Callable Code

Lesson on Java Callable and Future

In Java, the **Callable** interface is similar to **Runnable**, but it can return a result and throw checked exceptions. The **Future** interface represents the result of an asynchronous computation, allowing you to retrieve the result once the task is complete.

This lesson will cover three scenarios:

1. Fetching stock prices.
2. Fetching the status of system processes.
3. A recommendation service.

Key Concepts:

- **Callable Interface**: Allows you to define tasks that return a result (`V`) or throw an exception (`Exception`).

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

- **Future Interface**: Represents the result of a computation, which might not yet be available.
  Methods:
  - `get()`: Waits if necessary for the computation to complete, then retrieves the result.
  - `isDone()`: Returns `true` if the task has finished.

---

1. Fetching Stock Prices

Scenario:

A stock market application needs to fetch stock prices asynchronously for multiple companies.

Code Example:

```java
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class StockPriceFetcher implements Callable<Double> {
    private final String stockSymbol;

    public StockPriceFetcher(String stockSymbol) {
        this.stockSymbol = stockSymbol;
    }

    @Override
    public Double call() throws Exception {
        // Simulate fetching stock price (e.g., from an API or database)
        double price = Math.random() * 1000;
        System.out.println("Fetched price for " + stockSymbol + ": " + price);
        Thread.sleep(2000); // Simulate delay
```

```
        return price;
    }
}

public class StockPriceApp {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);
        Future<Double> applePrice = executor.submit(new StockPriceFetcher("AAPL"));
        Future<Double> googlePrice = executor.submit(new StockPriceFetcher("GOOG"));
        Future<Double> amazonPrice = executor.submit(new StockPriceFetcher("AMZN"));

        try {
            System.out.println("Apple stock price: $" + applePrice.get());
            System.out.println("Google stock price: $" + googlePrice.get());
            System.out.println("Amazon stock price: $" + amazonPrice.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Explanation:

- We define a `StockPriceFetcher` class that implements `Callable<Double>`, fetching stock prices for a company.
- Multiple tasks are submitted to an **ExecutorService**, and the **Future** objects store the results.
- `get()` waits for the results and then prints them.

---

2. Fetching Status of System Processes

Scenario:

An administrator needs to monitor the statuses of system processes asynchronously.

Code Example:

```
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class SystemProcessStatusFetcher implements Callable<String> {
    private final String processName;

    public SystemProcessStatusFetcher(String processName) {
        this.processName = processName;
    }

    @Override
    public String call() throws Exception {
        // Simulate checking the status of a process (e.g., from OS commands)
        System.out.println("Checking status of process: " + processName);
        Thread.sleep(1000); // Simulate delay
        String status = "Running"; // Assume all processes are running for simplicity
        System.out.println("Process " + processName + " is " + status);
```

```
        return status;
    }
}

public class SystemMonitorApp {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(3);

        Future<String> processA = executor.submit(new SystemProcessStatusFetcher("ProcessA"));
        Future<String> processB = executor.submit(new SystemProcessStatusFetcher("ProcessB"));
        Future<String> processC = executor.submit(new SystemProcessStatusFetcher("ProcessC"));

        try {
            System.out.println("Status of ProcessA: " + processA.get());
            System.out.println("Status of ProcessB: " + processB.get());
            System.out.println("Status of ProcessC: " + processC.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Explanation:

- The `SystemProcessStatusFetcher` class implements `Callable<String>`, simulating fetching the status of system processes.
- Multiple processes are checked concurrently, and the status is printed once retrieved using `get()`.

---

3. Recommendation Service

Scenario:

An e-commerce application needs to provide product recommendations asynchronously based on a user's shopping history.

Code Example:

```
import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

class RecommendationService implements Callable<List<String>> {
    private final String userId;

    public RecommendationService(String userId) {
        this.userId = userId;
    }

    @Override
    public List<String> call() throws Exception {
        // Simulate recommendation generation based on user history
        System.out.println("Fetching recommendations for user: " + userId);
        Thread.sleep(1500); // Simulate delay
```

```
        List<String> recommendations = new ArrayList<>();
        recommendations.add("Product A");
        recommendations.add("Product B");
        recommendations.add("Product C");
        System.out.println("Recommendations for " + userId + ": " + recommendations);
        return recommendations;
    }
}

public class RecommendationApp {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(2);

        Future<List<String>> user1Recommendations = executor.submit(new RecommendationService("User1"));
        Future<List<String>> user2Recommendations = executor.submit(new RecommendationService("User2"));

        try {
            System.out.println("User1 Recommendations: " + user1Recommendations.get());
            System.out.println("User2 Recommendations: " + user2Recommendations.get());
        } catch (InterruptedException | ExecutionException e) {
            e.printStackTrace();
        } finally {
            executor.shutdown();
        }
    }
}
```

Explanation:

- The `RecommendationService` class implements `Callable<List<String>>`, simulating fetching product recommendations based on user shopping history.
- Recommendations for two users are fetched concurrently, and the result is retrieved using the **Future** object's `get()` method.

Summary of Concepts:

1. **Callable**: Defines a task that returns a value and may throw exceptions.
2. **Future**: Represents the result of the asynchronous task, providing methods like `get()` to retrieve the result once the task is complete.
3. **ExecutorService**: Manages a pool of threads to run multiple tasks concurrently.

Key Methods:

- `submit(Callable<T> task)`: Submits a task for execution and returns a **Future**.
- `Future.get()`: Retrieves the result, blocking if necessary until the computation is complete.
- `isDone()`: Checks if the task has completed.

By understanding and applying these patterns, you can efficiently handle asynchronous tasks in Java applications.