Java Semaphore
Usage Considerations

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 Module

• 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
• Know the key methods defined by the Java Semaphore class
• Learn how Java semaphores enable multiple threads to
  • Mediate access to a limited number of shared resources
  • Coordinate the order in which operations occur
• Appreciate Java Semaphore usage considerations
Java Semaphore
Usage Considerations
Java Semaphore Usage Considerations

- Semaphore is more flexible than the more simple Java synchronizers

**Synchronized Statements**

Another way to create synchronized code is with *synchronized statements*. Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

```java
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

**Class ReentrantLock**

- `java.lang.Object`
- `java.util.concurrent.locks.ReentrantLock`

**All Implemented Interfaces:**
- `Serializable`, `Lock`
Java Semaphore Usage Considerations

- Semaphore is more flexible than the more simple Java synchronizers, e.g.
  - Can atomically acquire & release multiple permits with 1 operation
Java Semaphore Usage Considerations

- Semaphore is more flexible than the more simple Java synchronizers, e.g.
  - Can atomically acquire & release multiple permits with 1 operation
  - Its acquire() & release() methods need not be fully bracketed

```java
Semaphore

Run()

Print("ping")

Run()

Print("pong")
```
Java Semaphore Usage Considerations

- Semaphore is more flexible than the more simple Java synchronizers, e.g.
  - Can atomically acquire & release multiple permits with 1 operation
  - Its acquire() & release() methods need not be fully bracketed

Naturally, this flexibility comes at some additional cost in performance
Java Semaphore Usage Considerations

• When a semaphore is used for a resource pool, it tracks the # of free resources
Java Semaphore Usage Considerations

- When a semaphore is used for a resource pool, it tracks the # of free resources
- However, it does not track *which* resources are free
• When a semaphore is used for a resource pool, it tracks the # of free resources
  • However, it does not track which resources are free
• Other mechanisms may be needed to select a particular free resource
  • e.g., a List, HashMap, etc.

See docs.oracle.com/javase/8/docs/technotes/guides/collections
Java Semaphore Usage Considerations

- When a semaphore is used for a resource pool, it tracks the # of free resources
  - However, it does not track which resources are free
- Other collections may be needed to select a particular free resource
  - e.g., a List, HashMap, etc.

These collections require synchronizers to ensure thread-safety
Java Semaphore Usage Considerations

• Semaphores can be tedious & error-prone to program due to common traps & pitfalls
Java Semaphore Usage Considerations

- Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Holding a semaphore for a long time without needing it

```java
Semaphore semaphore = new Semaphore(1);

void someMethod() {
    semaphore.acquire();

    try {
        for (;;) {
            // Do something not involving semaphore
        }
    } finally {
        semaphore.release();
    }
}
```

Other thread(s) won’t be able to acquire the semaphore in a timely manner
Java Semaphore Usage Considerations

• Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  • Holding a semaphore for a long time without needing it
  • Releasing a semaphore more times than needed

Semaphore semaphore = new Semaphore(1);

void someMethod() {
  semaphore.acquire();  // 0

  ...  // code...

  semaphore.release();
  semaphore.release();  // 3
  semaphore.release();
}

These extra calls to release() will allow too many threads to acquire the semaphore
Java Semaphore Usage Considerations

• Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Holding a semaphore for a long time without needing it
  - Releasing a semaphore more times than needed
  - Prematurely releasing a semaphore that should have been held

```java
Semaphore semaphore = new Semaphore(count);

Resource acquireResource() {
    semaphore.acquire();
    // Obtain relevant // resource from the pool
    semaphore.release();
    return resource;
}

void releaseResource
    (Resource resource) {
    // Return the resource to // the pool.
    semaphore.release();
}
```

This semaphore should have been held for the duration of the returned resource’s utilization
Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.

- Holding a semaphore for a long time without needing it
- Releasing a semaphore more times than needed
- Prematurely releasing a semaphore that should have been held
- Acquiring a semaphore & forgetting to release it

```java
Semaphore semaphore = new Semaphore(1);
void someMethod() {
    semaphore.acquire();
    ... // Critical section
    return;
}
```

*The semaphore may be locked indefinitely!*
Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
- Holding a semaphore for a long time without needing it
- Releasing a semaphore more times than needed
- Prematurely releasing a semaphore that should have been held
- Acquiring a semaphore & forgetting to release it

```java
Semaphore semaphore = new Semaphore(1);

void someMethod() {
    semaphore.acquire();
    try {
        ... // Critical section
        return;
    } finally {
        semaphore.release();
    }
}
```

Use the try/finally idiom to ensure a fully-bracketed semaphore is always released, even if exceptions occur

See docs.oracle.com/javase/tutorial/essential/exceptions/finally.html
Semaphores are rather limited synchronizers that don’t scale to complex coordination use cases.
Java Semaphore Usage Considerations

- Semaphores are rather limited synchronizers that don’t scale to complex coordination use cases.
- Java ConditionObjects may be a better choice for complex coordination use-cases.

### Class

**AbstractQueuedSynchronizer.ConditionObject**

```java
java.lang.Object
    java.util.concurrent.locks.AbstractQueuedSynchronizer.ConditionObject

All Implemented Interfaces:
    Serializable, Condition

Enclosing class:
    AbstractQueuedSynchronizer

public class AbstractQueuedSynchronizer.ConditionObject
    extends Object
    implements Condition, Serializable

Condition implementation for a AbstractQueuedSynchronizer serving as the basis of a Lock implementation.

Method documentation for this class describes mechanics, not behavioral specifications from the point of view of Lock and Condition users. Exported versions of this class will in general need to be accompanied by documentation describing condition semantics that rely on those of the associated AbstractQueuedSynchronizer.

This class is Serializable, but all fields are transient, so deserialized conditions have no waiters.

See upcoming lessons on "Java ConditionObject"
End of Java Semaphore Usage Considerations