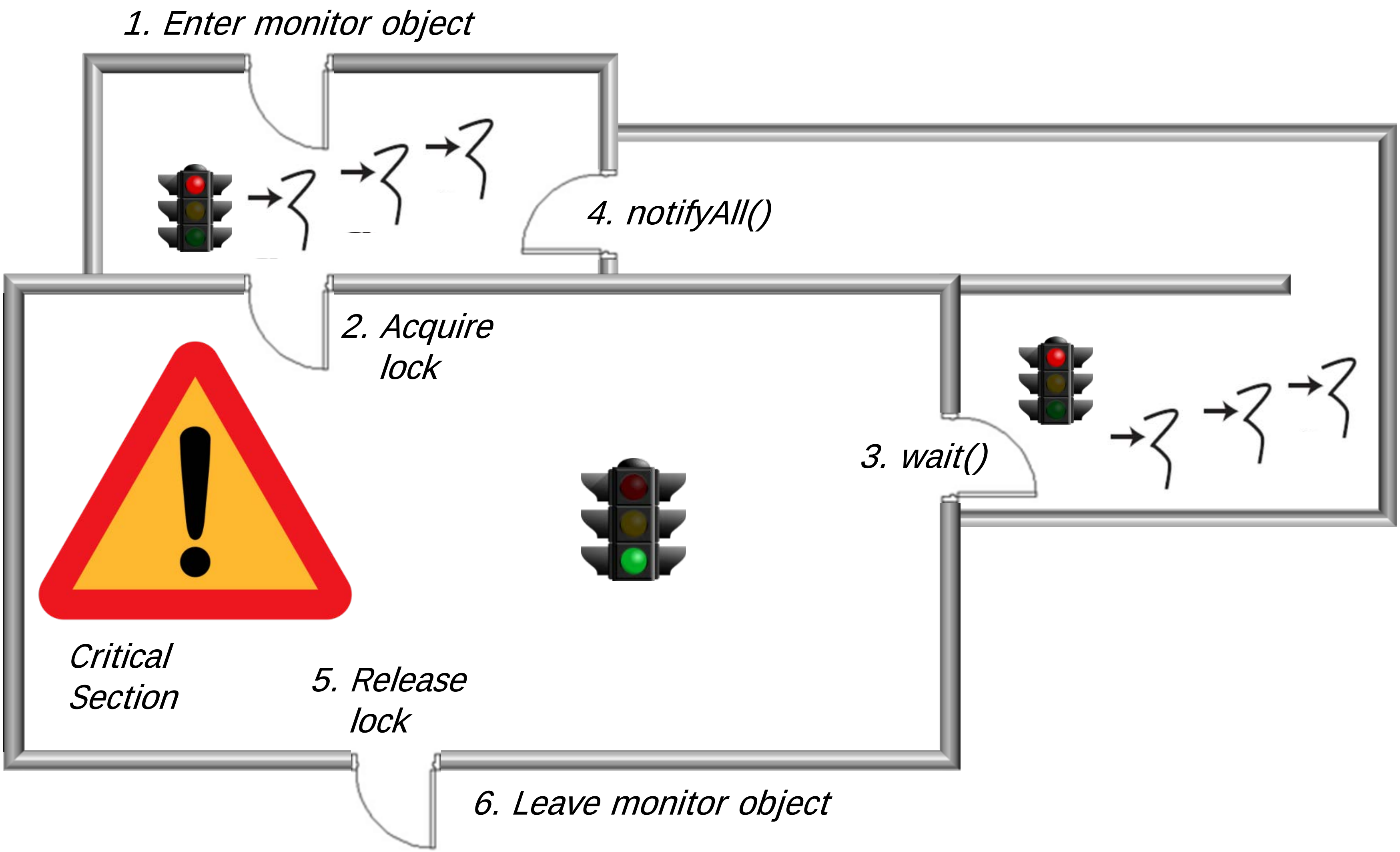
Java Concurrency : Built-in Monitor Objects (Part 2)

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 be used to ensure mutual exclusion & coordination between threads running in a concurrent program



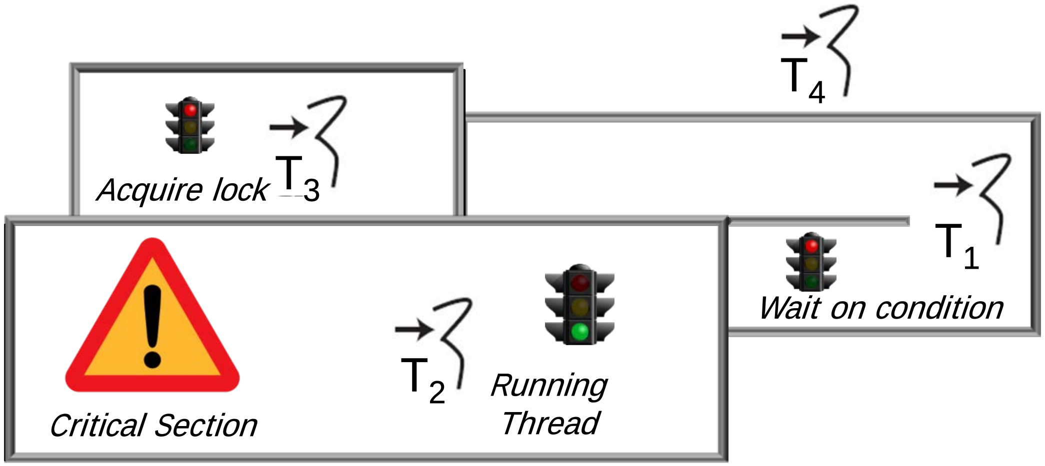
Java Built-in Waiting & Notification Mechanisms (Part 1)

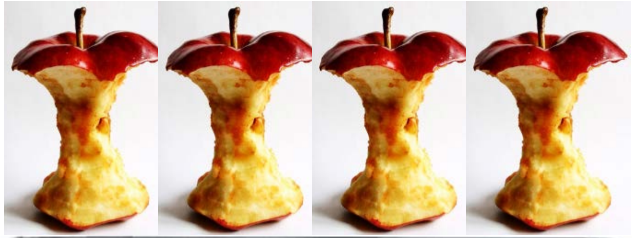
Java Built-in Waiting & Notification Mechanisms

Java synchronized methods & statements only provide a partial solution

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions





Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
  - via the wait(), notify(), & notifyAll() methods

void	wait() – Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
void	notify() – Wakes up a single thread that is waiting on this object's monitor
void	notifyAll() – Wakes up all threads that are waiting on this object's monitor

See docs.oracle.com/javase/7/docs/api/java/lang/Object.html

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
  - via the wait(), notify(), & notifyAll() methods

void	wait() – Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
void	notify() – Wakes up a single thread that is waiting on this object's monitor
void	notifyAll() – Wakes up all threads that are waiting on this object's monitor

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
  - via the wait(), notify(), & notifyAll() methods

void	wait() – Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
void	notify() – Wakes up a single thread that is waiting on this object's monitor
void	notifyAll() – Wakes up all threads that are waiting on this object's monitor

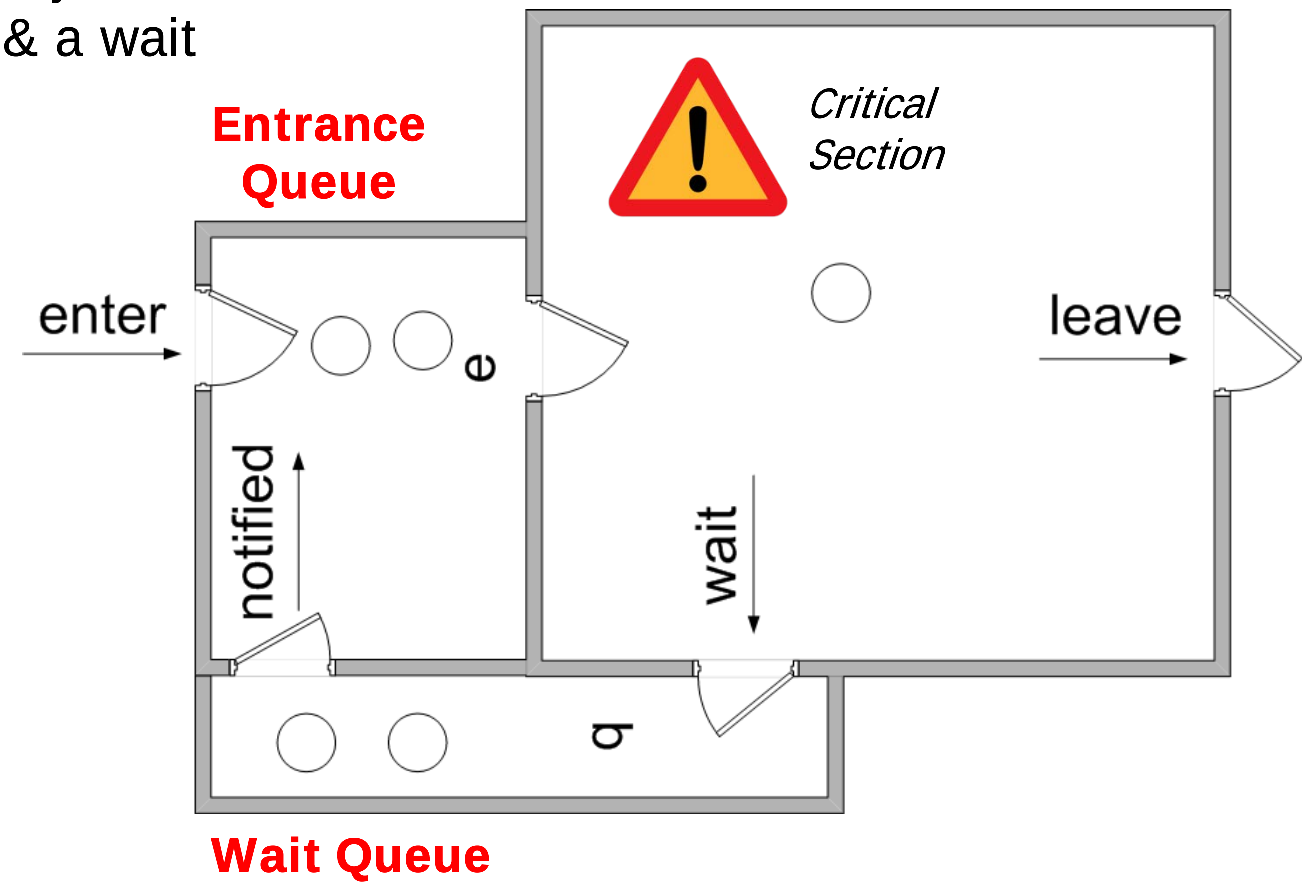
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
  - via the wait(), notify(), & notifyAll() methods

void	wait() – Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object
void	notify() – Wakes up a single thread that is waiting on this object's monitor
void	notifyAll() – Wakes up all threads that are waiting on this object's monitor

Java Built-in Waiting & Notification Mechanisms

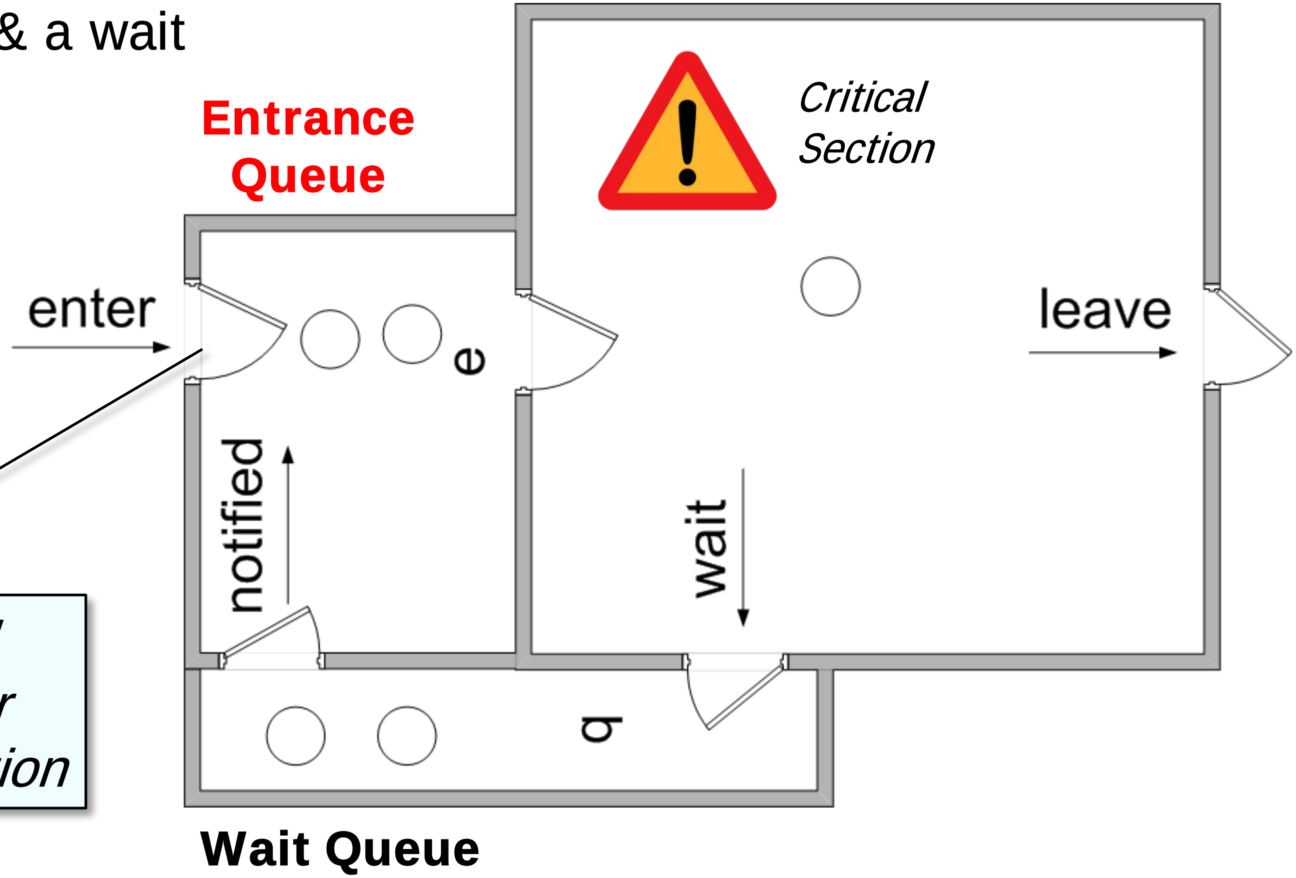
- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue



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

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue

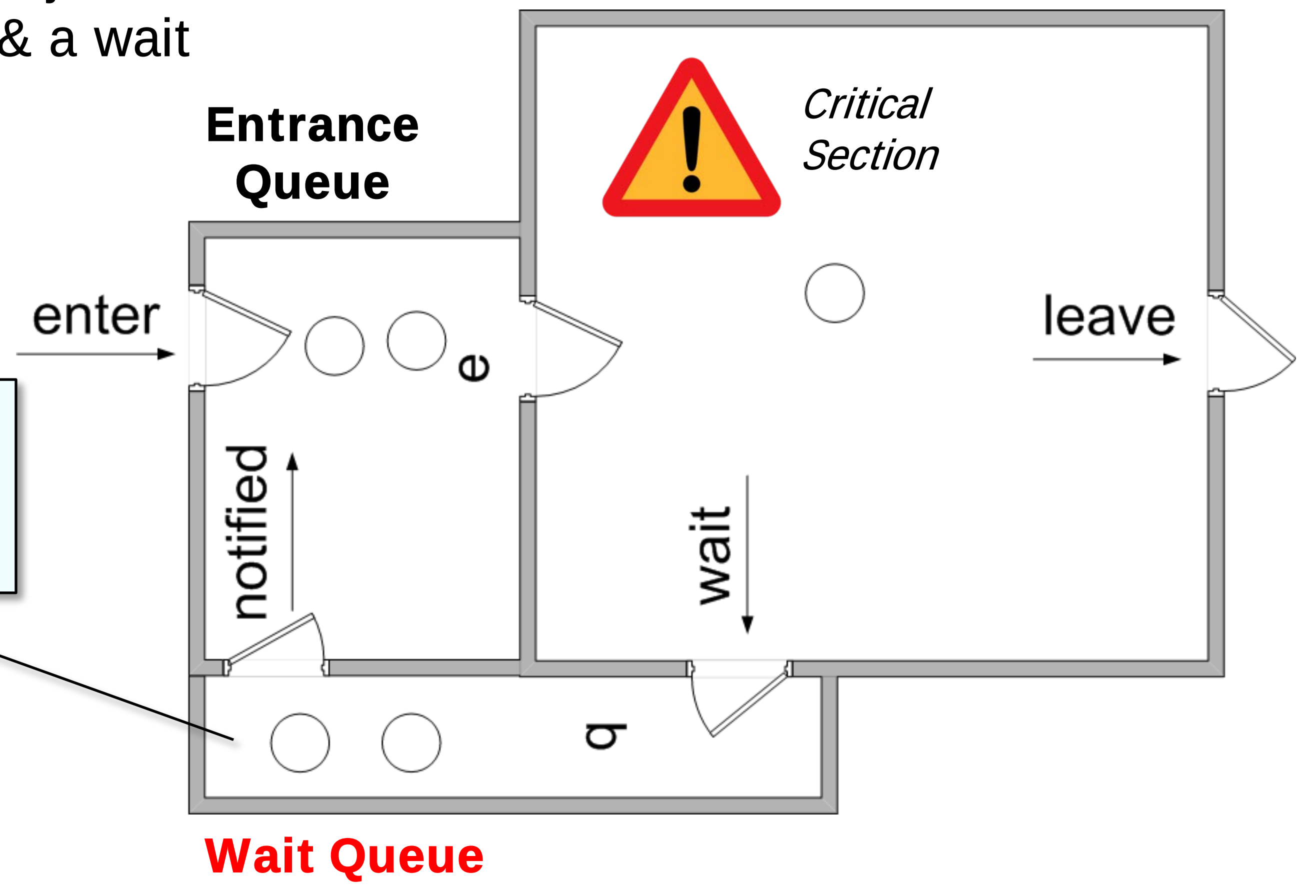


Serializes thread access to monitor object's critical section

Java Built-in Waiting & Notification Mechanisms

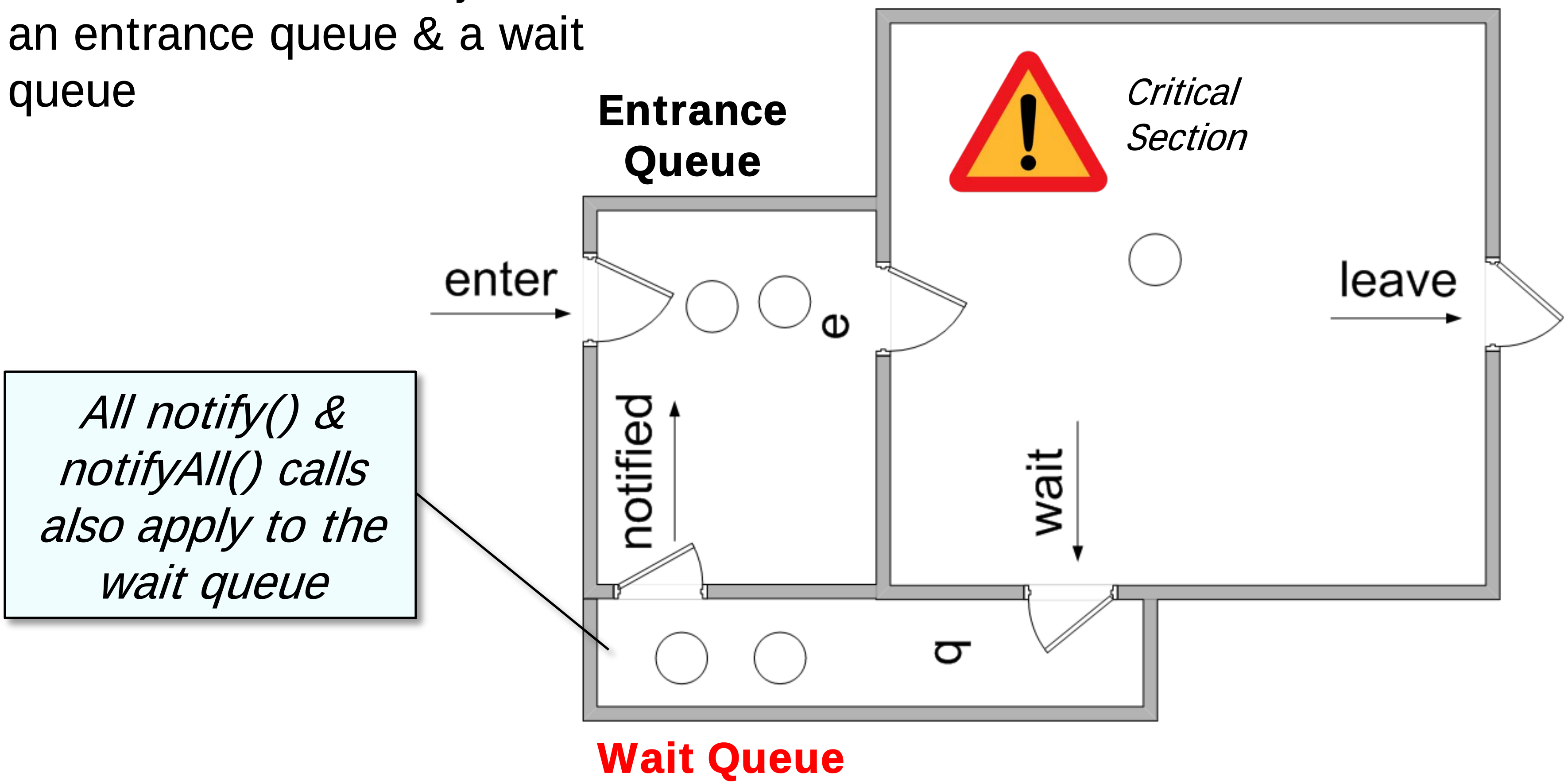
- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue

All calls to wait() are done on the wait queue



Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue



Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue, e.g.
  - put() calls wait() when the queue is full

```
class SimpleBlockingQueue<E>
      implements BlockingQueue<E> {
...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue, e.g.
  - put() calls wait() when the queue is full

```
class SimpleBlockingQueue<E>
      implements BlockingQueue<E> {
...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Atomically releases the intrinsic lock & sleeps on the wait queue

See en.wikipedia.org/wiki/Guarded_suspension

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue, e.g.
  - put() calls wait() when the queue is full
  - It also calls notifyAll() after adding an item

```
class SimpleBlockingQueue<E>
      implements BlockingQueue<E> {
...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Wakes up all the Threads blocked on the wait queue

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue

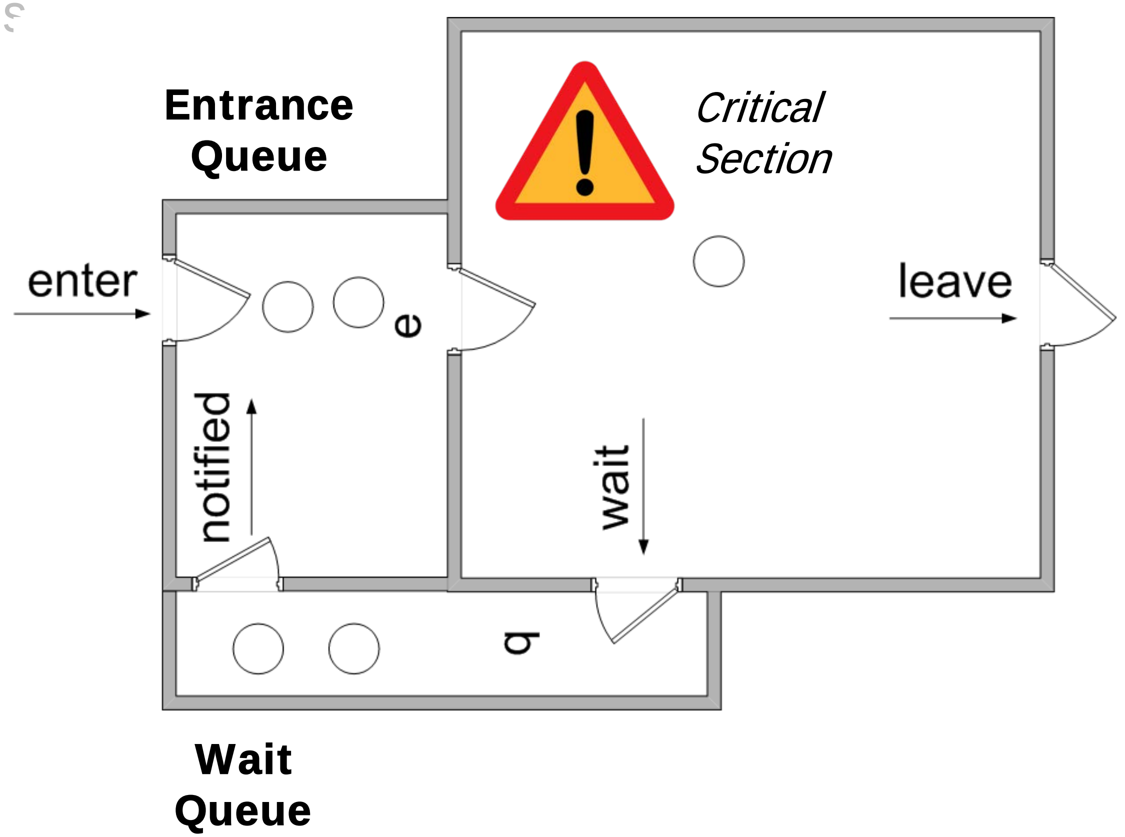
```
class SimpleBlockingQueue<E>
      implements BlockingQueue<E> {
...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

The put() & take() method are examined later in this video

Java Built-in Waiting & Notification Mechanisms (Part 2)

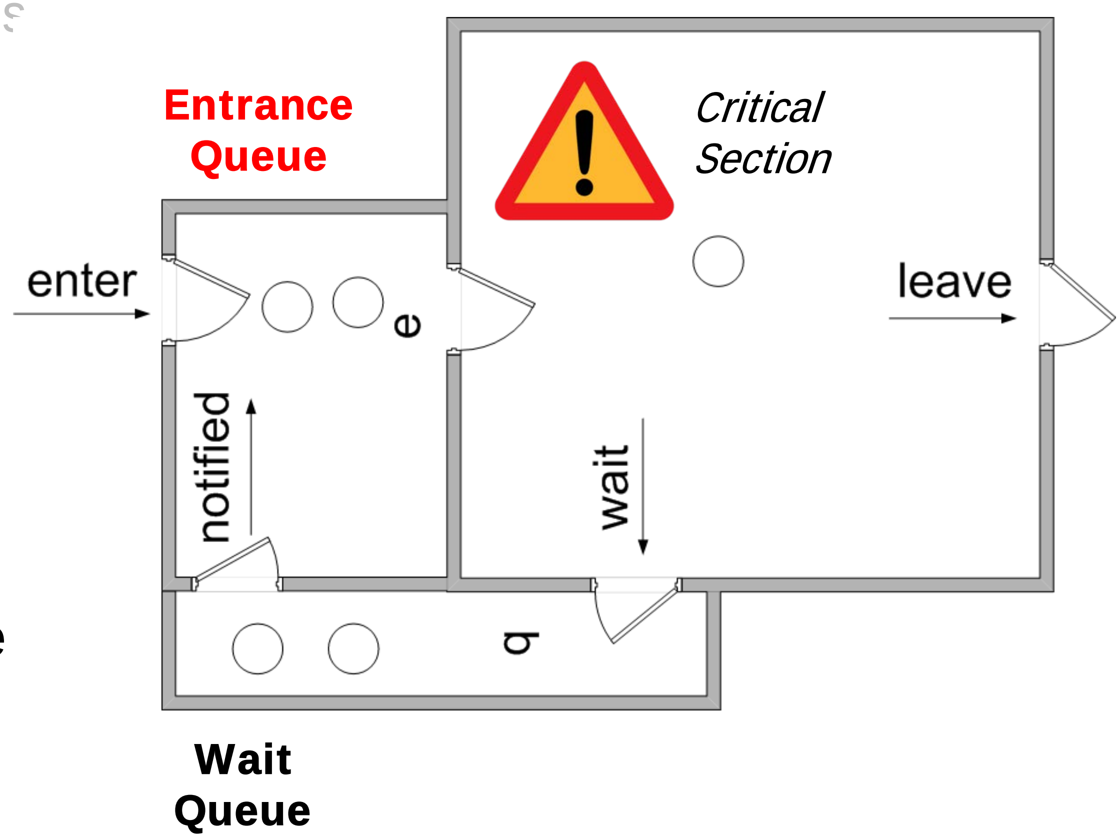
Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue
- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms



Java Built-in Waiting & Notification Mechanisms

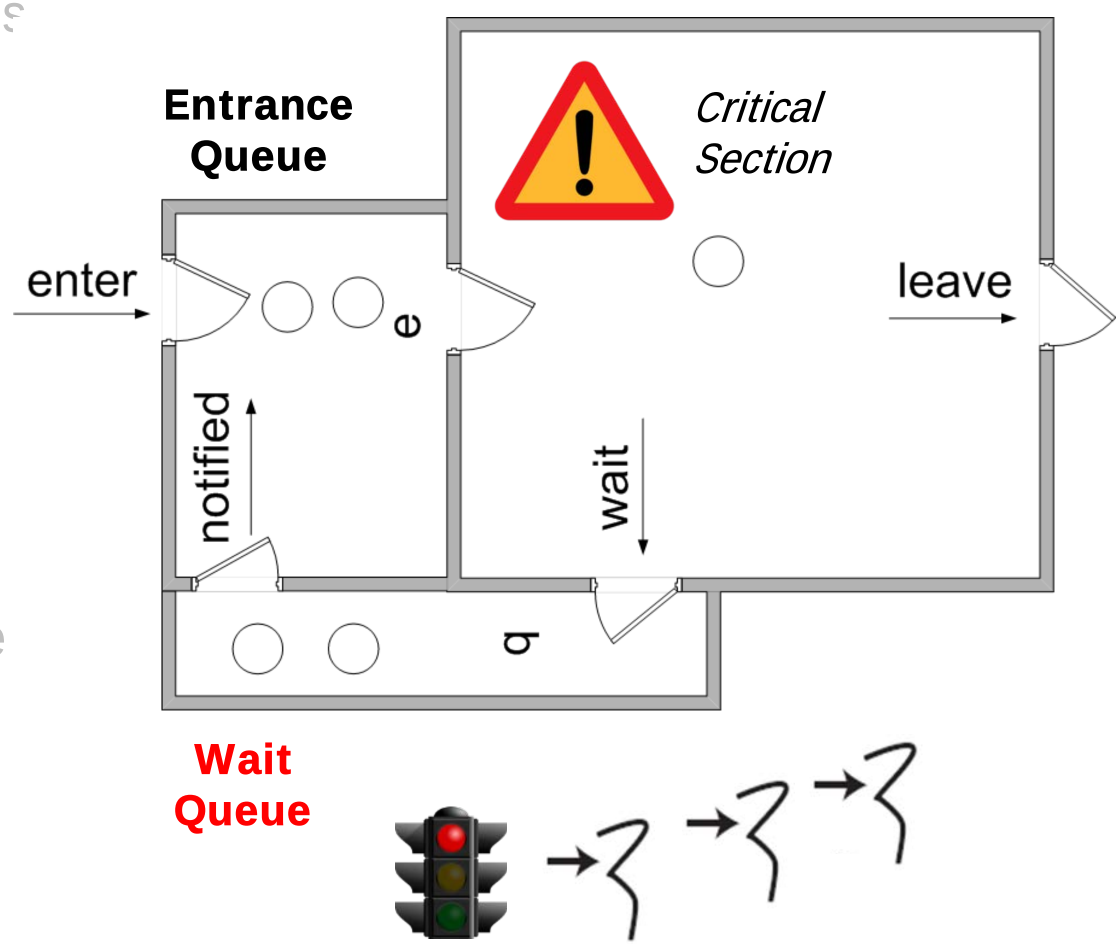
- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue
- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
  - Entrance queue can be a POSIX mutex with recursive locking semantics



See computing.llnl.gov/tutorials/pthreads/#Mutexes

Java Built-in Waiting & Notification Mechanisms

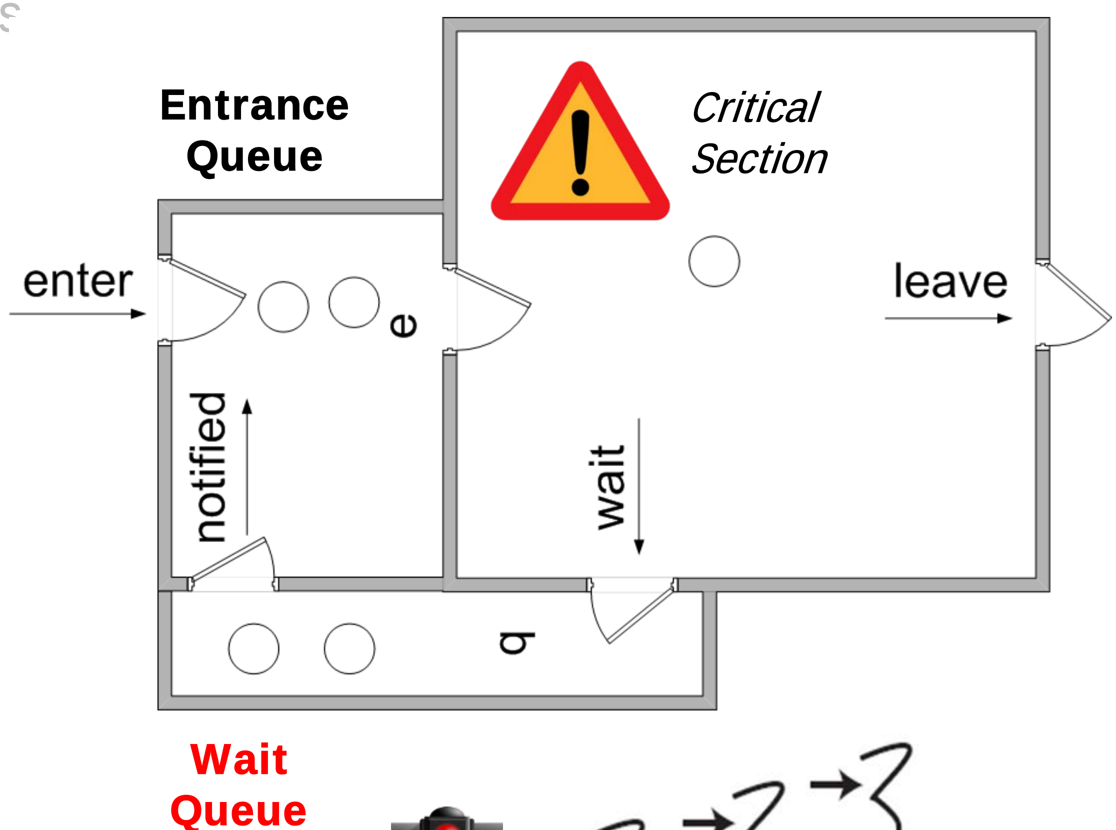
- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue
- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
  - Entrance queue can be a POSIX mutex with recursive locking semantics
  - Wait queue can be a POSIX condition variable

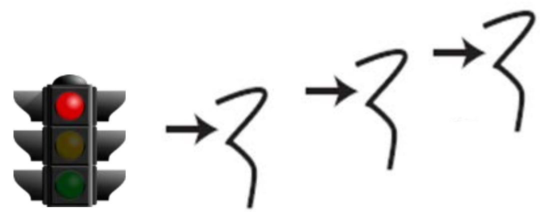


See computing.llnl.gov/tutorials/pthreads/#ConditionVariables

Java Built-in Waiting & Notification Mechanisms

- Java monitor objects provide mechanisms that Threads use to coordinate their interactions
- Each Java monitor object has an entrance queue & a wait queue
- Java built-in monitor object synchronizers are often implemented via POSIX mechanisms, e.g.
  - Entrance queue can be a POSIX mutex with recursive locking semantics
  - Wait queue can be a POSIX condition variable
    - Similar to Java ConditionObjects

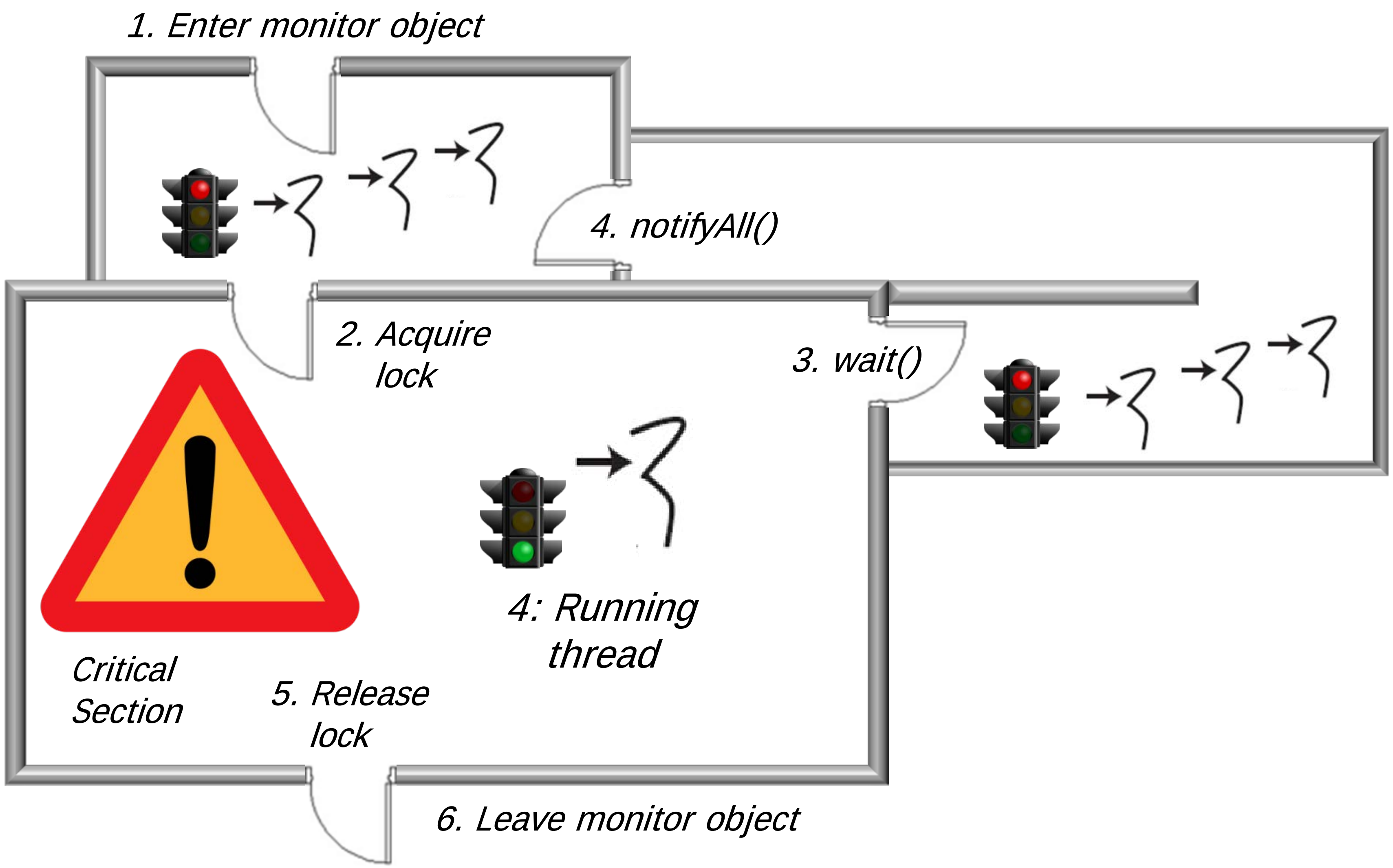




See upcoming part on "Java ConditionObjects"

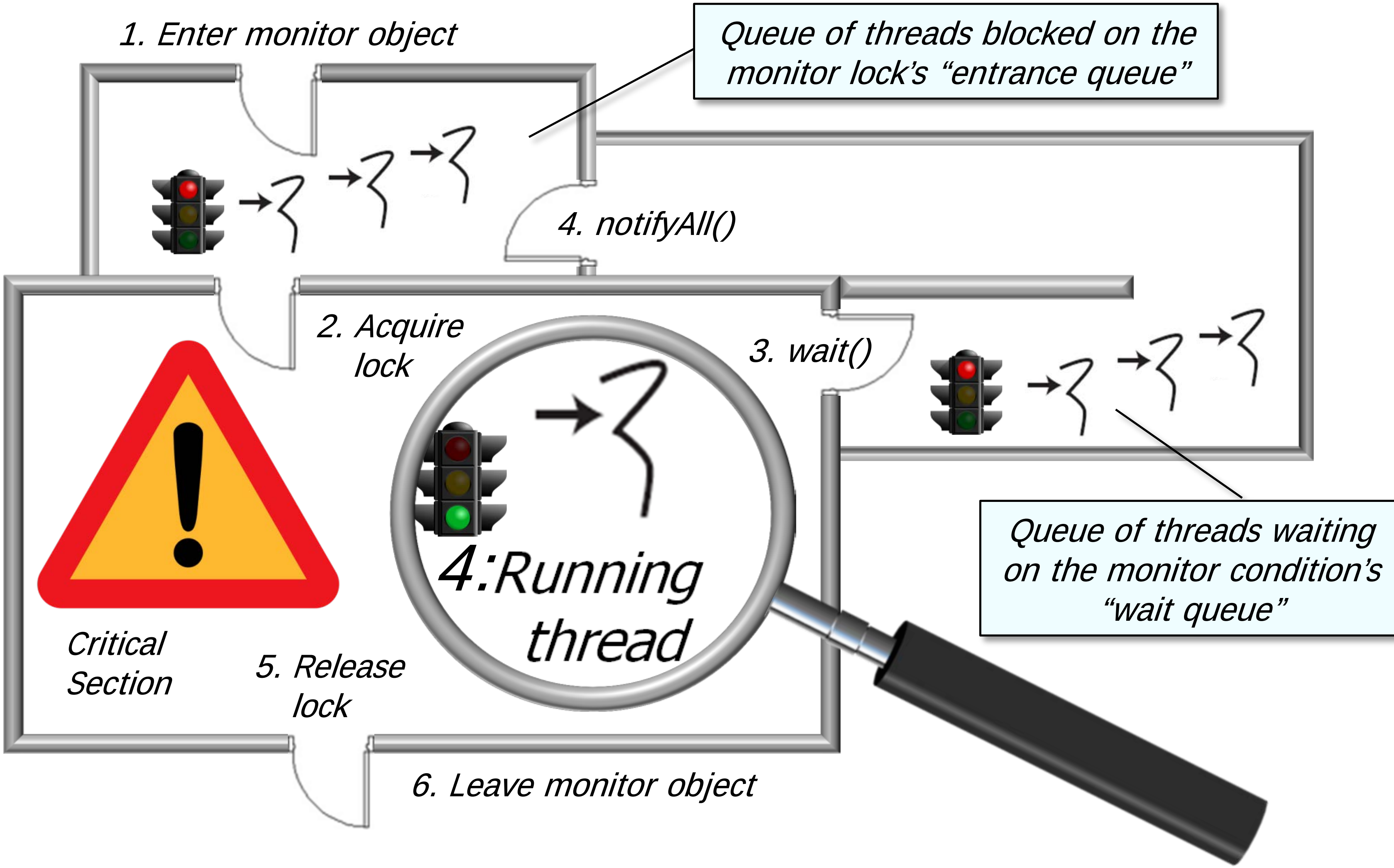
Visual Analysis of the SimpleBlocking Queue Example

Visual Analysis of SimpleBlockingQueue



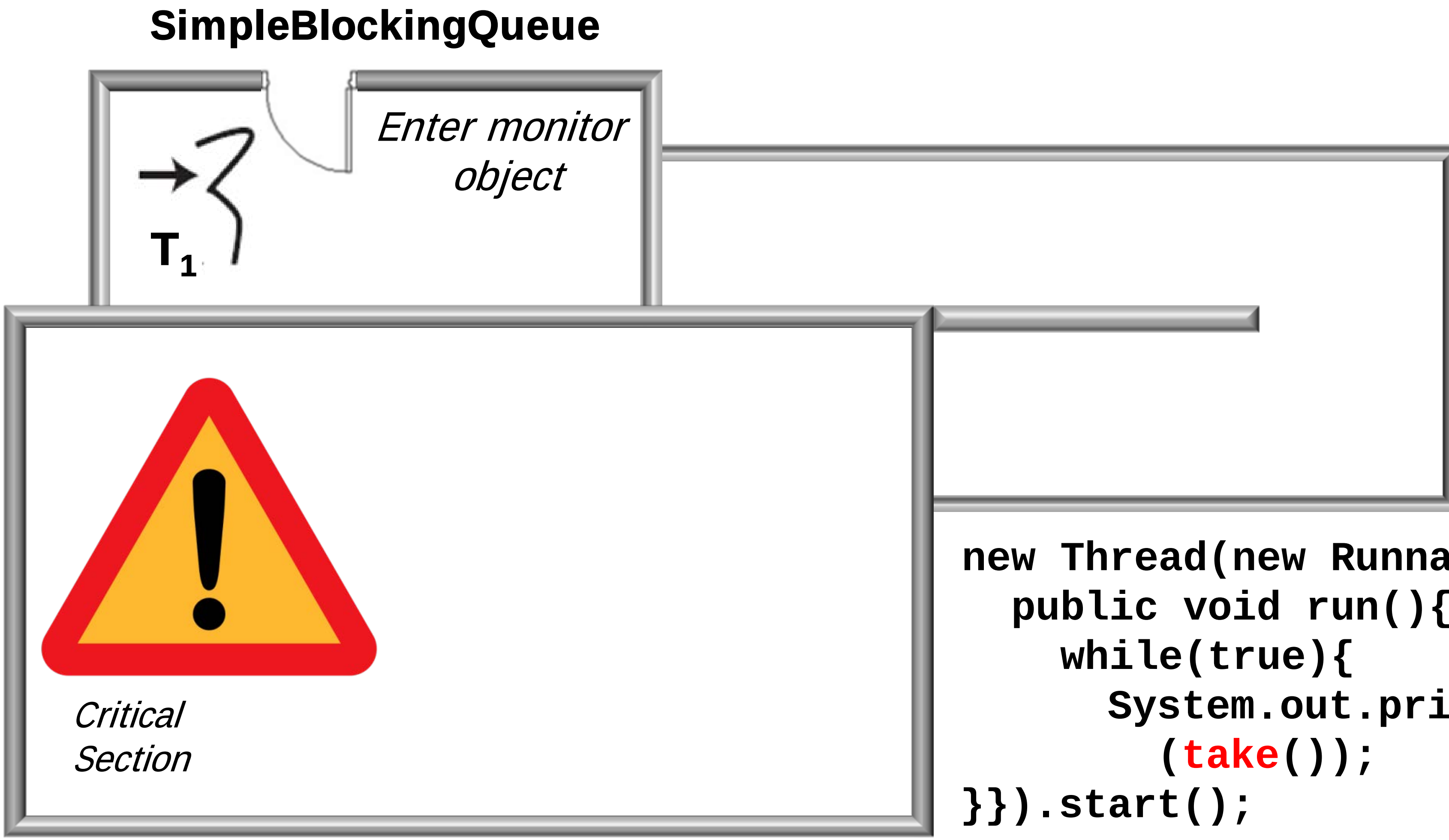
See github.com/douglascraigschmidt/LiveLessons/tree/master/SimpleBlockingQueue

Visual Analysis of SimpleBlockingQueue



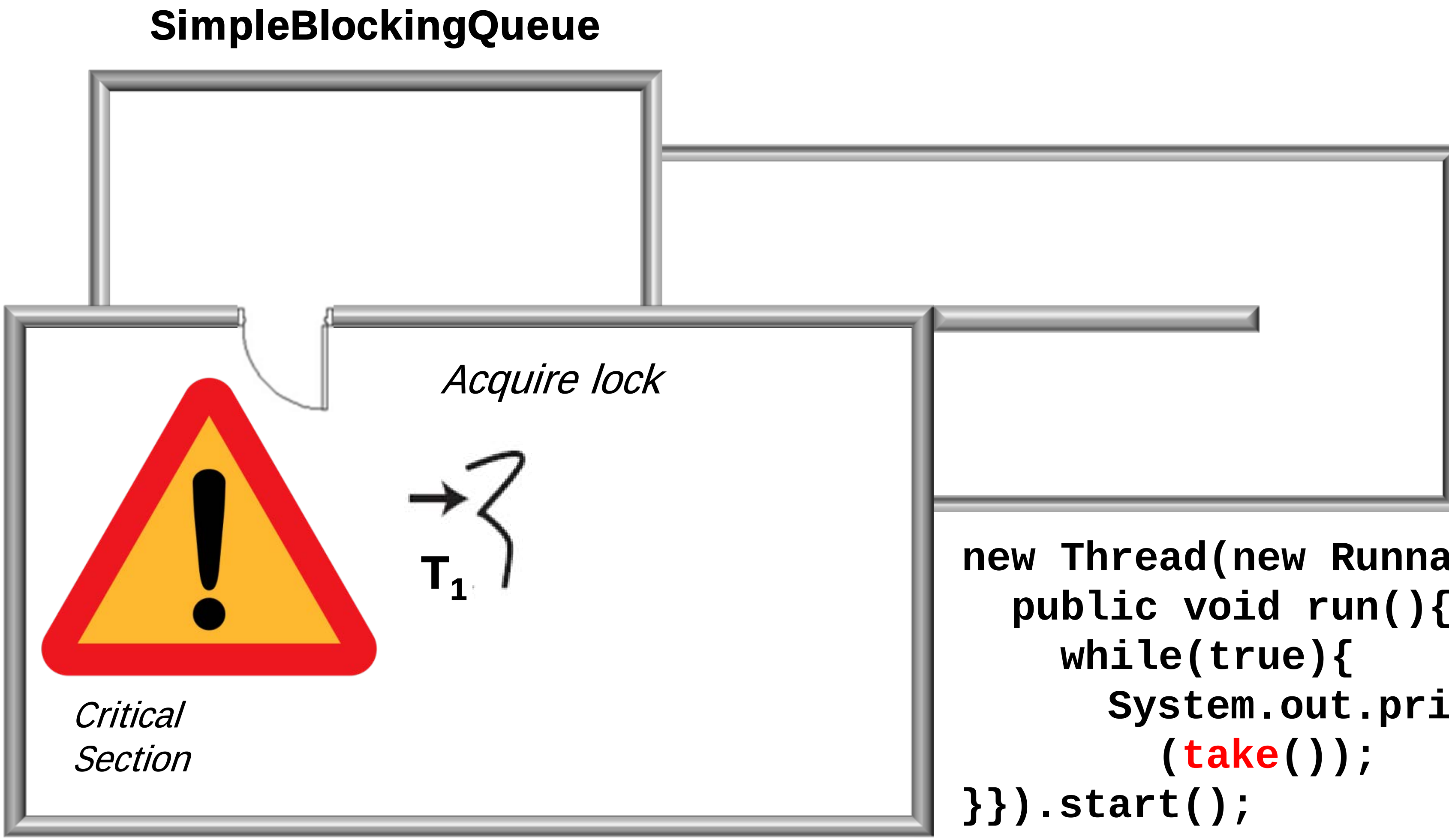
See en.wikipedia.org/wiki/Monitor_(synchronization)#Implicit_condition_variable_monitors

Visual Analysis of SimpleBlockingQueue




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

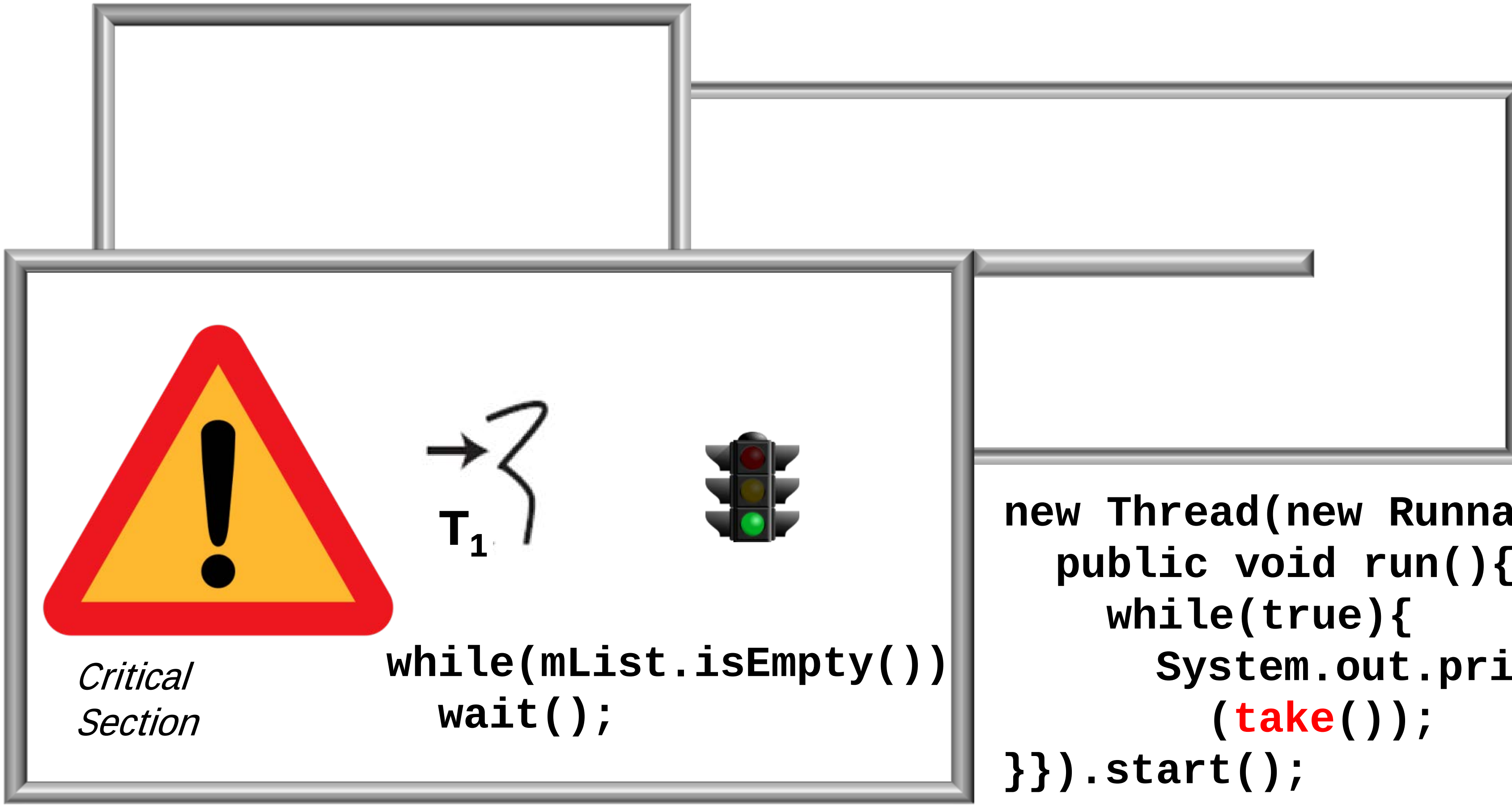
Visual Analysis of SimpleBlockingQueue




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

Visual Analysis of SimpleBlockingQueue

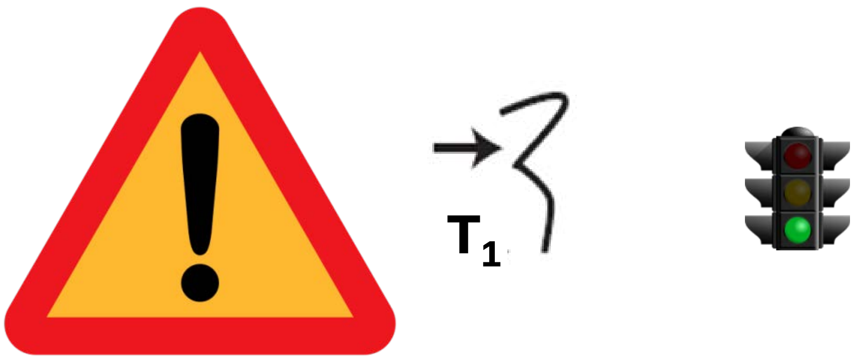
SimpleBlockingQueue




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

Visual Analysis of SimpleBlockingQueue

SimpleBlockingQueue



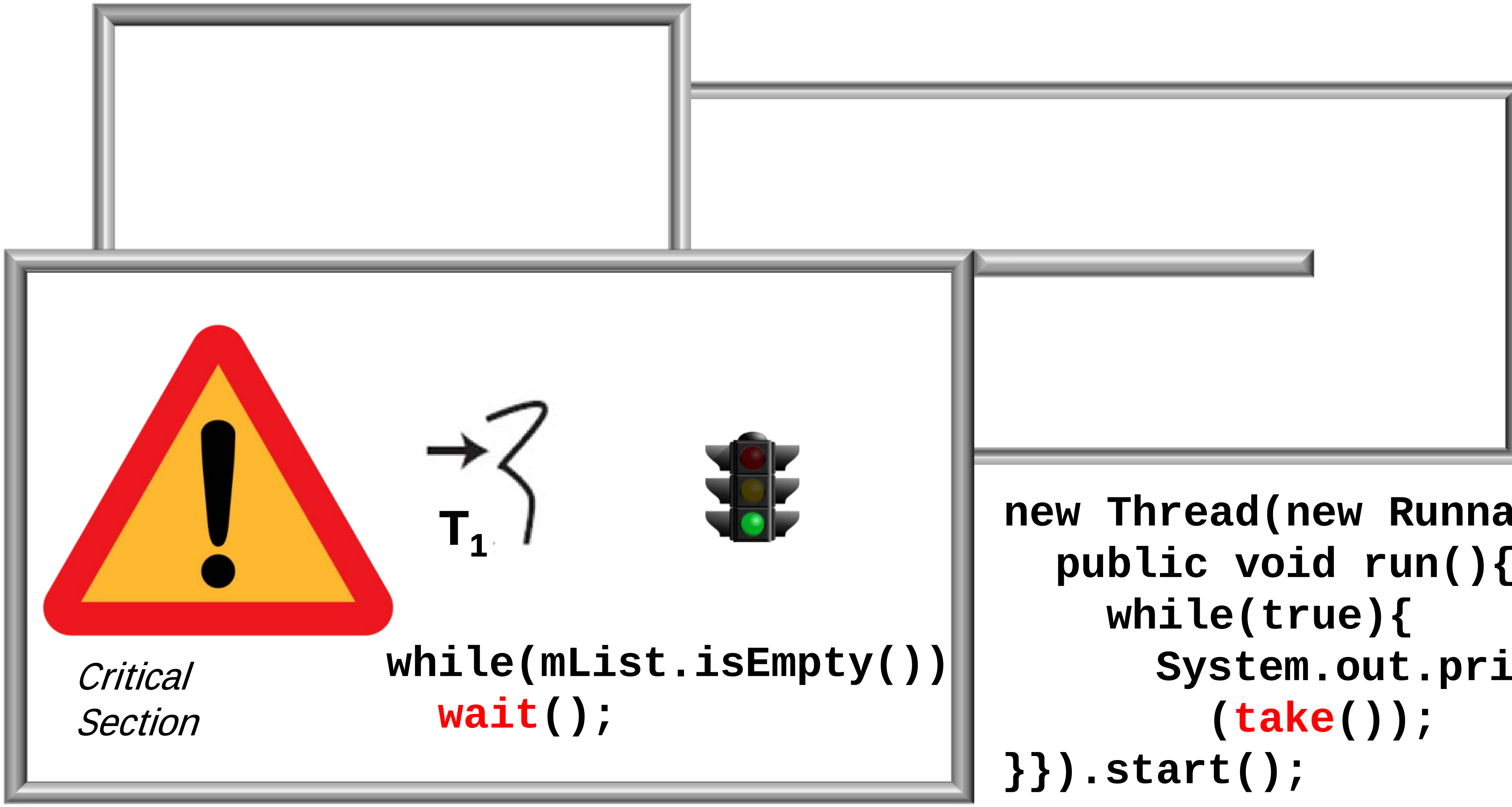
Critical Section

```
while(mList.isEmpty())
  wait();
```

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

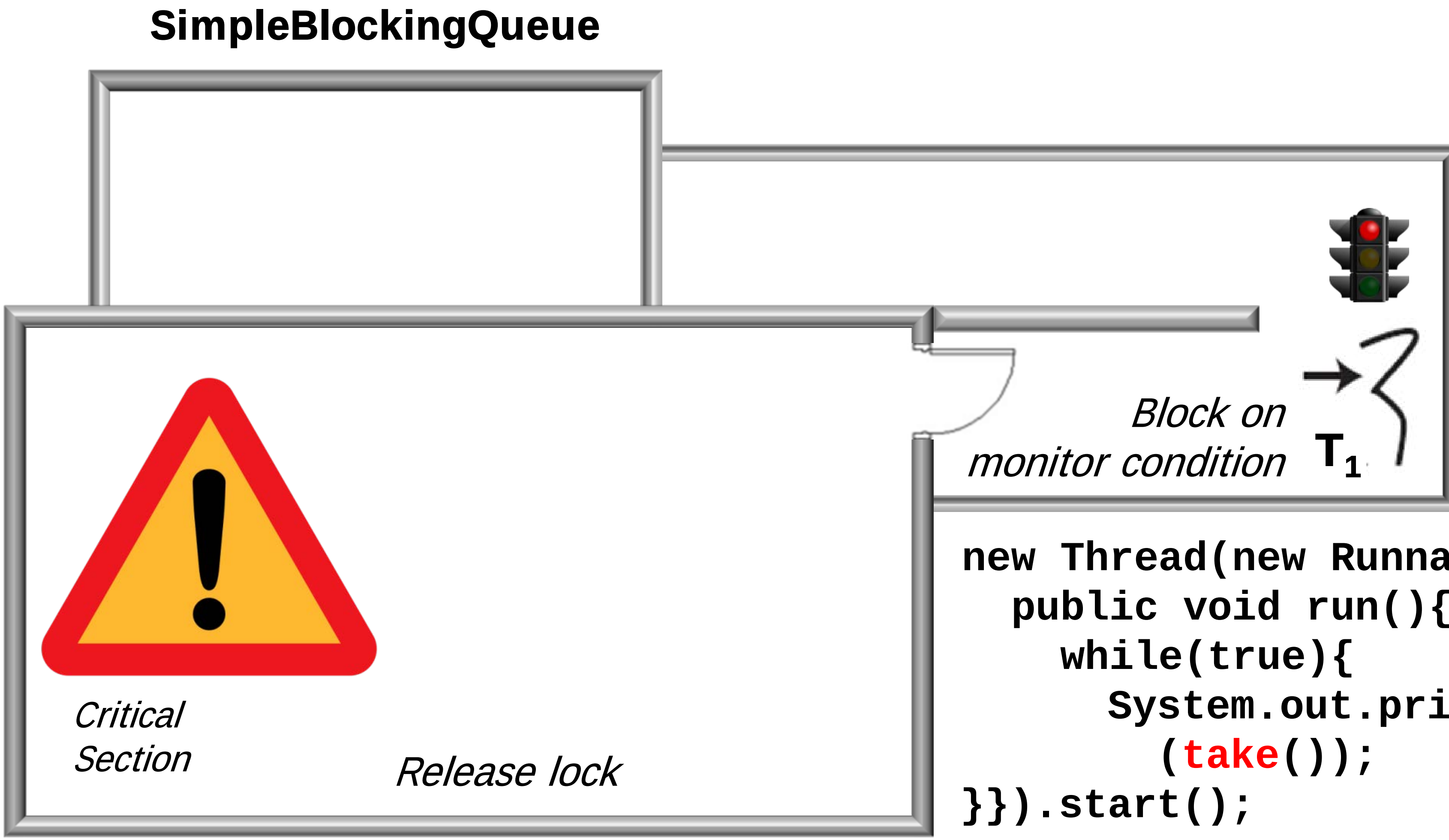
Visual Analysis of SimpleBlockingQueue

SimpleBlockingQueue



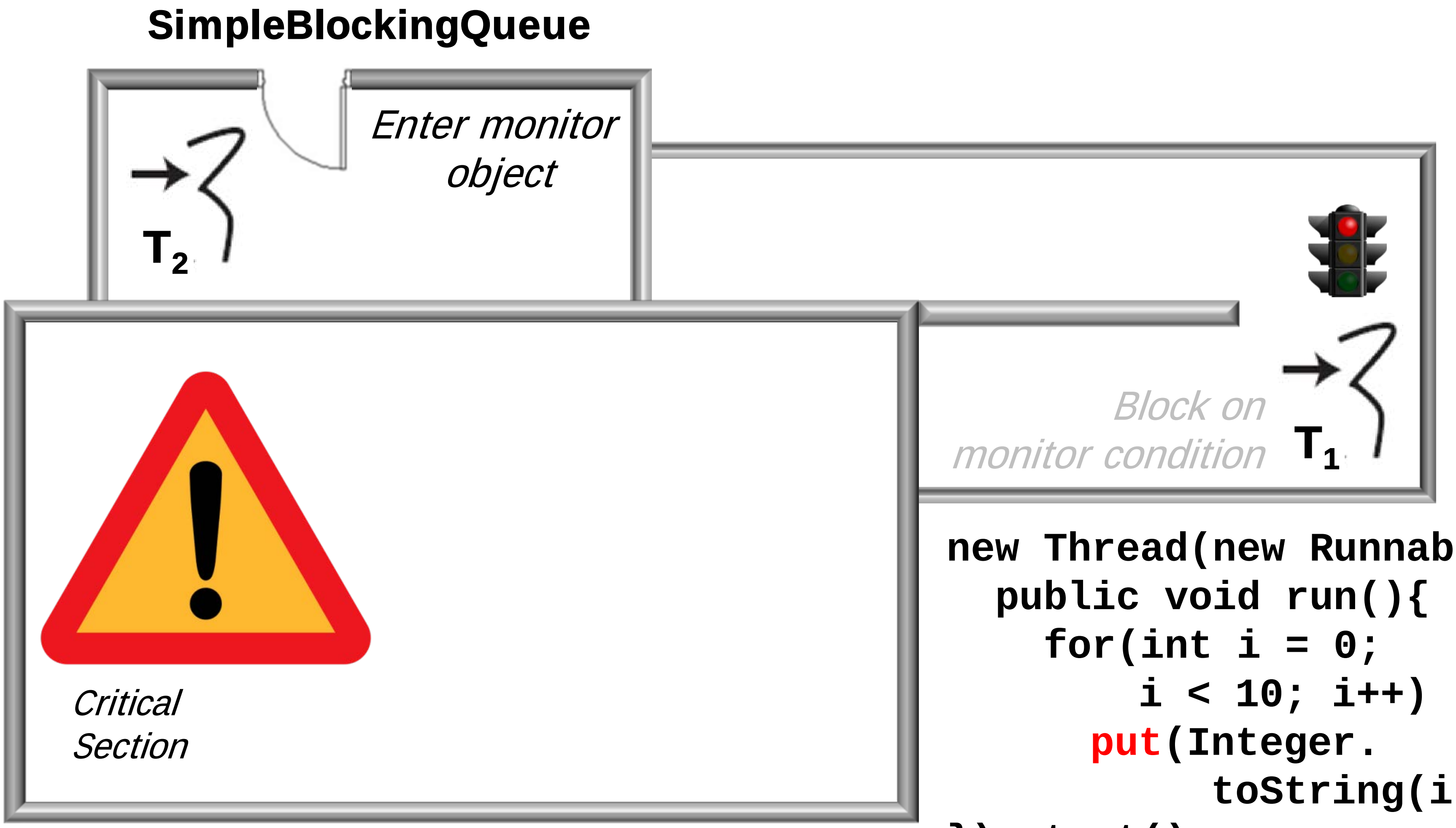

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of SimpleBlockingQueue



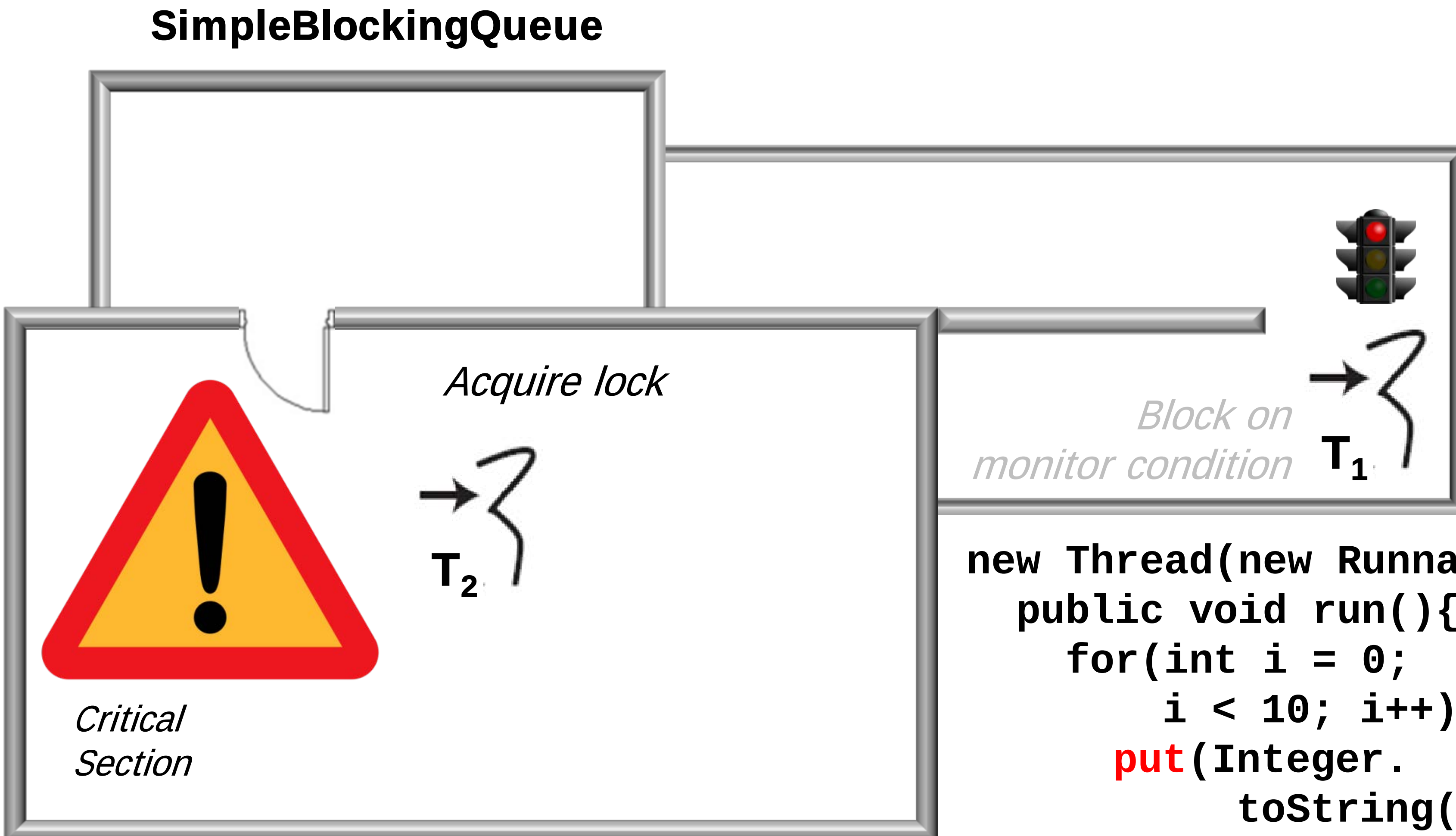

```
new Thread(new Runnable(){
  public void run(){
    while(true){
      System.out.println
        (take());
}}).start();
```

Visual Analysis of SimpleBlockingQueue



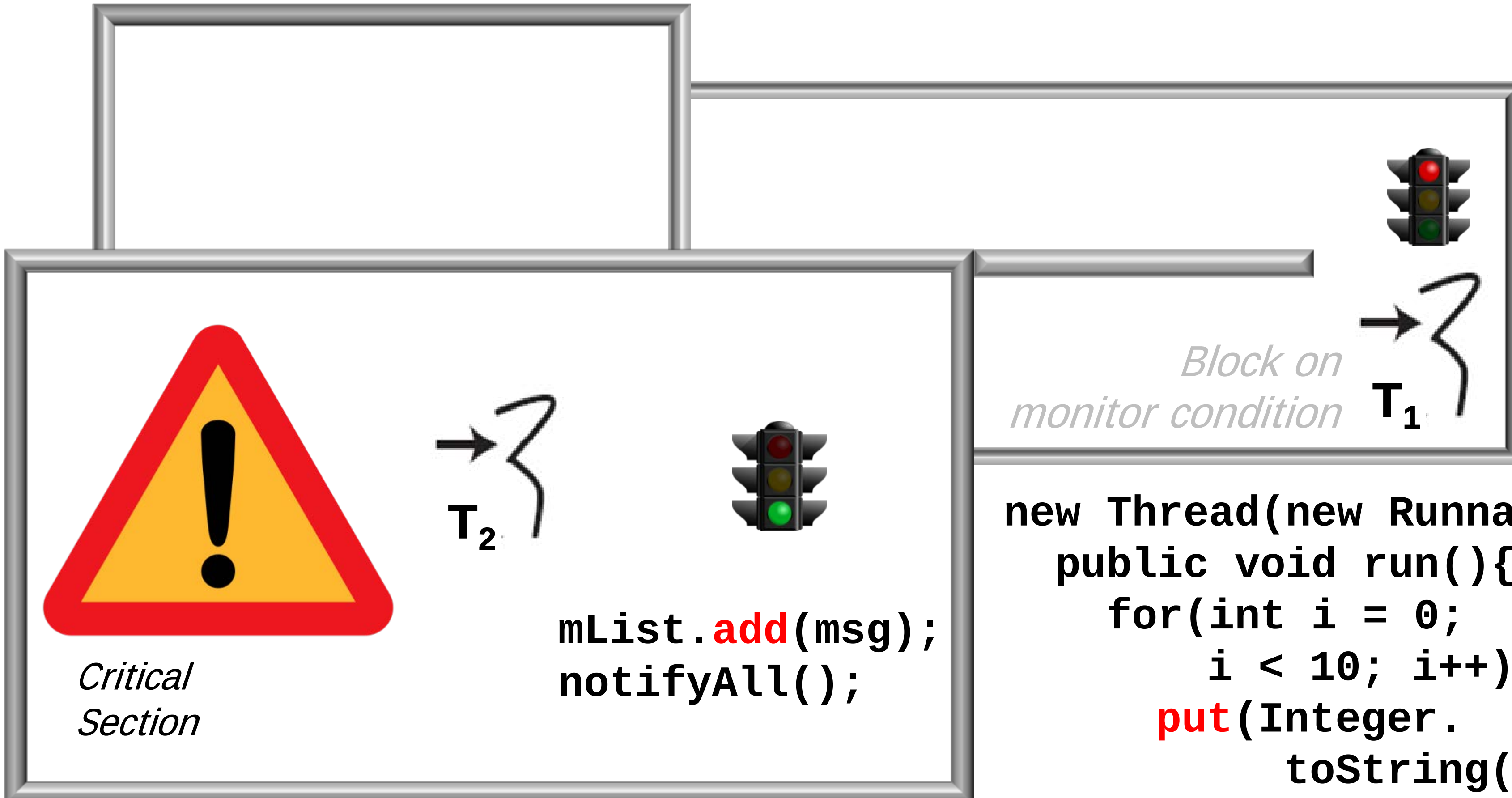

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of SimpleBlockingQueue




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

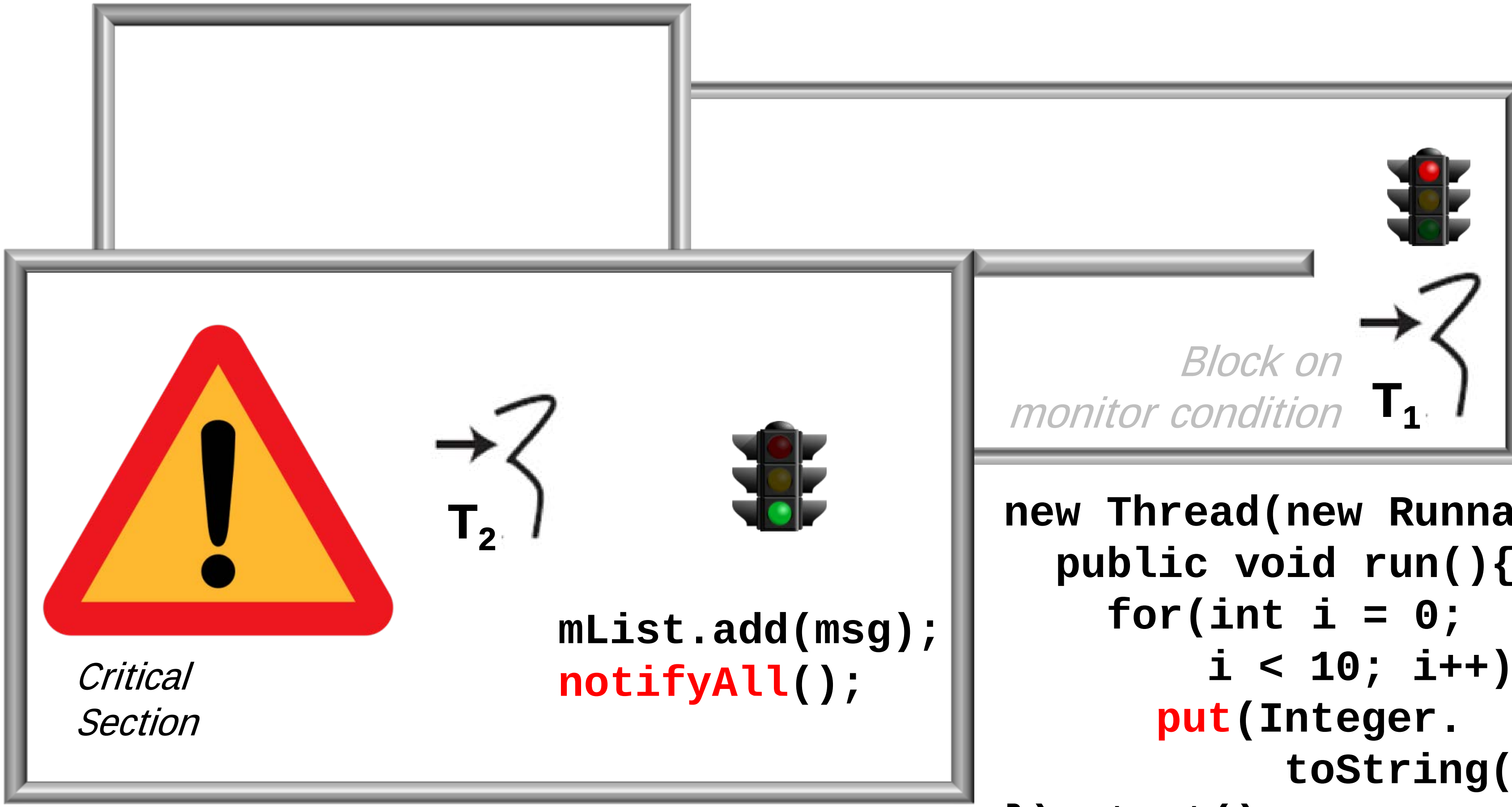
SimpleBlockingQueue




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

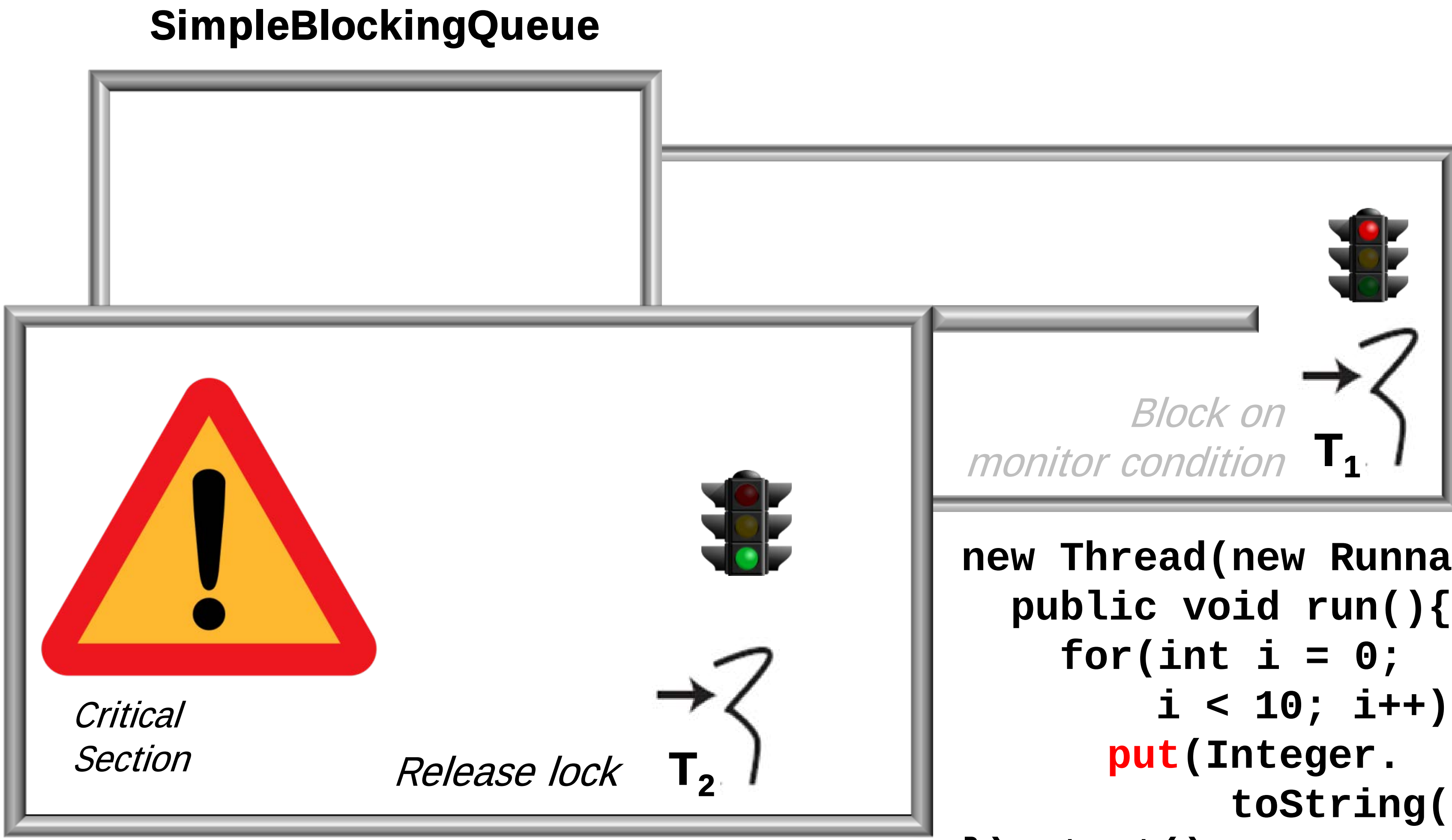
Visual Analysis of SimpleBlockingQueue

SimpleBlockingQueue



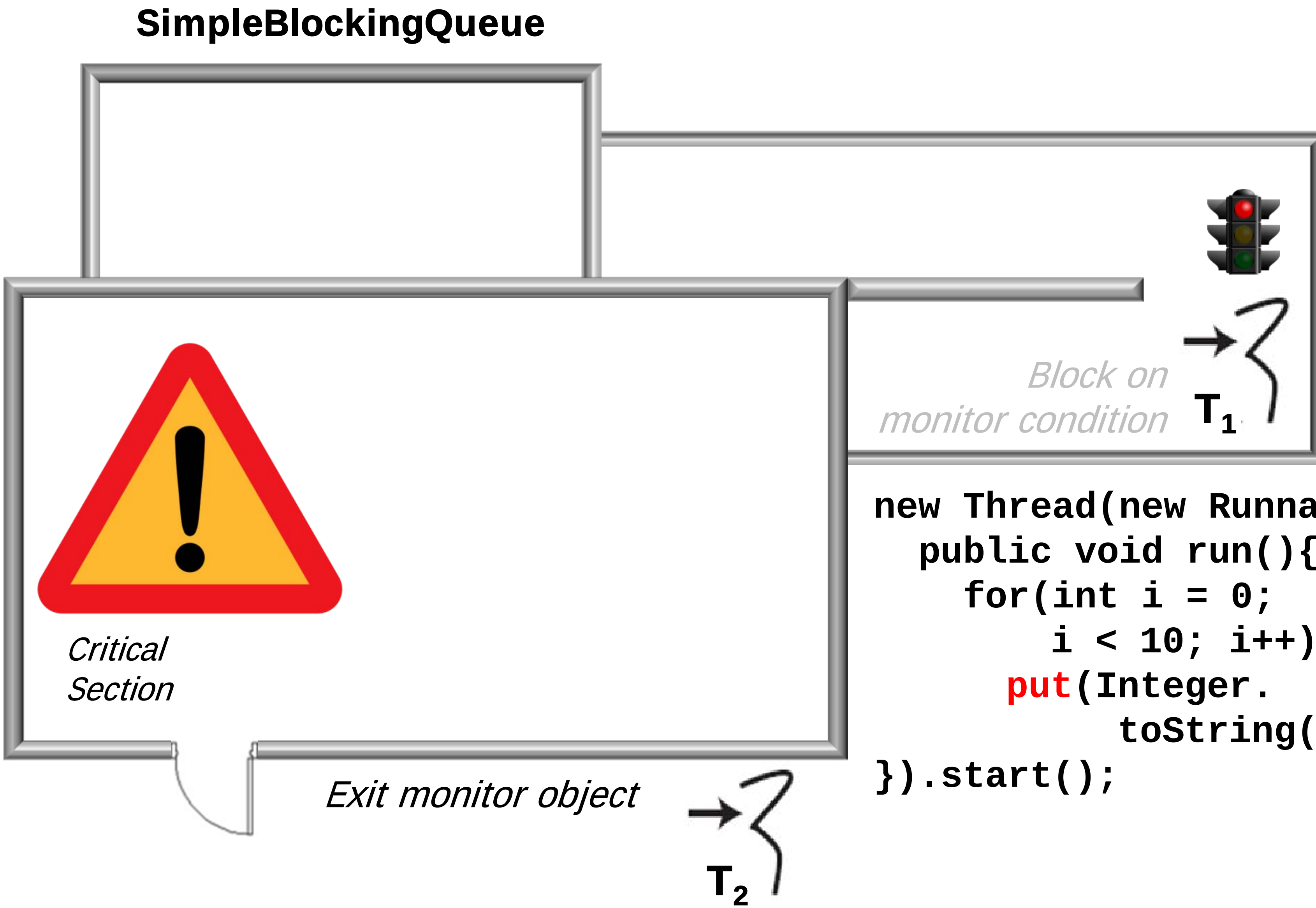

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
            toString(i));
}).start();
```

Visual Analysis of SimpleBlockingQueue



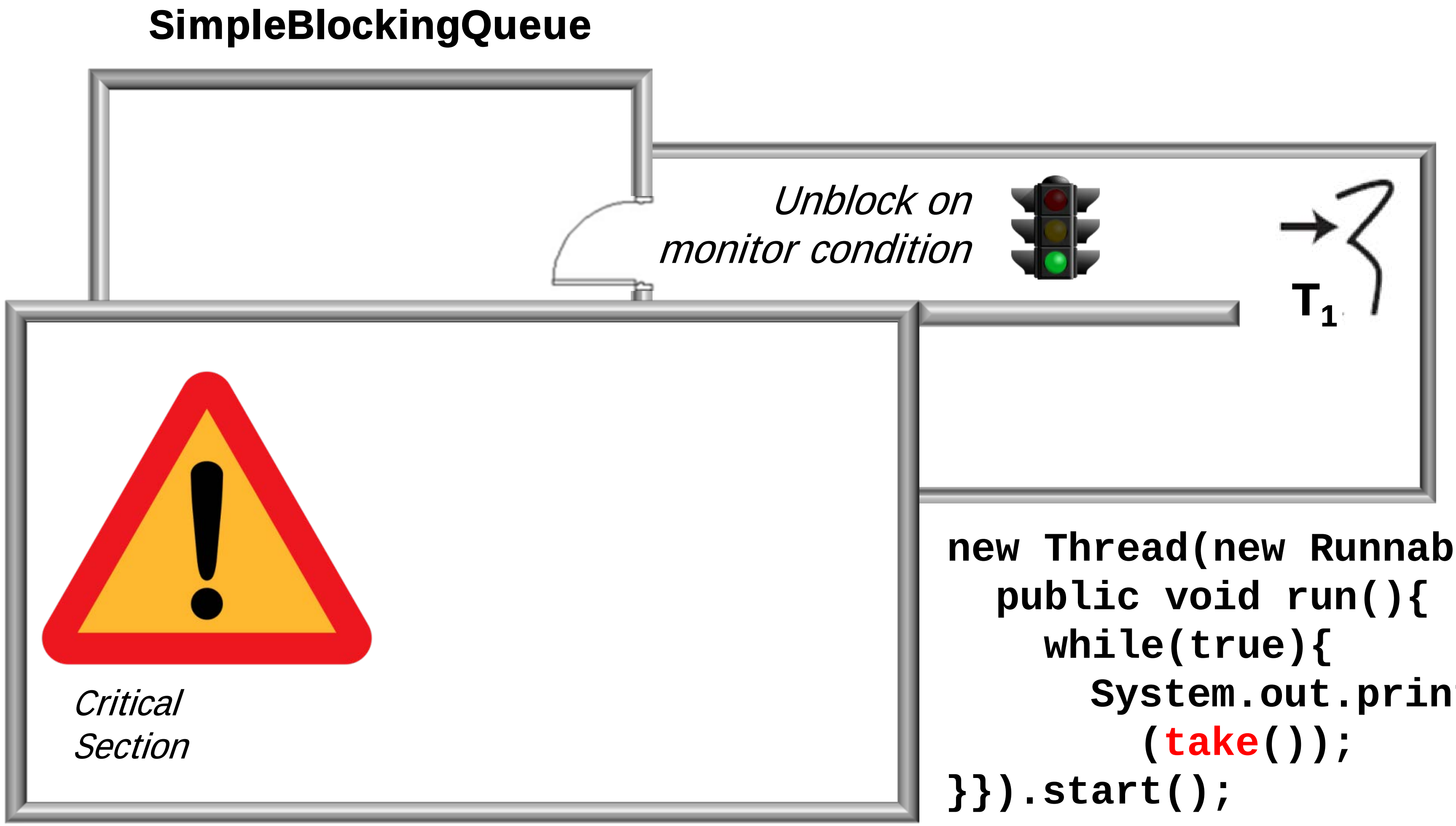

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
          toString(i));
}).start();
```

Visual Analysis of SimpleBlockingQueue



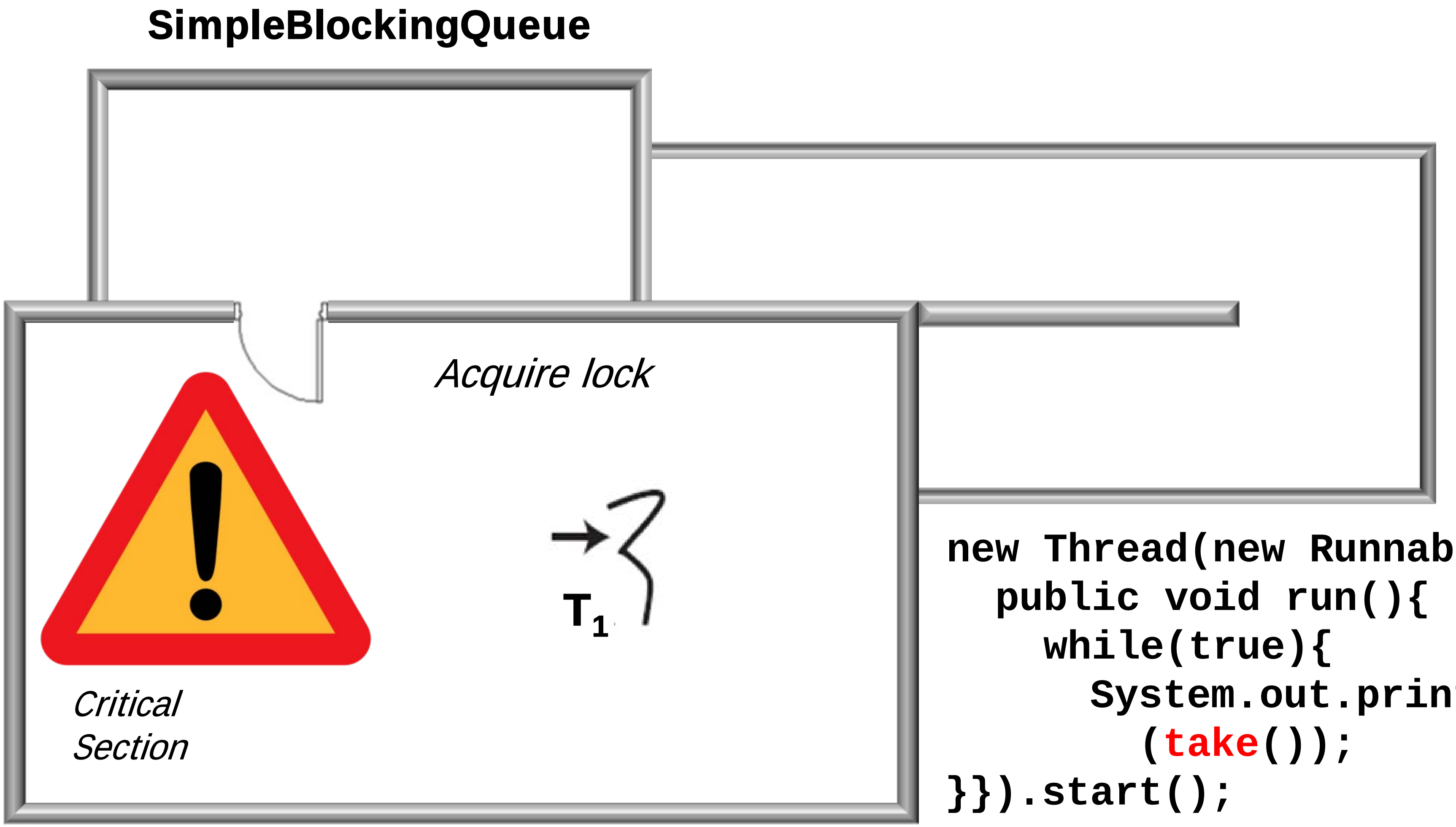

```
new Thread(new Runnable(){
  public void run(){
    for(int i = 0;
        i < 10; i++)
      put(Integer.
         toString(i));
}).start();
```

Visual Analysis of SimpleBlockingQueue




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

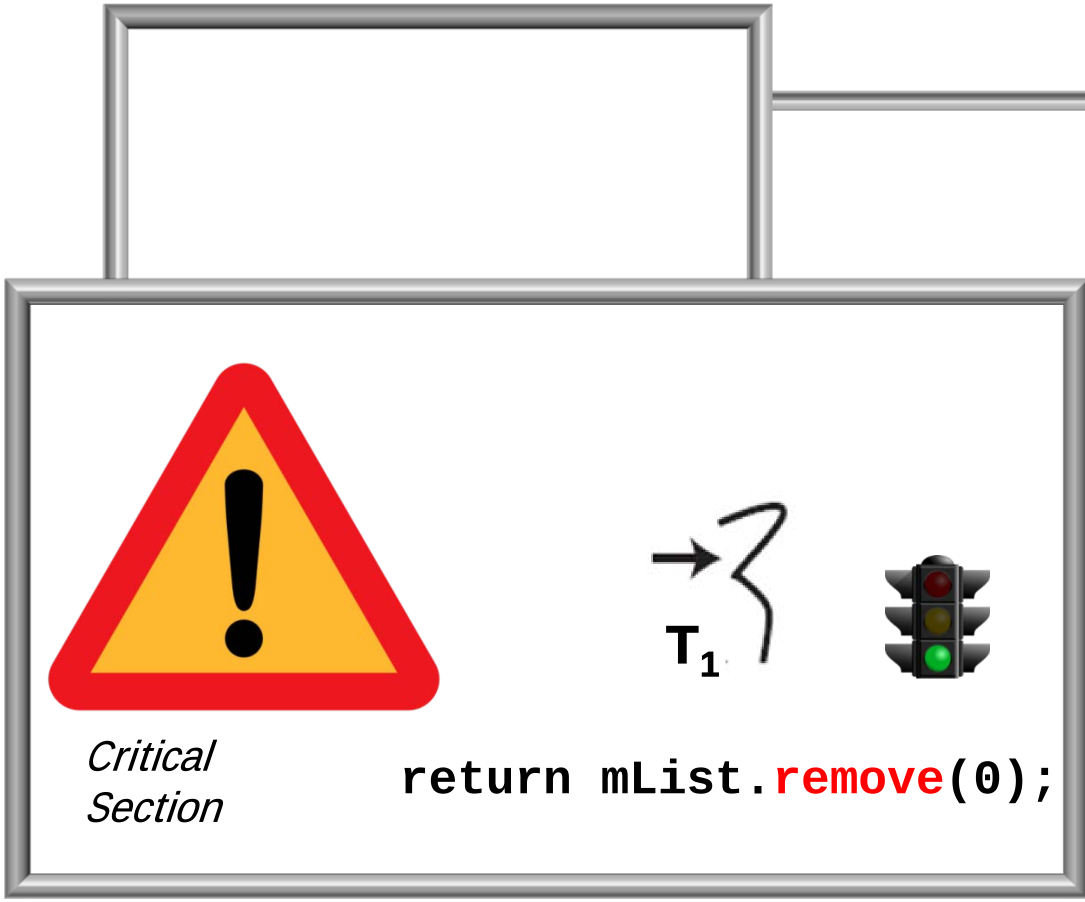
Visual Analysis of SimpleBlockingQueue




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

Visual Analysis of SimpleBlockingQueue

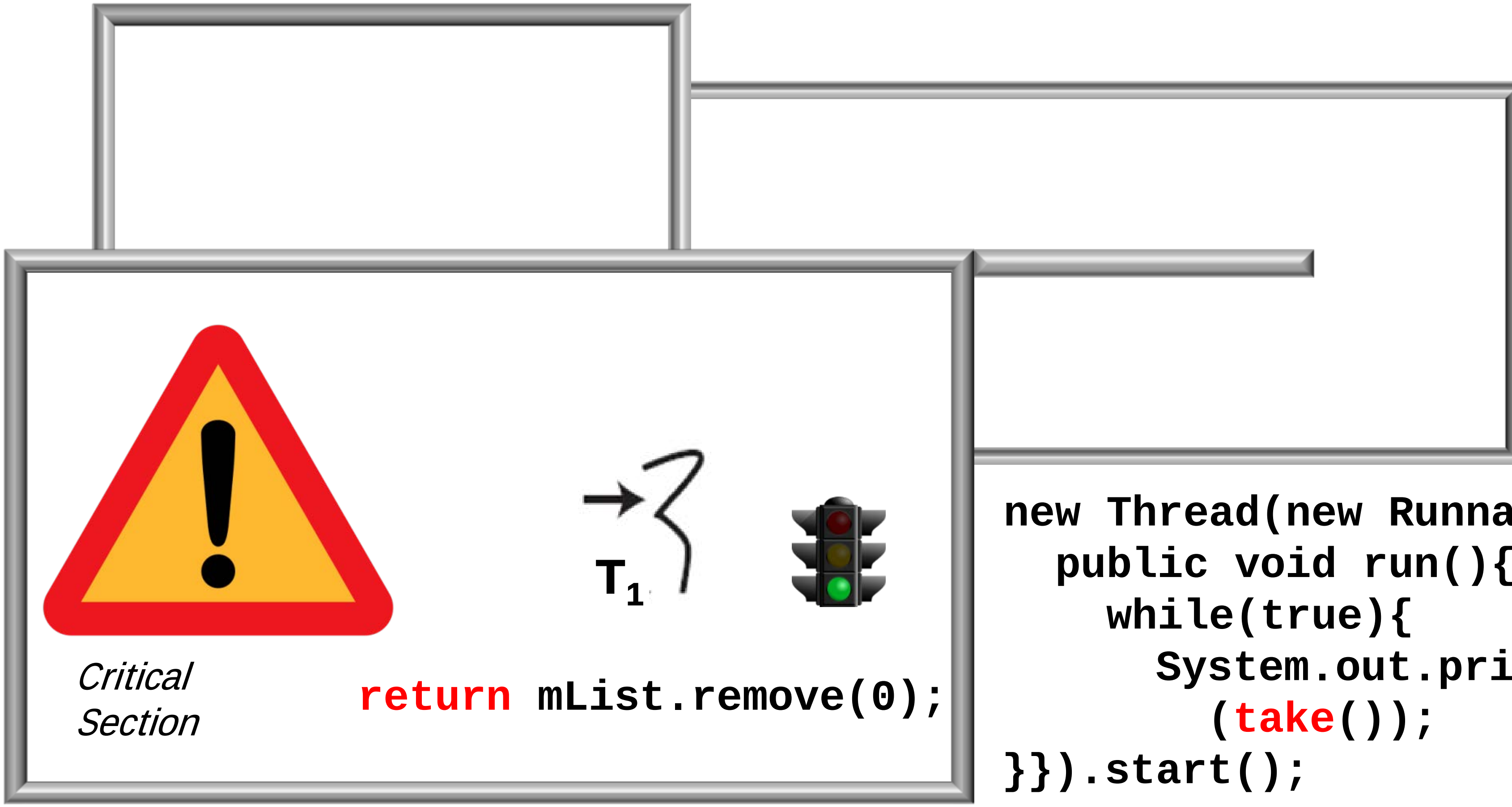
SimpleBlockingQueue




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

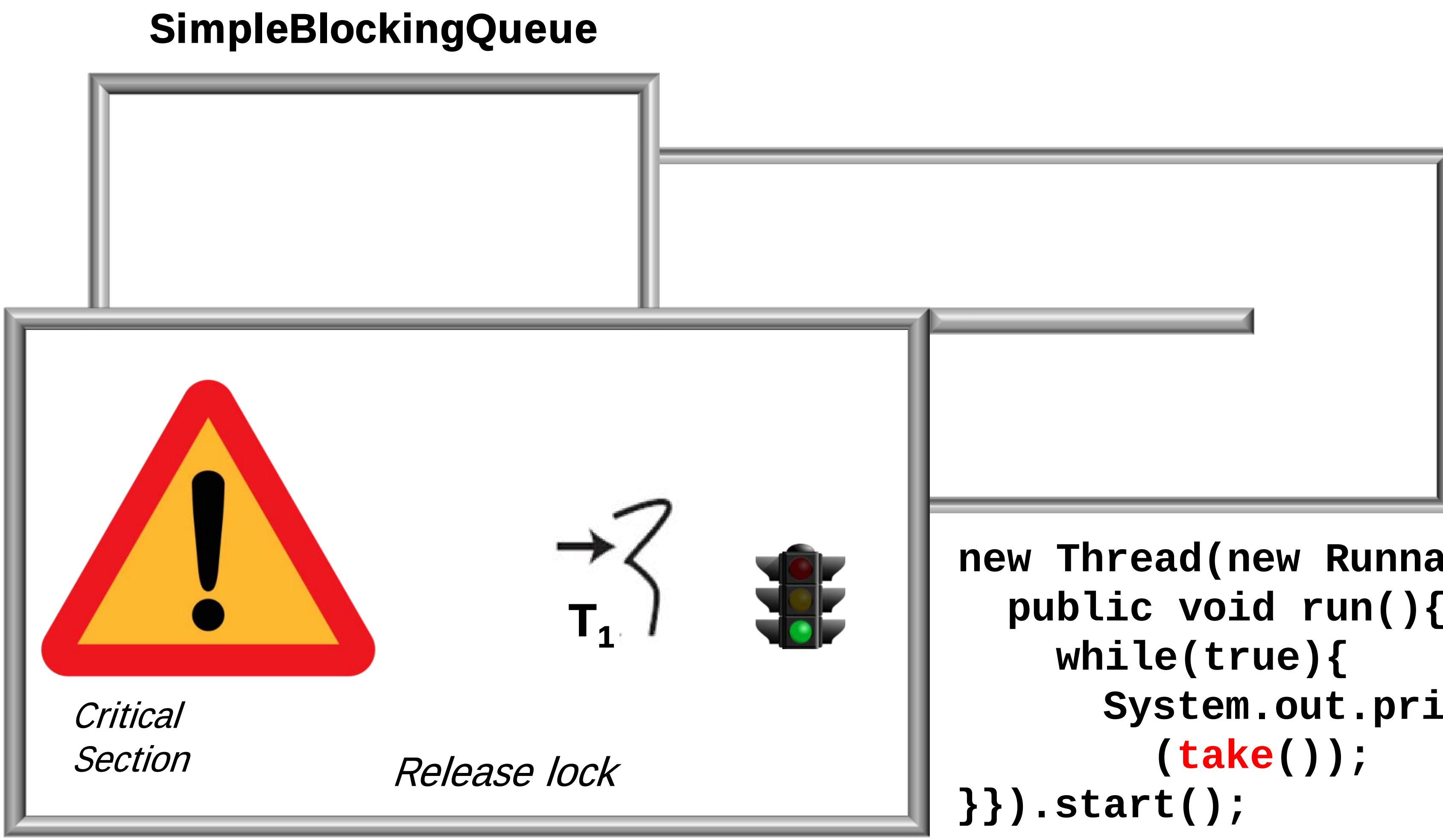
Visual Analysis of SimpleBlockingQueue

SimpleBlockingQueue




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

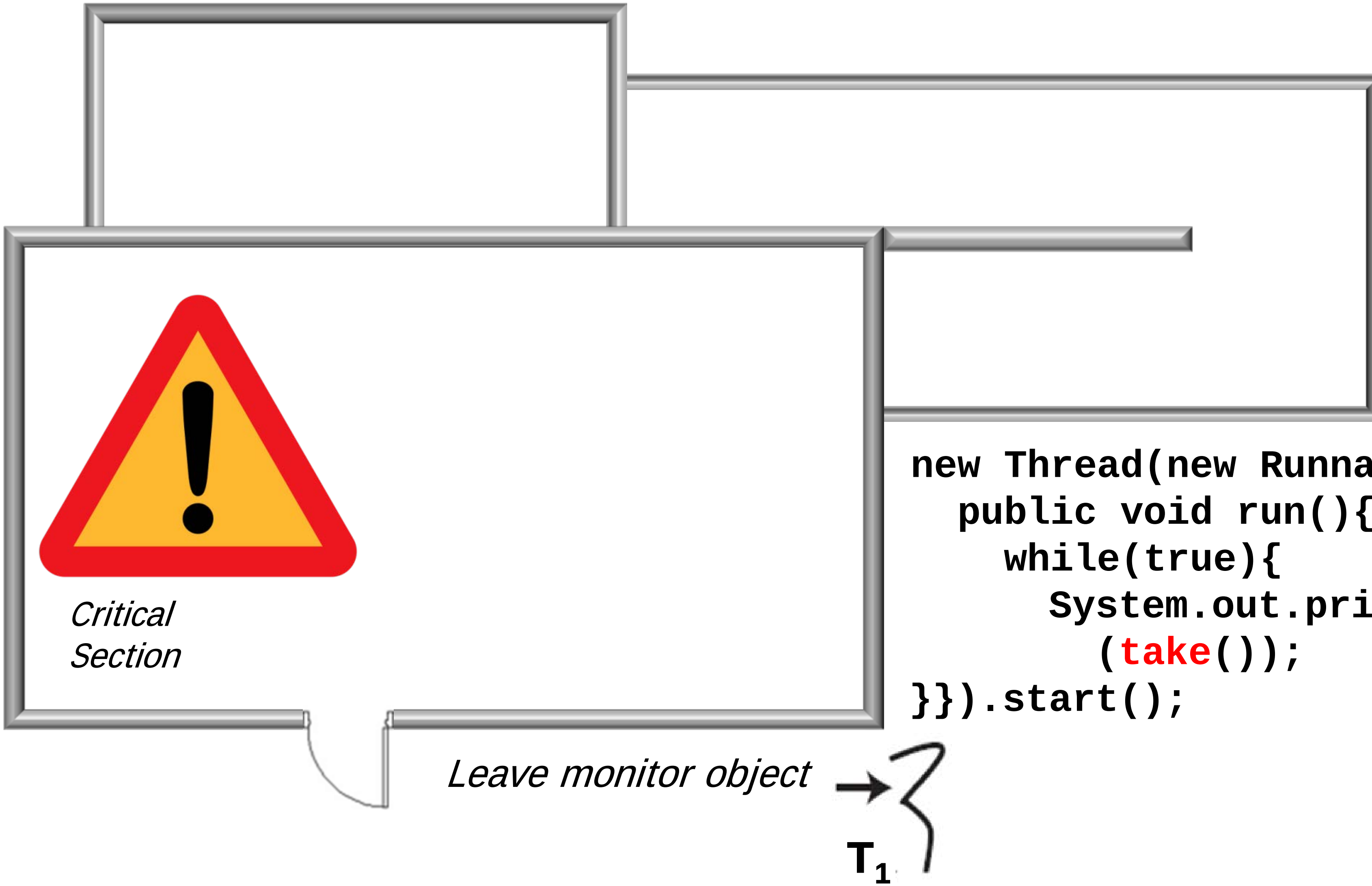
Visual Analysis of SimpleBlockingQueue




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

Visual Analysis of SimpleBlockingQueue

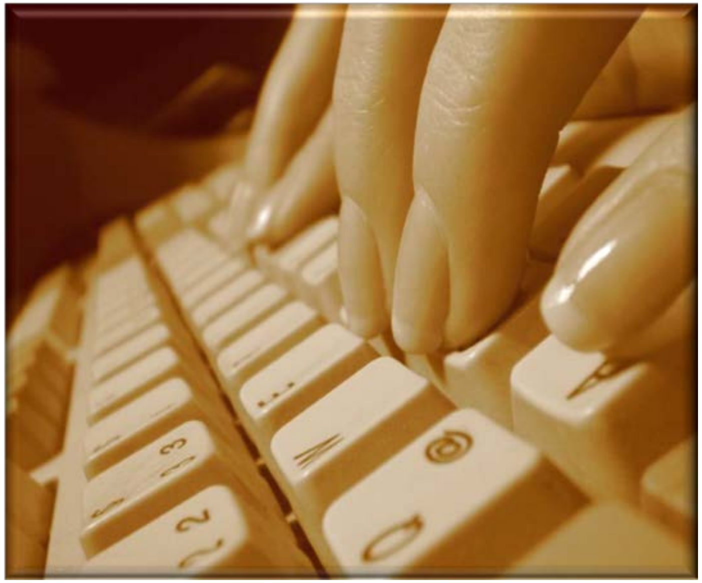
SimpleBlockingQueue




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

Code Analysis of the SimpleBlocking Queue Example

Code Analysis of SimpleBlockingQueue

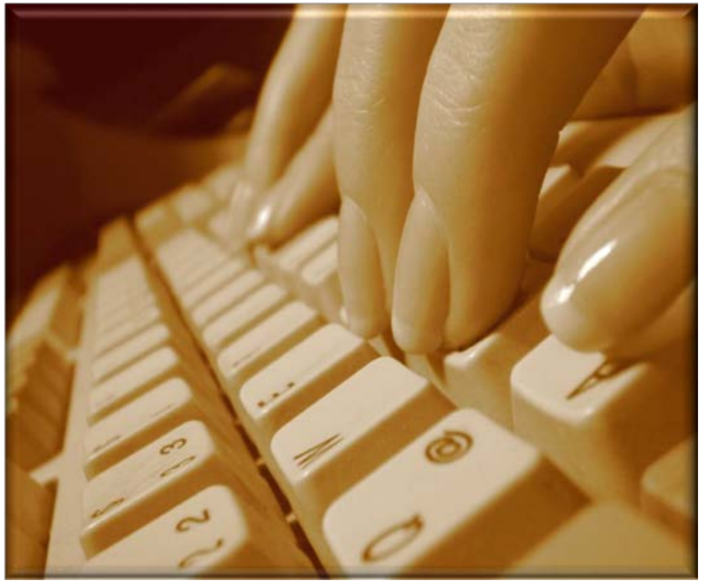


```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  private List<E> mList;
  private int mCapacity;

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

This internal state must be protected against race conditions

See github.com/douglascraigschmidt/CS282/tree/master/ex/SimpleBlockingQueue

Code Analysis of SimpleBlockingQueue



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

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

The constructor needn't be protected against race conditions

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

See en.wikipedia.org/wiki/Guarded_suspension

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
  - e.g., take() acquires the monitor lock & waits while the queue is empty

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
  - e.g., take() acquires the monitor lock & waits while the queue is empty

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can “wait” for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can “wait” for a condition in a synchronized method
- 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!

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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!

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

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

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Code Analysis of SimpleBlockingQueue

- A thread can “wait” for a condition in a synchronized method
- 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

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized void put(E msg){
    ...
    while (mList.isFull())
      wait();

    mList.add(msg);
    notifyAll();
  }

  private boolean isFull() {
    return mList.size() >= mCapacity;
  }
  ...
```

Note use of notifyAll() here, which stems from Java’s monitor object limitations

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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
- A blocked thread that's notified performs several steps

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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
- A blocked thread that's notified performs several steps
  - wakes up & obtains lock

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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
- A blocked thread that's notified performs several steps
  - wakes up & obtains lock
  - re-evaluates the condition

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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
- A blocked thread that's notified performs several steps
  - wakes up & obtains lock
  - re-evaluates the condition
  - continues after wait()

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

- A thread can "wait" for a condition in a synchronized method
- 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
- A blocked thread that's notified performs several steps
  - wakes up & obtains lock
  - re-evaluates the condition
  - continues after wait()

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Code Analysis of SimpleBlockingQueue

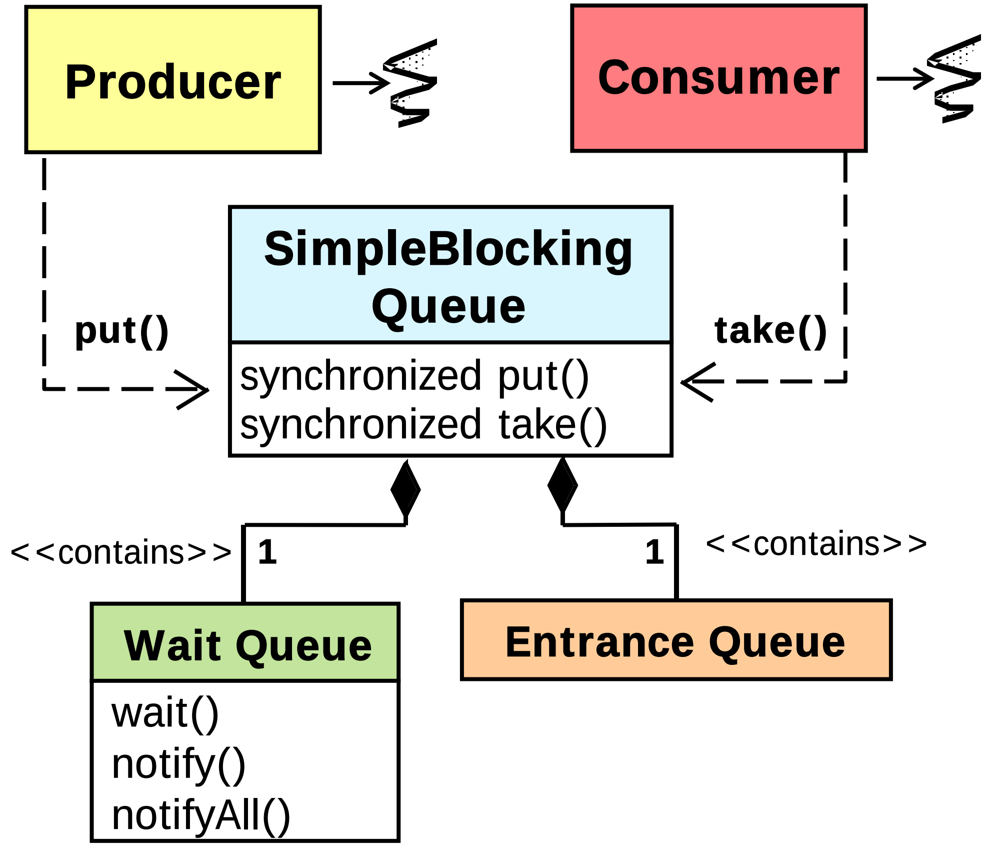
- A thread can "wait" for a condition in a synchronized method
- wait() should be called inside a loop that checks whether the condition is true or not
- A thread blocking on wait() won't continue until another thread notifies it that the condition may be true
- A blocked thread that's notified performs several steps
  - wakes up & obtains lock
  - re-evaluates the condition
  - continues after wait()
  - releases lock when it returns

```
class SimpleBlockingQueue<E> implements
      BlockingQueue<E> {
  ...

  public synchronized String take(){
    while (mList.isEmpty())
      wait();

    final E e = mList.remove(0);

    notifyAll();
    return e;
  }
```

Common Traps & Pitfalls of Java Built-in Monitor Objects (Part 1)

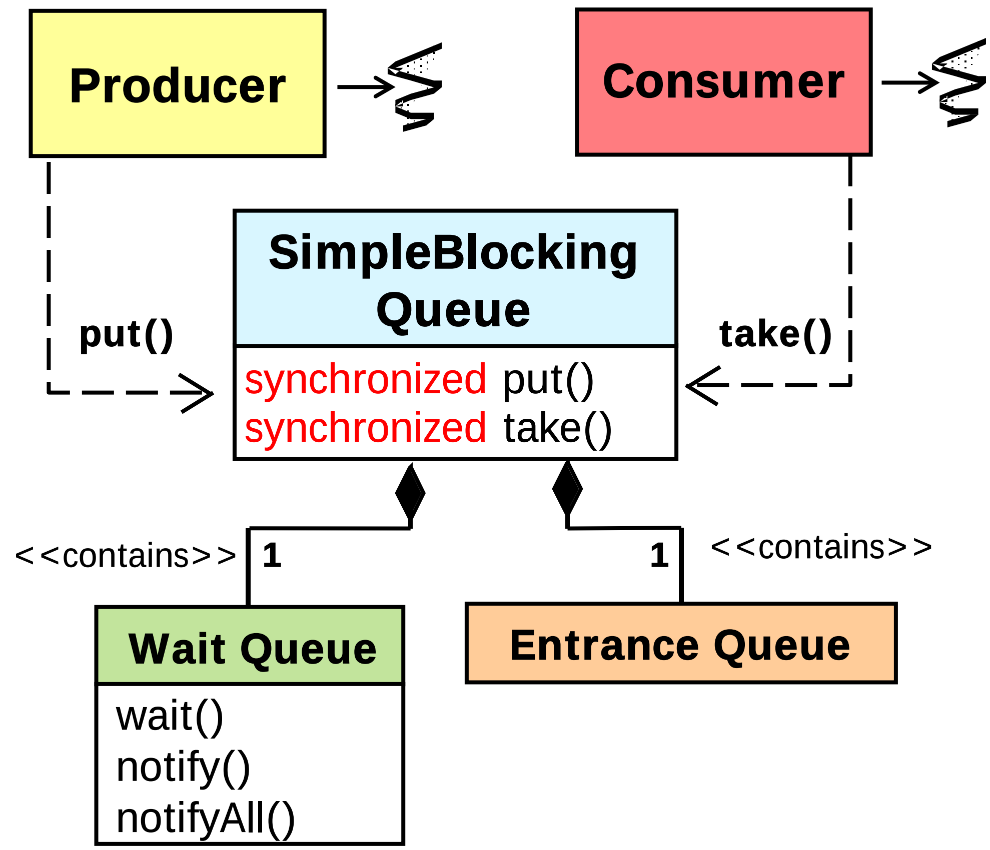
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects



Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Only one wait queue & one entrance queue per monitor object



See www.dre.vanderbilt.edu/~schmidt/C++2Java.html#concurrency

Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Only one wait queue & one entrance queue per monitor object
    - Can yield "nested monitor lockout"

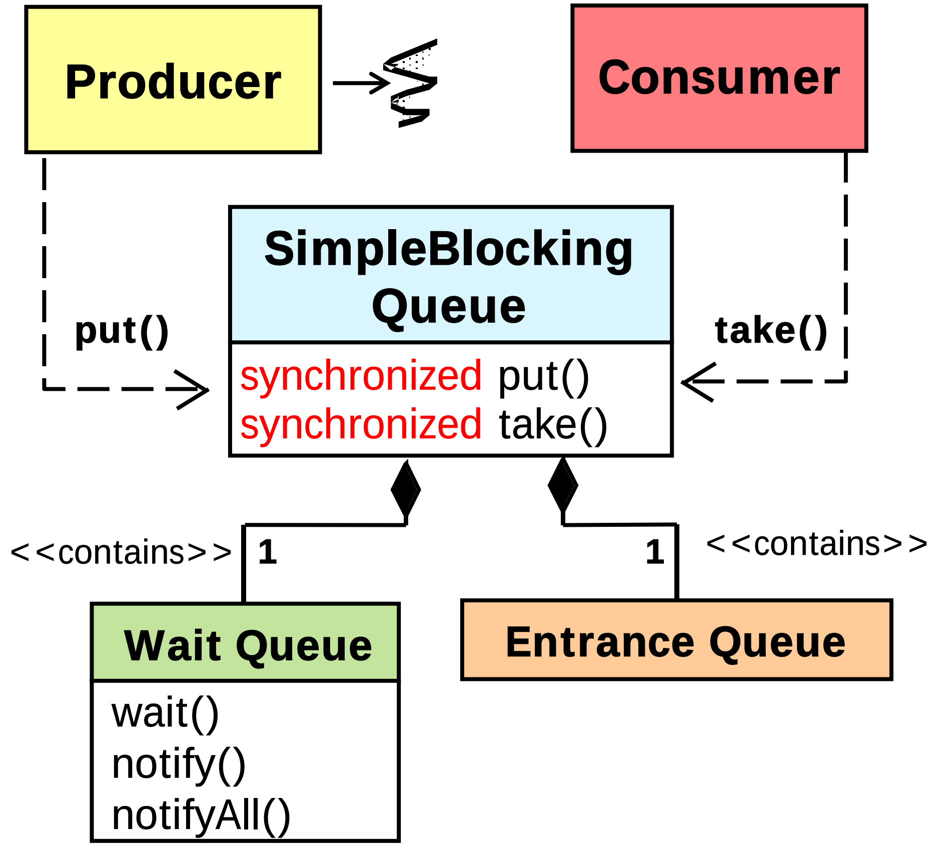
```
public class BuggyLock {
  protected Object mMonObj =
    new Object();
  protected boolean mLocked = false;

  public synchronized void lock() ... {
    while(mLocked){
      synchronized(mMonObj) {
        mMonObj.wait();
      }
    }
    mLocked = true;
  }

  public synchronized void unlock(){
    mLocked = false;
    synchronized(mMonObj){
      mMonObj.notify();
    }
  }
}
```

The BuggyLock monitor lock is still held here!

See tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html

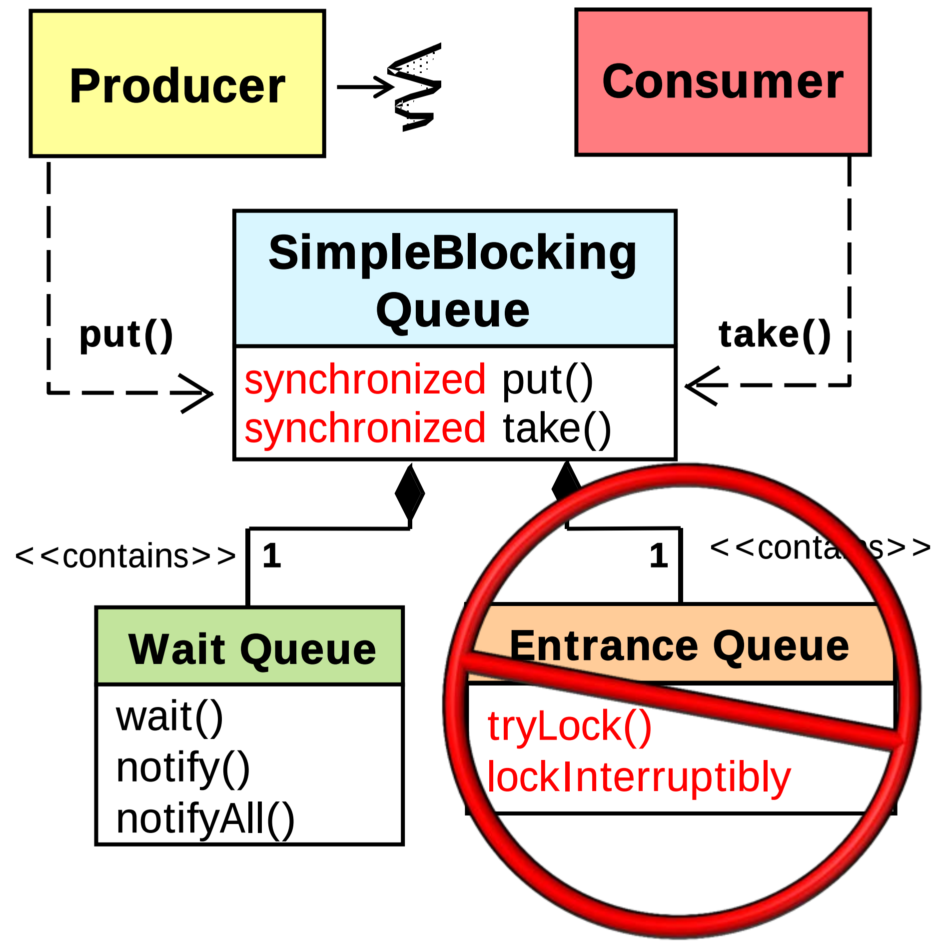
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Only one wait queue & one entrance queue per monitor object
  - Monitor locks lack certain features provided by ReentrantLock



Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Only one wait queue & one entrance queue per monitor object
  - Monitor locks lack certain features provided by ReentrantLock

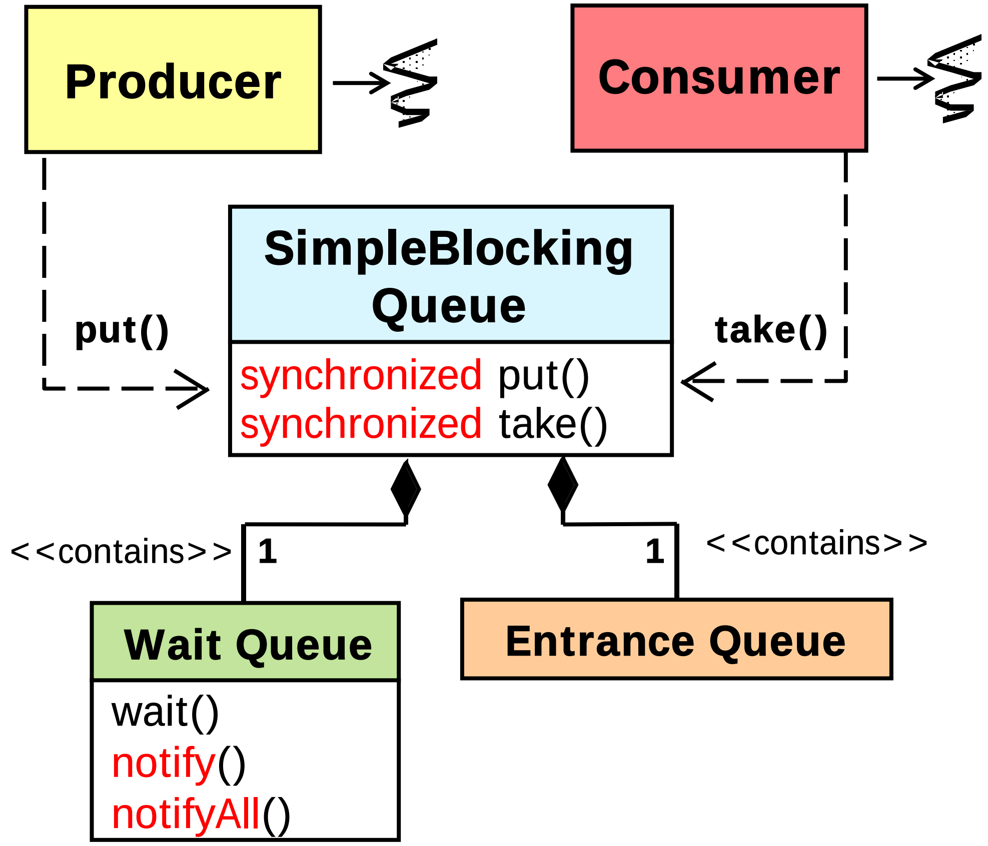


See libcore/luni/src/main/java/java/util/concurrent

Common Traps & Pitfalls of Java Built-in Monitor Objects (Part 2)

Common Traps & Pitfalls of Monitor Objects

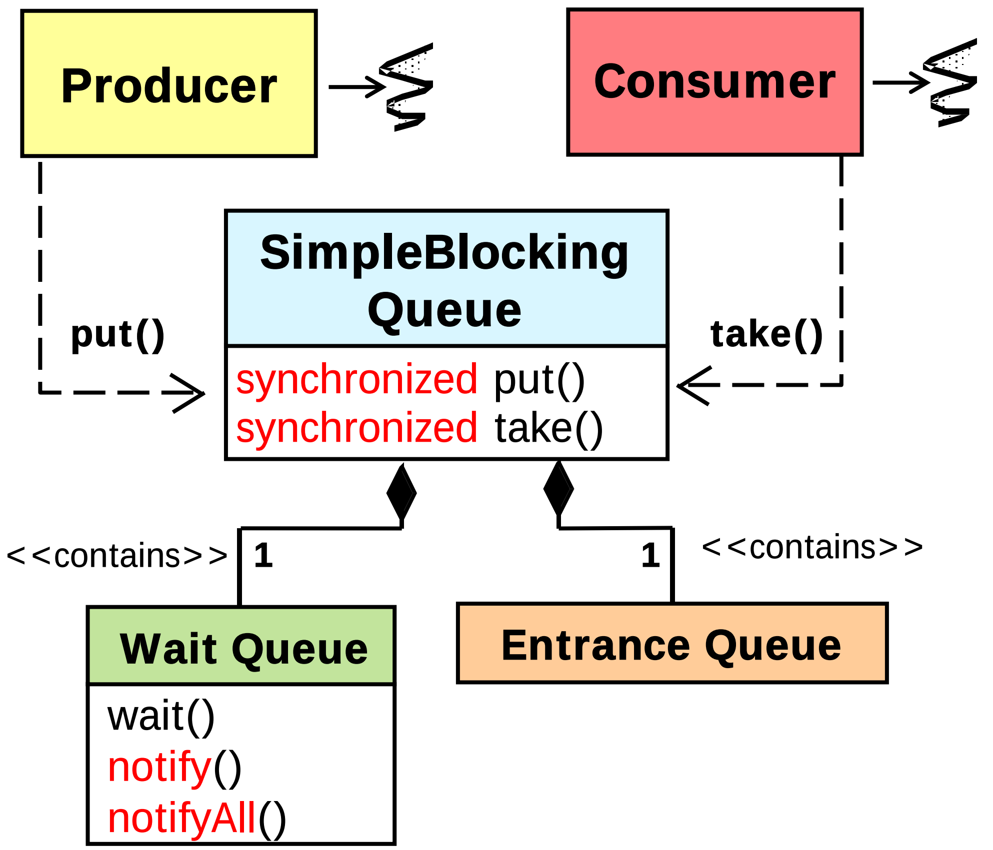
- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()



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

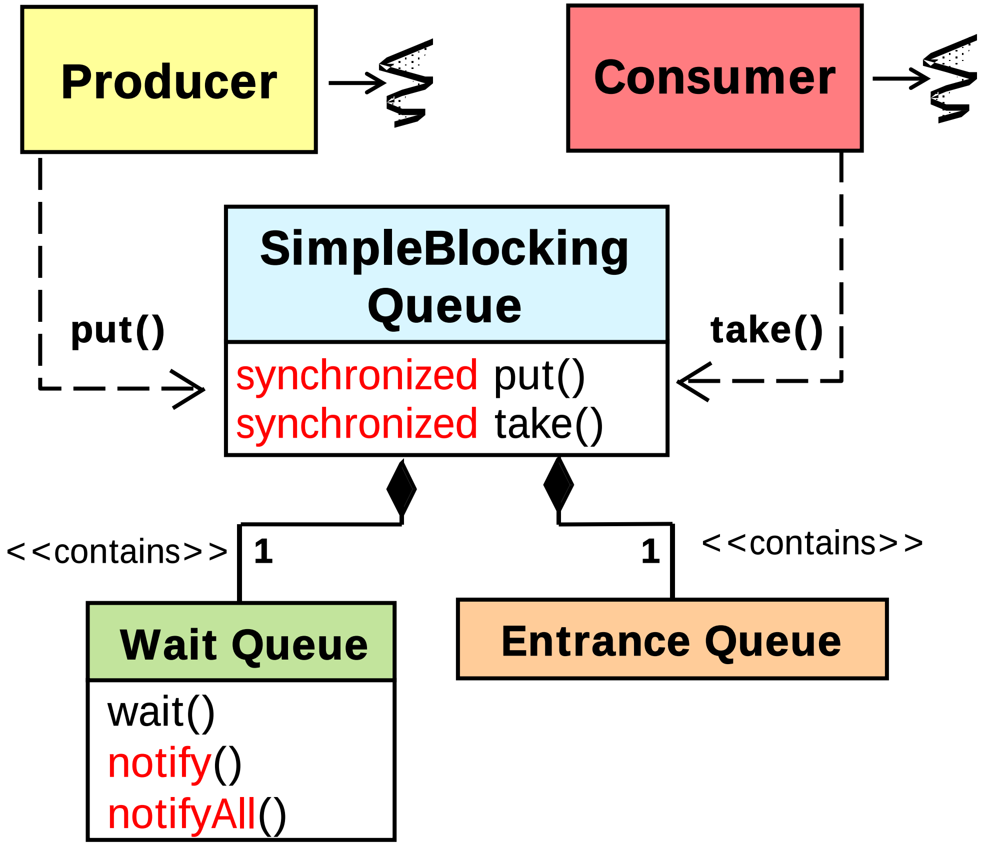
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
    - Use notify() in situations involving “uniform” waiters



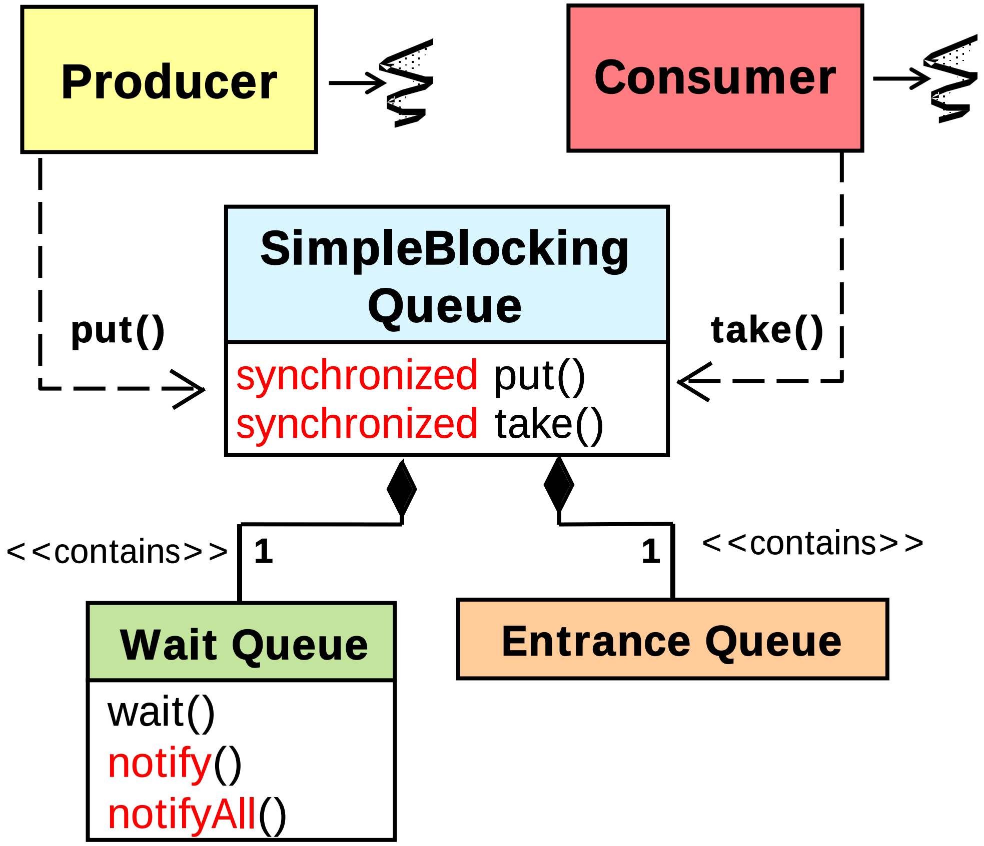
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
    - Use notify() in situations involving "uniform" waiters
      - Only one condition per wait queue



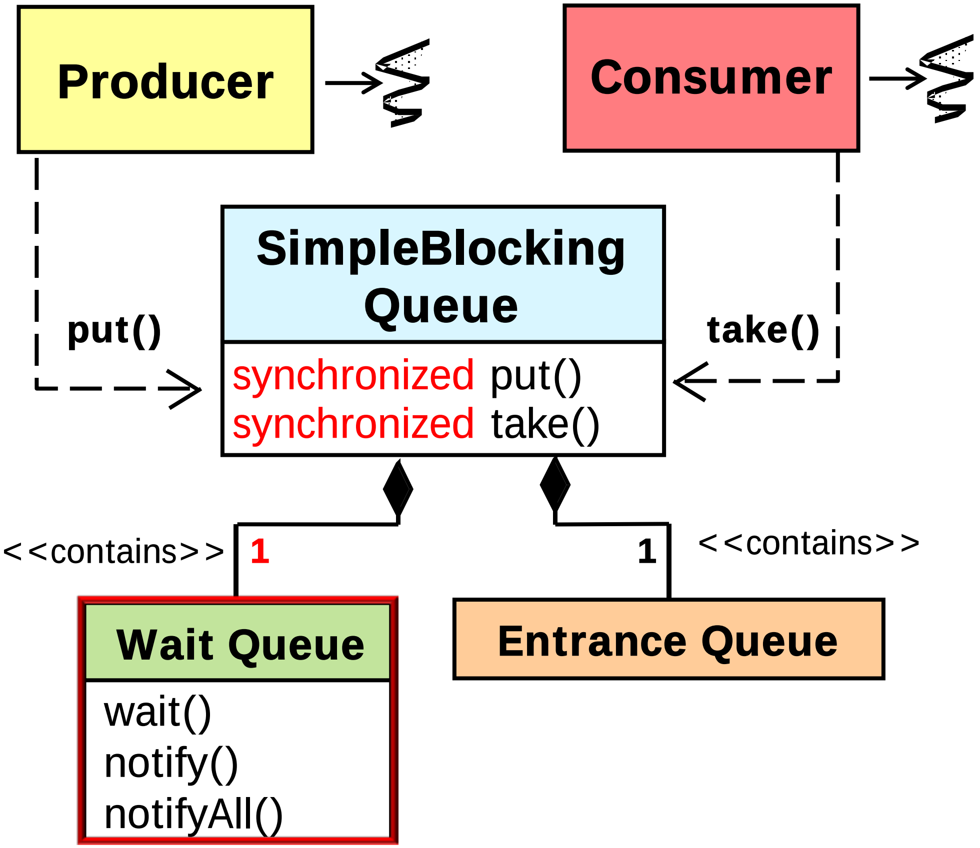
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
    - Use notify() in situations involving “uniform” waiters
      - Only one condition per wait queue
      - Each Thread executes the same logic after wait returns



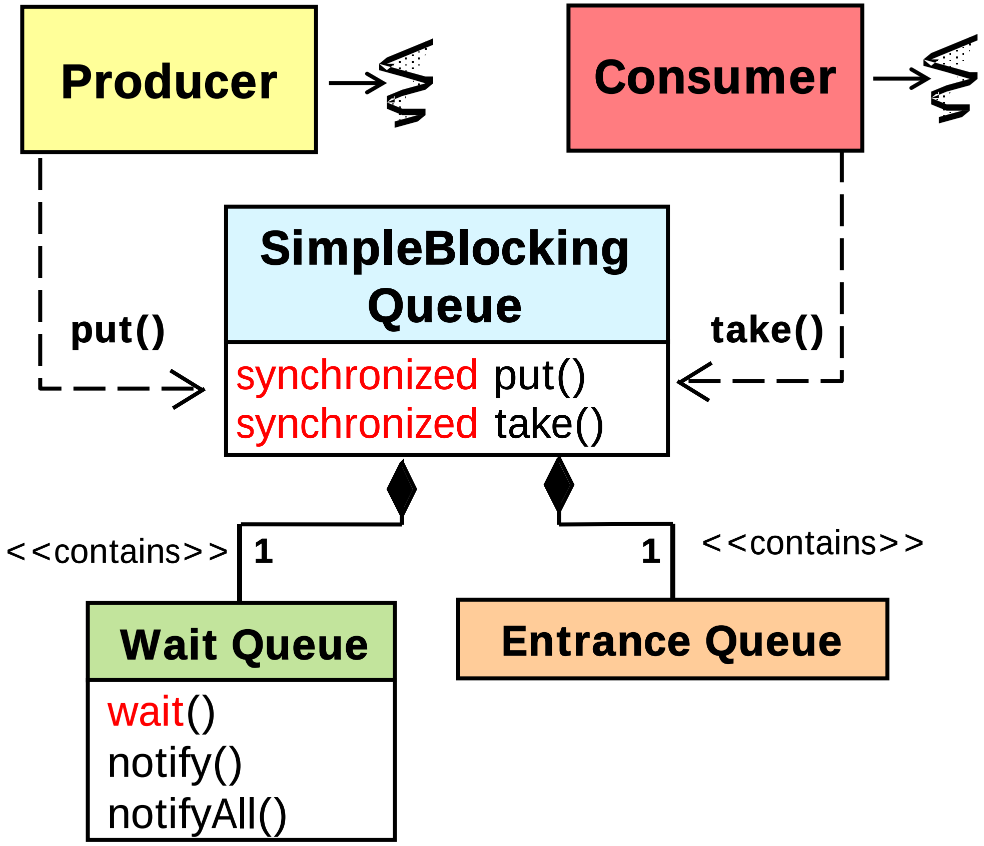
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
    - Use notify() in situations involving "uniform" waiters
    - Threads blocked on a monitor object typically wait for multiple conditions since there's just one wait queue



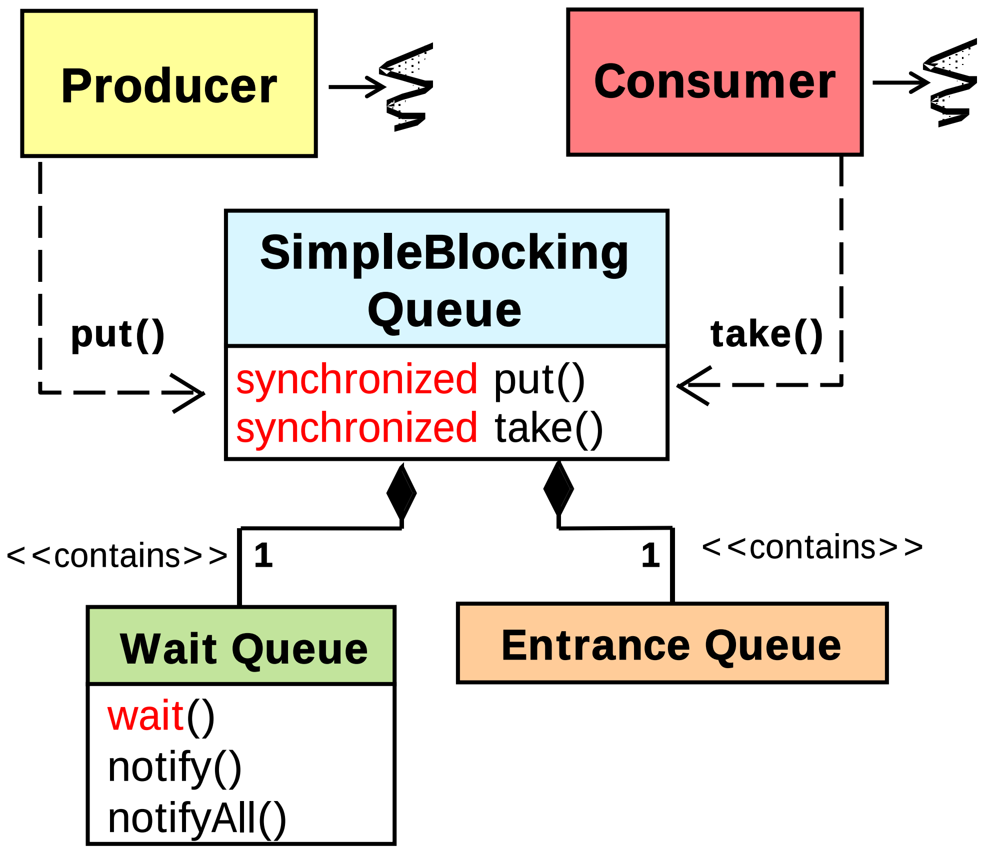
Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting Threads are notified



Common Traps & Pitfalls of Monitor Objects

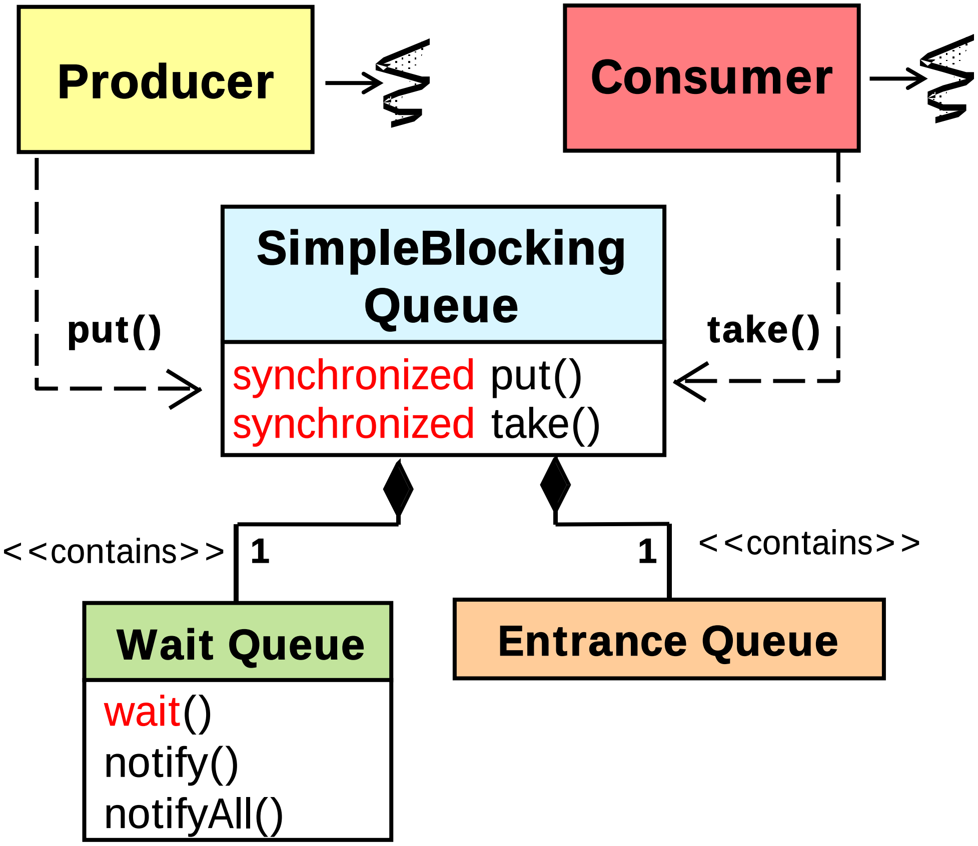
- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting Threads are notified
    - By default, monitor object's implement "haphazard notification" semantics



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

Common Traps & Pitfalls of Monitor Objects

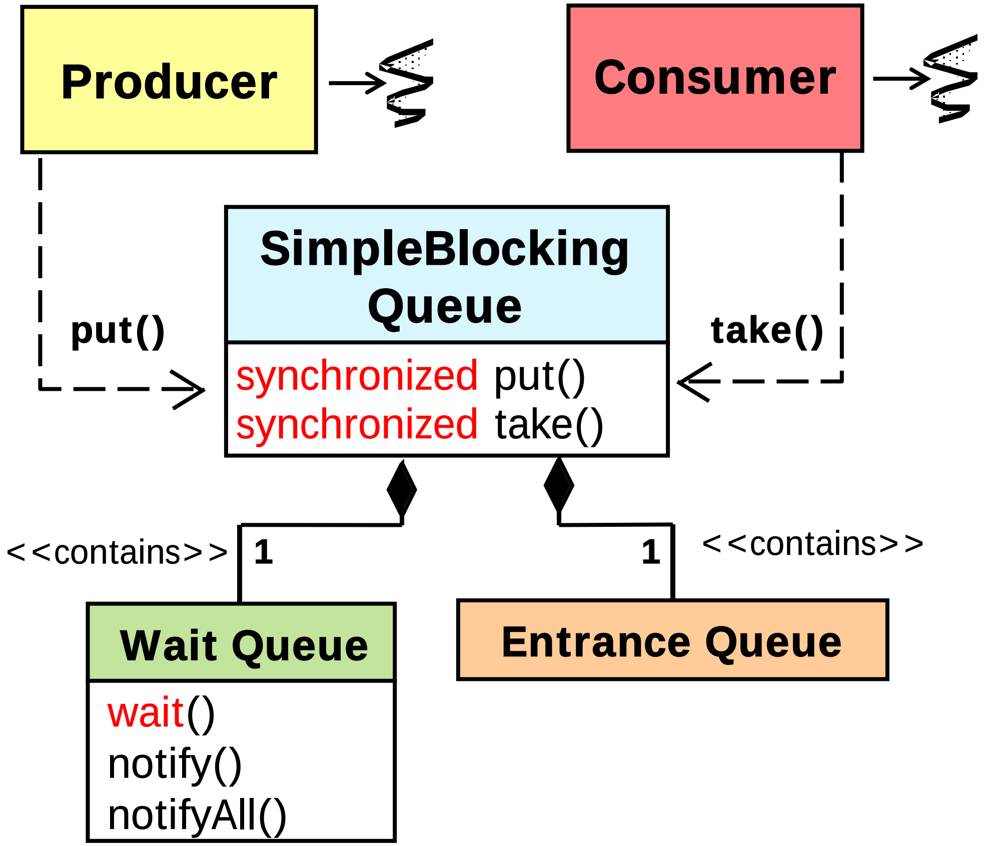
- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting Threads are notified
    - By default, monitor object's implement "haphazard notification" semantics
    - The *Specific Notification* pattern can be applied here



See www.ibm.com/developerworks/java/library/j-spnotif.html

Common Traps & Pitfalls of Monitor Objects

- Be aware of certain issues with Java built-in monitor objects
  - Subtleties associated with calling notify() vs. notifyAll()
  - Fairness issues related to the order in which waiting Threads are notified
    - By default, monitor object's implement "haphazard notification" semantics
    - The *Specific Notification* pattern can be applied here
      - Programmatically choose a particular Thread to run from a set of waiting Threads



Common Traps & Pitfalls of Monitor Objects

- 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

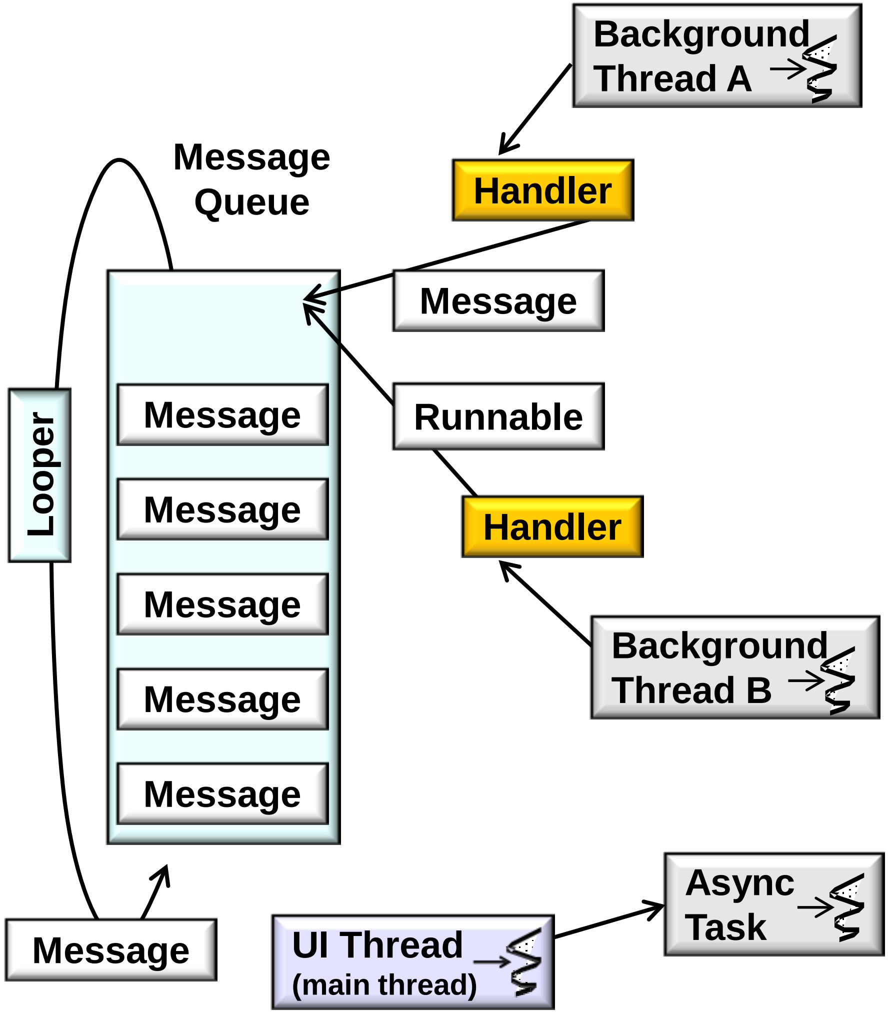
java.util.concurrent

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.

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

Common Traps & Pitfalls of Monitor Objects

- 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



See developer.android.com/guide/components/processes-and-threads.html#Threads