Thread Interruption and Termination in Java

Satyendra Jaiswal · Follow

3 min read · Dec 6, 2023

Multithreading is a powerful concept in Java, enabling concurrent execution of tasks for improved performance. However, managing threads effectively, especially when it comes to interruption and termination, is crucial for building robust applications. In this article, we will delve into the intricacies of interrupting a thread and explore various scenarios to understand when and how to terminate threads gracefully.

Understanding Thread Interruption:

In Java, interrupting a thread involves setting a flag that suggests the thread should stop execution. This mechanism is particularly useful when a long-running task needs to be aborted or when shutting down a multi-threaded application gracefully. Let's start by exploring the basics.

Basic Thread Interruption:

Consider the following scenario where a thread is performing a time-consuming task:

```java
public class InterruptExample extends Thread {
    public void run() {
        try {
            while (!Thread.interrupted()) {
                // Perform a time-consuming task
                System.out.println("Working...");
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            // Handle interruption gracefully
            System.out.println(Thread.currentThread().getName() + " Thread interrupted!");
        }
    }

    public static void main(String[] args) {
        InterruptExample thread = new InterruptExample();
        thread.start();

        // Allow the thread to work for some time
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Interrupt the thread
        thread.interrupt();
    }
}
```

InterruptExample.java hosted with ❤ by GitHub view raw

In this example, the `run` method contains a loop that performs a time-consuming task until the thread is interrupted. The `main` method starts the thread, allows it to work for some time, and then interrupts it.

Handling Interruption:

When a thread is interrupted, it throws an `InterruptedException`. It's essential to catch this exception and handle it appropriately. In the example above, the `catch` block outputs a message indicating that the thread has been interrupted.

Advanced Thread Termination:

While the basic interruption mechanism works well for simple scenarios, more complex applications may require a nuanced approach to thread termination. Let’s explore advanced techniques for terminating threads based on different scenarios.

Scenario 1: Using a Shared Flag for Cooperation:

Consider a scenario where multiple threads are performing tasks, and we want to gracefully shut down the entire application. We can use a shared flag to signal threads to terminate:

```java
public class SharedFlagTermination {
    private static volatile boolean shutdownRequested = false;

    public static void main(String[] args) {
        // Start multiple threads
        Thread thread1 = new WorkerThread();
        Thread thread2 = new WorkerThread();

        thread1.start();
        thread2.start();

        // Allow threads to work for some time
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Request shutdown
        shutdownRequested = true;

        // Interrupt threads
        thread1.interrupt();
        thread2.interrupt();
    }

    static class WorkerThread extends Thread {
        public void run() {
            while (!shutdownRequested) {
                // Perform tasks
                System.out.println("Working...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // Handle interruption if needed
                }
            }
            System.out.println("Thread terminated gracefully.");
        }
    }
}
```

In this example, the `WorkerThread` class performs tasks within a loop. The `shutdownRequested` flag is used to signal the threads to terminate gracefully. When the flag is set, the threads complete their current tasks and terminate.

Scenario 2: Using ExecutorService for Controlled Termination:

Java provides the `ExecutorService` framework for managing thread execution. Using this framework allows for controlled termination of threads:

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

public class ExecutorServiceTermination {
    public static void main(String[] args) {
        // Create an ExecutorService with a fixed thread pool
        ExecutorService executorService = Executors.newFixedThreadPool(2);

        // Submit tasks to the pool
        for (int i = 0; i < 5; i++) {
            executorService.submit(new WorkerTask());
        }

        // Allow tasks to work for some time
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Shut down the ExecutorService
        executorService.shutdown();

        // Attempt to interrupt any remaining tasks
        try {
            if (!executorService.awaitTermination(3, TimeUnit.SECONDS)) {
                // If tasks are not terminated after the specified time, interrupt them
                executorService.shutdownNow();
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    static class WorkerTask implements Runnable {
        public void run() {
            while (!Thread.interrupted()) {
                // Perform tasks
                System.out.println(Thread.currentThread().getName() + " Working...");
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // Handle interruption if needed
                    Thread.currentThread().interrupt(); // Restore interrupted status
                }
            }
            System.out.println(Thread.currentThread().getName() + " Task terminated gracefully.")
        }
    }
}
```

ExecutorServiceTermination.java hosted with ❤ by GitHub view raw

In this example, if the `awaitTermination` method does not return `true` after the specified time, it means that not all tasks have completed. In that case, we call `shutdownNow` to attempt to interrupt the remaining tasks and forcefully shut down the `ExecutorService`. Additionally, the `Thread.currentThread().interrupt()` statement inside the `catch` block in the `WorkerTask` class ensures that the interrupted status is properly restored.

Conclusion:

Mastering thread interruption and termination is essential for building reliable and responsive multithreaded applications in Java. Understanding the basic interruption mechanism and employing advanced techniques for controlled termination ensures that your application can gracefully handle scenarios ranging from simple thread interruption to complex, multi-threaded shutdown procedures. Choose the approach that best fits your application's requirements, and always prioritize clean and efficient thread management for optimal performance.

Support my work: Buy Me a Coffee

Threading In Java · Executorservice · Java · Coding · Interview Questions

Written by Satyendra Jaiswal

230 Followers · 206 Following

Follow

https://www.linkedin.com/in/satyendrajaiswal/

Responses (1)

Mohan

What are your thoughts?

David
Sep 4, 2024

InterruptedException is only thrown when the thread is sleeping. Your code only works because your threads are sleeping. If they where doing actual work, this code wouldn't interrupt gracefully.

2 · 1 reply · Reply

More from Satyendra Jaiswal

Satyendra Jaiswal

Understanding Spring IoC and Dependency Injection

Spring is one of the most popular frameworks in the Java ecosystem, primarily known for it…

Jun 25, 2024 30

Satyendra Jaiswal

Understanding @Configuration , @ComponentScan and…

Spring Boot simplifies the process of building robust, scalable applications by providing…

Jun 24, 2024 55

Satyendra Jaiswal

Caching Strategies for APIs: Improving Performance and…

In the dynamic landscape of web development, where responsiveness and…

Nov 25, 2023 55

Satyendra Jaiswal

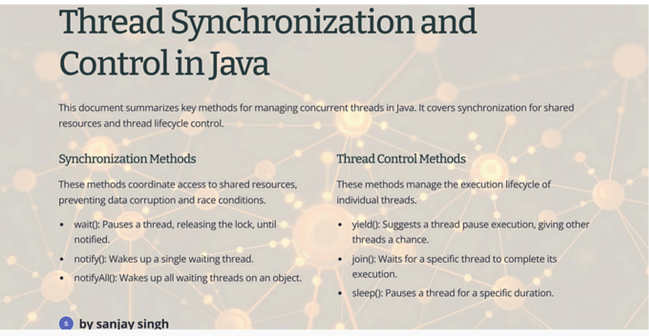
Securing APIs: OAuth 2.0 and API Keys Best Practices

In today's interconnected digital landscape, APIs (Application Programming Interfaces)…

Nov 23, 2023 6

See all from Satyendra Jaiswal

Recommended from Medium



Thread Synchronization and Control in Java

This document summarizes key methods for managing concurrent threads in Java. It covers synchronization for shared resources and thread lifecycle control.

Synchronization Methods

These methods coordinate access to shared resources, preventing data corruption and race conditions.

- wait(): Pauses a thread, releasing the lock, until notified.
- notify(): Wakes up a single waiting thread.
- notifyAll(): Wakes up all waiting threads on an object.

Thread Control Methods

These methods manage the execution lifecycle of individual threads.

- yield(): Suggests a thread pause execution, giving other threads a chance.
- join(): Waits for a specific thread to complete its execution.
- sleep(): Pauses a thread for a specific duration.

by sanjay singh

Sanjay Singh

Understanding wait(), notify(), and notifyAll() vs yield(), join(), and...

Thread Synchronization Methods

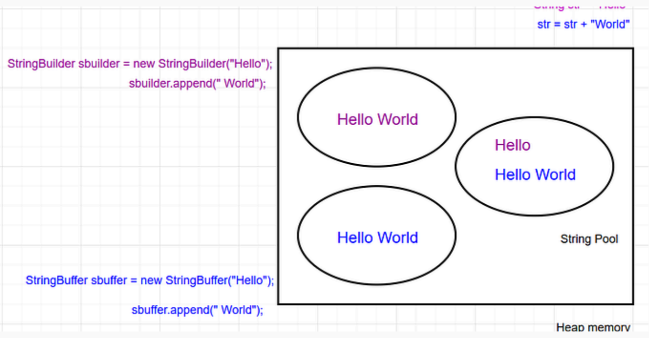
✦ Oct 19, 2024 👏 15

Hui

Stop Writing != null Everywhere! Here's a Smarter Way

Null pointer exceptions are a familiar pain point in Java development. The go-to solutio...

✦ Apr 3 👏 472 💬 14

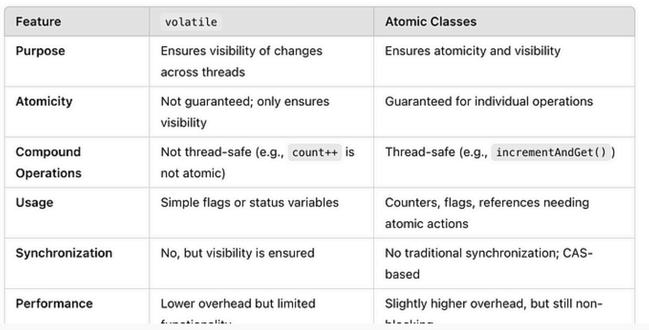

str = str + "World"
StringBuilder sbuilder = new StringBuilder("Hello");
sbuilder.append(" World");
Hello World
Hello
Hello World
Hello World
String Pool
StringBuffer sbuffer = new StringBuffer("Hello");
sbuffer.append(" World");
Heap memory

Dharshitha Senevirathne

The Difference Between String, StringBuffer, and StringBuilder in...

This is one of the most common questions asked in Java interviews. In this article, I will...

✦ Oct 19, 2024 👏 51



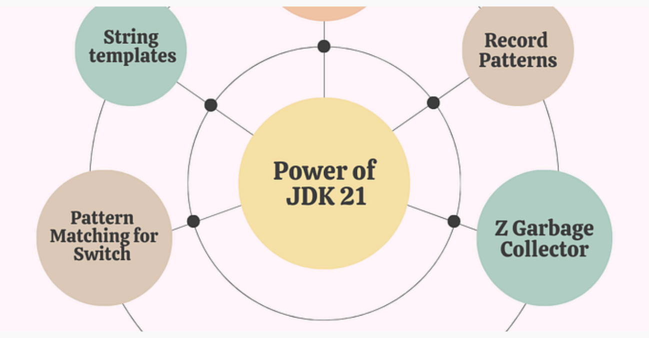
Feature	`volatile`	Atomic Classes
Purpose	Ensures visibility of changes across threads	Ensures atomicity and visibility
Atomicity	Not guaranteed; only ensures visibility	Guaranteed for individual operations
Compound Operations	Not thread-safe (e.g., `count++` is not atomic)	Thread-safe (e.g., `incrementAndGet()`)
Usage	Simple flags or status variables	Counters, flags, references needing atomic actions
Synchronization	No, but visibility is ensured	No traditional synchronization; CAS-based
Performance	Lower overhead but limited functionality	Slightly higher overhead, but still non-blocking

Alice Dai

Java — volatile vs atomic classes

In Java, both volatile and atomic classes (e.g., AtomicInteger, AtomicBoolean,...

Nov 3, 2024 👏 2



Umadevi R

Exploring Java's Latest Features: Unlocking the Power of JDK 21

Introduction

✦ Oct 18, 2024 👏 3 💬 1

In Pharos Pr... by Dmytro Nasyrov | Pharos Produ...

Garbage Collection in Java

What is garbage collection?

Nov 9, 2024 👏 3

See more recommendations