CS–735/835

Programming Assignment #2

due 27 Feb 2022 11:59 PM, tagged `2` or `2.#`

1 Introduction

This assignment implements a form of "background computation" that can be started to introduce parallelism in an application. These computations offer simplified versions of functionalities found for instance in Java's `Future` and `CompletionStage` interfaces, or Scala's `Future` trait. In this assignment:

- 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 computation (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 computation fails, the cause of failure can be retrieved via the `get` function. The associated callbacks are still run, but not the continuations.
- Computations are *thread-safe*: any thread can call `get`, `isFinished`, `onComplete`, `map` or `mapParallel` on a computation.

Newly created computations typically run in their own thread, referred to below as the "computation thread". Real "futures", like those of Java and Scala, tend to be supported by *execution services* (thread pools). This assignment has computations create threads on demand instead, for simplicity.

2 Basic Usage

Computations are created from tasks using one of two static methods `newComputation`. 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 created, its task starts to run in a new thread. The task runs without any lock held in order to guarantee that all the methods of a computation remain available. The status of a computation can be checked using `isFinished`, and the result of the computation can be retrieved using `get`. The `get` method can return a value or throw one of 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 discussion of continuations below).

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.

Lis. 1 illustrates the basic behavior of a computation.[1] A task 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 (line 7). After the task finishes, method `get` returns the value produced by the task (line 9). The test then checks that the task was indeed run by a different thread (line 11) and that the thread properly terminates (line 12).

3 Callbacks

A computation can be associated with callbacks, which are specified as instances of type `Runnable`. Callbacks run after the task finishes. A computation terminates after the registered callbacks have run, at which point `get` becomes unblocking and `isFinished` returns `true`. The order in which the callbacks run is not specified. In particular, they may not run in the order in which they were submitted, and a callback may even run concurrently with another callback. Callbacks can still be added after a computation has finished. They then run immediately, and the computation remains finished, even while these callbacks are running (a computation never transitions from finished to unfinished). Callbacks can be given initially when the computation is created, or added later via the `onComplete` method. Callbacks are run without any lock held in order to guarantee that all the methods of a computation remain available.

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 task.
- Callbacks that are added while the task is running 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 the task finishes but 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 (line 7), and is thus guaranteed to run in the computation thread (line 18). Callback 2 is given while the task is running (line 8), and will also run in the computation thread (line 19). Callback 3 is added while the task finishes (line 11). Depending on timing, it can run in the computation thread or in the calling thread (called `main` in the test) (line 21). Finally, callback 4 is added after the computation thread has finished (line 23) and is thus guaranteed to run in the calling thread (line 25).

It is important to remember that:

- Callbacks are considered part of the computation. A computation does not complete until all its (existing) callbacks are finished.
- Callbacks are run without any lock held so that all the computation methods remain available.

Lis. 3 illustrates these two points. A task is created with a callback (line 5). After the task finishes, the callback starts to run (line 8). While the callback is running, the computation is unfinished (line 9), and the `get` method of the computation remains blocking (line 12).

[1] Illustrative scenarios are given as sample tests. They are written in Scala and rely on a specific implementation of tasks, callbacks and continuations that can produce results or throw exceptions, terminate immediately or run until method `finish` is called, keep a record of what thread ran them, etc. Do not assume these specific tasks, callbacks and continuations types in your implementation, e.g., not all tasks have type `TestTask`, not all continuations are `TestFunction`, etc.

4 Continuations

Continuations run after a computation has finished. 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 could be added after continuations have started, but this is not a common pattern).
- Continuations are specified as *functions*, instead of using `Runnable`. They take the output of the computation as their input, and their output is the output of a new computation.
- Continuations can be made to run in an existing thread (like callbacks) by using method `map`, or in a new thread 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 new threads.
- Continuations created by method `map` while the computation is running 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) but before the computation thread terminates may run in the computation thread or in the calling thread (race condition).

In Lis. 4, 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 (lines 13–14). Its input is the output of the task: `"T"` (line 15). A second continuation `f2` is added while the first continuation is running (line 17). It starts when the first continuation finishes, in the same thread (line 20–21). Its input is also the output of the original task. A third continuation `f3` is added at the same time `f2` finishes (line 24). It is run by either the computation thread (like `f1` and `f2`) or by the calling thread (line 26), depending on the outcome of a race. A fourth continuation `f4` is added after the computation thread is terminated (line 32), and runs in the calling thread (line 34). Each continuation $\texttt{f}_i$ has a corresponding computation $\texttt{comp}_i$, on which method `get` can be used to retrieve the output of the continuation (lines 28, 29, 30 and 36). Note that the fourth computation `comp4` is run in the calling thread and is returned already finished (line 33).

In Lis. 4, all continuations are specified using method `map`. By contrast, Lis. 5 uses method `mapParallel`. As a consequence, multiple continuations can run in parallel in separate threads (line 23), and a new continuation added after all existing threads are finished will trigger the creation of a new thread (line 32) instead of running in the calling thread like before (when `map` was used instead of `mapParallel`).

Note that Lis. 4 and Lis. 5 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 $\texttt{comp}_i$ 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, the `Try` type in Scala, etc.). In this assignment, the goal is to implement the following semantics:

- *Exceptions* (of type `Exception`, checked or unchecked) are handled; *errors* (or type `Error`) 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`/`mapParallel`) are cancelled and never run;
  - 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 continuation is 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 simply logged, and other callbacks and continuations are run.
- When a continuation fails with an exception, its computation behaves like the computation of a failed task (above): its callbacks are run, but not its continuations. Other continuations of the same task ("siblings") should still run.

Lis. 6 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` with the task's exception as its cause (line 18). The continuation computations are then cancelled, and their `get` methods throw instances of `CancellationException` (lines 20–21). The first callback fails with a `RuntimeException`, but the second callback is properly executed (line 22). All the computations are now terminated (lines 24–26). The continuations are never run (lines 28–29).

6 Creating Computations

Computations are created from existing computations using methods `map` and `mapParallel`. Computations can also be created from scratch using static methods `newComputation`:

- From a single task:

  ```java
  static <T> Computation<T> newComputation(
    Callable<? extends T> task,
    Runnable... callbacks
  )
  ```

  This method creates a computation and exactly one thread to run it (the computation thread). The thread is started. Optional callbacks can be associated with the computation at this time. They will run in the computation thread.

- From a list of tasks:

  ```
  static <T> Computation<List<T>> newComputation(
    List<? extends Callable<? extends T>> tasks,
    Runnable... callbacks
  )
  ```

  This method creates at least one computation (the return value), and possibly more to run the tasks in the list. The computation that is returned will produce a list of the results of all the tasks after they have completed. The results in the list are in the same order as the tasks they come from. This method creates (and starts) exactly k threads, where k is the number of tasks in the list. All the tasks of the list are run in parallel.

  Note that method `newComputation` on a list of k tasks creates k threads, *not* $k + 1$. In particular:

  – If the list is empty ($k = 0$), no thread is created. A computation is returned already completed. The callbacks, if any, are run in the calling thread. Lis. 7 shows the case of an empty list of tasks.

  – If $k > 0$, one of the k threads that run the tasks will also be the computation thread (in charge of potentially running some callbacks and continuations). No extra thread is created for this purpose. As an illustration, Lis. 8 shows a computation created from a list of two tasks, plus one callback (line 7). Both tasks start and run in parallel, in two different threads (lines 11–12). After one task finishes, the computation is still running the other task (line 14), and the callback has not run (line 16). After the second task finishes, the callback is run (line 20), the computation terminates (lines 18–19), and produces a list of task results (lines 22–23). The callback was run by the computation thread, which also ran one of the tasks (line 25).

7 Holding Locks

As discussed in class, it is generally a bad idea to call foreign code (like callbacks and continuations) while holding locks. When implementing this assignment, you need to make sure that the callables, runnables and functions used to specify tasks, callbacks and continuations are run without holding locks that would prevent some methods from being called.

For instance, the test in lis. 9 checks that methods `isFinished`, `map`, `mapParallel` and `onComplete` can be called while a callback is running (lines 14–17), or while a continuation is running (lines 22–25). None of these calls should block the calling thread, and the test should quickly terminate.

Lis. 10 shows a scenario in which a callback is registering another callback. The registration should be successful, and the second callback should run. Note that it necessarily runs in the computation thread: it is either early enough to run as a regular callback (in the computation thread), or late and made to run in the registering thread (which is also the computation thread).

8 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 class 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`).

- Using `java.util.concurrent.CountDownLatch`. A latch can be used to implement a blocking `get` method (the latch opens after a task and its callbacks finish). 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 (but not that much more).

9 Work To Be Submitted

Students need to complete class `NewThreadComputation` and submit a report. The report must discuss the following scenario:

> *There is a race condition if a callback is added right when a task finishes. How does your implementation handle it? In other words, how does it guarantee that the callback is run, and is run only once? Give an explanation, not code.*

Notes:

- The API of the class to implement can be found in the `2/api` directory of the Git repository.
- Conceptually, this assignment implements a general-purpose library, and relies on complex method signatures. For instance, an argument of the second `newComputation` method has type `List<? extends Callable<? extends T>>`. This argument is basically a list of callables of type `T`, but the method is written in a way that makes it easier to apply it to types that are not exactly `List<Callable<T>>`. This is a consequence of Java's type system—namely, the lack of variance in generics. If these complicated types confuse you, be sure to ask for help.
- As another example, an object of type `Callable<? extends A>` does not have type `Callable<A>` (it's the other way around), but a `Callable<? extends A>` can be used to create a `Callable<A>` if needed:

```java
<A> void test(Callable<? extends A> c) throws Exception {
  A value = c.call();              // call it and get an A
  Callable<? extends A> myVar = c; // store it in variables of type Callable<? extends A>
  Callable<A> ca1 = () -> c.call(); // if needed, create a Callable<A> using a lambda
  Callable<A> ca2 = c::call;        // or create a Callable<A> using a method reference
}
```

- Using lambdas, there is no need for inner classes inside `Computation`. If you use inner classes instead, there is likely no need for them to be generic. If you do end up using inner generic classes, use a variable other than `A` for their type parameter to avoid getting confused between two incompatible types, both named `A`.
- A Java program `FutureTaskDemo` demonstrates the basic methods of type `FutureTask`.
- A Java program `CountDownLatchDemo` demonstrates the basic methods of type `CountDownLatch`.
- The ten tests of the handout are given as sample grading tests. More tests need to be written, including continuations of continuations and more complex failure scenarios.
- Computations are created in factory methods `newComputation` as a way to avoid starting threads inside a constructor. The `NewThreadComputation` class also has one or more constructor(s), but it/they are not public.
- Computations have a somewhat recursive structure: continuations of computations are themselves computations. What is needed to run a computation is also needed to run its continuations. This is made easier by introducing a suitable (private) method (say, `execute`), run by the computation thread. When needed, `execute` can call `execute` on other computations (within the same thread or in a new thread).

Sample Tests

```scala
test("Sample test 1") {
   val main = Thread.currentThread
   val task = SuccessfulTestTask("T") // the underlying task
   val comp = newComputation(task)

   assert(task.waitForStart())     // the task starts to run
   assert(!comp.isFinished)        // the computation is not finished
   task.finish()                   // finish the task
   assert(comp.get() == "T")       // get blocks the thread, then produces the result "T"
   assert(comp.isFinished)         // once get returns, the computation is finished
   assert(task.caller ne main)     // the task ran in a separate thread
   assert(task.caller.joined(1.0)) // this thread now terminates
}
```

Listing 1: Starting and running computations.

```
test("Sample test 2") {
   val main                                       = Thread.currentThread
   val callback1, callback2, callback3, callback4 = Callback()
   val task                                       = SuccessfulTestTask("T")

   // first callback specified at construction time
   val comp = newComputation(task, callback1)
   comp.onComplete(callback2) // callback added while the task is running
   assert(!comp.isFinished)
   task.finish()
   comp.onComplete(callback3) // racy callback as the task finishes
   comp.get()
   // get has returned; all the callbacks have run exactly once
   assert(callback1.callCount == 1)
   assert(callback2.callCount == 1)
   assert(callback3.callCount == 1)
   // callbacks 1 and 2 run by the computation thread
   assert(callback1.caller eq task.caller)
   assert(callback2.caller eq task.caller)
   // callback3 run by either thread
   assert((callback3.caller eq task.caller) || (callback3.caller eq main))
   assert(task.caller.joined(1.0)) // computation thread is terminated
   comp.onComplete(callback4)      // run in calling thread
   assert(callback4.callCount == 1)
   assert(callback4.caller == main)
}
```

Listing 2: Callbacks.

```
test("Sample test 3") {
   val task                 = SuccessfulTestTask("T")
   val callbackTask         = SuccessfulTestTask("C")
   val callback: Runnable = () => callbackTask.call() // a long running callback
   val comp                 = newComputation(task, callback)

   task.finish()
   assert(callbackTask.waitForStart()) // callback starts after task finishes
   assert(!comp.isFinished)            // computation is unfinished while callback is running
   val t = newThread(start = true)(comp.get()) // a thread calls comp.get()
   sleep(0.5)
   assert(t.isAlive)     // the thread is stuck on get()
   callbackTask.finish() // callback finishes
   // both threads terminate
   assert(task.caller.joined(1.0))
   assert(t.joined(1.0))
}
```

Listing 3: Running callbacks.

```scala
test("Sample test 4") {
   val main = Thread.currentThread
   val task = SuccessfulTestTask("T")
   val f1   = SuccessfulTestFunction[String, Int](1)
   val f2   = SuccessfulTestFunction[String, Int](2)
   val f3   = SuccessfulTestFunction[String, Int](3, autoFinish = true)
   val f4   = SuccessfulTestFunction[String, Int](4, autoFinish = true)
   val comp = newComputation(task)

   val comp1 = comp.map(f1) // first continuation specified while the task is running
   assert(!f1.isStarted) // continuation not started yet
   task.finish()
   assert(f1.waitForStart())        // first continuation starts...
   assert(f1.caller eq task.caller) // in the computation thread...
   assert(f1.input == "T")          // with the output of the computation as its input
   assert(!comp1.isFinished)
   val comp2 = comp.map(f2) // second continuation added before the first one finished
   assert(!f2.isStarted)            // continuation not started yet
   f1.finish()                      // first continuation finishes
   assert(f2.waitForStart())        // second continuation starts...
   assert(f2.caller eq task.caller) // in the computation thread...
   assert(f2.input == "T")          // with the output of the computation as its input
   f2.finish()                      // second continuation finishes...
   val comp3 = comp.map(f3) // at the same time a third computation is specified
   assert(f3.waitForStart())                                 // third continuation starts...
   assert((f3.caller eq task.caller) || (f3.caller eq main)) // in an existing thread...
   assert(f3.input == "T")         // with the output of the computation as its input
   assert(comp1.get() == 1)        // output of first continuation
   assert(comp2.get() == 2)        // output of second continuation
   assert(comp3.get() == 3)        // output of third continuation
   assert(task.caller.joined(1.0)) // computation thread terminates
   val comp4 = comp.map(f4)
   assert(comp4.isFinished)  // continuation is already terminated
   assert(f4.caller eq main) // it ran in the calling thread...
   assert(f4.input == "T")   // with the output of the computation as its input
   assert(comp4.get() == 4)  // output of fourth continuation
}
```

Listing 4: Continuations.

```
test("Sample test 5") {
   val main    = Thread.currentThread
   val threads = mutable.Set.empty[Thread]
   val task    = SuccessfulTestTask("T")
   val f1      = SuccessfulTestFunction[String, Int](1)
   val f2      = SuccessfulTestFunction[String, Int](2)
   val f3      = SuccessfulTestFunction[String, Int](3)
   val comp    = newComputation(task)

   val comp1 = comp.mapParallel(f1) // first continuation specified while the task is running
   assert(!f1.isStarted) // continuation not started yet
   task.finish()
   assert(f1.waitForStart()) // first continuation starts...
   threads += task.caller
   assert(threads.add(f1.caller))  // in a new thread...
   assert(f1.input == "T")         // with the output of the computation as its input
   assert(task.caller.joined(1.0)) // computation thread terminates
   assert(!comp1.isFinished)
   val comp2 = comp.mapParallel(f2) // second continuation added before the first one finished
   assert(f2.waitForStart())            // it starts running immediately...
   assert(threads.add(f2.caller))       // in a new thread...
   assert(f2.input == "T")              // with the output of the computation as its input
   assert(f1.isStarted && f2.isStarted) // both continuations run in parallel
   f1.finish()
   f2.finish()
   // the first two continuations finish...
   // all threads now done...
   assert(f1.caller.joined(1.0))
   assert(f2.caller.joined(1.0))
   val comp3 = comp.mapParallel(f3) // before a third computation is specified
   assert(f3.waitForStart())      // third continuation starts...
   assert(threads.add(f3.caller)) // in a new thread...
   assert(f3.input == "T")        // with the output of the computation as its input
   f3.finish()
   assert(comp1.get() == 1)        // output of first continuation
   assert(comp2.get() == 2)        // output of second continuation
   assert(comp3.get() == 3)        // output of third continuation
   assert(f3.caller.joined(1.0))   // last thread terminates
   assert(threads.size == 4)       // four threads were used...
   assert(!threads.contains(main)) // none of them the calling thread
}
```

Listing 5: Parallel continuations.

```scala
test("Sample test 6") {
   val ex   = Exception()                            // exception thrown by the task
   val task = FailedTestTask(ex)                     // a failing task
   val comp = newComputation(task)
   val f1   = SuccessfulTestFunction[String, Int](1) // first continuation
   val f2   = SuccessfulTestFunction[String, Int](2) // second continuation
   val callback1: Runnable = () => throw RuntimeException() // a failing callback
   val callback2           = Callback()                     // a second callback

   // registering callbacks
   comp.onComplete(callback1)
   comp.onComplete(callback2)
   // registering continuations
   val comp1 = comp.map(f1)
   val comp2 = comp.map(f2)
   task.finish() // the task fails
   // comp throws ExecutionException with task exception as its cause
   assert(intercept[ExecutionException](comp.get()).getCause eq ex)
   // comp1 and comp2 throw CancellationException
   assertThrows[CancellationException](comp1.get())
   assertThrows[CancellationException](comp2.get())
   assert(callback2.callCount == 1) // second callback was executed
   // all computations are finished
   assert(comp.isFinished)
   assert(comp1.isFinished)
   assert(comp2.isFinished)
   // the continuations were never called
   assert(!f1.isStarted)
   assert(!f2.isStarted)
   assert(task.caller.joined(1.0))
}
```

Listing 6: Failures and cancellations.

```scala
test("Sample test 7") {
   val main     = Thread.currentThread
   val callback = Callback()

   // computation created from an empty list of tasks
   val comp = newComputation(Collections.emptyList(), callback)
   assert(comp.isFinished)         // computation is returned finished
   assert(callback.caller == main) // callback is run in current thread
   assert(comp.get().isEmpty)      // computation result is an empty list
}
```

Listing 7: Creating a computation from an empty list of tasks.

```scala
test("Sample test 8") {
   val task1    = SuccessfulTestTask("T1")
   val task2    = SuccessfulTestTask("T2")
   val callback = Callback()

   // computation created from a list of 2 tasks, with a callback
   val comp = newComputation(util.List.of(task1, task2), callback)
   // both tasks start and run in parallel
   assert(task1.waitForStart())
   assert(task2.waitForStart())
   assert(task1.isStarted && task2.isStarted)
   assert(task1.caller ne task2.caller)
   task2.finish()                   // one task finishes
   assert(!task1.isFinished)        // the other task is still running
   assert(!comp.isFinished)         // and therefore the computation is not finished
   assert(callback.callCount == 0) // and the callback is not run
   task1.finish()                   // the other task finishes
   val results = comp.get() // the computation finishes and get() unblocks
   assert(comp.isFinished)
   assert(callback.callCount == 1) // the callback has been run
   // the computation produced a list of results: [T1,T2]
   assert(results.get(0) == "T1")
   assert(results.get(1) == "T2")
   // the callback was run in the computation thread, which is one of the task running threads
   assert((callback.caller eq task1.caller) || (callback.caller eq task2.caller))
   // both created threads properly terminate
   assert(task1.caller.joined(1.0))
   assert(task2.caller.joined(1.0))
}
```

Listing 8: Creating a computation from a list of tasks.

```scala
trait SampleTest9:
  self: AnyFunSuite =>

  test("Sample test 9") {
    val task              = SuccessfulTestTask("T")
    val callbackTask      = SuccessfulTestTask("C")
    val callback: Runnable = () => callbackTask.call()
    val f                 = SuccessfulTestFunction[String, Int](0)
    val comp              = newComputation(task, callback)

    task.finish()
    assert(callbackTask.waitForStart())
    // callback is running, nothing is blocking
    comp.isFinished
    comp.map(f)
    comp.mapParallel(SuccessfulTestFunction[String, Int](1, autoFinish = true))
    comp.onComplete(Callback())

    callbackTask.finish()
    assert(f.waitForStart())
    // continuation is running, nothing is blocking
    comp.isFinished
    comp.map(SuccessfulTestFunction[String, Int](2, autoFinish = true))
    comp.mapParallel(SuccessfulTestFunction[String, Int](3, autoFinish = true))
    comp.onComplete(Callback())

    f.finish()
    TestSucceeded
  }
```

Listing 9: Running callbacks and continuations without locks.

```scala
test("Sample test 10") { // callback from callback
  val task               = SuccessfulTestTask(42)
  val comp               = newComputation(task)
  val callback2          = Callback()
  val callback1: Runnable = () => comp.onComplete(callback2) // callback1 registers callback2

  comp.onComplete(callback1) // main thread registers callback1
  assert(task.waitForStart())
  task.finish()
  assert(task.caller.joined(1.0))
  assert(callback2.callCount == 1)        // both callbacks ran
  assert(callback2.caller eq task.caller) // callback2 ran in computation thread
}
```

Listing 10: Registering callbacks from callbacks.