The Pervasiveness & Complexity of Java Synchronizers

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

- Be aware of the Java memory model
- Understand the purpose of Java synchronizers
- Recognize the pervasiveness of Java synchronizers

e.g., Java atomics, locks, & other synchronizers

e.g., volatile variables & built-in monitor objects
Learning Objectives in this Part of the Lesson

• Be aware of the Java memory model
• Understand the purpose of Java synchronizers
• Recognize the pervasiveness of Java synchronizers
• As well as their complexities
The Pervasiveness of Synchronizers in Java
The Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform

See [en.wikipedia.org/wiki/Java_(software_platform)](en.wikipedia.org/wiki/Java_(software_platform))
The Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
- The Java language contains some features that synchronize threads

\[\text{e.g., volatile variables & built-in monitor objects}\]

See [en.wikipedia.org/wiki/Javaprogramming_language](en.wikipedia.org/wiki/Javaprogramming_language)
The Pervasiveness of Java Synchronizer Classes

- Multiple layers of synchronizers are provided on the Java platform, e.g.
  - The Java language contains some features that synchronize threads
  - Other synchronizers are provided by the Java Class Library

  e.g., Java atomics, various locks, conditions, semaphores, & barriers

See [en.wikipedia.org/wiki/Java_Class_Library](en.wikipedia.org/wiki/Java_Class_Library)
The Complexities of Synchronizers in Java
The Complexities of Java Synchronizer Classes

• Synchronization complexity arises from coordinating the interactions of entities that run concurrently
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities

See [en.wikipedia.org/wiki/No_Silver_Bullet](en.wikipedia.org/wiki/No_Silver_Bullet)
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - **Inherent complexities**

*These fundamental challenges constitute the “rocket science” of the synchronization domain*

The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - Inherent complexities
    - Mutual Exclusion
      - Ensure concurrent threads don’t simultaneously run in a program’s critical sections

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
The Complexities of Java Synchronizer Classes

• There are two general types of synchronization complexities

  • **Inherent complexities**
    • *Mutual Exclusion*
    • *Coordination*
      • Manage the order or time in which operations are performed to ensure threads access system resources correctly & efficiently
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - **Inherent complexities**
    - *Mutual Exclusion*
    - *Coordination*
    - *Deadlock*
      - Occurs when 2+ competing actions each wait for the other to finish, & thus none ever do
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - Inherent complexities
  - Accidental complexities

These complexities arise from common limitations with techniques, tools, & methods used to synchronize programs.

See [wiki.c2.com/?AccidentalComplexity](http://wiki.c2.com/?AccidentalComplexity)
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - Inherent complexities
  - **Accidental complexities**
    - Tool limitations make it hard to debug concurrent programs

See [en.wikipedia.org/wiki/Trepanning](en.wikipedia.org/wiki/Trepanning) for more on traditional “debugging” techniques!
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - Inherent complexities
  - Accidental complexities
    - Tool limitations make it hard to debug concurrent programs
    - The behavior in the debugger doesn’t reflect actual behavior

See [en.wikipedia.org/wiki/Heisenbug](en.wikipedia.org/wiki/Heisenbug)
The Complexities of Java Synchronizer Classes

- There are two general types of synchronization complexities
  - Inherent complexities
  - Accidental complexities
    - Tool limitations make it hard to debug concurrent programs
      - The behavior in the debugger doesn’t reflect actual behavior
    - Lack of tool support to identify & rectify race conditions

Occur when multiple threads “crash” into unprotected data structures & corrupt them

See en.wikipedia.org/wiki/Race_condition
There are two general types of synchronization complexities:

- **Inherent complexities**
- **Accidental complexities**
  - Tool limitations make it hard to debug concurrent programs
  - The behavior in the debugger doesn’t reflect actual behavior
  - Lack of tool support to identify & rectify *race conditions*
  - Conventional Java debuggers don’t detect race conditions

Problems often don’t surface until runtime
The Complexities of Java Synchronizer Classes

- Java’s parallelism frameworks help reduce synchronization complexities via “divide & conquer”

The Complexities of Java Synchronizer Classes

- Java’s parallelism frameworks help reduce synchronization complexities via “divide & conquer”
- These frameworks largely eliminate the need for synchronization when writing concurrent apps
End of the Pervasiveness & Complexity of Java Synchronizers