Java Vulnerabilities Meeting
5 July 2026

Attendees: Erhard Ploedereder
Tullio Vardenaga
Larry Wagoner
Sean

Edited WG 23 N 1576 to be submittable to SCC 22 for registration and CD Ballot.

Result of edits is N1586.

The following are the chat notes.

CHAT:

2026-08-05 14:13:55 From Sean to Everyone:
Even if you call cancel(true), a running task can still refuse to terminate if its code ignores the interrupt signal or does not check Thread.currentThread().isInterrupted()

2026-08-05 14:16:10 From Sean to Everyone:
Unstarted Tasks: If the task is still waiting in the queue and has not been assigned to a thread, cancel() removes or cancels it immediately. It will never run.
Executing Tasks: If the task is already running, the behavior depends on the boolean flag you pass to cancel(mayInterruptIfRunning):cancel(false): The running task is allowed to finish. It can completely refuse to terminate early.cancel(true): The thread executing the task is sent an interrupt signal (Thread.interrupt()).

2026-08-05 14:18:33 From Sean to Everyone:
boolean cancel(boolean mayInterruptIfRunning)
Attempts to cancel execution of this task. This attempt will fail if the task has already completed, has already been cancelled, or could not be cancelled for some other reason. If successful, and this task has not started when cancel is called, this task should never run. If the task has already started, then the mayInterruptIfRunning parameter determines whether the thread executing this task should be interrupted in an attempt to stop the task.
After this method returns, subsequent calls to isDone() will always return true. Subsequent calls to isCancelled() will always return true if this method returned true.

2026-08-05 14:18:44 From Sean to Everyone:
From: https://docs.oracle.com/javase/8/docs/api/java/util/

concurrent/Future.html#cancel-boolean-

2026-08-05 14:19:18 From Sean to Everyone:
	Parameters:
	mayInterruptIfRunning - true if the thread executing this task should be interrupted; otherwise, in-progress tasks are allowed to complete
	Returns:
	false if the task could not be cancelled, typically because it has already completed normally; true otherwise

2026-08-05 14:54:10 From Sean to Everyone:
	This needs verified and I am looking into it:

	In Java concurrency, a Task (represented by Runnable or Callable) does not have its own intrinsic lock and cannot be synchronized directly like an object or a method.
	Instead, tasks rely on the Thread executing them or the ExecutorService managing them to handle locking and synchronization.
	How Synchronization Applies to Tasks
	Inside the Task: You can use synchronized blocks inside a task's run() or call() method, but you must lock on a specific shared object, not the task itself.
	The Task Object: While you can theoretically write synchronized(taskInstance), this only locks that specific task wrapper, which rarely protects the actual shared data.
	Thread Pools: When using an ExecutorService, multiple worker threads pull tasks from a queue. Synchronization is needed to protect the shared data those tasks modify, not the tasks themselves.

2026-08-05 14:55:43 From Sean to Everyone:
	https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html

2026-08-05 14:57:40 From Sean to Everyone:
	https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html

2026-08-05 14:59:01 From Sean to Everyone:
	This reinforces Tullio's previous stance

2026-08-05 15:02:02 From Sean to Everyone:
	In 6.59 we state:
	The Java ExecutorService is a framework that aims to simplify the execution of tasks in asynchronous mode. It is intended to relieve the developer from doing direct thread management by separating thread management and creation from the rest of the application. Since tasks are executed by threads in a thread-pool, attempts to use Java’s thread synchronization mechanisms inside tasks can result in deadlock, as discussed in 6.63.

2026-08-05 15:14:02 From Sean to Everyone:
	I am seeing a problem with the last sentence ... still verifying...

	In standard Java, non-volatile reads and writes of 64-bit primitive types (long and double) are allowed to be split into two distinct 32-bit operations. This can lead to word tearing (data corruption) if one thread overwrites the first 32 bits and another thread overwrites the remaining 32 bits concurrently.

2026-08-05 15:15:33 From Sean to Everyone:
	Reads and writes are atomic for reference variables and for most primitive variables (all types except long and double).
	Ref: https://docs.oracle.com/javase/tutorial/essential/concurrency/atomic.html

2026-08-05 15:16:04 From Sean to Everyone:
	Reads and writes are atomic for all variables declared volatile (including long and double variables).

2026-08-05 15:35:18 From Sean to Everyone:
	I stand corrected, this is stated correctly and is OK as is. I misread it's meaning. Disregard.

2026-08-05 16:09:48 From Sean to Everyone:
	boolean cancel(boolean mayInterruptIfRunning)
	Attempts to cancel execution of this task. This method has no effect if the task is already completed or cancelled, or could not be cancelled for some other reason. Otherwise, if this task has not started when cancel is called, this task should never run. If the task has already started, then the mayInterruptIfRunning parameter determines whether the thread executing this task (when known by the implementation) is interrupted in an attempt to stop the task.
	The return value from this method does not necessarily indicate whether the task is now cancelled; use isCancelled().
	Parameters:
	mayInterruptIfRunning - true if the thread executing this task should be interrupted (if the thread is known to the implementation); otherwise, in-progress tasks are allowed to complete
	Returns:
	false if the task could not be cancelled, typically because it has already completed; true otherwise. If two or more threads cause a task to be cancelled, then at least one of them returns true. Im

2026-08-05 16:09:53 From Sean to Everyone:
	https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/concurrent/Future.html#cancel(boolean)

2026-08-05 16:42:03 From Sean to Everyone:

Two or more tasks can be assigned to the same thread in Java. While a single thread can only execute one instruction at a exact physical millisecond, you can assign multiple tasks to it to be executed either sequentially or concurrently through cooperative switching.

OPTIONS:

1. Sequential Execution (One after another)

The simplest way is to invoke multiple tasks inside the thread's run() method. The thread will complete the first task entirely before moving on to the second task.

2. Using a Single-Threaded Executor Pool

In production, you usually manage this via Java's Concurrency API using Executors.newSingleThreadExecutor(). This creates a single worker thread backed by an unbounded queue. You can submit multiple tasks (Runnable or Callable) to it; they will line up in the queue and execute sequentially on that exact same thread