23.14 (Advanced) Interfaces `Callable` and `Future`

Interface `Runnable` provides only the most basic functionality for multithreaded programming. In fact, this interface has limitations. Suppose a `Runnable` is performing a long calculation and the application wants to retrieve the result of that calculation. The `run` method cannot return a value, so *shared mutable data* would be required to pass the value back to the calling thread. As you now know, this would require thread synchronization. The `Callable` interface (of package `java.util.concurrent`) fixes this limitation. The interface declares a single method named `call` which returns a value representing the result of the `Callable`'s task—such as the result of a long-running calculation.

An application that creates a `Callable` likely wants to run it concurrently with other `Runnable`s and `Callable`s. `ExecutorService` method `submit` executes its `Callable` argument and returns an object of type `Future` (of package `java.util.concurrent`), which represents the `Callable`'s future result. The `Future` interface `get` method *blocks* the calling thread, and waits for the `Callable` to complete and return its result. The interface also provides methods that enable you to cancel a `Callable`'s execution, determine whether the `Callable` was canceled and

determine whether the `Callable` completed its task.

Executing Aysnchronous Tasks with `CompletableFuture`

8

Java SE 8 introduced class `CompletableFuture` (package `java.util.concurrent`), which implements the `Future` interface and enables you to *asynchronously* execute `Runnable`s that perform tasks or `Supplier`s that return values. Interface `Supplier`, like interface `Callable`, is a functional interface with a single method (in this case, `get`) that receives no arguments and returns a result. Class `CompletableFuture` provides many additional capabilities for advanced programmers, such as creating `CompletableFuture`s without executing them immediately, composing one or more `CompletableFuture`s so that you can wait for any or all of them to complete, executing code after a `CompletableFuture` completes and more.

Figure 23.31 performs two long-running calculations sequentially, then performs them again asynchronously using `CompletableFuture`s to demonstrate the performance improvement from asynchronous execution on a multi-core system. For demonstration purposes, our long-running

calculation is performed by a recursive `fibonacci` method (lines 69–76; similar to the one presented in Section 18.5). For larger Fibonacci values, the recursive implementation can require *significant* computation time—in practice, it's much faster to calculate Fibonacci values using a loop.

```
1   // Fig. 23.31: FibonacciDemo.java
2   // Fibonacci calculations performed synchronousl
3   import java.time.Duration;
4   import java.text.NumberFormat;
5   import java.time.Instant;
6   import java.util.concurrent.CompletableFuture;
7   import java.util.concurrent.ExecutionException;
8
9   // class that stores two Instants in time
10  class TimeData {
11     public Instant start;
12     public Instant end;
13
14     // return total time in seconds
15     public double timeInSeconds() {
16        return Duration.between(start, end).toMill
17     }
18  }
19
20  public class FibonacciDemo {
21     public static void main(String[] args)
22        throws InterruptedException, ExecutionExce
23
24        // perform synchronous fibonacci(45) and f
25        System.out.println("Synchronous Long Runni
26        TimeData synchronousResult1 = startFibonac
27        TimeData synchronousResult2 = startFibonac
28        double synchronousTime =
29           calculateTime(synchronousResult1, synch
30        System.out.printf(
31           "  Total calculation time = %.3f second
32
```

```
33          // perform asynchronous fibonacci(45) and
34          System.out.printf("%nAsynchronous Long Run
35          CompletableFuture<TimeData> futureResult1
36             CompletableFuture.supplyAsync(() -> sta
37          CompletableFuture<TimeData> futureResult2
38             CompletableFuture.supplyAsync(() -> sta
                       39
40          // wait for results from the asynchronous
41          TimeData asynchronousResult1 = futureResul
42          TimeData asynchronousResult2 = futureResul
       43          double asynchronousTime =
44             calculateTime(asynchronousResult1, asyn
          45          System.out.printf(
46             "  Total calculation time = %.3f second
                       47
48          // display time difference as a percentage
49          String percentage = NumberFormat.getPercen
50             (synchronousTime - asynchronousTime) /
51          System.out.printf("%nSynchronous calculati
52             " more time than the asynchronous ones%
                     53       }
                       54
55       // executes function fibonacci asynchronously
56       private static TimeData startFibonacci(int n)
57          // create a TimeData object to store times
 58          TimeData timeData = new TimeData();
                       59
60          System.out.printf(" Calculating fibonacci(
   61          timeData.start = Instant.now();
 62          long fibonacciValue = fibonacci(n);
    63          timeData.end = Instant.now();
64          displayResult(n, fibonacciValue, timeData)
           65          return timeData;
                     66       }
                       67
68       // recursive method fibonacci; calculates nth
 69       private static long fibonacci(long n) {
        70          if (n == 0 || n == 1) {
             71             return n;
                     72          }
```

```
                else {
            return fibonacci(n - 1) + fibonacci(n
                }
            }

        // display fibonacci calculation result and
        private static void displayResult(
        int n, long value, TimeData timeData) {

        System.out.printf(" fibonacci(%d) = %d%n"
            System.out.printf(
          "  Calculation time for fibonacci(%d)
            n, timeData.timeInSeconds());
            }

        // display fibonacci calculation result and
        private static double calculateTime(
        TimeData result1, TimeData result2) {

        TimeData bothThreads = new TimeData();

          // determine earlier start time
        bothThreads.start = result1.start.compare
            result1.start : result2.start;

          // determine later end time
        bothThreads.end = result1.end.compareTo(r
            result1.end : result2.end;

        return bothThreads.timeInSeconds();
            }
        }
```

```
Synchronous Long Running Calculations
Calculating fibonacci(45)
fibonacci(45) = 1134903170
Calculation time for fibonacci(45) = 4.395 seconds
Calculating fibonacci(44)
```

```
                fibonacci(44) = 701408733
    Calculation time for fibonacci(44) = 2.722 seconds
         Total calculation time = 7.122 seconds

        Asynchronous Long Running Calculations
                Calculating fibonacci(45)
                Calculating fibonacci(44)
                fibonacci(44) = 701408733
    Calculation time for fibonacci(44) = 2.707 seconds
               fibonacci(45) = 1134903170
    Calculation time for fibonacci(45) = 4.403 seconds
         Total calculation time = 4.403 seconds

  Synchronous calculations took 62% more time than the
```

Fig. 23.31

Fibonacci calculations performed synchronously and asynchronously.

Class `TimeData`

Class `TimeData` (lines 10–18) stores two `Instant`s representing the start and end time of a task, and provides method `timeInSeconds` to calculate the total time between them. We use `TimeData` objects throughout this example to calculate the time required to perform Fibonacci calculations.

Method `startFibonacci` for Performing and Timing Fibonacci Calculations

Method `startFibonacci` (lines 56–66) is called several times in `main` (lines 26, 27, 36 and 38) to initiate Fibonacci calculations and to calculate the time each calculation requires. The method receives the Fibonacci number to calculate and performs the following tasks:

- Line 58 creates a `TimeData` object to store the calculation's start and end times.
- Line 60 displays the Fibonacci number to be calculated.
- Line 61 stores the current time before method `fibonacci` is called.
- Line 62 calls method `fibonacci` to perform the calculation.
- Line 63 stores the current time after the call to `fibonacci` completes.
- Line 64 displays the result and the total time required for the calculation.
- Line 65 returns the `TimeData` object for use in method `main`.

Performing Fibonacci Calculations Synchronously

Method `main` first demonstrates synchronous Fibonacci calculations. Line 26 calls `startFibonacci(45)` to initiate the `fibonacci(45)` calculation and store the `TimeData` object containing the calculation's start and end

times. When this call completes, line 27 calls `startFibonacci(44)` to initiate the `fibonacci(44)` calculation and store its `TimeData`. Next, lines 28–29 pass both `TimeData` objects to method `calculateTime` (lines 89–103), which returns the total calculation time in seconds. Lines 30–31 display the total calculation time for the synchronous Fibonacci calculations.

Performing Fibonacci Calculations Asynchronously

Lines 35–38 in `main` launch the asynchronous Fibonacci calculations in separate threads. `CompletableFuture` `static` method `supplyAsync` executes an asynchronous task that returns a value. The method receives as its argument an object that implements interface `Supplier`—in this case, we use lambdas with empty parameter lists to invoke `startFibonacci(45)` (line 36) and `startFibonacci(44)` (line 38). The compiler infers that `supplyAsync` returns a `CompletableFuture<TimeData>` because method `startFibonacci` returns type `TimeData`. Class `CompletableFuture` also provides `static` method `runAsync` to execute an asynchronous task that does not return a result—this method receives a `Runnable`.

Getting the Asynchronous Calculations' Results

Class `CompletableFuture` implements interface `Future`, so we can obtain the asynchronous tasks' results by calling `Future` method `get` (lines 41–42). These are *blocking* calls—they cause the `main` thread to *wait* until the asynchronous tasks complete and return their results. In our case, the results are `TimeData` objects. Once both tasks return, lines 43–44 pass both `TimeData` objects to method `calculateTime` (lines 89–103) to get the total calculation time in seconds. Then, lines 45–46 display the total calculation time for the asynchronous Fibonacci calculations. Finally, lines 49–52 calculate and display the percentage difference in execution time for the synchronous and asynchronous calculations.

Program Outputs

On our quad-core computer, the synchronous calculations took a total of 7.122 seconds. Though the individual asynchronous calculations took approximately the same amount of time as the corresponding synchronous calculations, the total time for the asynchronous calculations was only 4.403 seconds, because the two calculations were actually performed *in parallel*. As you can see in the output, the synchronous calculations took 62% more time to complete, so asynchronous execution provided a significant performance

improvement.