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 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:

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

Class ReentrantLock

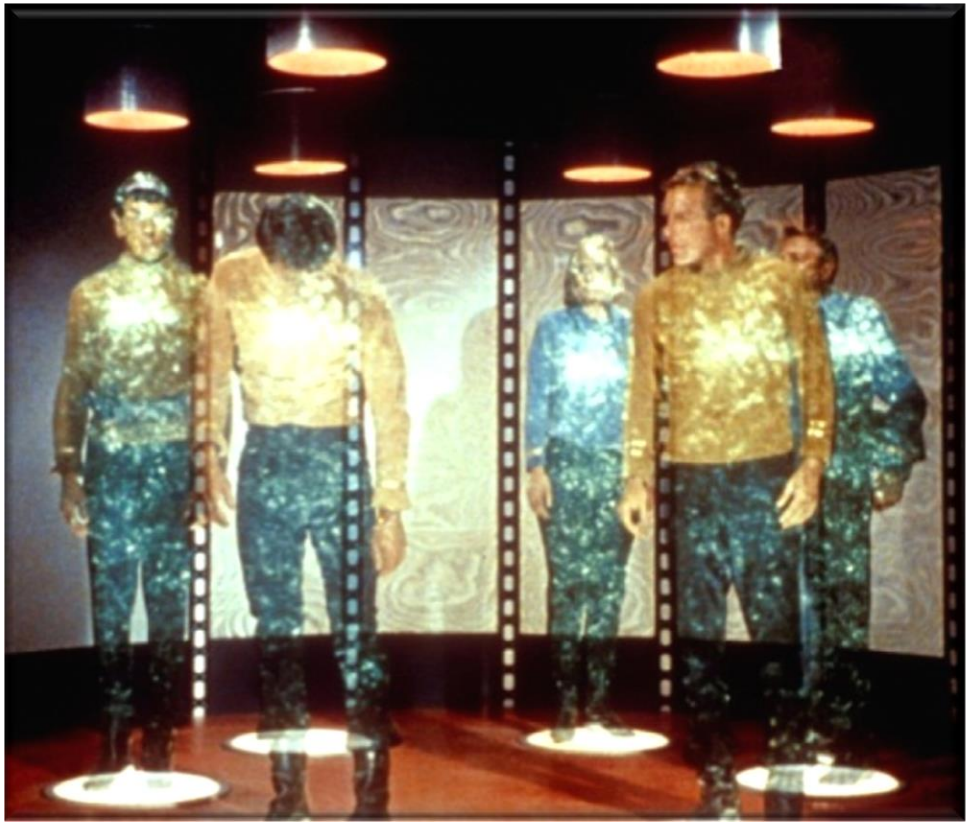
java.lang.Object
    java.util.concurrent.locks.ReentrantLock

All Implemented Interfaces:

Serializable, Lock

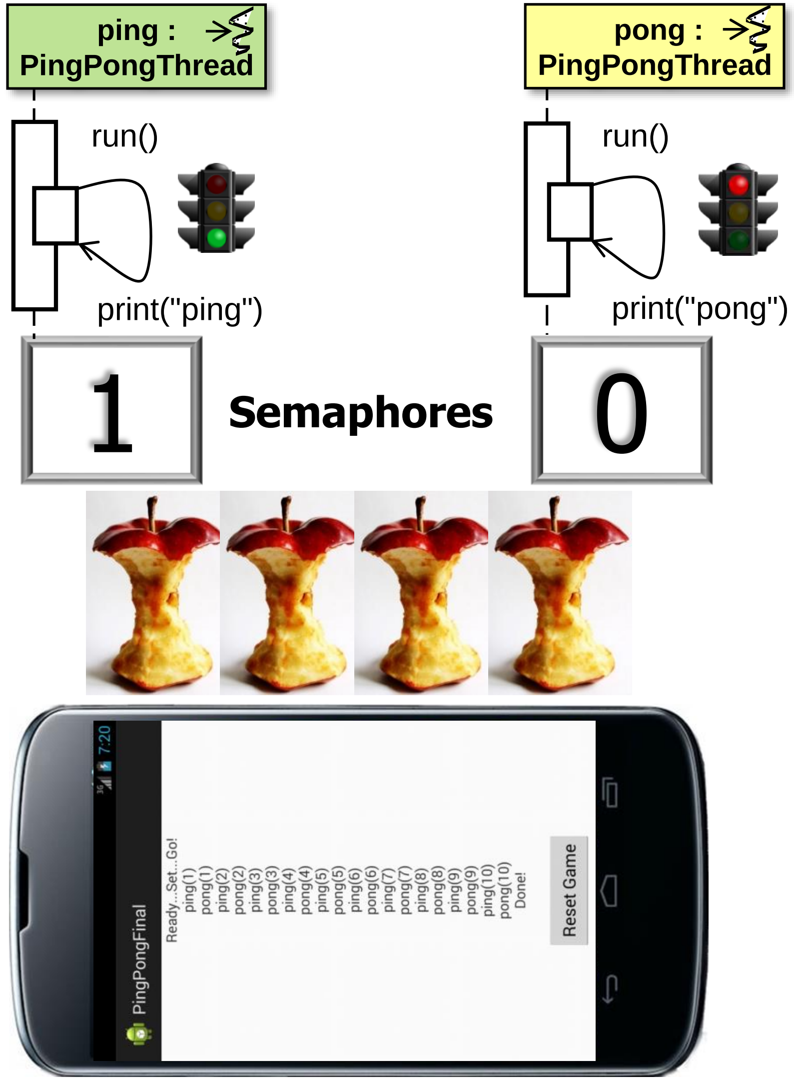
Java Semaphore Usage Considerations

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



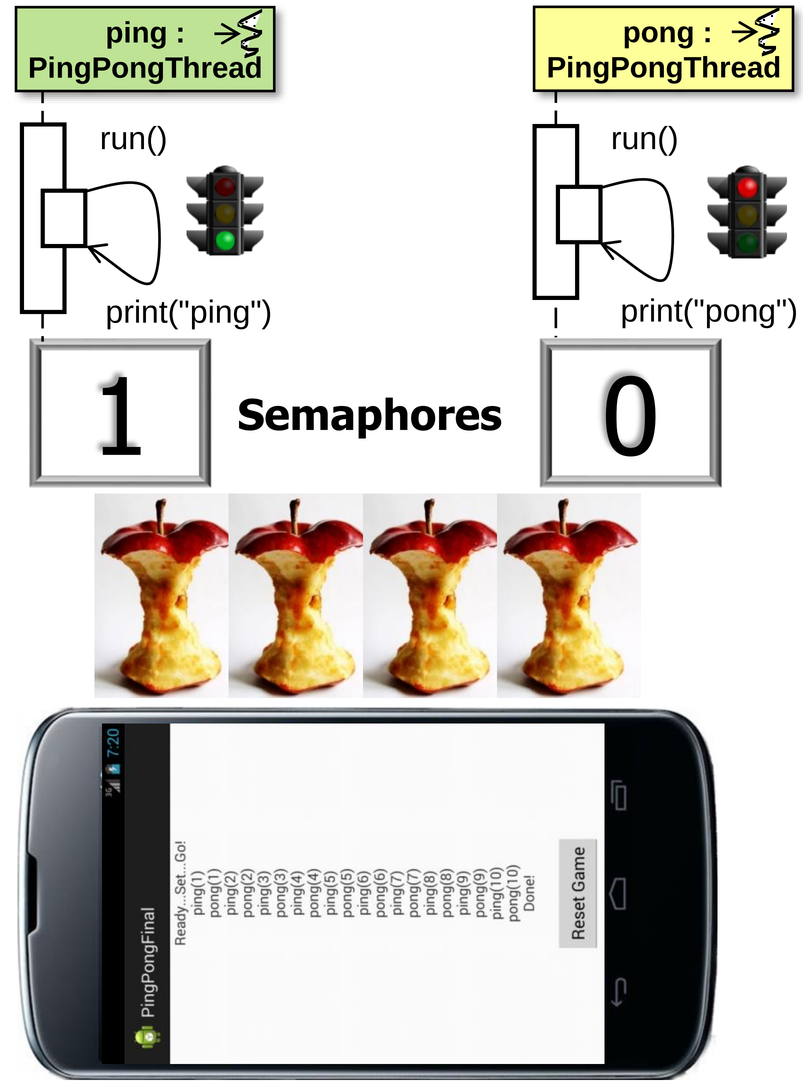
Java Semaphore Usage Considerations

- 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 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



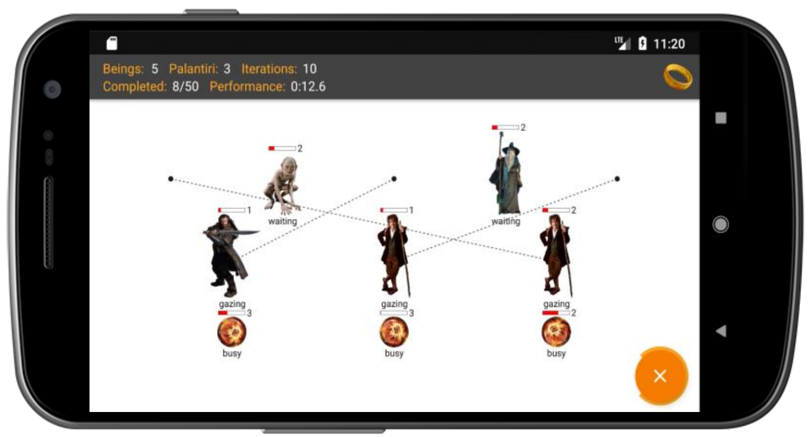
Naturally, this flexibility comes at some additional cost in performance

Java Semaphore Usage Considerations

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

2

Semaphore

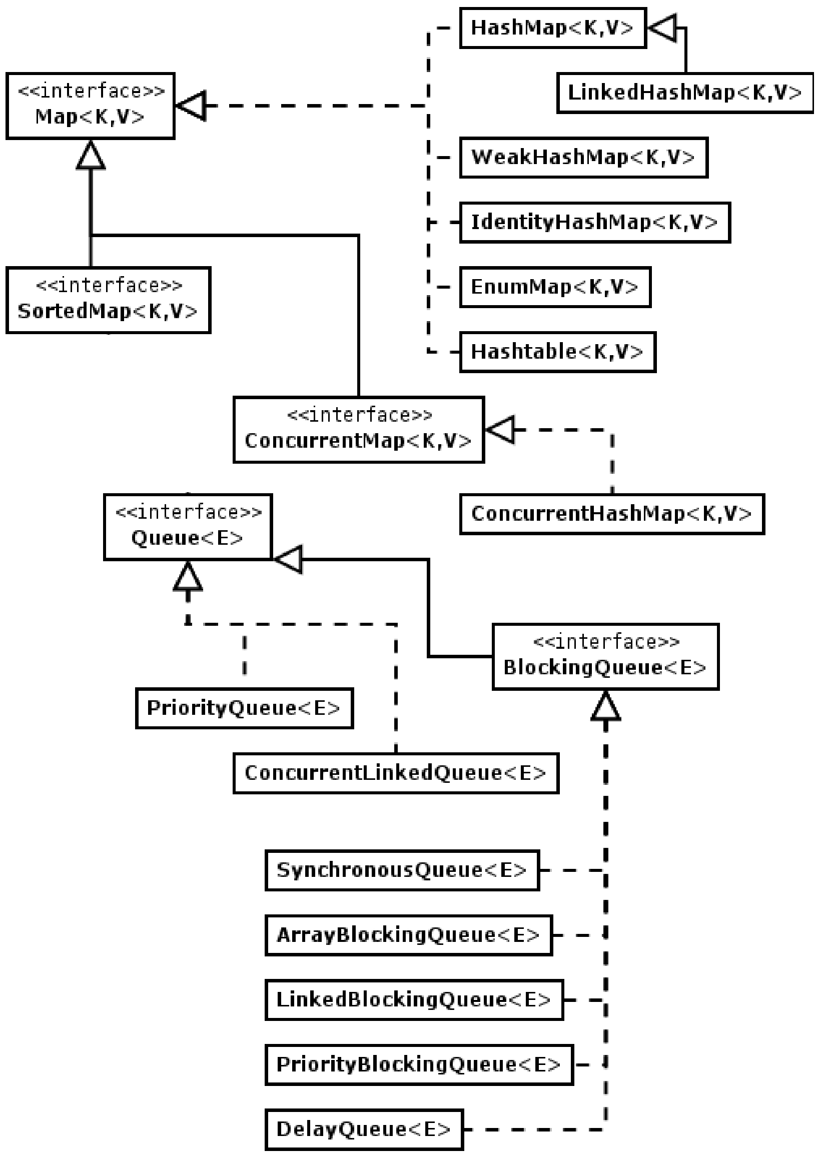


Java Semaphore Usage Considerations

- When a semaphore is used for a resource pool, it tracks the # of free resources
  - However, it does not track *which* resources are free

Java Semaphore Usage Considerations

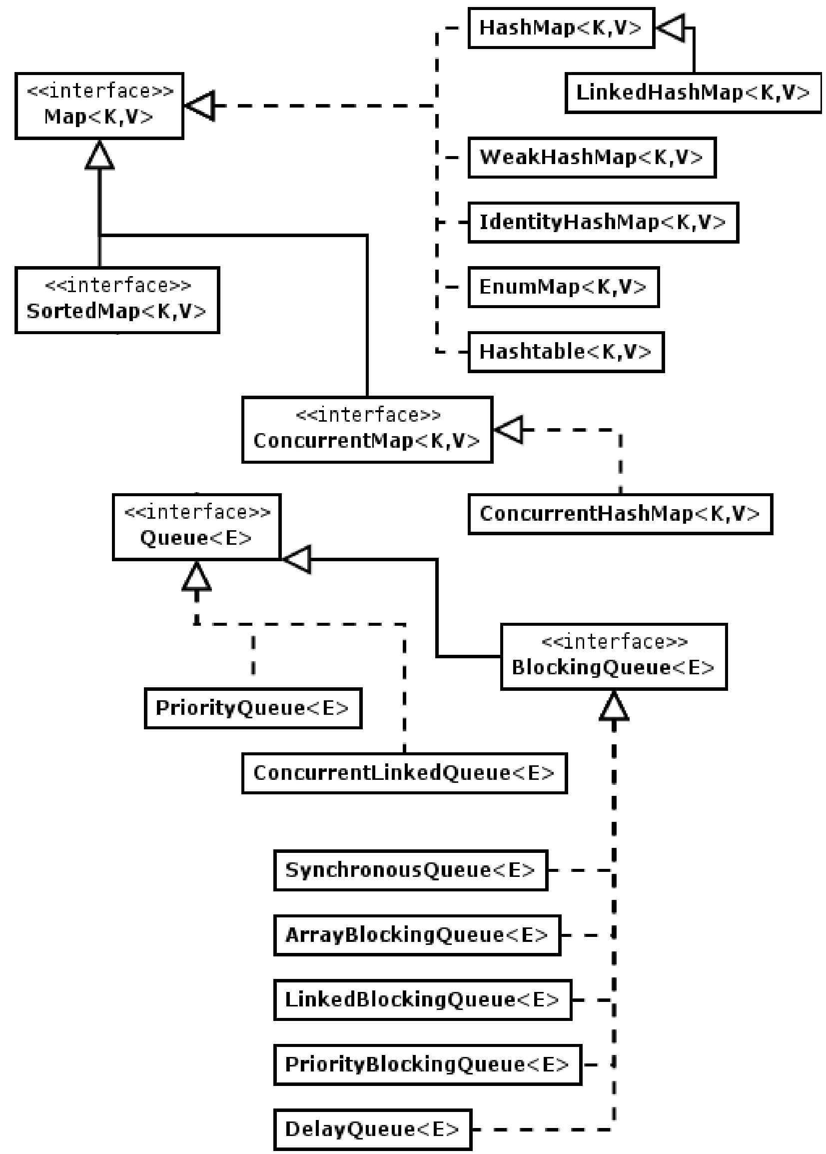
- When a semaphore is used for a resource pool, it tracks the # of free resources
  - However, it does not track *which* resources are free
  - Other mechanisms may be needed to select a particular free resource
    - e.g., a List, HashMap, etc.



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 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

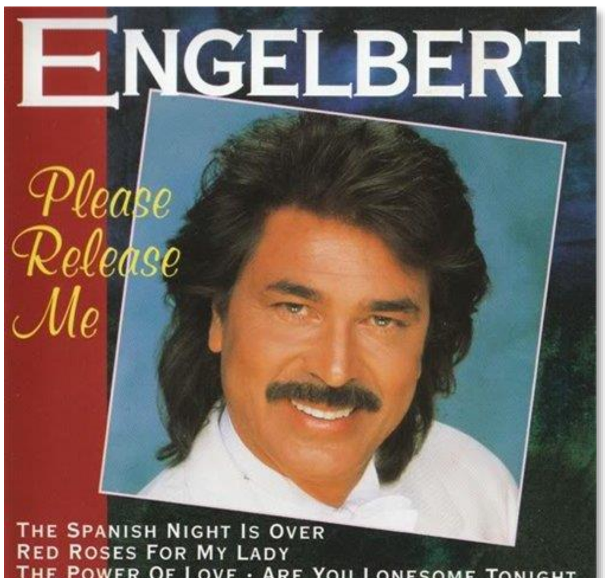
- Semaphores can be tedious & error-prone to program due to common traps & pitfalls, e.g.
  - Holding a semaphore for a long time without needing it


```
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 the semaphore more times than needed




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

void someMethod() {
  semaphore.acquire();   1
  ...

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

These extra calls to release() will falsely 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 the semaphore more times than needed
  - Acquiring a semaphore & forgetting to release it

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

void someMethod() {
  semaphore.acquire();

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

The semaphore may be locked indefinitely!

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

```
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

Java Semaphore Usage Considerations

- Semaphores are rather limited synchronizers that don't scale to complex coordination use cases

ConditionObjects may be better for more complex coordination use-cases

End of Java Semaphores (Part 4)