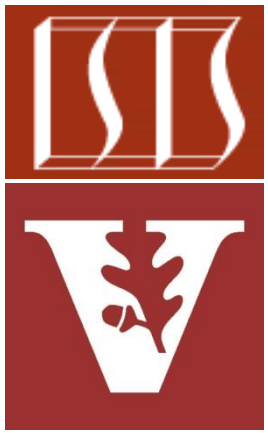
# Java Semaphore (Part 2)

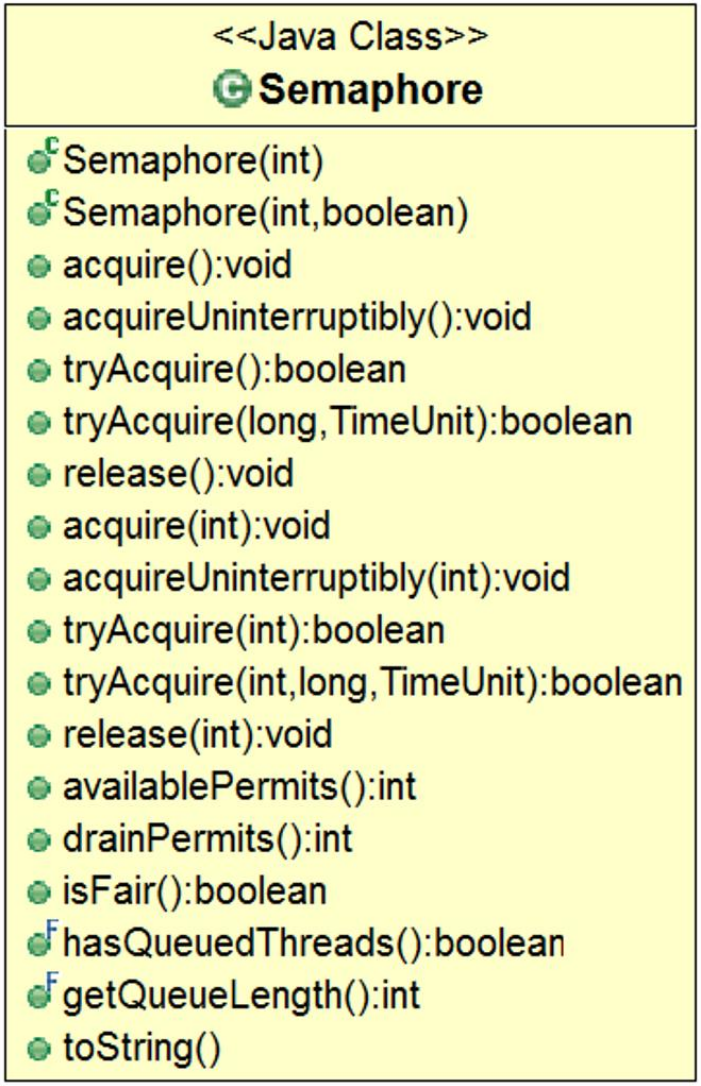


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

# Learning Objectives in this Part of the Lesson

- Appreciate the concept of semaphores
- Recognize the two types of semaphores
- Know a human known use of semaphores
- Understand the structure & functionality of Java Semaphore & its methods



# Overview of Java Semaphores

# Overview of Java Semaphores

- Implements a variant of counting semaphores

```
public class Semaphore
             implements ... {
  ...
```

## Class Semaphore

java.lang.Object
    java.util.concurrent.Semaphore

**All Implemented Interfaces:**

Serializable

---

```
public class Semaphore
extends Object
implements Serializable
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

# Overview of Java Semaphores

- Implements a variant of counting semaphores

```
public class Semaphore
             implements ... {
  ...
```

*Semaphore doesn't implement any synchronization-related interfaces*

## Class Semaphore

java.lang.Object
    java.util.concurrent.Semaphore

**All Implemented Interfaces:**

Serializable

---

```
public class Semaphore
extends Object
implements Serializable
```

A counting semaphore. Conceptually, a semaphore maintains a set of permits. Each `acquire()` blocks if necessary until a permit is available, and then takes it. Each `release()` adds a permit, potentially releasing a blocking acquirer. However, no actual permit objects are used; the `Semaphore` just keeps a count of the number available and acts accordingly.

Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

# Overview of Java Semaphores

- Constructors create semaphore with a given # of permits

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    ...
  }

  public Semaphore
             (int permits,
              boolean fair) {
    ...
  }
  ...
```

# Overview of Java Semaphores

- Constructors create semaphore with a given # of permits
  - This # is *not* a maximum, it's just an initial value

```
public class Semaphore
             implements ... {
  ...
  public Semaphore
              (int permits) {
    ...
  }

  public Semaphore
             (int permits,
              boolean fair) {
    ...
  }
  ...
```

See stackoverflow.com/questions/7554839/how-and-why-can-a-semaphore-give-out-more-permits-than-it-was-initialized-with

# Overview of Java Semaphores

- Constructors create semaphore with a given # of permits
  - This # is *not* a maximum, it's just an initial value
  - The initial permit value can be negative!!

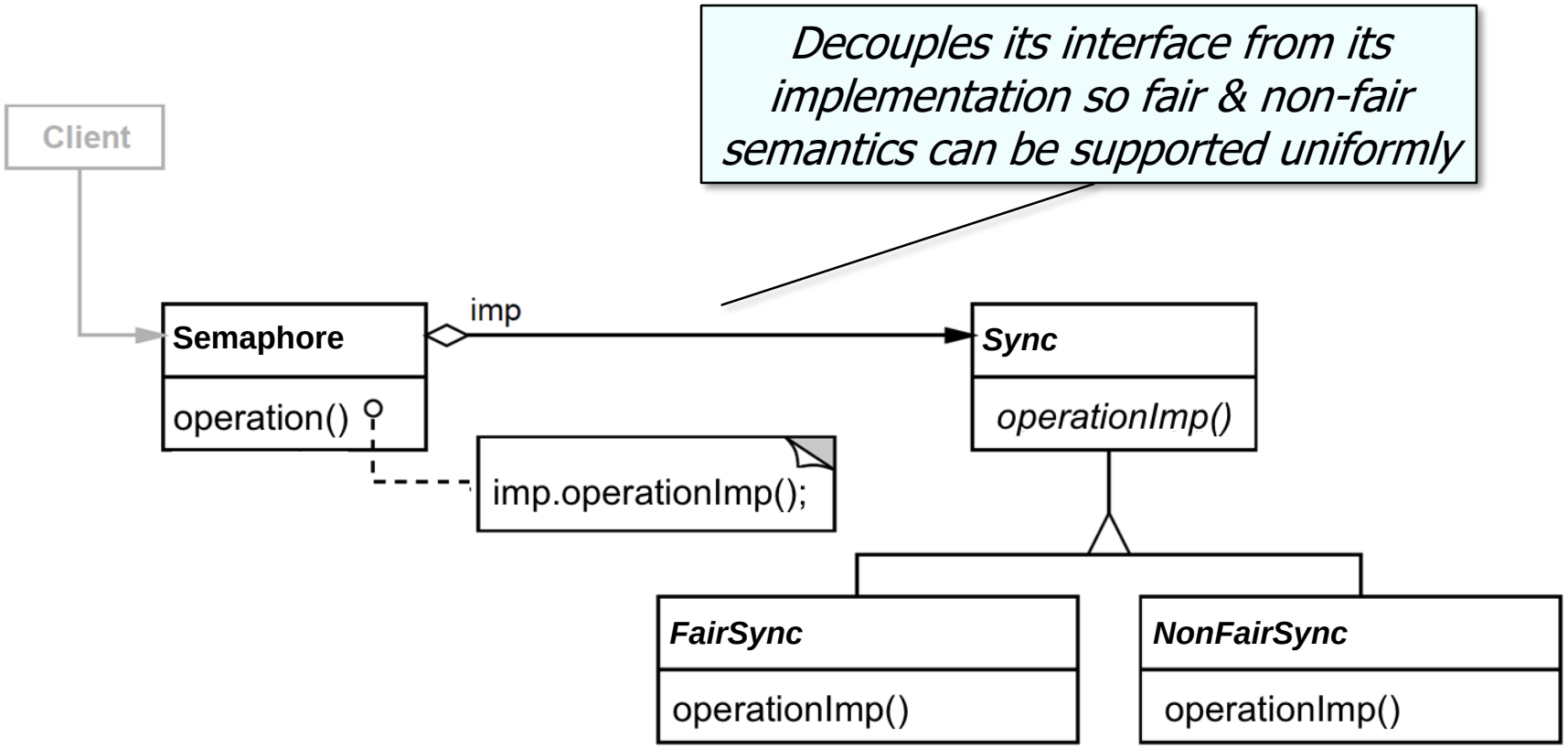
```
public class Semaphore
             implements ... {
   ...

Semaphore s = new Semaphore(-1);
...
```

In this case, all threads will block trying to acquire the semaphore until some thread(s) increment the permit value until it's positive

# Overview of Java Semaphores

- Applies the *Bridge* pattern

```
public class Semaphore
             implements ... {
  ...
```




See en.wikipedia.org/wiki/Bridge_pattern

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy

```
public class Semaphore
             implements ... {
  ...
  /** Performs sync mechanics */
  private final Sync sync;
```

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
    - Many Java synchronizers that rely on FIFO wait queues use this framework

```
public class Semaphore
             implements ... {
  ...
  /** Performs sync mechanics */
  private final Sync sync;

  /**
    * Synchronization implementation
    * for semaphore
    */
  abstract static class Sync extends
      AbstractQueuedSynchronizer {
      ...
  }
```

See gee.cs.oswego.edu/dl/papers/aqs.pdf

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new NonfairSync(permits);
  }
  ...
```

The Semaphore fair & non-fair models follow the same pattern used by the Java ReentrantLock

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new NonfairSync(permits);
  }
  ...
```

*This param determines whether FairSync or NonfairSync is used*

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
  - Optionally implement fair or non-fair lock acquisition model

*Ensures strict "FIFO" fairness, at the expense of performance*

```
public class Semaphore
             implements ... {
  ...

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new NonfairSync(permits);
  }
  ...
```

# Overview of Java Semaphores

- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new NonfairSync(permits);
  }
  ...
```

*Enables faster performance at the expense of fairness*

# Overview of Java Semaphores

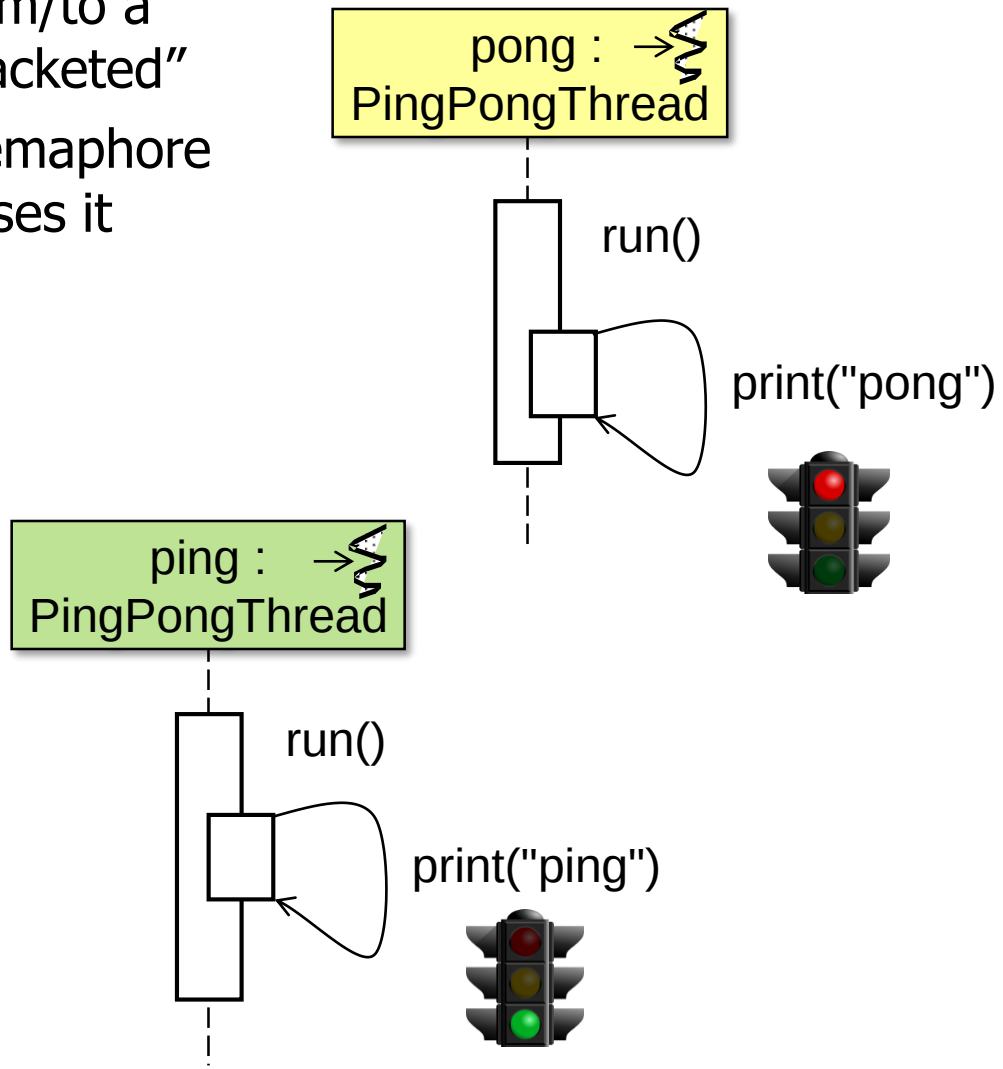
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Reuses functionality from AbstractQueuedSynchronizer
  - Optionally implement fair or non-fair lock acquisition model

```
public class Semaphore
             implements ... {
  ...

  public Semaphore
              (int permits,
               boolean fair) {
    sync = fair
      ? new FairSync(permits)
      : new NonfairSync(permits);
  }

  public Semaphore
               (int permits) {
    sync = new
      NonfairSync(permits);
  }
  ...
```

*The default behavior favors performance over fairness*

# Overview of Java Semaphores

- Acquiring & releasing permits from/to a semaphore need not be "fully bracketed"
  - i.e., a thread that acquires a semaphore need not be the one that releases it




See example in part 3 of this lesson

# Overview of Key Java Semaphore Methods

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore

```
public class Semaphore
             implements ... {
  ...
  public void acquire() { ... }

  public void
    acquireUninterruptibly()
  { ... }

  public boolean tryAcquire
          (long timeout,
           TimeUnit unit)
  { ... }

  public void release() { ... }
  ...
```

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore

```
public class Semaphore
             implements ... {
  ...
  public void acquire() { ... }

  public void
    acquireUninterruptibly()
  { ... }

  public boolean tryAcquire
         (long timeout,
          TimeUnit unit)
  { ... }

  public void release() { ... }
  ...
```

*These methods forward to their implementor methods, most of which are inherited from the AbstractQueuedSynchronizer framework*

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore

```
public class Semaphore
             implements ... {
  ...
  public void acquire() {
    sync.
    acquireSharedInterruptibly(1);
  }
  ...
```

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore
    - Can be interrupted

```
public class Semaphore
             implements ... {
  ...
  public void acquire() {
    sync.
    acquireSharedInterruptibly(1);
  }
  ...
```


See docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore
  - acquireUninterruptibly() also obtains a permit from the semaphore
    - Cannot be interrupted

```
public class Semaphore
             implements ... {
  ...
  public void
    acquireUninterruptibly() {
    sync.acquireShared(1)
  }
  ...
```

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore
  - acquireUninterruptibly() also obtains a permit from the semaphore
  - tryAcquire() obtains a permit if it's available at invocation time

```
public class Semaphore
             implements ... {
  ...
  public boolean tryAcquire()
                                  ... {
    sync.
      nonfairTryAcquireShared(1)
      >= 0;
  }
  ...
```

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore
  - acquireUninterruptibly() also obtains a permit from the semaphore
  - tryAcquire() obtains a permit if it's available at invocation time

```
public class Semaphore
             implements ... {
  ...
  public boolean tryAcquire()
                                  ... {
    sync.
      nonfairTryAcquireShared(1)
      >= 0;
  }
  ...
```

Untimed tryAcquire() methods will "barge", i.e., they don't honor the fairness setting & take any permits available

# Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() atomically obtains a permit from the semaphore
  - acquireUninterruptibly() also obtains a permit from the semaphore
  - tryAcquire() obtains a permit if it's available at invocation time
  - release() atomically increments the permit count by 1

```
public class Semaphore
             implements ... {
  ...
  public void release() {
    sync.releaseShared(1);
  }
  ...
```

Recall it's valid for the permit count to exceed the initial permit count!!

# Overview of Other Java Semaphore Methods

# Overview of Other Java Semaphore Methods

- There are many other Semaphore methods

| | |
|---|---|
| void | acquire(int permits) – Acquires # of permits from semaphore, blocking until all are available, or thread interrupted |
| void | acquireUninterruptibly(int permits) – Acquires # of permits from semaphore, blocking until all available |
| boolean | tryAcquire(int permits) – Acquires given # of permits from semaphore, only if all are available at the time of invocation |
| void | release(int permits) – Releases # of permits, returning them to semaphore |
| boolean | tryAcquire(long timeout, TimeUnit unit) – Acquires a permit from semaphore, if one is available within given waiting time & thread has not been interrupted |
| boolean | tryAcquire(int permits, long timeout, TimeUnit unit) – Acquires given # of permits from semaphore, if all available within given waiting time & current thread has not been interrupted |

# Overview of Other Java Semaphore Methods

- There are many other Semaphore methods
  - Some methods can acquire or release multiple permits at a time

| | |
|---|---|
| void | acquire(int permits) – Acquires # of permits from semaphore, blocking until all are available, or thread interrupted |
| void | acquireUninterruptibly(int permits) – Acquires # of permits from semaphore, blocking until all available |
| boolean | tryAcquire(int permits) – Acquires given # of permits from semaphore, only if all are available at the time of invocation |
| void | release(int permits) – Releases # of permits, returning them to semaphore |
| boolean | tryAcquire(long timeout, TimeUnit unit) – Acquires a permit from semaphore, if one is available within given waiting time & thread has not been interrupted |
| boolean | tryAcquire(int permits, long timeout, TimeUnit unit) – Acquires given # of permits from semaphore, if all available within given waiting time & current thread has not been interrupted |

# Overview of Other Java Semaphore Methods

- There are many other Semaphore methods
  - Some methods can acquire or release multiple permits at a time
  - Likewise, some of these methods use timeouts

| | |
|---|---|
| void | acquire(int permits) – Acquires # of permits from semaphore, blocking until all are available, or thread interrupted |
| void | acquireUninterruptibly(int permits) – Acquires # of permits from semaphore, blocking until all available |
| boolean | tryAcquire(int permits) – Acquires given # of permits from semaphore, only if all are available at the time of invocation |
| void | release(int permits) – Releases # of permits, returning them to semaphore |
| boolean | tryAcquire(long timeout, TimeUnit unit) – Acquires a permit from semaphore, if one is available within given waiting time & thread has not been interrupted |
| boolean | tryAcquire(int permits, long timeout, TimeUnit unit) – Acquires given # of permits from semaphore, if all available within given waiting time & current thread has not been interrupted |

Ironically, the timed tryAcquire() methods *do* honor the fairness setting, so they don't "barge"

# End of Java Semaphores (Part 2)