Managing the Java Thread Lifecycle
(Part 1)

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 the Java thread lifecycle & its various states

```
new MyThread()
myThread.start()
```

```
Runnable
```

```
Blocked
```

```
Waiting
```

```
Terminated
```

```
myThread.sleep()
wait(timeout)
joi(timeout)
```

```
run()
```

```
cond.wait(), cond.notifyAll()
```

```
cond.notify(), cond.notifyAll()
```

```
run() method returns
``
Learning Objectives in this Part of the Lesson

- Understand the Java thread lifecycle & its various states

The primarily focus of this overall lesson is on starting & stopping Java threads
Overview of the States in the Java Thread Lifecycle
Overview of the Java Thread Lifecycle

- A Java thread is a complex entity that interacts with many other entities.

See [docs.oracle.com/javase/8/docs/api/java/lang/Thread.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html)
Overview of the Java Thread Lifecycle

• The lifecycle of a thread must therefore be managed carefully
Overview of the Java Thread Lifecycle

- The lifecycle of a thread must therefore be managed carefully

A thread transitions through various "states" based on operations performed by a program.
Overview of the Java Thread Lifecycle

- The lifecycle of a thread must therefore be managed carefully

- Two of the most fundamental parts of a Java thread’s lifecycle involve starting & stopping it
Overview of the Java Thread Lifecycle

- You needn’t understand all the details of Java thread internals to program them effectively.

However, your reputation (& value) as a “full stack” developer will increase if you learn how threads work internally.
The State Machine for Java Threads
The State Machine for the Java Thread Lifecycle

See docs.oracle.com/javase/8/docs/api/java/lang/Thread.State.html
The State Machine for the Java Thread Lifecycle

See [www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html](http://www.uml-diagrams.org/examples/java-6-thread-state-machine-diagram-example.html)
The State Machine for the Java Thread Lifecycle

Begin by creating a new thread object

```
new MyThread()
```
The State Machine for the Java Thread Lifecycle

Transitions to the “New” state
The State Machine for the Java Thread Lifecycle

Call start() to launch the thread
The State Machine for the Java Thread Lifecycle

Transitions to the “Runnable” state
The Java & Android Linux thread scheduler controls what happens next since there may be multiple threads waiting for their chance to run.
The State Machine for the Java Thread Lifecycle

When the scheduler selects a thread to execute it transitions to the “Running” state.
The Java execution environment (e.g., JVM, Dalvik, ART, etc.) then invokes the thread’s run() hook method.
A thread can call various methods that cause it to wait for a period of time.
The State Machine for the Java Thread Lifecycle

Transition to the “Timed Waiting” state
The State Machine for the Java Thread Lifecycle

The wait time elapses or the operation completes
The State Machine for the Java Thread Lifecycle

Transitions to the “Runnable” state (i.e., it doesn’t start to run immediately)
When the scheduler selects a thread to execute it transitions to the “Running” state.
The Java execution environment then invokes the thread’s run() hook method.
A thread will block when it tries to access a “guarded resource” (e.g., a monitor lock) that is currently in use by another thread.
The State Machine for the Java Thread Lifecycle

Transitions to the “Blocked” state
The State Machine for the Java Thread Lifecycle

When the resource is released by the other thread the blocked thread will acquire it & become unblocked.
The State Machine for the Java Thread Lifecycle

Transitions to the “Runnable” state (i.e., it doesn’t start to run immediately)
When the scheduler selects a thread to execute it transitions to the "Running" state.
The Java execution environment then invokes the thread’s run() hook method
The State Machine for the Java Thread Lifecycle

A thread may call `wait()` on its monitor condition (note that the monitor lock must have already been acquired)
The State Machine for the Java Thread Lifecycle

Transitions to the “Waiting” state
When another thread calls notify() or notifyAll() the thread that is waiting will be released.
The State Machine for the Java Thread Lifecycle

Transitions to the “Runnable” state (i.e., it doesn’t start to run immediately)
The State Machine for the Java Thread Lifecycle

Transitions to the "Running" state
The Java execution environment then invokes the thread’s run() hook method.
The State Machine for the Java Thread Lifecycle

The run() method can exit either normally (by “falling off the end” of run()) or via an unhandled exception.
The State Machine for the Java Thread Lifecycle

Transitions to the “Terminated” state
The Java execution environment can then reclaim the thread’s resources
End of Managing the Java Thread Lifecycle (Part 1)
Managing the Java Thread Lifecycle

(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 Lesson

- Understand the Java thread lifecycle & its various states
- Recognize the steps involved in starting a Java thread
Steps Involved in Starting a Java Thread
Steps Involved in Starting a Java Thread

- Starting a Java thread involves interesting design & implementation issues
Steps Involved in Starting a Java Thread

- Calling `start()` on a thread triggers the execution of its `run()` hook method.
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

Different versions of Android & Java will implement these layers differently, though the key levels of abstraction are often similar.
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

Steps Involved in Starting a Java Thread

• Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

It’s important to realize that creating & starting new threads consumes a non-trivial amount of system resources, so use them judiciously!
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. `MyThread.start()`
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start() // Java method

See [libcore/luni/src/main/java/java/lang/Thread.java](https://libcore/luni/src/main/java/java/lang/Thread.java)
Steps Involved in Starting a Java Thread

• Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create() // Native method

See [libcore/luni/src/main/java/java/lang/VMThread.java](libcore/luni/src/main/java/java/lang/VMThread.java)
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
   // JNI method

See dalvik/vm/native/java_lang_VMThread.cpp
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread() // Dalvik method

See [dalvik/vm/Thread.cpp](#)
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread()
6. pthread_create(..., interpThreadStart)
   // Pthreads method

See bionic/libc/bionic/pthread.c
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread()
6. pthread_create(..., interpThreadStart)
   // Pthreads method

This is the entry point function used to transition between C & Java code
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread()
6. pthread_create(..., interpThreadStart)
7. Android Linux kernel...

See source.android.com/source/building-kernels.html
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread()
6. pthread_create(..., interpThreadStart)
7. Android Linux kernel...
8. interpThreadStart(void* arg) // Adapter

See dalvik/vm/Thread.cpp
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. MyThread.start()
2. Thread.start()
3. VMThread.create()
4. Dalvik_java_lang_VMThread_create()
5. dvmCreateInterpThread()
6. pthread_create(..., interpThreadStart)
7. Android Linux kernel...
8. interpThreadStart(void* arg)
9. dvmCallMethod(self, run, self->threadObj)
   // Dalvik method

See dalvik/vm/interp/Stack.cpp
Steps Involved in Starting a Java Thread

- Many steps occur at various layers of Java class libraries, Java execution environment, & operating system

1. `MyThread.start()`
2. `Thread.start()`
3. `VMThread.create()`
4. `Dalvik_java_lang_VMThread_create()`
5. `dvmCreateInterpThread()`
6. `pthread_create(..., interpThreadStart)`
7. `Android Linux kernel...`
8. `interpThreadStart(void* arg)`
9. `dvmCallMethod(self, run, self->threadObj)`
10. `MyThread.run()` // User-defined hook method
End of Managing the Java Thread Lifecycle (Part 2)
Managing the Java Thread Lifecycle (Part 3)

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 the Java thread lifecycle & its various states
• Recognize the steps involved in starting a Java thread
• Know various ways to stop Java threads
Learning Objectives in this Part of the Lesson

• Understand the Java thread lifecycle & its various states
• Recognize the steps involved in starting a Java thread
• Know various ways to stop Java threads
  • Stopping a thread with a volatile flag
Stopping Java Threads
Stopping Java Threads

• It may be necessary to stop a Java thread for various reasons
Stopping Java Threads

• It may be necessary to stop a Java thread for various reasons, e.g.
  • Users may want to cancel a long-running operation
    • e.g., they get bored or tired of waiting for it to complete
Stopping Java Threads

- It may be necessary to stop a Java thread for various reasons, e.g.
  - Users may want to cancel a long-running operation
  - An activity is destroyed, stopped, or paused
    - e.g., due to runtime configuration changes or pressing the “back” button

The GCD Concurrent app contains an (intentional) design flaw where it “leaks” threads when an orientation change occurs

Stopping Java Threads

• It may be necessary to stop a Java thread for various reasons, e.g.
  • Users may want to cancel a long-running operation
  • An activity is destroyed, stopped, or paused
  • Cancel other “speculative computations” after results are found
    • e.g., The ExecutorService invokeAny() method must cancel other threads after a result is found
Stopping Java Threads

• Stopping Java threads is surprisingly hard
Stopping Java Threads

• Stopping Java threads is surprisingly hard
  • i.e., the “Sorcerer’s Apprentice” problem

See www.youtube.com/watch?v=5rzyuY8-Ao8
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily

See docs.oracle.com/javase/7/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html
Stopping Java Threads

- There's no safe way to stop a Java thread involuntarily
- The stop() method is deprecated since it's inherently unsafe

See geekexplains.blogspot.com/2008/07/why-stop-suspend-resume-of-thread-are.html
Stopping Java Threads

- There’s no safe way to stop a Java thread involuntarily
  - The stop() method is deprecated since it’s inherently unsafe, e.g.
  - All locked monitors are unlocked as the exception propagates up the stack

```
Call Stack

method1()

↓ calls

method2() throws IOException {...}

↓ calls

method3() throws IOException {...}

↓ calls

method4() throw IOException {...}
```
Stopping Java Threads

• There’s no safe way to stop a Java thread involuntarily
  • The stop() method is deprecated since it’s inherently unsafe, e.g.
    • All locked monitors are unlocked as the exception propagates up the stack
  • Any objects protected by these monitors are thus left in an inconsistent state
Stopping Java Threads

- Long running operations in a thread must be coded to stop *voluntarily*!

```java
public void run()
{
    while (true) {
        // Check to see
        // if the thread
        // should stop
    }
}
```
Stopping Java Threads

- There are two ways to stop a Java thread voluntarily
Stopping Java Threads

• There are two ways to stop a Java thread voluntarily
  • Use a volatile flag

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean
        mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
...
```

See en.wikipedia.org/wiki/Volatile_variable#In_Java
Stopping Java Threads

- There are two ways to stop a Java thread voluntarily
  - Use a volatile flag
  - Use Java thread interrupt requests

Interrupts

An *interrupt* is an indication to a thread that it should stop what it is doing and do something else. It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate. This is the usage emphasized in this lesson.

A thread sends an interrupt by invoking **interrupt** on the **Thread** object for the thread to be interrupted. For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption.

See [docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html](docs.oracle.com/javase/tutorial/essential/concurrency/interrupt.html)
Stopping Java Threads with a Volatile Flag
Stopping Java Threads with a Volatile Flag

• One way to stop a Java thread is to use a “stop” flag

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean mIsStopped = true;

    public void stopMe() {
        mIsStopped = false;
    }

    public void run() {
        while (mIsStopped != true) {
            // a long-running operation
        }
    }

    ...
}
```
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.

- Add a volatile boolean flag “mIsStopped” to a class that implements Runnable

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}
```
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
  - Volatile ensures changes to a variable are consistent & visible to other threads atomically

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while (mIsStopped != true) {
            // a long-running operation
        }
    }
}
```

See [en.wikipedia.org/wiki/Volatile_variable#In_Java](en.wikipedia.org/wiki/Volatile_variable#In_Java)
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
  - Add a stopMe() method that sets “mIsStopped” to true

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean
        mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}
```
Stopping Java Threads with a Volatile Flag

- One way to stop a Java thread is to use a “stop” flag, e.g.
  - Add a volatile boolean flag “mIsStopped” to a class that implements Runnable
  - Add a stopMe() method that sets “mIsStopped” to true
  - Check “mIsStopped” periodically to see if thread’s been stopped

```java
public class MyRunnable implements Runnable {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while (mIsStopped != true) {
            // a long-running operation
        }
    }
}
```
Stopping Java Threads with a Volatile Flag

• One way to stop a Java thread is to use a “stop” flag, e.g.

  • Add a volatile boolean flag “mIsStopped” to a class that implements Runnable

  • Add a stopMe() method that sets “mIsStopped” to true

  • Check “mIsStopped” periodically to see if thread’s been stopped

  • Return from the run() method when the thread’s been stopped

  ```java
  public class MyRunnable
  implements Runnable
  {
    private volatile boolean mIsStopped = false;

    public void stopMe() {
      mIsStopped = true;
    }

    public void run() {
      while (mIsStopped != true) {
        // a long-running operation
      }
      return;
  }
  ```
Stopping Java Threads with a Volatile Flag

Although this solution is lightweight, it isn’t transparently integrated into the Java execution environment

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean
        mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    return;
```
Stopping Java Threads with a Volatile Flag

- Although this solution is lightweight, it isn’t transparently integrated into the Java execution environment
- e.g., blocking operations won’t be awakened, which impedes shutdown processing

```java
public class MyRunnable
    implements Runnable
{
    private volatile boolean mIsStopped = false;

    public void stopMe() {
        mIsStopped = true;
    }

    public void run() {
        while(mIsStopped != true) {
            // a long-running operation
        }
    }
}
```
End of Managing the Java Thread Lifecycle (Part 3)