Learning Objectives in this Part of the Module

• Appreciate the concept of semaphores
• Recognize the two types of semaphores
• Know a human known use of semaphores
• Understand the structure & functionality of Java semaphores
• Recognize how Java semaphores enable multiple threads to
  • Mediate access to a limited number of shared resources
Learning Objectives in this Part of the Module

- Appreciate the concept of semaphores
- Recognize the two types of semaphores
- Know a human known use of semaphores
- Understand the structure & functionality of Java semaphores
- Recognize how Java semaphores enable multiple threads to
  - Mediate access to a limited number of shared resources
- Coordinate the order in which operations occur
Applying a Java Semaphore to Mediate Access
Applying a Java Semaphore to Mediate Access

- This Android app shows how a Java semaphore can be used to limit the number of Middle-Earth beings who can gaze into Palantiri concurrently.

See en.wikipedia.org/wiki/Palantir

Each being is implemented to run in a separate thread.
Applying a Java Semaphore to Mediate Access

- This Android app shows how a Java semaphore can be used to limit the number of Middle-Earth beings who can gaze into Palantiri concurrently.
- The app can be configured to restrict the number of being threads that concurrently gaze into palantiri.

\[\text{e.g., limit use to two palantiri on a quad-core device to ensure system responsiveness}\]
Applying a Java Semaphore to Mediate Access

• This Android app shows how a Java semaphore can be used to limit the number of Middle-Earth beings who can gaze into Palantiri concurrently.

• The app can be configured to restrict the number of being threads that concurrently gaze into palantiri.

• A permit must be acquired from a semaphore before a being can gaze.

Acquiring a permit atomically decrements the permit count.
Applying a Java Semaphore to Mediate Access

- This Android app shows how a Java semaphore can be used to limit the number of Middle-Earth beings who can gaze into Palantiri concurrently.

- The app can be configured to restrict the number of being threads that concurrently gaze into Palantiri.

- A permit must be acquired from a semaphore before a being can gaze.

All available permits are now in use.
Applying a Java Semaphore to Mediate Access

- This Android app shows how a Java semaphore can be used to limit the number of Middle-Earth beings who can gaze into Palantiri concurrently.
  - The app can be configured to restrict the number of being threads that concurrently gaze into palantiri.
  - A permit must be acquired from a semaphore before a being can gaze.
  - Other being threads must block until a permit is available.
This example “fully brackets” the acquiring & releasing of permits, i.e., the thread that acquires a semaphore is the *same* as the one that releases it.
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
: Palantiri Presenter
start()

: BeingRunnables
start()
start()
start()

: Palantir
run()
run()
run()
run()

: mPalantiriManager
run()
r = acquire()
r = acquire()
r = acquire()
r = acquire()
r = acquire()
release(r)
release(r)
release(r)
release(r)
release(r)
```

Applying a Java Semaphore to Mediate Access
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

![UML sequence diagram with start() method and Palantiri Presenter transition]
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

![UML sequence diagram](image_url)
Applying a Java Semaphore to Mediate Access

• UML sequence diagram for this app

start()

: Palantiri Presenter →

: BeingRunnables →

start()

run()
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```java
start()
run()
start()
start()
start()

mPalantiriManager : PalantiriManager

: Palantiri Presenter
: BeingRunnables

p = acquire()
p = acquire()
p = acquire()
```
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
: Palantiri Presenter

start() ->
start() ->
start() ->
```

```
: BeingRunnables

run() ->
```

```
mPalantiriManager

: PalantiriManager

p = acquire() ->
```

```
p = acquire() ->
```

```
p = acquire() ->
```

```
p = acquire() ->
```

```
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```java
start()
run()
p = acquire()
p.gaze()
p = acquire()
p = acquire()
p.gaze()
p = acquire()
p = acquire()
p = acquire()
p = acquire()
```

: Palantiri Presenter
: BeingRunnables
p : Palantir
mPalantiriManager : PalantiriManager
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

: Palantiri Presenter

: BeingRunnables

p : Palantir

mPalantiriManager : PalantiriManager

start()

start()

start()

run()

p.gaze()

p = acquire()

release(p)

p.gaze()

p = acquire()

release(p)

p = acquire()
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
: Palantiri
  Presenter

: BeingRunnables

p : Palantir

mPalantiriManager : PalantiriManager

applying

run()

start()

start()

start()

start()

run()

p = acquire()

p.gaze()

release(p)

run()

p = acquire()

p.gaze()

release(p)

run()

p = acquire()

p.gaze()

release(p)
```
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
start()
start()
start()
start()

run()
p = acquire()
p.gaze()
run()
release(p)
p = acquire()
p.gaze()
run()
release(p)
p = acquire()
p.gaze()
run()
release(p)
p = acquire()
p.gaze()
run()
release(p)
```
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
start()
<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
start()               
|                      |
|                      |
start()               
|                      |
|                      |
run()                 
|                      |
|                      |
p = acquire()         
|                      |
|                      |
p.gaze()              
|                      |
|                      |
runtime              
|                      |
|                      |
run()                 
|                      |
|                      |
p = acquire()         
|                      |
|                      |
p.gaze()              
|                      |
|                      |
runtime              
|                      |
|                      |
run()                 
|                      |
|                      |
p = acquire()         
|                      |
|                      |
p.gaze()              
|                      |
|                      |
runtime              
|                      |
|                      |
run()                 
|                      |
|                      |
p = acquire()         
|                      |
|                      |
p.gaze()              
|                      |
|                      |
runtime              
|                      |
|                      |
run()                 
|                      |
|                      |
p = acquire()         
|                      |
|                      |
p.gaze()              
|                      |
|                      |
runtime              
|                      |
|                      |
```
Applying Java Semaphores to Coordinate Threads
Applying Java Semaphores to Coordinate Threads

- The Android ping-pong app coordinates thread interactions via various Java synchronizers, including Java semaphores.
- i.e., the threads alternate printing “ping” & “pong” on the display.

See [github.com/douglascairnschmidt/POSA/tree/master/ex/M3/PingPong](github.com/douglascairnschmidt/POSA/tree/master/ex/M3/PingPong)
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

: Play → PingPongThread
pong : PingPongThread
ping : PingPongThread
1stSema: Semaphore
2ndSema: Semaphore

run() → new()
start() → run()
acquire() → release()
println() → println()
join()
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

This class starts two threads, ping & pong, that alternate printing "Ping" and "Pong", respectively, on the display.
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

The PingPongThread class implements the core ping-pong algorithm, but defers synchronization aspects to subclasses via the *Template Method* pattern.
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

This app can be configured to use a pair of semaphores that coordinate the order in which the “ping” & “pong” threads are called to play ping-pong.
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

This example does *not* “fully bracket” acquiring & releasing permits, i.e., the thread that acquires a semaphore is different from the thread that releases it.
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

The PlayPingPongThread joins with the ping & pong threads once they are finished.
Semaphore Usage Considerations
Semaphore Usage Considerations

• Semaphore is more flexible than 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);
}
```

In this example, the addName method needs to synchronize changes to lastName and nameCount, but also needs to avoid synchronizing invocations of other objects' methods. (Invoking other objects' methods from synchronized code can create problems that are described in the section on Liveness.) Without synchronized statements, there would have to be a separate, unsynchronized method for the sole purpose of invoking nameList.add.

Synchronized statements are also useful for improving concurrency with fine-grained synchronization. Suppose, for example, class MsLunch has two instance fields, c1 and c2, that are never used together. All updates of these fields must be synchronized, but there's no reason to prevent an update of c1 from being interleaved with an update of c2 — and doing so reduces concurrency by creating unnecessary blocking. Instead of using synchronized methods or otherwise using the lock associated with this, we create two objects solely to provide locks.

Class ReentrantLock

```java
import java.lang.Object;
import java.util.concurrent.locks.ReentrantLock

public class ReentrantLock extends Object
    implements Lock, Serializable
```

A reentrant mutual exclusion Lock with the same basic behavior and semantics as the implicit monitor lock accessed using synchronized methods and statements, but with extended capabilities.

A ReentrantLock is owned by the thread last successfully locking, but not yet unlocking it. A thread invoking lock will return, successfully acquiring the lock, when the lock is not owned by another thread. The method will return immediately if the current thread already owns the lock. This can be checked using methods isHeldByCurrentThread(), and getHoldCount().

The constructor for this class accepts an optional fairness parameter. When set true, under contention, locks favor granting access to the longest-waiting thread. Otherwise this lock does not guarantee any particular access order. Programs using fair locks accessed by many threads may display lower overall throughput (i.e., are slower; often much slower) than those using the default setting, but have smaller variances in times to obtain locks and guarantee lack of starvation. Note however, that fairness of locks does
Semaphore Usage Considerations

- Semaphore is more flexible than simple Java synchronizers, e.g.
- Can acquire & release multiple permits in a single operation
Semaphore Usage Considerations

- Semaphore is more flexible than simple Java synchronizers, e.g.
  - Can acquire & release multiple permits in a single operation
  - Its acquire() & release() methods need not be fully bracketed
Semaphore Usage Considerations

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

• When a semaphore is used for a resource pool, it tracks # of free resources
• However, it does not track which resources are free
Semaphore Usage Considerations

- When a semaphore is used for a resource pool, it tracks the number of free resources.
  - However, it does not track which resources are free.
- Other mechanisms may be needed to select a particular free resource.
Semaphore Usage Considerations

- When a semaphore is used for a resource pool, it tracks # 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, other Semaphores, etc.
Semaphore Usage Considerations

- Semaphores can be tedious & error-prone to program due to common traps & pitfalls
Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.

• Acquiring a semaphore & forgetting to release it

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

void someMethod() {
    semaphore.acquire();
    ...
    // Critical section
    return;
}
```
Semaphore Usage Considerations

- Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  
- Acquiring a semaphore & forgetting to release it

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

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

It’s a good idea to use the try/finally idiom to ensure a Semaphore is always released, even if exceptions occur.
• Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  • Acquiring a semaphore & forgetting to release it
  • Holding a semaphore for a long time without needing it

Semaphore Usage Considerations

Semaphore semaphore =
new Semaphore(1);

void someMethod() {
    semaphore.acquire();

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

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

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

void someMethod() {
    semaphore.acquire();
    ...
    semaphore.release();
    semaphore.release();
    semaphore.release();
}
```
End of Java Semaphores (Part 3)
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 Semaphores
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

```java
public class Semaphore
{
    ...

    Semaphore doesn’t implement any synchronization-related interfaces
}
```

**Class Semaphore**

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

All Implemented Interfaces:
    Serializable

```java
public 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 Java Semaphores

- Applies the *Bridge* pattern

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

*Decouples its interface from its implementation so fair & non-fair semantics can be supported uniformly.*

See [en.wikipedia.org/wiki/Bridge_pattern](en.wikipedia.org/wiki/Bridge_pattern)
Overview of Java Semaphores

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

```java
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

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

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

See [docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html](https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html)
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

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

    ... 

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

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

    ... 

The Semaphore fair & non-fair models are like those used by the "Java ReentrantLock"
Applies the *Bridge* pattern

- Locking handled by Sync Implementor hierarchy
- Reuses functionality from AbstractQueuedSynchronizer
- Optionally implement fair or non-fair lock acquisition model
- Constructors create semaphore with a given # of permits

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

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

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

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

See stackoverflow.com/questions/7554839/how-and-why-can-a-semaphore-give-out-more-permits-than-it-was-initialized-with
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
- 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 ... {
    ...
    public Semaphore
        (int permits) {
        sync = new
            NonfairSync(permits);
    }
    public Semaphore
        (int permits,
        boolean fair) {
        sync = fair
            ? new FairSync(permits)
            : new NonfairSync(permits);
    }
    ...
```

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

- 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

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

    ...

    public void acquire() { ... }

    public void acquireUninterruptibly()
    { ... }

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

    public void release() { ... }

    ...

These methods all simply forward to their implementor methods, most of which are inherited from the AbstractQueuedSynchronizer framework.
Overview of Key Java Semaphore Methods

• Its key methods acquire & release the semaphore

• acquire() obtains a permit from the semaphore

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

    ... 

    public void acquire() {
        sync.
        acquireSharedInterruptibly(1);
    }

    ... 
```
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts

```java
public class Semaphore implements ...
{
    ...
    public void acquireUninterruptibly()
    {
        sync.acquireShared(1)
    }
    ...
}
```
Overview of Key Java Semaphore Methods

- Its key methods acquire & release the semaphore
  - `acquire()` obtains a permit from the semaphore
  - `acquireUninterruptibly()` ignores interrupts
  - `tryAcquire()` obtains a permit if it’s available at invocation time

```java
public class Semaphore implements ... {
    ...
    public boolean tryAcquire() {
        ...
        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() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if it’s available at invocation time
  - Release() returns a permit, increasing number 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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void acquire(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all are available, or thread interrupted</td>
</tr>
<tr>
<td><code>void acquireUninterruptibly(int permits)</code></td>
<td>Acquires # of permits from semaphore, blocking until all available</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits)</code></td>
<td>Acquires given # of permits from semaphore, only if all are available at the time of invocation</td>
</tr>
<tr>
<td><code>void release(int permits)</code></td>
<td>Releases # of permits, returning them to semaphore</td>
</tr>
<tr>
<td><code>boolean tryAcquire(long timeout, TimeUnit unit)</code></td>
<td>Acquires a permit from semaphore, if one is available within given waiting time &amp; thread has not been interrupted</td>
</tr>
<tr>
<td><code>boolean tryAcquire(int permits, long timeout, TimeUnit unit)</code></td>
<td>Acquires given # of permits from semaphore, if all available within given waiting time &amp; current thread has not been interrupted</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>acquire(int permits)</code></td>
<td>Acquires <code>#</code> of permits from semaphore, blocking until all are available, or thread interrupted</td>
</tr>
<tr>
<td><code>acquireUninterruptibly(int permits)</code></td>
<td>Acquires <code>#</code> of permits from semaphore, blocking until all available</td>
</tr>
<tr>
<td><code>tryAcquire(int permits)</code></td>
<td>Acquires given <code>#</code> of permits from semaphore, only if all are available at the time of invocation</td>
</tr>
<tr>
<td><code>release(int permits)</code></td>
<td>Releases <code>#</code> of permits, returning them to semaphore</td>
</tr>
<tr>
<td><code>tryAcquire(long timeout, TimeUnit unit)</code></td>
<td>Acquires a permit from semaphore, if one is available within given waiting time &amp; thread has not been interrupted</td>
</tr>
<tr>
<td><code>tryAcquire(int permits, long timeout, TimeUnit unit)</code></td>
<td>Acquires given <code>#</code> of permits from semaphore, if all available within given waiting time &amp; current thread has not been interrupted</td>
</tr>
</tbody>
</table>

Timed `tryAcquire()` methods do honor the fairness setting, so they don’t “barge”
End of Java Semaphores
(Part 2)