Assignment 3.8

Name: Liew Chooi Chin

Async vs Multithreaded Programming

Q1: What is asynchronous programming?

Asynchronous programming is a programming paradigm that allows tasks to be executed independently, without blocking the main thread of execution. This enables the program to continue executing other tasks while waiting for asynchronous operations to complete.

In this example, the CompletableFuture.supplyAsync() method creates a new thread to execute the long-running task. The main thread can continue executing other tasks while the asynchronous task is running. Once the task completes, the result is returned and printed to the console.

Q2: What is multithreaded programming?

Multithreaded programming involves creating and managing multiple threads of execution within a single process. Each thread can run concurrently, allowing for parallel execution of tasks.

Q3: What are the similarities and differences between them?

Both asynchronous programming and multithreaded programming are techniques for concurrent execution, they differ in their approach and the underlying mechanisms used.

Similarities:

- Both asynchronous programming and multithreaded programming can improve application performance and responsiveness.
- Both can handle multiple tasks concurrently.

Differences:

Asynchronous programming:

- Focuses on non-blocking operations.
- Often uses a single thread of execution.
- Commonly used for I/O-bound tasks.

Multithreaded programming:

- Focuses on creating multiple threads of execution.
- Can handle both CPU-bound and I/O-bound tasks.

- Requires careful management of thread synchronization to avoid race conditions and deadlocks.

Understanding CompletableFuture

Q1: What is the difference between supplyAsync and runAsync in CompletableFuture?

Both supplyAsync and runAsync are methods of CompletableFuture that create new asynchronous tasks. The main difference between runAsync() and supplyAsync() lies in the input they accept and the type of return value they produce.

supplyAsync:

- supplyAsync() is a method used to asynchronously execute a task that produces a result. It's ideal for tasks that require a result for further processing. For example, fetching data from a database, making an API call, or performing a computation asynchronously.
- Accepts a Supplier<T> as an argument.
- Executes the supplied task asynchronously and returns a CompletableFuture<T>.

The result of the task is the value returned by the supplier.

Example:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(()
-> {
    // Perform result-producing task
    return "Result of the asynchronous computation";
});

// Get the result later
String result = future.get();
System.out.println("Result: " + result);
Output:
Result: Result of the asynchronous computation
```

runAsync:

- runAsync() is a method used to execute a task asynchronously that doesn't produce a result. It's suitable for fire-and-forget tasks where we want to execute code asynchronously without waiting for a result. For example, logging, sending notifications, or triggering background tasks.
- supplyAsync() allows chaining operations due to its return value. Since supplyAsync() produces a result, we can use methods like thenApply() to transform the result, thenAccept() to consume the result,
- Accepts a Runnable as an argument.
- Executes the supplied task asynchronously and returns a CompletableFuture<Void>.

- The task doesn't return a value.

Example:

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
      // Perform non-result producing task
      System.out.println("Task executed asynchronously");
});
Output:
Task completed successfully
```

Q2: What is the difference between thenAccept, thenRun, and thenApply in CompletableFuture?

These methods are used to chain asynchronous operations onto a CompletableFuture. We can use them to define actions to be performed when the future completes:

thenAccept:

- Accepts a Consumer<T> as an argument.
- When the future completes successfully, the consumer is executed with the result of the future.

thenApply:

- Accepts a Function<T, U> as an argument.
- When the future completes successfully, the function is applied to the result, and a new CompletableFuture<U> is returned.

Example:

```
CompletableFuture<String> future = CompletableFuture.supplyAsync(()
-> {
      return "Result of the asynchronous computation";
});

future.thenApply(result -> {
      // Transform the result
      return result.toUpperCase();
}).thenAccept(transformedResult -> {
      // Consume the transformed result
      System.out.println("Transformed Result: " + transformedResult);
});
Output:
Transformed Result: RESULT OF THE ASYNCHRONOUS COMPUTATION
```

thenRun:

- runAsync() method cannot be directly chained with methods like thenApply() or thenAccept() as it doesn't produce a result. However, we can use thenRun() to execute another task after the runAsync() task completes. This method allows us to chain additional tasks for sequential execution without relying on the result of the initial task.

- Accepts a Runnable as an argument.
- When the future completes successfully, the runnable is executed.

Example:

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
     System.out.println("Task executed asynchronously");
});

future.thenRun(() -> {
     // Execute another task after the completion of runAsync()
     System.out.println("Another task executed after runAsync()
completes");
});
Output:
Task executed asynchronously
Another task executed after runAsync() completes
```

REferences:

- https://www.baeldung.com/java-completablefuture-runasync-supplyasync
- Completed with the help of Gemini.