The following notes pertain to the comment in **6.62.2 Avoidance mechanisms for language users** from last meeting … "*Under discussion 8 October 2025. Continue*"**.** There is no action required in my opinion and an example is included below that illustrates a vulnerability associated with `ThreadGroup`, and then a better approach using `ExecutorService` and `Executor` frameworks

API Note pertaining to ThreadGroup:

Ref: https://docs.oracle.com/en/java/javase/22/docs/api//java.base/java/lang/ThreadGroup.html

"Thread groups provided a way in early Java releases to group threads and provide a form of job control for threads. Thread groups supported the isolation of applets and defined methods intended for diagnostic purposes. ***It should be rare for new applications to create ThreadGroups and interact with this API****."*

Java **ThreadGroup** *deprecated* functions:

ref: https://docs.oracle.com/en/java/javase/24/docs/api/deprecated-list.html

1. **java.lang.ThreadGroup.checkAccess()**
   Deprecated in: 17
   This method originally determined if the currently running thread had permission to modify this thread group. This method was only useful in conjunction with the Security Manager, which is no longer supported. There is no replacement for the Security Manager or this method.

2. **java.lang.ThreadGroup.destroy()**
   Deprecated in: 16
   This method was originally specified to destroy an empty thread group. The ability to explicitly destroy a thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

3. **java.lang.ThreadGroup.isDaemon()**
   Deprecated in: 16
   This method originally indicated if the thread group is a *daemon thread group* that is automatically destroyed when its last thread terminates. The concept of daemon thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

4. **java.lang.ThreadGroup.isDestroyed()**
   Deprecated in: 16
   This method originally indicated if the thread group is destroyed. The ability to destroy a thread group and the concept of a destroyed thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

5. **java.lang.ThreadGroup.setDaemon(boolean)**

Deprecated in: 16
This method originally configured whether the thread group is a *daemon thread group* that is automatically destroyed when its last thread terminates. The concept of daemon thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

6. **java.lang.ThreadGroup.checkAccess()**
   Deprecated in: 17
   This method originally determined if the currently running thread had permission to modify this thread group. This method was only useful in conjunction with the Security Manager, which is no longer supported. There is no replacement for the Security Manager or this method.

7. **java.lang.ThreadGroup.destroy()**
   Deprecated in: 16
   This method was originally specified to destroy an empty thread group. The ability to explicitly destroy a thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

8. **java.lang.ThreadGroup.isDaemon()**
   Deprecated in: 16
   This method originally indicated if the thread group is a *daemon thread group* that is automatically destroyed when its last thread terminates. The concept of daemon thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

9. **java.lang.ThreadGroup.isDestroyed()**
   Deprecated in: 16
   This method originally indicated if the thread group is destroyed. The ability to destroy a thread group and the concept of a destroyed thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

10. **java.lang.ThreadGroup.setDaemon(boolean)**
    Deprecated in: 16
    This method originally configured whether the thread group is a *daemon thread group* that is automatically destroyed when its last thread terminates. The concept of daemon thread group no longer exists. A thread group is eligible to be GC'ed when there are no live threads in the group and it is otherwise unreachable.

java.lang.ThreadGroup is a legacy class and *should not* be used in modern Java development due to various security and reliability issues. The primary vulnerability is a Time-of-Check, Time-of-Use (TOCTOU) *race condition* when performing actions on a group of threads.

A TOCTOU vulnerability occurs when the state of an object is checked at one time, but changes before the action based on that check is executed, leading to a race condition. In the case of `ThreadGroup`, a program might perform the following steps:

1. **Time-of-Check:** Get the number of active threads in a group using `activeCount()`.
2. **Time-of-Use:** Allocate an array based on the retrieved count and populate it with thread references using `enumerate()`.
3. **Vulnerability:** If new threads are added to the group between steps 1 and 2, the array will be too small and the program will not operate on all the threads, leading to an incomplete or failed operation.

Example: The interrupted thread vulnerability

The following example demonstrates a TOCTOU vulnerability where the goal is to interrupt all threads within a `ThreadGroup`.

Vulnerable code:

```
import java.util.concurrent.TimeUnit;

public class VulnerableThreadGroup {

    public static void main(String[] args) throws InterruptedException {
        // Create a ThreadGroup for our worker threads
        ThreadGroup workerGroup = new ThreadGroup("workerGroup");

        // Start two initial worker threads
        new Worker("Worker-1", workerGroup).start();
        new Worker("Worker-2", workerGroup).start();

        // Let threads run for a short time
        TimeUnit.SECONDS.sleep(1);

        // --- Vulnerable Sequence: TOCTOU Race Condition ---

        // Time-of-Check: Get the current number of active threads.
        // Assume this number is 2.
        int activeThreads = workerGroup.activeCount();
        System.out.println("Main thread sees " + activeThreads + " active threads.");

        // An attacker (or another thread) could now inject a new thread into the group.
        // This simulates the race condition.
        new Worker("Attacker-Injected-Worker", workerGroup).start();

        // Time-of-Use: Enumerate the threads based on the *old* count.
        // This array will be too small to hold the new thread.
        Thread[] threads = new Thread[activeThreads];
        workerGroup.enumerate(threads);

        System.out.println("Main thread found " + threads.length + " threads to interrupt.");

        // Interrupt the threads found in the array.
        for (Thread t : threads) {
```

```
            if (t != null) {
                t.interrupt();
                System.out.println("Interrupted thread: " + t.getName());
            }
        }

        System.out.println("--- Race condition in progress ---");

        // Wait to see which threads were actually interrupted.
        TimeUnit.SECONDS.sleep(2);

        // Check the final state. The new thread is still running.
        System.out.println("\nFinal check:");
        System.out.println("Active threads in group after interrupt: " +
workerGroup.activeCount());
    }
}

class Worker extends Thread {
    public Worker(String name, ThreadGroup group) {
        super(group, name);
    }

    @Override
    public void run() {
        try {
            System.out.println(getName() + " started.");
            while (!isInterrupted()) {
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch (InterruptedException e) {
            System.out.println(getName() + " was interrupted.");
        }
    }
}
```

Output:

The output shows that the "Attacker-Injected-Worker" is not interrupted, even though the main thread attempted to interrupt all threads in the group.

```
Worker-1 started.
Worker-2 started.
Main thread sees 2 active threads.
Attacker-Injected-Worker started.
Main thread found 2 threads to interrupt.
Interrupted thread: Worker-1
Interrupted thread: Worker-2
--- Race condition in progress ---
Worker-1 was interrupted.
Worker-2 was interrupted.

Final check:
Active threads in group after interrupt: 1
```

Secure Alternative: ExecutorService

Modern Java concurrency utilities, particularly those in the java.util.concurrent package, offer robust and secure alternatives to ThreadGroup. The `ExecutorService` and `Executor` frameworks manage a pool of threads securely.

```
import java.util.concurrent.*;

public class SecureExecutorService {

    public static void main(String[] args) throws InterruptedException {
        // Create a secure thread pool (ExecutorService)
        ExecutorService executor = Executors.newFixedThreadPool(2);

        // Submit tasks to the executor
        Future<?> future1 = executor.submit(new SecureWorker("Worker-1"));
        Future<?> future2 = executor.submit(new SecureWorker("Worker-2"));

        // Let threads run for a short time
        TimeUnit.SECONDS.sleep(1);

        System.out.println("Main thread knows about two active tasks.");

        // It's not possible for an "attacker" thread to be injected into this pool
        // without going through the secure submission process.

        System.out.println("Attempting to shut down the executor gracefully...");
        executor.shutdownNow(); // Interrupts all running tasks

        // Check the final state
        if (executor.awaitTermination(3, TimeUnit.SECONDS)) {
            System.out.println("\nAll tasks terminated.");
        } else {
            System.out.println("\nSome tasks are still running.");
        }
    }
}

class SecureWorker implements Runnable {
    private String name;

    public SecureWorker(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        try {
            System.out.println(name + " started.");
            while (!Thread.currentThread().isInterrupted()) {
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch (InterruptedException e) {
            System.out.println(name + " was interrupted.");
        }
    }
}
```

Expected output

The output demonstrates that the `shutdownNow()` call correctly and reliably interrupts all threads managed by the executor. There is no TOCTOU vulnerability because the ExecutorService manages the threads internally and atomically.

```
Worker-1 started.
Worker-2 started.
Main thread knows about two active tasks.
Attempting to shut down the executor gracefully...
Worker-2 was interrupted.
Worker-1 was interrupted.
```