Practices for Lesson 12

Practices Overview

In these practices, you will use the `java.util.concurrent` package and sub-packages of the Java programming language.

(Optional) Practice 13-1: Using the `java.util.concurrent` Package

Overview

In this practice, you will modify an existing project to use an `ExecutorService` from the `java.util.concurrent` package.

Assumptions

You have reviewed the sections covering the use of the `java.util.concurrent package`.

Summary

You will create a multithread networking client that will rapidly read the price of a shirt from several different servers. Instead of manually creating threads, you will leverage an `ExecutorService` from the `java.util.concurrent` package.

Tasks

1. Open the `ExecutorService` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\12\practices`. `(or your other folder)`
   c. Select `ExecutorService` and select the "Open as Main Project" check box.
   d. Click the Open Project button.
2. Expand the project directories.
3. Run the `NetworkServerMain` class in the `com.example.server` package by right-clicking the class and selecting Run File.
4. Open the `NetworkClientMain` class in the `com.example.client` package.
5. Run the `NetworkClientMain` class package by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers sequentially.
6. Create a `NetworkClientCallable` class in the `com.example.client` package.
   - Add a constructor and a field to receive and store a `RequestResponse` reference.
   - Implement the `Callable` interface with a generic type of `RequestResponse`.

```
public class NetworkClientCallable implements
Callable<RequestResponse>
```

   - Complete the `call` method by using a `java.net.Socket` and a `java.util.Scanner` to read the response from the server. Store the result in the `RequestResponse` object and return it.

   Note: You may want to use a `try-with-resource` statement to ensure that the `Socket` and `Scanner` objects are closed.
7. Modify the `main` method of the `NetworkClientMain` class to query the servers concurrently by using an `ExecutorService`.
   a. Comment out the contents of the `main` method.
   b. Obtain an `ExecutorService` that reuses a pool of cached threads.

c. Create a Map that will be used to tie a request to a future response.

```
Map<RequestResponse, Future<RequestResponse>> callables = new
HashMap<>();
```

d. Code a loop that will create a `NetworkClientCallable` for each network request.

- The servers should be running on localhost, ports 10000–10009.
- Submit each `NetworkClientCallable` to the ExecutorService. Store each `Future` in the `Map` created in step 7c.

e. Shut down the `ExecutorService`.

f. Await the termination of all threads within the `ExecutorService` for 5 seconds.

g. Loop through the `Future` objects stored in the `Map` created in step 7c. Print out the servers' response or an error message with the server details if there was a problem communicating with a server.

8. Run the `NetworkClientMain` class by right-clicking the class and selecting Run File. Notice the amount of time it takes to query all the servers concurrently.
9. When done testing your client, be sure to select the ExecutorService output tab and terminate the server application.

(Optional) Practice 12-2: Using the Fork-Join Framework

Overview

In this practice, you will modify an existing project to use the Fork-Join framework.

Assumptions

You have reviewed the sections covering the use of the Fork-Join framework.

Summary

You are given an existing project that already leverages the Fork-Join framework to process the data contained within an array. Before the array is processed, it is initialized with random numbers. Currently the initialization is single-thread. You must use the Fork-Join framework to initialize the array with random numbers.

Tasks

1. Open the `ForkJoinFindMax` project as the main project.
   a. Select File > Open Project.
   b. Browse to `D:\labs\12\practices`. `(or your other folder)`
   c. Select `ForkJoinFindMax` and select the "Open as Main Project" check box.
   d. Click the Open Project button.
2. Expand the project directories.
3. Open the `Main` class in the `com.example` package.
   - Review the code within the `main` method. Take note of how the `compute` method splits the `data` array if the count of elements to process is too great.
4. Open the `FindMaxTask` class in the `com.example` package.
   - Review the code within the class. Take note of the `for` loop used to initialize the `data` array with random numbers.
5. Create a `RandomArrayAction` class in the `com.example` package.
   a. Add four fields.

```
private final int threshold;
private final int[] myArray;
private int start;
private int end;
```

   b. Add a constructor that receives parameters and saves their values within the fields defined in the previous step.

```
public RandomArrayAction(int[] myArray, int start, int end, int
threshold)
```

   c. Extend the `RecursiveAction` class from the `java.util.concurrent` package.

      Note: A `RecursiveAction` is used when a `ForkJoinTask` with no return values is needed.

d. Add the `compute` method. Note that unlike the `compute` method from a `RecursiveTask`, the `compute` method in a `RecursiveAction` returns `void`.

```
protected void compute() { }
```

e. Begin the `compute` method. If the number of elements to process is below the threshold, you should initialize the array.

```
for (int i = start; i <= end; i++) {
    myArray[i] = ThreadLocalRandom.current().nextInt();
}
```

Note: `ThreadLocalRandom` is used instead of `Math.random()` because `Math.random()` does not scale when executed concurrently by multiple threads and would eliminate any benefit of applying the Fork-Join framework to this task.

f. Complete the `compute` method. If the number of elements to process is above or equal to the threshold you should find the midway point in the array and create two new `RandomArrayAction` instances for each section of the array to process. Start each `RandomArrayAction`.

Note: When starting a `RecursiveAction`, you can use the `invokeAll` method instead of the `fork`/`join`/`compute` combination typically seen with a `RecursiveTask`.

```
RandomArrayAction r1 = new RandomArrayAction(myArray, start,
midway, threshold);
RandomArrayAction r2 = new RandomArrayAction(myArray, midway +
1, end, threshold);
invokeAll(r1, r2);
```

6. Modify the `main` method of the `Main` class to use the `RandomArrayAction` class.

   a. Comment out the `for` loop within the `main` method that initializes the `data` array with random values.

   b. After the line that creates the `ForkJoinPool`, create a new `RandomArrayAction`.

   c. Use the `ForkJoinPool` to invoke the `ForkJoinPool`.

7. Run the `ForkJoinFindMax` project by right-clicking the project and choosing *Run*.

   Note: If you have a multi-CPU system you can use `System.currentTimeMillis()` to benchmark the sequential and Fork-Join solutions.