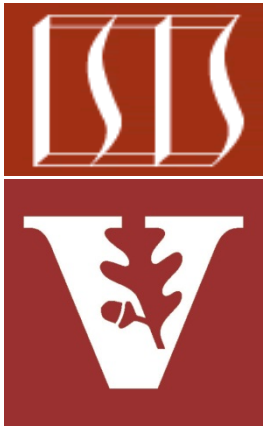
Android Concurrency : Java Built-in Monitor Objects

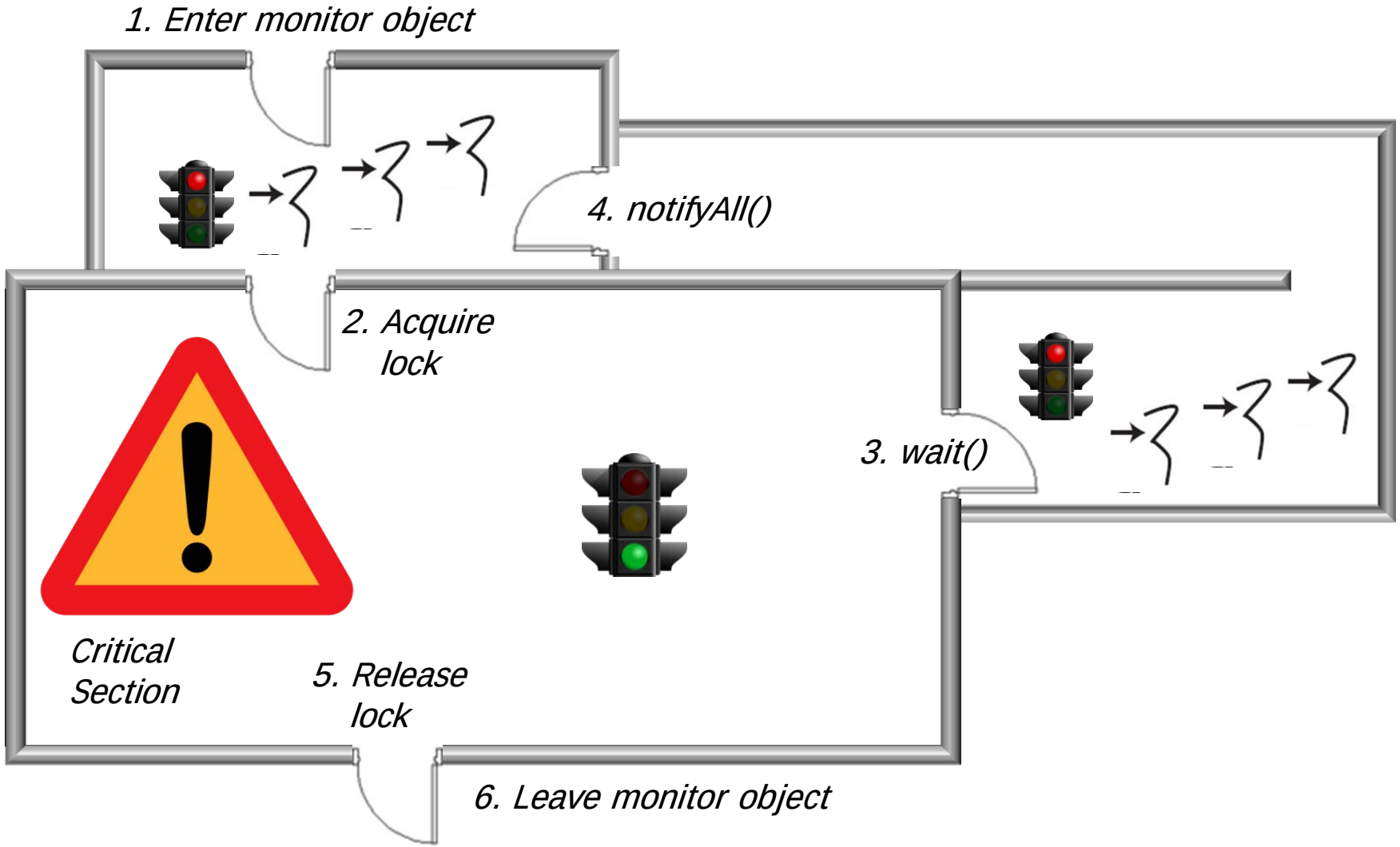


Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

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

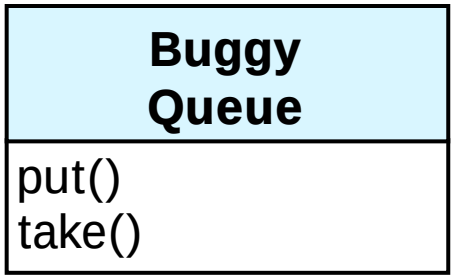
Learning Objectives in this Part of the Module

- Understand how Java built-in monitor objects can synchronize & schedule the interactions of threads running in a concurrent program



Motivating Java Built-in Monitor Objects

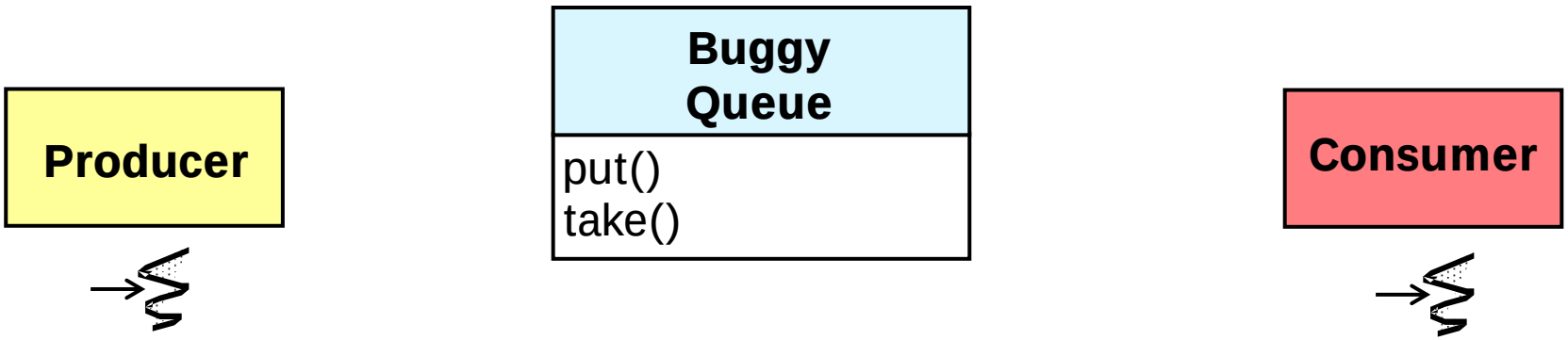
- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions



See earlier part on “Motivating Java Synchronization & Scheduling Mechanisms”

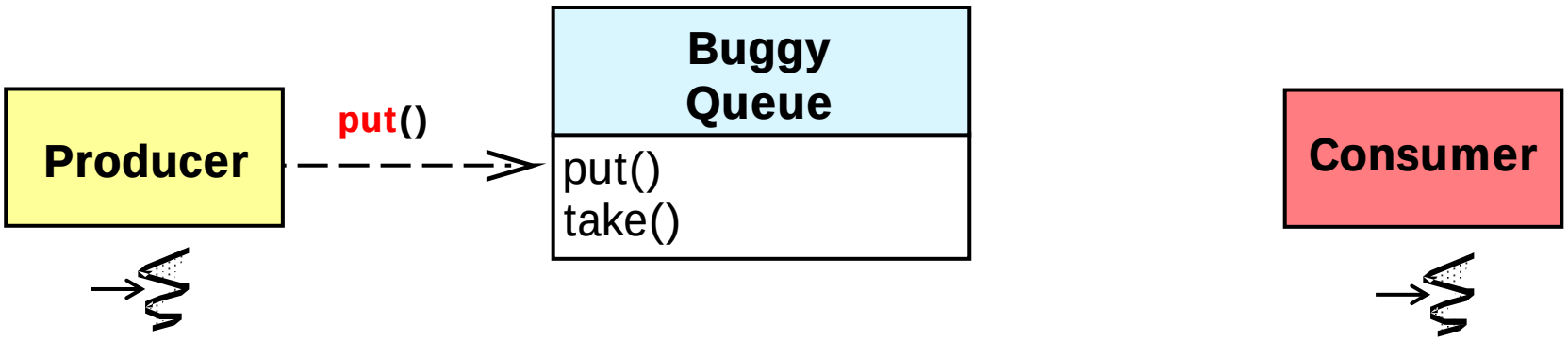
Motivating Java Built-in Monitor Objects

- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions



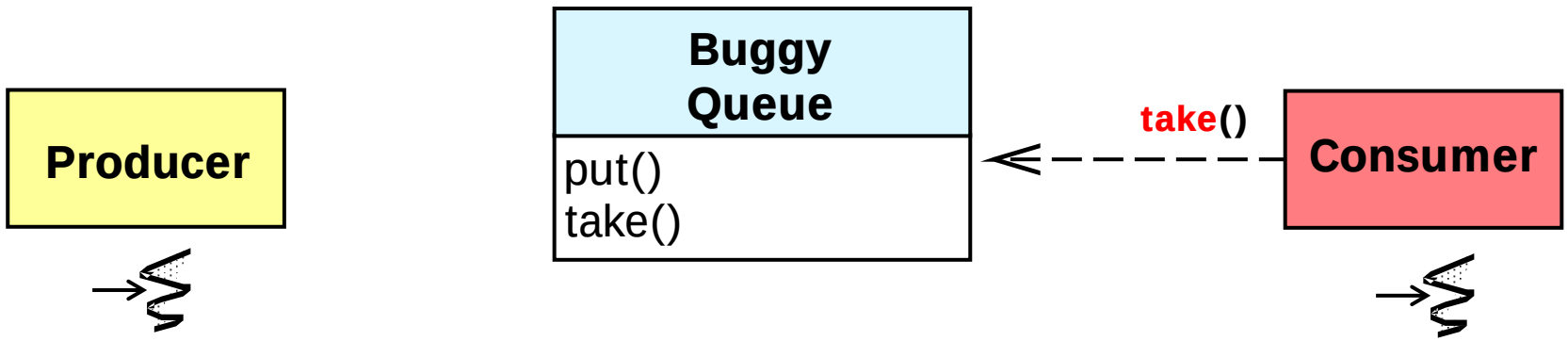
Motivating Java Built-in Monitor Objects

- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions



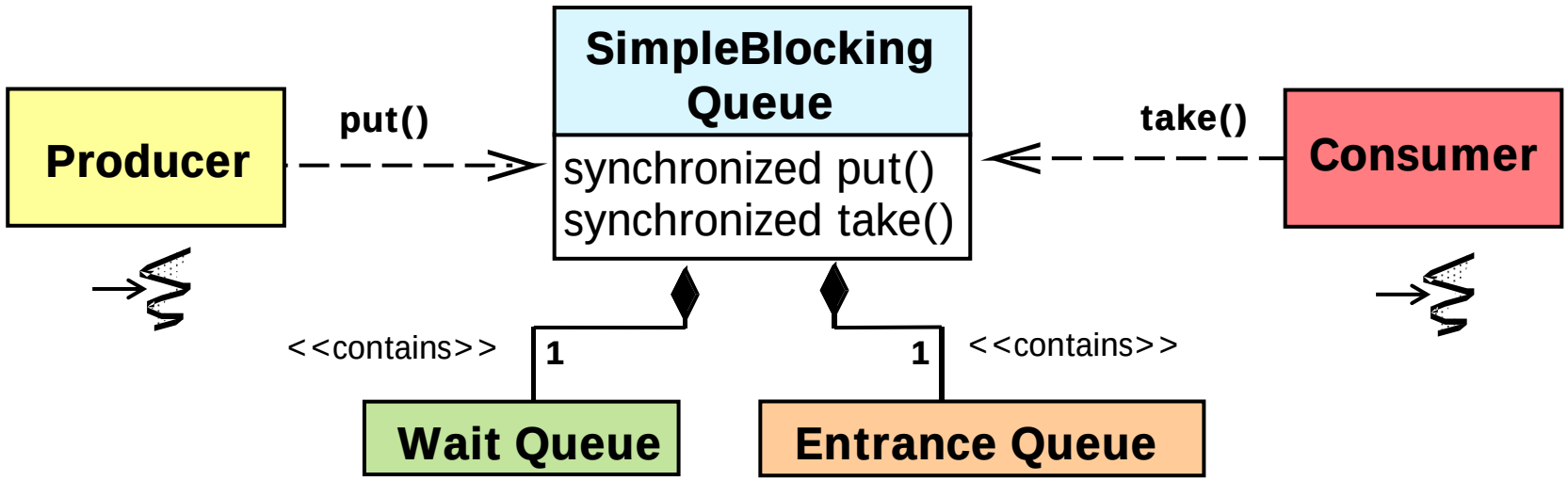
Motivating Java Built-in Monitor Objects

- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions



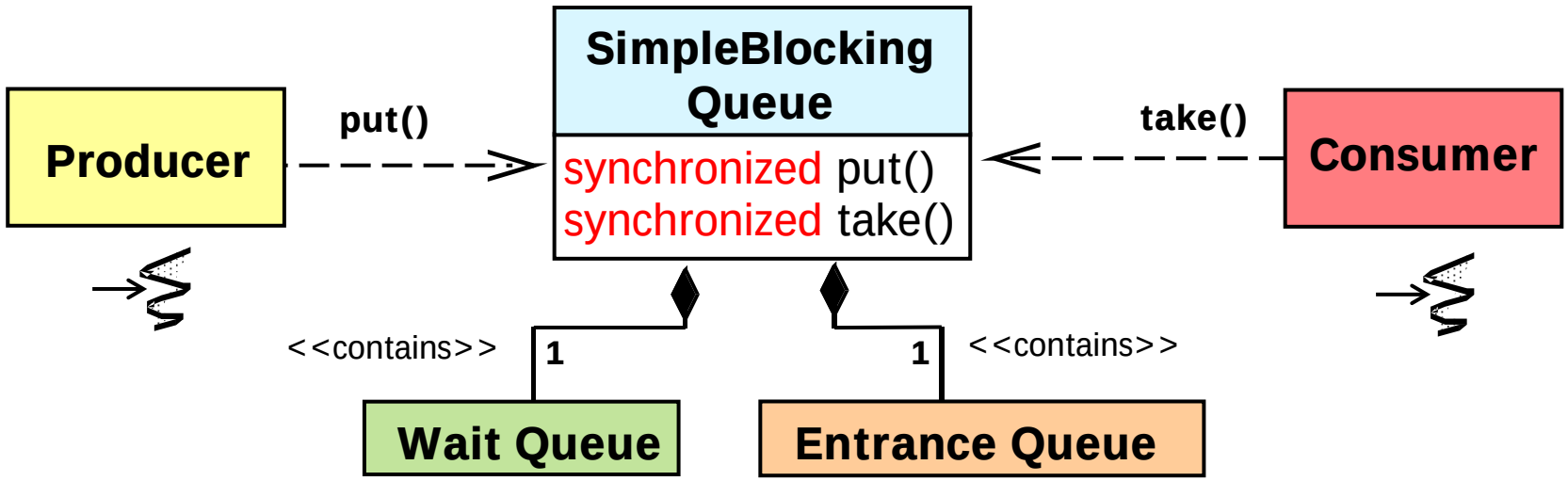
Motivating Java Built-in Monitor Objects

- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions
  - This solution uses a Java monitor object to synchronize the queue



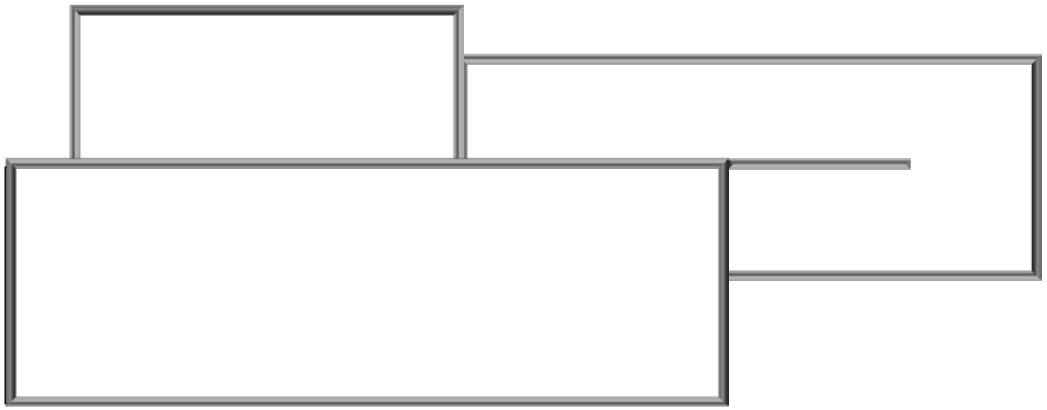
Motivating Java Built-in Monitor Objects

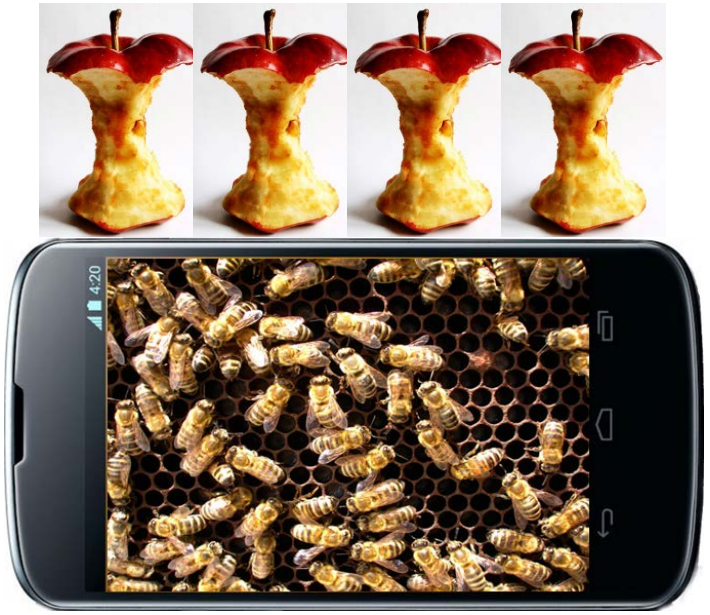
- Consider a concurrent producer/consumer portion of a Java program
  - Our initial solution incurred race conditions
  - This solution uses a Java monitor object to synchronize the queue



Overview of Monitors

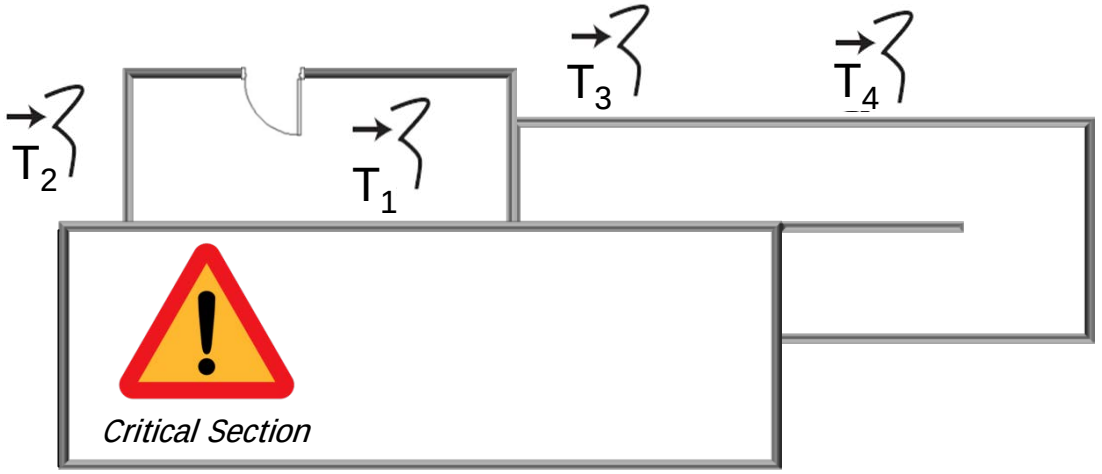
- A monitor is a concurrency control construct used for synchronization & scheduling

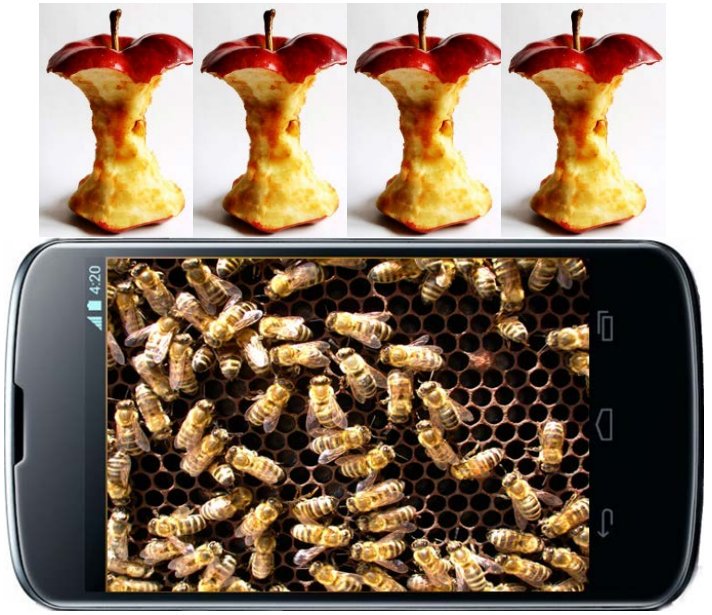




Overview of Monitors

- A monitor is a concurrency control construct used for synchronization & scheduling

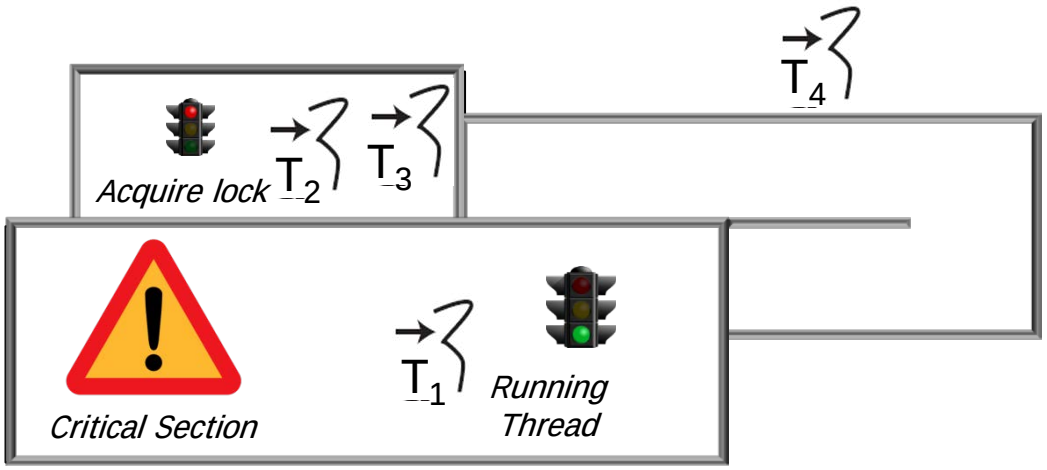


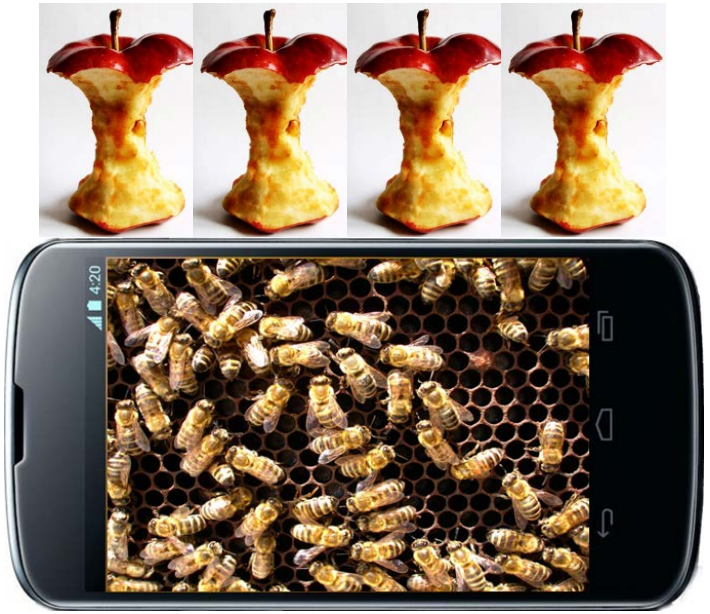


en.wikipedia.org/wiki/Monitor_(synchronization) has more on Monitors

Overview of Monitors

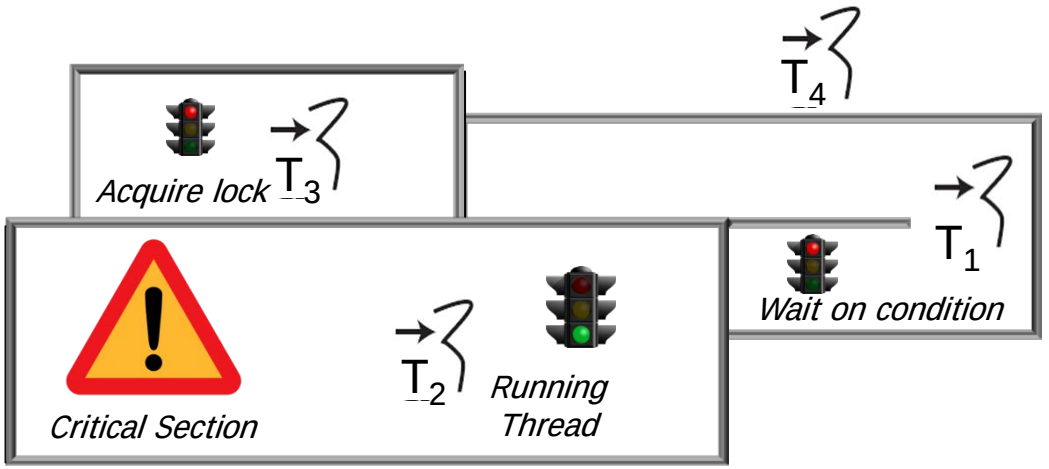
- A monitor is a concurrency control construct used for synchronization & scheduling
  - It allows threads to have mutual exclusion

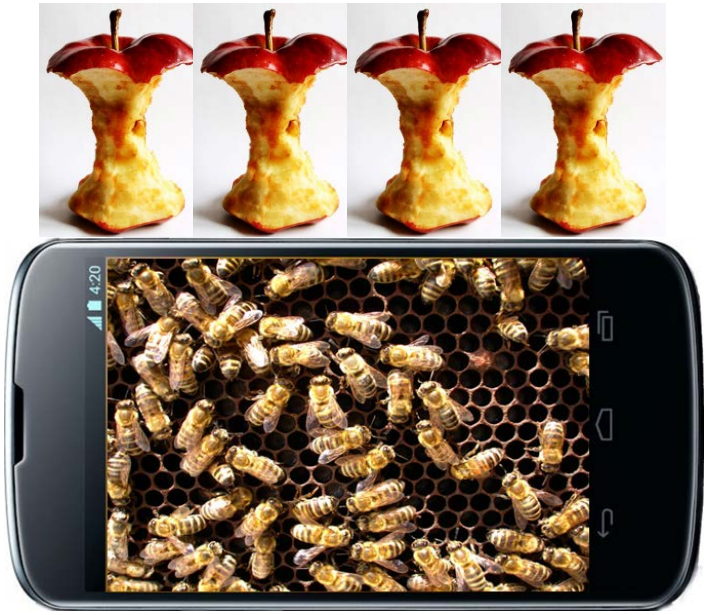




Overview of Monitors

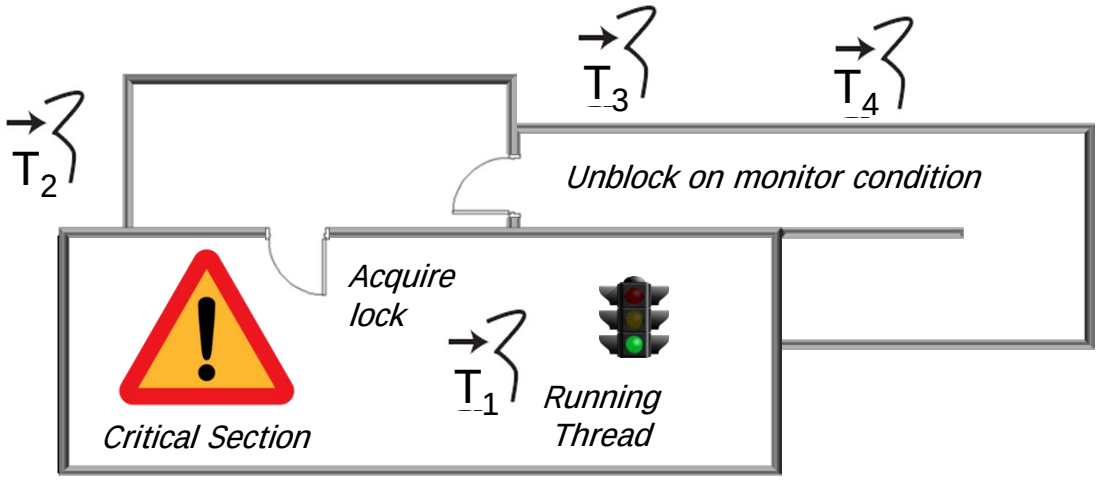
- A monitor is a concurrency control construct used for synchronization & scheduling
  - It allows threads to have mutual exclusion
  - & the ability to wait (block) for certain conditions to become true

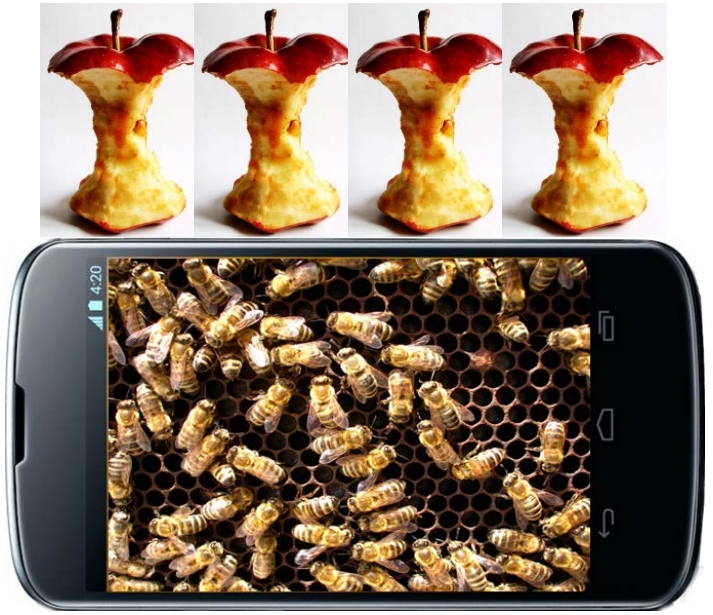




Overview of Monitors

- A monitor is a concurrency control construct used for synchronization & scheduling
  - It allows threads to have mutual exclusion
  - & the ability to wait (block) for certain conditions to become true
  - It also allows notifying other threads that their conditions have been met





Partial Solution Using Java Synchronized Methods

Partial Solution Using Java Synchronized Methods

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

Partial Solution Using Java Synchronized Methods

- Java synchronized methods protects critical sections from concurrent access

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.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

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

Partial Solution Using Java Synchronized Methods

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

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.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

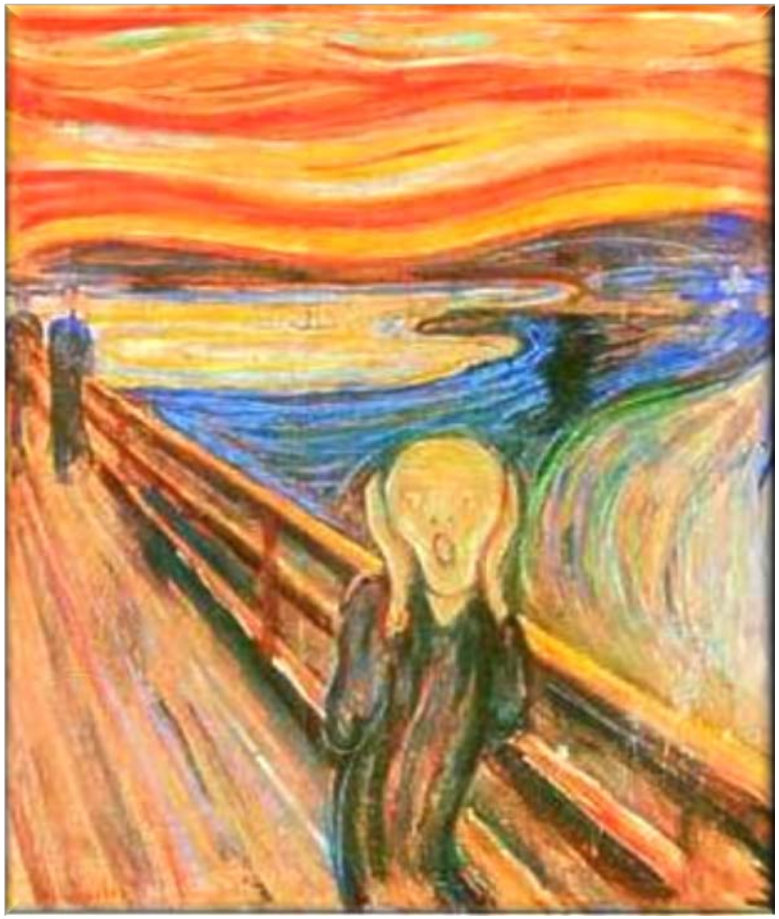
```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

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

Partial Solution Using Java Synchronized Methods

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

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```



There are still problems with this solution...

Partial Solution Using Java Synchronized Methods

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

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

Concurrent calls to this method will “busy wait” or worse..

Partial Solution Using Java Synchronized Methods

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

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

Concurrent calls to this method can exhaust heap memory

Partial Solution Using Java Synchronized Methods

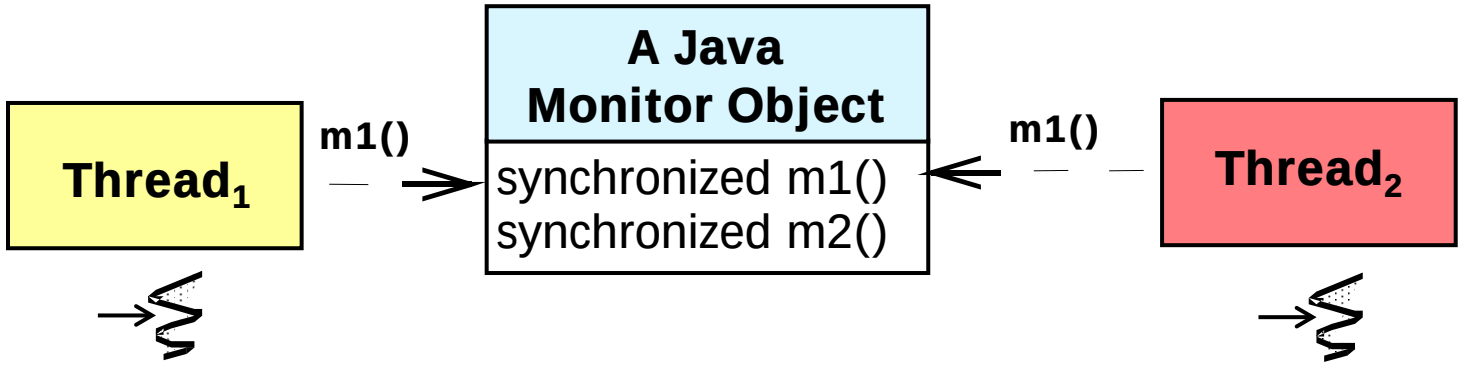
- Java synchronized methods protects critical sections from concurrent access
  - Adding the synchronized keyword has two effects

```
static class BuggySynchronizedQueue
{
  private List<String> mQ =
    new ArrayList<String>();

  public void synchronized
    put(String msg){ mQ.add(msg); }

  public String synchronized
    take(){ return mQ.remove(0); }
}
```

en.wikipedia.org/wiki/Busy_waiting has more info

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 & scheduling

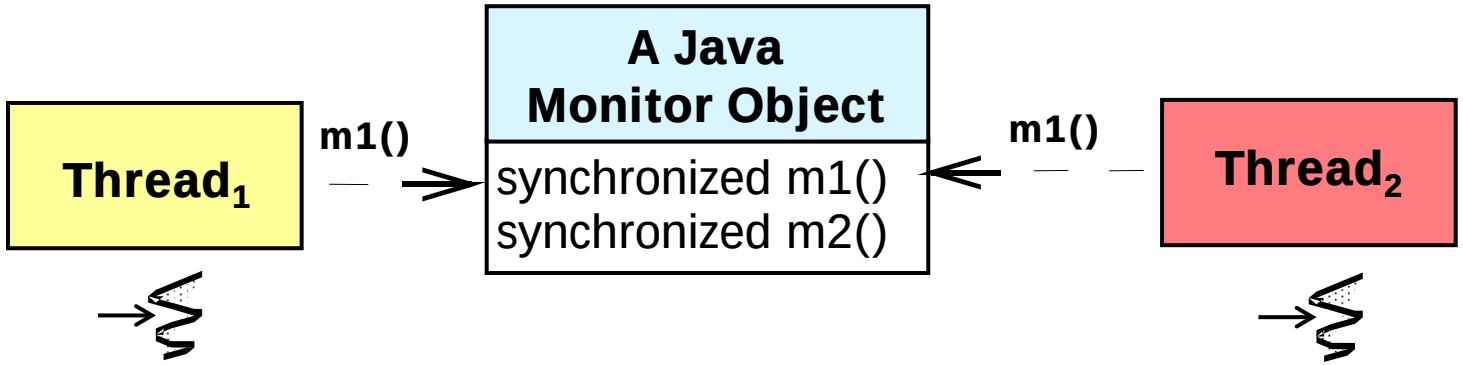


www.artima.com/insidejvm/ed2/threadsynch.html has more info

Overview of Built-in Java Monitor Objects

Overview of Java Built-in Monitor Objects

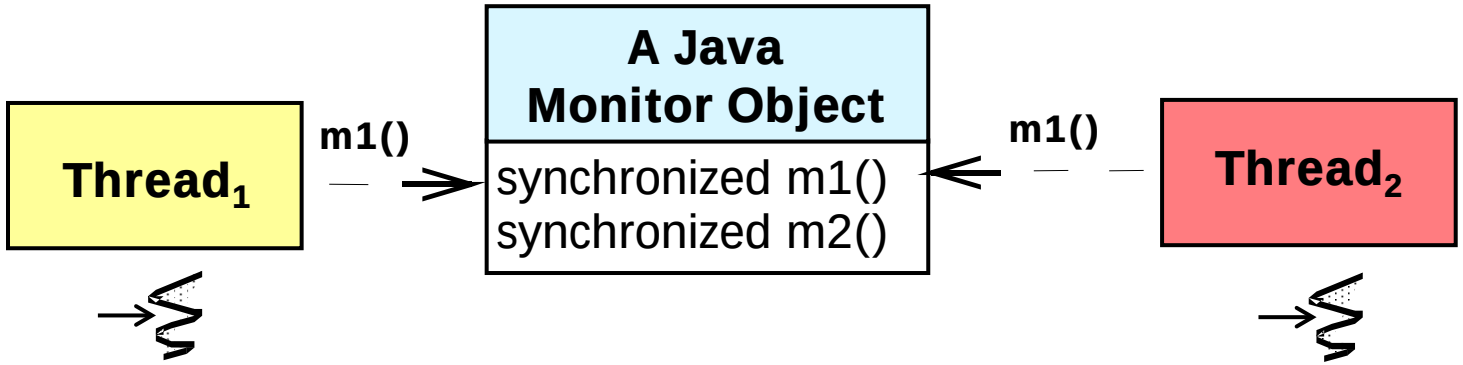
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization & scheduling



www.artima.com/insidejvm/ed2/threadsynch.html has more info

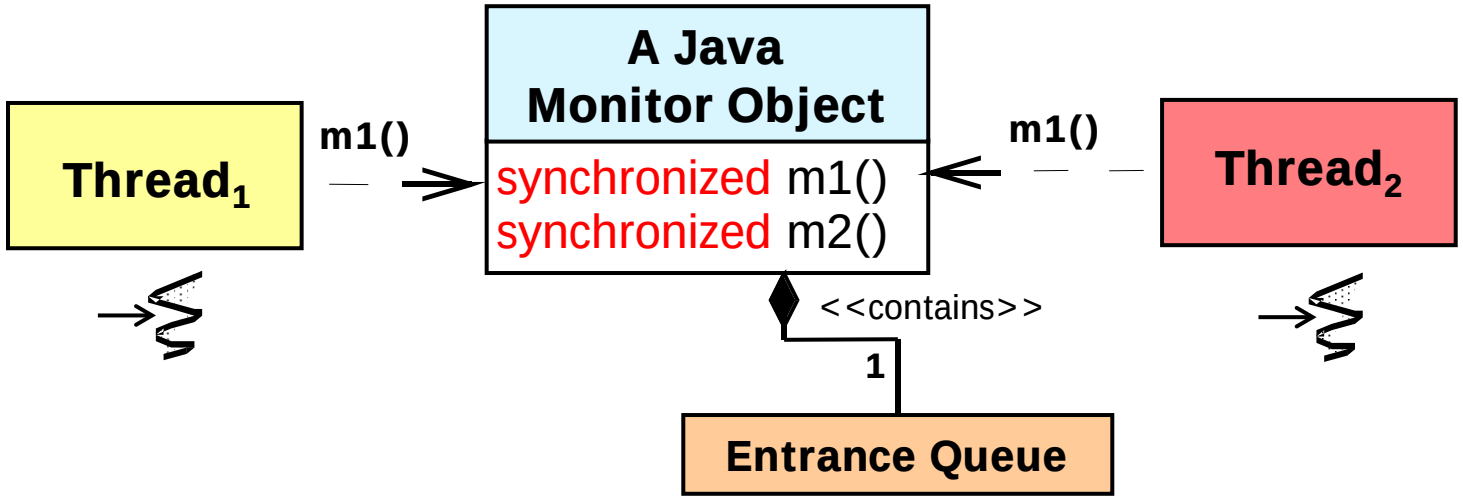
Overview of Java Built-in Monitor Objects

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



Overview of Java Built-in Monitor Objects

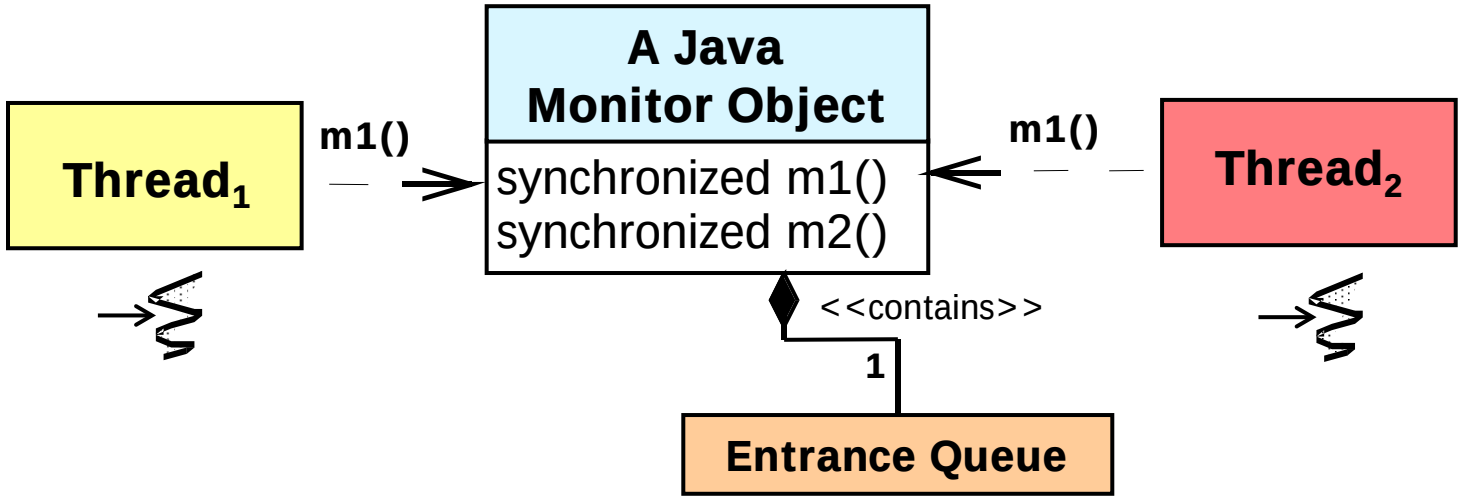
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization & scheduling
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions



JVM supports mutual exclusion by an entrance queue & synchronized methods

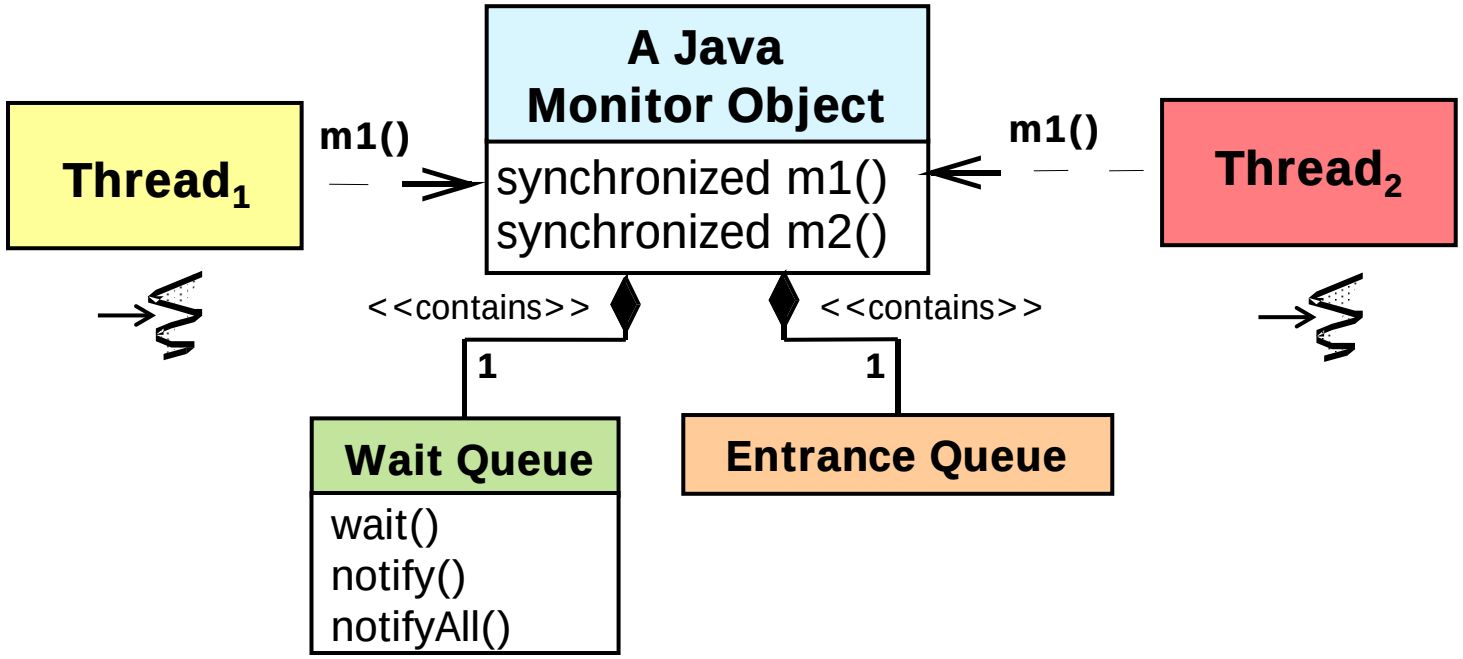
Overview of Java Built-in Monitor Objects

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



Overview of Java Built-in Monitor Objects

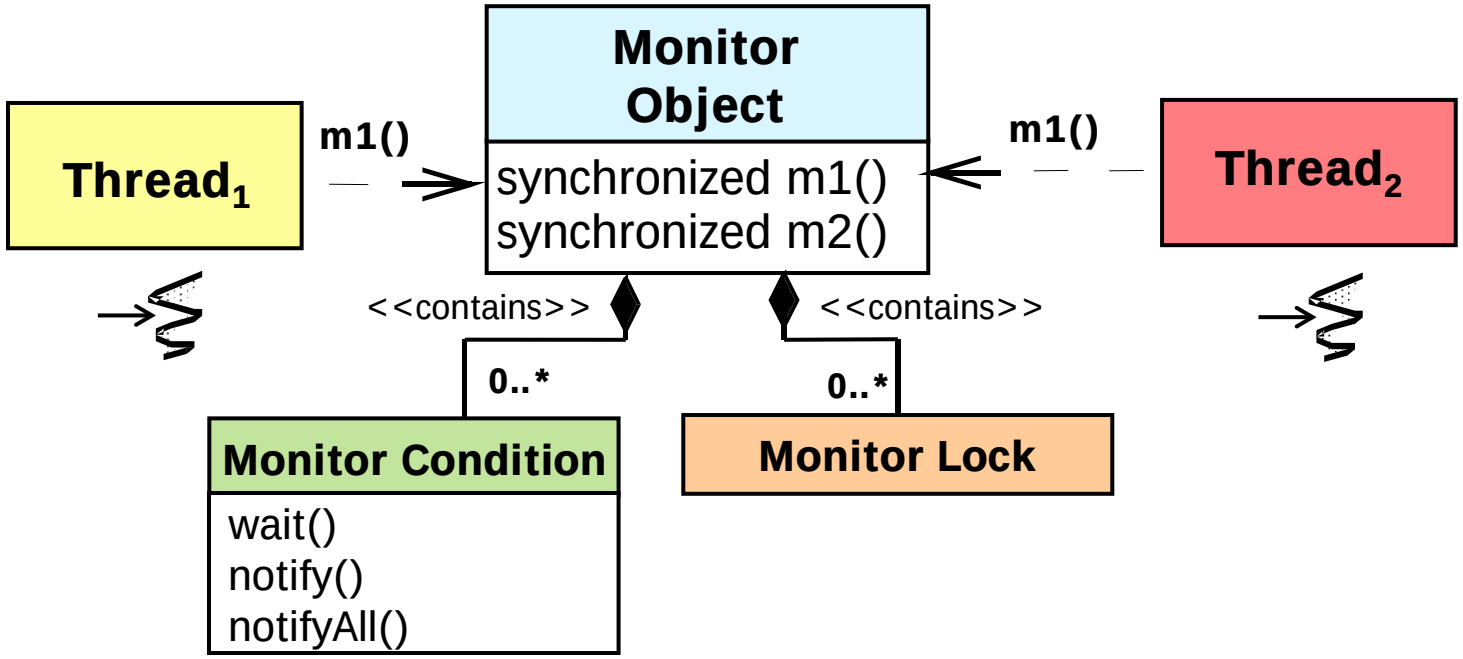
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization & scheduling
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions
  - **Cooperation** – enables threads to coordinate & schedule their interactions



JVM supports cooperation by a wait queue & notification mechanisms

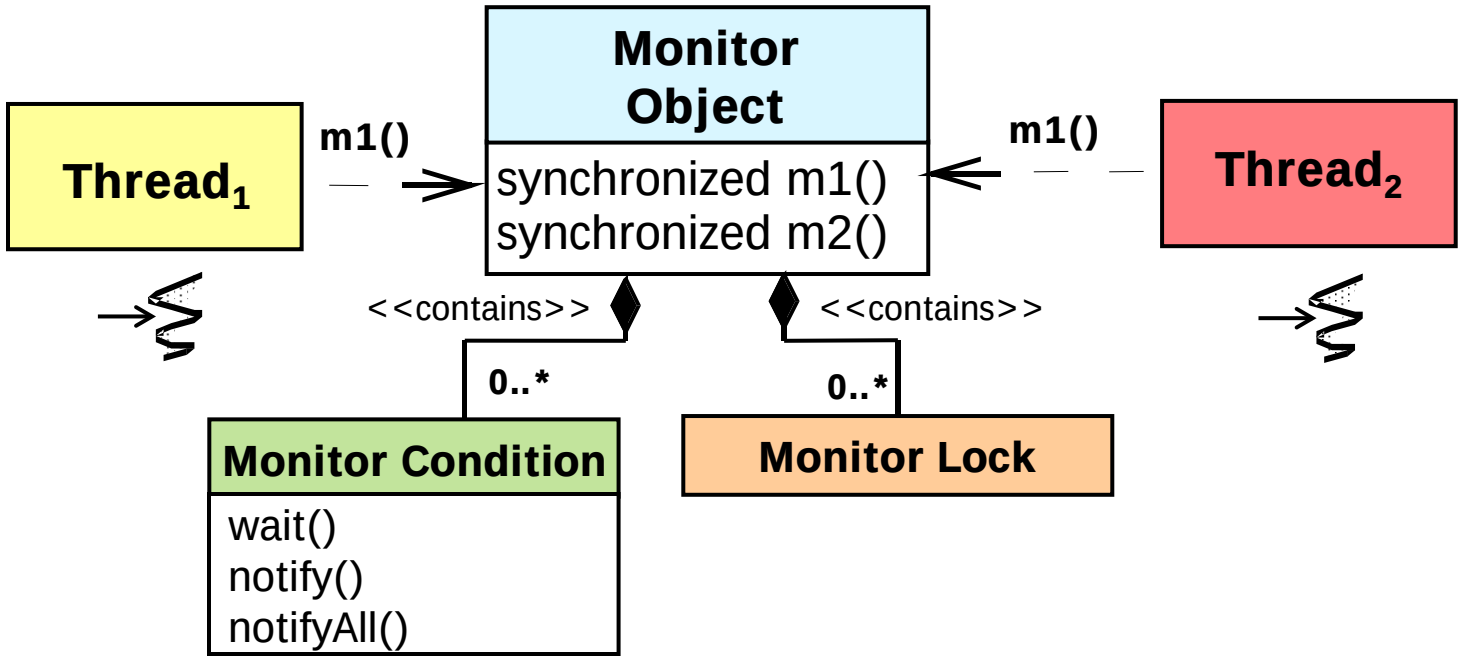
Overview of Java Built-in Monitor Objects

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



Overview of Java Built-in Monitor Objects

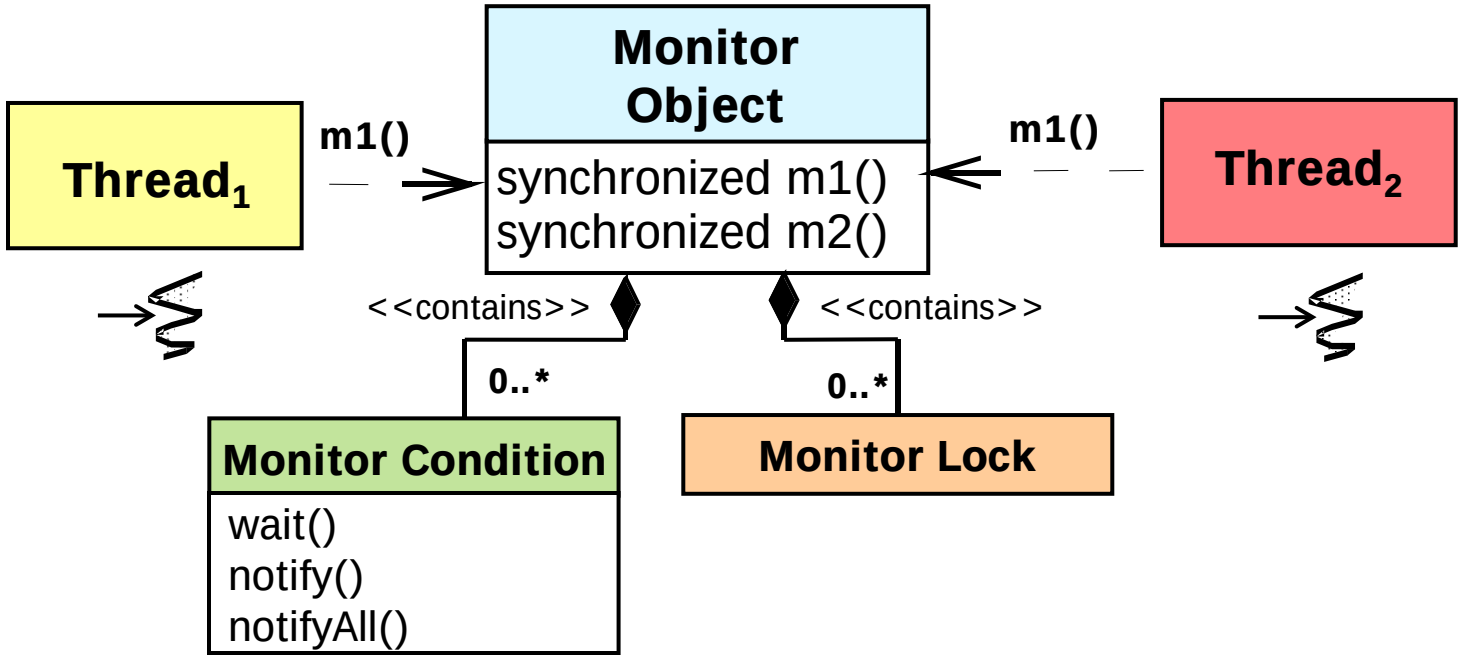
- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization & scheduling
- These mechanisms implement a variant of the *Monitor Object* pattern



See upcoming parts on “The Monitor Object Pattern”

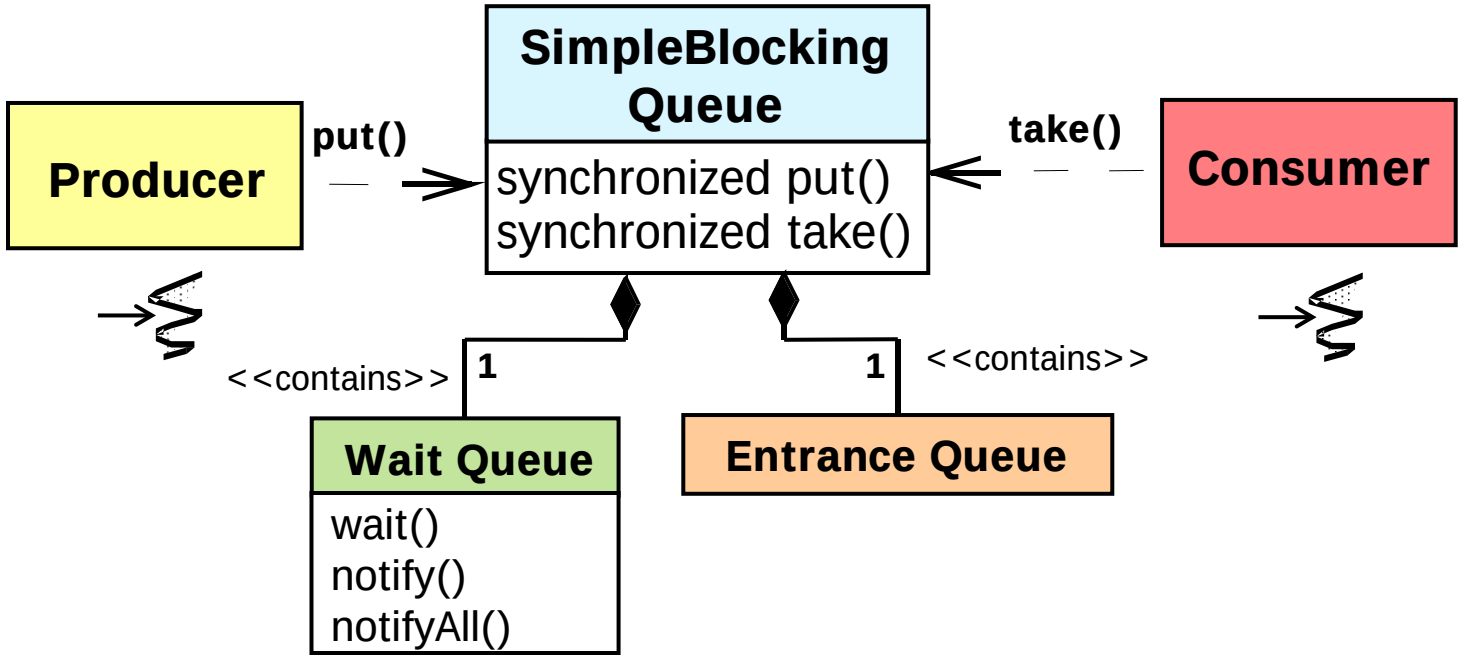
Overview of Java Built-in Monitor Objects

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



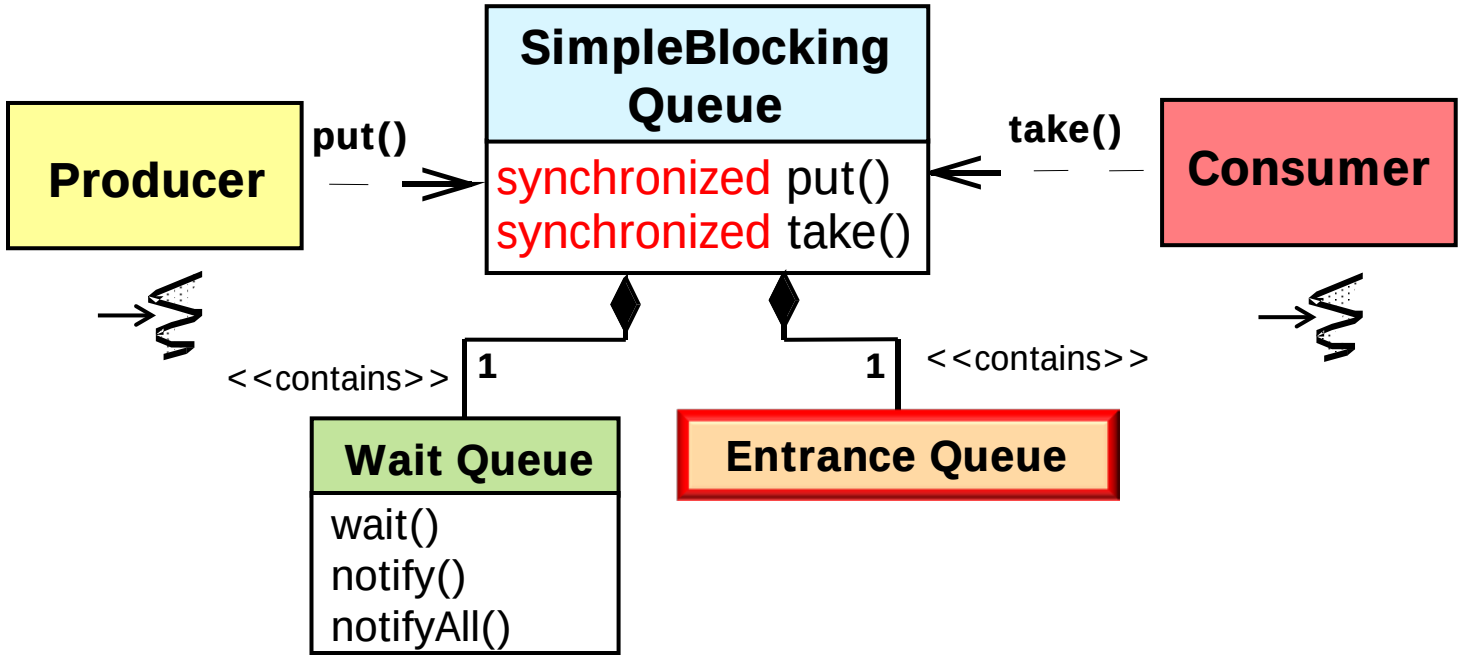
Overview of Java Built-in Monitor Objects

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



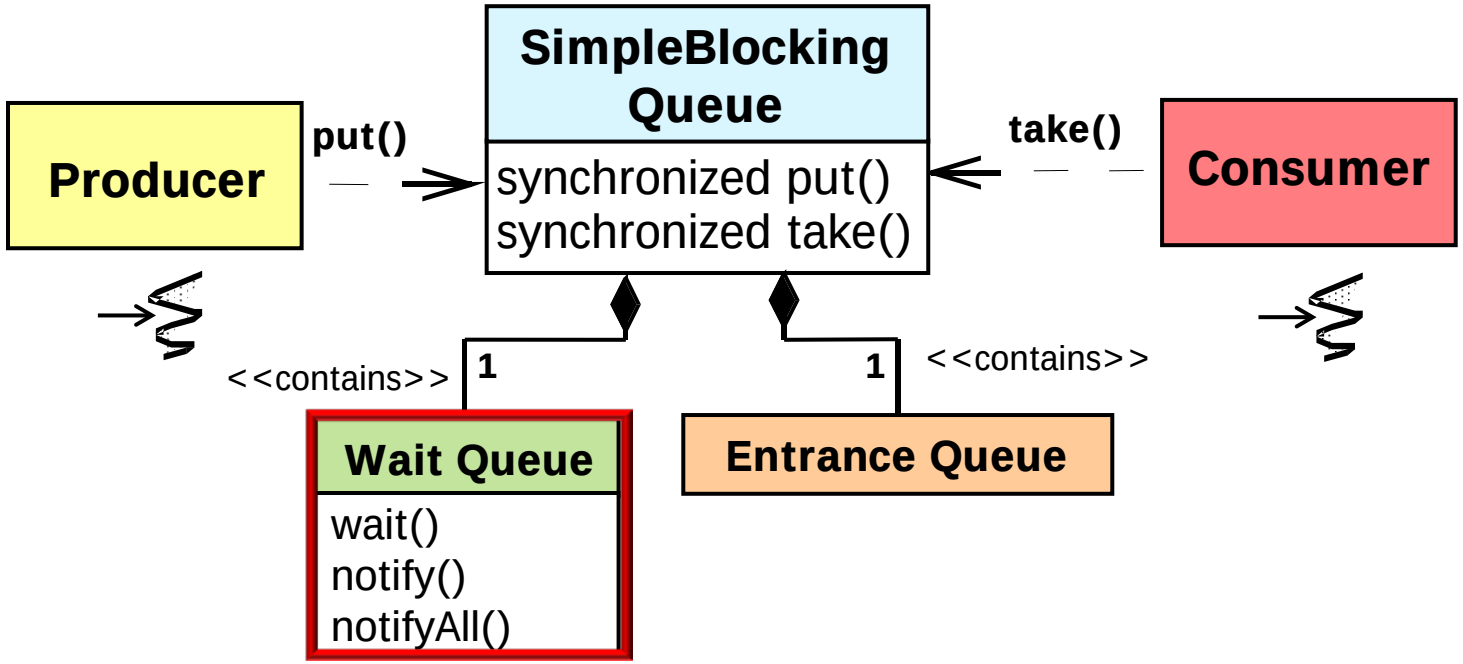
Overview of Java Built-in Monitor Objects

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



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 & scheduling
- These mechanisms implement a variant of the *Monitor Object* pattern
- Java built-in monitor objects can implement a better solution to the earlier BuggyQueue & BuggySynchronizedQueue solutions



Java Built-in Synchronized Methods & Statements

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized void put
                          (String msg){
    ...
    mQ.add(msg);
    notifyAll();
    ...
```

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized void put
                          (String msg){
    ...
    mQ.add(msg);
    notifyAll();
    ...

  public synchronized String take(){
    ...
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
    ...
```

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

Java Built-in Synchronized Methods & Statements

- 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


```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized void put
                          (String msg){
    ...
    mQ.add(msg);
    notifyAll();
    ...

  public synchronized String take(){
    ...
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
    ...
```

See earlier part on “Java ReentrantLock”

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword
- Java also supports synchronized statements

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public void put(String msg){
    ...
    synchronized (this) {
      mQ.add(msg);
      notifyAll();
    }
    ...
```

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword
- Java also supports synchronized statements

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public void put(String msg){
    ...
    synchronized(this) {
      mQ.add(msg);
      notifyAll();
    }
    ...
```

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword
- Java also supports synchronized statements
  - Synchronized statements can enable more fine-grained thread serialization than synchronized methods

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized void put
                        (String msg){
    ...
    mQ.add(msg);
    notifyAll();
    ...
```

Entire method is serialized

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword
- Java also supports synchronized statements
  - Synchronized statements can enable more fine-grained thread serialization than synchronized methods

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public void put(String msg){
    ...
    synchronized(this) {
      mQ.add(msg);
      notifyAll();
    }
    ...
```

Only this statement is serialized

Java Built-in Synchronized Methods & Statements

- Methods in a built-in monitor object must be marked with the synchronized keyword
- Java also supports synchronized statements
  - Synchronized statements can enable more fine-grained thread serialization than synchronized methods

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public void put(String msg){
    ...
    synchronized(mLock) {
      mQ.add(msg);
      notifyAll();
    }
    ...

  private Object mLock =
    new Object;
```

Synchronize the statement using an explicit lock object

Java Built-in Waiting & Notification Mechanisms

Java Built-in Waiting & Notification Mechanisms

Java synchronized methods & statements only provide a partial solution

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Java Built-in Waiting & Notification Mechanisms

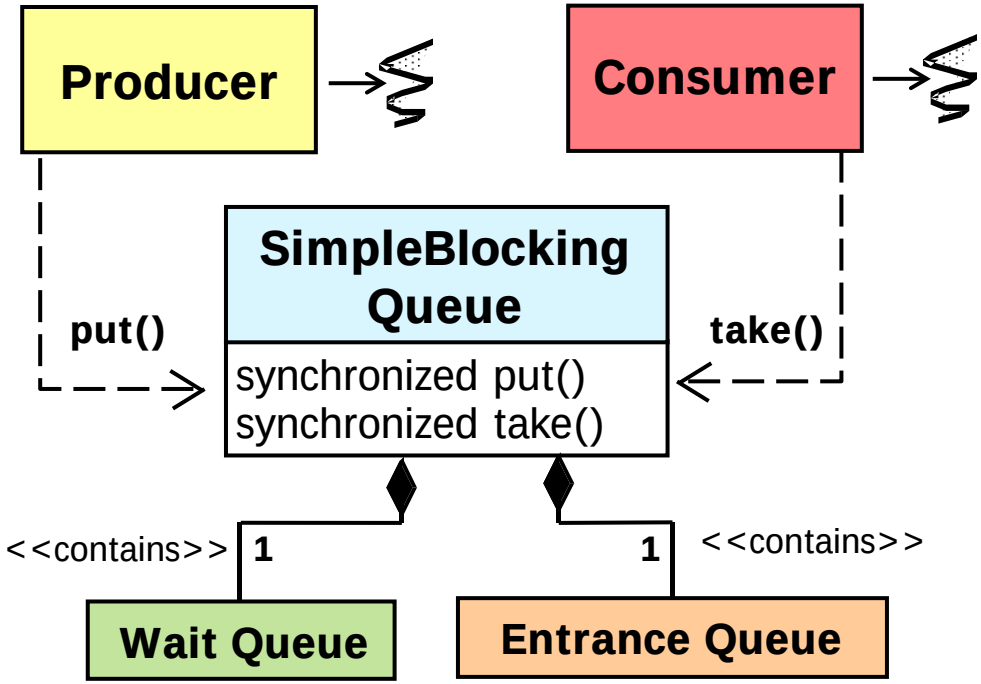
- Java monitor objects provide mechanisms that help threads interact cooperatively
  - via the wait(), notify(), & notifyAll() methods

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

See earlier part on “Java ConditionObject” for a comparison

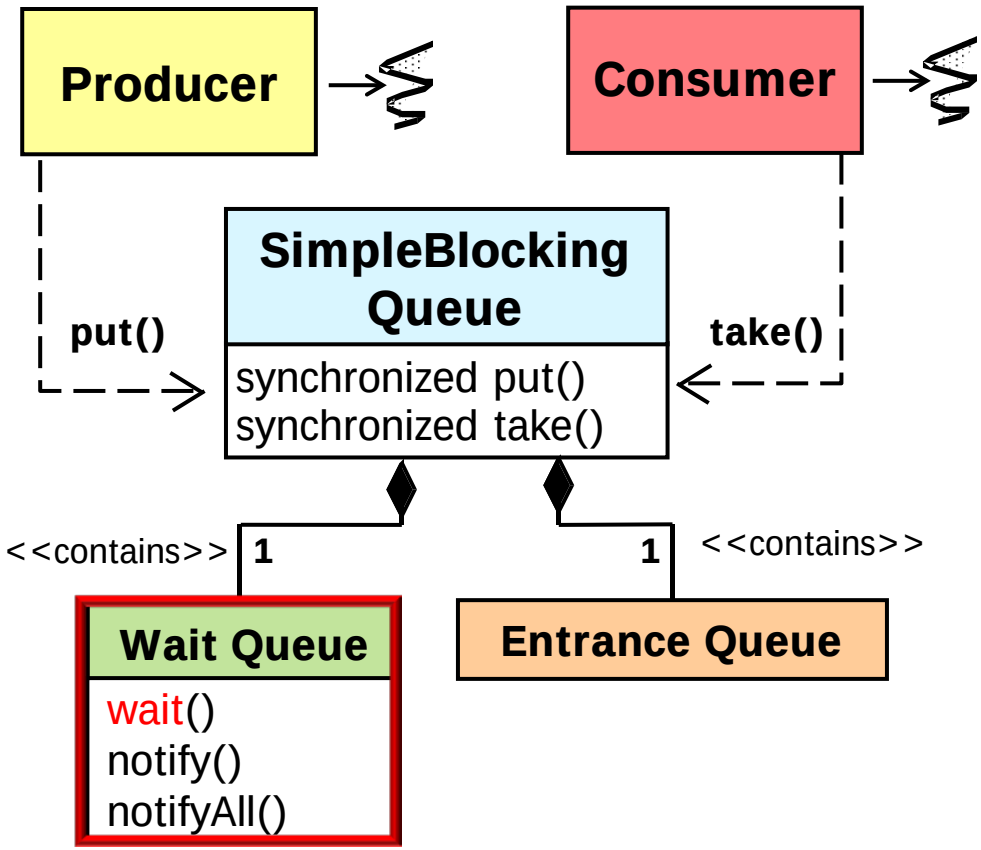
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue



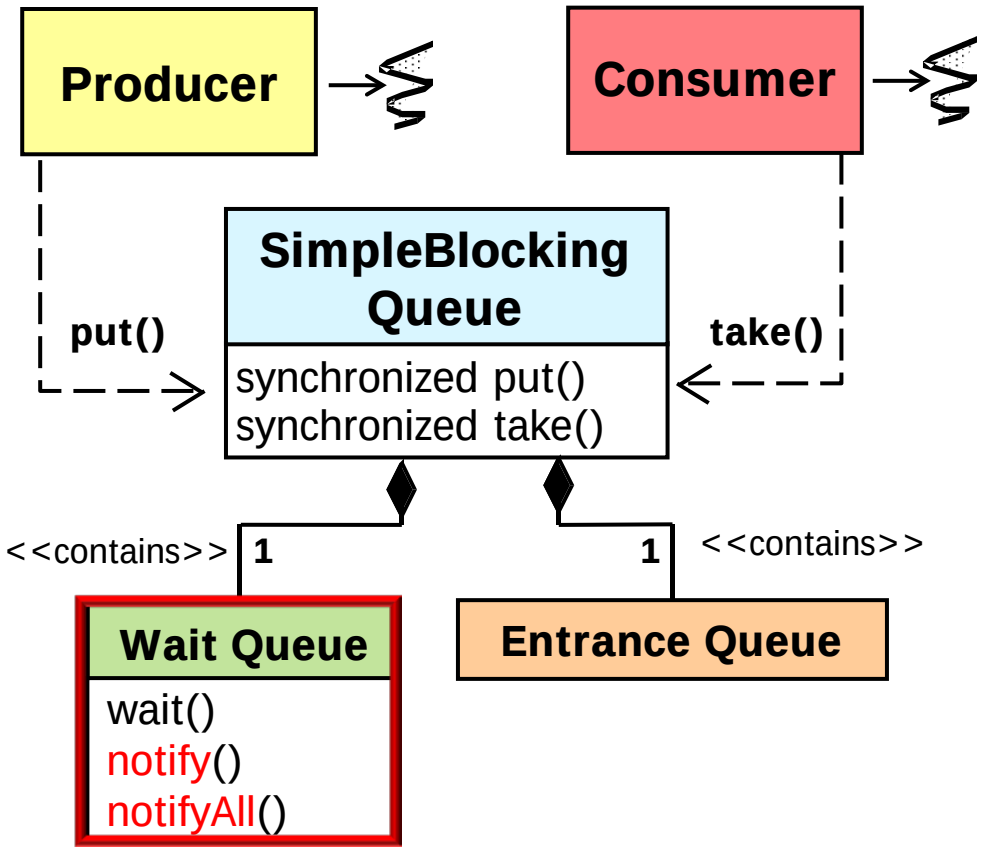
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
  - All wait() calls are done on the wait queue



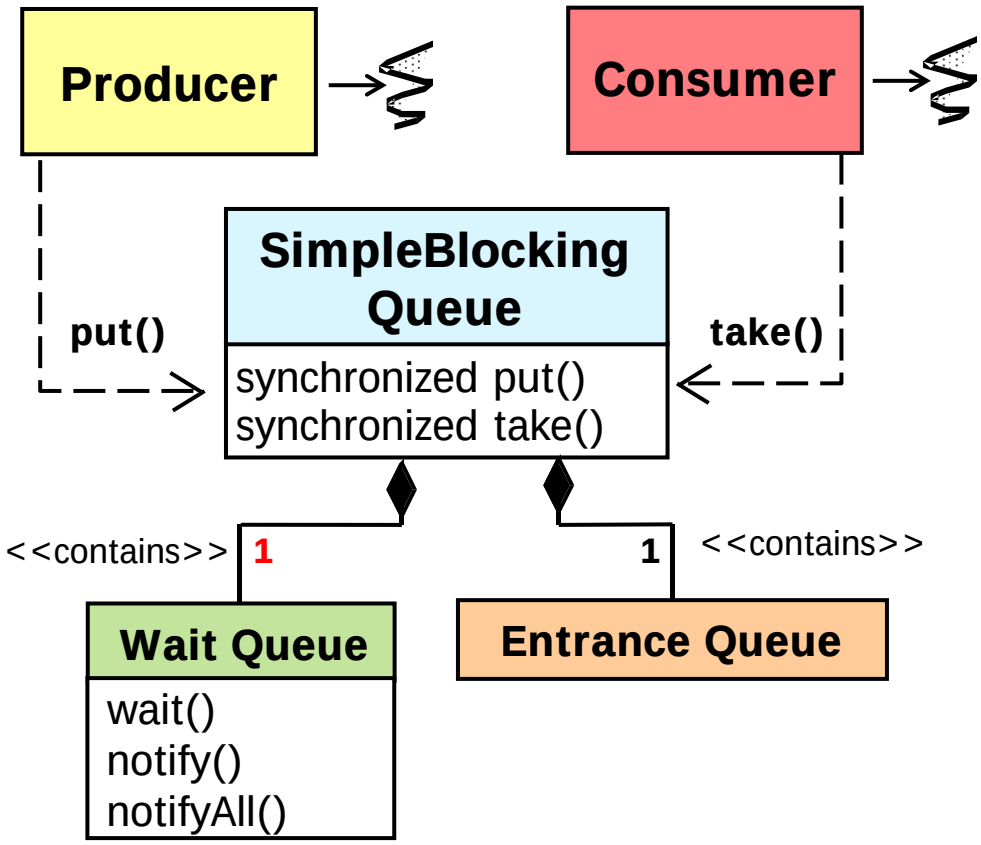
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
  - All wait() calls are done on the wait queue
  - All notify() & notifyAll() calls apply to this queue



Java Built-in Waiting & Notification Mechanisms

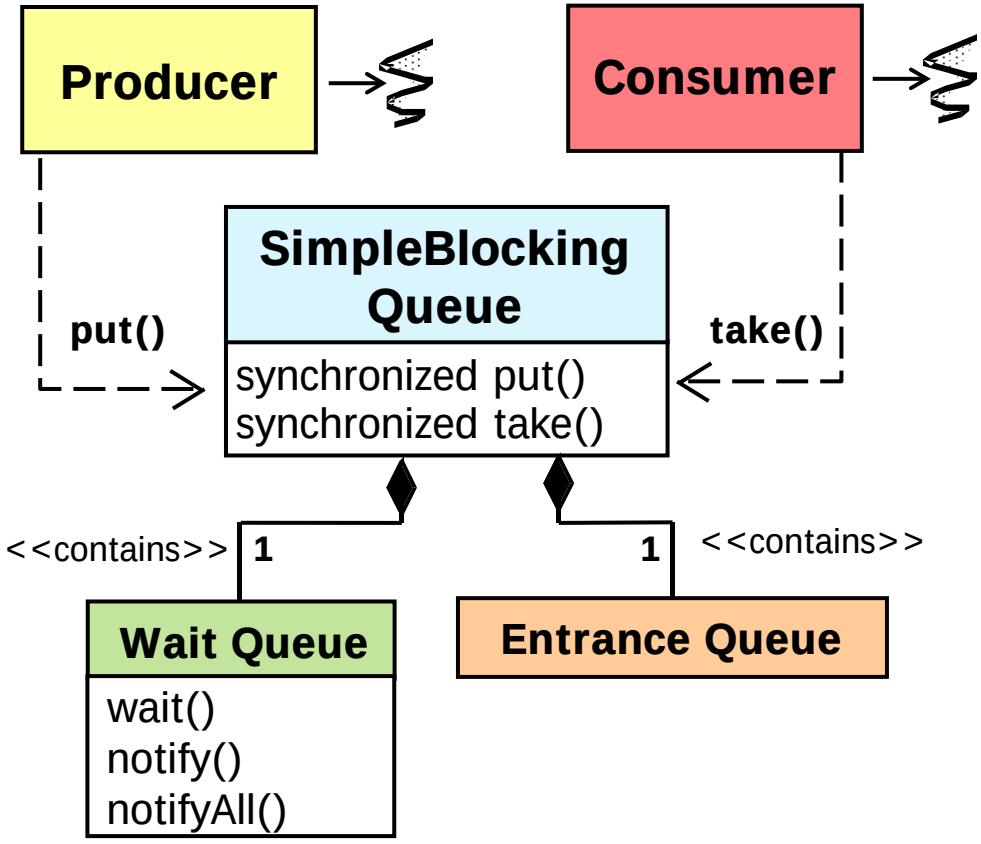
- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
  - All wait() calls are done on the wait queue
  - All notify() & notifyAll() calls apply to this queue
  - Having only one wait queue in a monitor object is more restrictive than programming with Java ConditionObjects



See earlier part on “Java ConditionObject” for a comparison

Java Built-in Waiting & Notification Mechanisms

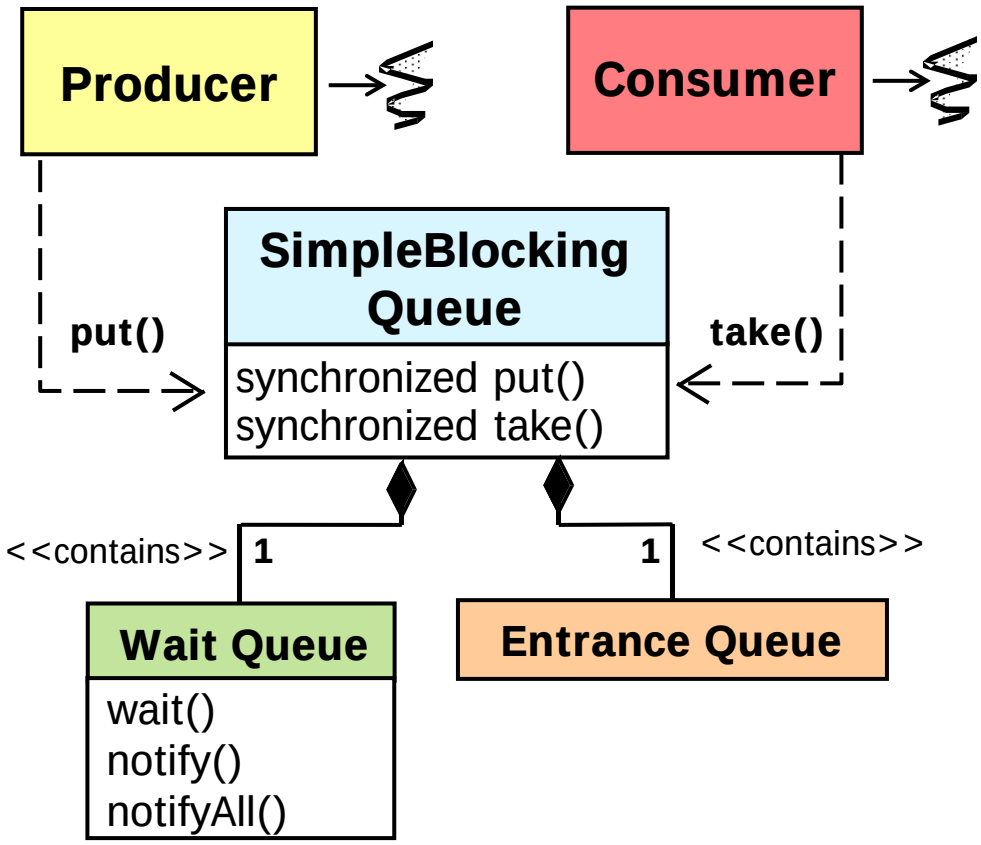
- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
  - All wait() calls are done on the wait queue
  - All notify() & notifyAll() calls apply to this queue
  - Having only one wait queue in a monitor object is more restrictive than programming with Java ConditionObjects



Consequences of this restriction are discussed in this video's summary

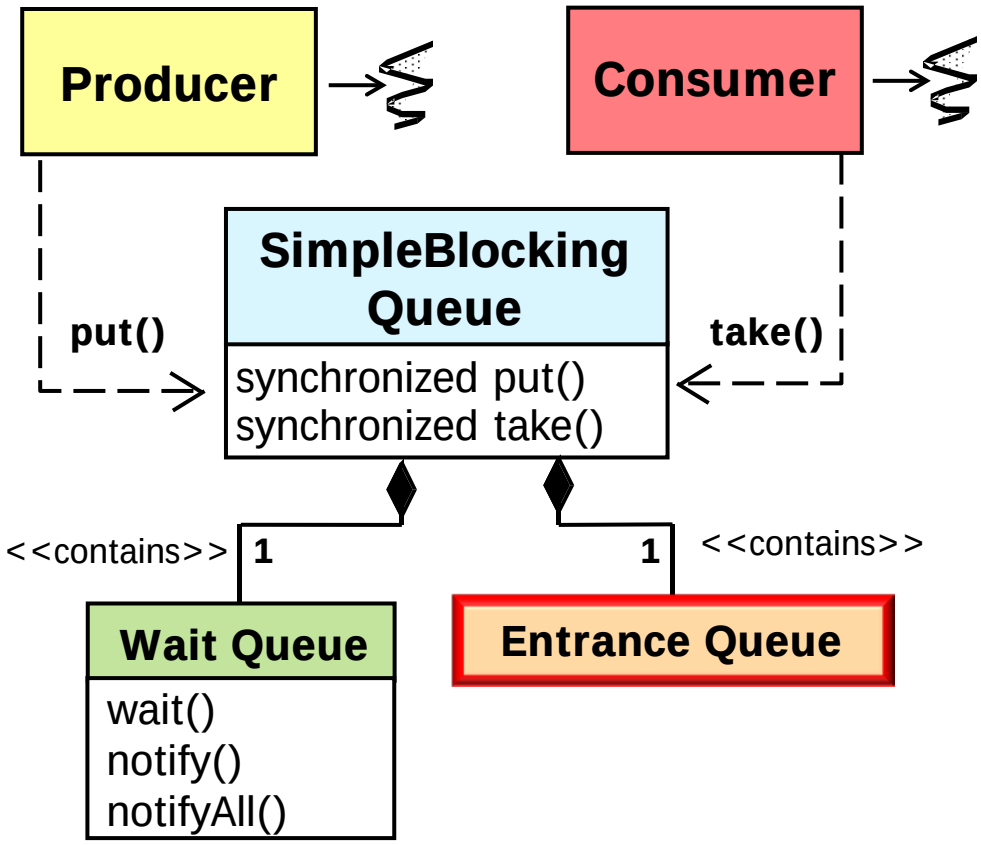
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
- Android implements built-in monitor object synchronizers via POSIX mechanisms



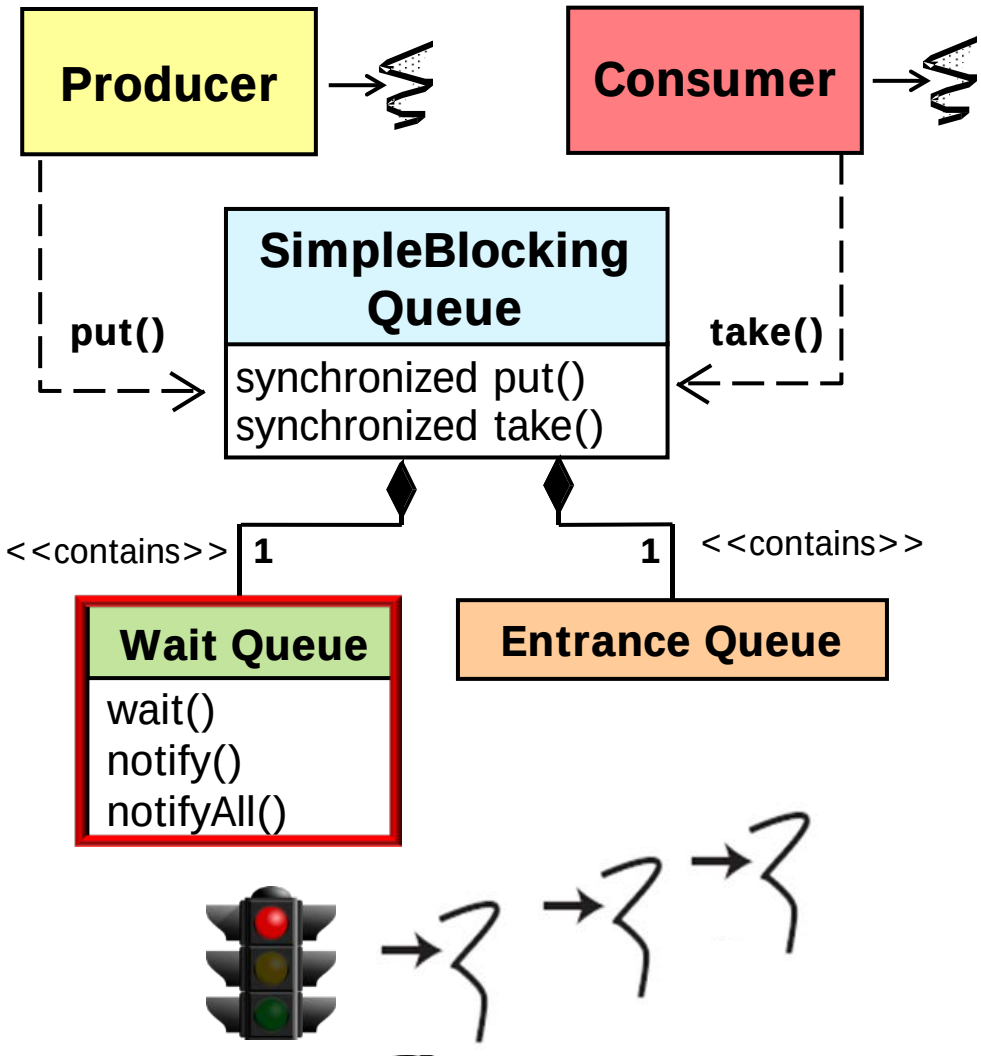
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
- Android implements built-in monitor object synchronizers via POSIX mechanisms
  - Entrance queue is a POSIX mutex with recursive locking semantics



Java Built-in Waiting & Notification Mechanisms

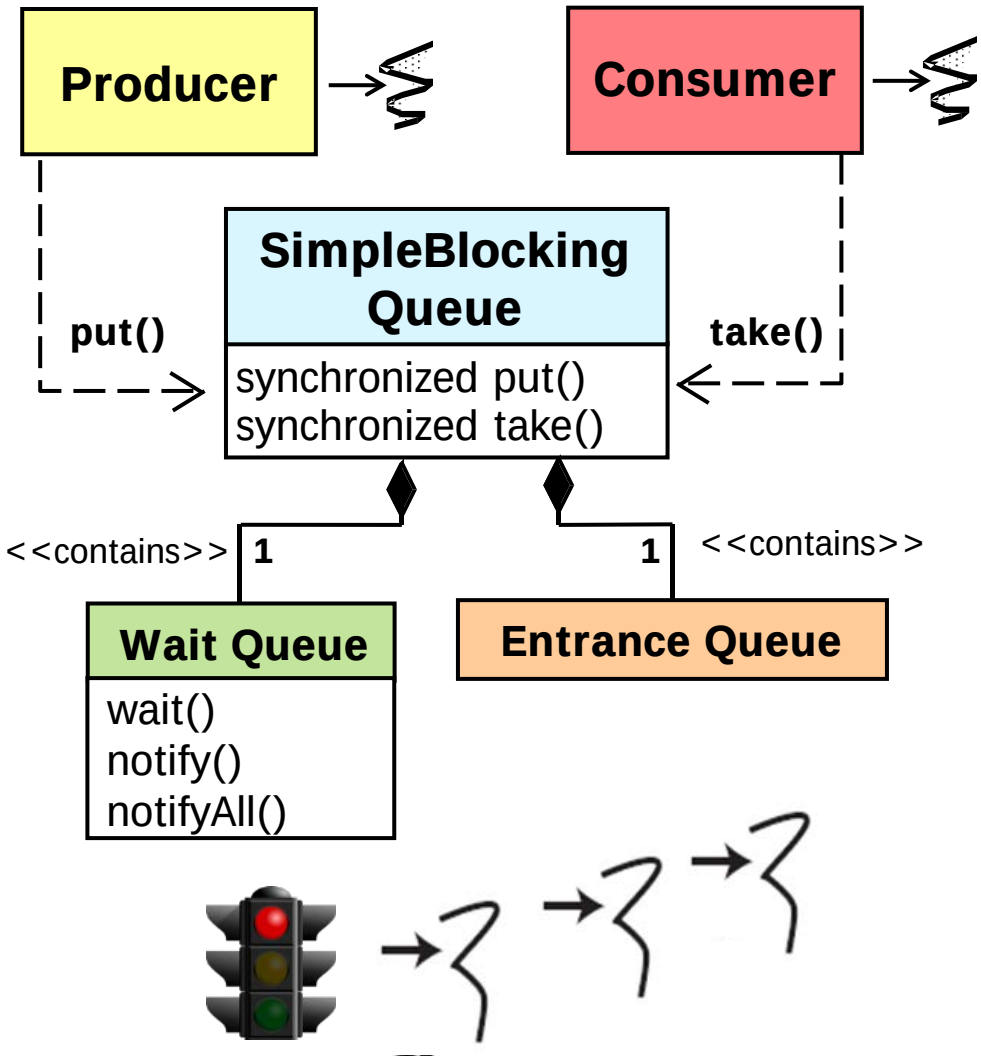
- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
- Android implements built-in monitor object synchronizers via POSIX mechanisms
  - Entrance queue is a POSIX mutex with recursive locking semantics
  - Wait queue is a POSIX condition variable



computing.llnl.gov/tutorials/pthreads/#ConditionVariables has more info

Java Built-in Waiting & Notification Mechanisms

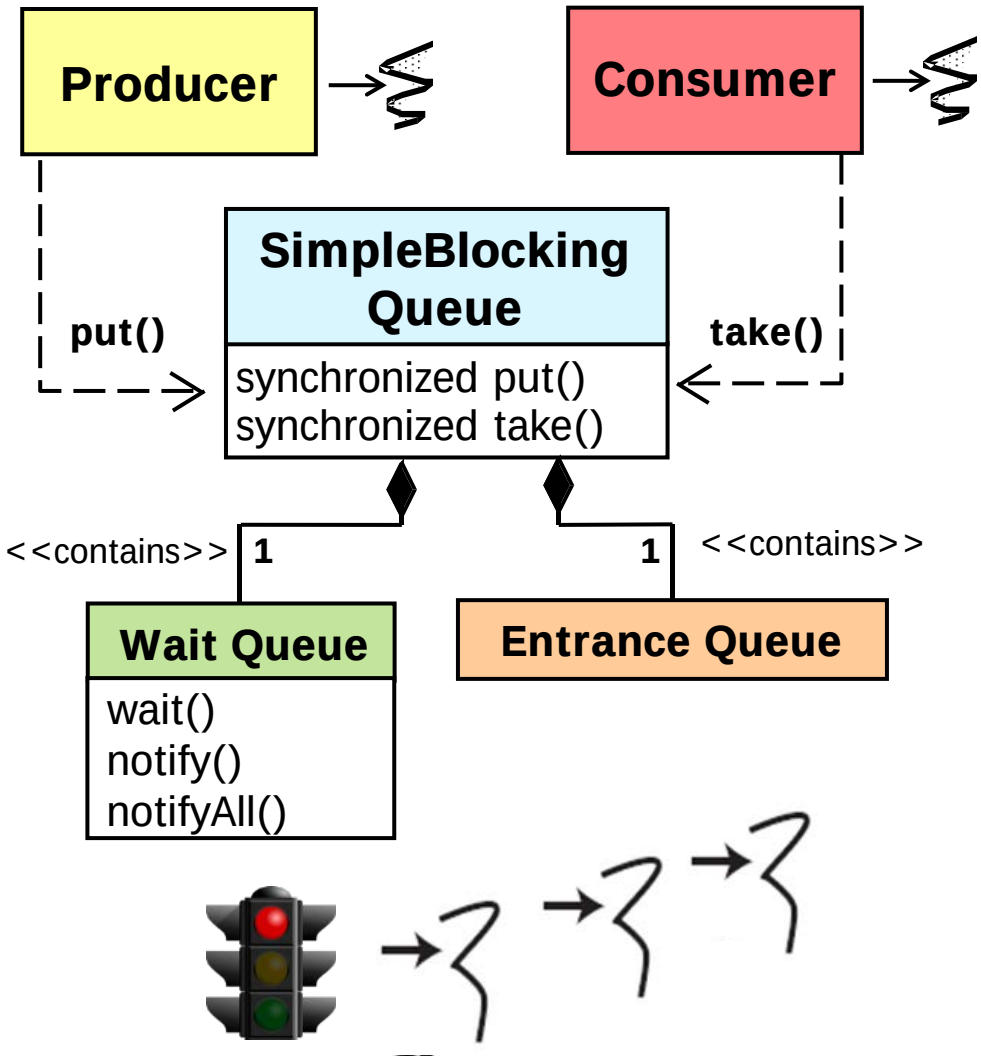
- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
- Android implements built-in monitor object synchronizers via POSIX mechanisms
- POSIX condition variables similar to Java ConditionObjects



See earlier part on “Java ConditionObjects”

Java Built-in Waiting & Notification Mechanisms

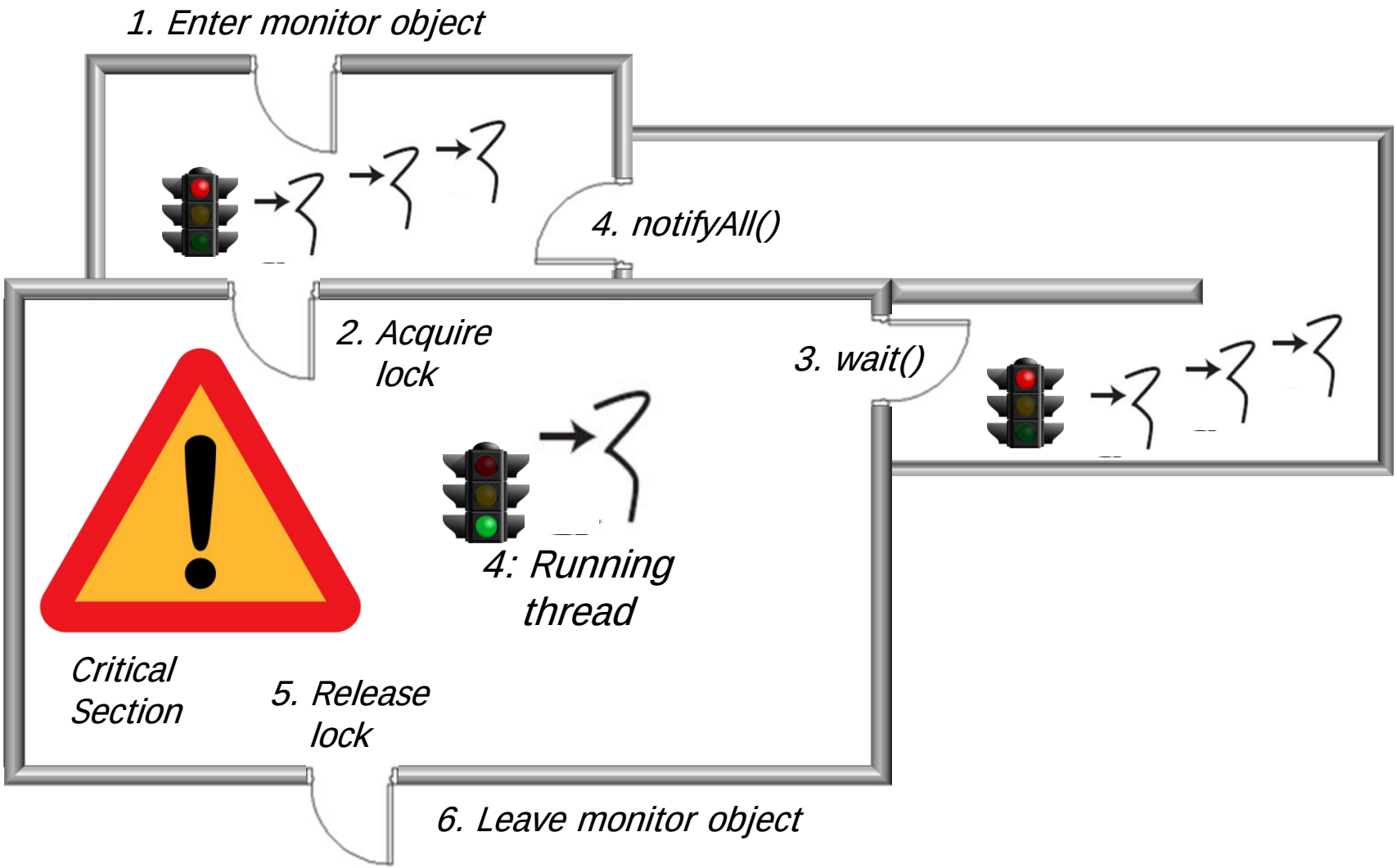
- Java monitor objects provide mechanisms that help threads interact cooperatively
- Each Java monitor object has *one* wait & *one* entrance queue
- Android implements built-in monitor object synchronizers via POSIX mechanisms
- POSIX condition variables similar to Java ConditionObjects
  - They are written in different languages, however



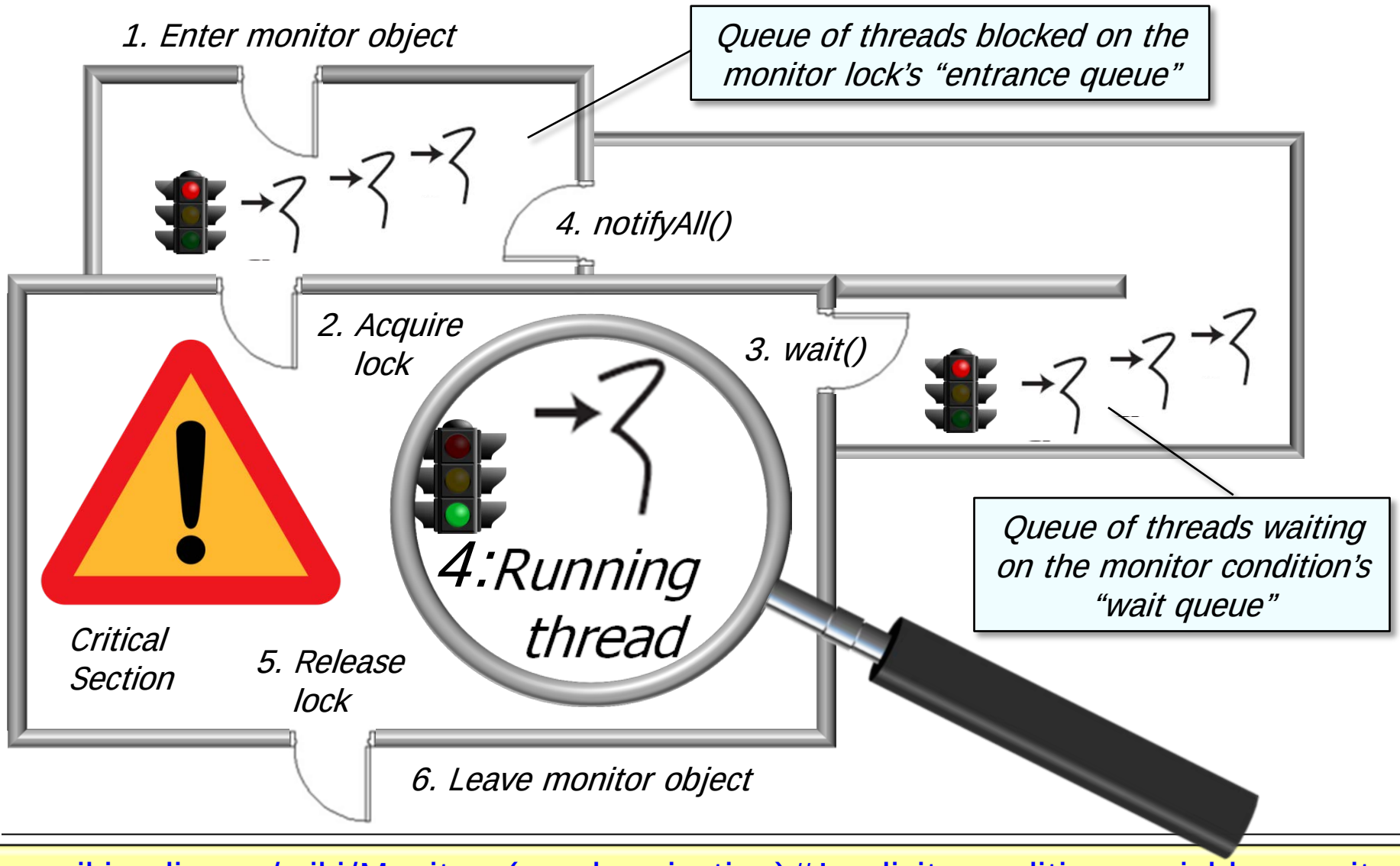
dalvik/vm/Sync.{h,cpp} has the Java monitor object source code for Android

Visual Analysis of Java Built-in Monitor Objects

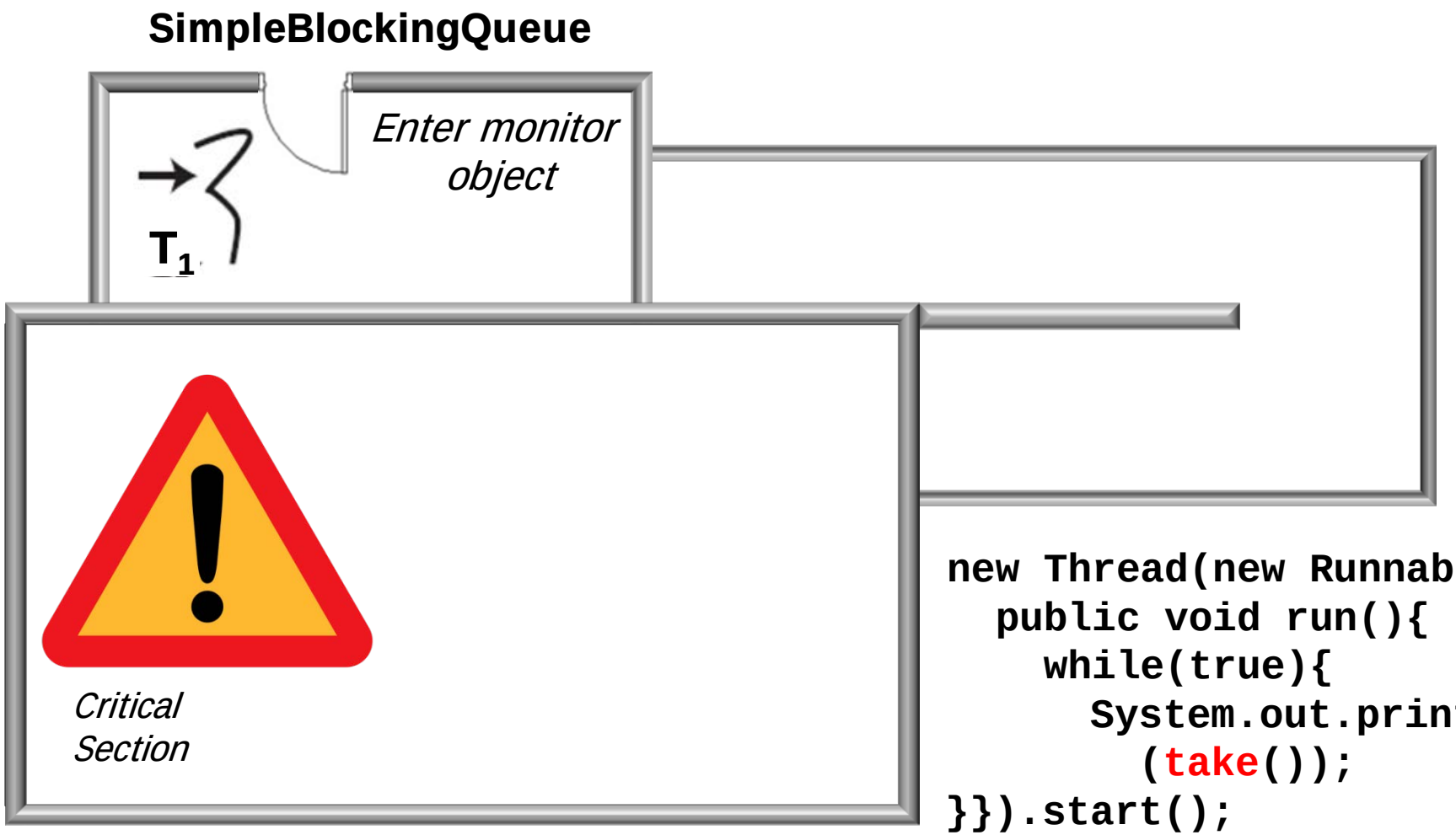
Visual Analysis of Java Built-in Monitor Objects



Visual Analysis of Java Built-in Monitor Objects

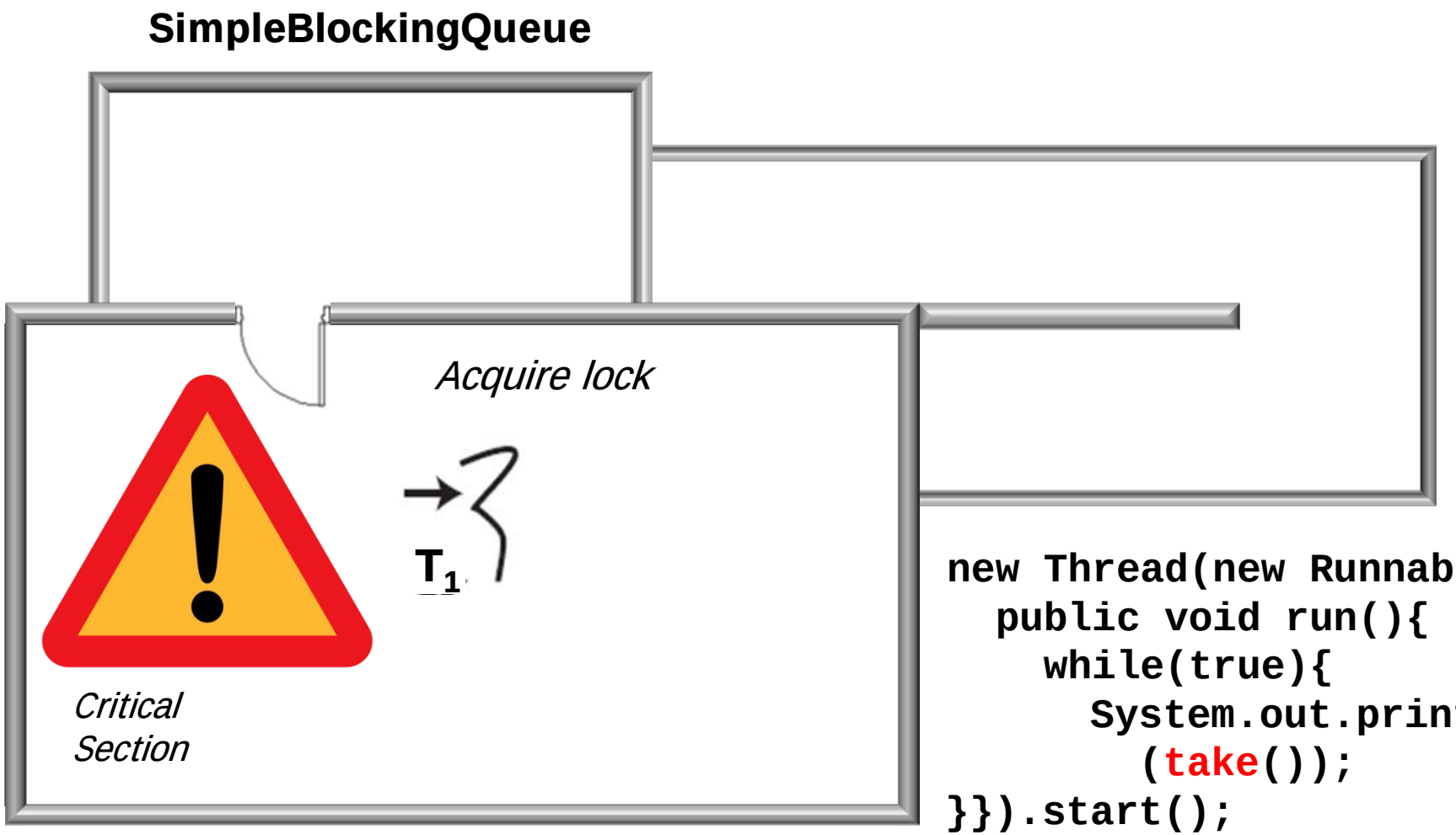


Visual Analysis of Java Built-in Monitor Objects



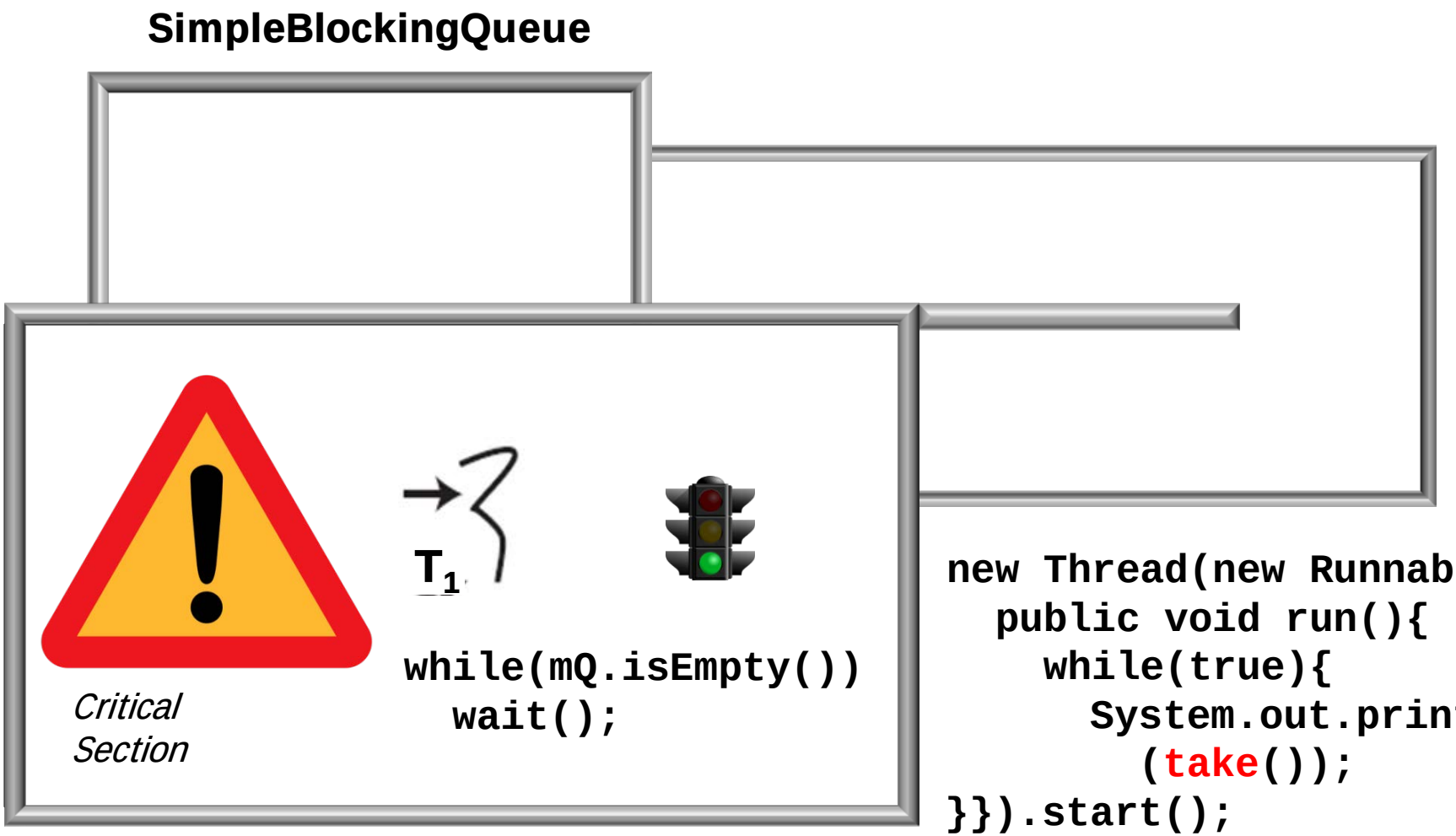

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of Java Built-in Monitor Objects

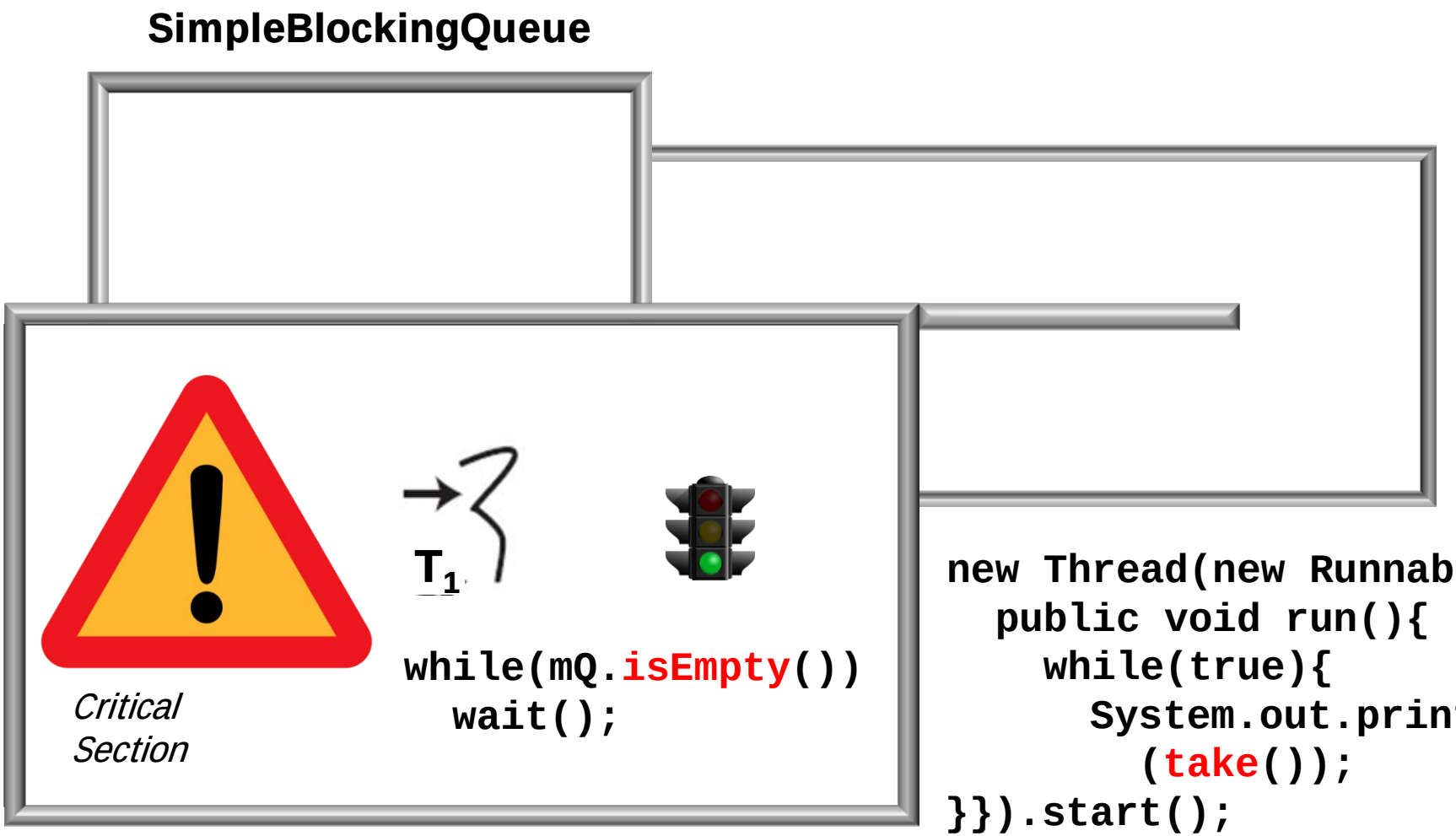



```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

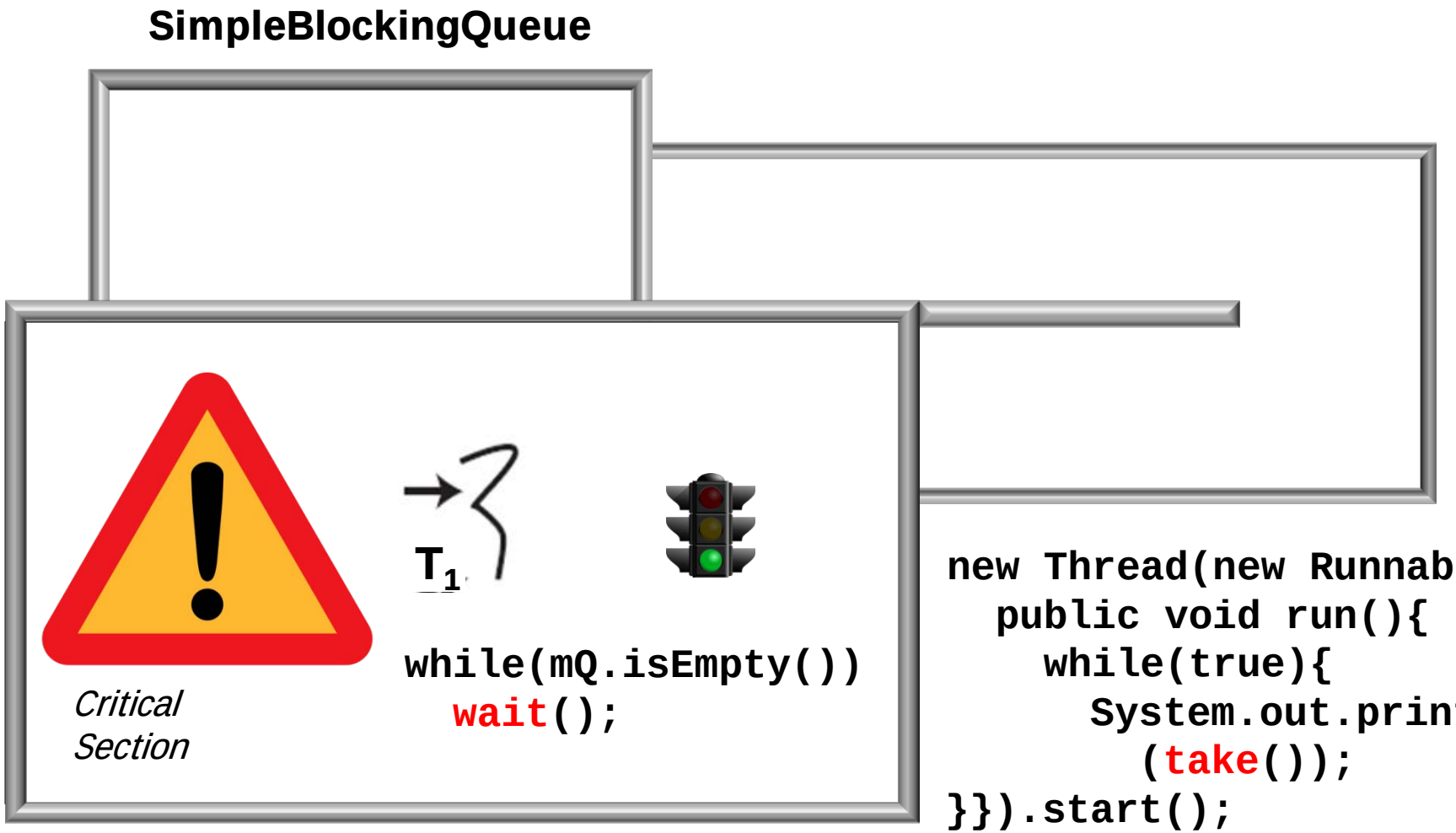
Visual Analysis of Java Built-in Monitor Objects



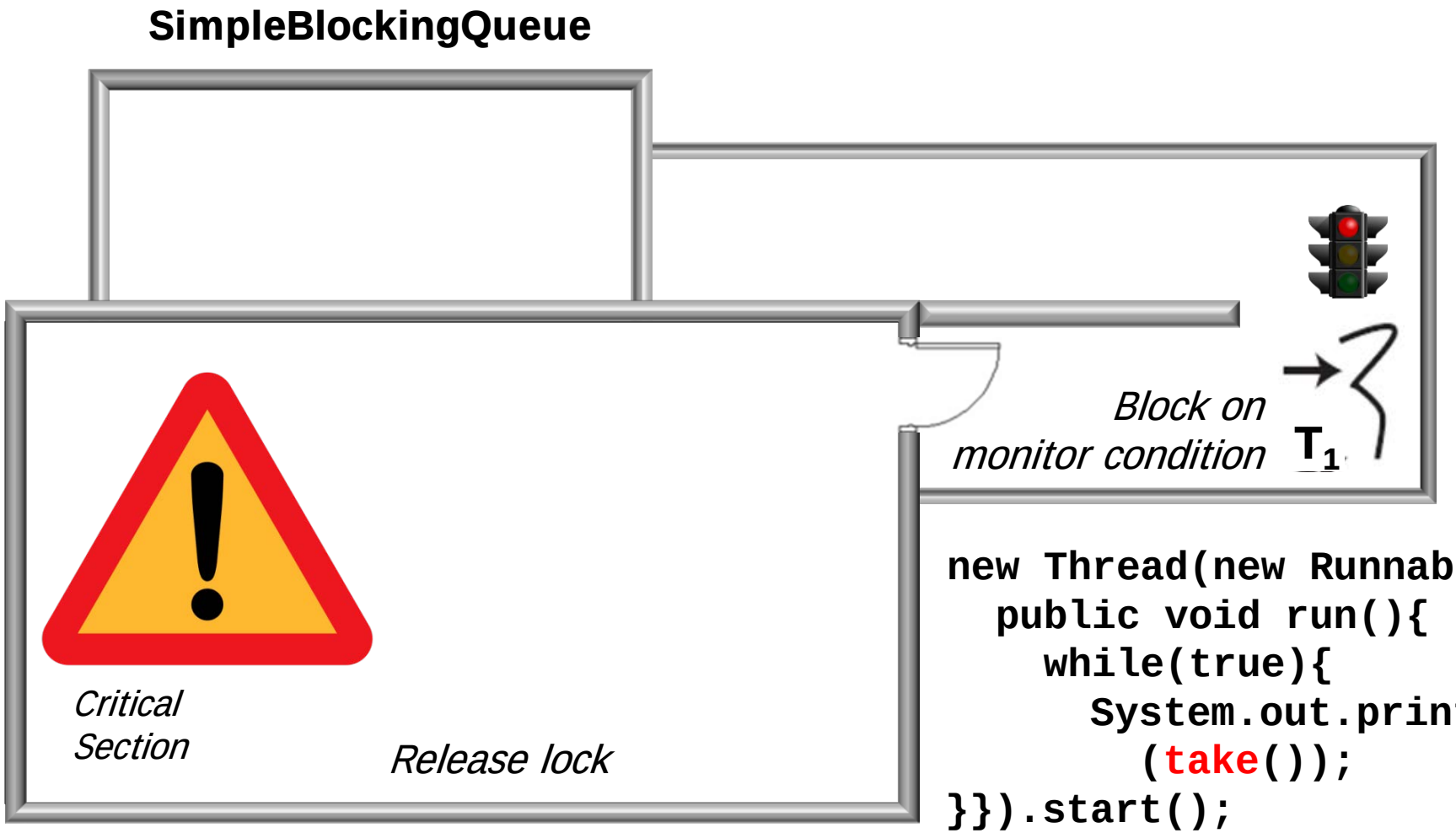
Visual Analysis of Java Built-in Monitor Objects



Visual Analysis of Java Built-in Monitor Objects

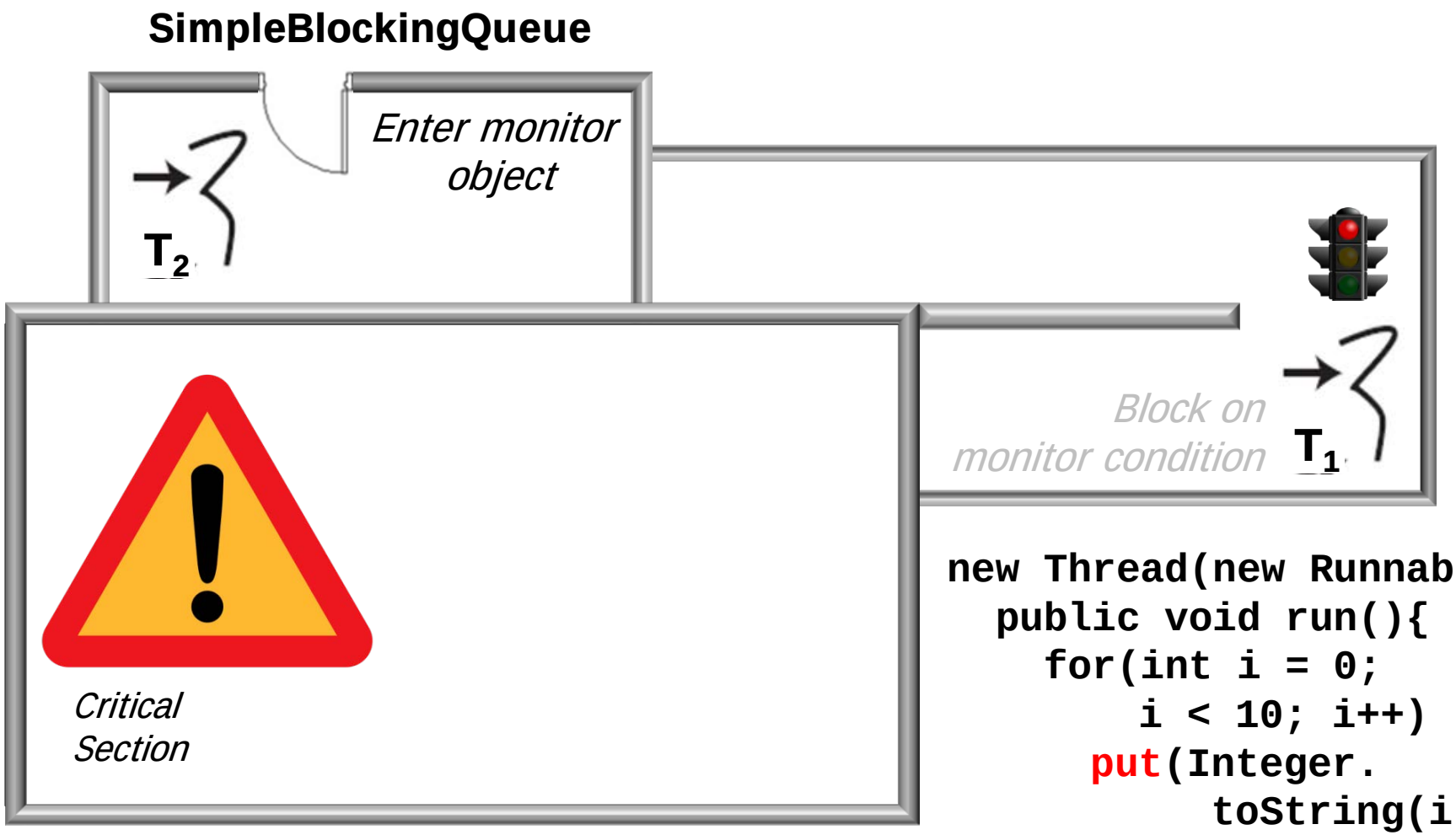


Visual Analysis of Java Built-in Monitor Objects



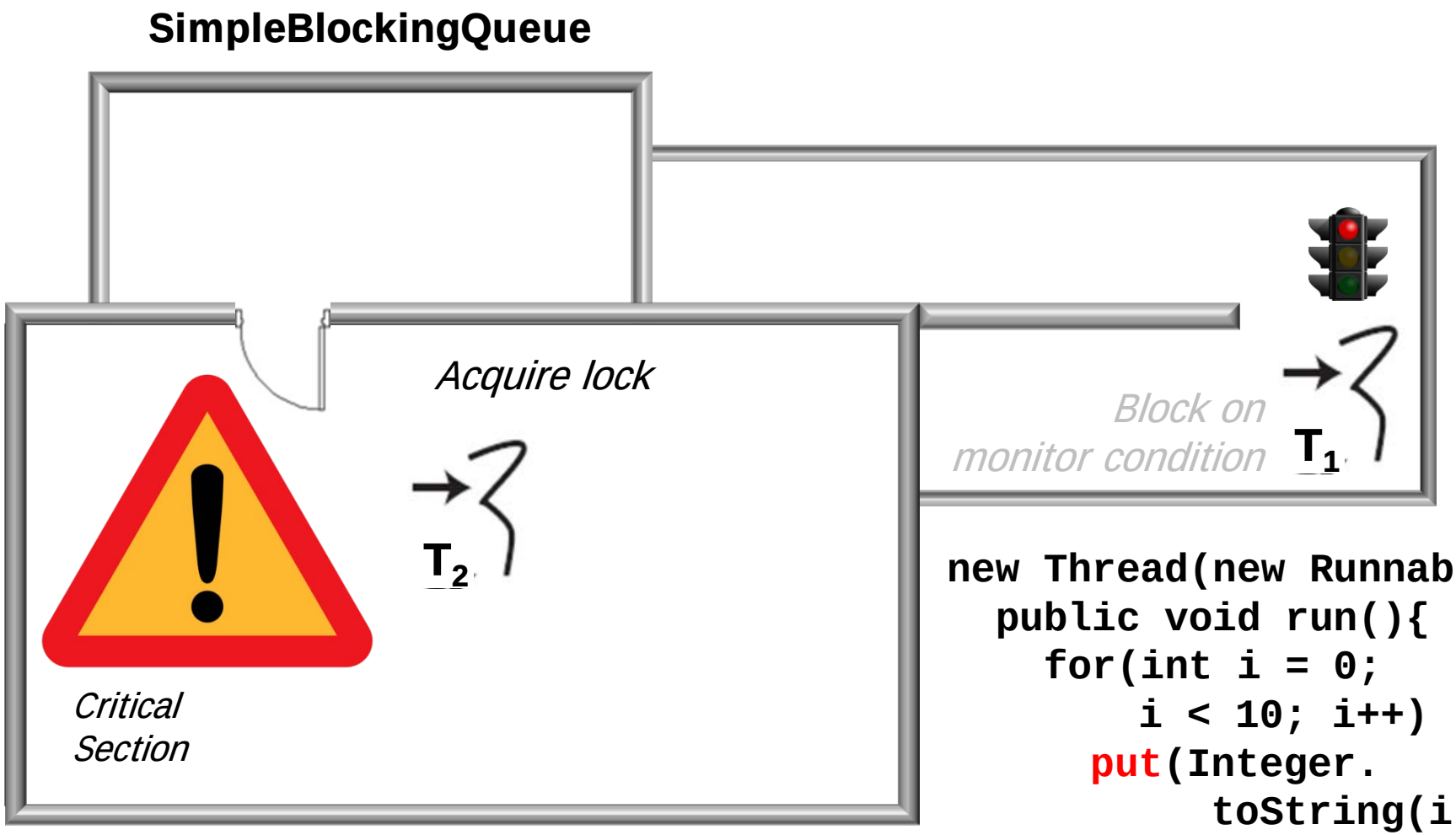

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of Java Built-in Monitor Objects



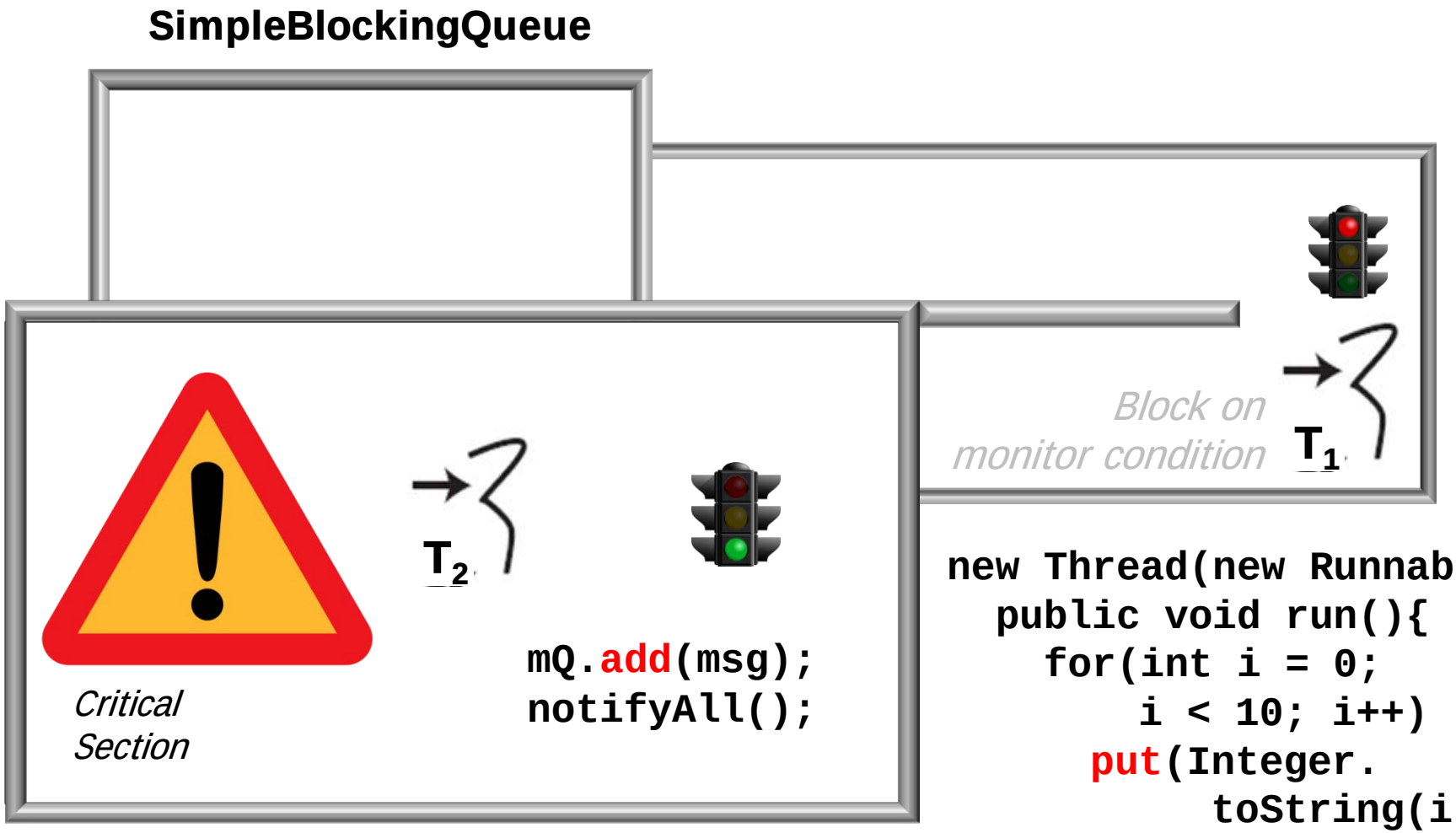

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of Java Built-in Monitor Objects

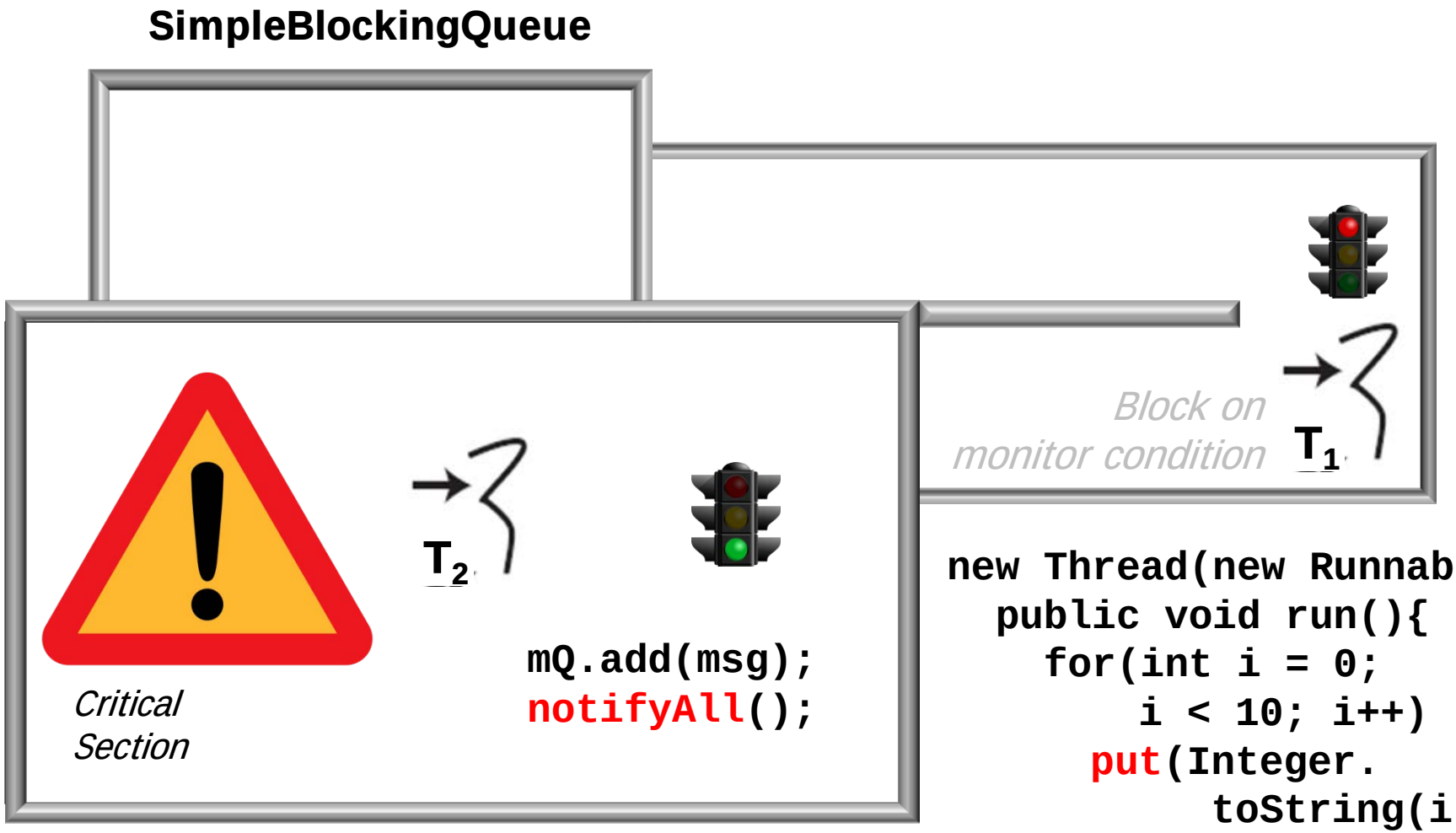



```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of Java Built-in Monitor Objects

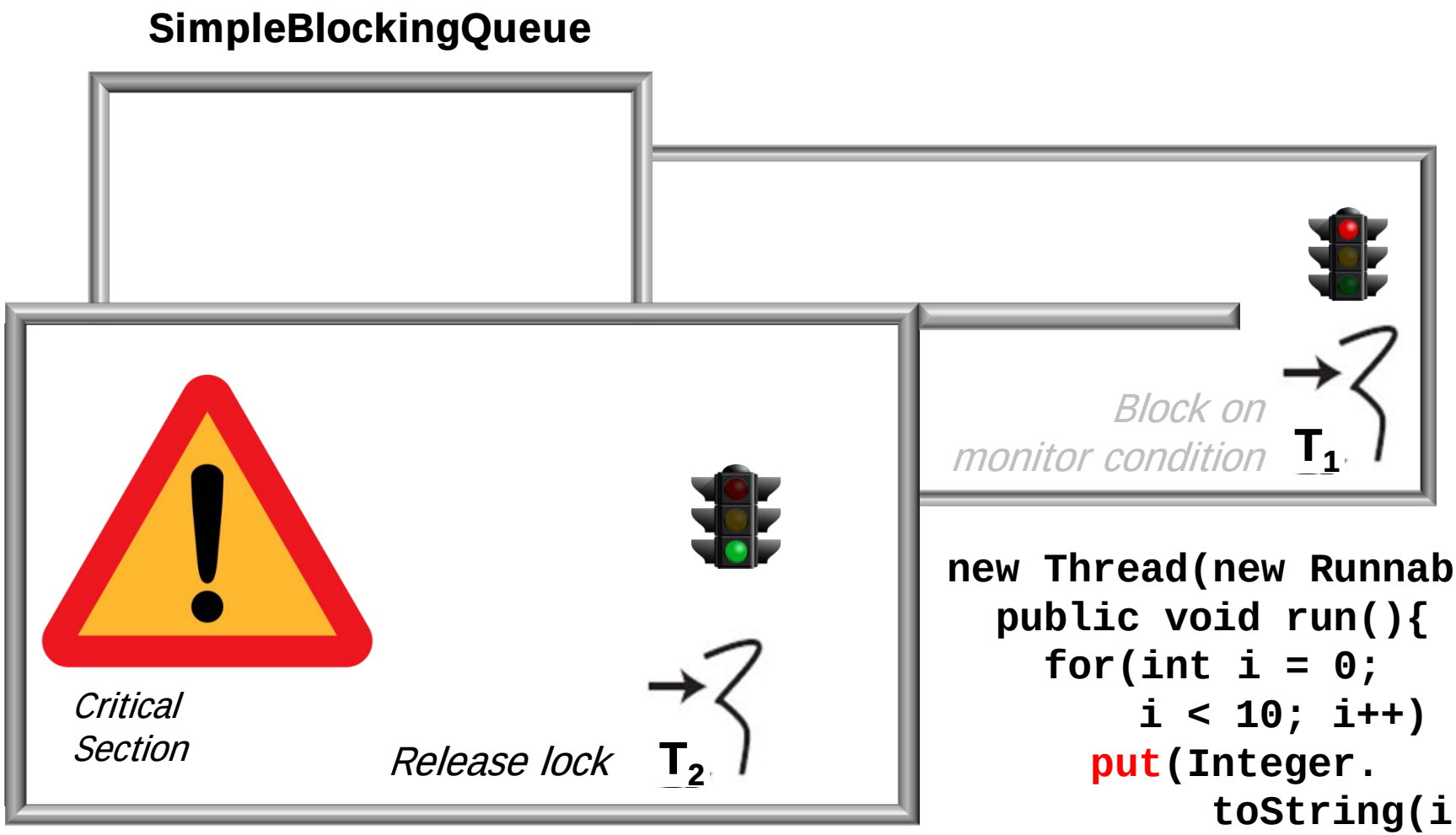


Visual Analysis of Java Built-in Monitor Objects



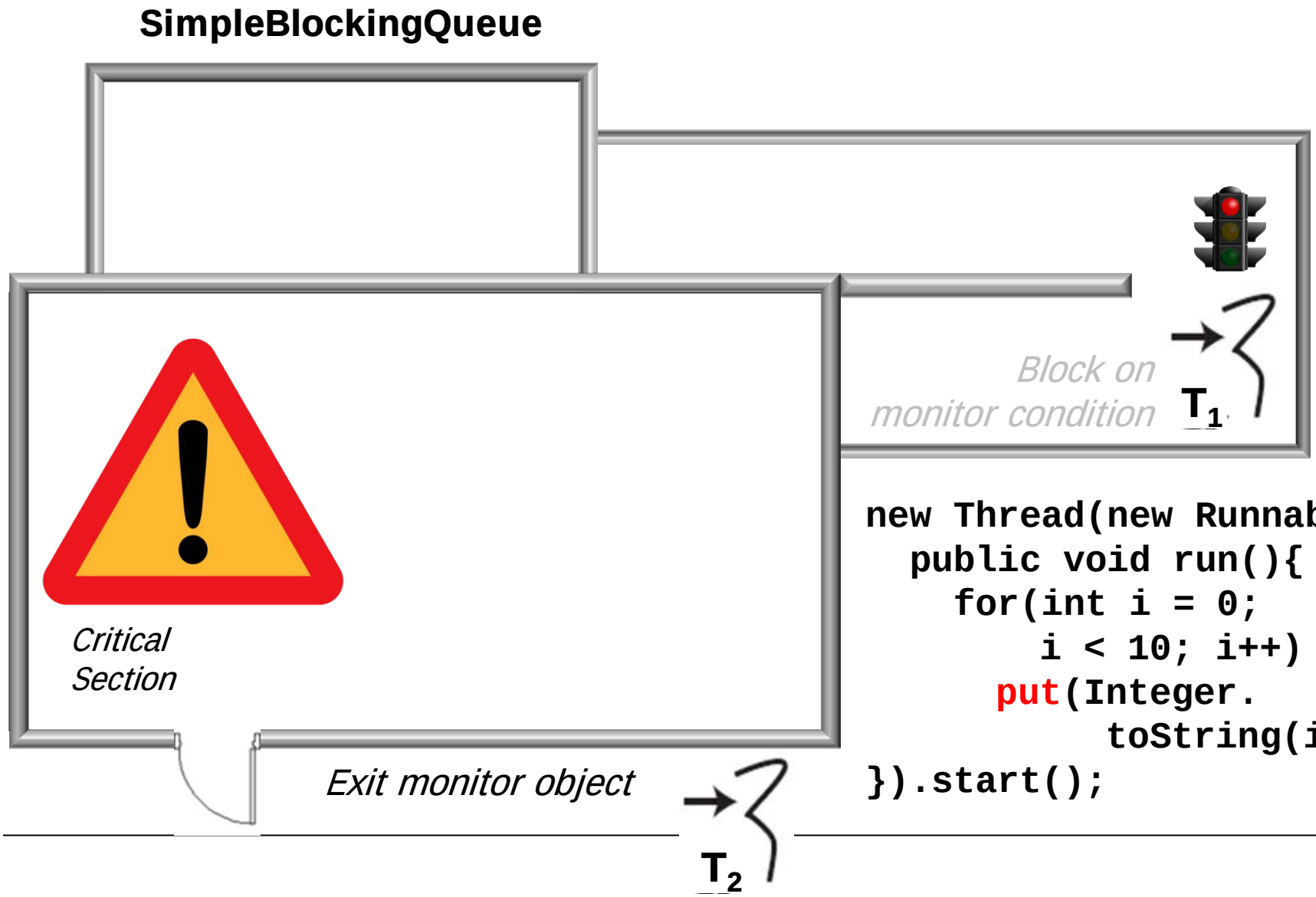

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of Java Built-in Monitor Objects



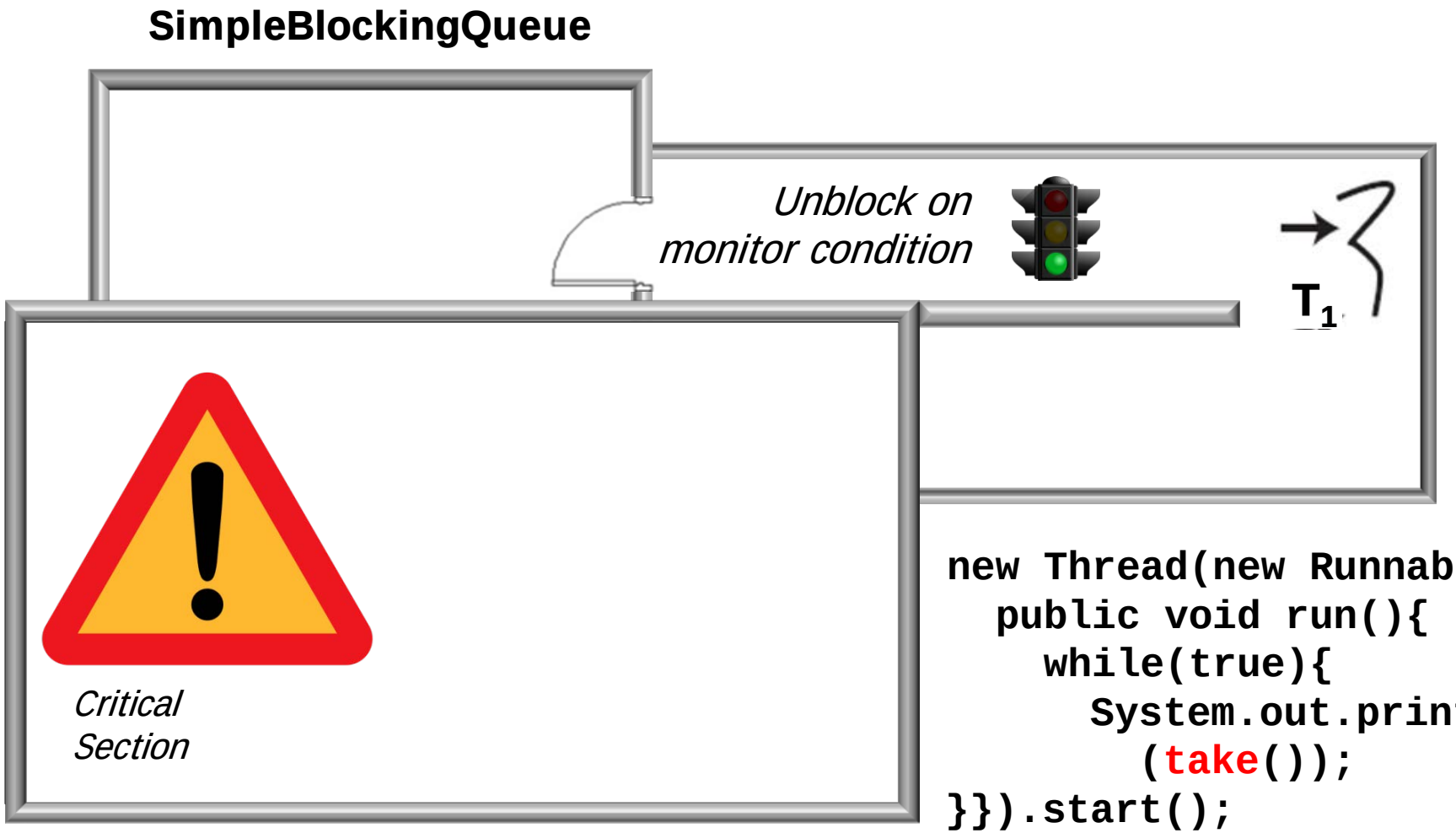

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of Java Built-in Monitor Objects



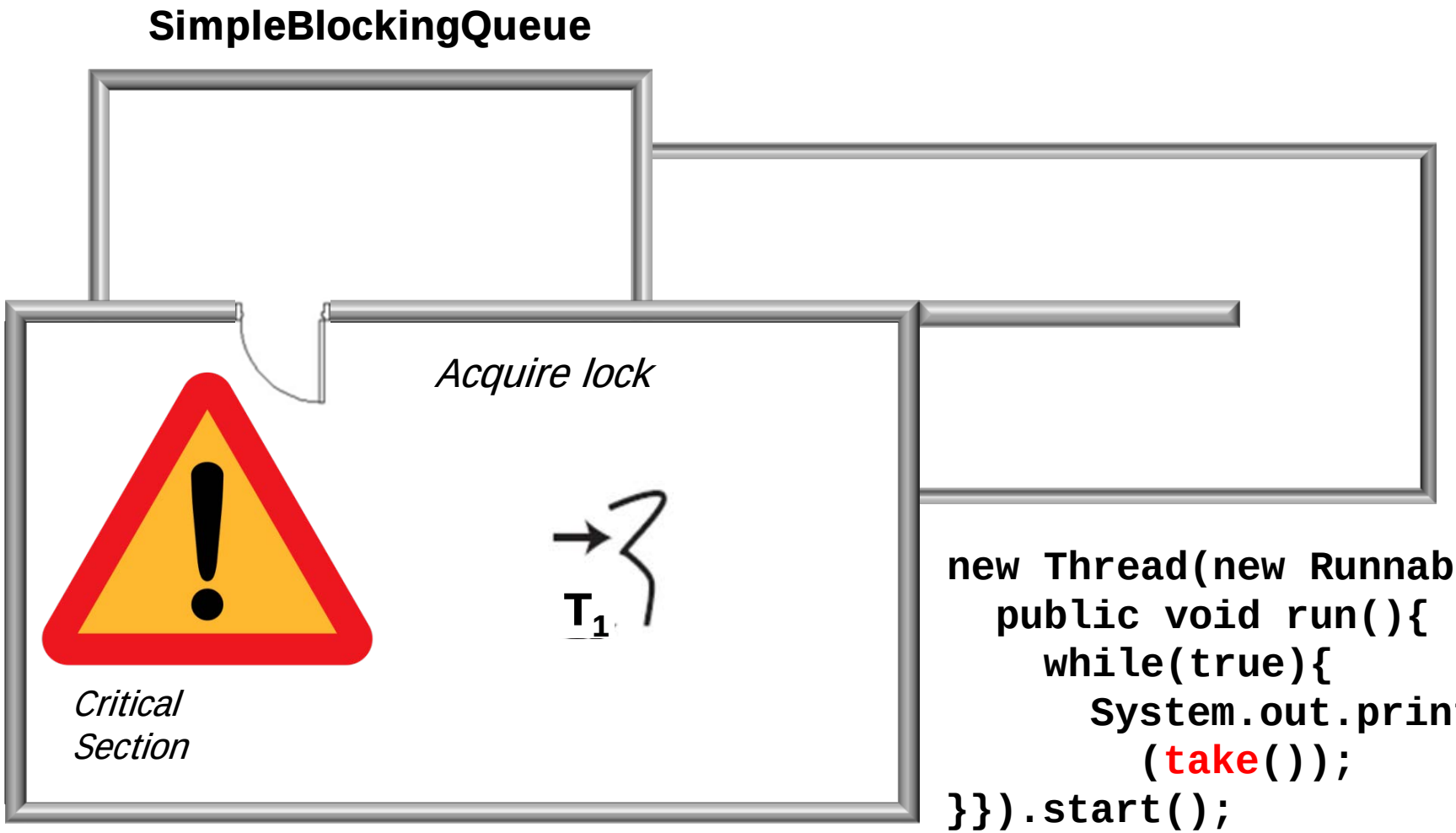

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of Java Built-in Monitor Objects



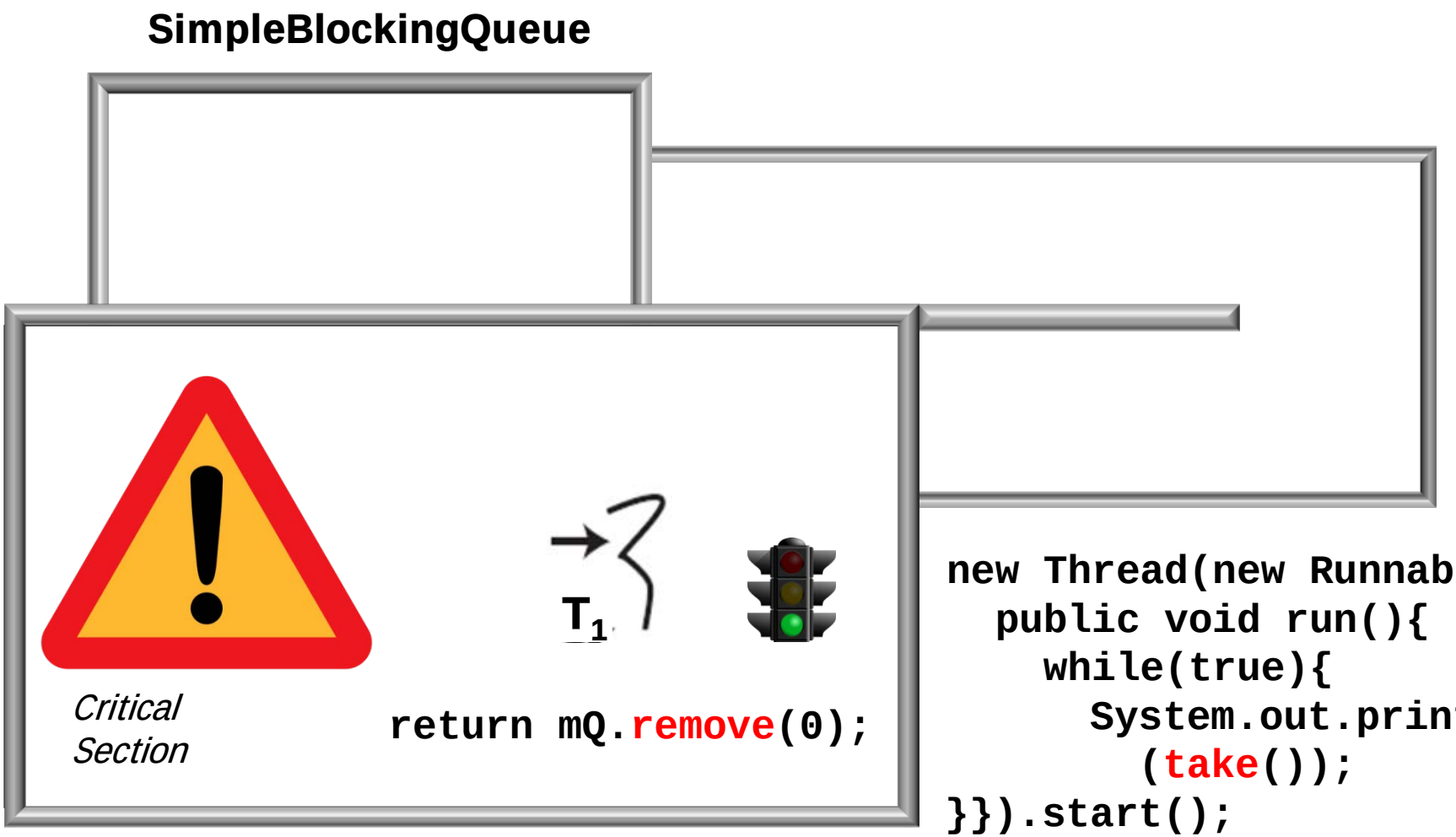

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of Java Built-in Monitor Objects

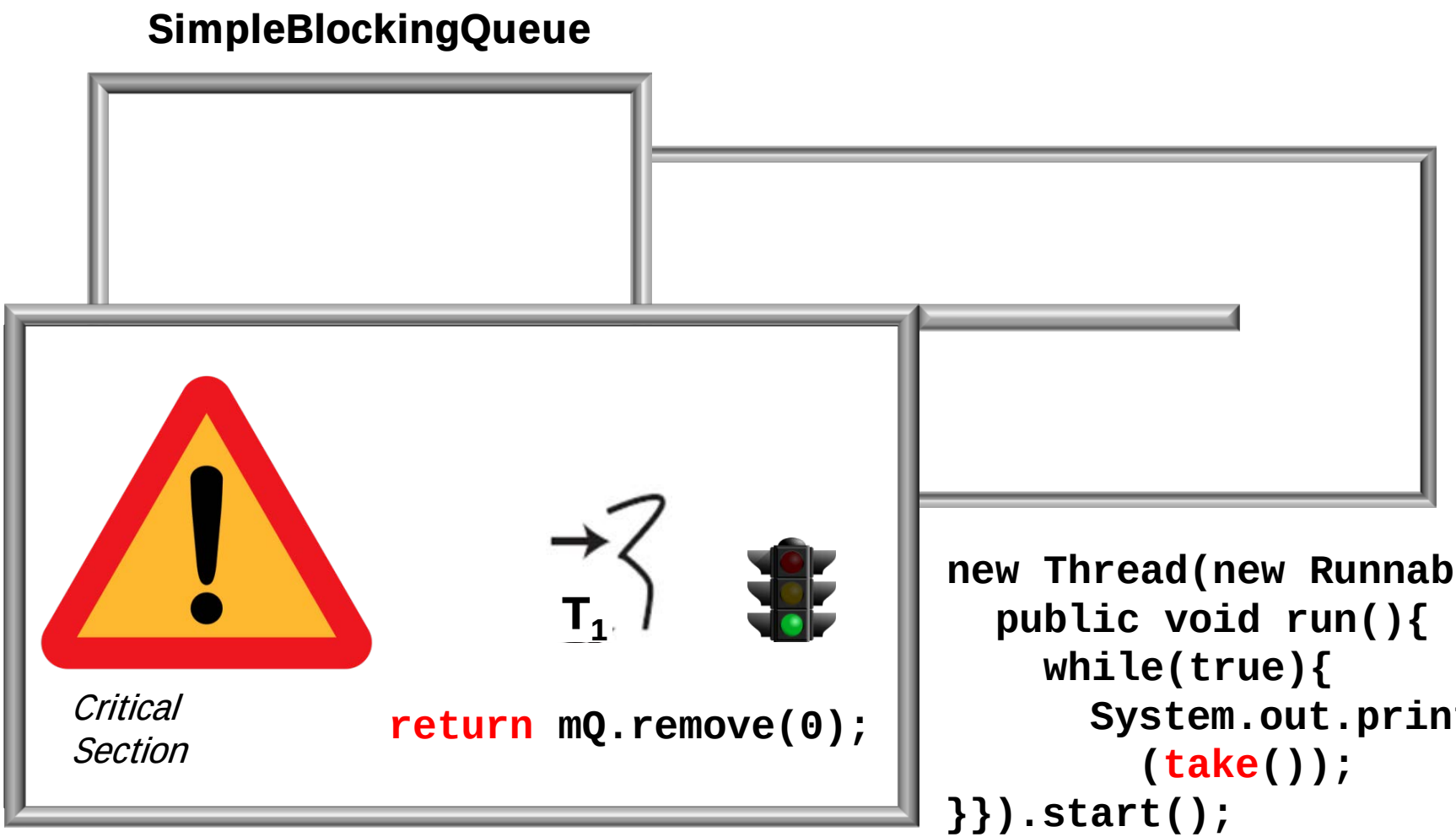



```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

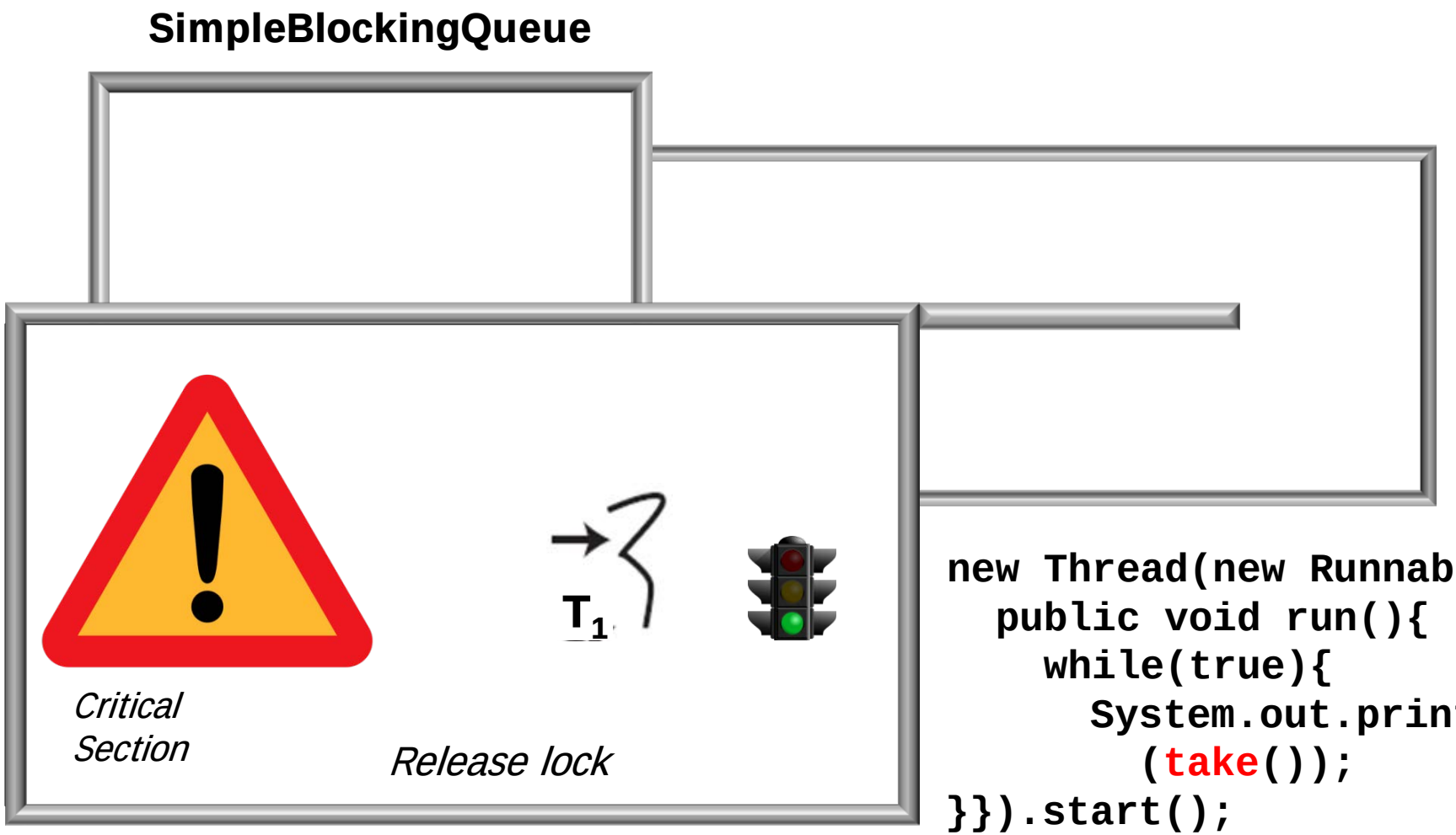
Visual Analysis of Java Built-in Monitor Objects



Visual Analysis of Java Built-in Monitor Objects

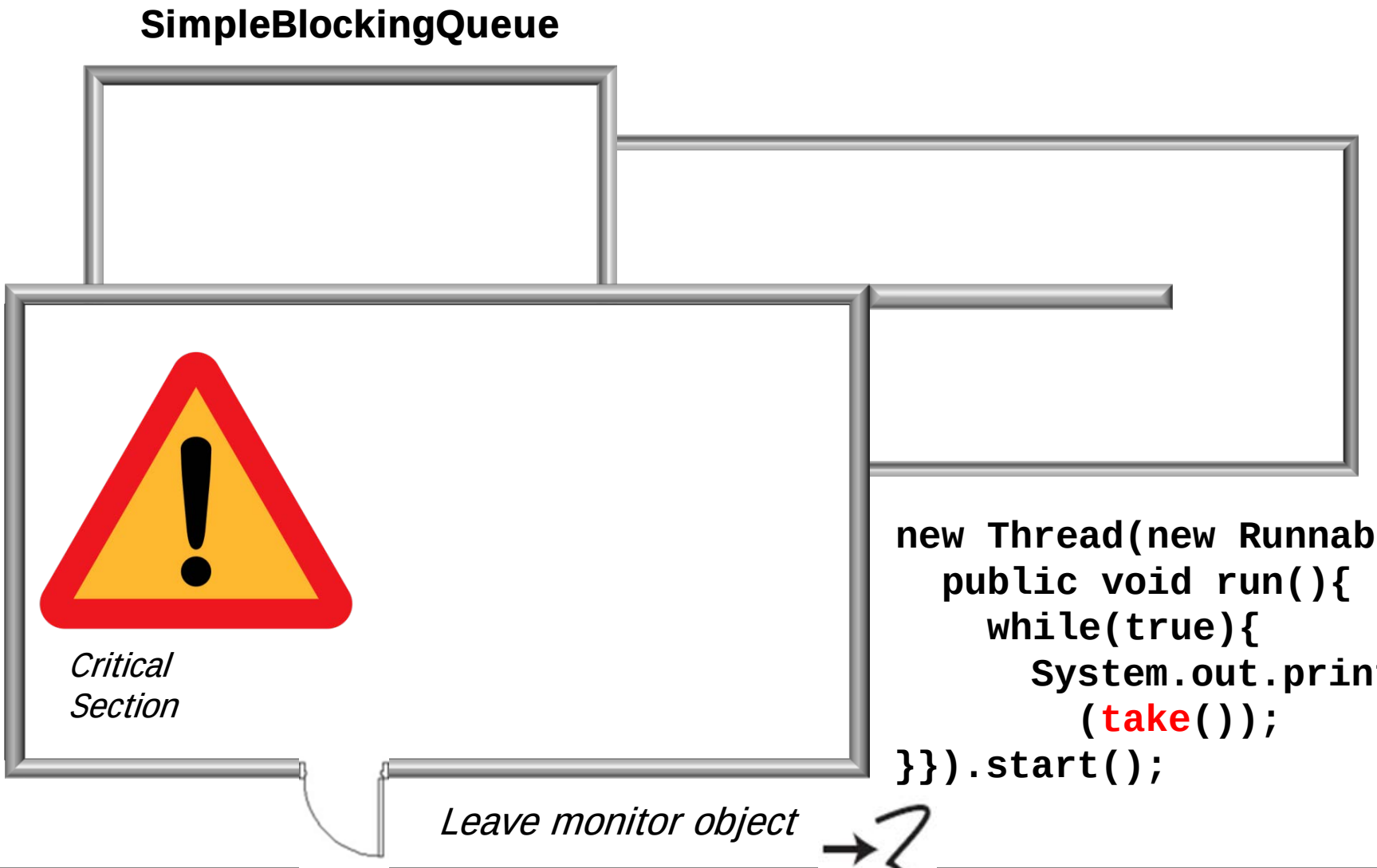


Visual Analysis of Java Built-in Monitor Objects



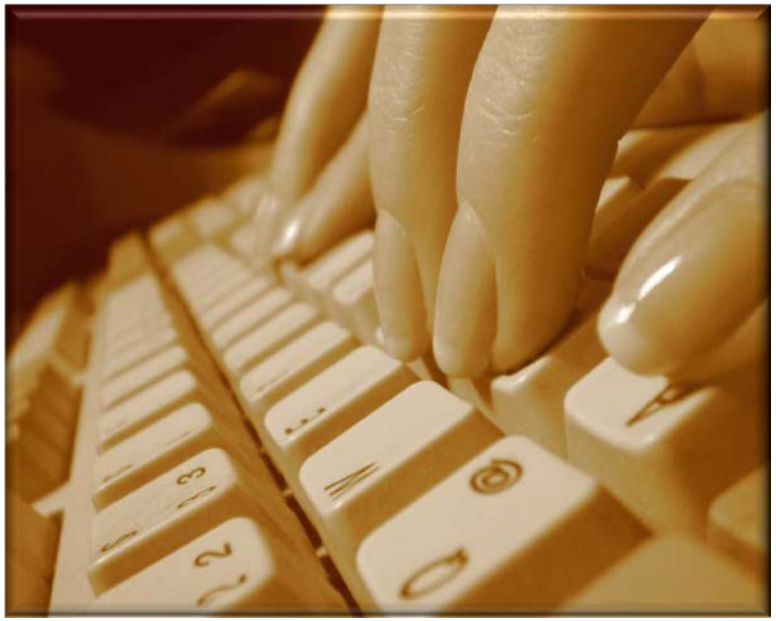

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of Java Built-in Monitor Objects



Code Analysis of Java Built-in Monitor Objects

Code Analysis of Java Built-in Monitor Objects



```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

en.wikipedia.org/wiki/Guarded_suspension has more on *Guarded Suspension*

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
  - e.g., take() acquires the monitor lock, checks the queue size, & blocks if the queue is empty

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
  - e.g., take() acquires the monitor lock, checks the queue size, & blocks if the queue is empty

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
  - e.g., take() acquires the monitor lock, checks the queue size, & blocks if the queue is empty

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
  - A waiting thread can’t assume a notification it receives is for its condition

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
  - A waiting thread can’t assume a notification it receives is for its condition
  - It also can’t assume the condition is even still true!

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
  - A waiting thread can’t assume a notification it receives is for its condition
  - It also can’t assume the condition is even still true!

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

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

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true
- When a blocked thread is notified, it wakes up, obtains the monitor lock, continues after wait(), & releases the lock when it returns

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true
- When a blocked thread is notified, it wakes up, obtains the monitor lock, continues after wait(), & releases the lock when it returns

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Code Analysis of Java Built-in Monitor Objects

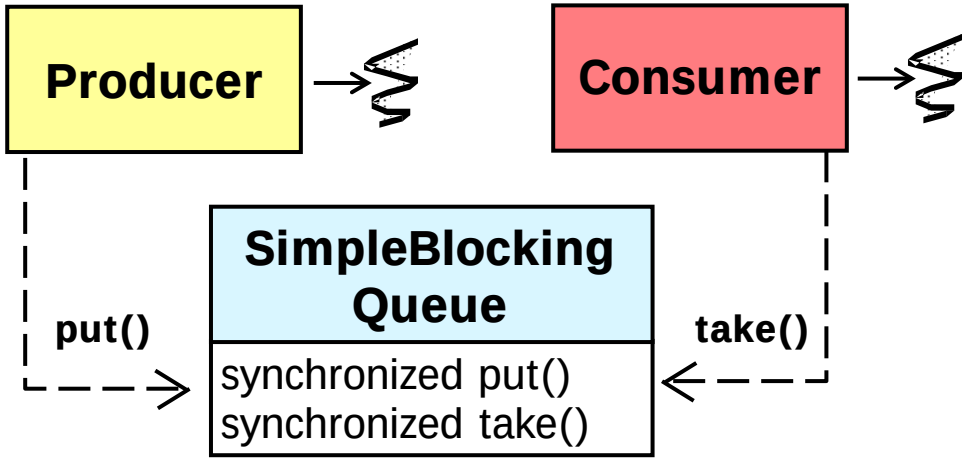
- Inside a synchronized method, a thread can “wait” for a condition
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won’t continue until another thread notifies it that the condition may be true
- When a blocked thread is notified, it wakes up, obtains the monitor lock, continues after wait(), & releases the lock when it returns

```
public class SimpleBlockingQueue {
  private List<String> mQ =
    new ArrayList<String>();

  public synchronized
    void put(String msg){
    ...
    mQ.add(msg);
    notifyAll();
  }

  public synchronized String take(){
    while (mQ.isEmpty()) {
      wait();
    }
    ...
    return mQ.remove(0);
  }
```

Summary

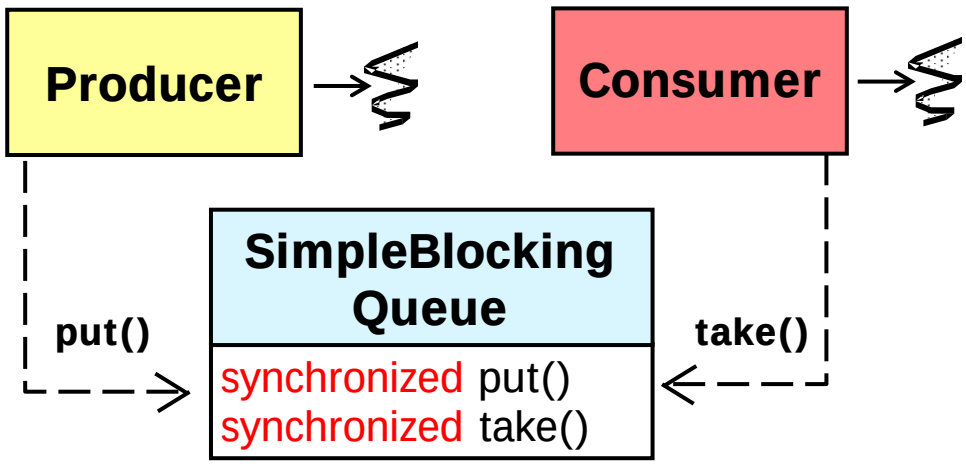
Summary

- Any Java object may be used as a monitor object



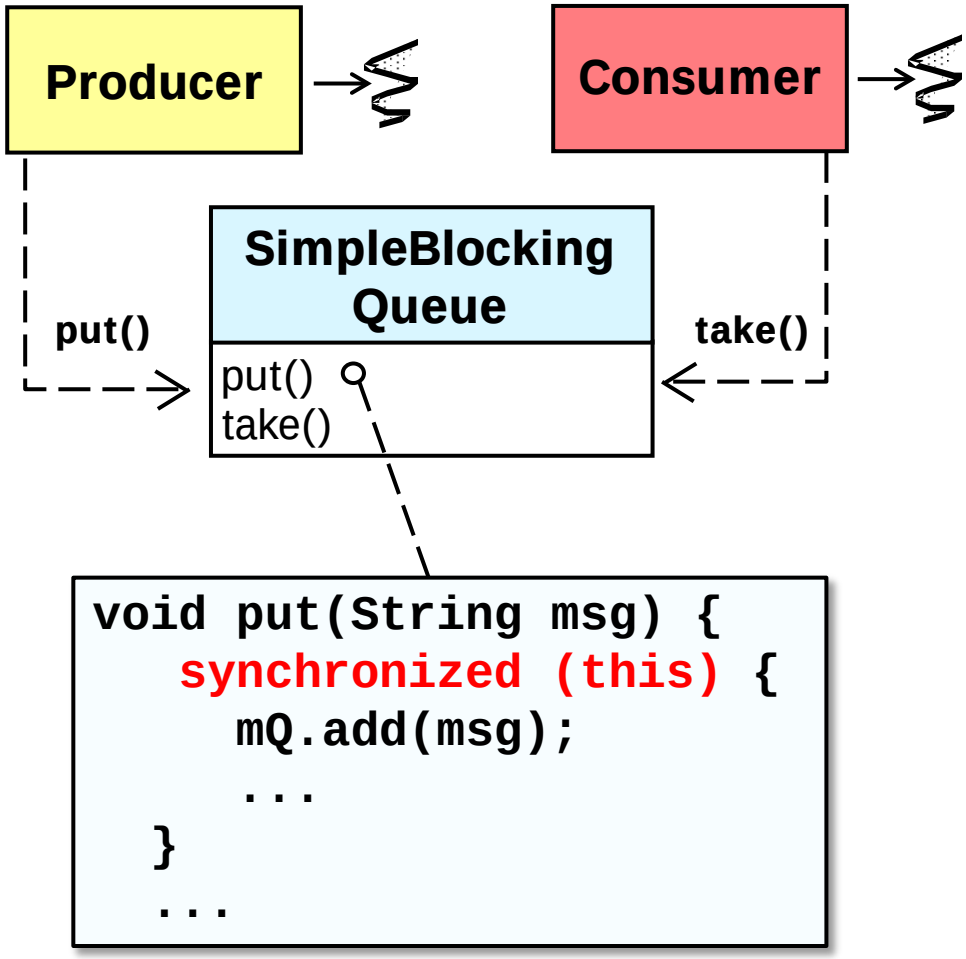
Summary

- Any Java object may be used as a monitor object
  - Methods requiring mutual exclusion must be marked as synchronized



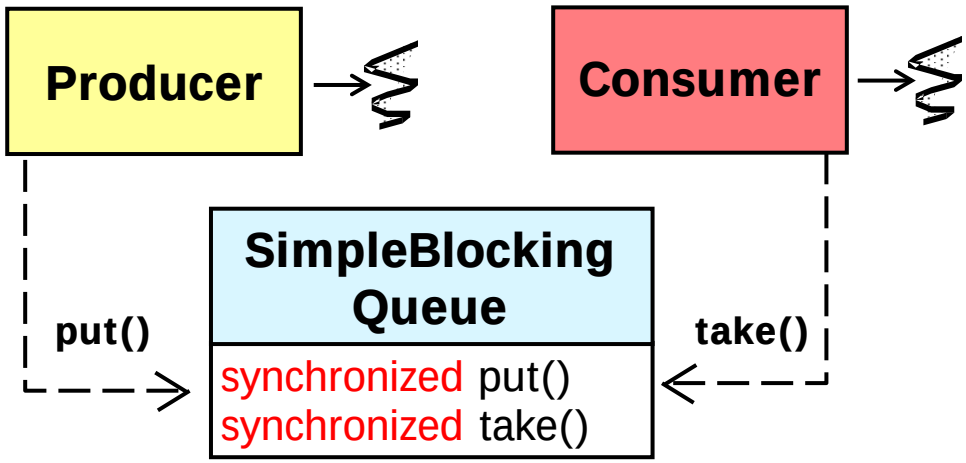
Summary

- Any Java object may be used as a monitor object
  - Methods requiring mutual exclusion must be marked as synchronized
  - Blocks of code may also be marked by synchronized



Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution

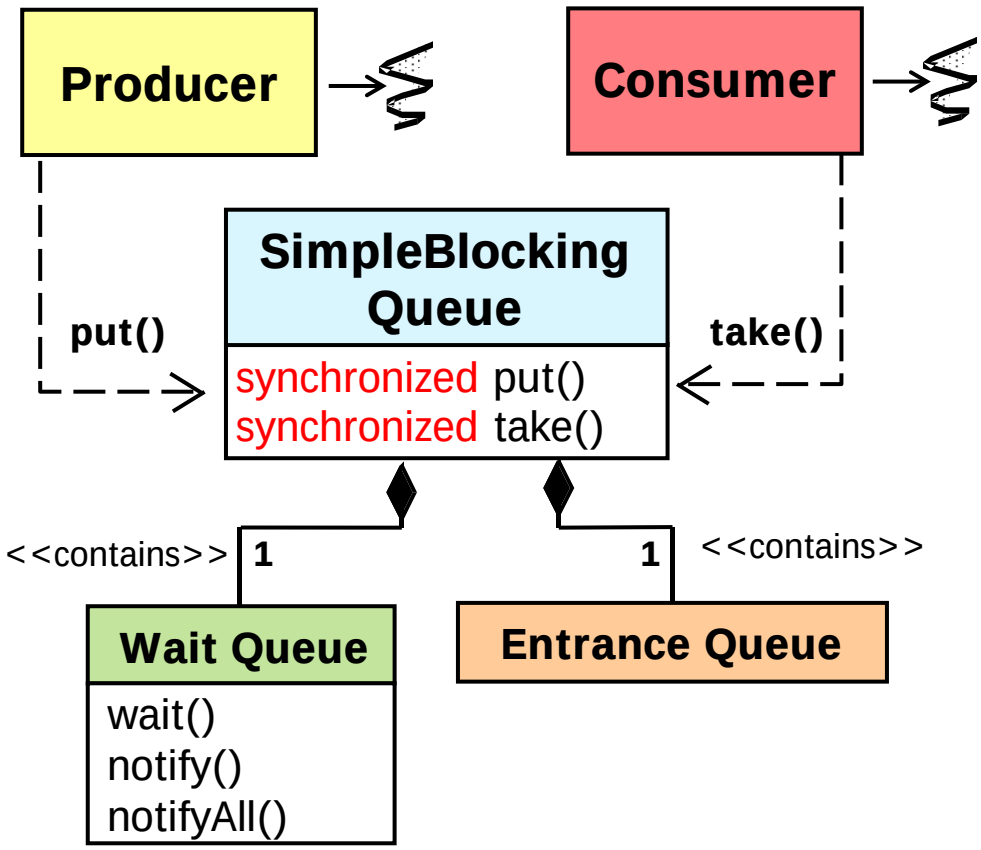



```
public void synchronized
  put(String msg){
    mQ.add(msg);
}

public String synchronized
  take(){
    return mQ.remove(0);
}
```

Concurrent calls to these methods will "busy wait" or worse..

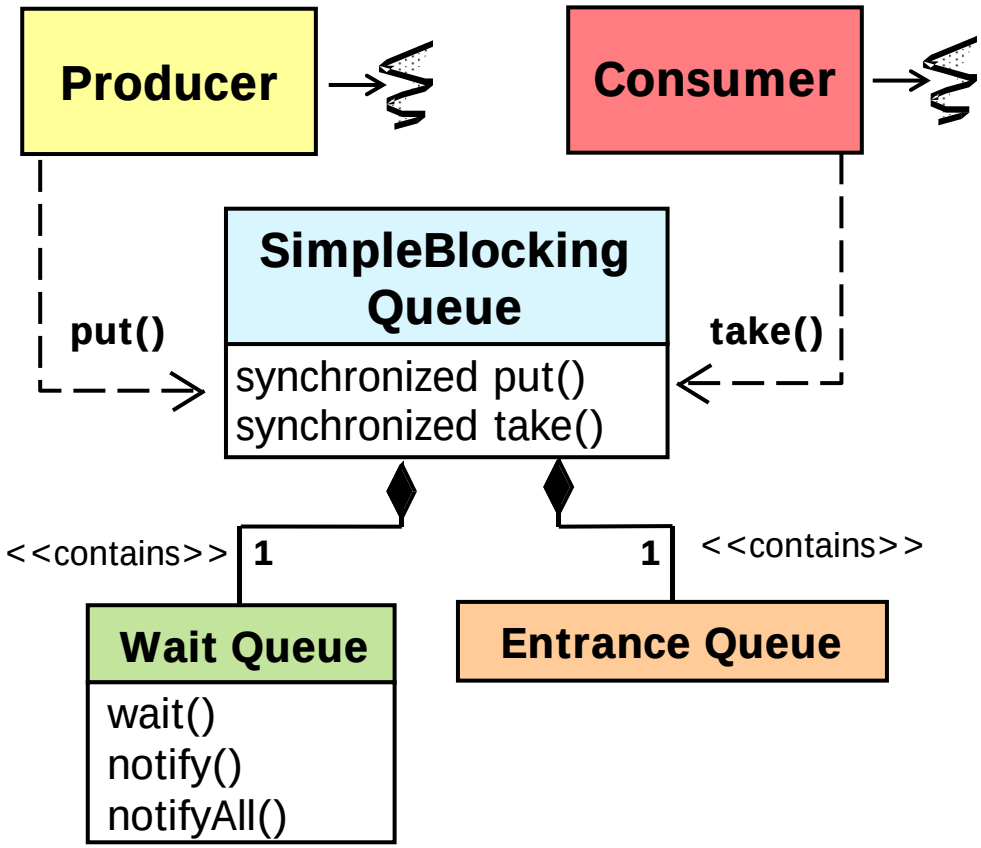
Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
  - Therefore, built-in monitor objects provide waiting & notification mechanisms



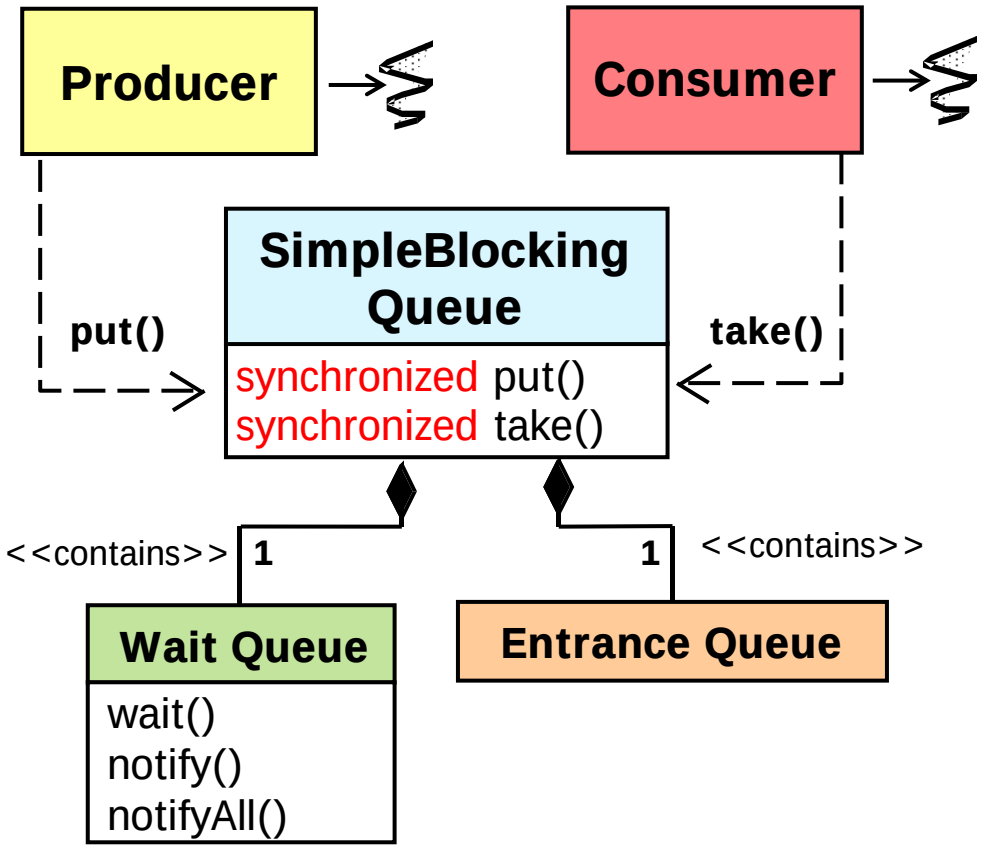
Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects



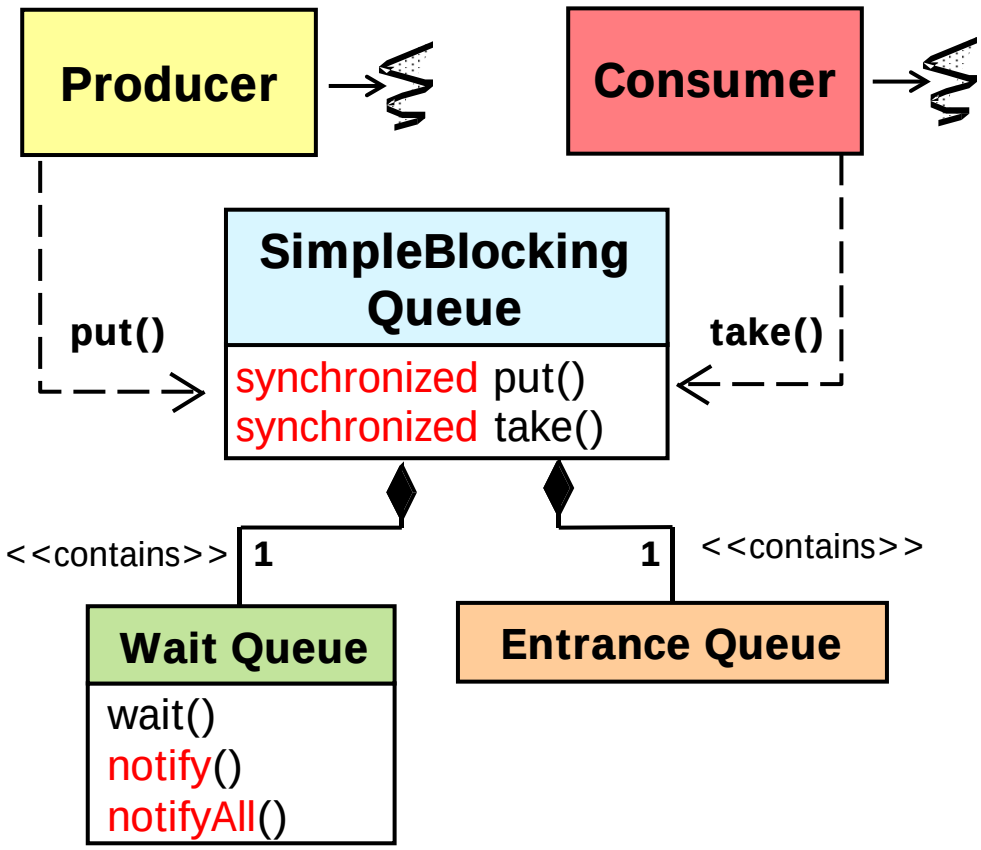
Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
  - Nested monitor lockout



Summary

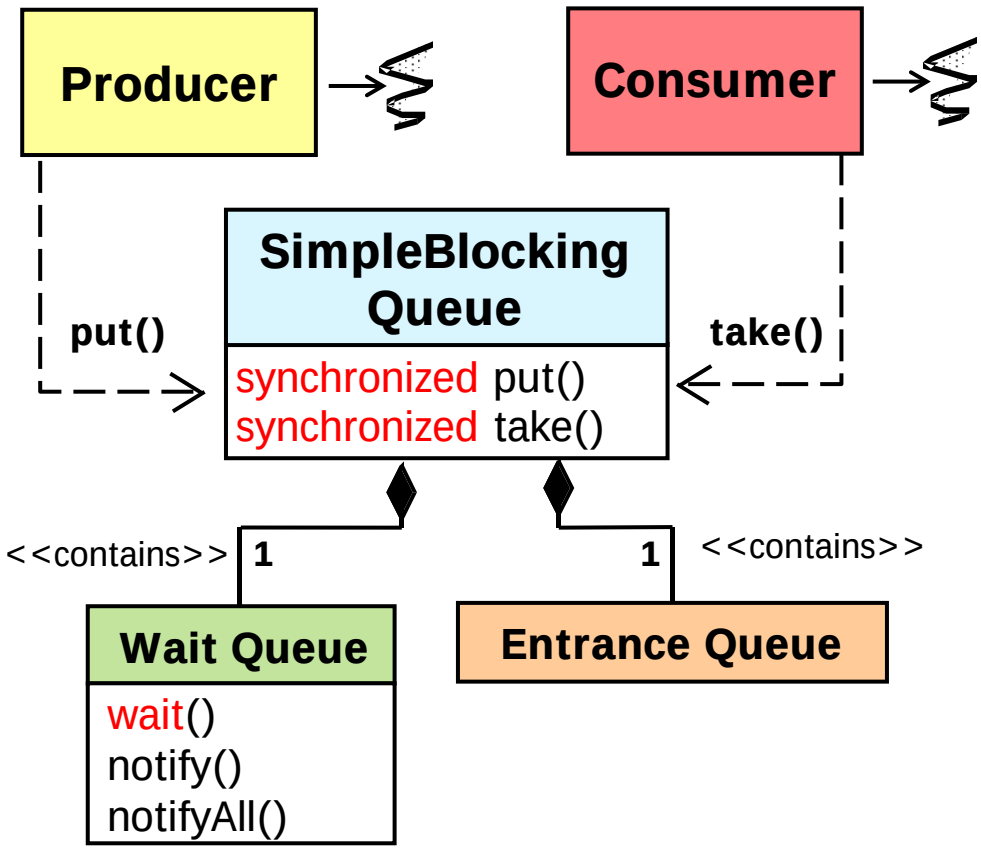
- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
  - Nested monitor lockout
  - Subtleties associated with calling notify() vs. notifyAll()



stackoverflow.com/questions/37026/java-notify-vs-notifyall-all-over-again

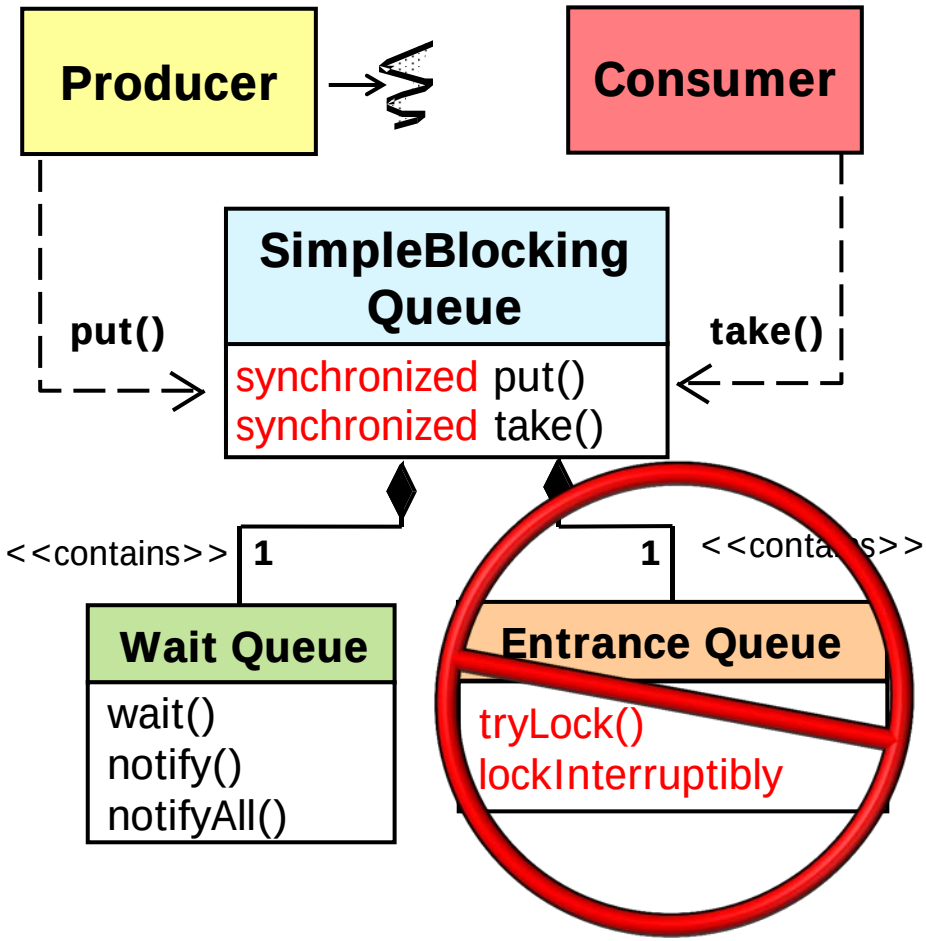
Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
  - Nested monitor lockout
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting threads are notified



Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
  - Nested monitor lockout
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting threads are notified
  - Monitor locks lack certain features provided by ReentrantLock



Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
- You may need more than Java's built-in monitor mechanisms
  - java.util.concurrent & java.util.concurrent.locks

package **Added in API level 1**

java.util.concurrent.locks

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax.
The `Lock` interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is `ReentrantLock`.

package **Added in API level 1**

java.util.concurrent

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the `java.util.concurrent.locks` and `java.util.concurrent.atomic` packages.

developer.android.com/reference/java/util/concurrent/package-summary.html

Summary

- Any Java object may be used as a monitor object
- Synchronized methods & statements are not a complete solution
- Be aware of certain issues with Java built-in monitor objects
- You may need more than Java's built-in monitor mechanisms
  - java.util.concurrent & java.util.concurrent.locks
  - Android concurrency frameworks