Nadege Gaju

Thread Pools in Java

1. Thread Concepts

1.1 What is a Thread?

A thread in Java is the smallest unit of execution within a process. A process can contain multiple threads that share the process's resources but execute independently. Threads are used to perform multiple tasks concurrently within a program, allowing for efficient use of CPU resources.

1.2 Creating and Managing Threads

1.2.1 Using the Runnable Interface

The Runnable interface is a functional interface that defines a single method run(). Implementing Runnable is a common way to define the code that a thread should execute.

```
public class RunnableExample {
    public static void main(String[] args) {
        Runnable task = () -> {
            for (int i = 1; i <= 5; i++) {
                System.out.println("Runnable Task - " + i);
            }
        };

        Thread thread = new Thread(task);
        thread.start();
    }
}
```

Explanation: In this example, a Runnable task is defined using a lambda expression. The task is passed to a Thread object, and the thread is started using the start() method.

1.2.2 Extending the Thread Class

Another way to create a thread is by extending the Thread class and overriding its run() method.

```
public class ThreadExample extends Thread {
    @Override
    public void run() {
        for (int i = 1; i <= 5; i++) {
            System.out.println("Thread Task - " + i);
        }
    }

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

Explanation: In this example, the ThreadExample class extends the Thread class and overrides the run() method to define the task to be executed by the thread.

1.3 Thread Life Cycle

A thread in Java goes through several states in its lifecycle:

- **NEW:** A thread that has been created but not yet started.
- **RUNNABLE:** A thread that is ready to run or is currently running.
- **BLOCKED:** A thread that is blocked and waiting for a monitor lock.
- **WAITING:** A thread that is waiting indefinitely for another thread to perform a particular action.
- **TIMED_WAITING:** A thread that is waiting for another thread to perform an action for up to a specified waiting time.
- **TERMINATED:** A thread that has completed its execution.

```
public class ThreadLifeCycleExample {
    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(() -> System.out.println("Thread is running"));

        System.out.println("Thread state before start: " + thread.getState());
        thread.start();
        System.out.println("Thread state after start: " + thread.getState());
        thread.join(); // Wait for the thread to finish
        System.out.println("Thread state after completion: " + thread.getState());
    }
}
```

Explanation: This example shows how to track a thread's state before and after it starts and after it finishes.

1.4 Thread Synchronization

When multiple threads access shared resources simultaneously, there is a risk of data inconsistency. Java provides synchronization mechanisms to prevent such issues.

Synchronized Methods

A method can be synchronized by using the synchronized keyword. This ensures that only one thread can execute the method at a time.

```
public class SynchronizedExample {
    private int counter = 0;

    public synchronized void increment() {
        counter++;
    }

    public static void main(String[] args) throws InterruptedException {
        SynchronizedExample example = new SynchronizedExample();

        Thread t1 = new Thread(example::increment);
        Thread t2 = new Thread(example::increment);

        t1.start();
        t2.start();

        t1.join();
        t2.join();


        System.out.println("Final Counter: " + example.counter);
    }
}
```

Explanation: The increment() method is synchronized to ensure that only one thread can modify the counter variable at a time.

2. Thread Pools

2.1 What is a Thread Pool?

A thread pool in Java is a collection of pre-instantiated reusable threads. Instead of creating new threads for each task, the thread pool manages a set number of threads and reuses them for executing tasks. This approach reduces the overhead of thread creation and destruction, making it more efficient for handling multiple tasks.

2.2 Creating and Using Thread Pools

2.2.1 Using ExecutorService

The ExecutorService is a higher-level replacement for working with threads directly. It provides various methods to manage and control thread execution.

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

import java.util.concurrent.Executors;


public class ThreadPoolExample {

    public static void main(String[] args) {

        ExecutorService executorService = Executors.newFixedThreadPool(3);


        for (int i = 1; i <= 5; i++) {

            int taskId = i;

            executorService.execute(() -> {

                System.out.println("Executing task " + taskId + " by " + Thread.currentThread().getName());

            });

        }
        executorService.shutdown();
    }
}
```

Explanation: In this example, a fixed thread pool with 3 threads is created using Executors.newFixedThreadPool(3). Five tasks are submitted to the pool. The thread pool will execute these tasks, reusing the threads as necessary. The shutdown() method is called to stop the thread pool after all tasks are completed.

2.3 Benefits of Using Thread Pools

- **Resource Management:** Thread pools limit the number of concurrent threads, helping manage CPU and memory resources more effectively.
- **Task Management:** By reusing threads, thread pools reduce the overhead of creating and destroying threads for each task.
- **Performance Improvement:** Thread pools improve application performance by minimizing the latency associated with thread creation.