Programming Assignment #3 (Java)

CS–671

due 16 March 2014 11:59pm

1. Implement the class `cs671.CompletedFuture`. 10 pts
2. Implement the class `cs671.FailedFuture`. 10 pts
3. Implement the class `cs671.FactoryExecutor`. 80 pts

This assignment illustrates two important concepts used when dealing with asynchronous executions: futures and callbacks. When a task is executed asynchronously (e.g., in parallel in another thread), it sometimes needs a mechanism to communicate with the entity that started the task. Futures and callbacks are such mechanisms. This assignment implements a simplified version of the facilities found in `java.util.concurrent.Future` and `scala.concurrent.Future`.

An *executor* is a facility that can execute tasks asynchronously. The `java.util.concurrent` library offers several kinds of executors. In this assignment, we will focus on a simple form of executor that starts a new thread with each task submitted (better executors use queues and pool threads).

In the example from Fig. 1, an executor `exec` is used to start a task asynchronously. First, the task is constructed as a `Runnable` object. It is then submitted to an executor. In this example, the executor starts the task immediately in a new thread. The `execute` method returns immediately (*asynchronous* call). The task and the main thread that created and submitted it run concurrently until both terminate.

In this example the task is run solely for its side effect. It does not produce any output. Fig. 2 shows an example of a task that produces an integer output. Instead of using `Runnable`, this task is specified as a `Callable`, an interface shown in Lis. 1. (It is actually identical to `java.util.concurrent.Callable`.) A *callable* object differs from a *runnable* in that its `call` method returns a value (and can throw checked exceptions).

When the task is submitted, the executor returns a *future*, which is a handle on the asynchronous computations. In general, a future can be used to retrieve a task output, cancel a task, check whether a task has completed, check if the task terminated abruptly, etc. After 5 seconds, the main thread uses the future to check if the task is finished (false). It then calls the `get` method of the future to retrieve the result of the computation. Since the task is still running, the result of the computation is not yet available and the `get` method blocks. When the task completes, the `get` method returns the value produced by the computation. At this point, the `isDone` method true. Further calls to `get` will return the value produced by the computation without further blocking. In other words, the `get` method can be called at any time. If the result of the computation is available, it is returned immediately. Otherwise, the method blocks until the computation finishes.

Callbacks are another approach used to retrieve the result of an asynchronous computation. In the example from Fig. 3, the same task is submitted alongside a callback object. It is the responsibility of the executor to "call back" by running this object after the task completes. Callbacks are specified using the interface from Lis. 2. The executor will either invoke `call` with the result of the computation, of `failure` is the computation terminates abruptly.

Both approaches (futures and callbacks) can be combined, as in the example from Fig. 4. The task is initially submitted without a callback and a future is obtained from the executor. Through the future, the main thread then adds a first callback and, 5 seconds later, a second callback. When the task completes, both callback objects are executed with the result of the computation, which is also available through the future's `get` method.

Both futures and callbacks need to have a way to deal with tasks that terminate abruptly by throwing an error or an exception. The example from Fig. 5 creates a task that fails with an exception. After the failure happens,

```java
package cs671;

public interface Callable<T> {
  public T call () throws Exception;
}
```

Listing 1: `Callable` interface.

```java
Runnable task = new Runnable() { // new task
  public void run () {
    print("task starts");
    try {
      Thread.sleep(5000); // 5 seconds
    } catch (InterruptedException e) {
      Thread.currentThread().interrupt();
    }
    print("task ends");
  }
};

print("submiting task to executor");
exec.execute(task);
for (int i=0; i<10; i++) {
  Thread.sleep(1000); // 1 seconds
  print("main thread running...");
}
print("main thread done");
```

Output:

```
14:35:10.454: submiting task to executor
14:35:10.467: task starts
14:35:11.468: main thread running...
14:35:12.469: main thread running...
14:35:13.471: main thread running...
14:35:14.471: main thread running...
14:35:15.468: task ends
14:35:15.472: main thread running...
14:35:16.473: main thread running...
14:35:17.475: main thread running...
14:35:18.476: main thread running...
14:35:19.477: main thread running...
14:35:20.478: main thread running...
14:35:20.479: main thread done
```

Figure 1: Example 1

```java
Callable<Integer> task = new Callable<Integer>() {
  public Integer call () throws InterruptedException {
    print("task starts");
    Thread.sleep(10000); // 10 seconds
    print("task ends");
    return 42;
  }
};

print("submiting task to executor");
Future<Integer> future = exec.execute(task);
Thread.sleep(5000); // 5 seconds
print("task completed: " + future.isDone());
print("result from future: " + future.get());
print("task completed: " + future.isDone());
print("result from future: " + future.get());
```

Output:

```
14:59:36.686: submiting task to executor
14:59:36.699: task starts
14:59:41.700: task completed: false
14:59:46.701: task ends
14:59:46.702: result from future: 42
14:59:46.702: task completed: true
14:59:46.702: result from future: 42
```

Figure 2: Example 2

```java
package cs671;

public interface Callback<T> {
  public void call (T value);
  public void failure (Throwable t);
}
```

Listing 2: `Callback` interface.

```java
Callback<Number> cb = new Callback<Number>() {
  public void call (Number n) {
    print("called back with " + n);
  }
  public void failure (Throwable t) {
    print("failed with " + t);
  }
};

print("submiting task to executor");
exec.execute(task, cb);
Thread.sleep(5000); // 5 seconds
print("main thread done");
```

Output:

```
15:17:26.401: submiting task to executor
15:17:26.415: task starts
15:17:31.416: main thread done
15:17:36.415: task ends
15:17:36.416: called back with 42
```

Figure 3: Example 3

```java
Callback<Number> cb2 = new Callback<Number>() {
  public void call (Number n) {
    print("also called back with " + n);
  }
  public void failure (Throwable t) {
    print("failed with " + t);
  }
};

print("submiting task to executor");
Future<Integer> future = exec.execute(task);
print("adding first callback");
future.whenComplete(cb);
Thread.sleep(5000); // 5 seconds
print("adding second callback");
future.whenComplete(cb2);
print("task completed: " + future.isDone());
Thread.sleep(10000); // 10 seconds
print("task completed: " + future.isDone());
print("result from future: " + future.get());
print("result from future: " + future.get());
```

Output:

```
15:24:10.807: submiting task to executor
15:24:10.819: adding first callback
15:24:10.819: task starts
15:24:15.821: adding second callback
15:24:15.821: task completed: false
15:24:20.821: task ends
15:24:20.822: called back with 42
15:24:20.822: also called back with 42
15:24:25.822: task completed: true
15:24:25.823: result from future: 42
15:24:25.823: result from future: 42
```

Figure 4: Example 4

```java
Callable<Integer> fail = new Callable<Integer>() {
  public Integer call () throws Exception {
    print("task starts");
    Thread.sleep(5000); // 5 seconds
    print("task fails");
    throw new Exception("Oops!");
  }
};

print("submiting task to executor");
Future<Integer> future = exec.execute(fail);
print("adding callback");
future.whenComplete(cb);
print("result from future: " + future.get());
print("failure from future: " + future.getFailure());
print("task completed: " + future.isDone());
```

Output:

```
15:30:08.273: submiting task to executor
15:30:08.287: adding callback
15:30:08.288: task starts
15:30:13.289: task fails
15:30:13.289: failed with java.lang.Exception: Oops!
15:30:13.289: result from future: null
15:30:13.290: failure: java.lang.Exception: Oops!
15:30:13.290: task completed: true
```

Figure 5: Example 5

```java
package cs671;

public interface Continue<A,B> { // This is basically a function from A to B
  public B call (A a) throws Exception;
}
```

Listing 3: `Continue` interface.

```java
Continue<Number,String> cont = new Continue<Number,String>() {
  public String call (Number n) {
    return "[" + n + "]";
  }
};

print("submiting task to executor");
Future<Integer> future1 = exec.execute(task);
Thread.sleep(5000); // 5 seconds
print("task completed: " + future1.isDone());
Future<String> future2 = future1.whenComplete(cont);
print("continuation completed: " + future2.isDone());
print("result from continuation: " + future2.get());
print("task completed: " + future1.isDone());
print("result from future: " + future1.get());
```

Output:

```
15:43:44.458: submiting task to executor
15:43:44.471: task starts
15:43:49.471: task completed: false
15:43:49.472: continuation completed: false
15:43:54.472: task ends
15:43:54.473: result from continuation: [42]
15:43:54.473: task completed: true
15:43:54.474: result from future: 42
```

Figure 6: Example 6

the callback's method `failure` is called with the throwable that caused the failure. At this point, the `get` method from the future returns `null` and the `getFailure` method returns the throwable.

Finally, as a generalization of callbacks, futures can be continued with further computations, which result in additional futures. The example from Fig. 6 obtains a `Future<Integer>` from submitting a task as before. It then creates a continuation that produces a string from a number and attaches it to the future, resulting in a `Future<String>` object. Continuations are specified as instances of the `Continue` type defined in Lis. 3.

When the task completes, its resulting value is then used to start the continuation computation, which finally produces a string. The main thread uses the second future to wait for the completion of this second computation and to retrieve its result. At this point, the first computation is also finished and its result can be obtained through the first future. Note that what to do with the number produced by the first computation was specified before this computation was completed. Thus, the second future is created as a handle on a task that has not even begun yet.

Different programming languages offers different flavors of futures and callbacks (see `java.util.concurrent.Future` and `scala.concurrent.Future` for examples). The futures used in this assignment are defined in Lis. 4 and details can be found in the API.

Futures are created by submitting tasks to an executor (or from other futures). The `Executor` interface is shown in Lis. 5. The first method takes in a task specified as a `callable` and returns a future that will contain the result of the computation (or the reason for failure). The second method is similar but uses `Runnable` to specify the task (the "result" of the computation will be `null`). The last two methods are similar but take a callback object at

```java
package cs671;

public interface Future<T> {
  public boolean isDone ();
  public T get () throws InterruptedException ;
  public Throwable getFailure () throws InterruptedException;
  public void whenComplete (Callback<? super T> callback);
  public void whenComplete (Runnable callback);
  public <U> Future<U> whenComplete (Continue<? super T, ? extends U> cont);
}
```

Listing 4: `Future` interface.

```java
package cs671;

public interface Executor {
  public <T> Future<T> execute (Callable<T> task);
  public <T> Future<T> execute (Runnable task);
  public <T> void execute (Callable<T> task, Callback<? super T> callback);
  public <T> void execute (Runnable task, Callback<T> callback);
}
```

Listing 5: `Executor` interface.

```java
package cs671;

public final class FactoryExecutor implements Executor {

  public static interface ThreadFactory {
    public Thread getThread (Runnable behavior);
  }

  public FactoryExecutor (ThreadFactory factory) { ... }
  ...
```

Listing 6: `FactoryExecutor` class.

submission time and do not return a future.

In this assignment, we will use an executor that creates a new thread with each submitted task. It is implemented as class `FactoryExecutor` from Lis. 6. Instances of this class are created with a thread factory, which is used by the executor to create threads when needed (see sample tests for an example of how to create a thread factory).

The main difficulty in implementing `FactoryExecutor` is to create the future objects returned by the `execute` methods. The state of those objects needs to be updated when the underlying computation finished (set the result value or exception, unblock threads blocked on `get` or `getFailure`, run the callbacks and continuations, etc.). One possible strategy is to return instances of a class that implements *both* `Future<T>` and `Runnable`. The idea is that when the object is run, its state (as a future) is updated. One reference is returned to the client (and used as a future) and one is kept by the executor (to run it). The `run` method of this object is responsible for executing the task, setting the result, running the callbacks and continuations, etc.

Be mindful, however, that this object is shared between the client thread (who calls methods from the `Future` interface) and the executor thread (who executes the task and sets the state of the future accordingly). Therefore, all the methods of this class must be properly synchronized. You need to think of all possible scenarios. For instance, the client thread might call `isDone` while the task is running or might call `whenComplete` in the middle of an execution of a callback.

Classes `CompletedFuture` and `FailedFuture` are used to create futures that are already finished, either with a value or with an exception. They are much easier to implement than the general future and can be written first as a "warm-up" exercise (they can also be used in the implementation of the more general future when appropriate).

Notes:

- APIs of classes, annotations and interfaces are described at: `http://www.cs.unh.edu/~cs671/Java`.

- It is possible to implement some simpler methods as special cases or more complex ones. This results in a concise implementation with very little code duplication. For instance, it is easy to implement `FactoryExecutor.execute(Callable,Callback)` in terms of `FactoryExecutor.execute(Callable)`. However, if implemented independently, the method with the callback is easier to write than the one with the future.

  For this assignment, it is fine to start with the simpler methods to get a better understanding of what needs to be done and to implement the more complex methods later and independently. Ideally, some methods could then be rewritten in terms of others to avoid code duplication, but it is not required to do so.

- Instances of `FactoryExecutor` should use exactly one thread from the factory for each submitted task. This thread is used to run the task and some of the callbacks and continuations (but not necessarily all of them, see below). If more "helper" threads are needed, they should be created directly, not from the factory.

- For the purpose of testing `FactoryExecutor`, it is necessary to create thread factories. The simplest thread factory can be an instance of the following class:

```
class SimpleThreadFactory implements FactoryExecutor.ThreadFactory {
  public Thread getThread (Runnable r) {
    return new Thread(r);
  }
}
```

- After a thread has run a task and its callbacks and continuations, it terminates. Therefore, it is not available to run any callback or continuation added to a future afterward. Different strategies are possible but for this assignment, callbacks or continuations added after the task has completed should be run by the client thread.

  More precisely, all the callbacks and continuations attached to a future while the task was running are run after the task completes by the same thread that ran the task and before the `get` and `getFailure` methods become unblocking and the `isDone` method returns true. Any callback or continuation added after `get` and `getFailure` unblock and `isDone` returns true are run by the client thread. Callbacks and continuations added after the task has finished but before `isDone` is true (e.g., while the existing callbacks and continuations are being executed) must be run exactly once by one of these two threads. Which thread exactly runs them is unspecified.

  As an example, consider the code below:

```
Thread me = Thread.currentThread();
Future<?> future = exec.execute(task);
future.whenComplete(cb1);
future.get();
future.whenComplete(cb2);
```

  In this example, `cb1` must be run by an executor thread, but `cb2` is run by `me` (assuming `task` is not so short that it is completed before the first call to `whenComplete`).

- As a special case of the previous item, callbacks and continuations attached to an instance of `CompletedFuture` or `FailedFuture` are run by the client thread.

- The order in which callbacks and continuations are run is not specified.

- Exceptions and errors thrown by tasks and continuations must be caught and properly handled. In contrast, it can be assumed that callbacks do not throw. If a callback fails with an exception, it is valid to let this exception kill the running thread (which means that some other callbacks or continuations may not run).

- If a task fails, its continuations are not executed (there is no value to execute them on). Instead, they themselves fail with a `NotExecutedException` that contains the initial failure as its cause.

  As an example, consider the code below:

```
Future<Integer> future1 = exec.execute(task);
Future<String> future2 = future1.whenComplete(cont);
if (future1.getFailure() != null) { // task failed
  Throwable t1 = future1.getFailure();
  Throwable t2 = future2.getFailure();
  assertTrue(t2 instanceof NotExecutedException);
  assertSame(t2.getCause(), t1);
}
```