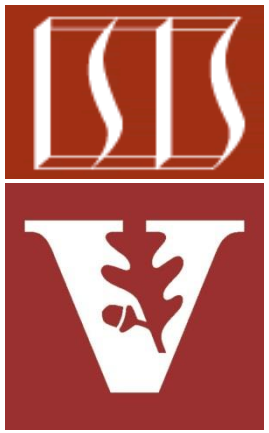
The Specific Notification Pattern: "Fair" Semaphore Semantics



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 *Specific Notification* pattern
- Be aware of the semantics of "fair" semaphores

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:

An Overview of Fair Semaphore Semantics

Overview of Fair Semaphore Semantics

- Threads calling acquire() on a "fair" semaphore obtain permits in "first-in, first-out" (FIFO) order

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 Fair Semaphore Semantics

- Threads calling acquire() on a "fair" semaphore obtain permits in "first-in, first-out" (FIFO) order
  - FIFO ordering applies to internal points of execution within semaphore methods

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 Fair Semaphore Semantics

- Threads calling acquire() on a "fair" semaphore obtain permits in "first-in, first-out" (FIFO) order
  - FIFO ordering applies to internal points of execution within semaphore methods
    - e.g., one thread can invoke acquire() *before* another, but reach the ordering point *after* the other

Class Semaphore

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

All Implemented Interfaces:

`Serializable`

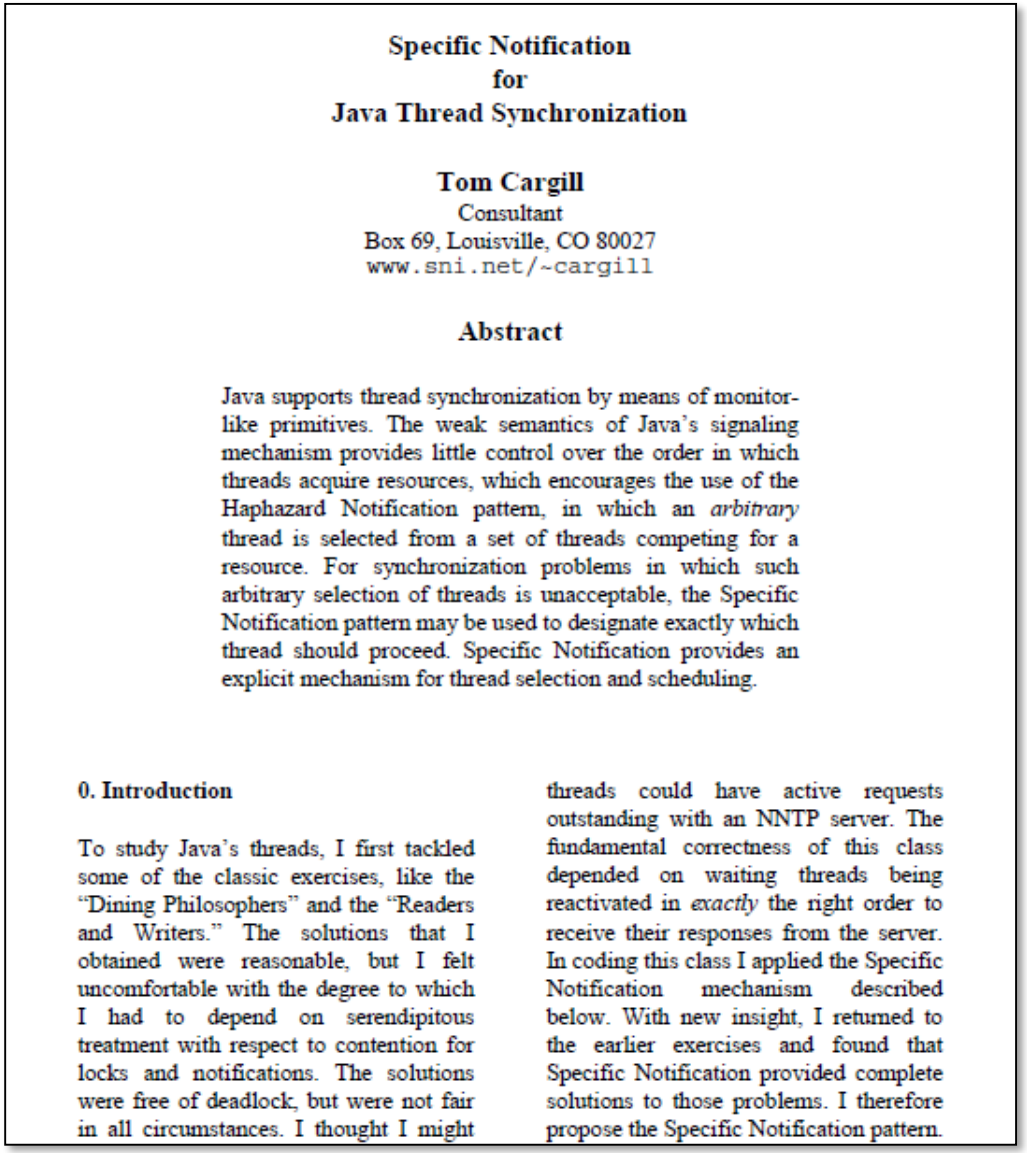
```
public class Semaphore
extends Object
implements Serializable
```

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 Fair Semaphore Semantics

- Threads calling acquire() on a "fair" semaphore obtain permits in "first-in, first-out" (FIFO) order
  - FIFO ordering applies to internal points of execution within semaphore methods
  - The *Specific Notification* pattern provides an effective model for implementing fair semaphore semantics



Specific Notification
for
Java Thread Synchronization

Tom Cargill
Consultant
Box 69, Louisville, CO 80027
www.sni.net/~cargill

Abstract

Java supports thread synchronization by means of monitor-like primitives. The weak semantics of Java's signaling mechanism provides little control over the order in which threads acquire resources, which encourages the use of the Haphazard Notification pattern, in which an *arbitrary* thread is selected from a set of threads competing for a resource. For synchronization problems in which such arbitrary selection of threads is unacceptable, the Specific Notification pattern may be used to designate exactly which thread should proceed. Specific Notification provides an explicit mechanism for thread selection and scheduling.

0. Introduction

To study Java's threads, I first tackled some of the classic exercises, like the "Dining Philosophers" and the "Readers and Writers." The solutions that I obtained were reasonable, but I felt uncomfortable with the degree to which I had to depend on serendipitous treatment with respect to contention for locks and notifications. The solutions were free of deadlock, but were not fair in all circumstances. I thought I might threads could have active requests outstanding with an NNTP server. The fundamental correctness of this class depended on waiting threads being reactivated in *exactly* the right order to receive their responses from the server. In coding this class I applied the Specific Notification mechanism described below. With new insight, I returned to the earlier exercises and found that Specific Notification provided complete solutions to those problems. I therefore propose the Specific Notification pattern.

See www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf (especially Listing 3)

End of the Specific Notification Pattern: Fair Semaphore Semantics