Structure & Functionality of Java Semaphore

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

• Understand the concept of semaphores
• Be aware of the two types of semaphores
• Note a human known use of semaphores
• Recognize the structure & functionality of Java Semaphore
Overview of the Java Semaphore Class
Overview of the Java Semaphore Class

- Implements a variant of counting semaphores

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

**Class Semaphore**

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

**All Implemented Interfaces:**

- Serializable

```java
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](docs.oracle.com/javase/8/docs/api/java/util/concurrent/Semaphore.html)
Overview of the Java Semaphore Class

• Implements a variant of counting semaphores

```java
class Semaphore
```

Semaphore doesn’t implement any synchronization-related interfaces

<table>
<thead>
<tr>
<th>Class Semaphore</th>
</tr>
</thead>
<tbody>
<tr>
<td>java.lang.Object</td>
</tr>
<tr>
<td>java.util.concurrent.Semaphore</td>
</tr>
</tbody>
</table>

All Implemented Interfaces:
- Serializable

```java
class Semaphore extends Object implements Object 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 the Java Semaphore Class

• Constructors create semaphore with a given # of permits

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

    ...

    public Semaphore (int permits) {
        ...
    }

    ...

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

    ...
```
Overview of the Java Semaphore Class

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

```java
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](https://stackoverflow.com/questions/7554839/how-and-why-can-a-semaphore-give-out-more-permits-than-it-was-initialized-with)
Overview of the Java Semaphore Class

• 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!!

```java
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 the Java Semaphore Class

- Applies the *Bridge* pattern

```java
public class Semaphore implements ...
{
...
}
```

Decouples an interface from its implementation(s) so fair & non-fair semantics can be supported uniformly

See [en.wikipedia.org/wiki/Bridge_pattern](en.wikipedia.org/wiki/Bridge_pattern)
Overview of the Java Semaphore Class

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

```java
public class Semaphore
    implements ...
{
    ...
    /** Performs sync mechanics */
    private final Sync sync;
```
Overview of the Java Semaphore Class

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer

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

  /** Performs sync mechanics */
  private final Sync sync;

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html](docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html)
Overview of the Java Semaphore Class

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

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

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

See gee.cs.oswego.edu/dl/papers/aqs.pdf
Overview of the Java Semaphore Class

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Inherits functionality from AbstractQueuedSynchronizer
- Defines NonfairSync & FairSync subclasses with non-FIFO & FIFO semantics

```java
public class Semaphore implements ... {
    private final Sync sync;

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

    static final class NonfairSync extends Sync { ...
    }

    static final class FairSync extends Sync { ...
    }

    See src/share/classes/java/util/concurrent/Semaphore.java
```
Overview of the Java Semaphore Class

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Constructor enables fair vs. non-fair semaphore acquisition model

```java
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 the Java Semaphore Class

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Constructor enables fair vs. non-fair semaphore acquisition model
- These models apply the same pattern used by ReentrantLock

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

See earlier lesson on “Java ReentrantLock”
Applies the *Bridge* pattern
• Locking handled by Sync Implementor hierarchy
• Constructor enables fair vs. non-fair semaphore acquisition model
• These models apply the same pattern used by ReentrantLock

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

Ensures strict “FIFO” fairness, at the expense of performance

Overview of the Java Semaphore Class

Ensures strict “FIFO” fairness, at the expense of performance
Overview of the Java Semaphore Class

• Applies the Bridge pattern  
  - Locking handled by Sync Implementor hierarchy  
• Constructor enables fair vs. non-fair semaphore acquisition model  
• These models apply the same pattern used by ReentrantLock

```
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
Applies the *Bridge* pattern

- Locking handled by Sync Implementor hierarchy
- Constructor enables fair vs. non-fair semaphore acquisition model
- These models apply the same pattern used by ReentrantLock

```java
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 the Java Semaphore Class

- Applies the *Bridge* pattern
- Locking handled by Sync Implementor hierarchy
- Constructor enables fair vs. non-fair semaphore acquisition model
- These models apply the same pattern used by ReentrantLock

```java
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);
    }

    ...

    FairSync is generally much slower than NonfairSync, so use it accordingly
```
Overview of the Java Semaphore Class

- 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 upcoming part on “Java Semaphore: Coordinating Threads"
End of Structure & Functionality of Java Semaphore