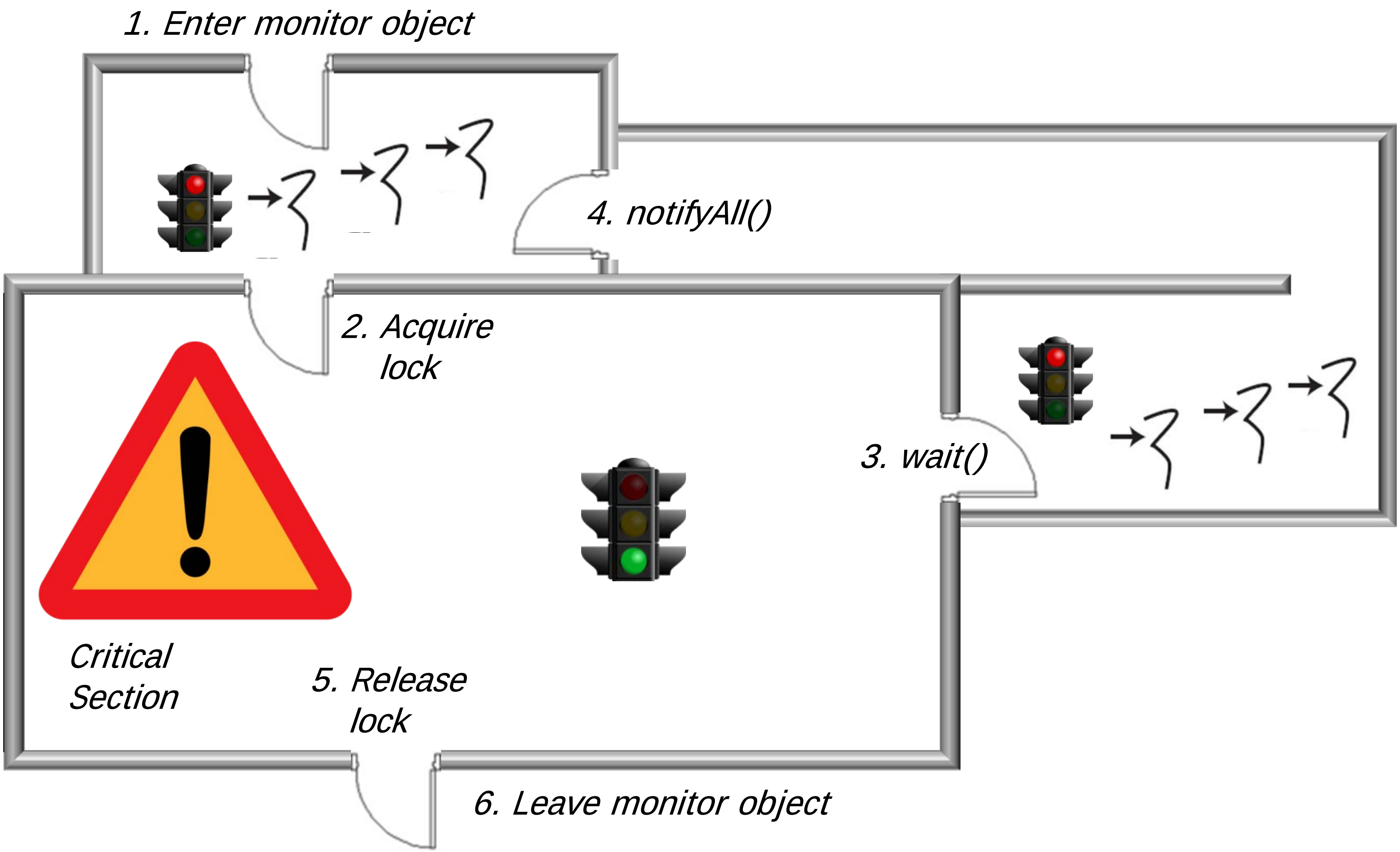
Java Concurrency: Java Built-in Monitor Objects (Part 1)

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

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

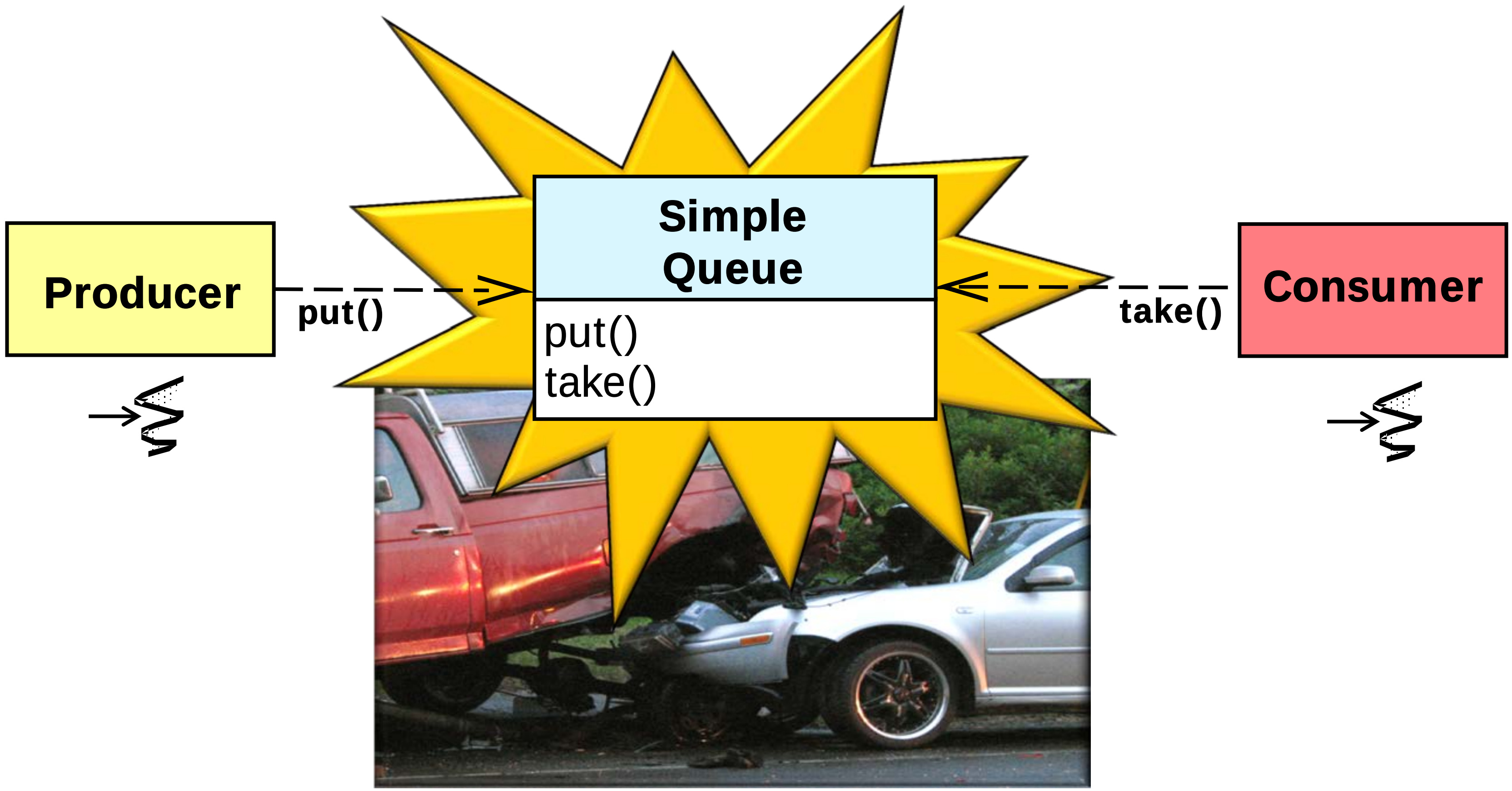
Learning Objectives in this Part of the Module

- Recognize how Java built-in monitor objects can ensure mutual exclusion & coordination between threads running in a concurrent program



Learning Objectives in this Part of the Module

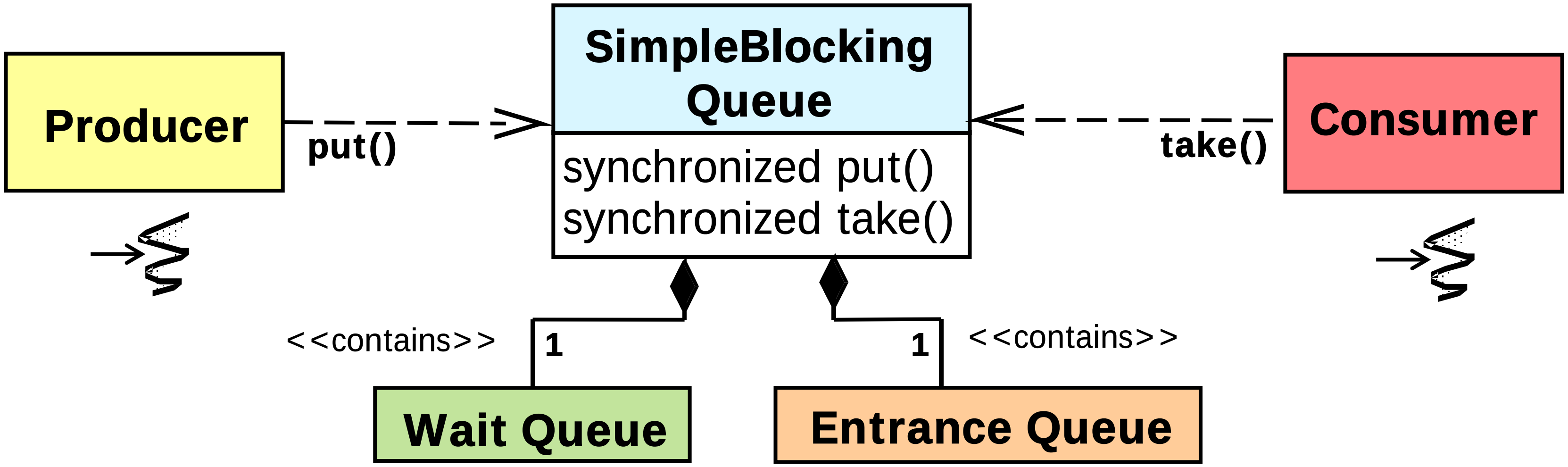
- Recognize how Java built-in monitor objects can ensure mutual exclusion & coordination between threads running in a concurrent program
- Understand how to fix a buggy concurrent Java program



Concurrent calls to put() & take() corrupt internal state in the SimpleQueue fields

Learning Objectives in this Part of the Module

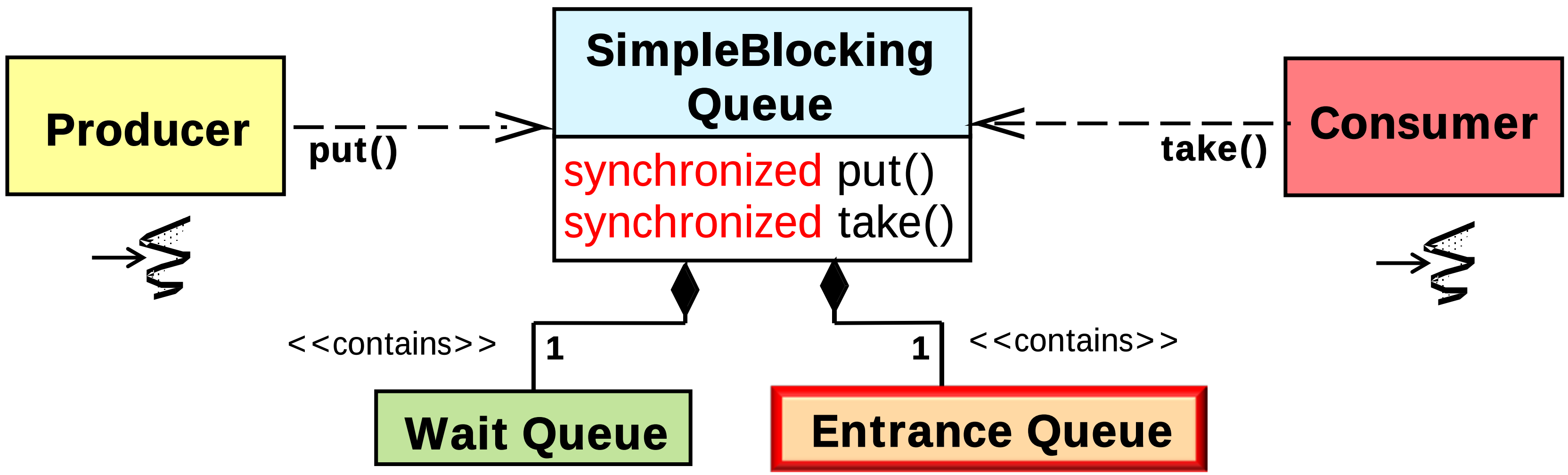
- Recognize how Java built-in monitor objects can ensure mutual exclusion & coordination between threads running in a concurrent program
- Understand how to fix a buggy concurrent Java program using various Java built-in monitor object features to synchronize the queue properly



See github.com/douglascraigschmidt/LiveLessons/tree/master/BusySynchronizedQueue

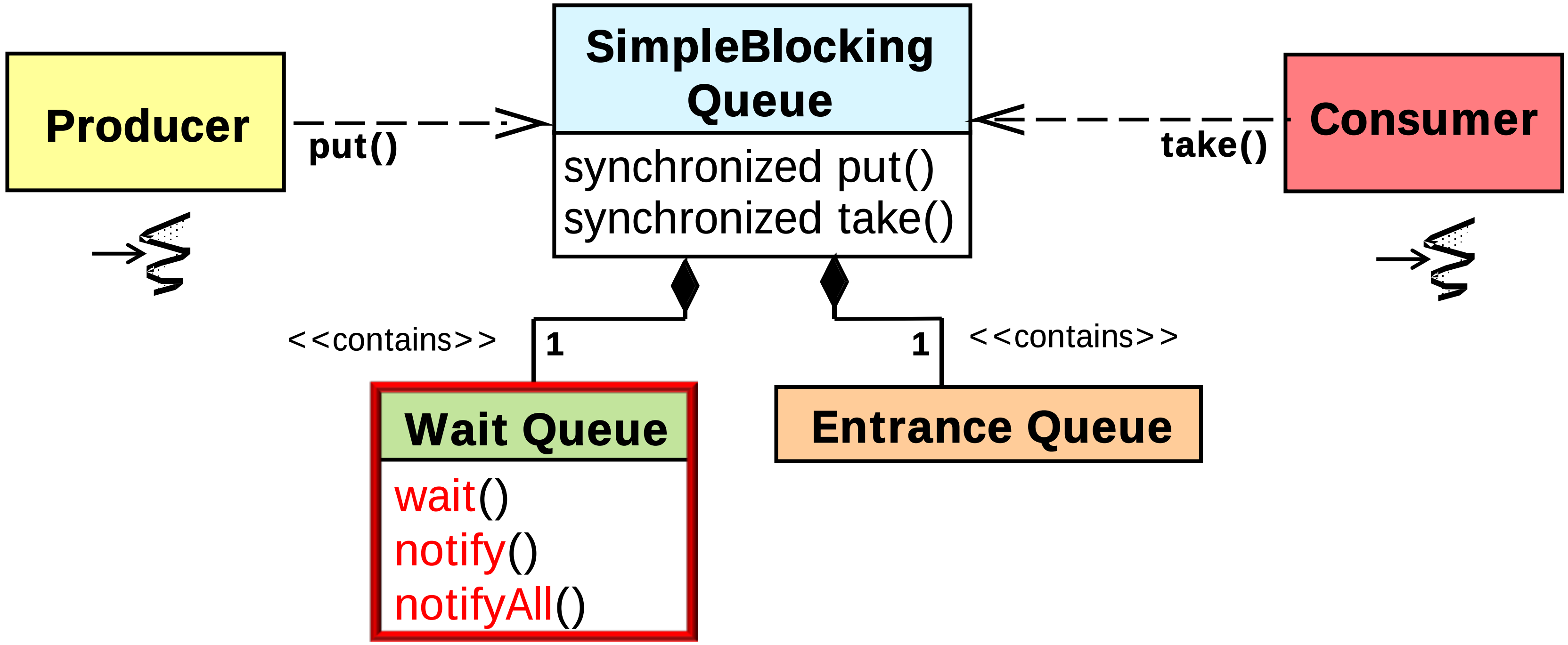
Learning Objectives in this Part of the Module

- Recognize how Java built-in monitor objects can ensure mutual exclusion & coordination between threads running in a concurrent program
- Understand how to fix a buggy concurrent Java program using various Java built-in monitor object features to synchronize the queue properly, e.g.
  - Synchronized methods/statements



Learning Objectives in this Part of the Module

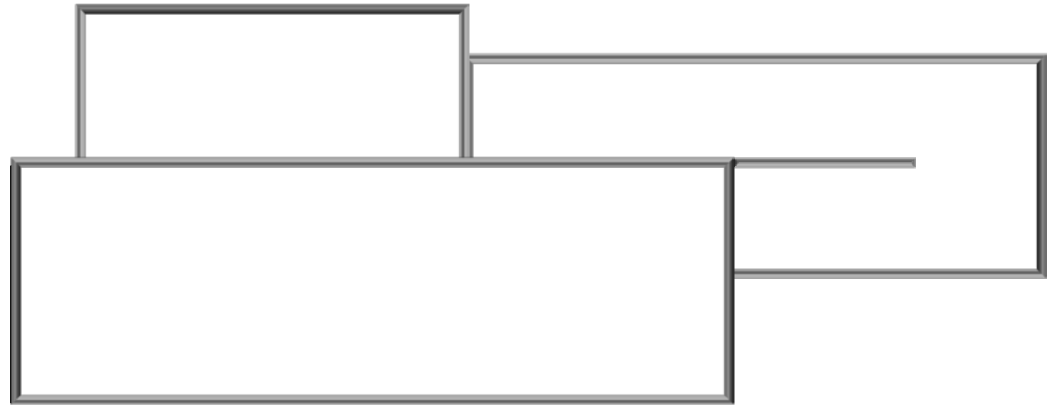
- Recognize how Java built-in monitor objects can ensure mutual exclusion & coordination between threads running in a concurrent program
- Understand how to fix a buggy concurrent Java program using various Java built-in monitor object features to synchronize the queue properly, e.g.
  - Synchronized methods/statements
  - Waiting & notification mechanisms

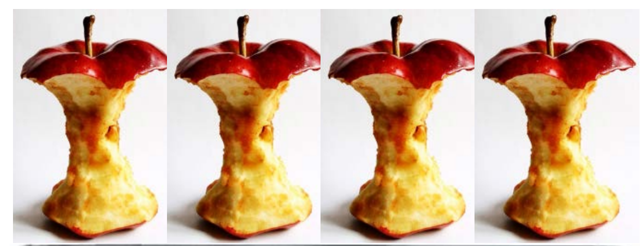


Overview of Monitors

Overview of Monitors

- A monitor is a concurrency control construct used for synchronization

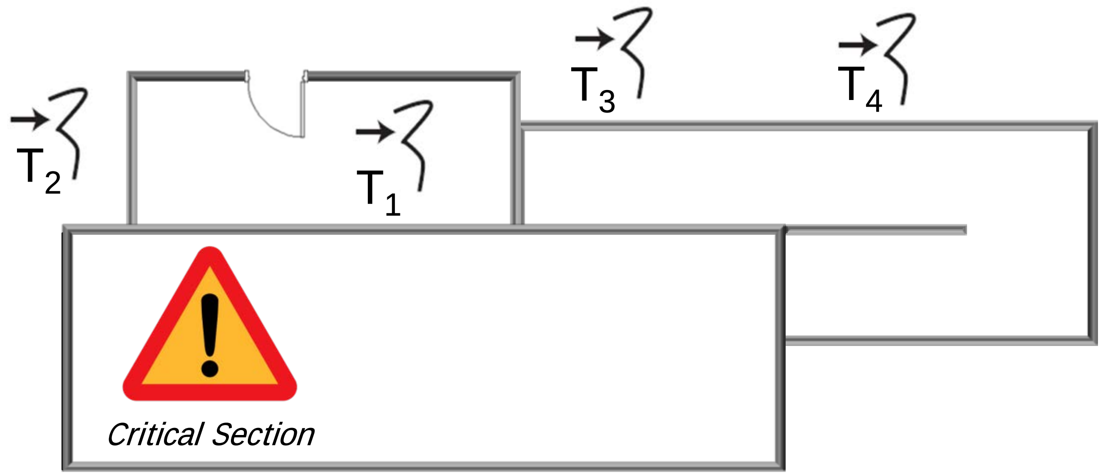


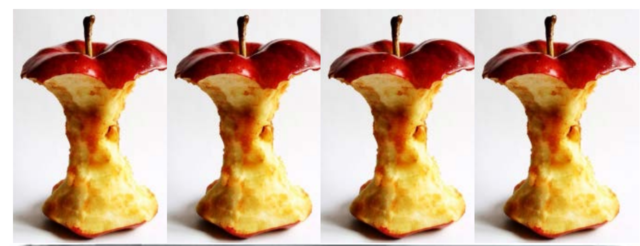


See en.wikipedia.org/wiki/Monitor_(synchronization)

Overview of Monitors

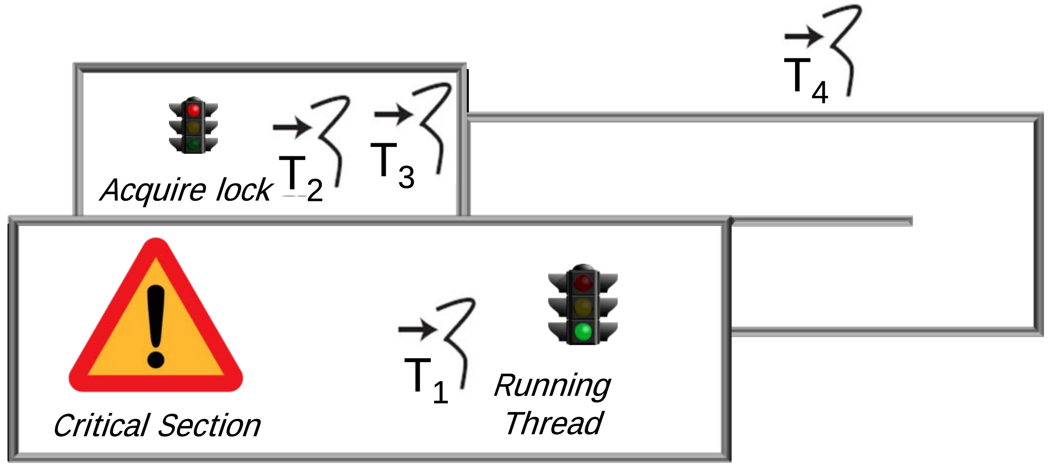
- A monitor is a concurrency control construct used for synchronization
- A monitor provides three capabilities to concurrent programs

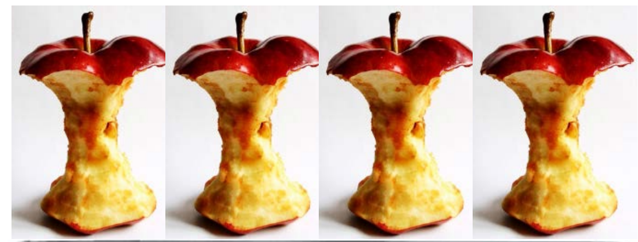




Overview of Monitors

- A monitor is a concurrency control construct used for synchronization
- A monitor provides three capabilities to concurrent programs
  1. One thread at a time to have mutually exclusive access to a critical section

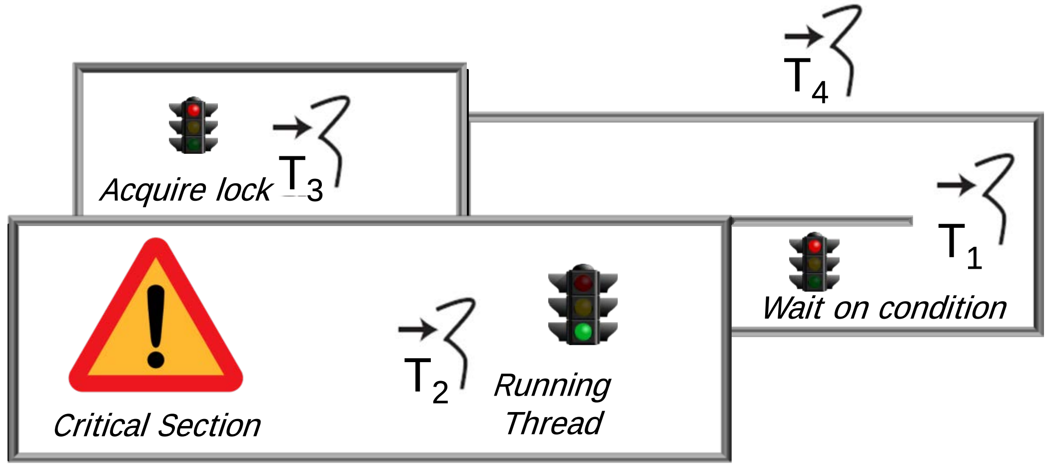


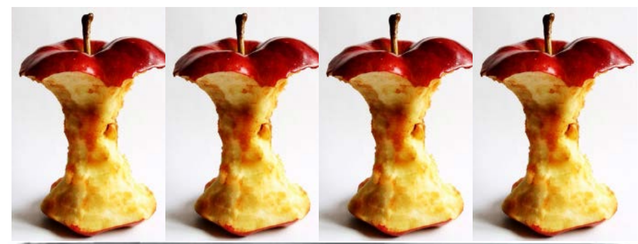


See en.wikipedia.org/wiki/Critical_section

Overview of Monitors

- A monitor is a concurrency control construct used for synchronization
- A monitor provides three capabilities to concurrent programs
  1. One thread at a time to have mutually exclusive access to a critical section
  2. Threads running in monitor block awaiting certain conditions to become true

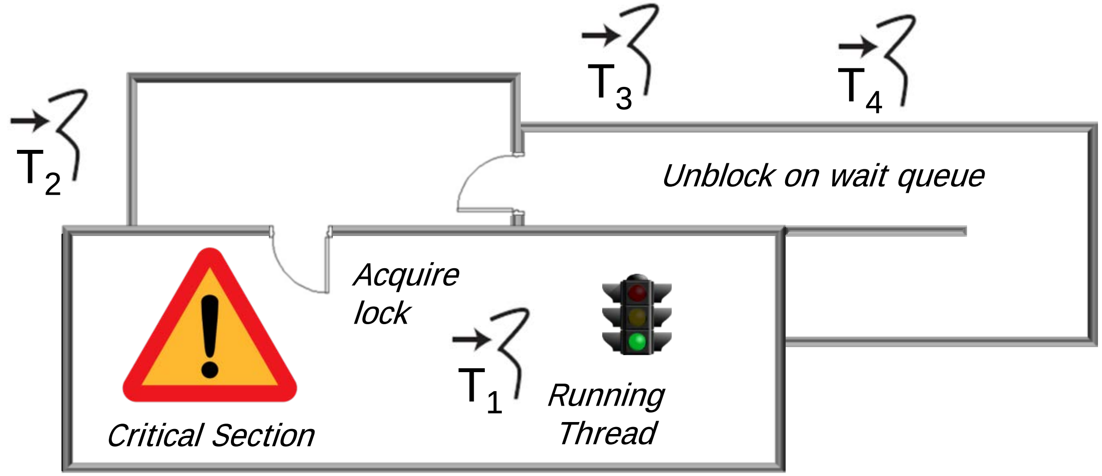


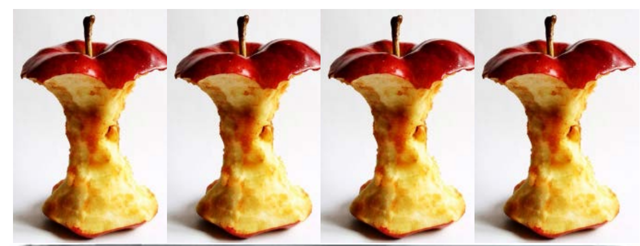


Overview of Monitors

- A monitor is a concurrency control construct used for synchronization
- A monitor provides three capabilities to concurrent programs

   1. One thread at a time to have mutually exclusive access to a critical section
   2. Threads running in monitor block awaiting certain conditions to become true
   3. Thread notifying others threads that conditions they're waiting on in monitor have been met

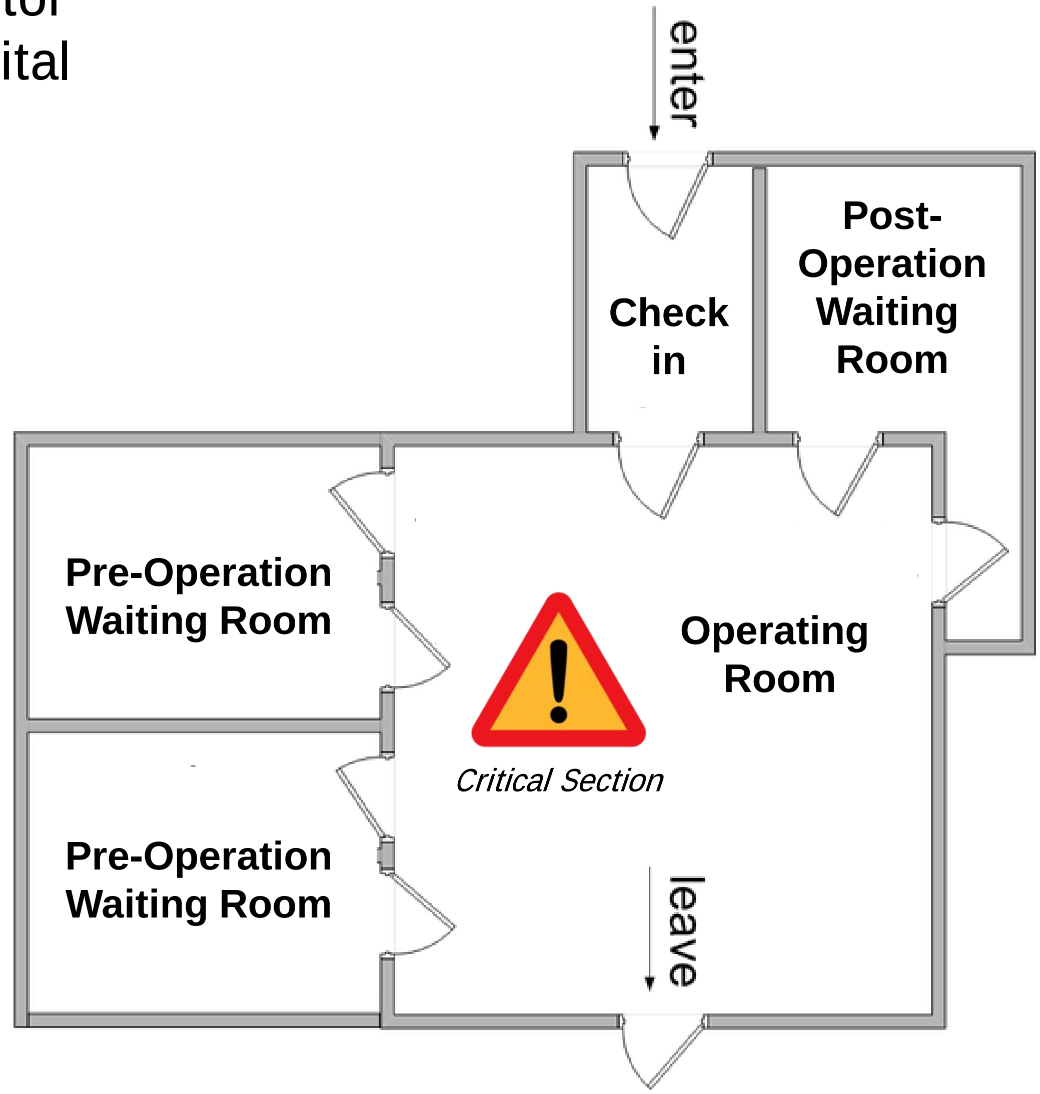




Human Known Use of Monitors

Human Know Use of Monitors

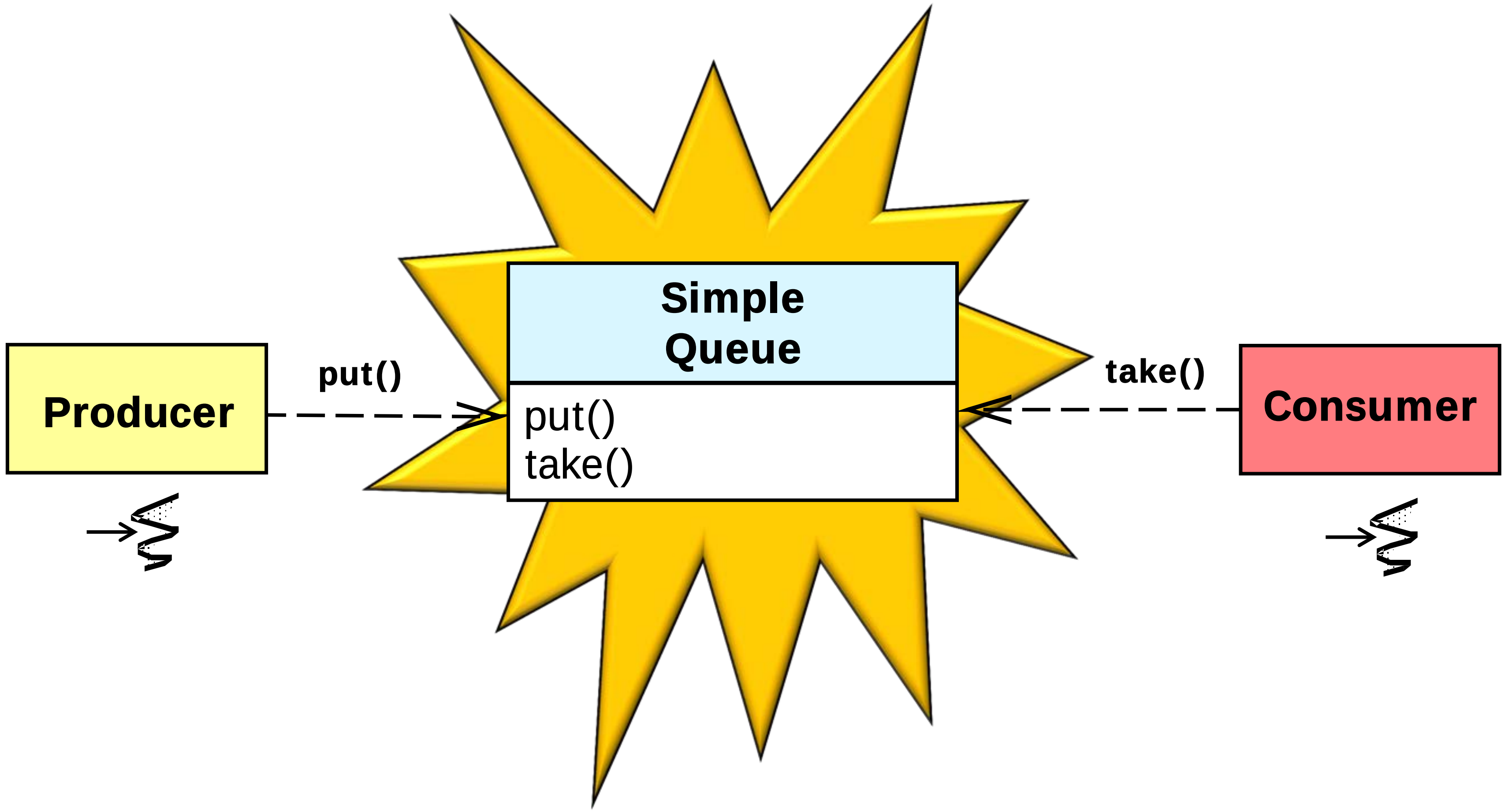
- A human known use of a monitor is an operating room in a hospital



Motivating Example: A Buggy Producer/ Consumer Program (Part 1)

Buggy Producer/Consumer Program Overview

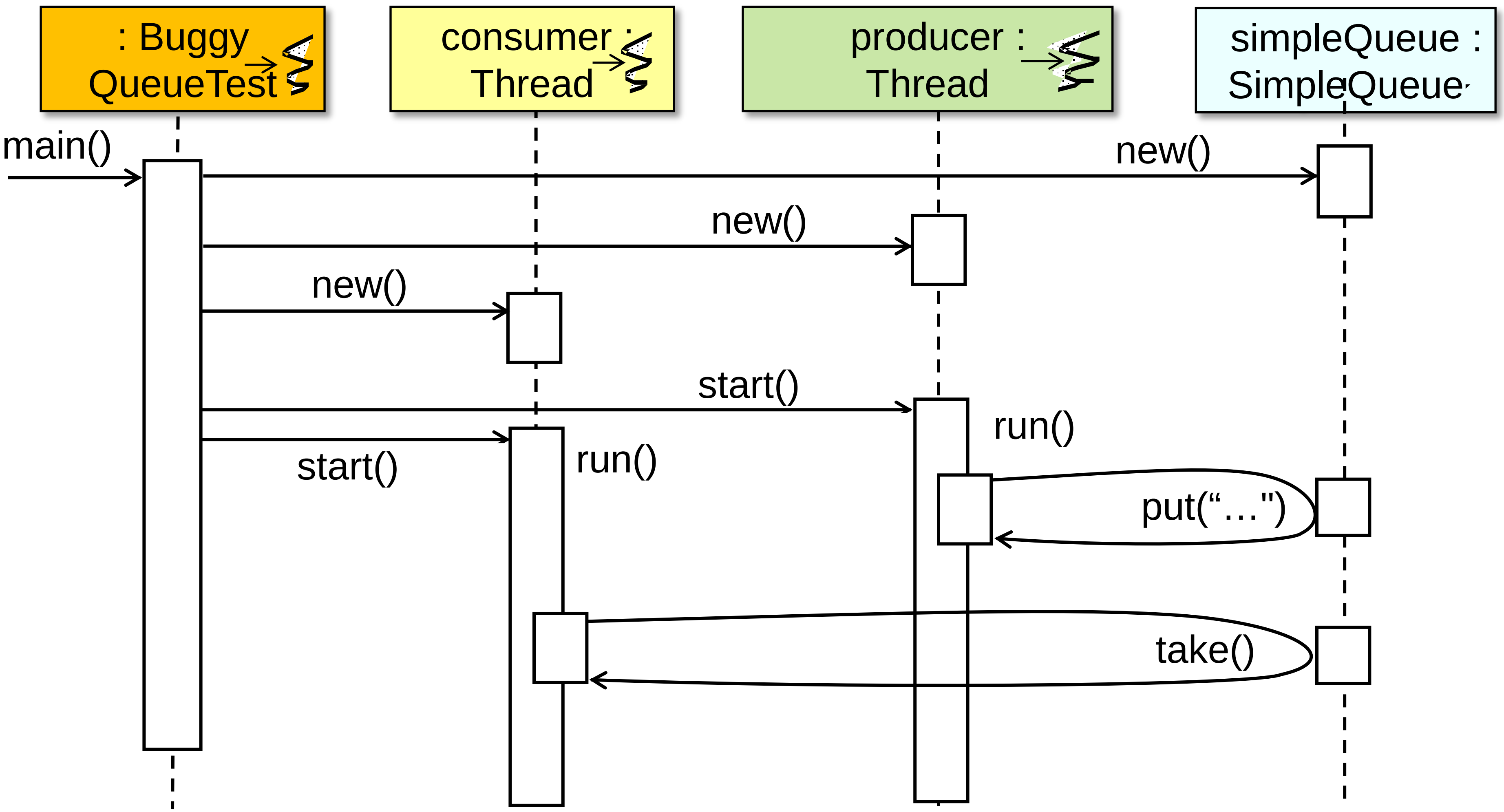
- A simple concurrent producer/consumer program that attempts to pass messages via an object modeled on the Java ArrayBlockingQueue class



See docs.oracle.com/javase/7/docs/api/java/util/concurrent/ArrayBlockingQueue.html

Buggy Producer/Consumer Program Overview

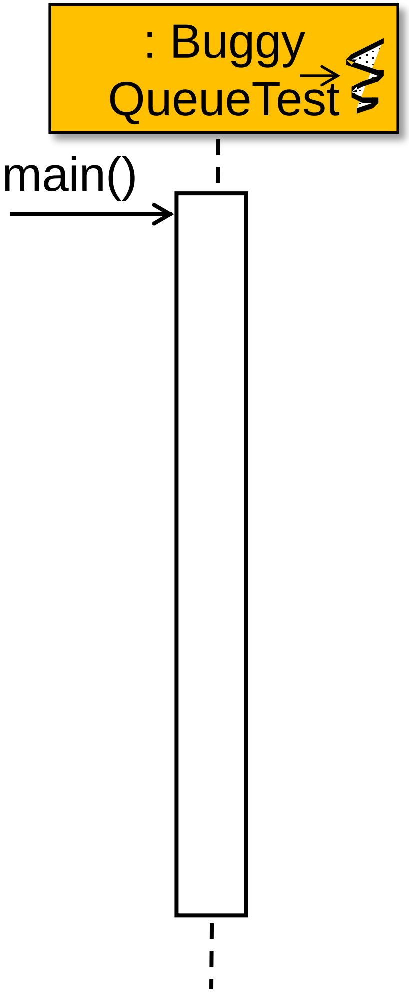
- UML sequence diagram showing the design of the buggy producer/consumer



See github.com/douglascraigschmidt/LiveLessons/tree/master/BuggyQueue

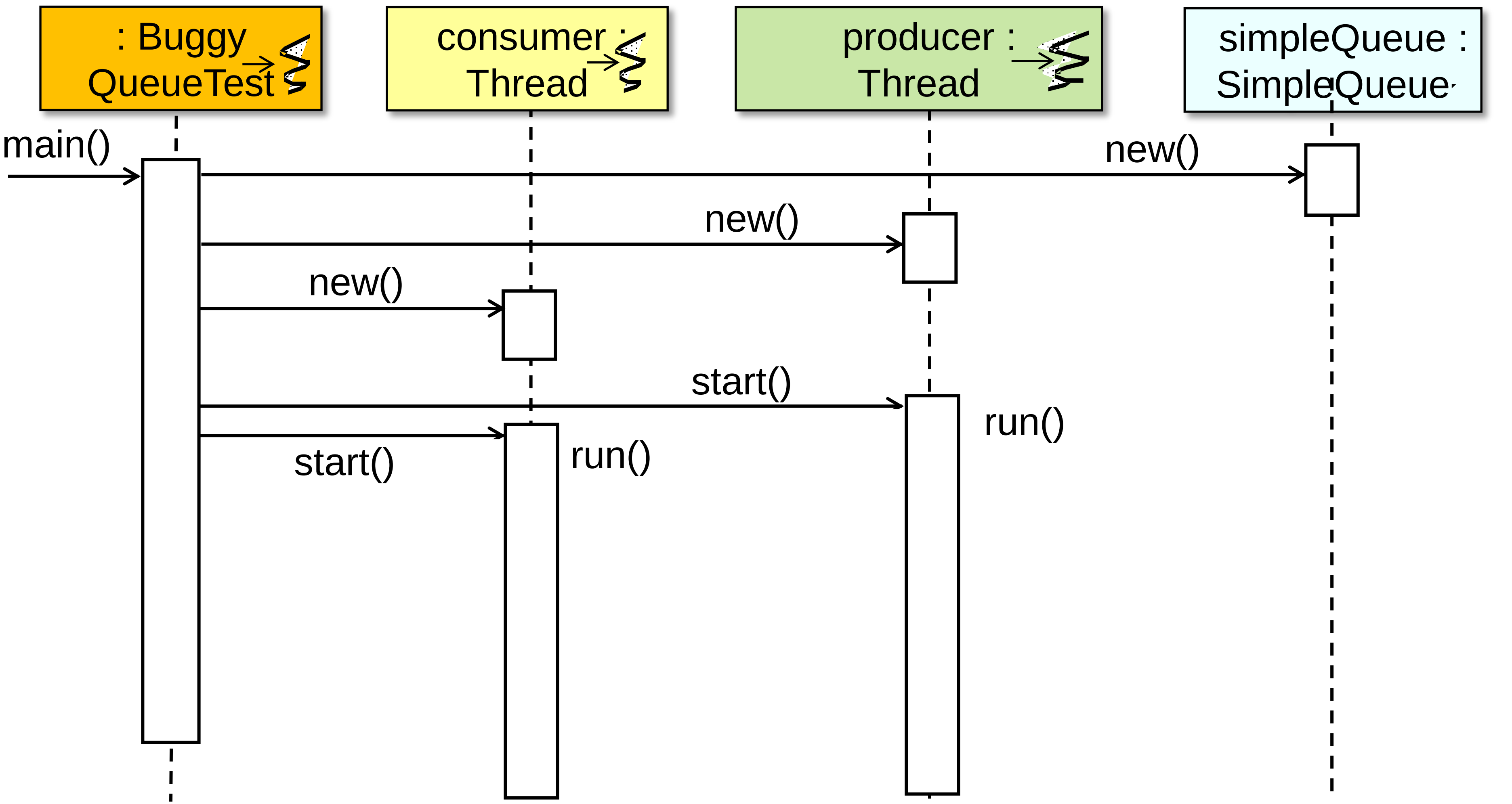
Buggy Producer/Consumer Program Overview

- UML sequence diagram showing the design of the buggy producer/consumer



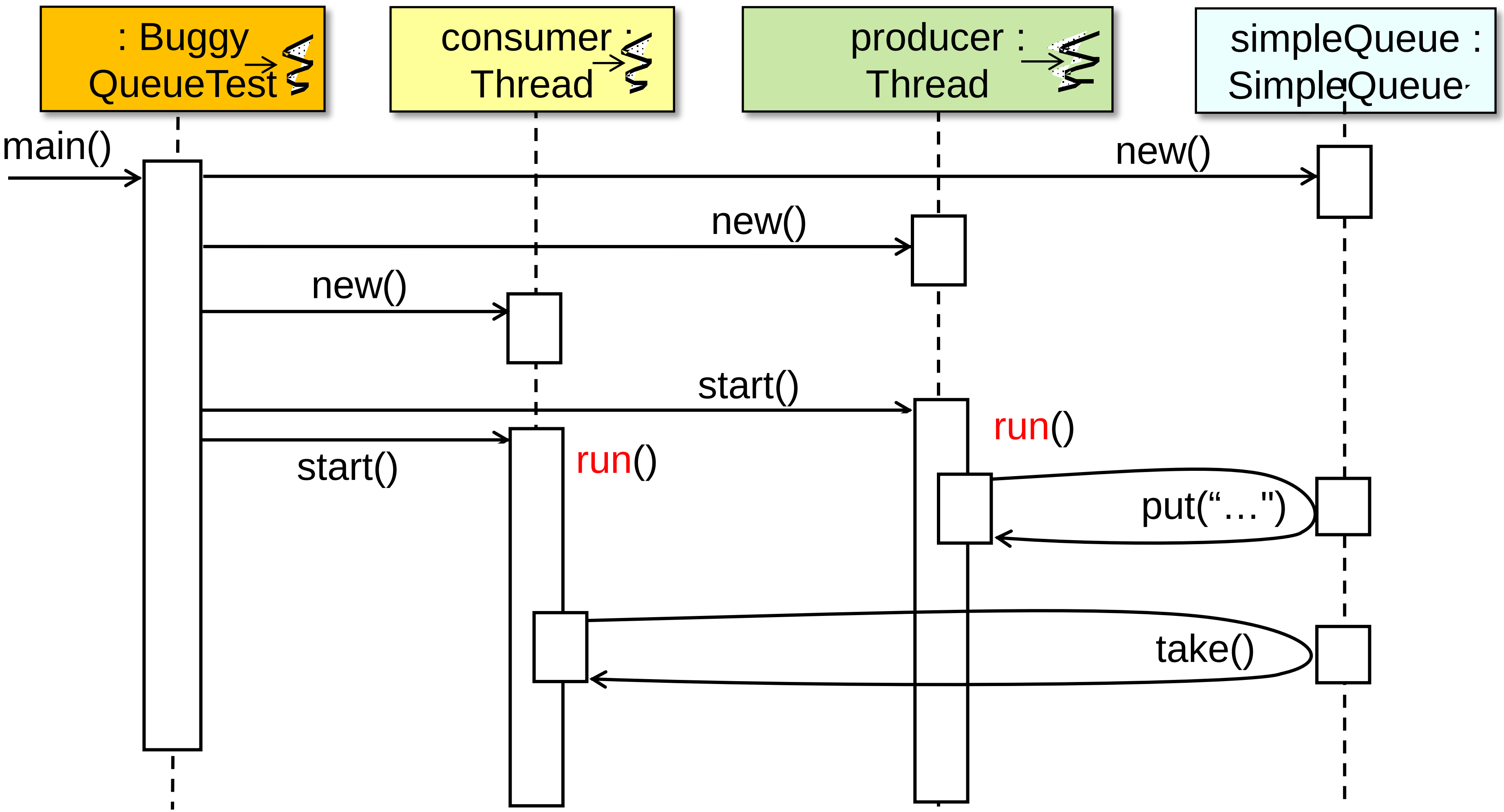
Buggy Producer/Consumer Program Overview

- UML sequence diagram showing the design of the buggy producer/consumer



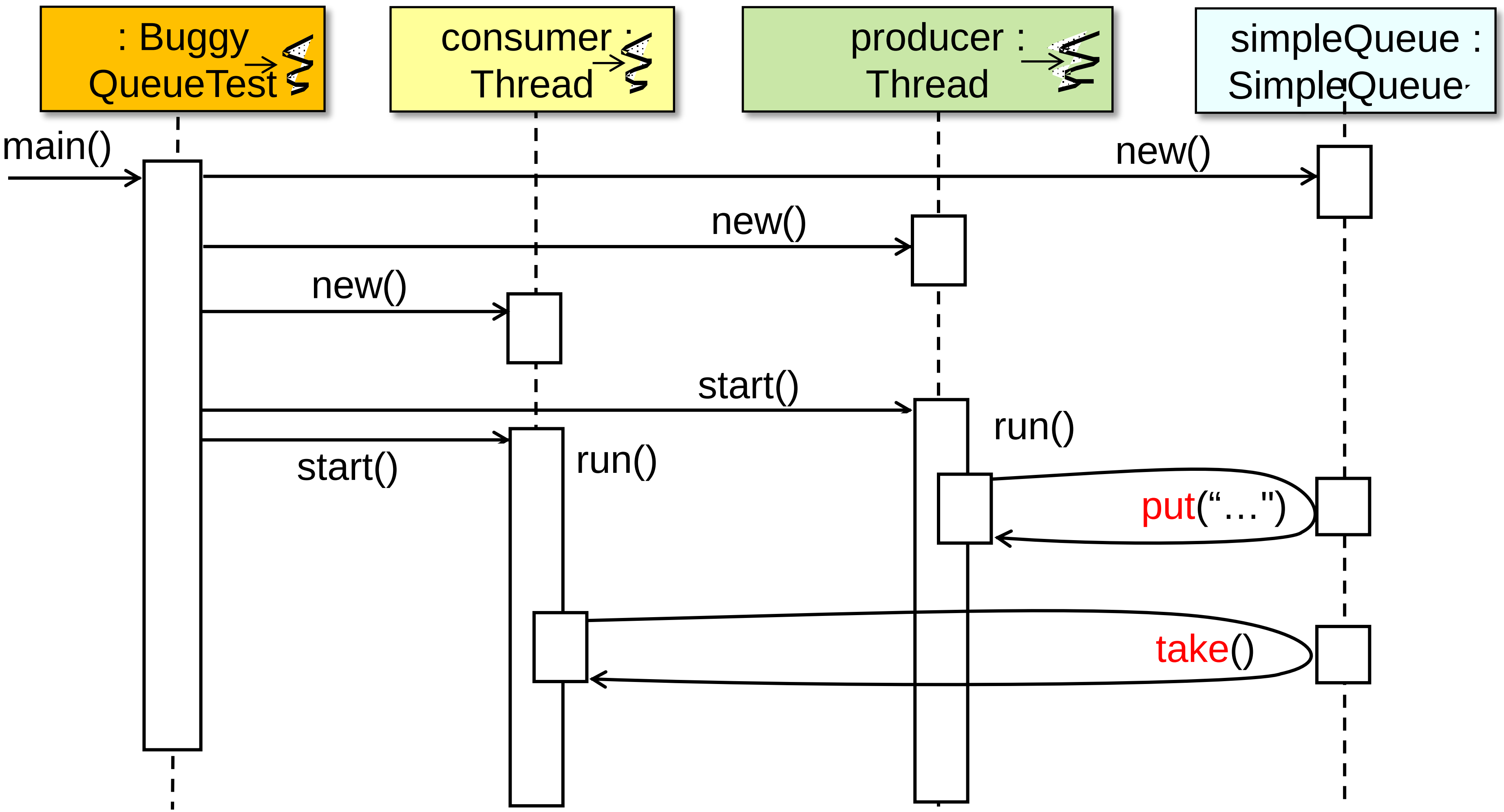
Buggy Producer/Consumer Program Overview

- UML sequence diagram showing the design of the buggy producer/consumer



Buggy Producer/Consumer Program Overview

- UML sequence diagram showing the design of the buggy producer/consumer



Motivating Example: A Buggy Producer/ Consumer Program (Part 2)

Implementation of the Buggy Producer/Consumer

- Implementation of the SimpleQueue class

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();


  public void put(E msg){ mList.add(msg); }



  public E take(){ return mList.remove(0); }
  ...
}
```

Implementation of the Buggy Producer/Consumer

- Implementation of the SimpleQueue class

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();


  public void put(E msg){ mList.add(msg); }



  public E take(){ return mList.remove(0); }
  ...
}
```

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

Implementation of the Buggy Producer/Consumer

- Implementation of the SimpleQueue class

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();

  public void put(E msg){ mList.add(msg); }

  public E take(){ return mList.remove(0); }
  ...
}
```

Resizable-array implementation

See docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Implementation of the Buggy Producer/Consumer

- Implementation of the SimpleQueue class

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();


  public void put(E msg){ mList.add(msg); }


  public E take(){ return mList.remove(0); }
  ...
}
```

Non-synchronized public methods

Implementation of the Buggy Producer/Consumer

- Implementation of the SimpleQueue class

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();


  public void put(E msg){ mList.add(msg); }
```

Add & remove elements into/from the queue

```
  public E take(){ return mList.remove(0); }
  ...
}
```

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    sQue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(sQue.take());
                }});

  producer.start();
  consumer.start(); ...
```

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    sQue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(sQue.take());
                }});

  producer.start();
  consumer.start(); ...
```

Create a SimpleQueue

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

Create producer & consumer threads

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      sQue.put(Integer.toString(i));
                  }});
     Thread consumer =
       new Thread(new Runnable(){
                    public void run(){
                      for(int i = 0; i < mMaxIterations; i++)
                        System.out.println(sQue.take());
                  }});

  producer.start();
  consumer.start(); ...
```

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                 public void run(){
                   for(int i = 0; i < mMaxIterations; i++)
                     sQue.put(Integer.toString(i));
                 }});
     Thread consumer =
       new Thread(new Runnable(){
                   public void run(){
                     for(int i = 0; i < mMaxIterations; i++)
                       System.out.println(sQue.take());
                 }});

  producer.start();
  consumer.start(); ...
```

Start producer & consumer threads

Implementation of the Buggy Producer/Consumer

- Implementation of the Main class

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    sQue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(sQue.take());
                }});

  producer.start();
  consumer.start(); ...
```

Produce & consume Strings concurrently

Evaluating the Buggy Producer/Consumer

Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?

```
public static void main(String argv[]) {
  final SimpleQueue<String> sQue = new SimpleQueue<String>();

  Thread producer =
     new Thread(new Runnable(){
                public void run(){
                  for(int i = 0; i < mMaxIterations; i++)
                    sQue.put(Integer.toString(i));
                }});
     Thread consumer =
       new Thread(new Runnable(){
                  public void run(){
                    for(int i = 0; i < mMaxIterations; i++)
                      System.out.println(sQue.take());
                }});

  producer.start();
  consumer.start(); ...
```

Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?

```
public static void
  final SimpleQueue

  Thread producer =
     new Thread(new
                pub
                  f

                }})
     Thread consume
       new Thread(n
                  p
```

```
Exception in thread "Thread-1"
java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
        at java.util.ArrayList.RangeCheck(ArrayList.java:547)
        at java.util.ArrayList.remove(ArrayList.java:387)
        at Main$sQue.take(Main.java:16)
        at Main$2.run(Main.java:34)
        at java.lang.Thread.run(Thread.java:662)
```

Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?

```
static class SimpleQueue<E> implements BlockingQueue<E> {
  private List<E> mList = new ArrayList<E>();


  public void put(E msg){ mList.add(msg); }
```

There's no protection against critical sections being run by multiple threads concurrently

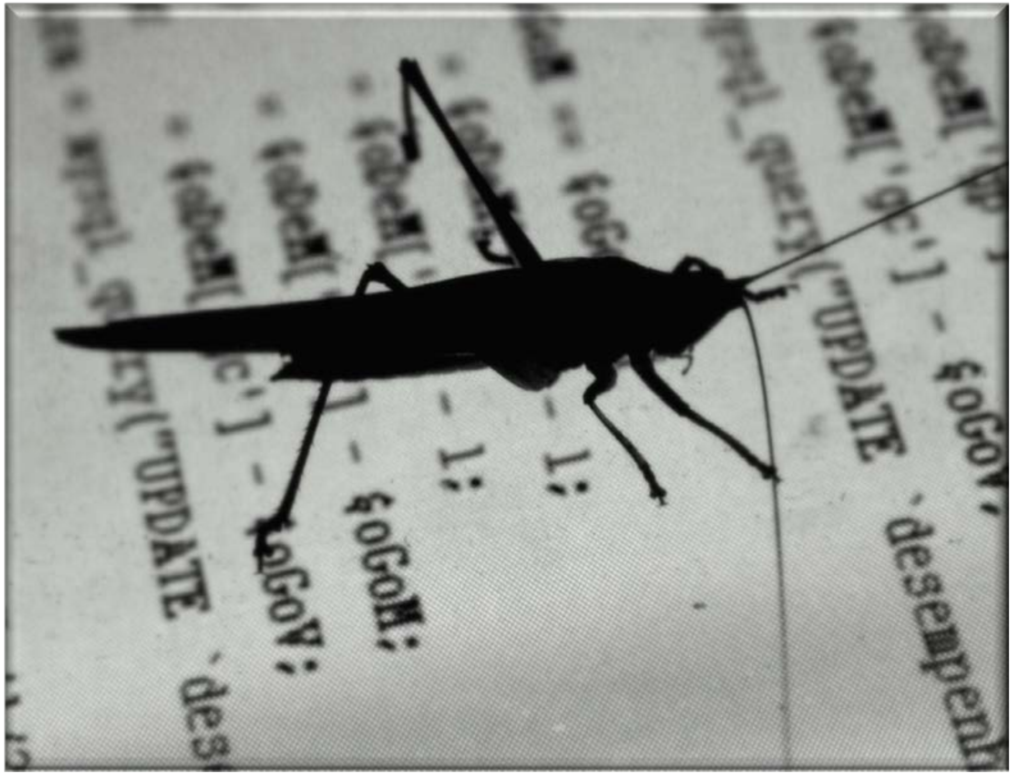
```
  public E take(){ return mList.remove(0); }
  ...
}
```

Note that this implementation is not synchronized. If multiple threads access an `ArrayList` instance concurrently, and at least one of the threads modifies the list structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.)

See docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?
- These race conditions are hard to detect & debug, due to inherent complexities of synchronization

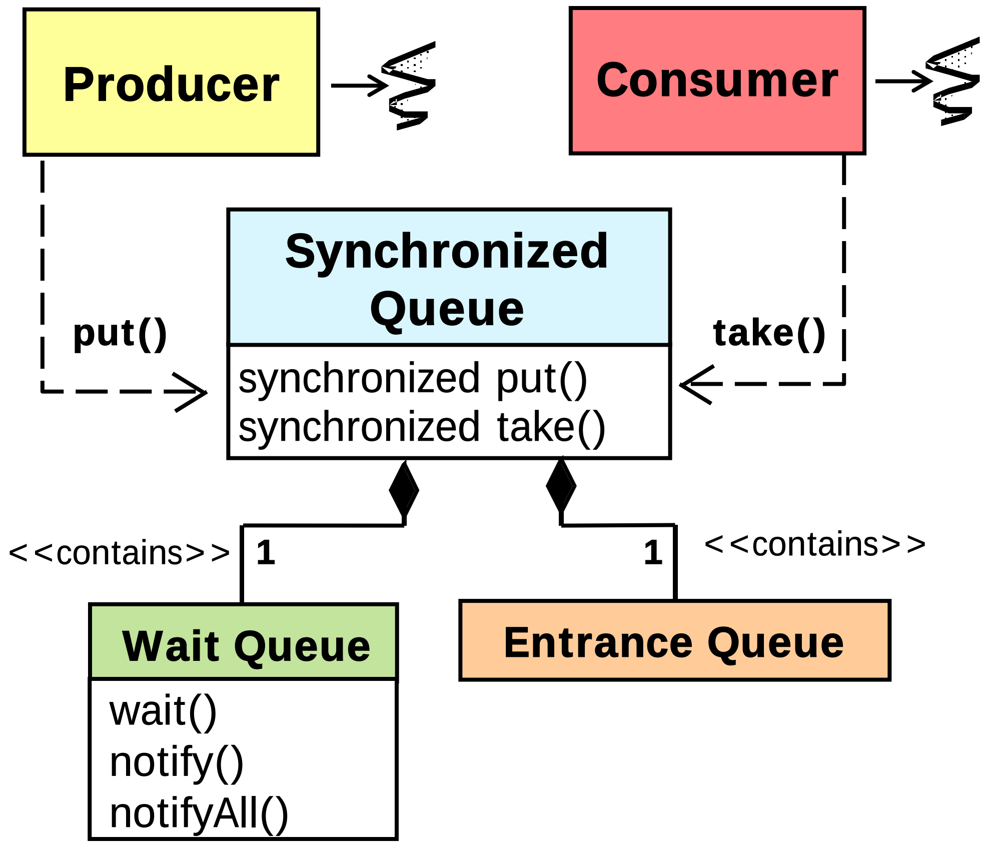


Evaluating the Buggy Producer/Consumer

- Key question: what's the output & why?
- These race conditions are hard to detect & debug, due to inherent complexities of synchronization
  - Development & quality assurance of concurrent programs is tedious, error-prone, & non-portable

Evaluating the Buggy Producer/Consumer

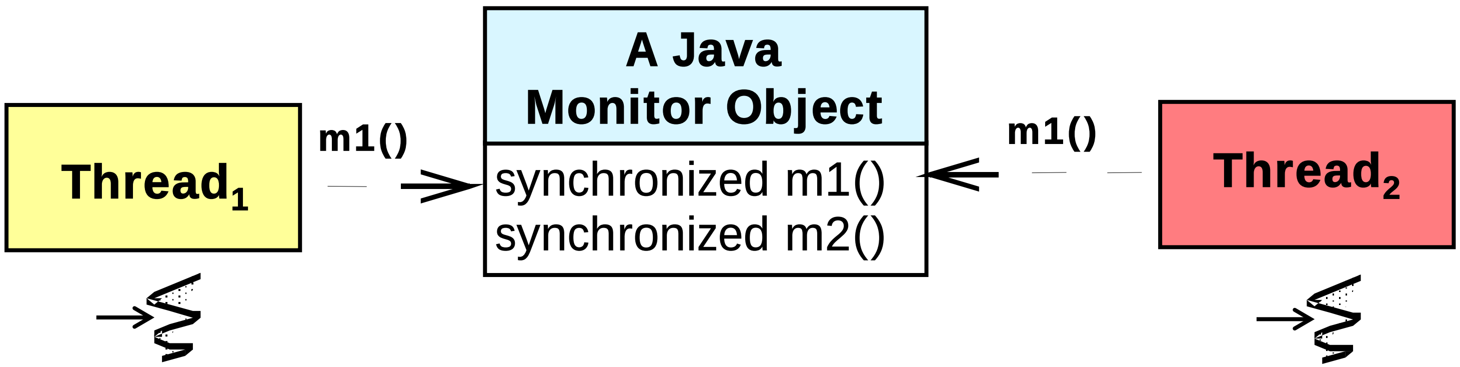
- Key question: what's the output & why?
- These race conditions are hard to detect & debug, due to inherent complexities of synchronization
- We'll fix these problems by applying various Java built-in monitor object mechanisms



Overview of Built-in Java Monitor Objects

Overview of Java Built-in Monitor Objects

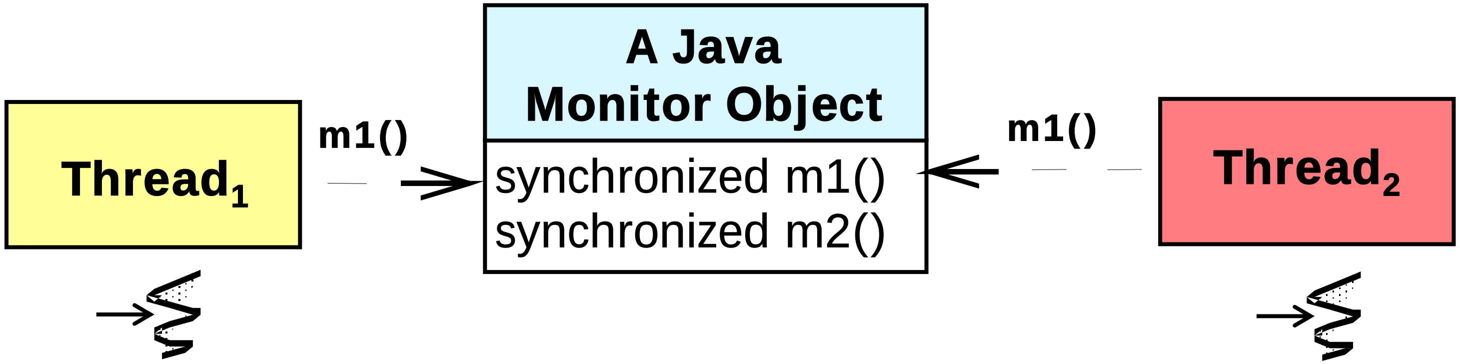
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization



See www.artima.com/insidejvm/ed2/threadsynch.html

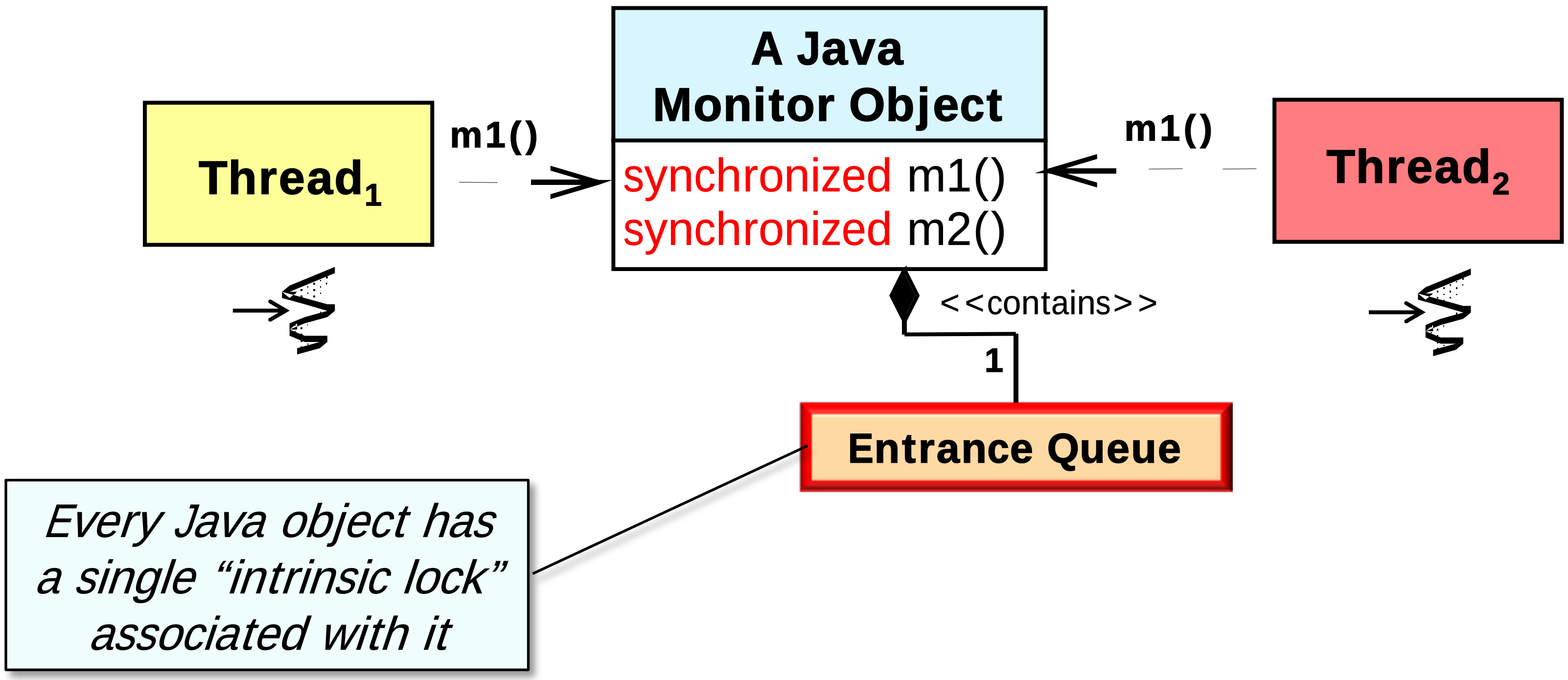
Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions



Overview of Java Built-in Monitor Objects

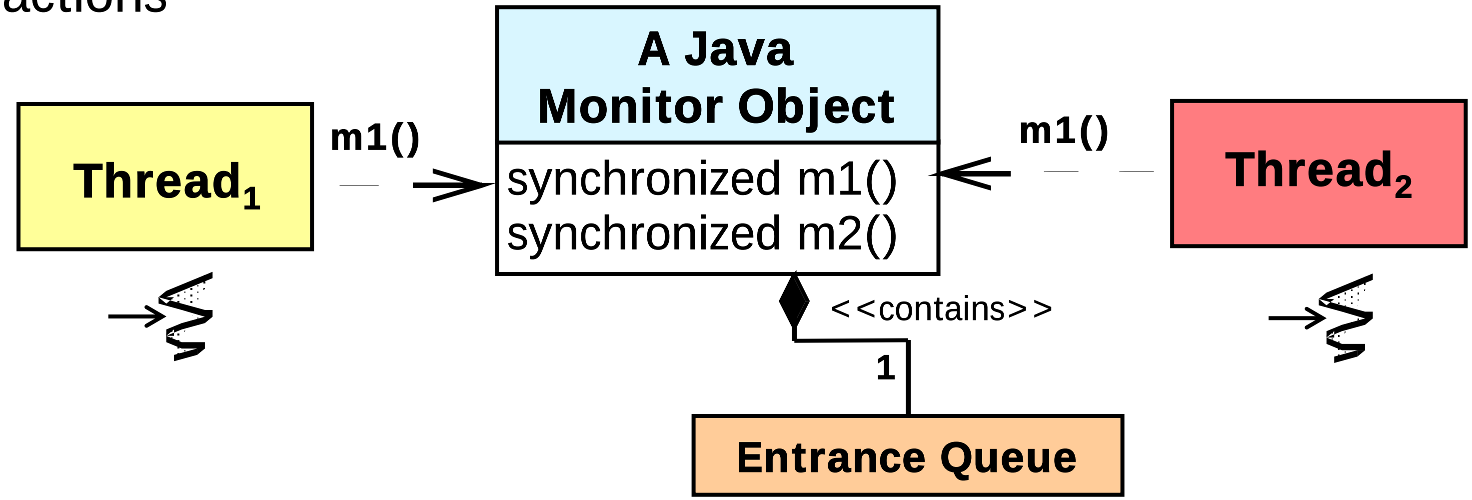
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions



The JVM supports mutual exclusion via an entrance queue & synchronized methods

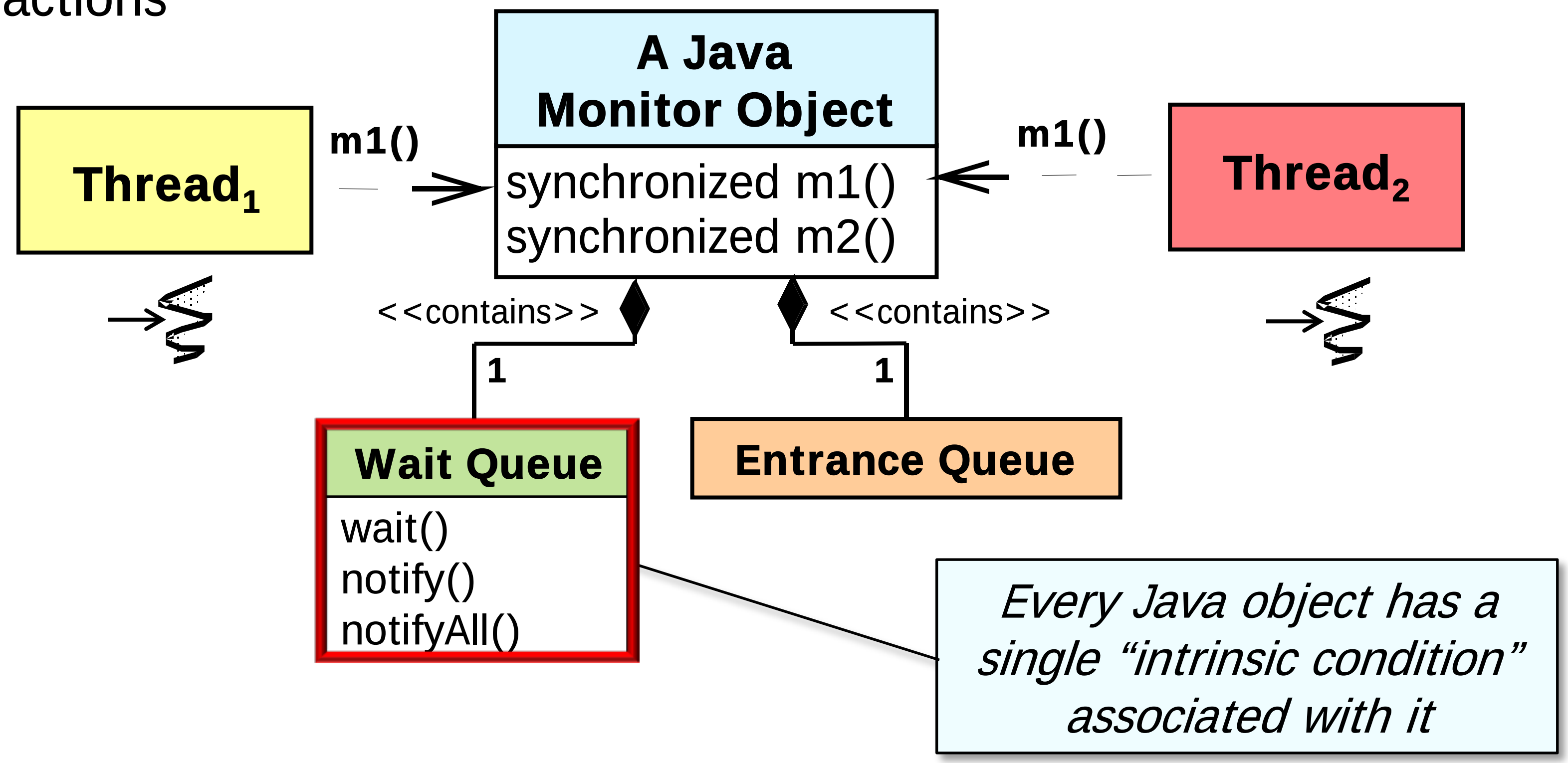
Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions
  - **Coordination** – enables threads to cooperatively schedule their interactions



Overview of Java Built-in Monitor Objects

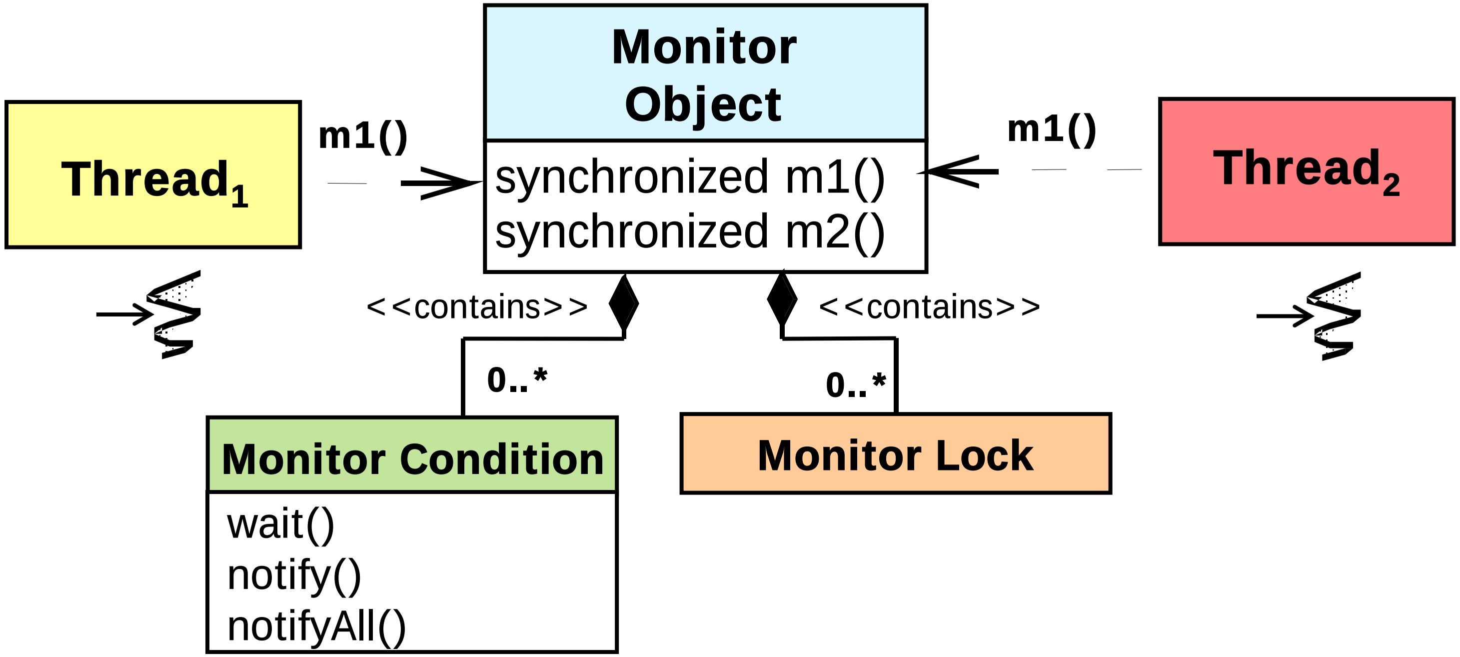
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions
  - **Coordination** – enables threads to cooperatively schedule their interactions



The JVM supports coordination via a wait queue & notification mechanisms

Overview of Java Built-in Monitor Objects

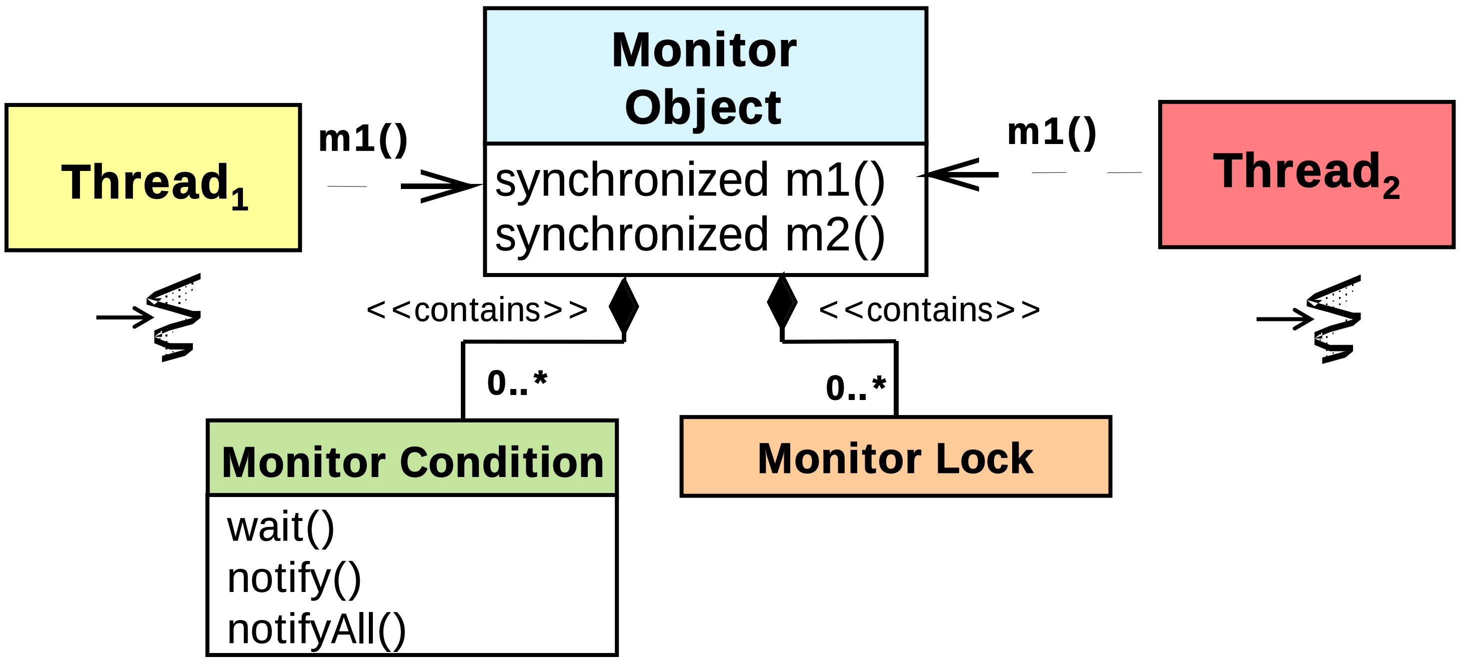
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
- These mechanisms implement a variant of the *Monitor Object* pattern



See www.dre.vanderbilt.edu/~schmidt/PDF/monitor.pdf

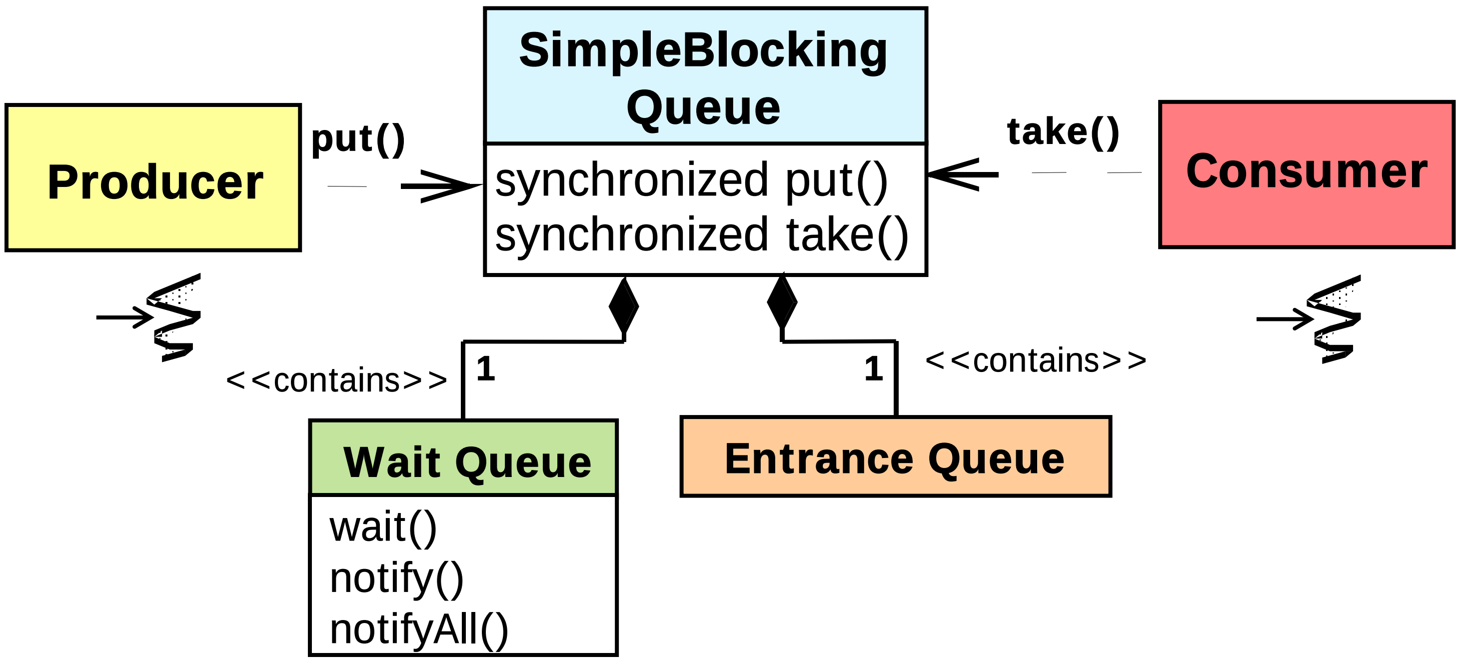
Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
- These mechanisms implement a variant of the *Monitor Object* pattern
  - Intent – Ensures that only one method runs within an object & allows an object's methods to cooperatively schedule their execution sequences



Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
- These mechanisms implement a variant of the *Monitor Object* pattern
- Java built-in monitor objects can implement a better solution to the earlier buggy SimpleQueue & upcoming inefficient BusySynchronizedQueue solutions



Java Built-in Synchronized Methods

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  private final List<E> mList;
  private final int mCapacity;

  BusySynchronizedQueue(int capacity) {
    mList = new ArrayList<E>();
    mCapacity = capacity;
  }
  ...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/BusySynchronizedQueue

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  private final List<E> mList;
  private final int mCapacity;

  BusySynchronizedQueue(int capacity){
    mList = new ArrayList<E>();
    mCapacity = capacity;
  }
  ...
```

See docs.oracle.com/javase/7/docs/api/java/util/concurrent/BlockingQueue.html

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  private final List<E> mList;
  private final int mCapacity;

  BusySynchronizedQueue(int capacity){
    mList = new ArrayList<E>();
    mCapacity = capacity;
  }
  ...
```

This internal state must be protected against race conditions

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  private final List<E> mList;
  private final int mCapacity;

  BusySynchronizedQueue(int capacity){
    mList = new ArrayList<E>();
    mCapacity = capacity;
  }
  ...
```

The constructor initializes the internal state

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms
- Methods in a built-in monitor object must be marked with the synchronized keyword

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  ...
  public synchronized void put(E msg)
  { ... }

  public synchronized E take()
  { ... }

  public synchronized boolean isEmpty()
  { ... }

  ...
```

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

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms
- Methods in a built-in monitor object must be marked with the synchronized keyword
  - Access to a synchronized method is serialized w/other synchronized methods

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  ...
  public synchronized void put(E msg)
  { ... }

  public synchronized E take()
  { ... }

  public synchronized boolean isEmpty()
  { ... }

  ...
```

Java Built-in Synchronized Methods

- The BusySynchronizedQueue class showcases Java built-in monitor object mechanisms
- Methods in a built-in monitor object must be marked with the synchronized keyword
  - Access to a synchronized method is serialized w/other synchronized methods
  - When used in the method signature, access to entire body of the method is serialized

```
class BusySynchronizedQueue<E>
      implements BlockingQueue<E> {
  ...
  public synchronized void put(E msg)
  { ... }

  public synchronized E take()
  { ... }

  public synchronized boolean isEmpty()
  { return mList.size() == 0; }


  ...
```

Java Built-in Synchronized Statements

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead

```
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 docs.oracle.com/javase/7/docs/api/java/util/concurrent/Exchanger.html

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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];
```

Synchronized statements enable more fine-grained serialization than synchronized methods

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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];
```

Create slot outside of lock to narrow the synchronization region

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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 Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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;
    }
  }
```

Or synchronize using an explicit object

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

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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];
```

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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];
```

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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);
```

Double-Checked Locking optimization is done here

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

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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);
```

Double-Checked Locking optimization is done here

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

```
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);
```

Double-Checked Locking optimization is done here

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

Double-Checked Locking optimization is done here

```
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);
```

Java Built-in Synchronized Statements

- Synchronized methods can yield excessive serialization overhead
- Statements within a method can therefore be marked with the synchronized keyword

Intrinsic Locks and Synchronization

Synchronization is built around an internal entity known as the *intrinsic lock* or *monitor lock*. (The API specification often refers to this entity simply as a "monitor.") Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object's state and establishing happens-before relationships that are essential to visibility.

Every object has an intrinsic lock associated with it. By convention, a thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them. A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquistion of the same lock.

Locks In Synchronized Methods

When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns. The lock release occurs even if the return was caused by an uncaught exception.

You might wonder what happens when a static synchronized method is invoked, since a static method is associated with a class, not an object. In this case, the thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class.

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);
}
```

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

Partial Solution Using Java Synchronized Methods

Partial Solution Using Java Synchronized Methods

See en.wikipedia.org/wiki/Crazy_Horse_Memorial

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
        mList.add(msg);
    }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

See github.com/douglascraigschmidt/LiveLessons/tree/master/BusySynchronizedQueue

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
         mList.add(msg);
    }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

Only one synchronized method can be active in any given object

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access
  - Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
         mList.add(msg);
   }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
 }
 ...
```

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

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access
  - Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
     if (mList.size() < capacity)
        mList.add(msg);
   }

  public E synchronized take() {
     return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

Invocations of put() & take() on the same object can't interleave

Partial Solution Using Java Synchronized Methods

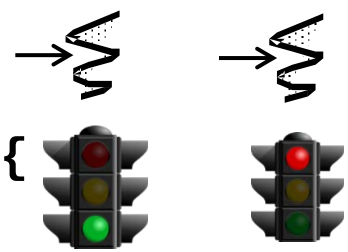
- Java synchronized methods protects critical sections from concurrent access
  - Adding the synchronized keyword has two effects

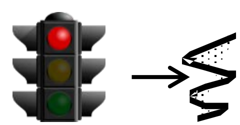
```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
         mList.add(msg);
    }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```





Each synchronized method is atomic

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access
  - Adding the synchronized keyword has two effects

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
        mList.add(msg);
   }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

Establishes a "happens-before" relation to ensure visibility of state changes to all threads

See en.wikipedia.org/wiki/Happened-before

Limitations of Java Synchronized Methods

Limitations with Java Synchronized Methods

- Although Java synchronized methods protects critical sections from concurrent access there are limitations with using them alone

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
         mList.add(msg);
   }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
 }
 ...
```

Limitations with Java Synchronized Methods

- Although Java synchronized methods protects critical sections from concurrent access there are limitations with using them alone

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
        mList.add(msg);
    }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

Concurrent calls to these methods will “busy wait”..

See en.wikipedia.org/wiki/Busy_waiting

Limitations with Java Synchronized Methods

- Although Java synchronized methods protects critical sections from concurrent access there are limitations with using them alone

```
class BusySynchronizedQueue<E> {
  private final List<E> mList;

  BusySynchronizedQueue(int capacity) {
    mCapacity = capacity;
    mList = new ArrayList<E>();
  }

  public void synchronized put(E msg) {
      if (mList.size() < capacity)
        mList.add(msg);
   }

  public E synchronized take() {
      return mList.isEmpty() ? null mList.remove(0);
  }
  ...
```

Need to coordinate put() & take() so they won't busy wait when there's nothing to do

Java built-in monitor object "wait" & "notify" mechanisms provide a convenient solution