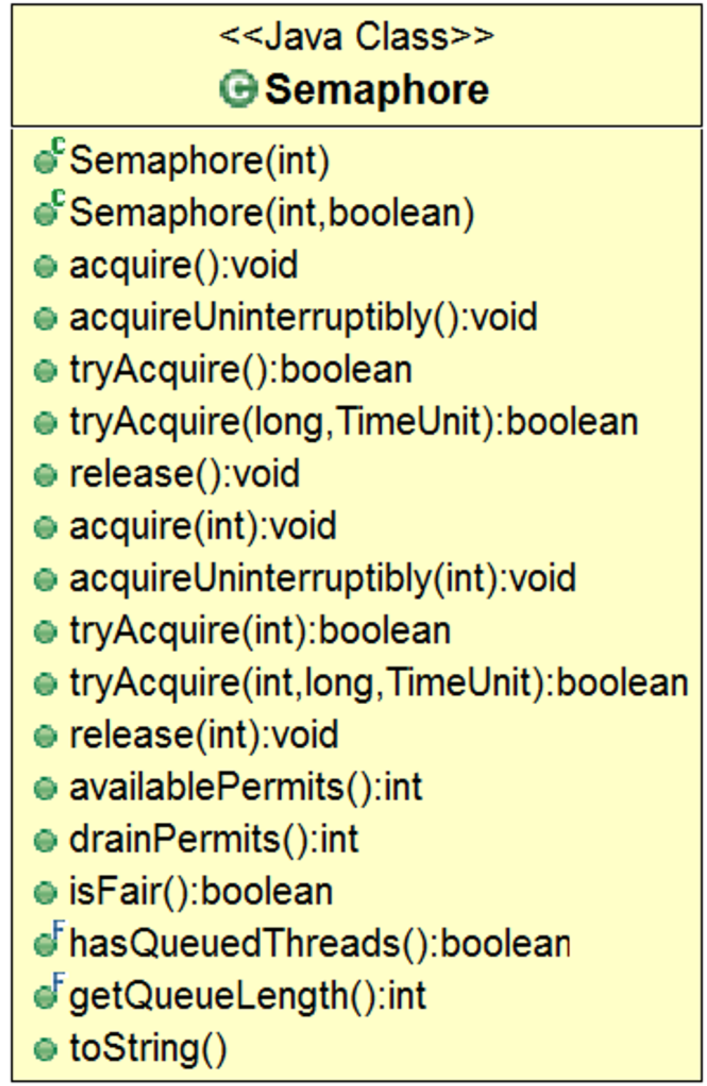
Java Concurrency: Java Semaphore

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 Module

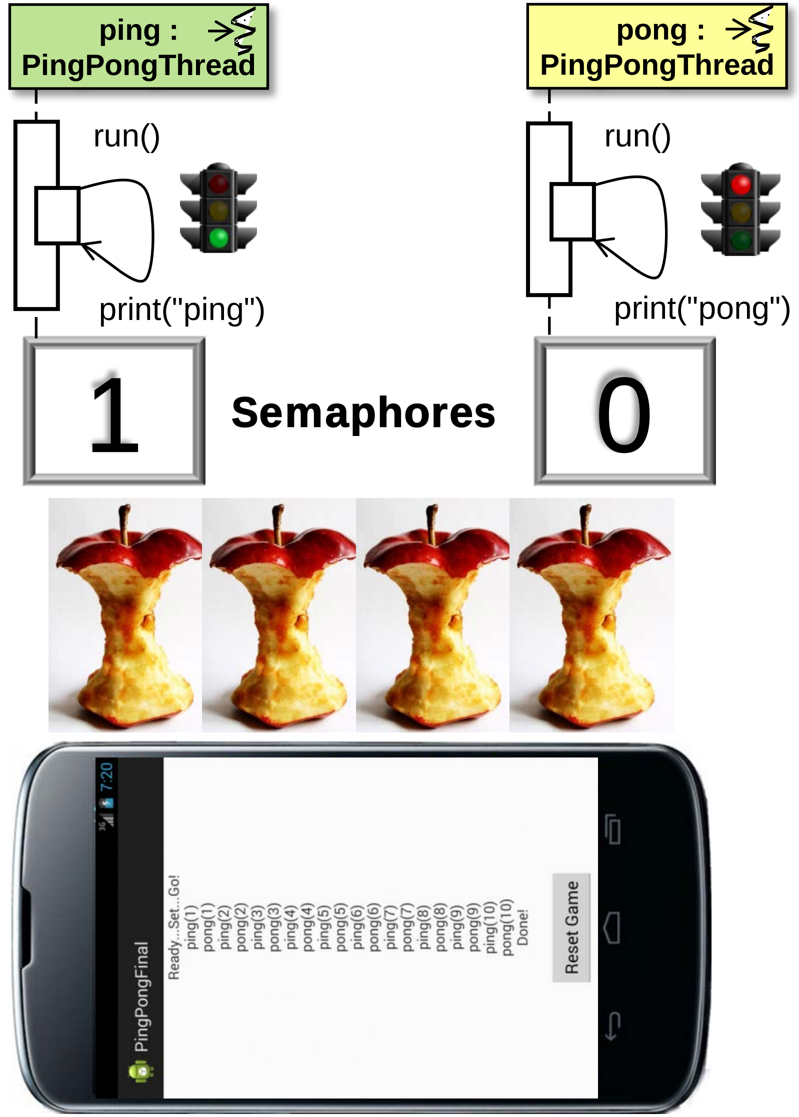
- Understand the structure & functionality of Java Semaphores



Learning Objectives in this Part of the Module

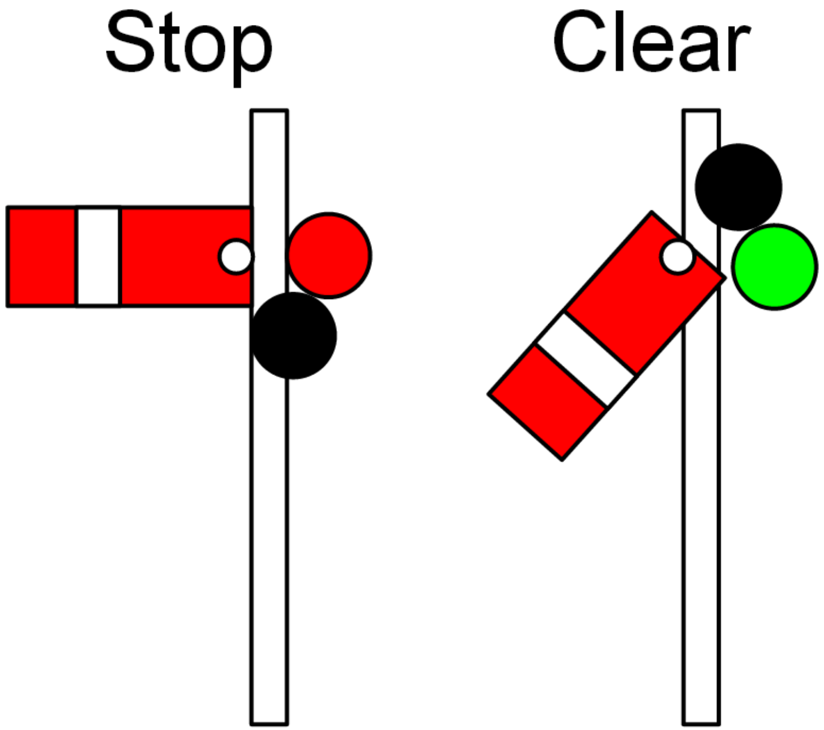
- Understand the structure & functionality of Java Semaphores
- Recognize how Java Semaphores enable multiple threads to
  - Mediate access to a limited number of shared resources

- 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



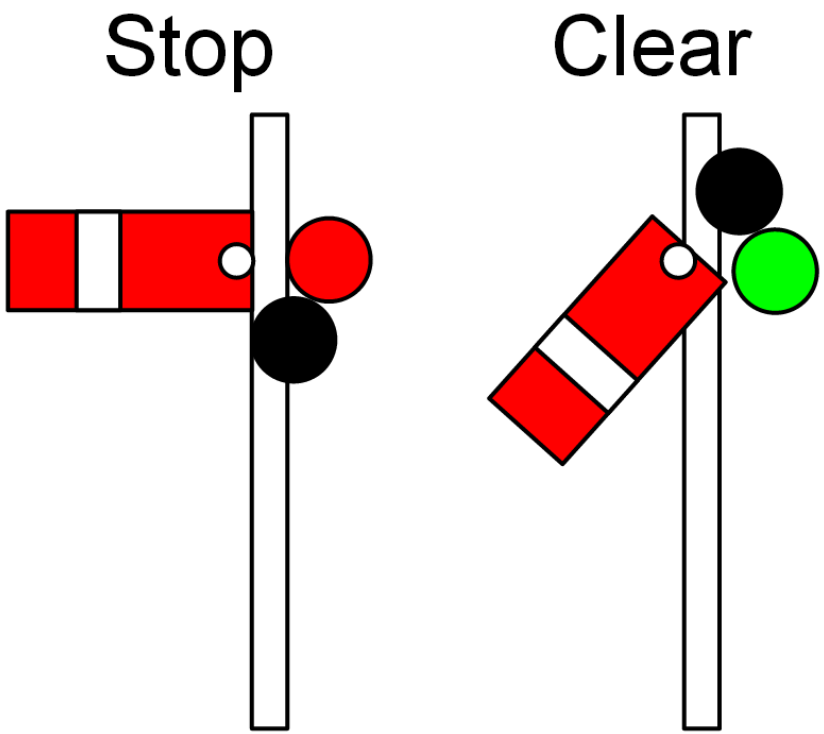
Overview of Semaphores

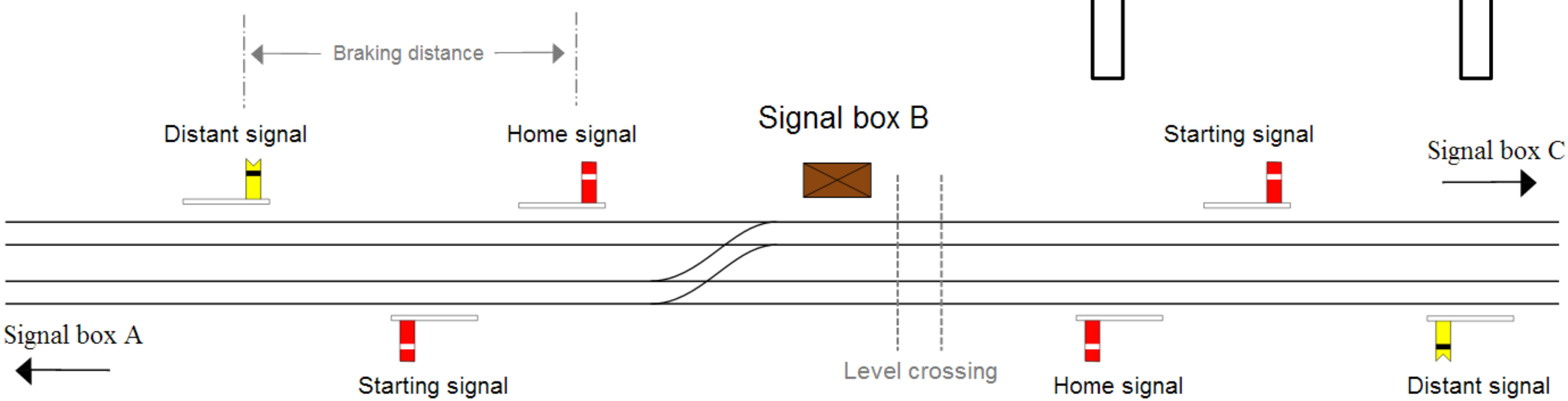
- A semaphore can be atomically incremented & decremented to control access to a shared resource



Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
  - e.g., used to control access to a shared railroad track

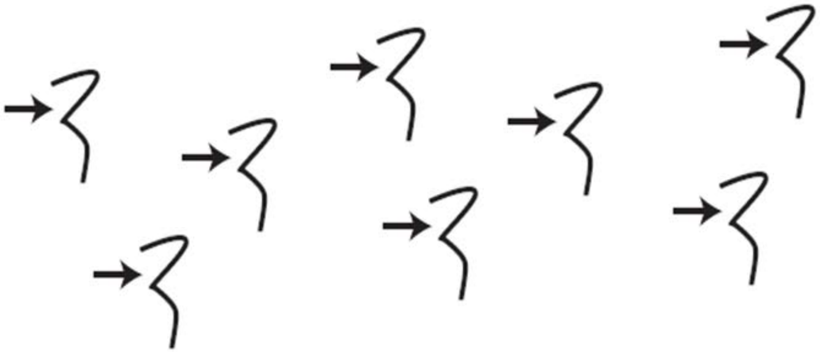




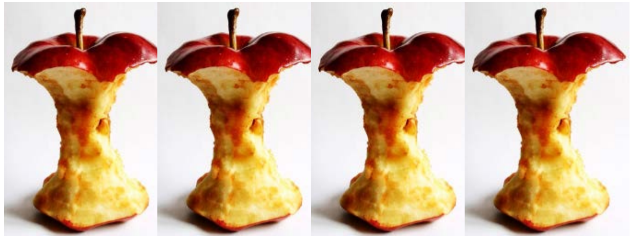
See en.wikipedia.org/wiki/Railway_semaphore_signal

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads



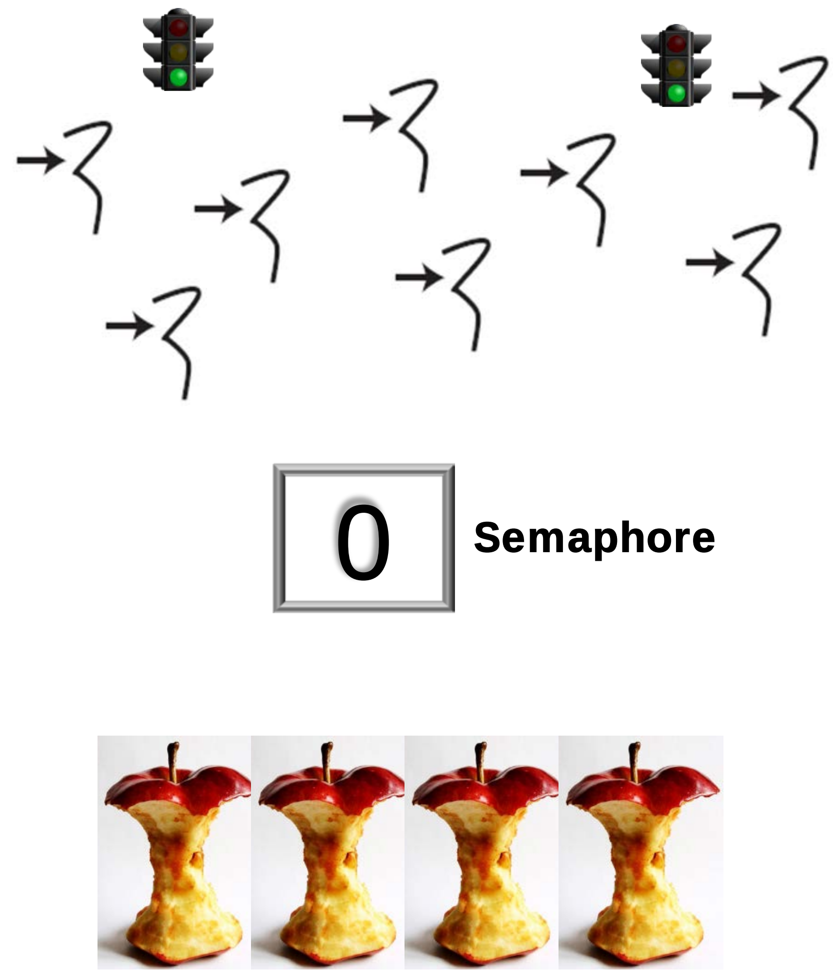
Semaphore



See en.wikipedia.org/wiki/Semaphore_(programming)

Overview of Semaphores

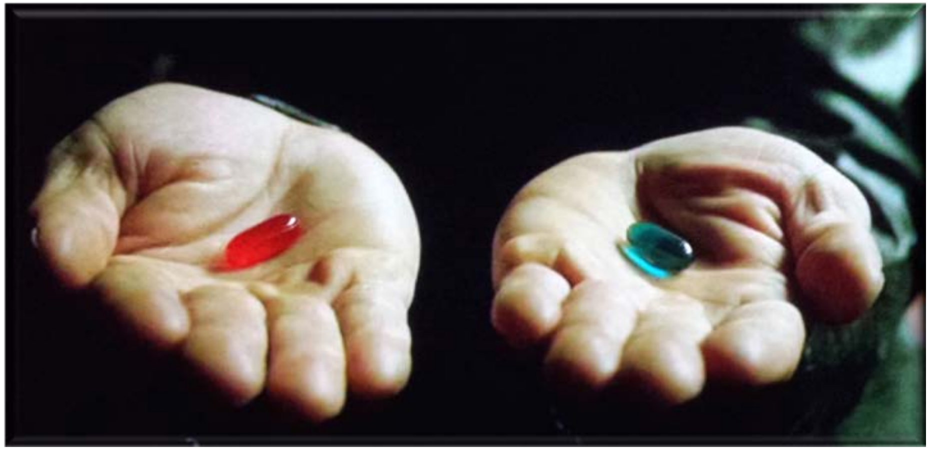
- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
  - e.g., limit the number of resources devoted to a particular task



See upcoming part on “A Concurrent Resource Management Application”

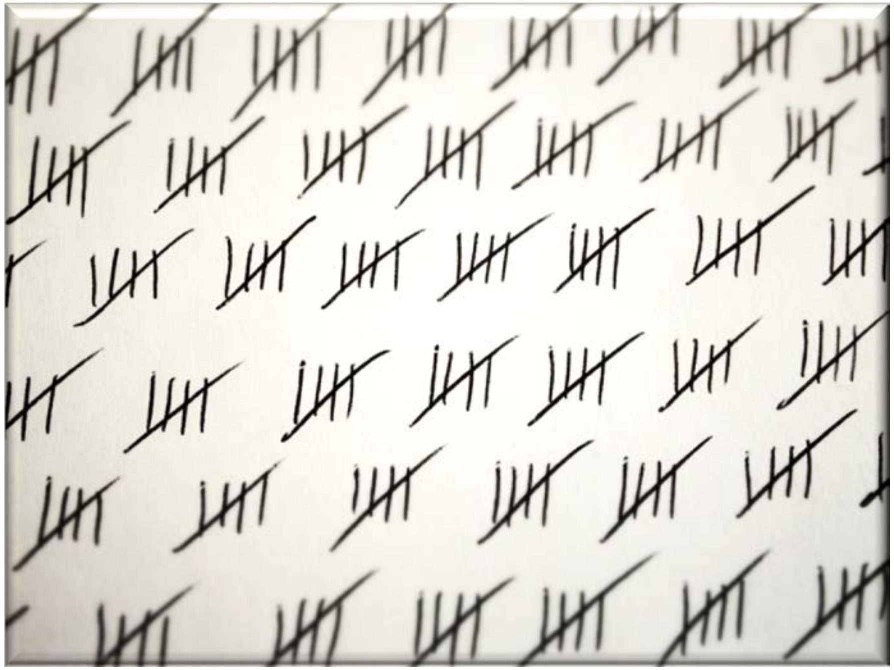
Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores



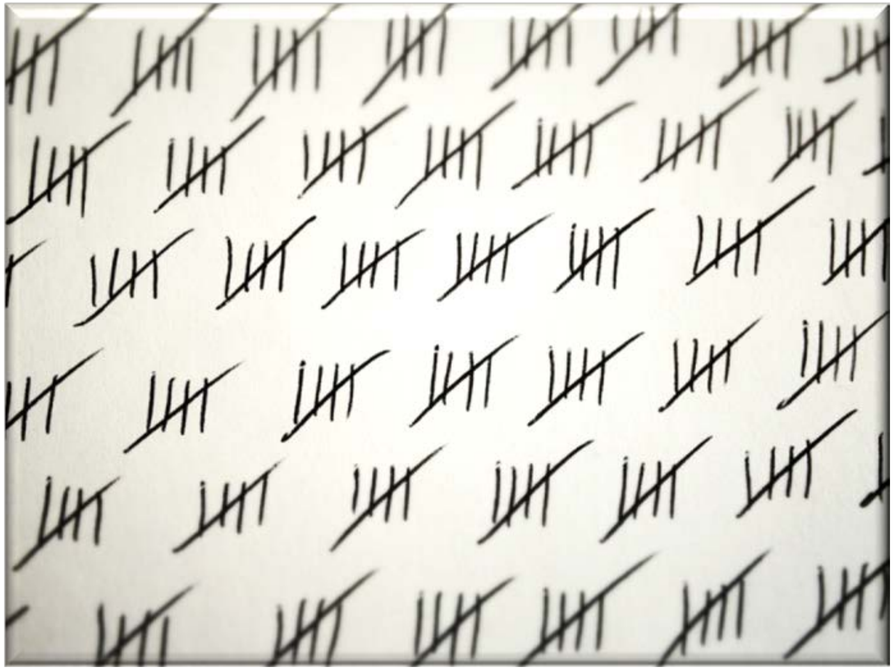
Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter (N) with precise meaning

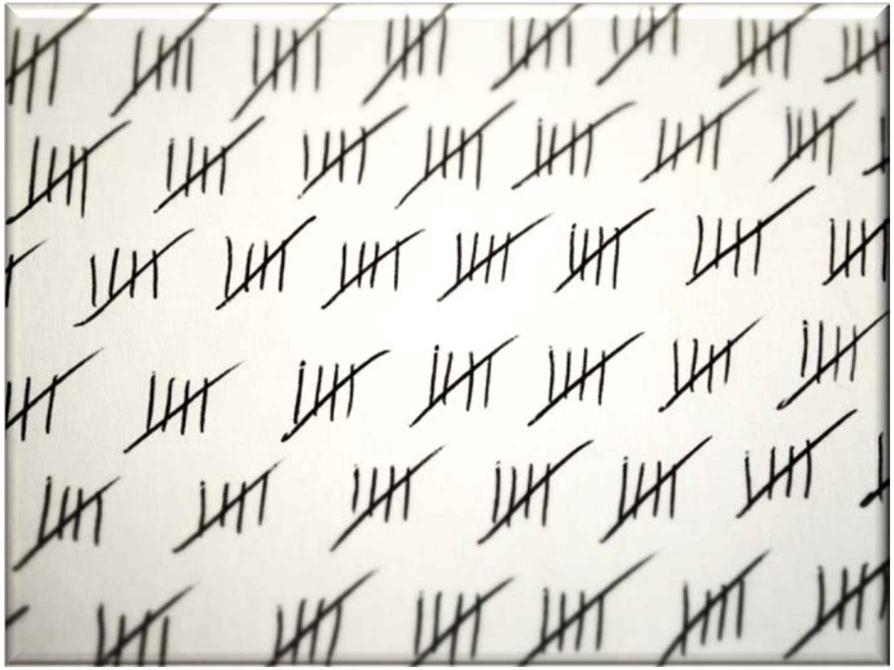


Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter (N) with precise meaning
      - **Negative**, exactly -N threads queued waiting to acquire semaphore



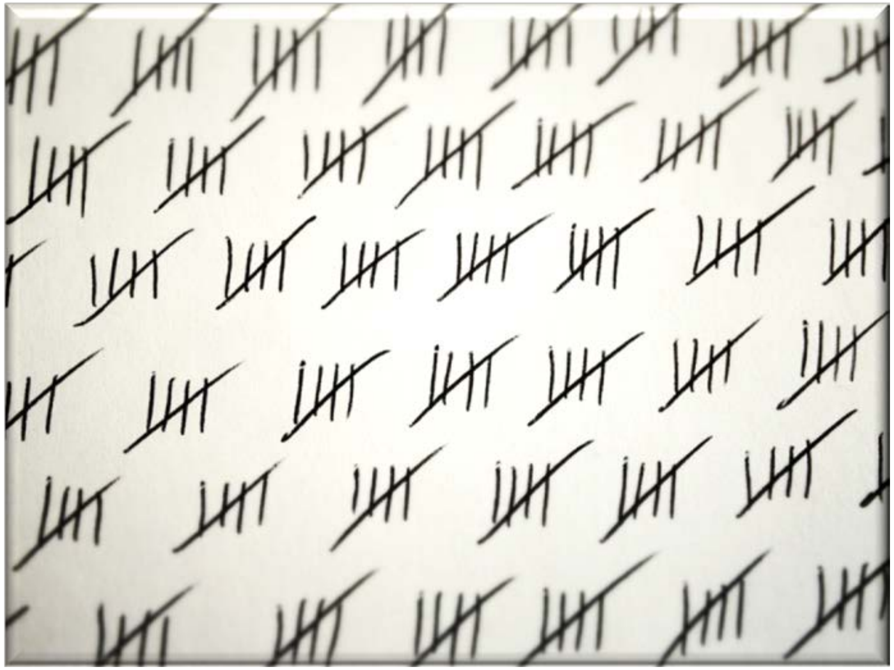
Overview of Semaphores



- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter (N) with precise meaning
      - **Negative**, exactly -N threads queued waiting to acquire semaphore
      - **Zero**, no waiting threads, an acquire operation would be put in a queue & block the invoking thread until counter N is positive

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
    - Have # of permits defined by a counter (N) with precise meaning
      - **Negative**, exactly -N threads queued waiting to acquire semaphore
      - **Zero**, no waiting threads, an acquire operation would be put in a queue & block the invoking thread until counter N is positive
      - **Positive**, no waiting threads, an acquire operation would not block invoking thread



Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
  - **Binary semaphores**

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
  - **Binary semaphores**
    - Has only 2 states: acquired & not acquired

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
  - **Counting semaphores**
  - **Binary semaphores**
    - Has only 2 states: acquired & not acquired
    - Restricts the counter N to the values 0 & 1

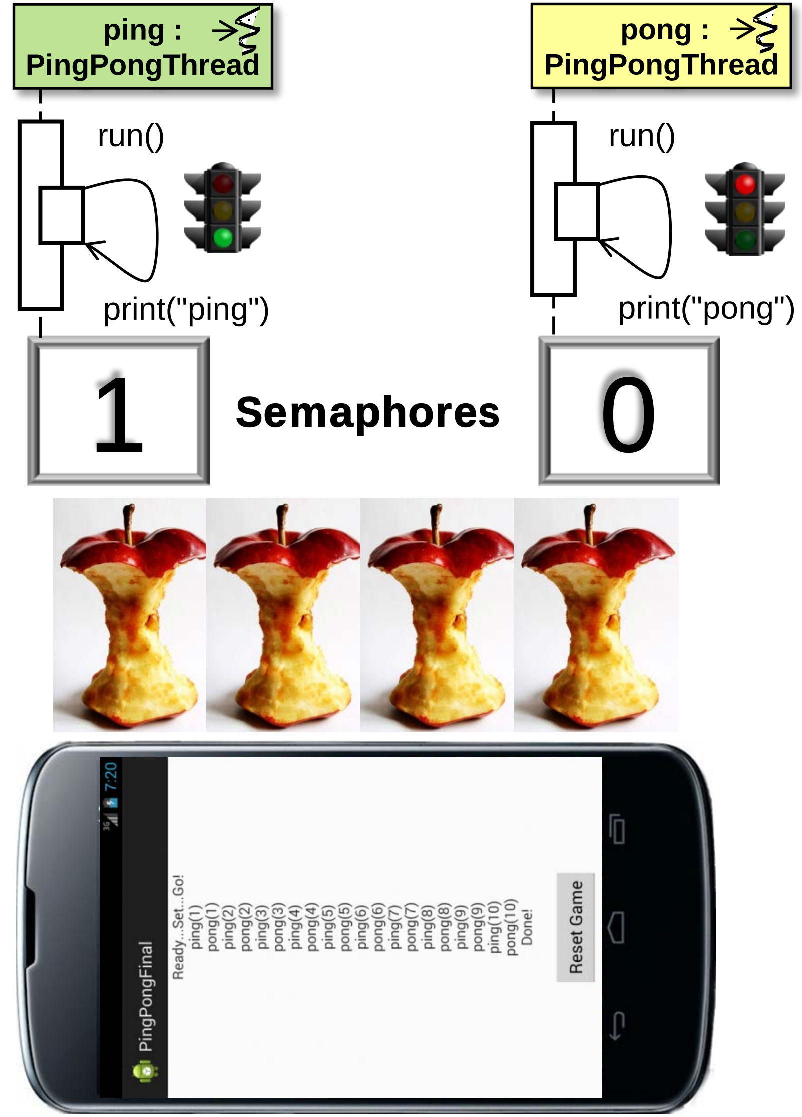
Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
- We'll analyze examples of both counting & binary semaphores in later videos

See upcoming part on "A Concurrent Resource Management Application"

Overview of Semaphores

- A semaphore can be atomically incremented & decremented to control access to a shared resource
- Concurrent programs use them to synchronize interactions between multiple threads
- There are two types of semaphores
- We'll analyze examples of both counting & binary semaphores in later videos



See upcoming part on "A Concurrent Ping/Pong Application"

A Counting Semaphore Example

A Counting Semaphore Example

- Consider an image rendering application that uses a thread pool

A Counting Semaphore Example

- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores

A Counting Semaphore Example

- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores
  - e.g., only allow use of two cores to ensure system responsiveness

A Counting Semaphore Example

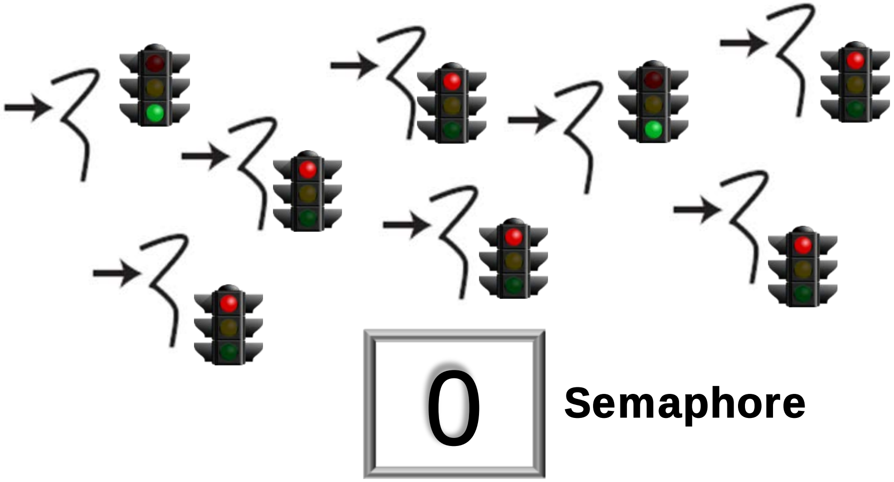
- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores
- A permit must be acquired from a semaphore before thread can run

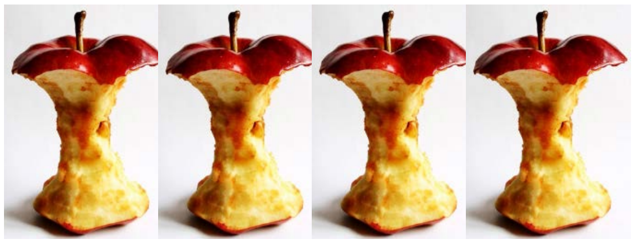
A Counting Semaphore Example

- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores
- A permit must be acquired from a semaphore before thread can run

A Counting Semaphore Example

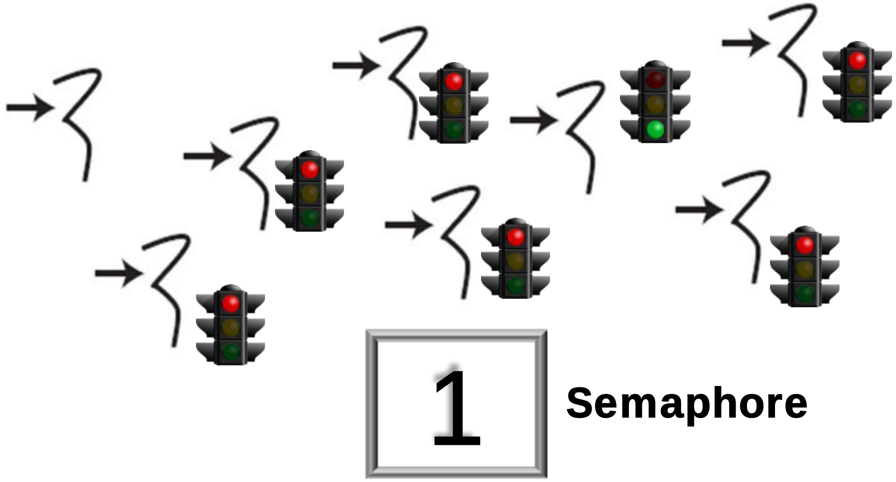
- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores
- A permit must be acquired from a semaphore before thread can run
  - Other threads will block

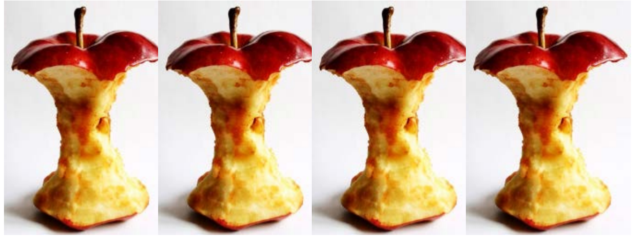




A Counting Semaphore Example

- Consider an image rendering application that uses a thread pool
- The application can be configured to restrict the # of threads that can run concurrently on processor cores
- A permit must be acquired from a semaphore before thread can run
  - Other threads will block
  - Until a permit is returned to semaphore



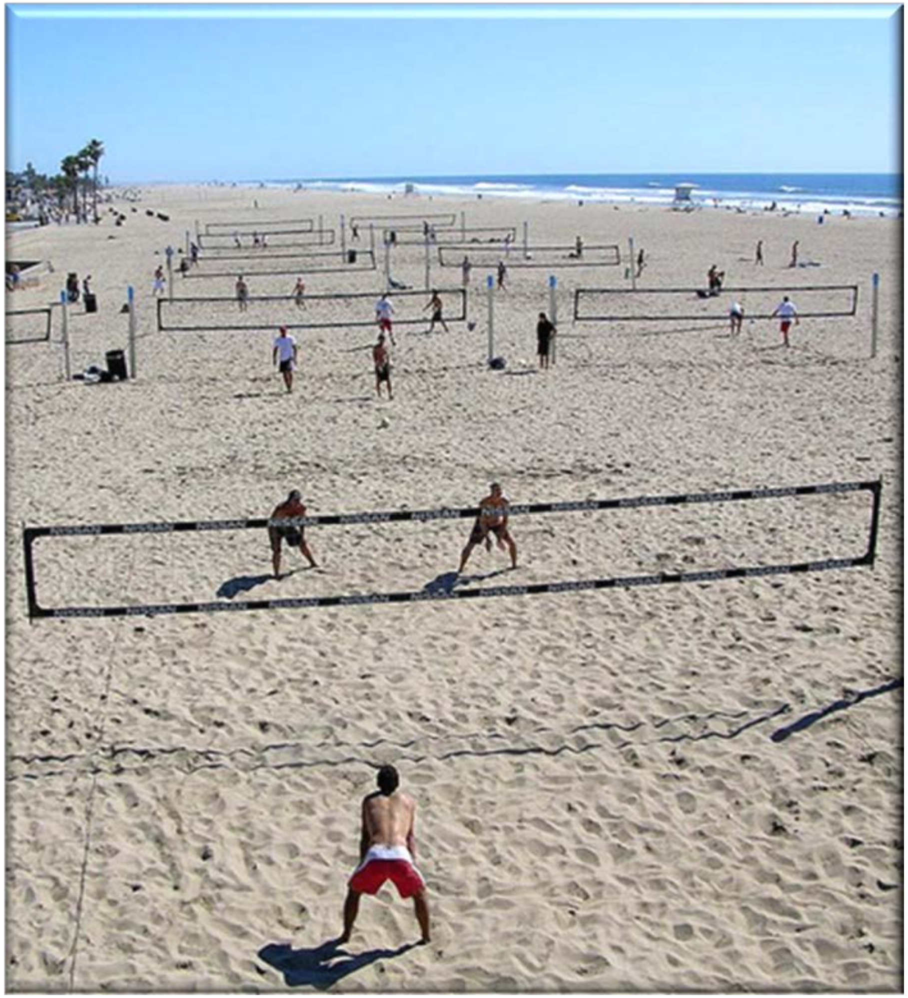


This example "fully brackets" acquire & release of permits to a semaphore

Human Known Use of Semaphores

Human Known Uses of Semaphores

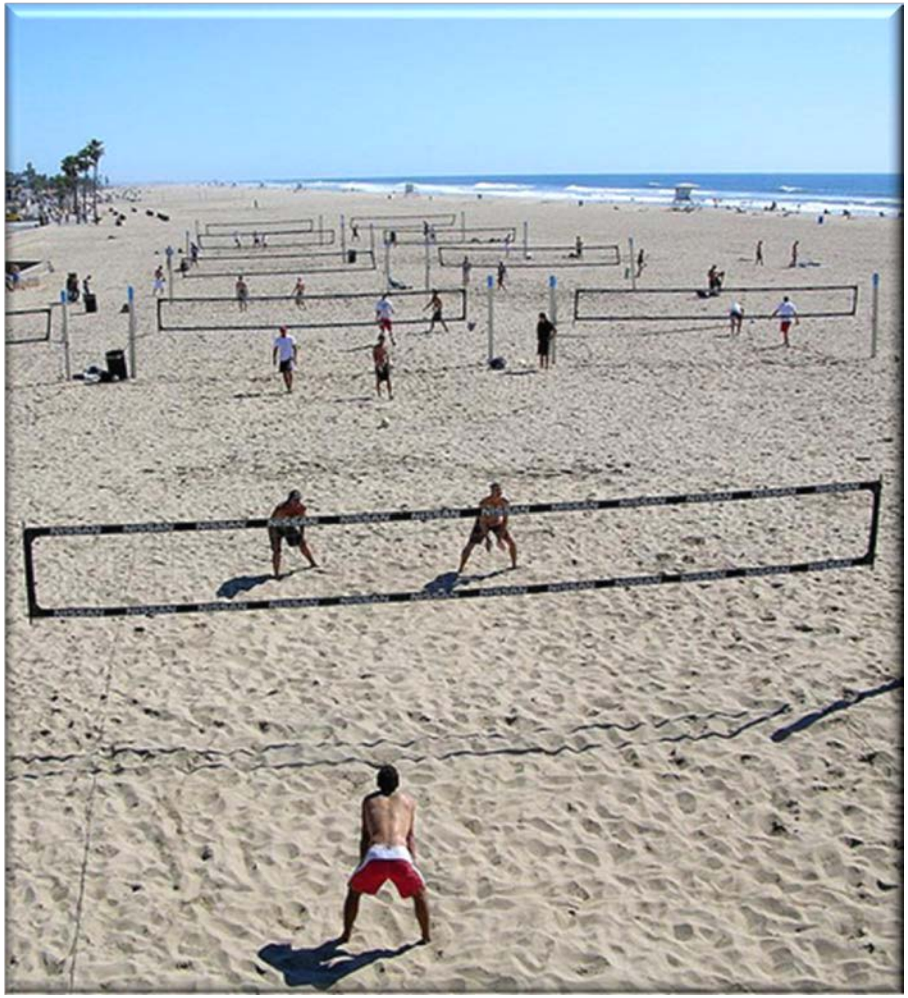
- A human known use of counting semaphores applies them to schedule access to beach volleyball courts



See en.wikipedia.org/wiki/Corona_del_Mar_State_Beach

Human Known Uses of Semaphores

- A human known use of counting semaphores applies them to schedule access to beach volleyball courts
  - A bag full of balls is used to limit the number of teams that can concurrently play volleyball



Overview of Java Semaphores (Part 1)

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/7/docs/api/java/util/concurrent/Semaphore.html

Overview of Java Semaphores

- Implements a variant of counting semaphores

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

Semaphore doesn't implement any synchronization-related interfaces

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.

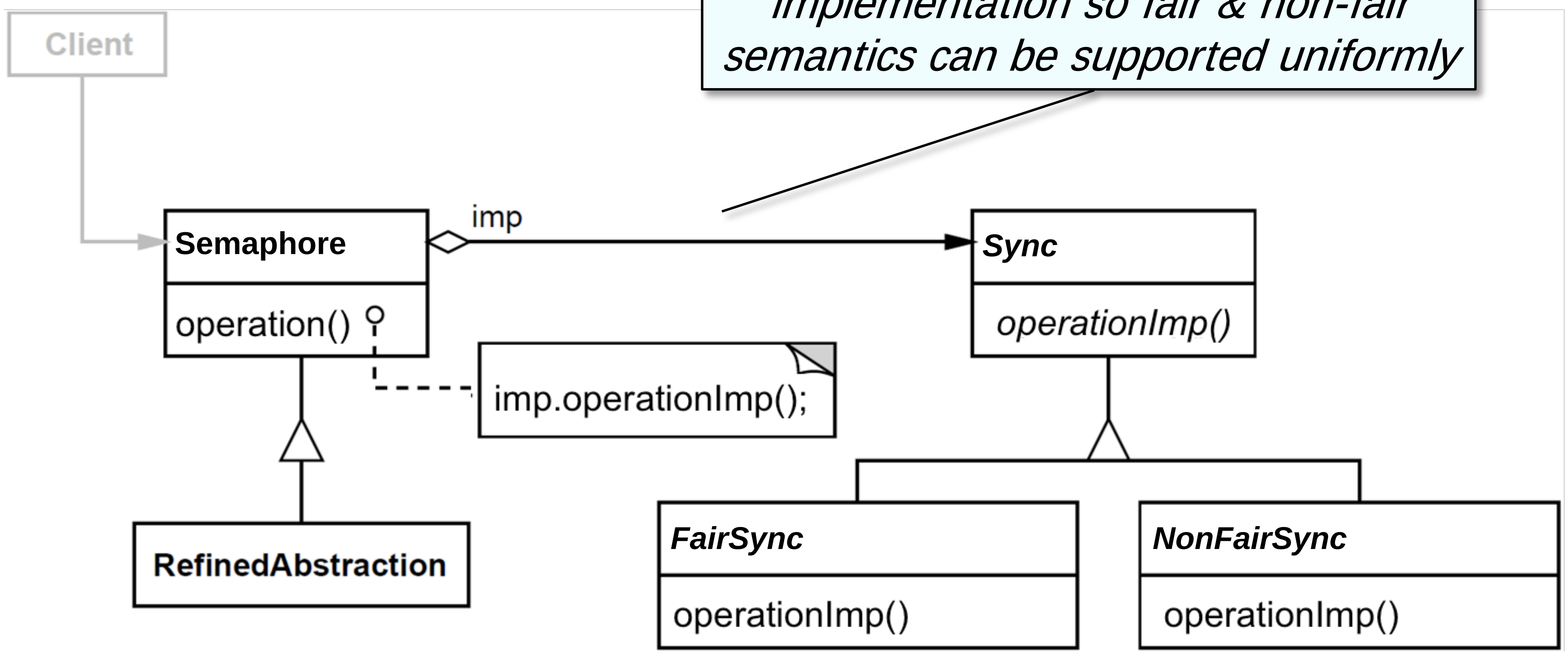
Semaphores are often used to restrict the number of threads than can access some (physical or logical) resource. For example, here is a class that uses a semaphore to control access to a pool of items:

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern

```
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

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy

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

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class

```
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/7/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model

```
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"

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model
  - The constructors create a Semaphore with a given # of permits

```
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

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model
  - The constructors create a Semaphore with a given # of permits

```
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

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model
  - The constructors create a Semaphore with a given # of permits
    - This number is *not* a maximum, just an initial value

```
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

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
  - Locking handled by Sync Implementor hierarchy
  - Inherits functionality from the AbstractQueuedSynchronizer class
  - Optionally implement fair or non-fair lock acquisition model
  - The constructors create a Semaphore with a given # of permits
    - This number is *not* a maximum, just an initial value
    - Permit value can be negative

```
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 (Part 2)

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore

```
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

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore

```
public class Semaphore
                  implements ... {
  ...
  public void acquire() {
    sync.
    acquireSharedInterruptibly(1);
  }
  ...
```

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts

```
public class Semaphore
                 implements ... {
  ...
  public void
    acquireUninterruptibly() {
    sync.acquireShared(1)
  }
  ...
```

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if one available at invocation time

```
public class Semaphore
                implements ... {
  ...
  public boolean tryAcquire()
                              ... {
     sync.
       nonfairTryAcquireShared(1)
       >= 0;
  }
  ...
```

Untimed tryAcquire() methods don't honor fairness setting & take any permits available

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
  - acquire() obtains a permit from the semaphore
  - acquireUninterruptibly() ignores interrupts
  - tryAcquire() obtains a permit if one available at invocation time
  - Release() returns a permit, increasing number by 1

```
public class Semaphore
                implements ... {
  ...
  public void release() {
    sync.releaseShared(1);
  }
  ...
```

It's valid for the permit count to exceed the initial permit count

Overview of Java Semaphores (Part 3)

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
- There are many other Semaphore methods

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 Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
- There are many other Semaphore methods

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 Java Semaphores

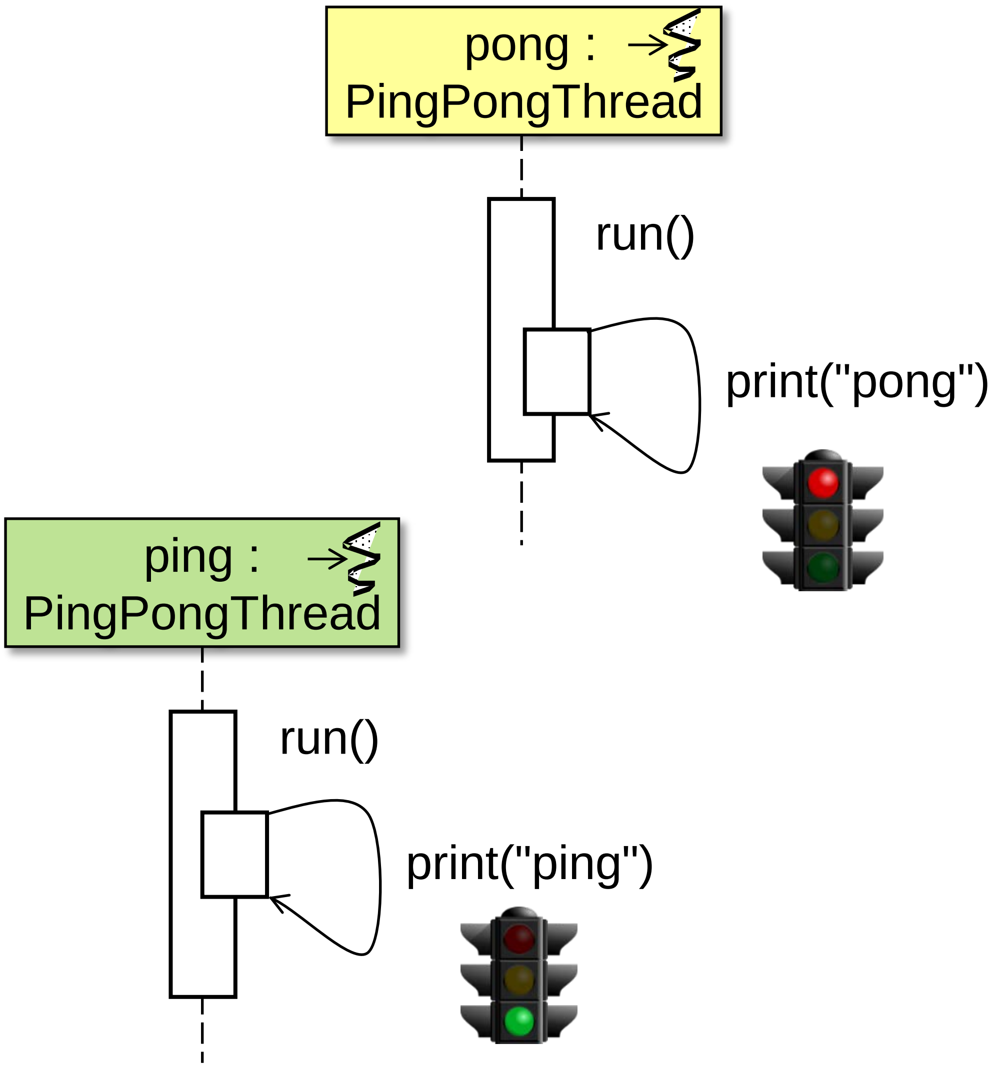
- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
- There are many other Semaphore methods

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

Timed tryAcquire() methods *do* honor the fairness setting, so they don't "barge"

Overview of Java Semaphores

- Implements a variant of counting semaphores
- Applies the *Bridge* pattern
- Its key methods acquire & release the semaphore
- There are many other Semaphore methods
- Acquiring & releasing permits to a Semaphore need not be fully bracketed

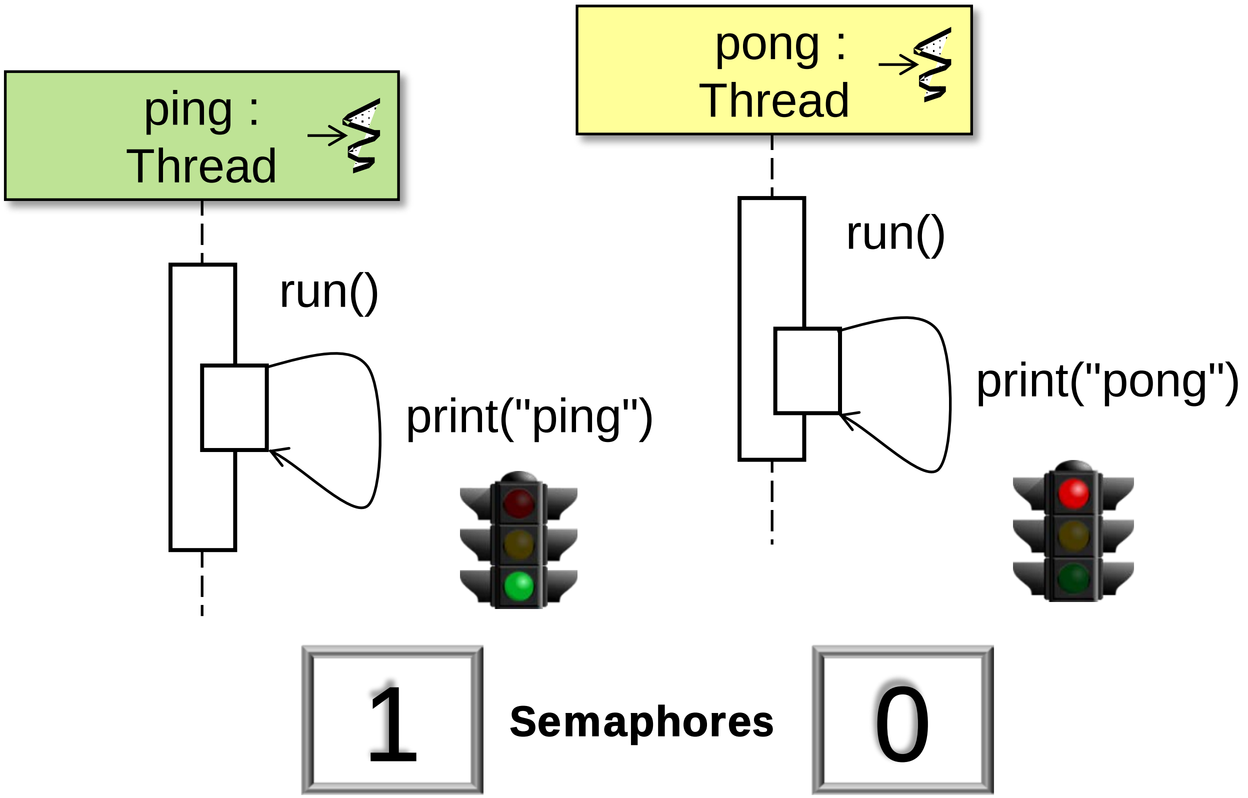


See upcoming part on "A Concurrent Ping/Pong Application"

Applying Java Semaphores in Practice (Part 1)

Applying the Java Semaphore in Practice

- This Java application coordinates thread interactions via Java Semaphores
  - i.e., the threads alternate printing "ping" & "pong" on the console

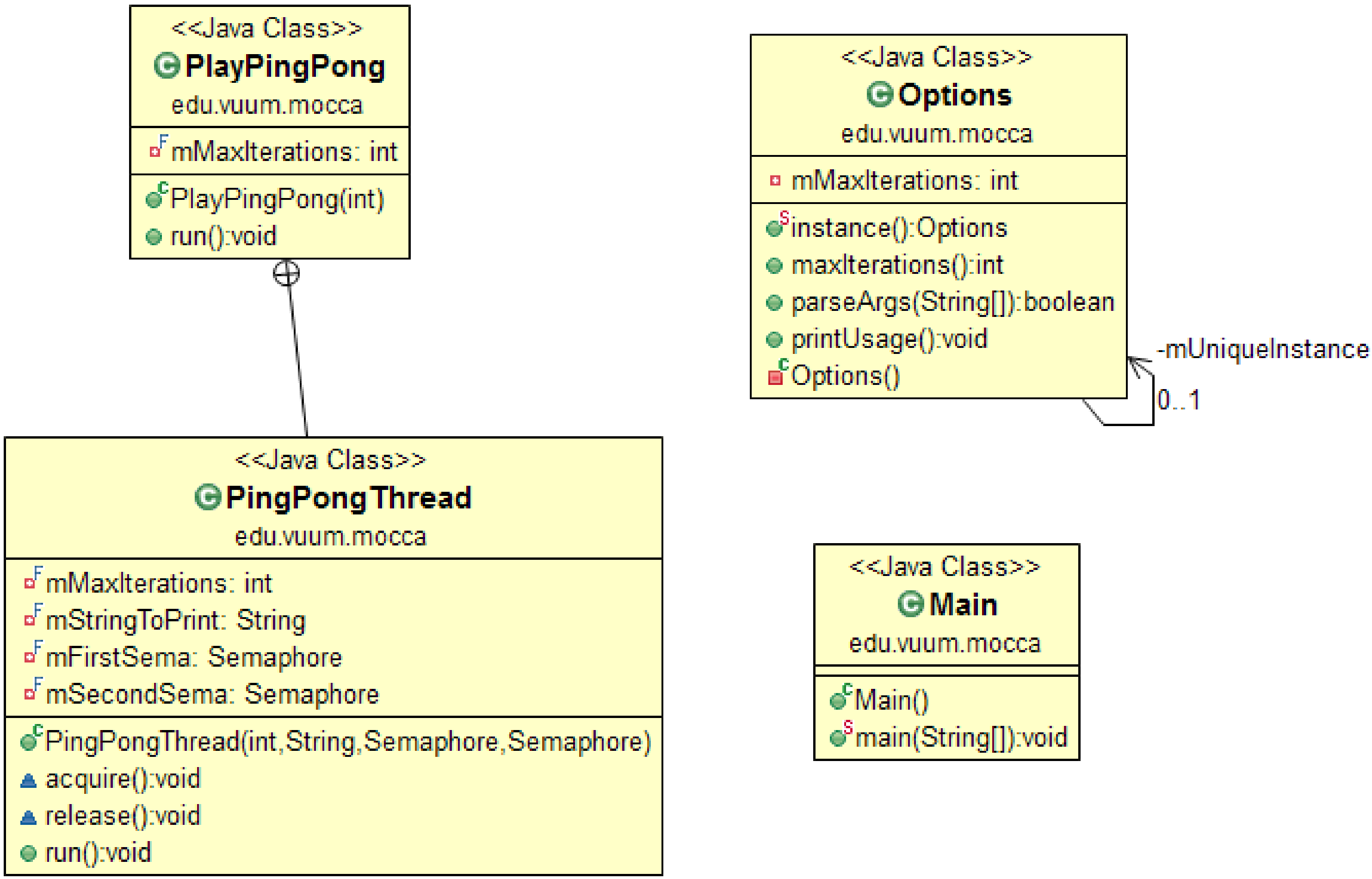



```
% java PlayPingPong
Ready...Set...Go!
Ping!(1)
Pong!(1)
Ping!(2)
Pong!(2)
Ping!(3)
Pong!(3)
Ping!(4)
Pong!(4)
Ping!(5)
Pong!(5)
Ping!(6)
Pong!(6)
Ping!(7)
Pong!(7)
Ping!(8)
Pong!(8)
Ping!(9)
Pong!(9)
Ping!(10)
Pong!(10)
Done!
```

See github.com/douglascraigschmidt/POSA-15/tree/master/ex/PingPong/console

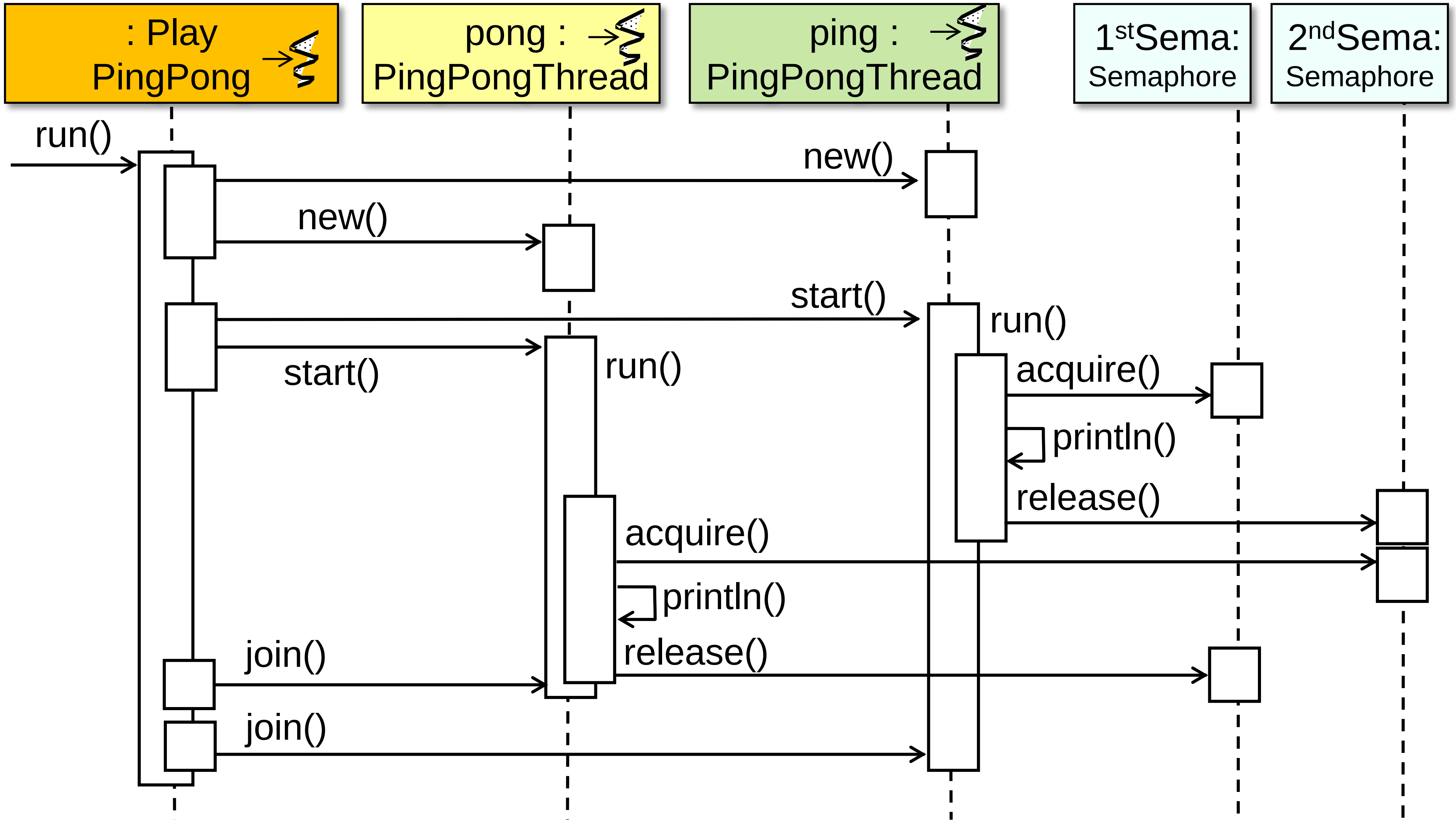
Applying the Java Semaphore in Practice

- UML class diagram for the ping-pong application



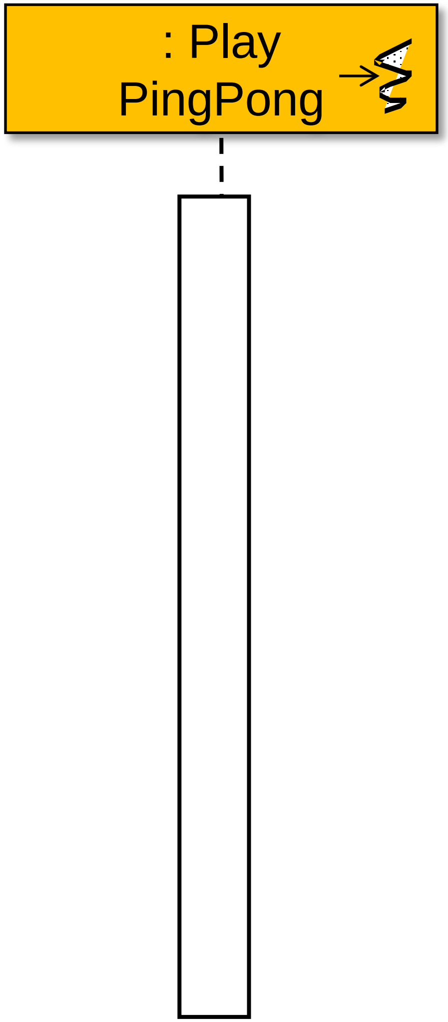
Applying the Java Semaphore in Practice

- UML sequence diagram for the ping-pong application



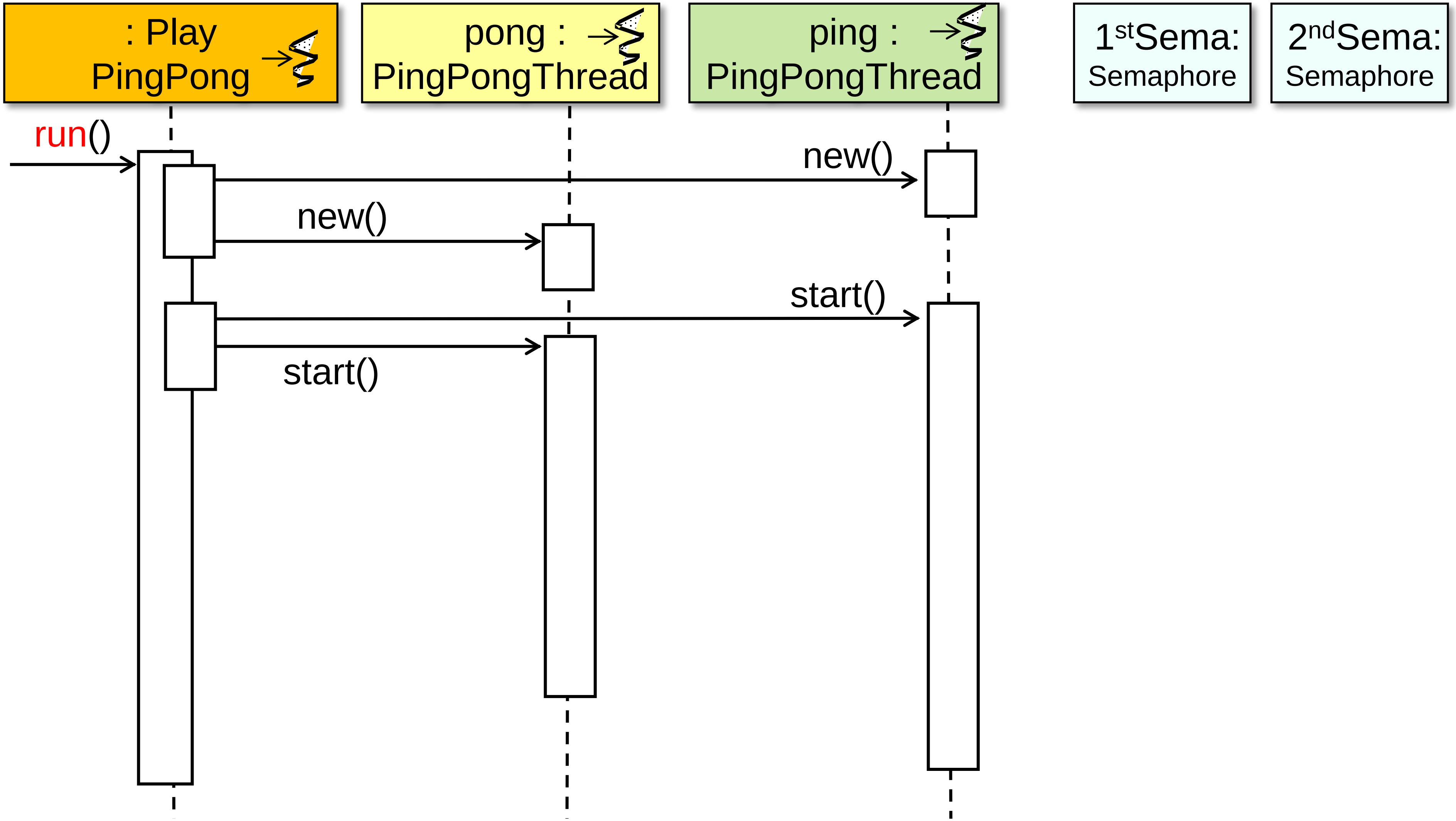
Applying the Java Semaphore in Practice

- UML sequence diagram for the ping-pong application



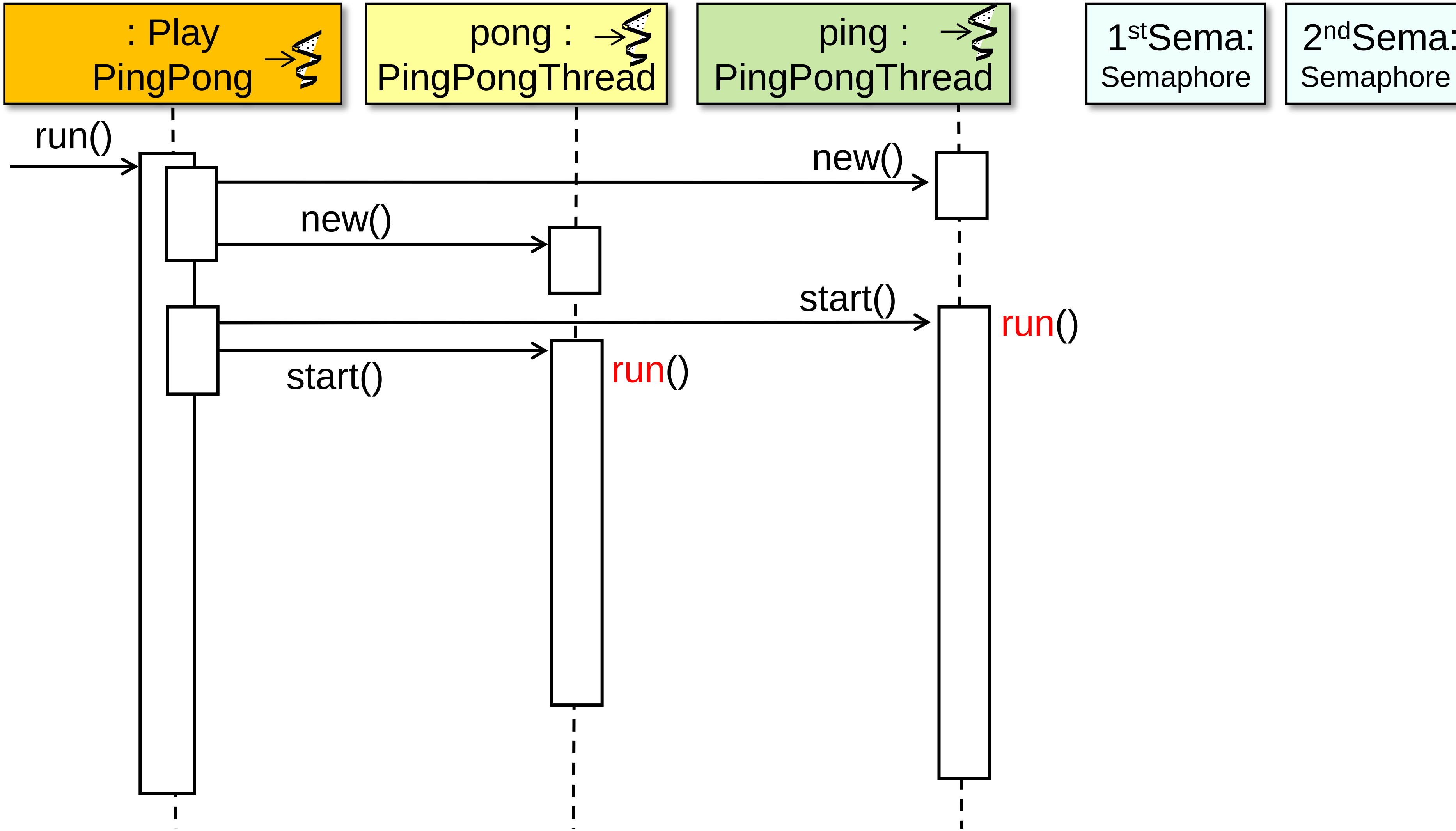
Applying the Java Semaphore in Practice

- UML sequence diagram for the ping-pong application



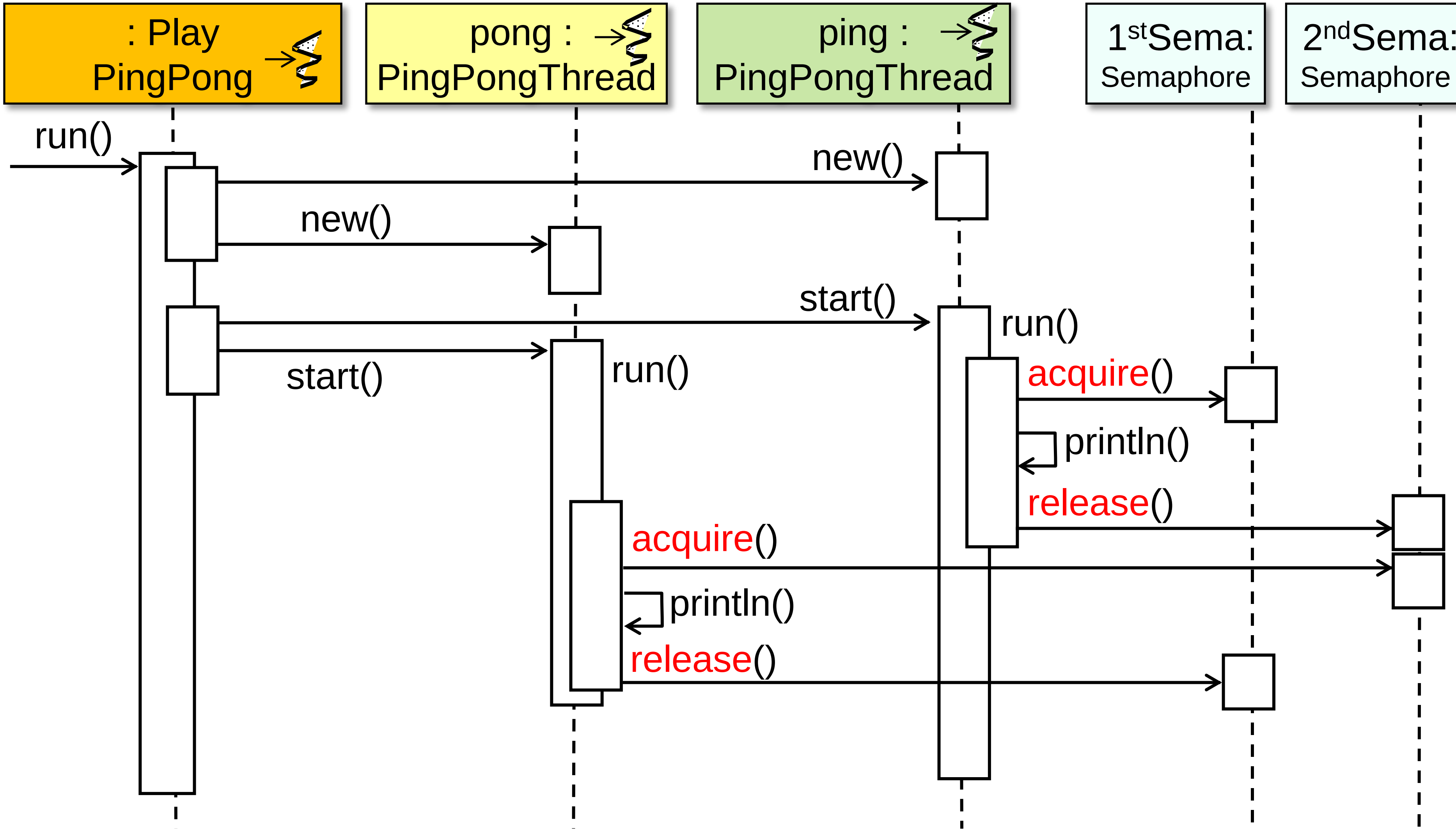
Applying the Java Semaphore in Practice

- UML sequence diagram for the ping-pong application



Applying the Java Semaphore in Practice

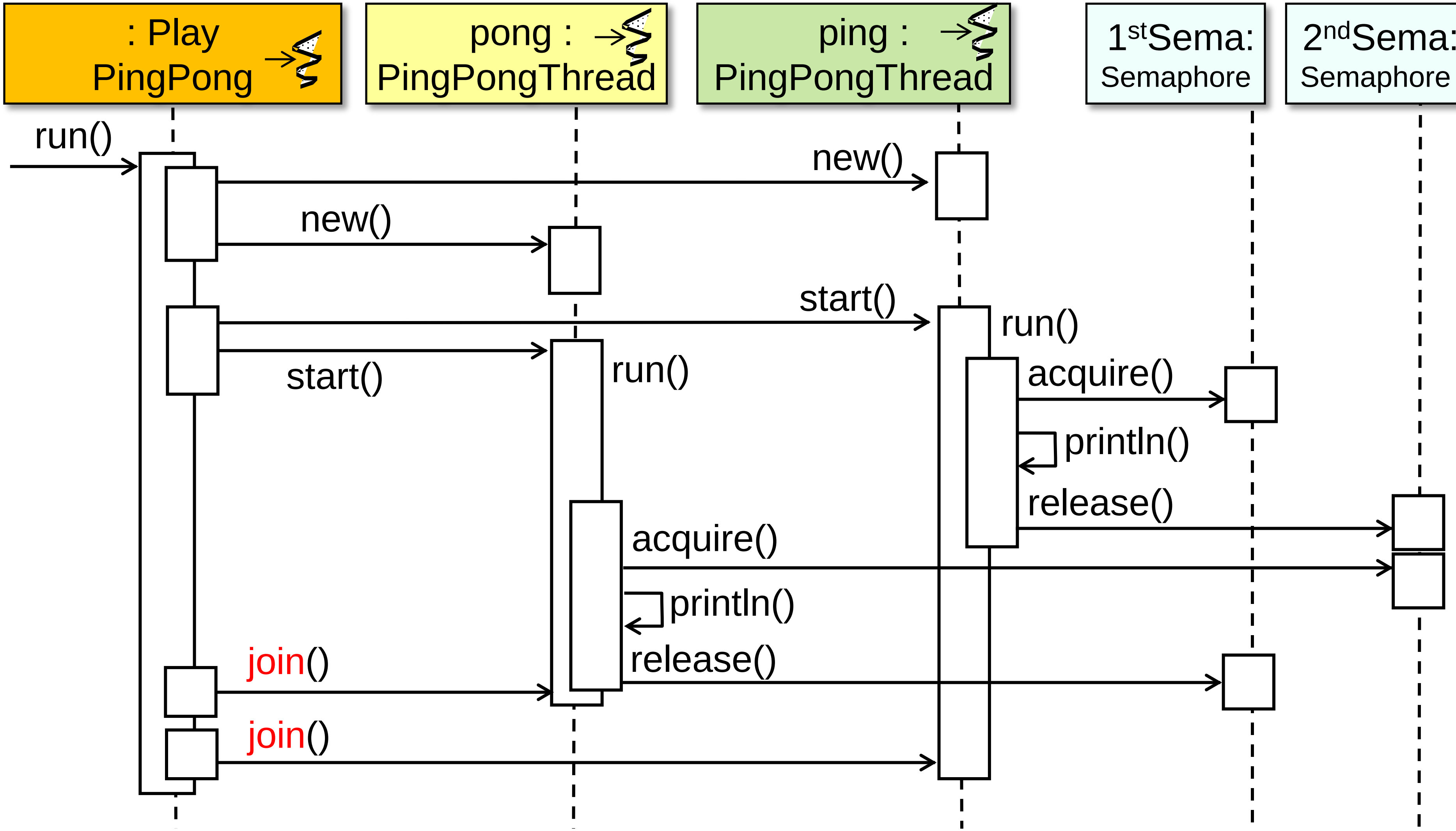
- UML sequence diagram for the ping-pong application



A pair of Semaphores coordinates the order in which the “ping” & “pong” Threads are called

Applying the Java Semaphore in Practice

- UML sequence diagram for the ping-pong application



The PlayPingPong Thread joins with the other Threads once they are finished

Semaphore Usage Considerations

Semaphore Usage Considerations

- Semaphore is more flexible than other 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:

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

All Implemented Interfaces:

Serializable, Lock

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

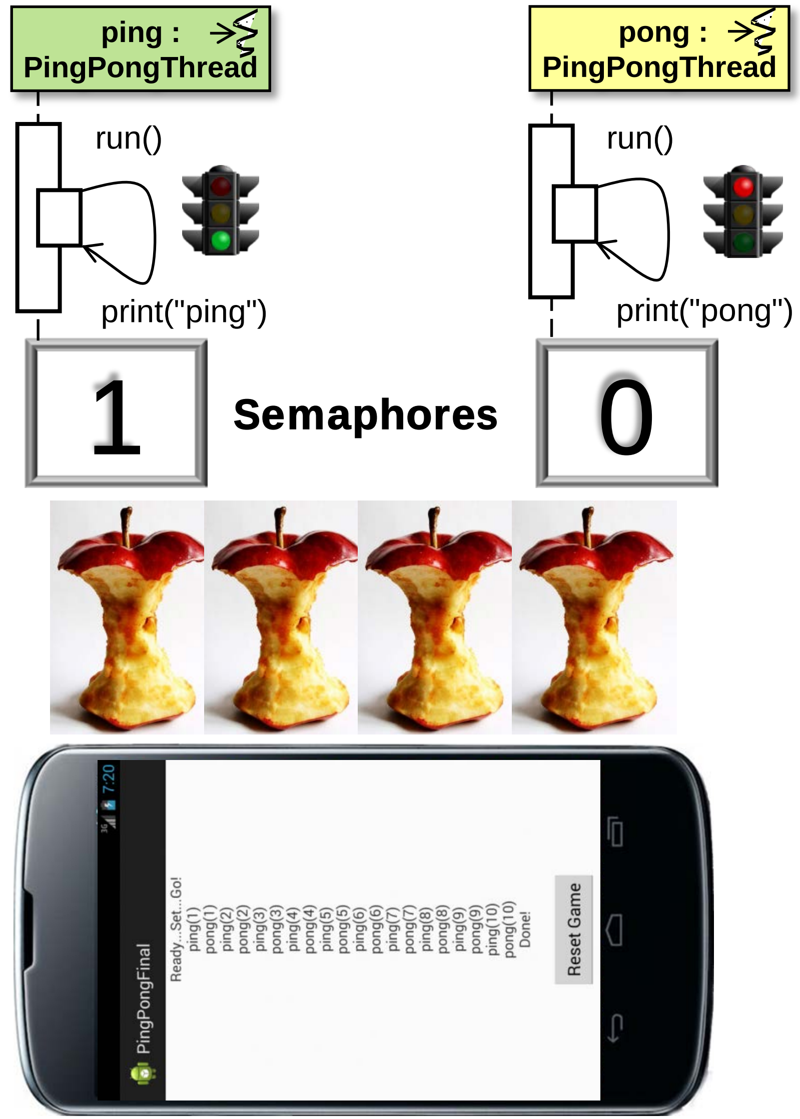
A reentrant mutual exclusion `Lock` with the same basic behavior and semantics as the implicit monitor lock accessed using `synchronized` methods and statements, but with extended capabilities.

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

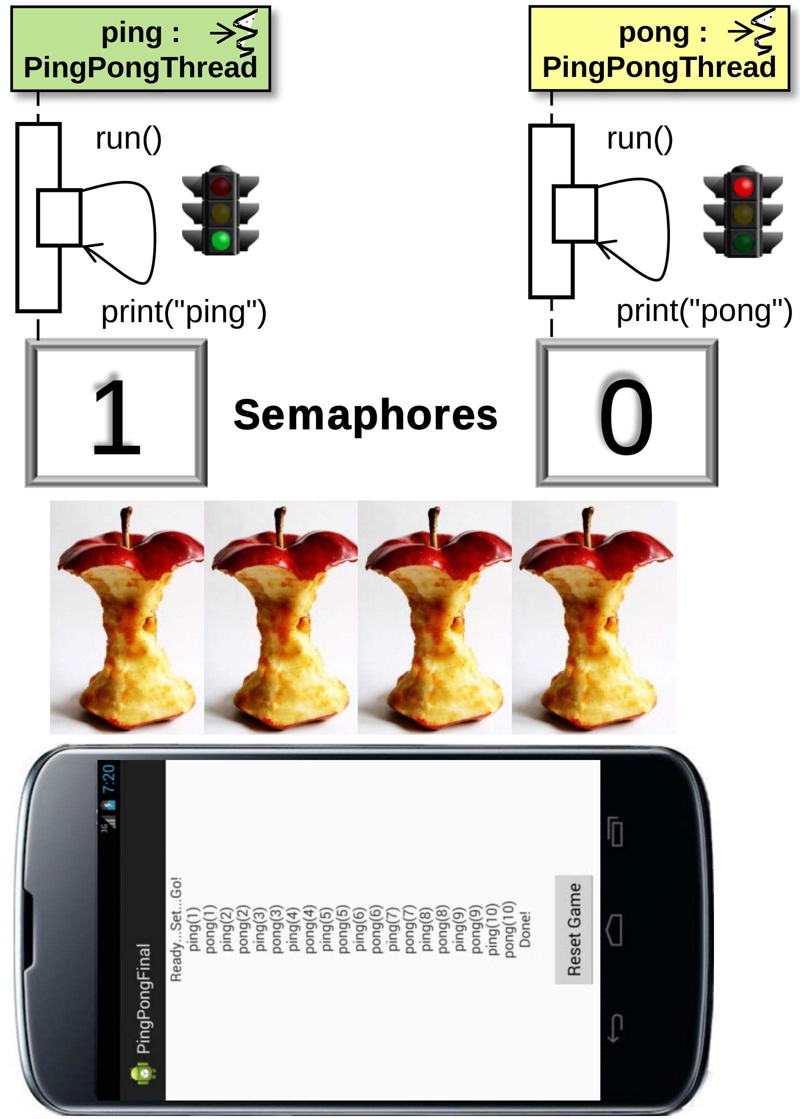
Semaphore Usage Considerations

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



Semaphore Usage Considerations

- Semaphore is more flexible than other 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

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free

Semaphore Usage Considerations

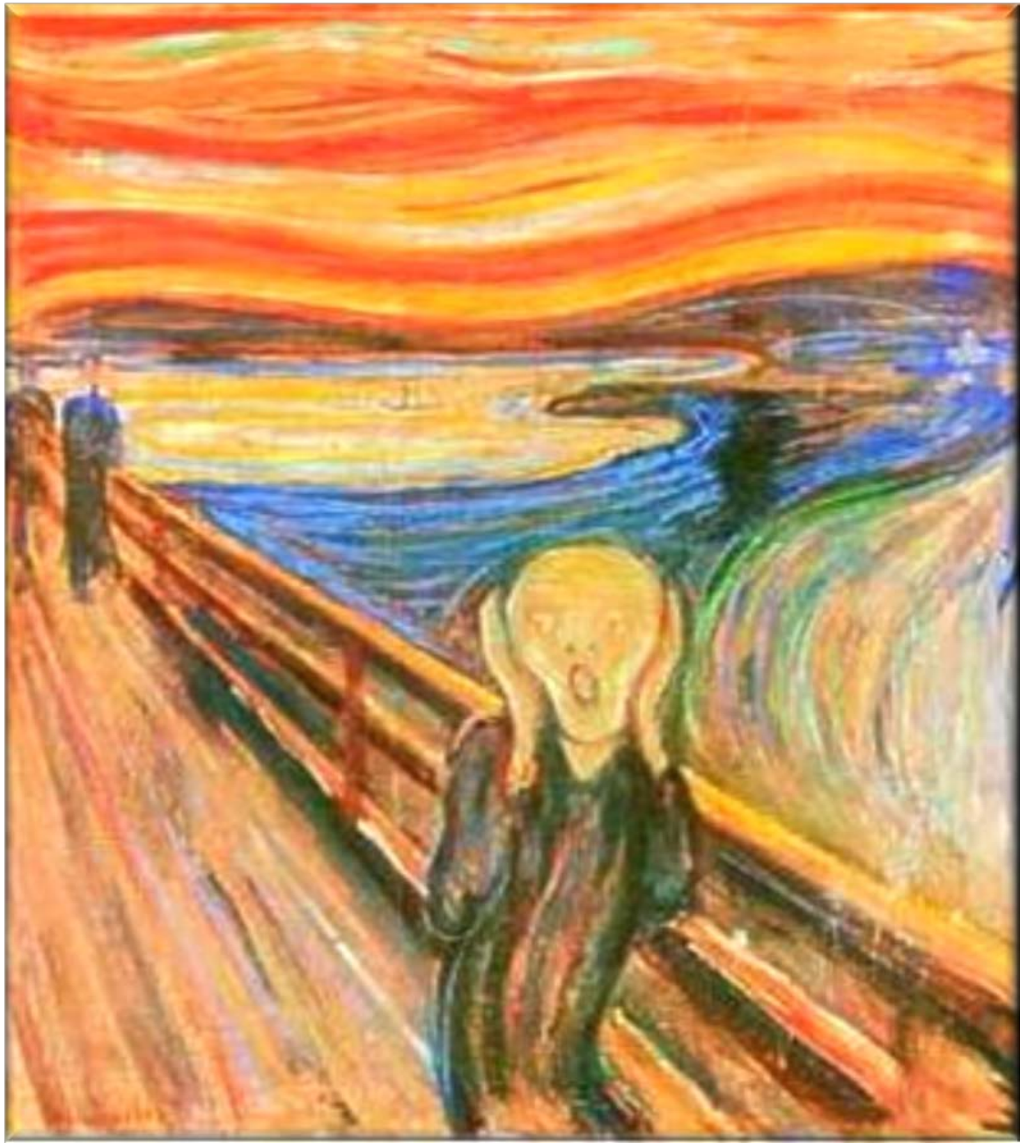
- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
  - Other mechanisms may be needed to select a particular free resource

Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
  - Other mechanisms may be needed to select a particular free resource
    - e.g., a HashMap, other Semaphores, etc.

Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
- Semaphores can be tedious & error-prone to program due to common traps & pitfalls



Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
- Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Acquiring a semaphore & forgetting to release it

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

void someMethod() {
  semaphore.acquire();

  ... // Critical section
  return;
}
```

Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
- Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Acquiring a semaphore & forgetting to release it

```
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

Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
- 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 semaphore =
  new Semaphore(1);

void someMethod() {
  semaphore.acquire();

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

Semaphore Usage Considerations

- Semaphore is more flexible than other Java synchronizers
- When used for a resource pool, it tracks # of free resources, *not* which resources are free
- 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

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

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

  semaphore.release();
  semaphore.release();
  semaphore.release();
}
```