Concurrency-2 Executors and Callables

In this tutorial, we will cover the following concepts.

- Executor Framework
    - Overview
    - Using Executor Framework
    - Thread Pools
    - Types of Thread Pool
    - Benefits of Executor Framework
- Callables & Future
    - How threads can return data?
- Coding Problems
    - Multi-threaded Merge Sort
    - Download Manager
    - Image Processing App
    - Scheduled Executor
- Synchronization Problems Introduction
    - Adder - Subtractor

Overview

Imagine you have a computer program that needs to do several tasks at the same time. For example, your program might need to download files, process data, and update a user interface simultaneously. In the traditional way of programming, you might use threads to handle these tasks. However, managing threads manually can be complex and error-prone. Java ExecutorService implementations let you stay focused on tasks that need to be run, rather than thread creation and management.

Think of a chef in a kitchen as your program. The chef has multiple tasks like chopping vegetables, cooking pasta, and baking a cake. Instead of the chef doing each task one by one, the chef hires sous-chefs (threads) to help. The chef (Executor Framework) can assign tasks to sous-chefs efficiently, ensuring that multiple tasks are happening simultaneously, and the kitchen operates smoothly.

In Java, the Executor Framework provides a convenient way to implement this idea in your code, making it more readable, maintainable, and efficient. It simplifies the process of managing tasks concurrently, so you can focus on solving the problems your program is designed to address.

Using Executor Framework

The Executor Framework in Java provides a high-level and flexible framework for managing and controlling the execution of tasks in concurrent programming. It is part of the `java.util.concurrent` package and was introduced in Java 5 to simplify the development of concurrent applications. The primary motivation behind using the

Executor Framework is to abstract away the complexities of thread management, providing a clean and efficient way to execute tasks asynchronously.

Now, let's look at a real-world example to illustrate the use of the Executor Framework. Consider a scenario where you have a set of tasks that need to be executed concurrently to improve performance. We'll use a ThreadPoolExecutor for this example:

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

public class ExecutorDemo {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        for(int i=0; i<100;i++){
            executor.execute(new NumberPrinter(i));
        }
        executor.shutdown();
    }
}
```

Here `NumberPrinter()` is runnable task as created earlier. The `Executor` interface is used to execute tasks. It is a generic interface that can be used to execute any kind of task. The `Executor` interface has only one method:

```
public interface Executor {
    void execute(Runnable command);
}
```

The `execute` method takes a `Runnable` object as a parameter. The `Runnable` interface is a functional interface that has only one method. Executors internally use a thread pool to execute the tasks. The `execute` method is non-blocking. It returns immediately after submitting the task to the thread pool. The `execute` method is used to execute tasks that do not return a result. A thread pool is a collection of threads that are used to execute tasks. Instead of creating a new thread for each task, a thread pool reuses the existing threads to execute the tasks. This improves the performance of the application.

Thread Pool - Deep Dive

- Creating threads, destroying threads, and then creating them again can be expensive.
- A thread pool mitigates the cost, by keeping a set of threads around, in a pool, for current and future work.
- Threads, once they complete one task, can then be reassigned to another task, without the expense of destroying that thread and creating a new one.

A thread pool consists of three components.

1. **Worker Threads** are available in a pool to execute tasks. They're pre-created and kept alive, throughout the lifetime of the application.

2. Submitted Tasks are placed in a **First-In First-Out queue**. Threads pop tasks from the queue, and execute them, so they're executed in the order they're submitted.

3. The **Thread Pool Manager** allocates tasks to threads, and ensures proper thread synchronization.

Types of Thread Pool

In Java, the `ExecutorService` interface, along with the `ThreadPoolExecutor` class, provides a flexible thread pool framework. Here are five types of thread pools in Java, each with different characteristics:

1. FixedThreadPool

- **Description:** A thread pool with a fixed number of threads.
- **Characteristics:**
  - Reuses a fixed number of threads for all submitted tasks.
  - If a thread is idle, it will be reused for a new task.
  - If all threads are busy, tasks are queued until a thread becomes available.
- **Creation:**

```
ExecutorService executor = Executors.newFixedThreadPool(nThreads);
```

2. CachedThreadPool

- **Description:** A thread pool that dynamically adjusts the number of threads based on demand.
- **Characteristics**
  - Creates new threads as needed, but reuses idle threads if available.
  - Threads that are idle for a certain duration are terminated.
  - Suitable for handling a large number of short-lived tasks.

```
ExecutorService executor = Executors.newCachedThreadPool();
```

3. SingleThreadExecutor

- **Description:** A thread pool with only one thread.
- **Characteristics:**
  - Executes tasks sequentially in the order they are submitted.
  - Useful for tasks that need to be executed in a specific order or when a single thread is sufficient.

```
ExecutorService executor = Executors.newSingleThreadExecutor();
```

4.ScheduledThreadPool

- **Description**: A thread pool that supports scheduling of tasks.
- **Characteristics**:
  - Similar to FixedThreadPool but with added support for scheduling tasks at fixed rates or delays.
  - Suitable for periodic tasks or tasks that need to be executed after a certain delay.

```
ScheduledExecutorService executor =
Executors.newScheduledThreadPool(nThreads);
```

5. WorkStealingPool

- **Description**: Introduced in Java 8, it's a parallelism-friendly thread pool.
- **Characteristics:**
  - Creates a pool of worker threads that dynamically adapt to the number of available processors.
  - Each worker thread has its own task queue.
  - Suitable for parallel processing tasks.

```
ExecutorService executor = Executors.newWorkStealingPool();
```

These different types of thread pools cater to various scenarios and workloads. The choice of a thread pool depends on factors such as task characteristics, execution requirements, and resource constraints in your application.

Benefits of Executor Framework

1. Simplifies Task Execution

The Executor Framework makes it easier to execute tasks concurrently. It abstracts away the low-level details of managing threads, so you don't have to worry about creating and controlling them yourself.

2. Efficient Resource Utilization:

When you have many tasks to perform, creating a new thread for each task can be inefficient. The Executor Framework provides a pool of threads that can be reused for multiple tasks. This reuse of threads reduces the overhead of creating and destroying threads for every task.

3. Better Control and Flexibility

With the Executor Framework, you can control how many tasks can run simultaneously, manage the lifecycle of threads, and specify different policies for task execution. This level of control is important for optimizing the performance of your program.

4. Enhanced Scalability

When your program needs to handle more tasks, the Executor Framework makes it easier to scale. You can adjust the size of the thread pool to accommodate more tasks without rewriting a lot of code.

5. Task Scheduling

The framework allows you to schedule tasks to run at specific times or after certain intervals. This is useful for scenarios where you want to automate repetitive tasks or execute tasks at specific points in time.

Summary

- Managing threads manually can be complex and error-prone.

- It can lead to complex issues like resource contention, thread creation overhead, and scalability challenges.
- For these reasons, you'll want to use an ExecutorService, even when working with a single thread.

Callable and Future

Runnables do not return a result. If we want to execute a task that returns a result, we can use the `Callable` interface. The `Callable` interface is a functional interface that has only one method:

```
public interface Callable<V> {
    V call() throws Exception;
}
```

The `call` method returns a result of type `V`. The `call` method can throw an exception. The `Callable` interface is used to execute tasks that return a result. For instance we can use the `Callable` interface to execute a task that returns the sum of two numbers:

```
Callable<Integer> sumTask = () -> 2 + 3;
```

In order to execute a task that returns a result, we can use the `submit` method of the `ExecutorService` interface. The `submit` method takes a `Callable` object as a parameter. The `submit` method returns a `Future` object. The `Future` interface has a method called `get` that returns the result of the task. The `get` method is a blocking method. It waits until the task is completed and then returns the result of the task.

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<Integer> future = executorService.submit(() -> 2 + 3);
Integer result = future.get();
```

Futures can be used to cancel tasks. The `Future` interface has a method called `cancel` that can be used to cancel a task. The `cancel` method takes a boolean parameter. If the boolean parameter is `true`, the task is cancelled even if the task is already running. If the boolean parameter is `false`, the task is cancelled only if the task is not running.

```
ExecutorService executorService = Executors.newCachedThreadPool();
Future<Integer> future = executorService.submit(() -> 2 + 3);
future.cancel(false);
```

Coding Problem 1 : Merge Sort

Implement multi-threaded merge sort.

Solution Sorter.java

```
import java.util.ArrayList;
import java.util.concurrent.Callable;
import java.util.List;
import java.util.concurrent.ExecutorService;
```

```java
import java.util.concurrent.Future;

public class Sorter implements Callable<List<Integer>> {
    private List<Integer> arr;
    private ExecutorService executor;
    Sorter(List<Integer> arr,ExecutorService executor){
        this.arr = arr;
        this.executor = executor;

    }
    @Override
    public List<Integer> call() throws Exception {
        //Business Logic
        //base case
        if(arr.size()<=1){
            return arr;
        }

        //recursive case
        int n = arr.size();
        int mid = n/2;

        List<Integer> leftArr = new ArrayList<>();
        List<Integer> rightArr = new ArrayList<>();

        //Division of array into 2 parts
        for(int i=0;i<n;i++){
            if(i<mid){
                leftArr.add(arr.get(i));
            }
            else{
                rightArr.add(arr.get(i));
            }
        }

        //Recursively Sort the 2 array
        Sorter leftSorter = new Sorter(leftArr,executor);
        Sorter rightSorter = new Sorter(rightArr,executor);

        Future<List<Integer>> leftFuture = executor.submit(leftSorter);
        Future<List<Integer>> rightFuture = executor.submit(rightSorter);

        leftArr = leftFuture.get();
        rightArr = rightFuture.get();


        //Merge
        List<Integer> output = new ArrayList<>();
        int i=0;
        int j=0;
        while(i<leftArr.size() && j<rightArr.size()){
            if(leftArr.get(i)<rightArr.get(j)){
```

```
                output.add(leftArr.get(i));
                i++;
            }
            else{
                output.add(rightArr.get(j));
                j++;
            }
        }
        // copy the remaining elements
        while(i<leftArr.size()){
            output.add(leftArr.get(i));
            i++;
        }
        while(j<rightArr.size()){
            output.add(rightArr.get(j));
            j++;
        }
        return output;
    }
}
```

Main.java

```
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class Main {
    public static void main(String[] args) throws Exception {
        List<Integer> l = List.of(7,3,1,2,4,6,17,12);
        ExecutorService executorService = Executors.newCachedThreadPool();

        Sorter sorter = new Sorter(l,executorService);
        Future<List<Integer>> output = executorService.submit(sorter);
        System.out.println(output.get()); //Blocking Code
        executorService.shutdown();
    }
}
```

Coding Problem 2 : Download Manager (Homework)

Consider a simple download manager application that needs to download multiple files concurrently. Implement the download manager using the Java Executor Framework. Requirements:

- The download manager should be able to download multiple files simultaneously.
- Each file download is an independent task that can be executed concurrently.
- The download manager should use a thread pool from the Executor Framework to manage and execute the download tasks.
- Implement a mechanism to track the progress of each download task and display it to the user.

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

class DownloadManager {
    private ExecutorService executorService;

    public DownloadManager(int threadPoolSize) {
        // TODO: Initialize the ExecutorService with a fixed-size thread pool.
    }

    public void downloadFiles(List<String> fileUrls) {
        // TODO: Implement a method to submit download tasks for each file URL.
    }

    // TODO: Implement a method to track and display the progress of each download
task.

    public void shutdown() {
        // TODO: Shutdown the ExecutorService when the download manager is done.
    }
}
```

```
public class DownloadManagerApp {
    public static void main(String[] args) {
        // TODO: Create a DownloadManager instance with an appropriate thread pool
size.
        // TODO: Test the download manager by downloading multiple files concurrently.
        // TODO: Display the progress of each download task.
        // TODO: Shutdown the download manager after completing the downloads.
    }
}
```

Tasks for Implementation

- Initialize the ExecutorService in the DownloadManager constructor.
- Implement the downloadFiles method to submit download tasks for each file URL using the ExecutorService.
- Implement a mechanism to track and display the progress of each download task.
- Test the download manager in the DownloadManagerApp by downloading multiple files concurrently.
- Shutdown the ExecutorService when the download manager is done.
- Feel free to adapt and extend the code as needed. This example focuses on using the Executor Framework for concurrent file downloads in a download manager application.

Solution

```
import java.util.List;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
```

```java
class DownloadTask implements Runnable {
    private String fileUrl;

    public DownloadTask(String fileUrl) {
        this.fileUrl = fileUrl;
    }

    @Override
    public void run() {
        // Simulate file download
        System.out.println("Downloading file from: " + fileUrl);

        // Simulate download progress
        for (int progress = 0; progress <= 100; progress += 10) {
            System.out.println("Progress for " + fileUrl + ": " + progress + "%");
            try {
                Thread.sleep(500); // Simulate download time
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        System.out.println("Download complete for: " + fileUrl);
    }
}

class DownloadManager {
    private ExecutorService executorService;

    public DownloadManager(int threadPoolSize) {
        executorService = Executors.newFixedThreadPool(threadPoolSize);
    }

    public void downloadFiles(List<String> fileUrls) {
        for (String fileUrl : fileUrls) {
            DownloadTask downloadTask = new DownloadTask(fileUrl);
            executorService.submit(downloadTask);
        }
    }

    public void shutdown() {
        executorService.shutdown();
    }
}

public class DownloadManagerApp {
    public static void main(String[] args) {
        DownloadManager downloadManager = new DownloadManager(3); // Use a thread pool
size of 3

        List<String> filesToDownload = List.of("file1", "file2", "file3", "file4",
"file5");
```

```
        downloadManager.downloadFiles(filesToDownload);

        // Display progress (simulated)
        // Note: In a real-world scenario, you might need to implement a more
sophisticated progress tracking mechanism.
        for (int i = 0; i < 10; i++) {
            System.out.println("Main thread is doing some work...");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        downloadManager.shutdown();
    }
}
```

Coding Problem 3 : Image Processing

Many image processing applications like Lightroom & Photoshop use multiple threads to process an image quickly. In this problem, you will build a simplified image repainting task using multiple threads, the repainting task here simply doubles the value of every pixel stored in the form of a 2D array. Take Input a NXN matrix and repaint it by using 4 threads, one for each quadrant.

Solution Repainting a 2D array using four threads can be achieved by dividing the array into quadrants, and assigning each quadrant to a separate thread for repainting.

This example divides the 2D array into four quadrants and assigns each quadrant to a separate thread for repainting. The ArrayRepainterTask class represents the task for repainting a specific quadrant. The program then uses an ExecutorService with a fixed thread pool to concurrently execute the tasks. Finally, it prints the repainted 2D array.

Below is an example code using the Java Executor Framework

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

class ArrayRepainterTask implements Runnable {
    private final int[][] array;
    private final int startRow;
    private final int endRow;
    private final int startCol;
    private final int endCol;

    public ArrayRepainterTask(int[][] array, int startRow, int endRow, int startCol,
int endCol) {
        this.array = array;
        this.startRow = startRow;
        this.endRow = endRow;
```

```
        this.startCol = startCol;
        this.endCol = endCol;
    }

    @Override
    public void run() {
        // Simulate repainting for the specified quadrant
        for (int i = startRow; i <= endRow; i++) {
            for (int j = startCol; j <= endCol; j++) {
                array[i][j] = array[i][j] * 2; // Repaint by doubling the values
(simulated)
            }
        }
    }
}

public class ArrayRepaintingExample {
    public static void main(String[] args) {
        int[][] originalArray = {
                {1, 2, 3, 4},
                {5, 6, 7, 8},
                {9, 10, 11, 12},
                {13, 14, 15, 16}
        };

        int rows = originalArray.length;
        int cols = originalArray[0].length;

        ExecutorService executorService = Executors.newFixedThreadPool(4);

        // Divide the array into four quadrants
        int midRow = rows / 2;
        int midCol = cols / 2;

        // Create tasks for each quadrant
        ArrayRepainterTask task1 = new ArrayRepainterTask(originalArray, 0, midRow -
1, 0, midCol - 1);
        ArrayRepainterTask task2 = new ArrayRepainterTask(originalArray, 0, midRow -
1, midCol, cols - 1);
        ArrayRepainterTask task3 = new ArrayRepainterTask(originalArray, midRow, rows
- 1, 0, midCol - 1);
        ArrayRepainterTask task4 = new ArrayRepainterTask(originalArray, midRow, rows
- 1, midCol, cols - 1);

        // Submit tasks to the ExecutorService
        executorService.submit(task1);
        executorService.submit(task2);
        executorService.submit(task3);
        executorService.submit(task4);

        // Shutdown the ExecutorService
        executorService.shutdown();
```

```
        // Wait for all tasks to complete
        while (!executorService.isTerminated()) {
            // Wait
        }

        // Print the repainted array
        for (int[] row : originalArray) {
            for (int value : row) {
                System.out.print(value + " ");
            }
            System.out.println();
        }
    }
}
```

Coding Problem 4: Scheduled Executor

Write a Java program that uses ScheduledExecutorService to schedule a task to run periodically. Implement a task that prints a message “Hello” at fixed intervals of 5s.

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

public class ScheduledExecutorExample {

    public static void main(String[] args) {
        // Create a ScheduledExecutorService with a single thread
        ScheduledExecutorService scheduledExecutorService =
Executors.newSingleThreadScheduledExecutor();

        // Schedule the task to run periodically every 5 seconds
        scheduledExecutorService.scheduleAtFixedRate(() -> {
            System.out.println("Hello");
        }, 0, 5, TimeUnit.SECONDS);

        // Sleep for a while to allow the task to run multiple times
        try {
            Thread.sleep(20000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Shutdown the ScheduledExecutorService
        scheduledExecutorService.shutdown();
    }
}
```

Synchronisation

Whenever we have multiple threads that access the same resource, we need to make sure that the threads do not interfere with each other. This is called synchronisation.

Synchronisation can be seen in the adder and subtractor example. The adder and subtractor threads access the same counter variable. If the adder and subtractor threads do not synchronise, the counter variable can be in an inconsistent state.

- Create a count class that has a count variable.
- Create two different classes `Adder` and `Subtractor`.
- Accept a count object in the constructor of both the classes.
- In `Adder`, iterate from 1 to 10000 and increment the count variable by 1 on each iteration.
- In `Subtractor`, iterate from 1 to 10000 and decrement the count variable by 1 on each iteration.
- Print the final value of the count variable.
- What would the ideal value of the count variable be?
- What is the actual value of the count variable?
- Try to add some delay in the `Adder` and `Subtractor` classes using inspiration from the code below. What is the value of the count variable now?

Steps to implement

- Implement Adder & Subtractor
- Shared counter via constructor
- Create a package called addersubtractor
- Create two tasks under adder and subtractor

Adder

```
package addersubtractor;
    public class Adder implements Runnable {
        private Count count;
        public Adder (Count count) {
                this.count = count;
              }
         @Override
        public void run() {
            for (int i = 1 ; i <= 100; ++ 1) {
                    count.value += i;
            }
}
```

Subtracter

```
package addersubtractor;
    public class Subtractor implements Runnable  {
        private Count count;
        public Subtractor (Count count) {
            this.count = count
            }
        @Override
        public void run() {
            for (int i = 1 ; i <= 100; ++ 1) {
                    count.value -= i;
```

```
            }
}
```

Count class

```
package addersubtractor;
    public class Count {
        int value = 0;
    }
```

Now, let’s make our client class here:

```
public static void main(String[] args) {
    Count count = new Count;
    Adder adder = new Adder (count);
    Subtractor subtractor = new Subtractor (count);
    Thread t1 = new Thread (adder);
    Thread t2 = new Thread (subtractor);
    t1.start();
    t2.start();
    t1.join();
    t2.join();

    system.out.println(count.value);
```

Output is some random number every time we run the code.

Now, this particular problem is known to be a data synchronization problem. This happens because the same data object is shared among various multi-threads, and they both are trying to modify the same data. This is an unexpected result that we have seen, but we will continue this in the next tutorial.