THE EXPERT'S VOICE® IN JAVA

Java Threads and the Concurrency Utilities

Jeff Friesen

Apress®

Java Threads and the Concurrency Utilities

Jeff Friesen

Apress®

Java Threads and the Concurrency Utilities

Copyright © 2015 by Jeff Friesen

ISBN-13 (pbk): 978-1-4842-1699-6

ISBN-13 (electronic): 978-1-4842-1700-9

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc. is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781484216996. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/. Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

To my sister and her family.

Contents at a Glance

Contents

About the Author

Jeff Friesen is a freelance tutor and software developer with an emphasis on Java. In addition to authoring *Learn Java for Android Development* and co-authoring *Android Recipes*, Jeff has written numerous articles on Java and other technologies for JavaWorld (`JavaWorld.com`), informIT (`InformIT.com`), Java.net, and DevSource (`DevSource.com`). Jeff can be contacted via his web site at `TutorTutor.ca`.

About the Technical Reviewer

Sumit Pal has more than 22 years of experience in the Software Industry in various roles spanning companies from startups to enterprises. He is a big data, visualization and data science consultant and a software architect and big data enthusiast and builds end-to-end data-driven analytic systems.

Sumit has worked for Microsoft (SQL server development team), Oracle (OLAP development team) and Verizon (Big Data analytics team) in a career spanning 22 years.

Currently, he works for multiple clients advising them on their data architectures and big data solutions and does hands on coding with Spark, Scala, Java and Python. He has extensive experience in building scalable systems across the stack from middle tier, data tier to visualization for analytics applications, using Big Data, NoSQL DB. Sumit has deep expertise in Database Internals, Data Warehouses, Dimensional Modeling, Data Science with Java and Python and SQL.

Sumit has MS and BS in Computer Science.

Acknowledgments

I have many people to thank for assisting me in the development of this book. I especially thank Steve Anglin for asking me to write it and Mark Powers for guiding me through the writing process.

Introduction

Threads and the concurrency utilities are not sexy subjects, but they are an important part of non-trivial applications. This book introduces you to most of Java's thread features and concurrency utilities as of Java 8 update 60.

Chapter 1 introduces you to the `Thread` class and the `Runnable` interface. You learn how to create `Thread` and `Runnable` objects, get and set thread state, start a thread, interrupt a thread, join a thread to another thread, and cause a thread to sleep.

Chapter 2 focuses on synchronization. You learn about problems such as race conditions that cannot be solved without synchronization. You also learn how to create synchronized methods and blocks, and how to use a light version of synchronization that ignores mutual exclusion.

Chapter 3 explores the important topics of waiting and notification. I first review a small API in the `Object` class that supports these concepts, and then demonstrate this API via a producer/consumer application where one thread produces items that another thread consumes.

Chapter 4 presents three concepts that were not covered in the previous chapters. First, you learn about thread groups, which are not as useful as you might think. Then, you explore thread-local variables. Finally, you learn about the Timer Framework, which simplifies threading for timer tasks.

The previous four chapters covered low-level threading. Chapter 5 switches to a higher level by introducing the concurrency utilities, which can simplify the development of multithreaded applications and improve performance. This chapter then explores executors along with callables and futures.

Chapter 6 focuses on synchronizers (high-level synchronization constructs). You learn about countdown latches (one or more threads wait at a "gate" until another thread opens this gate, at which point these other threads can continue), cyclic barriers, exchangers, semaphores, and phasers.

Chapter 7 explores the Locking Framework, which provides interfaces and classes for locking and waiting for conditions in a manner that's distinct from an object's intrinsic lock-based synchronization and `Object`'s wait/notification mechanism. It offers improvements such as lock polling.

Finally, Chapter 8 presents additional concurrency utilities that were not covered in Chapters 5 through 7. Specifically, it introduces you to concurrent collections, atomic variables, the Fork/Join Framework, and completion services.

Each chapter ends with assorted exercises that are designed to help you master the content. Along with long answers and true/false questions, you are often confronted with programming exercises. Appendix A provides the answers and solutions.

Appendix B provides a tutorial on threading in Swing. You learn about Swing's single-threaded programming model and various APIs for avoiding problems when additional threads are used in graphical contexts. You also explore a slide show Swing application as a fun way to end this book.

Note I briefly use Java 8's lambda expression feature in some examples, but don't provide a tutorial on it. You'll need to look elsewhere for that knowledge.

Thanks for purchasing this book. I hope you find it helpful in understanding threads and the concurrency utilities.

— Jeff Friesen (October 2015)

Note You can download this book's source code by pointing your web browser to `www.apress.com/9781484216996` and clicking the Source Code tab followed by the Download Now link.

PART I

Thread APIs

CHAPTER 1

Threads and Runnables

Java applications execute via *threads*, which are independent paths of execution through an application's code. When multiple threads are executing, each thread's path can differ from other thread paths. For example, a thread might execute one of a `switch` statement's `case`s, and another thread might execute another of this statement's `case`s.

Each Java application has a *default main thread* that executes the `main()` method. The application can also create threads to perform time-intensive tasks in the background so that it remains responsive to its users. These threads execute code sequences encapsulated in objects that are known as *runnables*.

The Java virtual machine (JVM) gives each thread its own JVM stack to prevent threads from interfering with each other. Separate stacks let threads keep track of their next instructions to execute, which can differ from thread to thread. The stack also provides a thread with its own copy of method parameters, local variables, and return value.

Java supports threads primarily through its `java.lang.Thread` class and `java.lang.Runnable` interface. This chapter introduces you to these types.

Introducing Thread and Runnable

The `Thread` class provides a consistent interface to the underlying operating system's threading architecture. (The operating system is typically responsible for creating and managing threads.) A single operating system thread is associated with a `Thread` object.

The `Runnable` interface supplies the code to be executed by the thread that's associated with a `Thread` object. This code is located in `Runnable`'s `void run()` method—a thread receives no arguments and returns no value, although it might throw an exception, which I discuss in Chapter 4.

Creating Thread and Runnable Objects

Except for the default main thread, threads are introduced to applications by creating the appropriate `Thread` and `Runnable` objects. `Thread` declares several constructors for initializing `Thread` objects. Several of these constructors require a `Runnable` object as an argument.

There are two ways to create a Runnable object. The first way is to create an anonymous class that implements Runnable, as follows:

```
Runnable r = new Runnable()
             {
                @Override
                public void run()
                {
                   // perform some work
                   System.out.println("Hello from thread");
                }
             };
```

Before Java 8, this was the only way to create a runnable. Java 8 introduced the lambda expression to more conveniently create a runnable:

```
Runnable r = () -> System.out.println("Hello from thread");
```

The lambda is definitely less verbose than the anonymous class. I'll use both language features throughout this and subsequent chapters.

■ **Note** A *lambda expression* (*lambda*) is an anonymous function that's passed to a constructor or method for subsequent execution. Lambdas work with *functional interfaces* (interfaces that declare single abstract methods), such as Runnable.

After creating the Runnable object, you can pass it to a Thread constructor that receives a Runnable argument. For example, Thread(Runnable runnable) initializes a new Thread object to the specified runnable. The following code fragment demonstrates this task:

```
Thread t = new Thread(r);
```

A few constructors don't take Runnable arguments. For example, Thread() doesn't initialize Thread to a Runnable argument. You must extend Thread and override its run() method (Thread implements Runnable) to supply the code to run, which the following code fragment accomplishes:

```
class MyThread extends Thread
{
   @Override
   public void run()
   {
      // perform some work
      System.out.println("Hello from thread");
   }
}
// ...
MyThread mt = new MyThread();
```

Getting and Setting Thread State

A Thread object associates state with a thread. This state consists of a name, an indication of whether the thread is alive or dead, the execution state of the thread (is it runnable?), the thread's priority, and an indication of whether the thread is daemon or nondaemon.

Getting and Setting a Thread's Name

A Thread object is assigned a name, which is useful for debugging. Unless a name is explicitly specified, a default name that starts with the Thread- prefix is chosen. You can get this name by calling Thread's String getName() method. To set the name, pass it to a suitable constructor, such as Thread(Runnable r, String name), or call Thread's void setName(String name) method. Consider the following code fragment:

```
Thread t1 = new Thread(r, "thread t1");
System.out.println(t1.getName()); // Output: thread t1
Thread t2 = new Thread(r);
t2.setName("thread t2");
System.out.println(t2.getName()); // Output: thread t2
```

■ **Note** Thread's long getId() method returns a unique long integer-based name for a thread. This number remains unchanged during the thread's lifetime.

Getting a Thread's Alive Status

You can determine if a thread is alive or dead by calling Thread's boolean isAlive() method. This method returns true when the thread is alive; otherwise, it returns false. A thread's lifespan ranges from just before it is actually started from within the start() method (discussed later) to just after it leaves the run() method, at which point it dies. The following code fragment outputs the alive/dead status of a newly-created thread:

```
Thread t = new Thread(r);
System.out.println(t.isAlive()); // Output: false
```

Getting a Thread's Execution State

A thread has an execution state that is identified by one of the Thread.State enum's constants:

- NEW: A thread that has not yet started is in this state.
- RUNNABLE: A thread executing in the JVM is in this state.
- BLOCKED: A thread that is blocked waiting for a monitor lock is in this state. (I'll discuss monitor locks in Chapter 2.)

- `WAITING`: A thread that is waiting indefinitely for another thread to perform a particular action is in this state.
- `TIMED_WAITING`: A thread that is waiting for another thread to perform an action for up to a specified waiting time is in this state.
- `TERMINATED`: A thread that has exited is in this state.

`Thread` lets an application determine a thread's current state by providing the `Thread.State getState()` method, which is demonstrated here:

```
Thread t = new Thread(r);
System.out.println(t.getState()); // Output: NEW
```

Getting and Setting a Thread's Priority

When a computer has enough processors and/or processor cores, the computer's operating system assigns a separate thread to each processor or core so the threads execute simultaneously. When a computer doesn't have enough processors and/or cores, various threads must wait their turns to use the shared processors/cores.

■ **Note** You can identify the number of processors and/or processor cores that are available to the JVM by calling the `java.lang.Runtime` class's `int availableProcessors()` method. The return value could change during JVM execution and is never smaller than 1.

The operating system uses a *scheduler* (`http://en.wikipedia.org/wiki/Scheduling_(computing)`) to determine when a waiting thread executes. The following list identifies three different schedulers:

- Linux 2.6 through 2.6.23 uses the *O(1) Scheduler* (`http://en.wikipedia.org/wiki/O(1)_scheduler`).
- Linux 2.6.23 also uses the *Completely Fair Scheduler* (`http://en.wikipedia.org/wiki/Completely_Fair_Scheduler`), which is the default scheduler.
- Windows NT-based operating systems (such as NT, XP, Vista, and 7) use a *multilevel feedback queue scheduler* (`http://en.wikipedia.org/wiki/Multilevel_feedback_queue`). This scheduler has been adjusted in Windows Vista and Windows 7 to optimize performance.

A multilevel feedback queue and many other thread schedulers take *priority* (thread relative importance) into account. They often combine *preemptive scheduling* (higher priority threads *preempt*—interrupt and run instead of—lower priority threads) with *round robin scheduling* (equal priority threads are given equal slices of time, which are known as *time slices*, and take turns executing).

■ **Note** Two terms that are commonly encountered when exploring threads are parallelism and concurrency. According to Oracle's "Multithreading Guide" (http://docs.oracle.com/cd/E19455-01/806-5257/6je9h032b/index.html), *parallelism* is "a condition that arises when at least two threads are *executing* simultaneously." In contrast, *concurrency* is "a condition that exists when at least two threads are *making progress*. [It is a] more generalized form of parallelism that can include time-slicing as a form of virtual parallelism."

Thread supports priority via its `int getPriority()` method, which returns the current priority, and its `void setPriority(int priority)` method, which sets the priority to `priority`. The value passed to `priority` ranges from `Thread.MIN_PRIORITY` to `Thread.MAX_PRIORITY`—`Thread.NORMAL_PRIORITY` identifies the default priority. Consider the following code fragment:

```
Thread t = new Thread(r);
System.out.println(t.getPriority());
t.setPriority(Thread.MIN_PRIORITY);
```

■ **Caution** Using `setPriority()` can impact an application's portability across operating systems because different schedulers can handle a priority change in different ways. For example, one operating system's scheduler might delay lower priority threads from executing until higher priority threads finish. This delaying can lead to *indefinite postponement* or *starvation* because lower priority threads "starve" while waiting indefinitely for their turn to execute, and this can seriously hurt the application's performance. Another operating system's scheduler might not indefinitely delay lower priority threads, improving application performance.

Getting and Setting a Thread's Daemon Status

Java lets you classify threads as daemon threads or nondaemon threads. A *daemon thread* is a thread that acts as a helper to a nondaemon thread and dies automatically when the application's last nondaemon thread dies so that the application can terminate.

You can determine if a thread is daemon or nondaemon by calling Thread's `boolean isDaemon()` method, which returns `true` for a daemon thread:

```
Thread t = new Thread(r);
System.out.println(t.isDaemon()); // Output: false
```

By default, the threads associated with Thread objects are nondaemon threads. To create a daemon thread, you must call Thread's void setDaemon(boolean isDaemon) method, passing true to isDaemon. This task is demonstrated here:

```
Thread t = new Thread(r);
t.setDaemon(true);
```

■ **Note** An application will not terminate when the nondaemon default main thread terminates until all background nondaemon threads terminate. If the background threads are daemon threads, the application will terminate as soon as the default main thread terminates.

Starting a Thread

After creating a Thread or Thread subclass object, you start the thread associated with this object by calling Thread's void start() method. This method throws java.lang.IllegalThreadStateException when the thread was previously started and is running or when the thread has died:

```
Thread t = new Thread(r);
t.start();
```

Calling start() results in the runtime creating the underlying thread and scheduling it for subsequent execution in which the runnable's run() method is invoked. (start() doesn't wait for these tasks to be completed before it returns.) When execution leaves run(), the thread is destroyed and the Thread object on which start() was called is no longer viable, which is why calling start() results in IllegalThreadStateException.

I've created an application that demonstrates various fundamentals from thread and runnable creation to thread starting. Check out Listing 1-1.

Listing 1-1. Demonstrating Thread Fundamentals

```
public class ThreadDemo
{
   public static void main(String[] args)
   {
      boolean isDaemon = args.length != 0;
      Runnable r = new Runnable()
                   {
                      @Override
                      public void run()
                      {
                         Thread thd = Thread.currentThread();
                         while (true)
```