Advanced Java T.T.
API Thread Control

Simon Roberts

WELCOME

Mark Your Attendance:

- ➢ Go to the Course Calendar Invite
- ➢ Click on the Course Event Link
- ➢ Press Check-In

Join Us in Making Learning Technology Easier

Our mission…

Over 16 years ago, we embarked on a journey to improve the world by making learning technology easy and accessible to everyone.

…impacts everyone daily.

And it's working. Today, we're known for delivering customized tech learning programs that drive innovation and transform organizations.

In fact, when you talk on the phone, watch a movie, connect with friends on social media, drive a car, fly on a plane, shop online, and order a latte with your mobile app, you are experiencing the impact of our solutions.

Over The Past Few Decades, We've Provided

Over **62,300,000** expert-led learning hours

In 2019 Alone, We Provided

Training to over **13,500** engineers

Over **120** active trainers, with an average of over two decades of experience each.

Programs in **30** countries

Upskilling and Reskilling Offerings

Develop Intelligence

Intimately customized learning experiences just for your teams.

Workshop	2-3 day upskilling experiences
Fast Track	5-day reskilling experiences
Learning Spike	1-day technology overviews
Target Topics	90-minute instructor-led micro-learnings
Hack-a-thon	Learn and build an MVP in 2-3 days

BACK END DEVELOPMENT

BIG DATA

CLOUD COMPUTING

DEVOPS

FRONT END DEVELOPMENT

MACHINE LEARNING

MOBILE APP DEVELOPMENT

SOFTWARE ENGINEERING

SYSTEM ADMINISTRATION

Ruby · Jenkins · akka · ANGULAR · ANSIBLE · APACHE Spark

aws · Azure · cassandra · CHEF · docker

GO · GraphQL · iOS · Java · JS

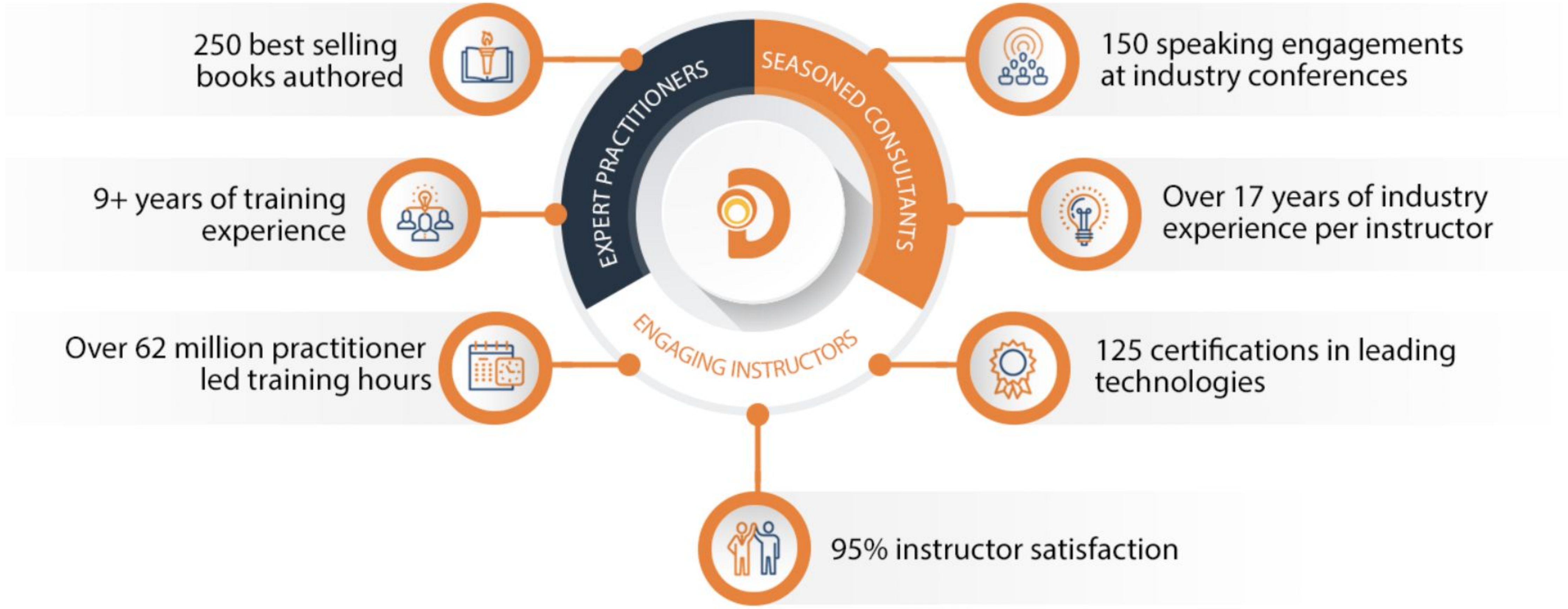
kafka · mongoDB · MySQL · node js · puppet · python

React Native · redis · Scala · spring

TS

AND MANY OTHER TRENDING TECHNOLOGIES

World Class Practitioners



EXPERT PRACTITIONERS
SEASONED CONSULTANTS
ENGAGING INSTRUCTORS
250 best selling books authored
9+ years of training experience
Over 62 million practitioner led training hours
150 speaking engagements at industry conferences
Over 17 years of industry experience per instructor
125 certifications in leading technologies
95% instructor satisfaction

Recording Policy

Recordings are provided to participants who have attended the training, in its entirety. Recordings are provided as a value-add to your training, and should not be utilized as a replacement to the classroom experience.

Participants can expect the following:

- Recordings will be provided the Monday after class is completed
- Recordings will be provided for 14 days
- Recordings will be accessible as “View Only” status and cannot be copied
- Sharing recordings is strictly prohibited

To request recordings, please fill out the form linked in Learn++.

Thank you for your understanding and adhering to the policy.

Note About Virtual Trainings

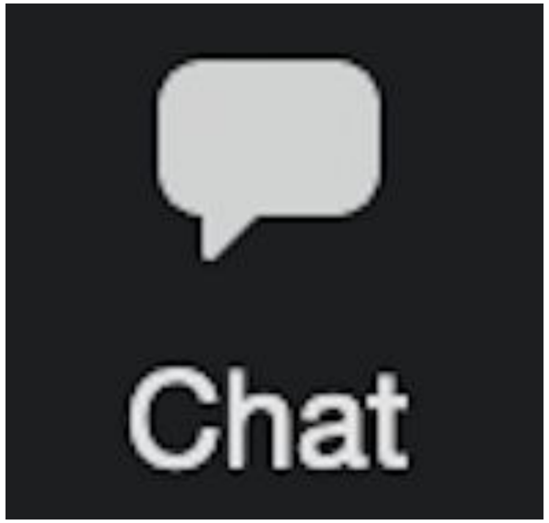
What we want

...what we've got

Virtual Training Expectations for You

Arrive on time / return on time

Mute unless speaking



Use chat or ask questions verbally

Virtual Training Expectations for Me

I pledge to:

- Make this as interesting and interactive as possible
- Ask questions in order to stimulate discussion
- Use whatever resources I have at hand to explain the material
- Try my best to manage verbal responses so that everyone who wants to speak can do so
- Use an on-screen timer for breaks so you know when to be back

Prerequisites

- Good understanding of the Java programming language to Java 8
- Basic understanding of creating threads in Java

Objectives

At the end of this course you will be able to:

- Create and use thread pools
- Ensure the orderly shutdown of threads
- Use ReentrantLock
- Use StampedLock
- Use Semaphore

About you

In 90 seconds!

- What you hope to learn
- What your background level is

THANK YOU

Thread pool goals

Creating a thread is a non-trivial task at the OS level.

Rather than running one task and dying, a thread can execute multiple `Runnable` tasks sequentially, and those tasks can be pulled from a queue structure (typically a `BlockingQueue`)

If a number of worker threads are attached to a single `BlockingQueue`, the result is known as a *thread-pool*.

Thread pools are provided by the Java APIs, implementing the interfaces `Executor`, and/or `ExecutorService`.

Using `ExecutorService`

Create in various ways, commonly using factories in the `Executors` class

Send work to the pool using either `submit` or `execute` methods

The `submit` method accepts an instance of `Callable<E>`, similar to `Runnable`, but returns a value (of type `E`), and can throw checked exceptions. (Can also submit a `Runnable`)

The `submit` method returns a `Future<E>`, which is a handle on the submitted job.

Using `Future<E>`

The `isDone` method polls to see if the submitted task has completed

The `isCancelled` method polls to see if the submitted task was canceled

The `get` methods (an overload provides a timeout) block until the job is completed, and returns the result of that job, or throws an `ExecutionException` if the job threw an exception

- `ExecutionException` has the job's exception embedded as the "cause".

The `cancel` method attempts to cancel the job, this might remove it from the input queue, or if it has started can send an interrupt to ask the job to shutdown (controlled by a boolean argument).

Thread shutdown

Threads should not be killed from outside, they might hold locks, or have data or devices in transactionally unsafe states.

Instead, they should be sent a message requesting the thread clean up and shut itself down.

The conventional notification is to send an interrupt; interrupts should not be used for other purposes.

Library code should clean up that method call, and perhaps that library, but rethrow the InterruptedException to the caller, so that the business logic can shut itself down cleanly too.

ReentrantLock

The `synchronized/wait/notify` mechanism has several significant issues:

- The only way forward from a `synchronized` call is successfully obtaining the lock. This can cause problems shutting down threads.
- Similarly, there is no timeout on a call to `synchronized`
- There is only one "condition variable" that a thread can block (wait) on, this is rarely enough to model any real situation.
- Using `notifyAll` to solve the previous issue impedes scalability.

The `ReentrantLock` API addresses these problems.

Using `ReentrantLock`

Locks must be released reliably in all situations. Use a `try/finally` structure to achieve this:

```
lock.lock();
try {
    // …
} finally {
    lock.unlock();
}
```

This ensures lock release even in the face of unhandled exceptions or premature return.

Timing with `ReentrantLock`

Create one or more rendezvous objects using `aLock.newCondition()`

While holding the lock call `aCondition.signal()` or `aCondition.await()`

Behavior is parallel to `notify()` and `wait()` of `Object`

StampedLock

StampedLock allows us to build access control based on three modes

- Exclusive (write) lock
- Non exclusive (read) lock
- Optimistic lock

Lock using: `[write|read]Lock()`, `[write|read]LockInterruptibly()`, `try[Write|Read|OptimisticRead]Lock()`. These return a `long` value identifying the lock obtained, or zero if the attempt failed.

Release the lock using `unlock[Write|Read](lock)`, verify optimistic lock using `validate(lock)`

Semaphore

`Semaphore` is a classic mutual exclusion construct. It often serves better as a resource counter.

Can be tricky to use for mutual exclusion as it is not reentrant.

`sem.acquire()` attempts to decrement the counter of the semaphore, if the counter would reduce below zero, then the thread is blocked until another thread causes a sufficient increase in the count.

`sem.release()` increments the counter of the semaphore, possibly releasing one or more threads blocked on acquire calls.

`try/finally` constructs can be used to ensure `release()` is called reliably.