Programming Assignment #2

CS–735/835

due 26 Feb 2017 11:59 PM

1 Introduction

This assignment implements a form of "background computation" that can be started to introduce parallelism. Newly created computations typically run in their own thread, referred to below as the "computation thread". The functionalities offered by these computation are based on Java's `Future` interface, Google's `ListenableFuture` interface and Scala's `Future` trait:

- A computation is created with a task to run, specified as a `Callable`.
- After completion, the output of the task can be retrieved via a `get` method.
- While the task is running, this `get` method is blocking and interruptible.
- The status of a task (running or completed) can be queried using an `isFinished` method.
- *Callbacks* can be registered with a computation and will be executed after the task finishes.
- *Continuations* can be created as well, using functions. A continuation is a computation that applies a function to the output of a previous computation.
- Users can choose to run continuations in new threads (for parallelism) or in existing threads (for efficiency).
- If a task fails, the cause of failure can be retrieved via the `get` function. The associated callbacks are still run, but not any of the continuations.

Note that real *futures*, like those of Java, Google and Scala, tend to be supported by *execution services* (thread pools). This assignment has computations create threads on demand instead, for simplicity.

2 Starting Computations

Computations are created from tasks using a static method `Computation.newComputation`.[1] Tasks are specified using the interface `Callable` (a generalization of `Runnable` through which tasks can return values and throw checked exceptions).

Once a computation is returned, its status can be checked using `isFinished` and the result of the computation can be retrieved using `get`. Method `get` blocks the calling thread until the computation finishes (or the thread is interrupted). Once `isFinished` returns `true`, method `get` is guaranteed not to block.

Method `get` can throw 3 types of exceptions:

- `java.lang.InterruptedException` if the calling thread is interrupted while waiting.
- `java.util.concurrent.ExecutionException` if the task fails. This is a wrapper for the exception that caused the task to fail.
- `java.util.concurrent.CancellationException` if the computation was never run because it is a continuation of a failed or cancelled computation (see below for a discussion of continuations).

```
1 @Test void test1() throws Exception {
2   Thread main = Thread.currentThread();
3   TestTask<String> task = new TestTask<>("T"); // the underlying task
4   Computation<String> comp = Computation.newComputation(task);
5   assertTrue(task.waitForStart(1000)); // the task starts to run
6   assertFalse(comp.isFinished()); // the computation is not finished
7   task.finish();
8   assertEquals(comp.get(), "T"); // get blocks the thread, then produces the result "T"
9   assertTrue(comp.isFinished()); // once get returns, the computation is finished
10  assertNotSame(task.getCaller(), main); // the task ran in a separate thread
11  task.getCaller().join(1000); // this thread now terminates
12  assertFalse(task.getCaller().isAlive());
13 }
```

Listing 1: Starting and running computations.

Lis. 1 illustrates the basic behavior of a computation. A task[2] is created as a `Callable` (line 3) and a computation is started from it (line 4). The computation starts to run the task immediately. Method `isFinished` returns `false` while the task is running. After the task finishes, method `get` returns the value produced by the task (line 8). The test then checks that the task was indeed run by a different thread (line 10) and that the thread properly terminates (lines 11 and 12).

3 Callbacks

A computation can be associated with callbacks, which are specified as instances of type `Runnable`. The callbacks run after the task finishes. A computation terminates after all the callbacks have run. In other words, while the callbacks are running, `get` is blocking and `isFinished` returns `false`. The order in which the callbacks run is not specified. In particular, they may not run in the order in which they were submitted.

Callbacks can be given initially when the computation is started or added while the computation is running via the `onComplete` method. Callbacks are run by specific threads as follows:

- Callbacks that were specified at construction time (i.e., via method `newComputation`) are run by the computation thread, that is, the same thread that ran the computation.
- Callbacks that are added while the task is running and method `get` is blocking also run in the computation thread.
- Callbacks that are added after the computation thread is terminated are run by the calling thread, that is, the thread that calls `onComplete`.
- Callbacks that are added at other times (after `get` returns and before the computation thread terminates) are run by either thread (computation thread or calling thread).

Note that a first callback added at the same time the task finishes is a race condition: The callback may be run by the computation thread or by the calling thread. However, it is still guaranteed that the callback only runs once. The same is true of a callback added at the same time the last of the callbacks already in place finishes running.

Lis. 2 illustrates the different scenarios. Callback 1 is given at construction time and is thus guaranteed to run in the computation thread (line 20). Callback 2 is given while the task is running and will also run in the computation thread (line 21). Callback 3 is added while the task finishes (line 13). Depending on timing, it can run in the computation thread or in the calling thread (called `main` in the test). Finally, callback 4 is added after the computation thread has finished (line 27) and is thus guaranteed to run in the calling thread (line 28).

[1] We use a static method instead of a public constructor because a thread is created *and started* in the method, something that would be unsafe in a constructor.

[2] The tests discussed here use a specific implementation of tasks that can produce results or throw exception, can run for a specific amount of time or until method `finish` is called, keep a record of what thread ran them, etc.

```java
@Test void test2() throws Exception {
  Thread main = Thread.currentThread();
  Callback callback1 = new Callback();
  Callback callback2 = new Callback();
  Callback callback3 = new Callback();
  Callback callback4 = new Callback();
  TestTask<String> task = new TestTask<>("T");
  // first callback specified at construction time
  Computation<String> comp = Computation.newComputation(task, callback1);
  comp.onComplete(callback2); // callback added while the task is running
  assertFalse(comp.isFinished());
  task.finish();
  comp.onComplete(callback3); // racy callback
  comp.get();
  // get has returned; all the callbacks have run exactly once
  assertEquals(callback1.getCallCount(), 1);
  assertEquals(callback2.getCallCount(), 1);
  assertEquals(callback3.getCallCount(), 1);
  // callbacks 1 and 2 run by the computation thread
  assertSame(callback1.getCaller(), task.getCaller());
  assertSame(callback2.getCaller(), task.getCaller());
  // callback3 run by either thread
  assertTrue(callback3.getCaller() == task.getCaller() || callback3.getCaller() == main);
  task.getCaller().join(1000);
  assertFalse(task.getCaller().isAlive());
  // computation thread is terminated
  comp.onComplete(callback4); // run in calling thread
  assertSame(callback4.getCaller(), main);
}
```

Listing 2: Callbacks.

4 Continuations

Continuations also run after a computation has finished. In that way, they are similar to callbacks, but with several important differences:

- Method `get` returns the value of a computation and method `isFinished` returns `true` *before* continuations are started. It follows that existing callbacks are run before continuations begin (more callbacks can be added after continuations have started, although this is not a common pattern).
- Continuations are specified as *functions*, instead of using `Runnable`, so they can take the output of the computation as their input.
- Continuations can be made to run in an existing thread (like callbacks) by using method `map`, or a new thread can be created by using method `mapParallel`.
- Continuations are themselves computations, to which callbacks and further continuations can be added.
- Failures are handled differently in callbacks and in continuations (see below).

Continuations are run by specific threads as follows:

- Continuations created by method `mapParallel` *always* run in a new thread.
- Continuations that are created while the task is running and method `get` is blocking run in the computation thread.
- Furthermore, continuations created by method `map` before the previously registered continuations have all completed also run in the computation thread.

- Continuations created by method `map` after the computation thread has terminated are run in the calling thread, that is, the thread that calls `map`.

- Continuations created by method `map` after the computation and all the non-parallel continuations have finished (i.e., when the computation thread is done with all its work) may run in the computation thread or in the calling thread (race condition).

```java
1 @Test void test3() throws Exception {
2   Thread main = Thread.currentThread();
3   TestTask<String> task = new TestTask<>("T");
4   TestFunction<String, Integer> f1 = new TestFunction<>(1);
5   TestFunction<String, Integer> f2 = new TestFunction<>(2);
6   TestFunction<String, Integer> f3 = new TestFunction<>(3, 1.0); // f3 terminates automatically after 1 second
7   TestFunction<String, Integer> f4 = new TestFunction<>(4, 1.0); // f4 terminates automatically after 1 second
8   Computation<String> comp = Computation.newComputation(task);
9   // first continuation specified while the task is running
10  Computation<Integer> comp1 = comp.map(f1);
11  assertFalse(f1.isRunning()); // continuation not started yet
12  task.finish();
13  assertTrue(f1.waitForStart(1000)); // first continuation starts...
14  assertSame(f1.getCaller(), task.getCaller()); // in the computation thread...
15  assertEquals(f1.getInput(), "T"); // with the output of the computation as its input
16  // second continuation added before the first one finished
17  assertFalse(comp1.isFinished());
18  Computation<Integer> comp2 = comp.map(f2);
19  assertFalse(f2.isRunning()); // continuation not started yet
20  f1.finish(); // first continuation finishes...
21  assertEquals(comp1.get().intValue(), 1); // with its output: 1
22  assertTrue(f2.waitForStart(1000)); // second continuation starts...
23  assertSame(f2.getCaller(), task.getCaller()); // in the computation thread...
24  assertEquals(f2.getInput(), "T"); // with the output of the computation as its input
25  f2.finish(); // second continuation finishes...
26  // at the same time a third computation is specified
27  Computation<Integer> comp3 = comp.map(f3);
28  assertTrue(f3.waitForStart(1000)); // third continuation starts...
29  assertTrue(f3.getCaller() == task.getCaller() || f3.getCaller() == main); // in an existing thread...
30  assertEquals(f3.getInput(), "T"); // with the output of the computation as its input
31  f3.finish();
32  assertEquals(comp2.get().intValue(), 2); // output of second computation
33  assertEquals(comp3.get().intValue(), 3); // output of third computation
34  task.getCaller().join(1000);
35  assertFalse(task.getCaller().isAlive());
36  // computation thread is terminated
37  Computation<Integer> comp4 = comp.map(f4);
38  assertTrue(comp4.isFinished()); // continuation is terminated
39  assertSame(f4.getCaller(), main); // it ran in the calling thread...
40  assertEquals(f4.getInput(), "T"); // with the output of the computation as its input
41  assertEquals(comp4.get().intValue(), 4); // output of fourth computation
42 }
```

Listing 3: Continuations.

In Lis. 3, a continuation `f1` is added to a running task (line 10). It starts to run after the task finishes, in the same thread that ran the task (line 14). Its input is the output of the task: `"T"` (line 15). A second continuation `f2` is added while the first continuation is running (line 18). It starts when the first continuation finishes, in the same thread (line 23). Its input is also the output of the original task. A third continuation `f3` is added at the same time `f2` finishes (line 27). It is run by either the computation thread (like `f1` and `f2`) or by the calling thread, depending on the outcome of a race. After `f3` finishes, the computation thread terminates. A fourth continuation `f4` is then added (line 37) and runs in the calling thread (line 39). Each continuation $\mathtt{f}_i$ has a corresponding computation `comp`$_i$, on which method `get` can be used to retrieve the output of the continuation (lines 21, 32, 33 and 41). Note that the fourth computation `comp4` is run in the calling thread and is returned already finished (line 38).

In Lis. 3, all continuations are specified using method `map`. By contrast, Lis. 4 uses method `mapParallel`. As a consequence, multiple continuations can run in parallel in separate threads (line 28) and a new continuation added after all existing threads are finished will trigger the creation of a new thread (line 43) instead of running in the calling thread like before (when `map` was used instead of `mapParallel`).

Note that Lis. 3 and Lis. 4 keep adding continuations to the same computation `comp`, and no callbacks. More complicated scenarios would add continuations and callbacks to the computations that were created from continuations, like the `comp`$_i$ in these examples. The semantics of what thread runs what and when would still need to apply.

5 Failures

Dealing with failures is an important but difficult part of concurrent and distributed programming. One issue is that failures must be carried from one thread (or distributed process) to another. Languages and libraries usually offer mechanisms to help with this (e.g., serialization of exceptions or the wrapper `ExecutionException` in Java). In this assignment, the goal is to implement the following semantics:

- *Exceptions* (checked and unchecked) are handled; *errors* are not. If an error occurs in any task, callback or continuation, the behavior of the system is undefined.
- If a task used to create a computation throws an exception:
  - the computation's callbacks are run;
  - all the computation's continuations (created from `map` and `mapParallel`) are cancelled;
  - method `isFinished` returns `true`;
  - method `get` throws an instance of `java.util.concurrent.ExecutionException`; the `getCause` method of this instance returns the exception thrown in the task.
- When a computation is cancelled (because it is a continuation of a computation that failed or was cancelled):
  - its own callbacks are not run;
  - its own continuations are cancelled;
  - its `isFinished` method returns `true`;
  - its `get` method throws an instance of `java.util.concurrent.CancellationException`.
- When a callback fails with an exception, the exception is ignored (or simply logged) and other callbacks are run.
- When a continuation fails with an exception, its computation behaves like the computation of a failed task (above); other continuations of the same task ("siblings") may or may not run. If they don't run, they are properly cancelled.

Lis. 5 illustrates the behavior of computations under failures. A failing computation is created with two continuations and two callbacks. After the task fails, the `get` method of its computation throws an instance of `ExecutionException` (line 24) with the task's exception as its cause (line 25). The continuation computations are then cancelled. Their `get` methods throw instances of `CancellationException` (lines 33 and 41). All the computations are now terminated. The first callback fails with a `RuntimeException` but the second callback was properly executed (line 43). The continuations never ran (lines 49 and 50).

6 Possible Implementations

Using the building blocks discussed in class, this assignment can be implemented in one of two ways:

- Using `java.util.concurrent.FutureTask`. This is the basis for the implementation of futures in the Java concurrency library, and this assignment's "computations" are a form of future. `FutureTask` provides several of the mechanisms needed here: a blocking `get` method that wraps failures inside instances of `ExecutionException`; a cancellation mechanism that uses `CancellationException` and a notion of completion (method `isDone`).

```java
@Test void test4() throws Exception {
  Thread main = Thread.currentThread();
  Set<Thread> threads = new java.util.HashSet<>(4);
  TestTask<String> task = new TestTask<>("T");
  TestFunction<String, Integer> f1 = new TestFunction<>(1);
  TestFunction<String, Integer> f2 = new TestFunction<>(2);
  TestFunction<String, Integer> f3 = new TestFunction<>(3);
  Computation<String> comp = Computation.newComputation(task);
  // first continuation specified while the task is running
  Computation<Integer> comp1 = comp.mapParallel(f1);
  assertFalse(f1.isRunning()); // continuation not started yet
  task.finish();
  assertTrue(f1.waitForStart(1000)); // first continuation starts...
  assertNotSame(f1.getCaller(), task.getCaller()); // in a new thread...
  assertEquals(f1.getInput(), "T"); // with the output of the computation as its input
  // threads used so far
  threads.add(task.getCaller());
  threads.add(f1.getCaller());
  // computation thread is done
  task.getCaller().join(1000);
  assertFalse(task.getCaller().isAlive());
  // second continuation added before the first one finished
  assertFalse(comp1.isFinished());
  Computation<Integer> comp2 = comp.mapParallel(f2);
  assertTrue(f2.waitForStart(1000)); // it starts running immediately...
  assertTrue(threads.add(f2.getCaller())); // in a new thread...
  assertEquals(f2.getInput(), "T"); // with the output of the computation as its input
  assertTrue(f1.isRunning() && f2.isRunning()); // both continuations run in parallel
  // the first two continuations finish...
  f1.finish();
  f2.finish();
  // with their outputs: 1 and 2
  assertEquals(comp1.get().intValue(), 1);
  assertEquals(comp2.get().intValue(), 2);
  // all threads now done
  f1.getCaller().join(1000);
  assertFalse(f1.getCaller().isAlive());
  f2.getCaller().join(1000);
  assertFalse(f2.getCaller().isAlive());
  // all threads are finished before a third computation is specified
  Computation<Integer> comp3 = comp.mapParallel(f3);
  assertTrue(f3.waitForStart(1000)); // third continuation starts...
  assertTrue(threads.add(f3.getCaller())); // in a new thread...
  assertEquals(f3.getInput(), "T"); // with the output of the computation as its input
  f3.finish();
  assertEquals(comp1.get().intValue(), 1); // output of first continuation
  assertEquals(comp2.get().intValue(), 2); // output of second continuation
  assertEquals(comp3.get().intValue(), 3); // output of third continuation
  // last thread terminates
  f3.getCaller().join(1000);
  assertFalse(task.getCaller().isAlive());
  assertEquals(threads.size(), 4); // four threads were used...
  assertFalse(threads.contains(main)); // none of them the calling thread
}
```

Listing 4: Parallel continuations.

```java
@Test void test5() throws Exception {
  Exception ex = new Exception(); // exception thrown by the task
  TestTask<String> task = new TestTask<>(ex); // a failing task
  Computation<String> comp = Computation.newComputation(task);
  TestFunction<String, Integer> f1 = new TestFunction<>(1); // first continuation
  TestFunction<String, Integer> f2 = new TestFunction<>(2); // second continuation
  Runnable callback1 = () -> { // a failing callback
    throw new RuntimeException();
  };
  Callback callback2 = new Callback(); // a second callback
  // registering callbacks
  comp.onComplete(callback1);
  comp.onComplete(callback2);
  // registering continuations
  Computation<Integer> comp1 = comp.map(f1);
  Computation<Integer> comp2 = comp.mapParallel(f2);
  task.finish(); // the task fails
  try {
    comp.get(); // should throw ExecutionException
    fail("exception expected");
  } catch (InterruptedException e) {
    throw e;
  } catch (Exception e) {
    assertTrue(e instanceof ExecutionException);
    assertSame(e.getCause(), ex); // the exception thrown by the task
  }
  try {
    comp1.get(); // should throw CancellationException
    fail("exception expected");
  } catch (InterruptedException e) {
    throw e;
  } catch (Exception e) {
    assertTrue(e instanceof CancellationException);
  }
  try {
    comp2.get(); // should throw CancellationException
    fail("exception expected");
  } catch (InterruptedException e) {
    throw e;
  } catch (Exception e) {
    assertTrue(e instanceof CancellationException);
  }
  assertEquals(callback2.getCallCount(), 1); // second callback was executed
  // all computations are finished
  assertTrue(comp.isFinished());
  assertTrue(comp1.isFinished());
  assertTrue(comp2.isFinished());
  // the continuations were never called
  assertNull(f1.getCaller());
  assertNull(f2.getCaller());
  task.getCaller().join(1000);
  assertFalse(task.getCaller().isAlive());
}
```

Listing 5: Failures and cancellations.

- Using `java.util.concurrent.CountDownLatch`. A latch can be used to implement a blocking `get` method (the latch opens when a task finishes). In that case, the wrapping of failures needs to be implemented by hand using `try/catch` blocks and a cancellation mechanism must also be added.

There is a little less work to do when following the first approach, but it involves using `FutureTask`, a construct that is more abstract and more complex than a simple latch. The second approach is more "hands-on" and could be easier to achieve even though it uses a little more code.

7 Work to be submitted

You need to submit the source code of class `Computation` as well as a report. The report *must* address the following questions:

- Discuss the overall design of the `Computation` class. Is it based on `FutureTask` or on `CountDownLatch`? What data structures are used to keep track of callbacks and continuations?
- Discuss the strategies used to ensure correct behavior in the presence of races. For instance, a callback is added right when a task finishes: How do you ensure that the callback is run and is run only once? Discuss all the other race scenarios and explain how your implementation handles them.
- Discuss the strategy used to handle failures, and especially the cancellation of continuations. Focus on tricky scenarios and explain how your implementation handles them. For instance, a task with two continuations finishes and one of its continuations fails. The continuations of this continuation must be cancelled. Furthermore, the other continuation of the original task may run or be cancelled, but its `get` method should not remain blocking forever.
- What tests were used to show that the `Computation` implementation adheres to the semantics specified in this document? Make sure you describe in detail the testing scenarios that were used (e.g., like the descriptions of the listing figures in this document). Tests need to go beyond the scenarios used in the sample tests: race conditions, continuations of continuations, callbacks on continuations, failures of continuations, mixture of `map` and `mapParallel` continuations, hundreds of threads, thousand of computations, etc.