Overview of How Concurrent Programs are Developed in Java

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

Professor of Computer Science

Institute for Software Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand the meaning of key concurrent programming concepts
- Recognize how Java supports concurrent programming concepts
  - e.g., via threads, shared objects (synchronizers), & message passing
Learning Objectives in this Part of the Lesson

• Understand the meaning of key concurrent programming concepts

• Recognize how Java supports concurrent programming concepts
  • e.g., via threads, shared objects (synchronizers), & message passing

Traditional Java threads are undergoing major changes as part of Project Loom

See wiki.openjdk.java.net/display/loom/Main
Learning Objectives in this Part of the Lesson

• Understand the meaning of key concurrent programming concepts
• Recognize how Java supports concurrent programming concepts
• Be aware of common concurrency hazards faced by Java programmers
An Overview of Concurrent Programming in Java
An Overview of Concurrent Programming in Java

- A Java Thread is an object

```java
public class Thread
extends Object
implements Runnable

A thread is a thread of execution in a program. The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

Every thread has a priority. Threads with higher priority are executed in preference to threads with lower priority. Each thread may or may not also be marked as a daemon. When code running in some thread creates a new Thread object, the new thread has its priority initially set equal to the priority of the creating thread, and is a daemon thread if and only if the creating thread is a daemon.
```

See docs.oracle.com/javase/8/docs/api/java/lang/Thread.html
An Overview of Concurrent Programming in Java

• A Java Thread is an object, e.g.
  • It contains methods & fields

Historically each Java Thread had its own unique id, name, priority, runtime stack, thread-local storage, instruction pointer, & other registers, etc.

See blog.jamesdbloom.com/JVMInternals.html
An Overview of Concurrent Programming in Java

- A Java Thread is an object, e.g.
- It contains methods & fields

Platform threads

Thread supports the creation of platform threads that are typically mapped 1:1 to kernel threads scheduled by the operating system. Platform threads will usually have a large stack and other resources that are maintained by the operating system. Platforms threads are suitable for executing all types of tasks but may be a limited resource.

Platform threads are designated daemon or non-daemon threads. When the Java virtual machine starts up, there is usually one non-daemon thread (the thread that typically calls the application's main method). The Java virtual machine terminates when all started non-daemon threads have terminated. Unstarted daemon threads do not prevent the Java virtual machine from terminating. The Java virtual machine can also be terminated by invoking the Runtime.exit(int) method, in which case it will terminate even if there are non-daemon threads still running.

In addition to the daemon status, platform threads have a thread priority and are members of a thread group.

Platform threads get an automatically generated thread name by default.

Virtual threads

Thread also supports the creation of virtual threads. Virtual threads are typically user-mode threads scheduled by the Java virtual machine rather than the operating system. Virtual threads will typically require few resources and a single Java virtual machine may support millions of virtual threads. Virtual threads are suitable for executing tasks that spend most of the time blocked, often waiting for I/O operations to complete. Virtual threads are not intended for long running CPU intensive operations.

Virtual threads typically employ a small set of platform threads are use as carrier threads. Locking and I/O operations are the scheduling points where a carrier thread is re-scheduled from one virtual thread to another. Code executing in a virtual thread will usually not be aware of the underlying carrier thread, and in particular, the currentThread() method, to obtain a reference to the current thread, will return the Thread object for the virtual thread, not the underlying carrier thread.

See download.java.net/java/early_access/loom/docs/api/java.base/java/lang/Thread.html
An Overview of Concurrent Programming in Java

- A Java Thread is an object, e.g.
  - It contains methods & fields
  - It can also be in one of various “states”

See docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html

States of "classic" Java (platform) threads
An Overview of Concurrent Programming in Java

- A Java Thread is an object, e.g.
  - It contains methods & fields
  - It can also be in one of various “states”

See www.youtube.com/watch?v=5brCaY31y1M
An Overview of Concurrent Programming in Java

- Java threads interact via shared objects and/or message passing

See docs.oracle.com/javase/8/docs/api/?java/util/concurrent/package-summary.html
An Overview of Concurrent Programming in Java

- Java threads interact via shared objects and/or message passing

- **Shared objects**
  - Synchronize concurrent operations on objects to ensure certain properties

See [en.wikipedia.org/wiki/Synchronization_(computer_science)](en.wikipedia.org/wiki/Synchronization_(computer_science))
An Overview of Concurrent Programming in Java

- Java threads interact via shared objects and/or message passing

  - **Shared objects**
    - Synchronize concurrent operations on objects to ensure certain properties, e.g.
      - *Mutual exclusion*
        - Interactions between threads does not corrupt shared mutable data

See [en.wikipedia.org/wiki/Monitor_(synchronization)#Mutual_exclusion](en.wikipedia.org/wiki/Monitor_(synchronization)#Mutual_exclusion)
An Overview of Concurrent Programming in Java

- Java threads interact via shared objects and/or message passing
  
  **Shared objects**
  - Synchronize concurrent operations on objects to ensure certain properties, e.g.
    - Mutual exclusion
    - Coordination
      - Operations occur in the right order, at the right time, & under the right conditions

See [en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables](en.wikipedia.org/wiki/Monitor_(synchronization)#Condition_variables)
An Overview of Concurrent Programming in Java

• Java threads interact via shared objects and/or message passing

• **Shared objects**
  • Synchronize concurrent operations on objects to ensure certain properties

• Examples of Java synchronizers:
  • Synchronized statements/methods
  • Reentrant locks & intrinsic locks
  • Atomic operations
  • Semaphores
  • Condition objects
  • Barriers

See [dzone.com/articles/the-java-synchronizers](http://dzone.com/articles/the-java-synchronizers)
An Overview of Concurrent Programming in Java

• Java threads interact via shared objects and/or message passing

  • Shared objects

  • Message passing
    • Send message(s) from producer thread(s) to consumer thread(s) via a thread-safe queue

See en.wikipedia.org/wiki/Message_passing
An Overview of Concurrent Programming in Java

- Java threads interact via shared objects and/or message passing
  
  - **Shared objects**
  
  - **Message passing**
    - Send message(s) from producer thread(s) to consumer thread(s) via a thread-safe queue
      
      - Examples of Java thread-safe queues
        - Array & linked blocking queues
        - Priority blocking queue
        - Synchronous queue
        - Concurrent linked queue

See [docs.oracle.com/javase/tutorial/collections/implementations/queue.html](http://docs.oracle.com/javase/tutorial/collections/implementations/queue.html)
An Overview of Concurrent Programming Hazards
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards

See [en.wikipedia.org/wiki/Thread_safety](en.wikipedia.org/wiki/Thread_safety)
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
    - Race conditions occur when a program depends upon the sequence or timing of threads for it to operate properly

See en.wikipedia.org/wiki/Race_condition#Software
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.

- Race conditions
  - Race conditions occur when a program depends upon the sequence or timing of threads for it to operate properly

This test program induces race conditions due to lack of synchronization between producer & consumer threads accessing a bounded queue

See github.com/douglascraigschmidt/LiveLessons/tree/master/BuggyQueue
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
    - These errors occur when different threads have inconsistent views of what should be the same data

An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
    - These errors occur when different threads have inconsistent views of what should be the same data

```java
class LoopMayNeverEnd {
    boolean mDone = false;

    void work() {
        // Thread T2 read
        while (!mDone) {
            // do work
        }
    }

    void stopWork() {
        mDone = true;
        // Thread T1 write
    }

    ...
```
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
    - These errors occur when different threads have inconsistent views of what should be the same data

```java
class LoopMayNeverEnd {
    boolean mDone = false;

    void work() {
        // Thread T2 read
        while (!mDone) {
            // do work
        }
    }

    void stopWork() {
        mDone = true;
        // Thread T1 write
    }
    ...
```

Unsynchronized & mutable shared data
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
    - These errors occur when different threads have inconsistent views of what should be the same data

```java
class LoopMayNeverEnd {
    boolean mDone = false;

    void work() {
        // Thread T₂ read
        while (!mDone) {
            // do work
        }
    }

    void stopWork() {
        mDone = true;
        // Thread T₁ write
    }
    ...
```

*T₂ may never stop, even after T₁ sets mDone to true*
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
  - Deadlocks
    - Occur when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do

See en.wikipedia.org/wiki/Deadlock
An Overview of Concurrent Programming Hazards

- Java shared objects & message passing are designed to share resources safely & avoid concurrency hazards, e.g.
  - Race conditions
  - Memory inconsistencies
  - Deadlocks
    - Occur when 2+ competing threads are waiting for the other(s) to finish, & thus none ever do

$T_2$ & $T_1$ will be stuck in a “deadly embrace”

See [github.com/douglascraigschmidt/LiveLessons/tree/master/DeadlockQueue](github.com/douglascraigschmidt/LiveLessons/tree/master/DeadlockQueue)
End of Overview of How Concurrent Programs are Developed in Java