Java Monitor Objects:
Overview

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

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand what monitors are & know how Java built-in monitor objects can ensure mutual exclusion & coordination between threads.
Learning Objectives in this Part of the Lesson

- Understand what monitors are & know how Java built-in monitor objects can ensure mutual exclusion & coordination between threads
- Recognize a human known use of monitors
Overview of Monitors
Overview of Monitors

• A monitor is a synchronization mechanism designed in the early 1970s

See en.wikipedia.org/wiki/Monitor_(synchronization)
Overview of Monitors

- A monitor provides three capabilities to concurrent programs.
Overview of Monitors

A monitor provides three capabilities to concurrent programs:

1. Only one thread at a time has mutually exclusive access to a critical section.

See en.wikipedia.org/wiki/Critical_section
Overview of Monitors

- A monitor provides three capabilities to concurrent programs
  1. Only one thread at a time has mutually exclusive access to a critical section
  2. Threads running in a monitor can block awaiting certain conditions to become true
Overview of Monitors

- A monitor provides three capabilities to concurrent programs

1. Only one thread at a time has mutually exclusive access to a critical section

2. Threads running in a monitor can block awaiting certain conditions to become true

3. A thread can notify one or more threads that conditions they’re waiting on have been met
Overview of Built-in Java Monitor Objects
Overview of Java Built-in Monitor Objects

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

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

# Implicit_condition_variable_monitors
Overview of Java Built-in Monitor Objects

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

  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions

![Diagram](https://via.placeholder.com/150)

- **Thread 1**
  - `m1()` synchronized
  - `m2()` synchronized

- **Thread 2**
  - `m1()`
  - `m2()`
All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization:

- **Mutual exclusion** - allows concurrent access & updates to shared data without race conditions.
Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions
  - **Coordination** – Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.

A Java Monitor Object

![Diagram showing Thread 1 entering synchronized m1() and m2(), Thread 2 in an entrance queue, and mutual exclusion and coordination with mutual exclusion锁图标。]
Overview of Java Built-in Monitor Objects

- All objects in Java can be used as built-in monitor objects, which support two types of thread synchronization
  - **Mutual exclusion** – allows concurrent access & updates to shared data without race conditions
  - **Coordination** – Ensures computations run properly, e.g., in the right order, at the right time, under the right conditions, etc.

Java's execution environment supports coordination via a wait queue & notification mechanisms.

All Java objects have one “intrinsic condition” associated with it.
Overview of Java Built-in Monitor Objects

- These mechanisms implement a variant of the *Monitor Object* pattern.
Overview of Java Built-in Monitor Objects

- These mechanisms implement a variant of the *Monitor Object* pattern

- **Intent** – Ensure that only one method runs within an object & allow an object’s methods to cooperatively schedule their execution sequences

![Diagram showing the interaction between threads and monitor objects](image-url)
Human Known Use of Monitors
A human known use of a monitor is an operating room in a hospital.
End of Java Monitor Objects: Overview
Java Monitor Objects:
Motivating Example & Common Concurrency Problems

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

Institute for Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand synchronization problems in a buggy concurrent Java program

Concurrent calls to offer() & poll() corrupt internal state in the BuggyQueue fields
Learning Objectives in this Part of the Lesson

- Understand synchronization problems in a buggy concurrent Java program
- Recognize some common complexities of concurrent programs
A Buggy Producer
/Consumer App
A Buggy Producer/Consumer App

- A concurrent producer/consumer app that attempts to pass messages via an “BuggyQueue” class

See [github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BuggyQueue](https://github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BuggyQueue)
A Buggy Producer/Consumer App

- The BuggyQueue class is modeled on the Java ArrayBoundedQueue class

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/ArrayBoundedQueue.html
A Buggy Producer/Consumer App

- UML class diagram showing the design of the BuggyQueue

A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test

See github.com/douglascraigschmidt/POSA/tree/master/ex/M3/Queues/BuggyQueue/app/src/test/java/edu/vandy/buggyqueue
A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test
A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test
A Buggy Producer/Consumer App

- UML sequence diagram of the BuggyQueue producer/consumer unit test
A Buggy Producer/Consumer App

• UML sequence diagram of the BuggyQueue producer/consumer unit test

Since the offer() & poll() methods aren’t synchronized chaos & insanity will result when this app and/or unit test is run!!
The BuggyQueue Implementation
The BuggyQueue Implementation

- The BuggyQueue class is a simply wrapper around Java’s LinkedList class

```java
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>(); ...

    public void offer(E e) {
        if (!isFull())
            { mList.add(e); return true; }
        else
            return false;
    }

    public E poll() {
        if (!isEmpty())
            return mList.remove(0);
        else
            return null;
    }

    ...
```

The BuggyQueue Implementation

• The BuggyQueue class is a simply wrapper around Java’s LinkedList class

```java
class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>(); ...

    public void offer(E e) {
        if (!isFull())
            { mList.add(e); return true; }
        else
            return false;
    }

    public E poll() {
        if (!isEmpty())
            return mList.remove(0);
        else
            return null;
    }
    ...
```
The BuggyQueue Implementation

- The BuggyQueue class is a simply wrapper around Java's LinkedList class

```java
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>(); ...

    public void offer(E e) {
        if (!isFull())
            { mList.add(e); return true; }
        else
            return false;
    }

    public E poll() {
        if (!isEmpty())
            return mList.remove(0);
        else
            return null;
    }
...
```

See docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
The BuggyQueue Implementation

• The BuggyQueue class is a simply wrapper around Java’s LinkedList class

```java
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>();
    ...

    public void offer(E e) {
        if (!isFull()) {
            mList.add(e); return true;
        } else {
            return false;
        }
    }

    public E poll() {
        if (!isEmpty()) {
            return mList.remove(0);
        } else {
            return null;
        }
    }

    ...
}
```

Non-synchronized public methods

- `offer(E e)`
- `poll()`
The BuggyQueue Implementation

- The BuggyQueue class is a simply wrapper around Java's LinkedList class

```java
static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<>(); ...

    public void offer(E e) {
        if (!isFull())
            { mList.add(e); return true; }
        else
            return false;
    }

    public E poll() {
        if (!isEmpty())
            return mList.remove(0);
        else
            return null;
    }

    ...}
```

See en.wikipedia.org/wiki/Robot_B-9
Evaluating the Buggy Producer/Consumer
Evaluating the Buggy Producer/Consumer

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

```
consumer : Thread
producer : Thread
buggyQueue : BuggyQueue

: Buggy QueueTest
```

```
main():
  start()
  new()

consumer:
  run()
  new()

producer:
  run()
  offer("…")
  poll()

buggyQueue:
  new()
```
Evaluating the Buggy Producer/Consumer

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

Exception in thread "Thread-1" java.lang.NullPointerException
at java.util.LinkedList.unlink(LinkedList.java:211)
at java.util.LinkedList.remove(LinkedList.java:526)
at edu.vandy.buggyqueue.model.BuggyQueue.poll(BuggyQueue.java:52)
at edu.vandy.BuggyQueueTest$Consumer.run(BuggyQueueTest.java:104)
at java.lang.Thread.run(Thread.java:745)

Depending on the implementation of the BuggyQueue class & the underlying LinkedList the app & test program may simply “hang”
Evaluating the Buggy Producer/Consumer

• Key question: what’s the output & why?

static class BuggyQueue<E> implements BoundedQueue<E> {
    private LinkedList<E> mList = new LinkedList<E>();

    public boolean offer(E e) {
        if (!isFull()) { mList.add(e); return true; }
        else return false;
    }

    public E poll() {
        if (!isEmpty()) return mList.remove(0); else return false; }
    ...
}

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

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

See docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html
Evaluating the Buggy Producer/Consumer

• Concurrent programs are hard to develop & debug, due to various complexities

See stackoverflow.com/questions/499634/how-to-detect-and-debug-multi-threading-problems
Concurrent programs are hard to develop & debug, due to various complexities, e.g.

- Deadlock
  - Occurs when two or more competing actions are each waiting for the other to finish, & thus none ever do

See en.wikipedia.org/wiki/Deadlock
Concurrent programs are hard to develop & debug, due to various complexities, e.g.

- **Deadlock**
- **Starvation**
  - A thread is perpetually denied necessary resources to process its work

See en.wikipedia.org/wiki/Starvation_(computer_science)
Evaluating the Buggy Producer/Consumer

- Concurrent programs are hard to develop & debug, due to various complexities, e.g.
  - Deadlock
  - Starvation
  - Race conditions
  - Arise when an application depends on the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition
Concurrent programs are hard to develop & debug, due to various complexities, e.g.

- Deadlock
- Starvation
- Race conditions
- Tool limitations
  
  - e.g., behavior in the debugger doesn’t reflect actual behavior

See [en.wikipedia.org/wiki/Heisenbug](https://en.wikipedia.org/wiki/Heisenbug)

The act of observing a system can alter its state
Evaluating the Buggy Producer/Consumer

- Some of these problems can be fixed by applying Java built-in monitor object mechanisms
There are also helpful techniques for debugging concurrent software.

See www.drdobbs.com/cpp/multithreaded-debugging-techniques/199200938
End of Java Monitor Objects: Motivating Example