Completing and linking tasks asynchronously

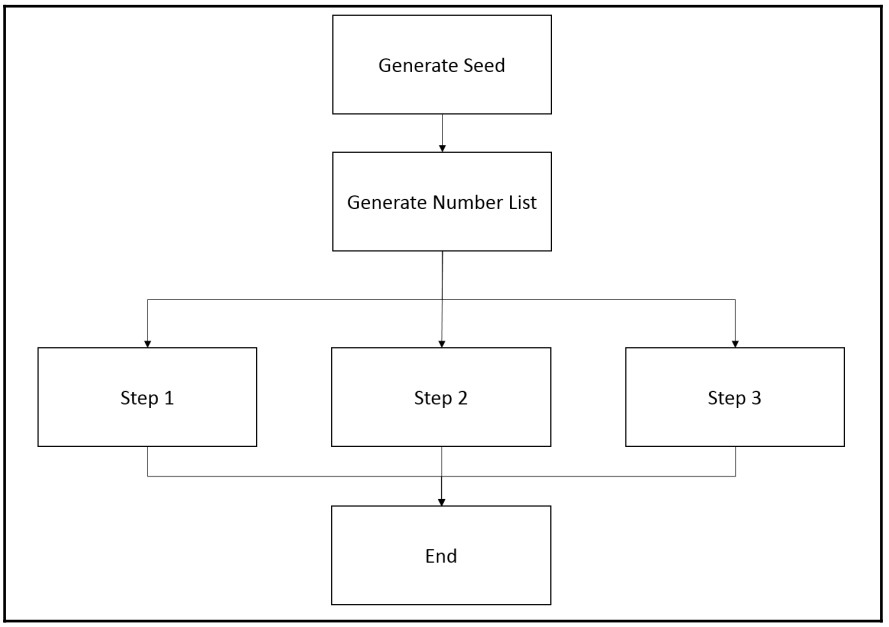
Java 8 Concurrency API includes a new synchronization mechanism with the `CompletableFuture` class. This class implements the `Future` object and the `CompletionStage` interface that gives it the following two characteristics:

- As the `Future` object, a `CompletableFuture` object will return a result sometime in future
- As the `CompletionStage` object, you can execute more asynchronous tasks after the completion of one or more `CompletableFuture` objects

You can work with a `CompletableFuture` class in different ways:

- You can create a `CompletableFuture` object explicitly and use it as a synchronization point between tasks. One task will establish the value returned by `CompletableFuture`, using the `complete()` method, and the other tasks will wait for this value, using the `get()` or `join()` methods.
- You can use a static method of the `CompletableFuture` class to execute `Runnable` or `Supplier` with the `runAsync()` and `supplyAsync()` methods. These methods will return a `CompletableFuture` object that will be completed when these tasks end their execution. In the second case, the value returned by `Supplier` will be the completion value of `CompletableFuture`.
- You can specify other tasks to be executed in an asynchronous way after the completion of one or more `CompletableFuture` objects. This task can implement the `Runnable`, `Function`, `Consumer` or `BiConsumer` interfaces.

These characteristics make the `CompletableFuture` class very flexible and powerful. In this chapter, you will learn how to use this class to organize different tasks. The main purpose of the example is that the tasks will be executed, as specified in the following diagram:



First, we're going to create a task that will generate a seed. Using this seed, the next task will generate a list of random numbers. Then, we will execute three parallel tasks:

1. Step 1 will calculate the nearest number to 1,000, in a list of random numbers.
2. Step 2 will calculate the biggest number in a list of random numbers.
3. Step 3 will calculate the average number between the largest and smallest numbers in a list of random numbers.

Getting ready

The example of this recipe has been implemented using the Eclipse IDE. If you use Eclipse or a different IDE, such as NetBeans, open it and create a new Java project.

How to do it...

Follow these steps to implement the example:

1. First, we're going to implement the auxiliary tasks we will use in the example. Create a class named `SeedGenerator` that implements the `Runnable` interface. It will have a `CompletableFuture` object as an attribute, and it will be initialized in the constructor of the class:

```
public class SeedGenerator implements Runnable {

  private CompletableFuture<Integer> resultCommunicator;

  public SeedGenerator (CompletableFuture<Integer> completable) {
    this.resultCommunicator=completable;
  }
```

2. Then, implement the `run()` method. It will sleep the current thread for 5 seconds (to simulate a long operation), calculate a random number between 1 and 10, and then use the `complete()` method of the `resultCommunicator` object to complete `CompletableFuture`:

```
@Override
public void run() {

  System.out.printf("SeedGenerator: Generating seed...\n");
  // Wait 5 seconds
  try {
    TimeUnit.SECONDS.sleep(5);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
    int seed=(int) Math.rint(Math.random() * 10);

    System.out.printf("SeedGenerator: Seed generated: %d\n",
                      seed);

    resultCommunicator.complete(seed);
  }
```

3. Create a class named `NumberListGenerator` that implements the `Supplier` interface parameterized with the `List<Long>` data type. This means that the `get()` method provided by the `Supplier` interface will return a list of large numbers. This class will have an integer number as a private attribute, which will be initialized in the constructor of the class:

```
public class NumberListGenerator implements Supplier<List<Long>> {

  private final int size;

  public NumberListGenerator (int size) {
    this.size=size;
  }
```

4. Then, implement the `get()` method that will return a list with millions of numbers, as specified in the size parameter of larger random numbers:

```
  @Override
  public List<Long> get() {
    List<Long> ret = new ArrayList<>();
    System.out.printf("%s : NumberListGenerator : Start\n",
                      Thread.currentThread().getName());

    for (int i=0; i< size*1000000; i++) {
      long number=Math.round(Math.random()*Long.MAX_VALUE);
      ret.add(number);
    }
    System.out.printf("%s : NumberListGenerator : End\n",
                      Thread.currentThread().getName());

    return ret;
  }
```

5. Finally, create a class named `NumberSelector` that implements the `Function` interface parameterized with the `List<Long>` and `Long` data types. This means that the `apply()` method provided by the `Function` interface will receive a list of large numbers and will return a `Long` number:

```
public class NumberSelector implements Function<List<Long>, Long> {

  @Override
  public Long apply(List<Long> list) {

    System.out.printf("%s: Step 3: Start\n",
                      Thread.currentThread().getName());
    long max=list.stream().max(Long::compare).get();
    long min=list.stream().min(Long::compare).get();
    long result=(max+min)/2;
    System.out.printf("%s: Step 3: Result - %d\n",
                      Thread.currentThread().getName(), result);
    return result;
  }
}
```

6. Now it's time to implement the `Main` class and the `main()` method:

```
public class Main {
  public static void main(String[] args) {
```

7. First, create a `CompletableFuture` object and a `SeedGenerator` task and execute it as a `Thread`:

```
System.out.printf("Main: Start\n");
CompletableFuture<Integer> seedFuture = new CompletableFuture<>();
Thread seedThread = new Thread(new SeedGenerator(seedFuture));
seedThread.start();
```

8. Then, wait for the seed generated by the `SeedGenerator` task, using the `get()` method of the `CompletableFuture` object:

```
System.out.printf("Main: Getting the seed\n");
int seed = 0;
try {
  seed = seedFuture.get();
} catch (InterruptedException | ExecutionException e) {
  e.printStackTrace();
}
System.out.printf("Main: The seed is: %d\n", seed);
```

9. Now create another `CompletableFuture` object to control the execution of a `NumberListGenerator` task, but in this case, use the static method `supplyAsync()`:

```
System.out.printf("Main: Launching the list of numbers
                   generator\n");
NumberListGenerator task = new NumberListGenerator(seed);
CompletableFuture<List<Long>> startFuture = CompletableFuture
                                            .supplyAsync(task);
```

10. Then, configure the three parallelized tasks that will make calculations based on the list of numbers generated in the previous task. These three steps can't start their execution until the `NumberListGenerator` task has finished its execution, so we use the `CompletableFuture` object generated in the previous step and the `thenApplyAsync()` method to configure these tasks. The first two steps are implemented in a functional way, and the third one is an object of the `NumberSelector` class:

```
System.out.printf("Main: Launching step 1\n");
CompletableFuture<Long> step1Future = startFuture
                                      .thenApplyAsync(list -> {
  System.out.printf("%s: Step 1: Start\n",
                    Thread.currentThread().getName());
  long selected = 0;
  long selectedDistance = Long.MAX_VALUE;
  long distance;
  for (Long number : list) {
    distance = Math.abs(number - 1000);
    if (distance < selectedDistance) {
      selected = number;
      selectedDistance = distance;
    }
  }
  System.out.printf("%s: Step 1: Result - %d\n",
                    Thread.currentThread().getName(), selected);
  return selected;
});

System.out.printf("Main: Launching step 2\n");
CompletableFuture<Long> step2Future = startFuture
.thenApplyAsync(list -> list.stream().max(Long::compare).get());

CompletableFuture<Void> write2Future = step2Future
                                      .thenAccept(selected -> {
  System.out.printf("%s: Step 2: Result - %d\n",
                    Thread.currentThread().getName(), selected);
});

System.out.printf("Main: Launching step 3\n");
NumberSelector numberSelector = new NumberSelector();
CompletableFuture<Long> step3Future = startFuture
                                  .thenApplyAsync(numberSelector);
```

11. We wait for the finalization of the three parallel steps with the `allOf()` static method of the `CompletableFuture` class:

```
System.out.printf("Main: Waiting for the end of the three
                   steps\n");
CompletableFuture<Void> waitFuture = CompletableFuture
                                .allOf(step1Future, write2Future,
                                       step3Future);
```

12. Also, we execute a final step to write a message in the console:

```
CompletableFuture<Void> finalFuture = waitFuture
                                      .thenAcceptAsync((param) -> {
  System.out.printf("Main: The CompletableFuture example has
                     been completed.");
});
finalFuture.join();
```

How it works...

We can use a `CompletableFuture` object with two main purposes:

- Wait for a value or an event that will be produced in future (creating an object and using the `complete()` and `get()` or `join()` methods).
- To organize a set of tasks to be executed in a determined order so one or more tasks won't start their execution until others have finished their execution.

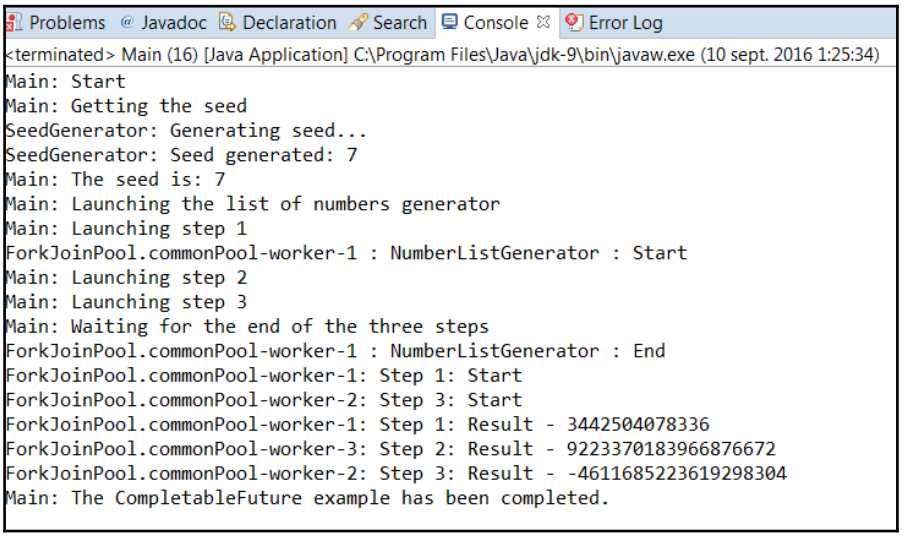
In this example, we made both uses of the `CompletableFuture` class. First, we created an instance of this class and sent it as a parameter to a `SeedGenerator` task. This task uses the `complete()` method to send the calculated value, and the `main()` method uses the `get()` method to obtain the value. The `get()` method sleeps the current thread until `CompletableFuture` has been completed.

Then, we used the `supplyAsync()` method to generate a `CompletableFuture` object. This method receives an implementation of the `Supplier` interface as a parameter. This interface provides the `get()` method that must return a value. The `supplyAsync()` method returns `CompletableFuture`, which will be completed when the `get()` method finishes its execution; the value of completion is the value returned by that method. The `CompletableFuture` object returned will be executed by a task in the `ForkJoinPool` returns the static method `commonPool()`.

Then, we used the `thenApplyAsync()` method to link some tasks. You call this method in a `CompletableFuture` object, and you must pass an implementation of the `Function` interface as a parameter that can be expressed directly in the code using a functional style or an independent object. One powerful characteristic is that the value generated by `CompletableFuture` will be passed as a parameter to the `Function`. That is to say, in our case, all the three steps will receive a random list of numbers as parameters. The `CompletableFuture` class returned will be executed by a task in the `ForkJoinPool` returns the static method `commonPool()`.

Finally, we used the `allOf()` static method of the `CompletableFuture` class to wait for the finalization of various tasks. This method receives a variable list of `CompletableFuture` objects and returns a `CompletableFuture` class that will be completed when all the `CompletableFuture` class passed as parameters are completed. We also used the `thenAcceptAsync()` method as another way to synchronize tasks because this method receives `Consumer` as a parameter that is executed by the default executor when the `CompletableFuture` object used to call the method is completed. Finally, we used the `join()` method to wait for the finalization of the last `CompletableFuture` object.

The following screenshot shows the execution of the example. You can see how the tasks are executed in the order we organized:



There's more...

In the example of this recipe, we used the `complete()`, `get()`, `join()`, `supplyAsync()`, `thenApplyAsync()`, `thenAcceptAsync()`, and `allOf()` methods of the `CompletableFuture` class. However, this class has a lot of useful methods that help increase the power and flexibility of this class. These are the most interesting ones:

- Methods to complete a `CompletableFuture` object: In addition to the `complete()` method, the `CompletableFuture` class provides the following three methods:
    - `cancel()`: This completes `CompletableFuture` with a `CancellationException` exception.
    - `completeAsync()`: This completes `CompletableFuture` with the result of the `Supplier` object passed as a parameter. The `Supplier` object is executed in a different thread by the default executor.
    - `completeExceptionally()`: This method completes `CompletableFuture` with the exception passed as a parameter.
- Methods to execute a task: In addition to the `supplyAsync()` method, the `CompletableFuture` class provides the following method:
    - `runAsync()`: This is a static method of the `CompletableFuture` class that returns a `CompletableFuture` object. This object will be completed when the `Runnable` interface is passed as a parameter to finish its execution. It will be completed with a void result.
- Methods to synchronize the execution of different tasks: In addition to the `allOf()`, `thenAcceptAsync()`, and `thenApplyAsync()` methods, the `CompletableFuture` class provides the following methods to synchronize the execution of tasks:
    - `anyOf()`: This is a static method of the `CompletableFuture` class. It receives a list of `CompletableFuture` objects and returns a new `CompletableFuture` object. This object will be completed with the result of the first `CompletableFuture` parameter that is completed.

- `runAfterBothAsync()`: This method receives `CompletionStage` and `Runnable` objects as parameters and returns a new `CompletableFuture` object. When `CompletableFuture` (which does the calling) and `CompletionStage` (which is received as a parameter) are completed, the `Runnable` object is executed by the default executor and then the `CompletableFuture` object returned is completed.
- `runAfterEitherAsync()`: This method is similar to the previous one, but here, the `Runnable` interface is executed after one of the two (`CompletableFuture` or `CompletionStage`) are completed.
- `thenAcceptBothAsync()`: This method receives `CompletionStage` and `BiConsumer` objects as parameters and returns `CompetableFuture` as a parameter. When `CompletableFuture` (which does the calling) and `CompletionStage` (which is passed as a parameter), `BiConsumer` is executed by the default executor. It receives the results of the two `CompletionStage` objects as parameters but it won't return any result. When `BiConsumer` finishes its execution, the returned `CompletableFuture` class is completed without a result.
- `thenCombineAsync()`: This method receives a `CompletionStage` object and a `BiFunction` object as parameters and returns a new `CompletableFuture` object. When `CompletableFuture` (which does the calling) and `CompletionStage` (which is passed as a parameter) are completed, the `BiFunction` object is executed; it receives the completion values of both the objects and returns a new result that will be the completion value of the returned `CompletableFuture` class.
- `thenComposeAsync()`:This method is analogous to `thenApplyAsync()`, but it is useful when the supplied function returns `CompletableFuture` too.
- `thenRunAsync()`: This method is analogous to the `thenAcceptAsync()` method, but in this case, it receives a `Runnable` object as a parameter instead of a `Consumer` object.

- Methods to obtain the completion value: In addition to the `get()` and `join()` methods, the `CompletableFuture` object provides the following method to get the completion value:
    - `getNow()`: This receives a value of the same type of the completion value of `CompletableFuture`. If the object is completed, it returns the completion value. Else, it returns the value passed as the parameter.

See also...

- The *Creating a thread executor and controlling its rejected tasks* and *Executing tasks in an executor that returns a result* recipes in `Chapter 4`, *Thread Executors*