number = new AtomicInteger(0);
                next = 0;
        }
        public void lock() {
                int turn = number.getAndAdd(1);
                // int turn = number.getAndIncrement();
                while (turn != next) { Thread.onSpinWait(); }
        }
        public void unlock() {
                next = next + 1;
        }
}
```

Test and Set алгоритам

```
public class TestAndSet extends SimpleLock {
        private final AtomicBoolean lock;
        public TestAndSet() {
                lock = new AtomicBoolean(false);
        }
        public void lock() {
                while (lock.getAndSet(true)) {
                        Thread.onSpinWait();
                }
        }
        public void unlock() {
                lock.set(false);
        }
}
```

Test and Test and Set алгоритам

```
public class TestAndTestAndSet extends SimpleLock {
        private final AtomicBoolean lock;
        public TestAndTestAndSet() {
                lock = new AtomicBoolean(false);
        }
        public void lock() {
                while (lock.get()) {
                        Thread.onSpinWait();
                }
                while (lock.getAndSet(true)) {
                        while (lock.get()) {
                                Thread.onSpinWait();
                        }
                }
        }
        public void unlock() { lock.set(false); }
}
```

CLH lock алгоритам

```
public class CLHLock extends SimpleLock {
        AtomicReference<Node> tail;
        ThreadLocal<Node> myPred;
        ThreadLocal<Node> myNode;
        public CLHLock() {
                tail = new AtomicReference<Node>(new Node());
                myNode = new ThreadLocal<Node>() {
                        protected Node initialValue() {
                                return new Node();
                        }
                };
                myPred = new ThreadLocal<Node>() {
                        protected Node initialValue() {
                                return null;
                        }
                };
        }
```

CLH lock алгоритам

```
public void lock() {
        /* entry protocol */
        Node node = myNode.get();
        node.locked = true;
        Node pred = tail.getAndSet(node);
        myPred.set(pred);
        while (pred.locked) {
                Thread.onSpinWait();
        }
}
public void unlock() {
        /* exit protocol */
        Node node = myNode.get();
        node.locked = false;
        myNode.set(myPred.get());
}
```

CLH lock алгоритам

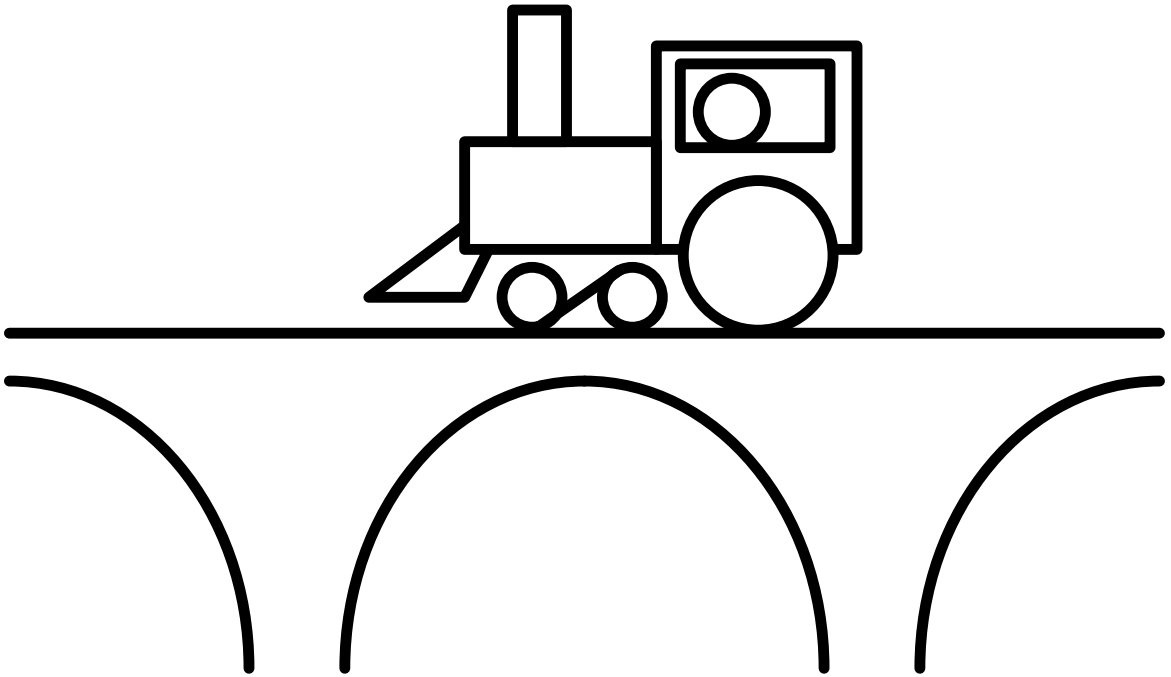
```
        static final class Node {
                volatile boolean locked = false;
        }
}
```

или:

```
public void lock() {
  /* entry protocol */
  Node node = new Node();
  node.locked = true;
  myNode.set(node);
  Node pred = tail.getAndSet(node);
  while (pred.locked) {
        Thread.onSpinWait();
  }
}
```

```
public void unlock() {
        /* exit protocol */
        Node node = myNode.get();
        node.locked = false;
}
```

One-lane bridge problem



One-lane bridge problem

```
public class Bridge{

    public int north;
    public int south;
    public Bridge(int north, int south){
        this.north = north;
        this.south = south;
    }
}
```

One-lane bridge problem

```
public class Car{
    Bridge bridge = null;
    int id;
    public Car(Bridge bridge, int id){
        this.bridge = bridge;
        this.id = id;
    }
    public void crossing(){
        try { Thread.sleep(5000+(int)(Math.random()*1000));
        } catch (InterruptedException e) { }
    }
    public void starting(){
        try {Thread.sleep(5000+(int)(Math.random()*1000));
        } catch (InterruptedException e) { }
    }
    public void start() { }
}
```

One-lane bridge problem

```
public class North extends Car implements Runnable {

    private volatile Thread thread = null;

    public North(Bridge bridge, int ID){
        super(bridge, ID);
    }

    public void start() {
        if (thread == null) {
            thread = new Thread(this);
            thread.start();
        }
    }
```

One-lane bridge problem

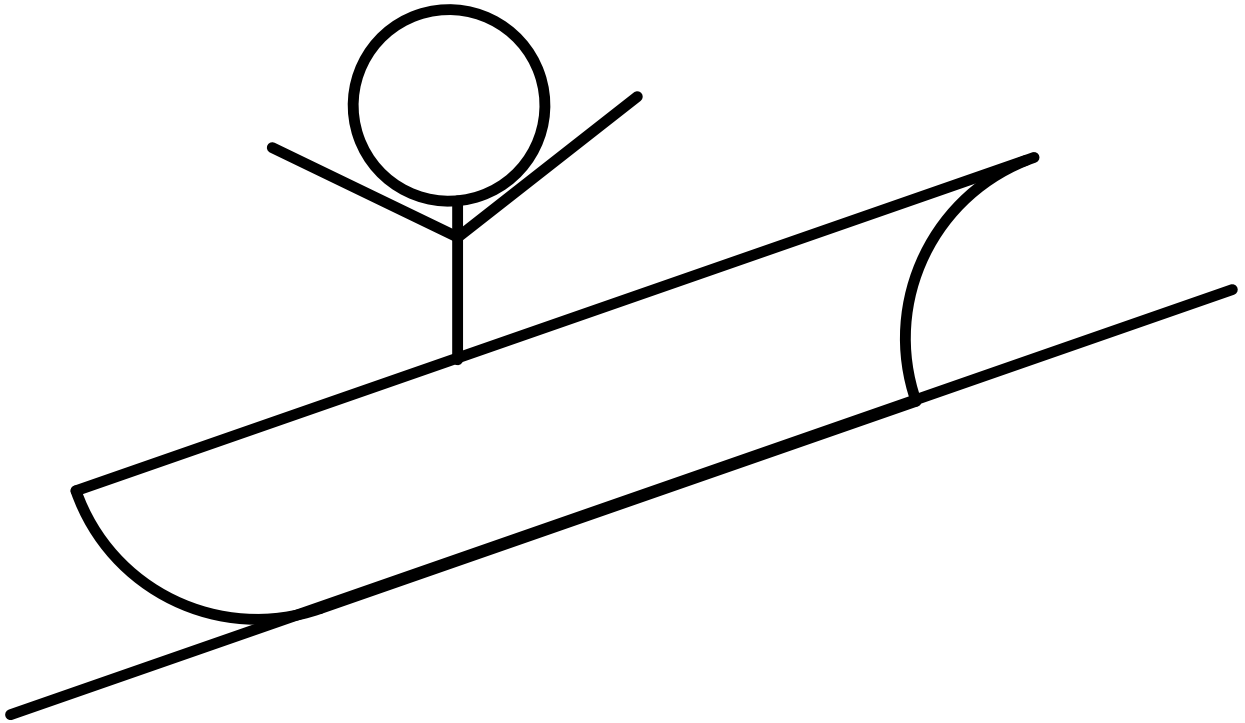
```
public void run() {
        starting();
        synchronized(bridge){
                while (bridge.south != 0) {
                        try {
                                bridge.wait();
                        } catch (InterruptedException e) { }
                }
                bridge.north ++;
                bridge.notifyAll();
        }
        crossing();
        synchronized(bridge){
                bridge.north --;
                if(bridge.north == 0)
                bridge.notifyAll();
        }
    }
}
```

Вишак!

One-lane bridge problem

```
public class BridgeCrossing{
    public static final int N = ...;
    public static void main(String []args){
        int n = N;
        Bridge bridge = new Bridge(0,0);
        Car [] southt = new Car[n];
        Car [] north = new Car[n];
        for(int i=0; i < n; i++) {
            south[i] = new South(bridge, 2*i);
            north[i] = new North(bridge, 2*i+1);
        }
        for(int i=0; i < n; i++) {
            south[i].start();
            north[i].start();
        }
    }
}
```

The roller coaster problem



The roller coaster problem

Проблем вожње тобоганом. Претпоставити да постоји n нити које представљају путнике и једна нит која представља возило на тобогану. Путници се наизменично шетају по луна парку и возе на тобогану. Тобоган може да прими највише K путника при чему је K < n. Вожња тобоганом може да почне само уколико се сакупило тачно K путника. Написати програм на језику Java који симулира описани систем.

The roller coaster problem – V1

```
public class Coaster extends Thread {
    …
    public void run() {
        while (true) {
            boardCar();
            ride();
            leaveCar();
        }
    }
```

The roller coaster problem – V1

```
public void boardCar() {
    for(int i = 0; i < k; i++) car.V();
    allAboard.P();
}

public void leaveCar() {
    for (int i = 0; i < k; i ++) lastStop.V();
    allOut.P();
}
```

The roller coaster problem – V1

```
public class Passenger extends Thread{
    …
    public void run(){
        while(true){
            boardCar();
            ride();
            leaveCar();
        }
    }
```

The roller coaster problem – V1

```
public void boardCar(){
        car.P();
        mutex.P();
        coaster.passengers++;
        if (coaster.passengers == k){
                allAboard.V();
        }
        mutex.V();
    }
    public void leaveCar(){
        lastStop.P();
        mutex.P();
        coaster.passengers--;
        if (coaster.passengers == 0){
                allOut.V();
        }
        mutex.V();
    }
}
```

The roller coaster problem – V1

```
public class RollerCoaster{

    public RollerCoaster(){
        car.initS(0);
        allAboard.initS(0);
        lastStop.initS(0);
        mutex.initS(1);
        allOut.initS(0);
        coaster.passengers = 0;
    }
}
```

The roller coaster problem – V2

```
public class Coaster extends Thread{
    …

    public void boardCar(){
        car.release(k);
        allAboard.acquire();
    }

    public void leaveCar(){
        lastStop.release(k);
        allOut.acquire();
    }
}
```

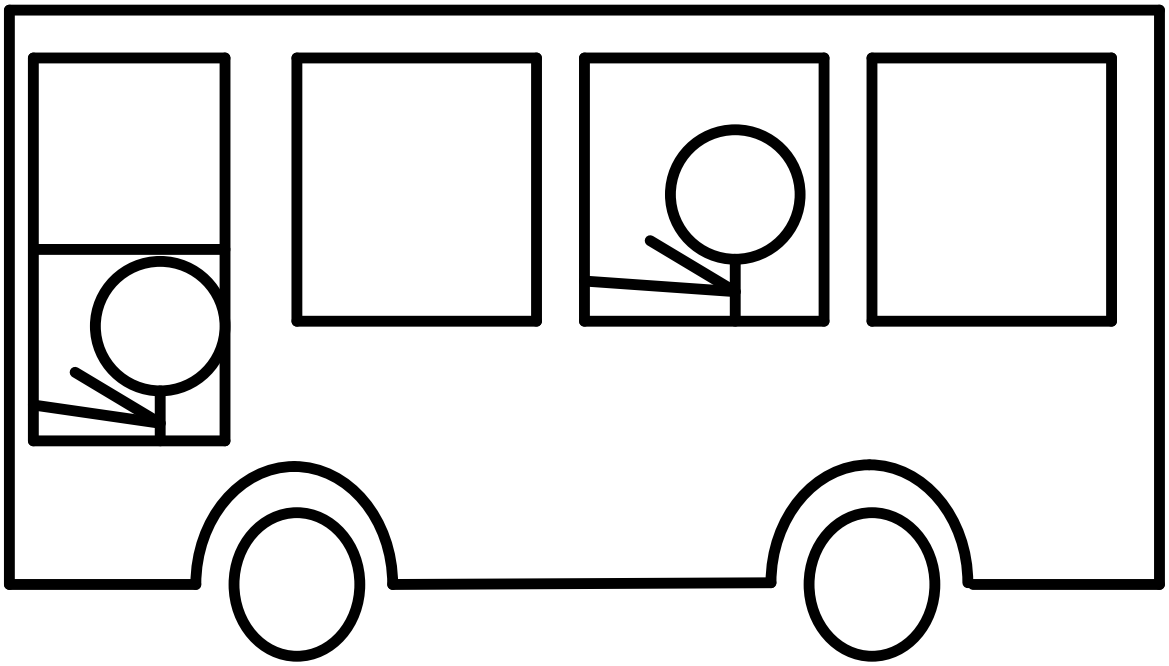
The roller coaster problem – V2

```
public void boardCar() throws InterruptedException {
    car.acquire();
    mutex.acquire();
    coaster.passengers++;
    if (coaster.passengers == k) {
        allAboard.release();
    }
    mutex.release();
}
public void leaveCar() throws InterruptedException {
    lastStop.acquire();
    mutex.acquire();
    coaster.passengers--;
    if (coaster.passengers == 0) {
        allOut.release();
    }
    mutex.release();
}
```

The roller coaster problem – V2

```
public class RollerCoaster{

    public RollerCoaster(){
        car = new Semaphore(0);
        allAboard = new Semaphore(0);
        lastStop  = new Semaphore(0);
        allOut  = new Semaphore(0);
        mutex  = new Semaphore(1);
        passengers = 0;
    }
}
```

The bus problem



The bus problem

Проблем вожње аутобусом. Путници долазе на аутобуску станицу и чекају први аутобус који наиђе. Када аутобус наиђе сви путници који су били на станици пробају да уђу у аутобус. Уколико има места у аутобусу путници улазе у њега. Капацитет аутобуса је K места. Путници који су дошли док је аутобус био на станици чекају на следећи аутобус. Када сви путници који су били на станици у тренутку доласка аутобуса провере да ли могу да уђу и уђу уколико има места аутобус креће. Уколико аутобус дође на празну станицу одмах продужава даље. Сви путници силазе на завршној станици. Написати програм на језику Java који симулира описани систем.

The bus problem

```
public class Station{
    private String name;
    private int id;
    private int numP, numFP;
    private boolean busIN;
    private Bus bus;
    public Station(String name){
        this.name = name;
        bus = null;
        busIN = false;
    }
```

The bus problem

```
public synchronized Bus waitBus(){
        while(true){
                while (busIN) {
                        try {wait();
                        } catch (InterruptedException e) { }
                }
                numP ++;
                while (!busIN) {
                        try {wait();
                        } catch (InterruptedException e) { }
                }
                numP--;
                if(numP == 0) notifyAll();
                if(numFP > 0){
                        numFP--;
                        return bus;
                }
        }
    }
```

The bus problem

```
public synchronized int busEnter(int numFree, Bus b) {
        while (busIN) {
                try {wait();
                } catch (InterruptedException e) { }
        }
        busIN = true;
        numFP = numFree;
        bus = b;
        notifyAll();
        while (numP !=0) {
                try { wait();
                } catch (InterruptedException e) {System.out.println(e); }
        }
        busIN = false;
        bus = null;
        notifyAll();
        return numFP;
    }
}
```

The bus problem

```
public class Passenger extends Thread{
    public static int ID = 0;
    int PID;
    Station start, end;
    Bus bus;
    public Passenger(Station start, Station end){
            PID = ID ++;
            this.start = start;
            this.end = end;
    }
```

The bus problem

```
public void run(){
        while(true){
                bus = start.waitBus();
                travel();
                bus.exitTheBus(end);
        }
    }
}
```

The bus problem

```
public class Bus extends Thread{
    private static final int maxNumPasage = 50;
    private StationLIst sl;
    private Station nextStation;
    private int numFree;
    private boolean toExit = false;
    private static int ID = 0;
    private int BID = ID++;
    public Bus(){
        numFree = maxNumPasage;
        sl = new StationLIst();
    }
```

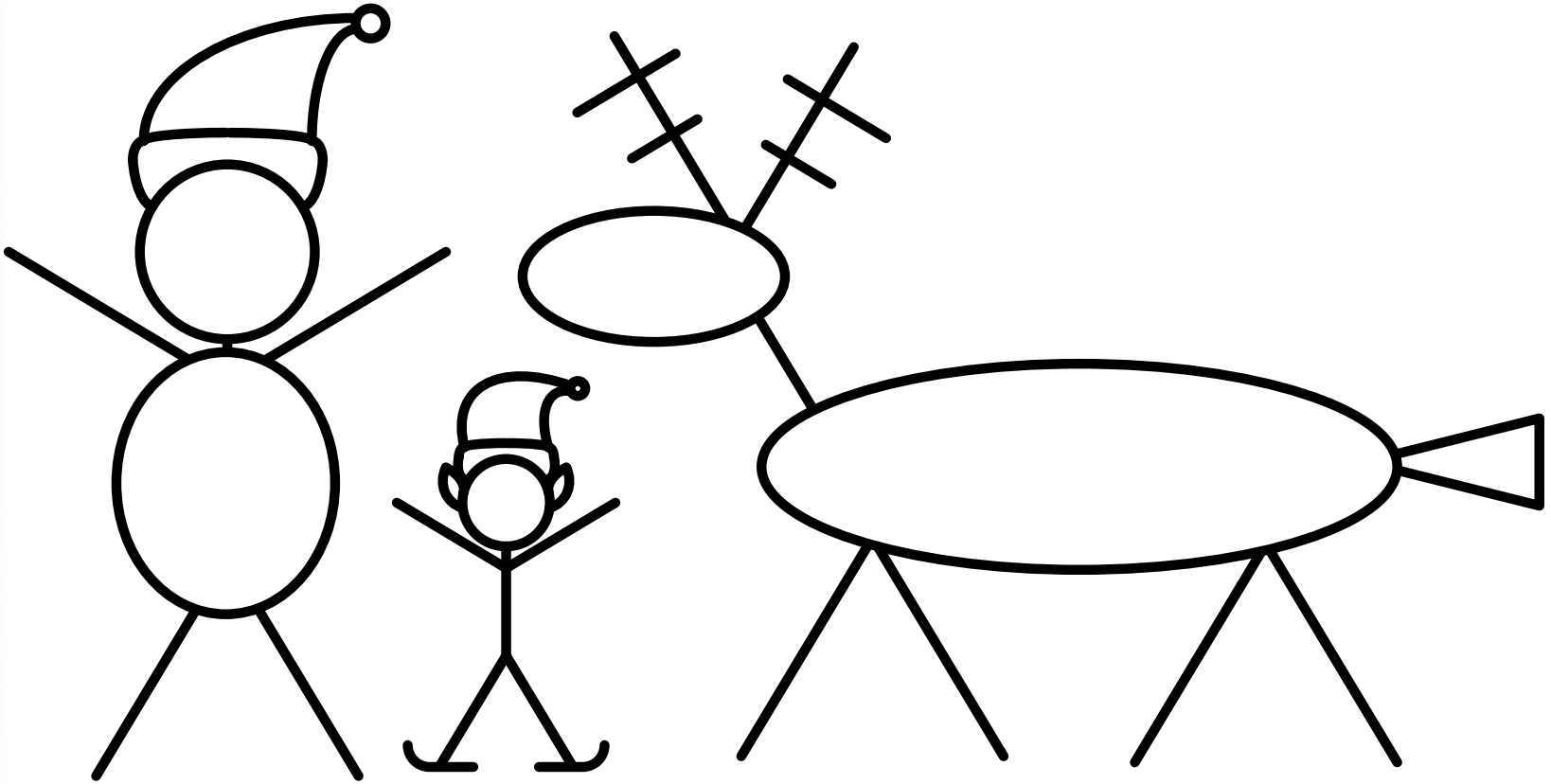
The bus problem

```
private synchronized Station getNextStation(){
        notifyAll();
        nextStation = sl.getNext();
        return nextStation;
}
public synchronized void exitTheBus(Station exitStation){
        while (!toExit || exitStation != nextStation) {
                try {
                        wait();
                } catch (InterruptedException e) { }
        }
        numFree++;
}
public synchronized void permitionToExit(){
        toExit = true;
        notifyAll();
        try {
                wait(500);
        } catch (InterruptedException e) { }
        toExit = false;
}
```

The bus problem

```
public void run(){
        while(true){
                nextStation = getNextStation();
                travel();
                permitionToExit();
                numFree = nextStation.busEnter(numFree, this);
        }
    }
}
```

The Santa Claus problem



The Santa Claus problem

Деда Мраз који живи на северном полу већи део свог времена проводи спавајући. Могу га пробудити или уколико се испред врата појаве свих 9 његових ирваса или 3 од укупно 10 патуљака. Када се Деда Мраз пробуди он ради једну од следећих ствари: Уколико га је пробудила група ирваса одмах се спрема и креће на пут да подели деци играчке. Када се врати са пута свим ирвасима даје награду. Уколико га је пробудила група патуљака онда их он уводи у своју кућу, разговара са њима и на крају их испрати до излазних врата. Група ирваса треба да буде опслужена пре групе патуљака. Написати програм на језику Java који симулира описани систем.

The Santa Claus problem

```
public class Elf extends Thread{
        private int id;
        private SantaClausHouse sc;

        public void run(){
                for(;;){
                        work();
                        sc.wakeupSantaE();
                        talk();
                        sc.exitTheRoom();
                }
        }
}
```

The Santa Claus problem

```
public class Reindeer extends Thread{
    private int id;
    private SantaClausHouse sc;

    public void run(){
                    for(;;){
                            rest();
                            sc.wakeupSantaR();
                            riding();
                            sc.exitTheSleigh();
                    }
            }
    }
```

The Santa Claus problem

```
public class SantaClaus extends Thread{
    private int dir;
    private SantaClausHouse sc;

    public void run(){
        for(;;) {
            dir = sc.sleeping();
            if (dir == SantaClausHouse.ELVES) {
                talk();
                sc.sendoffElves();
            }
            else {
                riding();
                sc.outspanReindeers();
            }
        }
    }
}
```

The Santa Claus problem

```
public class SantaClausHouse {
    public static final int ELVES = 0;
    public static final int REINDEERS = 1;
    private static final int numReindeers = 9;
    private static final int numElves = 10;
    private static final int minElves = 3;
    private static final int minReindeers = 9;
    private int elvesAtTheDoor;
    private int reindeersAtTheDoor;
    private int elvesInTheRoom;
    private int reindeersInTheSleigh;
    private boolean wakeupE, wakeupR;
    private boolean enterElves, exitElves;
    private boolean enterReindeers, exitReindeers;
    private boolean isSleeping;
```

The Santa Claus problem

```
public SantaClausHouse() {
        elvesAtTheDoor = 0;
        reindeersAtTheDoor = 0;
        elvesInTheRoom = 0;
        reindeersInTheSleigh = 0;
        wakeupE = false;
        wakeupR = false;
        enterElves = false;
        exitElves = false;
        enterReindeers = false;
        exitReindeers = false;
        isSleeping = true;
    }
```

The Santa Claus problem

```
public synchronized void wakeupSantaE(){
        while (!isSleeping) {
                try {
                        wait();
                } catch (InterruptedException e) {; }
        }
        elvesAtTheDoor ++;
        if(elvesAtTheDoor == minElves){
                wakeupE = true;
        }
        notifyAll();
        while (!enterElves) {
                try {
                        wait();
                } catch (InterruptedException e) {; }
        }
        elvesAtTheDoor --;
        elvesInTheRoom ++;
        notifyAll();
    }
```

The Santa Claus problem

```
public synchronized void exitTheRoom(){
        while (!exitElves) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        elvesInTheRoom --;
        notifyAll();
    }
```

The Santa Claus problem

```
public synchronized void wakeupSantaR(){
        while (!isSleeping) {
                try {
                        wait();
                } catch (InterruptedException e) {; }
        }
        reindeersAtTheDoor ++;
        if(reindeersAtTheDoor == minReindeers){
                wakeupR = true;
        }
        notifyAll();
        while (!enterReindeers) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        reindeersAtTheDoor --;
        reindeersInTheSleigh ++;
        notifyAll();
    }
```

The Santa Claus problem

```
public synchronized void exitTheSleigh(){
        while (!exitReindeers) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        reindeersInTheSleigh --;
        notifyAll();
    }
```

The Santa Claus problem

```
public synchronized int sleeping(){
        while (!(wakeupR || wakeupE)) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        isSleeping = false;
        if(wakeupR){
                wakeupR = false;
                enterReindeers = true;
                notifyAll();
                while (reindeersAtTheDoor != 0) {
                        try {
                                wait();
                        } catch (InterruptedException e) { ;}
                }
                enterReindeers = false;
                notifyAll();
                return SantaClausHouse.REINDEERS;
}
```

The Santa Claus problem

```
        else{
                wakeupE = false;
                enterElves= true;
                notifyAll();
                while (elvesAtTheDoor != 0) {
                        try {
                                wait();
                        } catch (InterruptedException e) { ;}
                }
                enterElves = false;
                notifyAll();

                return SantaClausHouse.ELVES;
        }

}
```

The Santa Claus problem

```
public synchronized void sendoffElves(){
        exitElves = true;
        notifyAll();
        while (elvesInTheRoom != 0) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        exitElves = false;
        enterElves = false;
        isSleeping = true;
        notifyAll();

}
```

The Santa Claus problem

```
public synchronized void outspanReindeers(){
        exitReindeers = true;
        notifyAll();
        while (reindeersInTheSleigh != 0) {
                try {
                        wait();
                } catch (InterruptedException e) { ;}
        }
        exitReindeers = false;
        enterReindeers = false;
        isSleeping = true;
        notifyAll();

    }
}
```

The Santa Claus problem

```
public class SantaClausTest{
    public static final int numReindeers = 9;
    public static final int numElves = 10;
    public static void main(String [] args){
        SantaClausHouse sc = new SantaClausHouse();
        SantaClaus santa = new SantaClaus(sc);
        Elf [] Elves = new Elf[numElves];
        Reindeer [] Reindeers = new Reindeer[numReindeers];
        for(int i = 0; i < numElves; i ++) {
            Elves[i] = new Elf(i, sc);
            Elves[i].start();
        }
        for(int i = 0; i < numReindeers; i ++){
            Reindeers[i] = new Reindeer(i, sc);
            Reindeers[i].start();
        }
        santa.start();
    }
}
```

Питања?

Захарије Радивојевић, Сања Делчев
Електротехнички Факултет
Универзитет у Београду
zaki@etf.rs, sanjad@etf.rs