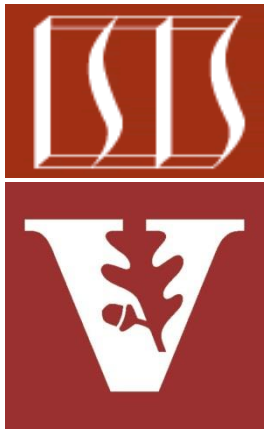
Java Monitor Object Synchronized Statements

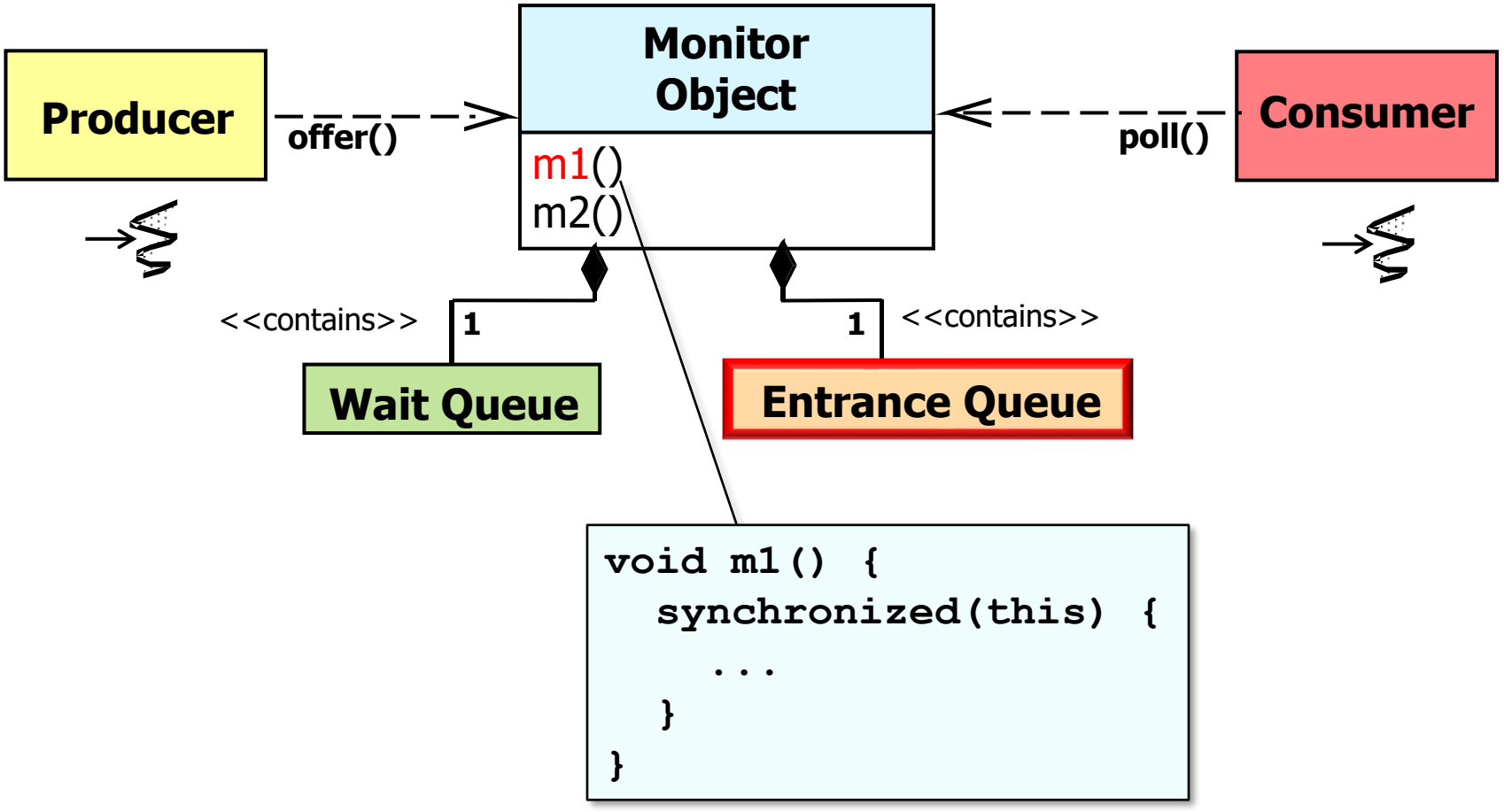


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize how the synchronized methods/statements provided by Java built-in monitor objects support *mutual exclusion*



Mutual exclusion is used to protect shared state from corruption due to concurrent access by multiple threads

Java Synchronized Statements

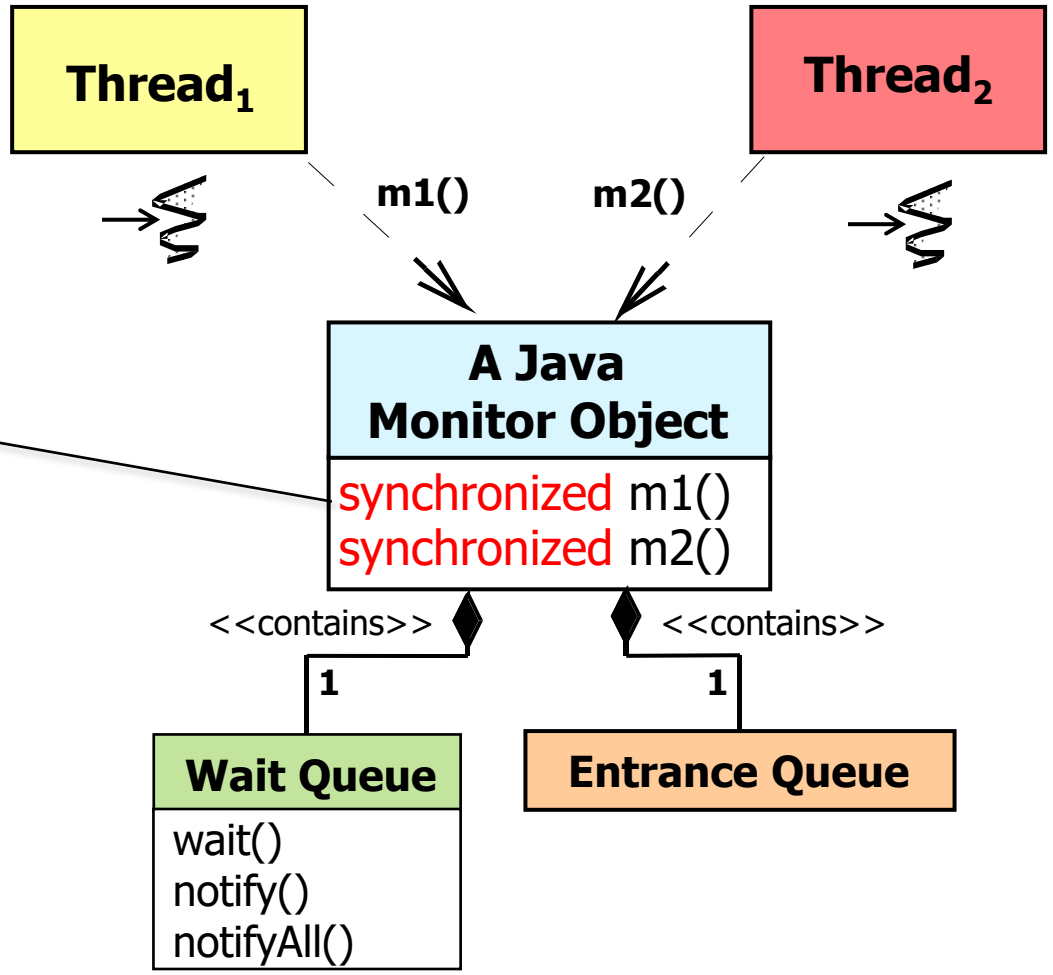
Java Synchronized Statements

- Synchronized methods incur several constraints

See previous lessons on "*Java Synchronized Methods*"

Java Synchronized Statements

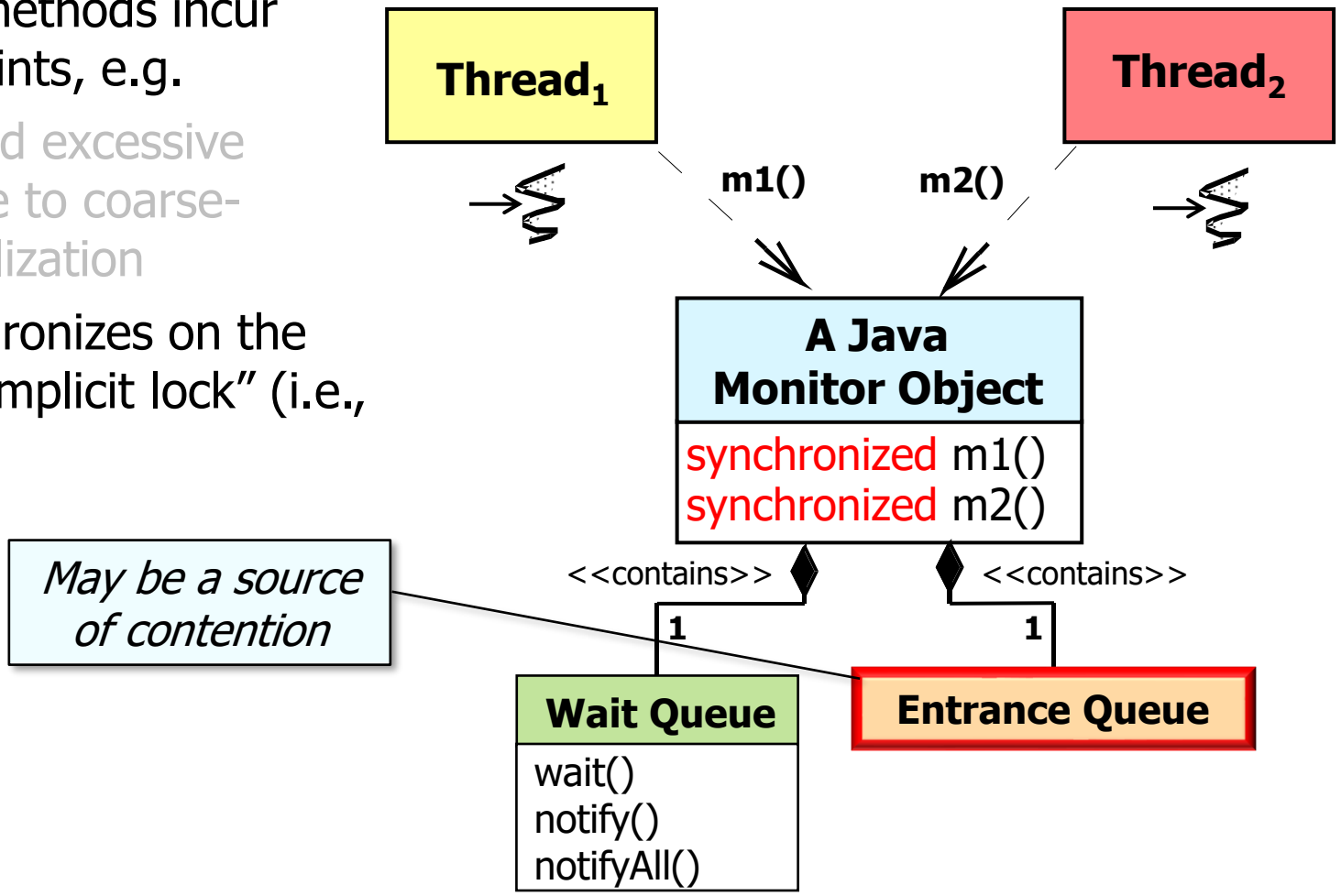
- Synchronized methods incur several constraints, e.g.
  - They can yield excessive overhead due to coarse-grained serialization



Synchronization occurs at the method level

Java Synchronized Statements

- Synchronized methods incur several constraints, e.g.
  - They can yield excessive overhead due to coarse-grained serialization
  - Always synchronizes on the one & only "implicit lock" (i.e., `this`)



Java Synchronized Statements

- e.g., consider the Java Exchanger class

Defines a synchronization point where threads can pair & swap elements within pairs

```
public class Exchanger<V> {
  ...
  private synchronized
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    if (a[index] == null)
        a[index] = newSlot;
  }
```

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

See src/share/classes/java/util/concurrent/Exchanger.java

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level

Synchronized methods are "course-grained"

```
public class Exchanger<V> {
  ...
  private synchronized
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    if (a[index] == null)
        a[index] = newSlot;
  }
```

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level

```
public class Exchanger<V> {
  ...
  private synchronized
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    if (a[index] == null)
        a[index] = newSlot;
  }
```

Lazily create slot if this is the first time it's accessed

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (this) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

See docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (this) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

Synchronized statements are "finer-grained" than synchronized methods

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (this) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

Create slot outside of lock to narrow the synchronization region

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements
    - "Intrinsic lock" is often used to synchronize a statement

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (this) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

Only this statement is serialized via the "intrinsic lock"

```
  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements
    - "Intrinsic lock" is often used to synchronize a statement
    - "Explicit lock" synchronization can also be used

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

Can also synchronize using an explicit object

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

See stackoverflow.com/questions/3369287/what-is-the-difference-between-synchronized-on-lockobject-and-using-this-as-the

Java Synchronized Statements

- e.g., consider the Java Exchanger class
  - One approach synchronizes at the method level
  - Another approach synchronizes individual statements
    - "Intrinsic lock" is often used to synchronize a statement
    - "Explicit lock" synchronization can also be used
      - e.g., when the intrinsic lock is too limited or too contended

```
public class Exchanger<V> {
  ...
  private
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }
```

Can also synchronize using an explicit object

```
private volatile Slot[] arena =
   new Slot[CAPACITY];
```

See www.dre.vanderbilt.edu/~schmidt/PDF/specific-notification.pdf

Pros & Cons of Java Synchronized Statements

Pros & Cons of Java Synchronized Statements

- **Pros of synchronized statements**

See stackoverflow.com/questions/574240/is-there-an-advantage-to-use-a-synchronized-method-instead-of-a-synchronized-blo/574525#574525

Pros & Cons of Java Synchronized Statements

- **Pros of synchronized statements**
  - Allows a private field to be used as the synchronizer

```
Exchanger<Long> e
  = new Exchanger<>();

// Thread T1
for (;;)
  ... e.exchange(v);

// Thread T2
synchronized(e) {
  ...
}
```

Will not keep Thread T1 from accessing `e`'s critical section

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private volatile Slot[] arena =
     new Slot[CAPACITY];
  ...
}
```

Pros & Cons of Java Synchronized Statements

- **Pros of synchronized statements**
  - Allows a private field to be used as the synchronizer
  - Enables finer-grained control of synchronization

Only synchronize what is absolutely necessary

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private volatile Slot[] arena =
     new Slot[CAPACITY];
  ...
}
```

Pros & Cons of Java Synchronized Statements

- **Cons of synchronized statements**

Pros & Cons of Java Synchronized Statements

- **Cons of synchronized statements**
  - The syntax is a bit more complicated

This code is harder to understand

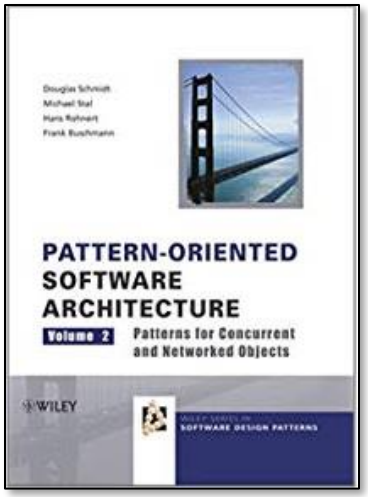
```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  } ...

public class Exchanger<V> {
  ...
  private synchronized
          void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    if (a[index] == null)
        a[index] = newSlot;
  }
```

Implementing the Double-Checked Locking Pattern

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*




```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

See en.wikipedia.org/wiki/Double-checked_locking

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*
  - Synchronization is done "lazily" when initialization is first performed

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

See en.wikipedia.org/wiki/Lazy_initialization

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*
  - Synchronization is done "lazily" when initialization is first performed

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

Double-Checked Locking optimization is done here

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*
  - Synchronization is done "lazily" when initialization is first performed

There's no need to synchronize this check since reference reads & writes are atomic

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

See docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.7

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*
  - Synchronization is done "lazily" when initialization is first performed

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

A new slot is created only if the current slot is null

Implementing the Double-Checked Locking Pattern

- Synchronized statements can be used to implement patterns like *Double-Checked Locking*
  - Synchronization is done "lazily" when initialization is first performed

Only synchronize when the slot is first created

```
public class Exchanger<V> {
  ...
  private void createSlot(int index){
    final Slot newSlot = new Slot();
    final Slot[] a = arena;
    synchronized (a) {
      if (a[index] == null)
        a[index] = newSlot;
    }
  }

  private Object doExchange(...) {
    ...
    final Slot slot = arena[index];
    if (slot == null)
      // Lazily initialize slots
      createSlot(index);

  private volatile Slot[] arena =
     new Slot[CAPACITY];
```

End of Java Monitor Object Synchronized Statements