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 Semaphore & its methods

• Recognize how Java semaphores enable multiple threads to
  • Mediate access to a limited # 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 Semaphore & its methods
- Recognize how Java semaphores enable multiple threads to
  - Mediate access to a limited # 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
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.

*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 show how an Java semaphore can be used to limit the # of Middle-Earth beings who can gaze into Palantiri concurrently.
- The app can be configured to restrict the # 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.
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.

- When a permit is released back to the semaphore, another thread can acquire it and proceed.

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

: BeingRunnables

: Palantir

mPalantiriManager
: PalantiriManager

start()
run()
start()
start()

r = acquire()
r.gaze()
release(r)

r = acquire()
r.gaze()
release(r)

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

- UML sequence diagram for this app

: Palantiri Presenter

start()
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
\text{: Palantiri Presenter} \rightarrow \text{: BeingRunnables}
```

start()
Applying a Java Semaphore to Mediate Access:

- UML sequence diagram for this app

Diagram showing:
- Palantiri Presenter
- BeingRunnable states

Sequence:
- start()
- run()
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```plaintext
mPalantiriManager : PalantiriManager

: Palantiri

Presenter

: BeingRunnables

run()

p = acquire()

run()

p = acquire()

run()

p = acquire()
```

start()
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()
start()

run()
run()
p = acquire()
p = acquire()
p = acquire()

:mPalantiriManager : PalantiriManager

: PalantiriPresenter

: BeingRunnables
```
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
: Palantiri
  Presenter

: BeingRunnables

p : Palantir

mPalantiriManager : PalantiriManager
```

start()
start()
start()
start()

run()
p = acquire()
p = acquire()
p = acquire()
p = acquire()
p = acquire()

run()
p.gaze()
p.gaze()
p.gaze()
p.gaze()
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

```
\[\text{start()}
\text{start()}
\text{start()}
\text{start()}
\text{run()}
\text{p.gaze()}
\text{p = acquire()}
\text{run()}
\text{p.gaze()}
\text{p = acquire()}
\text{run()}
\text{p.gaze()}
\text{p = acquire()}
\text{run()}
\text{p.gaze()}
\text{p = acquire()}
\text{run()}
\text{p.gaze()}
\text{p = acquire()}
\text{release(p)}
\text{release(p)}
\text{release(p)}
\text{release(p)}
\]
Applying a Java Semaphore to Mediate Access

- UML sequence diagram for this app

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

: Palantiri Presenter

: BeingRunnables

p:

mPalantiriManager :

: Palantir

: PalantiriManager

BeingRunnables:

Palantiri:

Presenter:

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

- UML sequence diagram for this app

```
start()      mPalantiriManager : PalantiriManager
run()        BeingRunnables : Palantiri
start()      p : Palantir
start()      p.gaze()
run()        p = acquire()
run()        release(p)
p.gaze()     p = acquire()
p.gaze()     release(p)
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()
p = acquire()
p.gaze()
run()
p = acquire()
p.gaze()
run()
p = acquire()
p.gaze()
run()
p = acquire()
p.gaze()
run()
p = acquire()
p.gaze()
run()
p = acquire()
p.gaze()
release(p)
release(p)
release(p)
release(p)
```

: Palantiri Presenter
: BeingRunnables
p : Palantir
mPalantiriManager : PalantiriManager
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/douglascraigschmidt/POSA/tree/master/ex/M3/PingPong](https://github.com/douglascraigschmidt/POSA/tree/master/ex/M3/PingPong)
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

```
Applying Java Semaphores to Coordinate Threads

play : Play
    join()

pong : PingPongThread
    run()
    start()
    run()
    acquire()
    release()
    println()

ping : PingPongThread
    run()
    start()
    run()
    acquire()
    release()
    println()

pingSem : Semaphore
pongSem : Semaphore
```
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

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.

```java
Semaphore pingSem = new Semaphore(1);
Semaphore pongSem = new Semaphore(0);
```
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 acquiring a semaphore is different from the thread releasing it!
Applying Java Semaphores to Coordinate Threads

- UML sequence diagram for the ping-pong app

```java
private final Semaphore mMine;
private final Semaphore mOther;
...

protected void acquire() { mMine.acquire(); }

protected void release() { mOther.release(); }
```

This example does not “fully bracket” acquiring & releasing permits, i.e., the thread acquiring a semaphore is different from the thread releasing 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.

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

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

```java
pong : PingPongThread
```

```java
ping : PingPongThread
```

```java
pingSem : Semaphore
pongSem : Semaphore
```
End of Java Semaphores
(Part 3)
Java Semaphore (Part 4)

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

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

All Implemented Interfaces:

- Serializable, Lock

```java
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
Java Semaphore Usage Considerations

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

- When a semaphore is used for a resource pool, it tracks # of free resources.
Java 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
Java 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, 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 number 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.

These mechanisms 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();
    }
}
```
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 the semaphore more times than needed

```java
Semaphore semaphore = new Semaphore(1);
void someMethod() {
    semaphore.acquire();
    ...
    semaphore.release();
    semaphore.release();
    semaphore.release();
}
```
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 the semaphore more times than needed
  - Acquiring a semaphore & forgetting to release it

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

void someMethod() {
    semaphore.acquire();
    ... // Critical section
    return;
}
```
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 the semaphore more times than needed
  - 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.

See docs.oracle.com/javase/tutorial/essential/exceptions/finally.html
End of Java Semaphores
(Part 4)