The CompletableFuture class

This is a synchronization mechanism introduced in the Java 8 concurrency API that has new methods in Java 9. It extends the `Future` mechanism, giving it more power and flexibility. It allows you to implement an event-driven model, linking tasks that will only be executed when others have finished. As with the `Future` interface, `CompletableFuture` must be parameterized with the type of the result that will be returned by the operation. As with a `Future` object, the `CompletableFuture` class represents a result of an asynchronous computation, but the result of `CompletableFuture` can be established by any thread. It has the `complete()` method to establish the result when the computation ends normally and the method `completeExceptionally()` when the computation ends with an exception. If two or more threads call the `complete()` or `completeExceptionally()` methods over the same `CompletableFuture`, only the first call will take effect.

First, you can create `CompletableFuture` using its constructor. In this case, you have to establish the result of the task using the `complete()` method, as we explained before. But you can also create one using the `runAsync()` or `supplyAsync()` methods. The `runAsync()` method executes a `Runnable` object and returns `CompletableFuture<Void>` so that computation can't return any results. The `supplyAsync()` method executes an implementation of the `Supplier` interface parametrized with the type that will be returned by this computation. The `Supplier` interface provides the `get()` method. In that method, we have to include the code of the task and return the result generated by it. In this case, the result of `CompletableFuture` will be the result of the `Supplier` interface.

This class provides a lot of methods that allow you to organize the order of execution of tasks implementing an event-driven model, where one task doesn't start its execution until the previous one has finished. These are some of those methods:

- `thenApplyAsync()`: This method receives an implementation of the `Function` interface that can be represented as a lambda expression as a parameter. This function will be executed when the calling `CompletableFuture` has been completed. This method will return `CompletableFuture` to get the result of the

`Function`.

- `thenComposeAsync()`: This method is analogue to `thenApplyAsync`, but is useful when the supplied function returns `CompletableFuture` too.
- `thenAcceptAsync()`: This method is similar to the previous one, but the parameter is an implementation of the `Consumer` interface that can also be specified as a lambda expression; in this case, the computation won't return a result.
- `thenRunAsync()`: This method is equivalent to the previous one, but in this case receives a `Runnable` object as a parameter.
- `thenCombineAsync()`: This method receives two parameters. The first one is another `CompletableFuture` instance. The other is an implementation of the `BiFunction` interfaces that can be specified as a lambda function. This `BiFunction` will be executed when both `CompletableFuture` (the calling one and the parameter) have been completed. This method will return `CompletableFuture` to get the result of the `BiFunction`.
- `runAfterBothAsync()`: This method receives two parameters. The first one is another `CompletableFuture`. The other one is an implementation of the `Runnable` interface that will be executed when both `CompletableFuture` (the calling one and the parameter) have been completed.
- `runAfterEitherAsync()`: This method is equivalent to the previous one, but the `Runnable` task is executed when one of the `CompletableFuture` objects is completed.
- `allOf()`: This method receives a variable list of `CompletableFuture` objects as a parameter. It will return a `CompletableFuture<Void>` object that will return its result when all the `CompletableFuture` objects have been completed.
- `anyOf()`: This method is equivalent to the previous one, but the returned `CompletableFuture` returns its result when one of the `CompletableFuture` is completed.

Finally, if you want to obtain the result returned by `CompletableFuture`, you can use the `get()` or `join()` methods. Both methods block the calling thread until `CompletableFuture` has been completed and then returns its result. The main difference between both methods is that `get()` throws `ExecutionException`, which is a checked exception, but `join()` throws `RuntimeException` (which is an unchecked exception). Thus, it's easier to use `join()` inside non-throwing lambdas (like `Supplier`, `Consumer`, or `Runnable`).

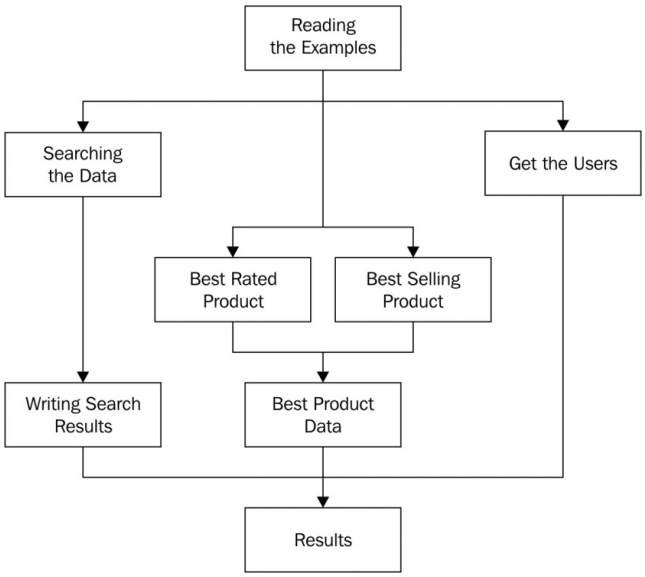
Most of the methods explained before have the `Async` suffix. This means that these methods will be executed in a concurrent way using the `ForkJoinPool.commonPool` instance. Those methods that have versions without the `Async` suffix will be executed in a serial way (that is to say, in the same thread where `CompletableFuture` is executed) and with the `Async` suffix and an executor instance as an additional parameter. In this case, `CompletableFuture` will be executed asynchronously in the executor passed as a parameter.

Java 9 has added some methods to give more power to the `CompletableFuture` class.

- `defaultExecutor()`: This method returns the default `Executor` used for `Async` operations that don't receive an `Executor` as a parameter. Normally, it will be the returned value of the `ForkJoinPool.commonPool()` method.
- `copy()`: This method creates a copy of a `CompletableFuture` object. If the original `CompletableFuture` completes normally, the copy will also be completed normally with the same value. If the original `CompletableFuture` completes exceptionally, the copy completes exceptionally with a `CompletionException`.
- `completeAsync()`: This method receives a `Supplier` object as a parameter (and optionally, an `Executor`). Completes the `CompletableFuture` with the result of the `Supplier`.
- `orTimeout()`: Receives a timeout (a period of time and a `TimeUnit`). If the `CompletableFuture` is not completed after that period of time, completes exceptionally with a `TimeoutException`.
- `completeOnTimeout()`: This method is similar to the previous one, but it completes normally with the value received as a parameter.
- `delayedExecutor()`: This method returns an `Executor` that executes a task after the specified delay.

Using the CompletableFuture class

In this example, you will learn how to use the `CompletableFuture` class to implement the execution of some asynchronous tasks in a concurrent way. We will use our collection of 20,000 products from Amazon to implement the following tree of tasks:



First, we're going to use the examples. Then, we will execute four concurrent tasks. The first one will make a search of products. When the search finishes, we will write the results to a file. The second one will obtain the best-rated product. The third one will obtain the best-selling product. When these both finish, we will concatenate their information using another task. Finally, the fourth task will get a list with the users who have purchased a product. The `main()` program will wait for the finalization of all the tasks and then will write the results.

Let's see the details of the implementation.

Auxiliary tasks

In this example, we will use some auxiliary tasks. The first one is `LoadTask` that will load the product information from the disk and will return a list of `Product` objects:

```
public class LoadTask implements Supplier<List<Product>> {

  private Path path;

  public LoadTask (Path path) {
    this.path=path;
  }
  @Override
  public List<Product> get() {
    List<Product> productList=null;
    try {
      productList = Files.walk(path, FileVisitOption.FOLLOW_LINKS)
                         .parallel().filter(f -> f.toString()
                         .endsWith(".txt")).map(ProductLoader::load)
                         .collect (Collectors.toList());
    } catch (IOException e) {
      e.printStackTrace();
    }

    return productList;
  }
}
```

It implements the `Supplier` interface to be executed as `CompletableFuture`. Inside, it uses a stream to process and parse all the files obtaining a list of products.

The second task is `SearchTask` that will implement the search in the list of `Product` objects, looking for the ones that contain a word in the title. This task is an implementation of the `Function` interface.

```
public class SearchTask implements Function<List<Product>,
                                    List<Product>> {

  private String query;

  public SearchTask(String query) {
    this.query=query;
  }

  @Override
  public List<Product> apply(List<Product> products) {
    System.out.println(new Date()+": CompletableTask: start");
    List<Product> ret = products.stream()
                          .filter(product -> product.getTitle()
                          .toLowerCase().contains(query))
```

```
                          .collect(Collectors.toList());
    System.out.println(new Date()+": CompletableTask: end:
                       "+ret.size());
    return ret;
  }

}
```

It receives `List<Product>` with the information of all the products and returns `List<Product>` with the products that meet the criteria. Internally, it creates the stream on the input list, filters it, and collects the results in another list.

Finally, the `WriteTask` is going to write the products obtained in the search task in a `File`. In our case, we generate a HTML file, but feel free to write this information in the format you want. This task implements the `Consumer` interface, so its code must be something like the following:

```
public class WriteTask implements Consumer<List<Product>> {

  @Override
  public void accept(List<Product> products) {
    // implementation is omitted
  }
}
```

The main() method

We have organized the execution of the tasks in the `main()` method. First, we execute the `LoadTask` using the `supplyAsync()` method of the `CompletableFuture` class. We are going to wait three seconds before the start of the `LoadTask` to show how the `delayExecutor()` method works.

```
public class CompletableMain {

  public static void main(String[] args) {
    Path file = Paths.get("data","category");
    System.out.println(new Date() + ": Main: Loading products
                       after three seconds....");
    LoadTask loadTask = new LoadTask(file);

    CompletableFuture<List<Product>>loadFuture = CompletableFuture
                       .supplyAsync(loadTask,CompletableFuture
                       .delayedExecutor(3, TimeUnit.SECONDS));
```

Then, with the resultant `CompletableFuture`, we use `thenApplyAsync()` to execute the search task when the load task has been completed:

```
System.out.println(new Date() + ": Main: Then apply for
                   search");

CompletableFuture<List<Product>> completableSearch = loadFuture
                      .thenApplyAsync(new SearchTask("love"));
```

Once the search task has been completed, we want to write the results of the execution in a file. As this task won't return a result, we use the `thenAcceptAsync()` method:

```
CompletableFuture<Void> completableWrite = completableSearch
                       .thenAcceptAsync(new WriteTask());

completableWrite.exceptionally(ex -> {
  System.out.println(new Date() + ": Main: Exception "
                     + ex.getMessage());
  return null;
});
```

We have used the `exceptionally()` method to specify what we want to do if the write task throws an exception.

Then, we use the `thenApplyAsync()` method over the `completableFuture` object to execute the task to get the list of users who purchased a product. We specify this

task as a lambda expression. Take into account that this task will be executed in parallel with the search task:

```
System.out.println(new Date() + ": Main: Then apply for users");

CompletableFuture<List<String>> completableUsers = loadFuture
                                .thenApplyAsync(resultList -> {

  System.out.println(new Date() + ": Main: Completable users :start");
  List<String> users = resultList.stream()
                       .flatMap(p -> p.getReviews().stream())
                       .map(review -> review.getUser())
                       .distinct()
                       .collect(Collectors.toList());
  System.out.println(new Date() + ": Main: Completable users :end");

  return users;
});
```

In parallel with these tasks, we also executed the tasks using the `thenApplyAsync()` method to find the best-rated product and the best-selling product. We have defined these tasks using a lambda expression too:

```
  System.out.println(new Date() + ": Main: Then apply for best
                                  rated product....");

  CompletableFuture<Product> completableProduct = loadFuture
                            .thenApplyAsync(resultList -> {
    Product maxProduct = null;
    double maxScore = 0.0;

    System.out.println(new Date() + ": Main: Completable product:
                                    start");
    for (Product product : resultList) {
    if (!product.getReviews().isEmpty()) {
      double score = product.getReviews().stream()
                     .mapToDouble(review -> review.getValue())
                     .average().getAsDouble();
      if (score > maxScore) {
        maxProduct = product;
        maxScore = score;
      }
    }
  }
  System.out.println(new Date() + ": Main: Completable product : end");
  return maxProduct;
});

System.out.println(new Date() + ": Main: Then apply for best
                                 selling product....");
CompletableFuture<Product> completableBestSellingProduct =
                  loadFuture.thenApplyAsync(resultList -> {
  System.out.println(new Date() + ": Main: Completable best
                                  selling: start");
  Product bestProduct = resultList.stream()
                        .min(Comparator.comparingLong
                             (Product::getSalesrank))
                        .orElse(null);
  System.out.println(new Date() + ": Main: Completable best
```

```
                                    selling: end");
  return bestProduct;

});
```

As we mentioned before, we want to concatenate the results of the last two tasks. We can do this using the `thenCombineAsync()` method to specify a task that will be executed after both tasks have been completed:

```
CompletableFuture<String> completableProductResult =
                    completableBestSellingProduct
                    .thenCombineAsync(
                    completableProduct, (bestSellingProduct,
                                         bestRatedProduct) -> {
  System.out.println(new Date() + ": Main: Completable product
                                    result: start");
  String ret = "The best selling product is "
              + bestSellingProduct.getTitle() + "\n";
  ret += "The best rated product is "
         + bestRatedProduct.getTitle();
  System.out.println(new Date() + ": Main: Completable product
                                    result: end");
  return ret;
});
```

Finally, we give one second to the `completableProductResult` task to finish using the `completeOnTimeout()` method. If it doesn't finish before one second, we complete that `CompletableFuture` with the result `"TimeOut"`. Then, we wait for the end of the final tasks using the `allOf()` and `join()` methods and write the results using the `get()` method to obtain them:

```
System.out.println(new Date() + ": Main: Waiting for results");

completableProductResult.completeOnTimeout("TimeOut", 1,
                                           TimeUnit.SECONDS);
CompletableFuture<Void> finalCompletableFuture = CompletableFuture
                .allOf(completableProductResult, completableUsers,
                       completableWrite);
finalCompletableFuture.join();

try {
  System.out.println("Number of loaded products: "
                     + loadFuture.get().size());
  System.out.println("Number of found products: "
                     + completableSearch.get().size());
  System.out.println("Number of users: "
                     + completableUsers.get().size());
  System.out.println("Best rated product: "
                     + completableProduct.get().getTitle());
  System.out.println("Best selling product: "
                     + completableBestSellingProduct.get()
                     .getTitle());
  System.out.println("Product result:
                     "+completableProductResult.get());
} catch (InterruptedException | ExecutionException e) {
  e.printStackTrace();
```

```
}
```

In the following screenshot, you can see the results of an execution of this example:

```
<terminated> CompletableMain [Java Application] C:\Program Files\Java\jdk-9\bin\javaw.exe (12 abr. 2(
Wed Apr 12 01:37:12 CEST 2017: Main: Loading products after three seconds....
Wed Apr 12 01:37:13 CEST 2017: Main: Then apply for search....
Wed Apr 12 01:37:13 CEST 2017: Main: Then apply for users....
Wed Apr 12 01:37:13 CEST 2017: Main: Then apply for best rated product....
Wed Apr 12 01:37:13 CEST 2017: Main: Then apply for best selling product....
Wed Apr 12 01:37:13 CEST 2017: Main: Waiting for results
Wed Apr 12 01:37:16 CEST 2017: LoadTast: starting....
Wed Apr 12 01:38:19 CEST 2017: LoadTast: end
Wed Apr 12 01:38:19 CEST 2017: Main: Completable best selling: start
Wed Apr 12 01:38:19 CEST 2017: CompletableTask: start
Wed Apr 12 01:38:19 CEST 2017: Main: Completable product: start
Wed Apr 12 01:38:19 CEST 2017: Main: Completable best selling: end
Wed Apr 12 01:38:19 CEST 2017: Main: Completable users: start
Wed Apr 12 01:38:19 CEST 2017: CompletableTask: end: 208
Wed Apr 12 01:38:19 CEST 2017: WriteTask: start
Wed Apr 12 01:38:19 CEST 2017: WriteTask: end
Wed Apr 12 01:38:19 CEST 2017: Main: Completable product: end
Wed Apr 12 01:38:19 CEST 2017: Main: Completable users: end
Number of loaded products: 20000
Number of found products: 208
Number of users: 158288
Best rated product: Patterns of Preaching
Best selling product: The Da Vinci Code
Product result: TimeOut
Wed Apr 12 01:38:19 CEST 2017: Main: end
```

First, the `main()` method executes all the configurations and waits for the finalization of the tasks. The execution of the tasks follows the order we have configured. You can see how the `LoadTask` starts after three seconds and how the `completableProductResult` returns the `String "TimeOut"`, as it isn't completed in one second.

Summary

In this chapter, we have reviewed two components of all concurrent applications. The first one is data structures. Every program uses them to store in memory the information it has to process. We have quickly been introduced to concurrent data structures to create a detailed description of the new features introduced in the Java 8 Concurrency API that affects the `ConcurrentHashMap` class and the classes that implement the `Collection` interface.

The second one is the synchronization mechanisms that allow you to protect your data when more than one concurrent task wants to modify them, and to control the order of execution of the tasks if it's necessary. In this case, we have also quickly been introduced to the synchronization mechanisms, giving a detailed description of `CompletableFuture`, a new feature of the Java 8 Concurrency API.

In the next chapter, we will show you how you can implement complete concurrent systems, integrating different parts that can also be concurrent and using different classes to implement concurrency.