Completing and linking tasks asynchronously

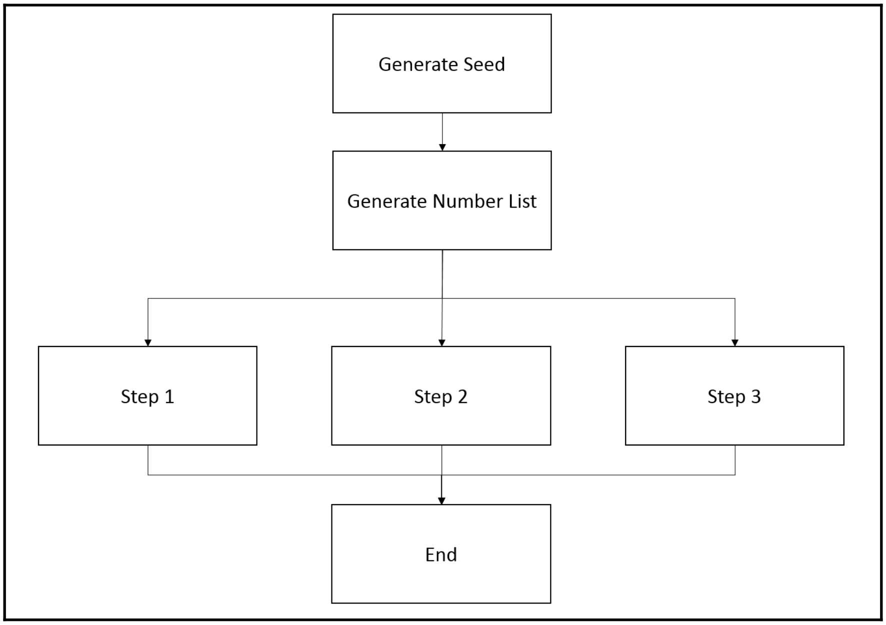
Java 8 Concurrency API includes a new synchronization mechanism with the `CompletableFuture` class. This class implements the `Future` object and the `CompletionStage` interface that gives it the following two characteristics:

- 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: