11. Asynchronous

Learning Objectives

After this lecture, students should:

- familiar with the concept of asynchronous method calls and be able to use it effectively
- familiar with the concept of promise through Java 8 `CompletableFuture` class

Synchronous vs. Asynchronous

In synchronous programming, when we call a method, we expect the method to be executed, and when the method returns, the result of the method is available.

```
int multiple(int x, int y) {
    return x * y;
}

int z = multiple(3, 4);
```

In the simple example above, our code continues executing after, and only after `add()` completes.

If a method takes a long time to run, however, the execution will delay the execution of subsequent methods, and maybe undesirable.

Asynchronous call to a method allows execution to continue immediately after calling the method, so that we can continue executing the rest of our code, while the long-running method is off doing its job.

You have seen examples of asynchronous calls:

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
```

The call above returns immediately even before the matrix multiplication is complete. We can later return to this task, and call `task.join()` to get the result (waiting for it if necessary).

A `RecursiveTask` also has a `isDone()` method that it implements as part of the `Future` interface. Now, we can do something like this:

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
    while (!task.isDone()) {
        System.out.print(".");
        Thread.sleep(1000);
    }
    System.out.print("done");
```

So, while the task is running, we can print out a series of "."s to feedback to the users to indicate that it is running.

`Thread.sleep(1000)` cause the current running thread to sleep for 1s. It might throw an `InterruptedException`, if the user interrupts the program (by Control-C). To complete the snippet, we should catch the exception and cancel the task.

```
    task = new MatrixMultiplyerTask(m1, m2);
    task.fork();
    try {
        while (!task.isDone()) {
            System.out.print(".");
            Thread.sleep(1000);
        }
        System.out.println("done");
    } catch (InterruptedException e) {
        task.cancel();
        System.out.println("cancelled");
    }
```

Future

Let's look at the `Future` interface a bit more. `Future<T>` represents the result (of type `T`) of an asynchronous task that may not be available yet. It has five simple operations:

- `get()` returns the result of the computation (waiting for it if needed).
- `get(timeout, unit)` returns the result of the computation (waiting for up to the timeout period if needed).
- `cancel(interrupt)` tries to cancel the task -- if `interrupt` is true, cancel even if the task has started. Otherwise, cancel only if the task is still waiting to get started.
- `isCancelled()` returns `true` of the task has been cancelled.
- `isDone()` returns `true` if the task has been completed.

Both `RecursiveTask` and `RecursiveAction` implements the `Future` interface, so you can use the above methods on your tasks.

> **In Other Languages**
> Scala's `Future` is more powerful -- it allows us to specify what to do when the task completes, and it hands abnormal completions (e.g., exceptions). Python 3.2 supports `Future` through `concurrent.futures` [https://docs.python.org/3/library/concurrent.futures.html] module. C++11 supports `std::future`](http://en.cppreference.com/w/cpp/thread/future [http://en.cppreference.com/w/cpp/thread/future]) as well.

CompletableFuture

The example code above tries every second to see if task is done. For some applications, the response time is critical, and we would like to know as soon as a task is done. For instance, response time is important in stock trading applications and web services.

One way to do so, is to sleep for a shorter duration. Or even not sleeping all together:

```
    task.fork();
```

```
    while (!task.isDone()) {
        System.out.print(".");
    }
    System.out.print("done");
```

This is problematic in many ways, besides printing out too many dots:

- this is known as *busy waiting* -- and it occupies the CPU while doing nothing. Such code should be avoided at all cost.
- we may want to continue doing other things besides printing out "."s, so the code won't be a simple for loop anymore. We can do something like this instead:

```
    task.fork();
    if (!task.isDone()) {
        // do something
    } else {
        task.join();
    }
    if (!task.isDone()) {
        // do something else
    } else {
        task.join();
    }
    if (!task.isDone()) {
        // do yet something else
    } else {
        task.join();
    }
```

You can see that the code gets out of hand quickly, and this is only if we have one asynchronous call!

What we need is have a way to specify a *callback*. A callback is basically a method that will be executed when a certain event happens. In this case, we need to specify a callback when an asynchronous task is complete. This way, we can just call an asynchronous task, specify what to do when the task is completed, and forget about it. We do not need to check again and again if the task is done.

To do exactly this, Java 8 introduces the class `CompletableFuture<V>`, which implements the `Future<V>` interface. Thus, just like `Future<V>`, a `CompletableFuture<V>` object returns a value of type `V` when it completes. But `CompletableFuture<V>` is more powerful, it allows us to specify an asynchronous task, and an action to perform when the task completes.

The notion of "complete" is important for `CompletableFuture`. If the `CompletableFuture` is complete, then the value to return is available. We can create an already-completed `CompletableFuture`, passing in a value, or a yet-to-be-completed `CompletableFuture`, by passing in a function to be executed asynchronously. When this function returns, the `CompletableFuture` completes.

To create a `CompletableFuture` object, we can call one of its static method. For instance, `supplyAsync` takes in a `Supplier`:

```
CompletableFuture<Matrix> future = CompletableFuture.supplyAsync(() -> m1.multiply(m2));
```

As explained above, `future` completes when `m1.multiply(m2)` returns.
Let's say that we want to print out the result with a `Consumer` when `future` completes, we can use the `thenAccept` method:

```
future.thenAccept(System.out::println);
```

Or, you can use the oneliner:

```
CompletableFuture
    .supplyAsync(() -> m1.multiply(m2))
    .thenAccept(System.out::println);
```

Waiting for Completion

If you want your code to block until a `CompletableFuture` completes, you can call `join()`.

```
m = future.join();
```

Suppose you have several `CompletableFuture` objects, say `cf1`, `cf2`, and `cf3`, and you want to block until all of these `CompletableFuture` completes. You can create a composite `CompletableFuture` objects, using `allOf()`:

```
CompletableFuture.allOf(cf1, cf2, cf3).join();
```

The object created by `CompletableFuture.allOf(cf1, cf2, cf3)` completes, only after all of `cf1`, `cf2`, `cf3` completes.

There is also a `anyOf`, for cases where it is sufficient for any one of the `CompletableFuture` to complete:

```
CompletableFuture.anyOf(cf1, cf2, cf3).join();
```

CompletableFuture is a Functor / Monad

`CompletableFuture` is a functor. Recall that a functor, in OO-speak, is a class that implements a (hypothetical) interface that looks like the following:

```
interface Functor<T> {
  public <R> Functor<R> f(Function<T,R> func);
}
```

In `CompletableFuture`, the method that makes `CompletableFuture` a functor is the `thenApply` method:

```
<U> CompletableFuture<U> thenApply(Function<? super T,? extends U> func)
```

The method `thenApply` is similar to `thenAccept`, except that instead of a `Consumer`, the callback that gets invoked when the asynchronous task completes is a `Function.

There are other variations:

- `thenRun`, which takes a `Runnable`,
- `thenAcceptBoth`, which takes a `BiConsumer` and another `CompletableFuture`
- `thenCombine`, which takes a `BiFunction` and another `CompletableFuture`
- `thenCompose`, which takes in a `Function` `fn`, which instead of returning a "plain" type, `fn` returns a `CompletableFuture`.

All the methods above return a `CompletableFuture`.

BTW, `CompletableFuture` is a monad too! The `thenCompose` method is analougous to the `flatMap` method of `Stream` and `Optional`.

This also means that `CompletableFuture` satisfies the monad laws, one of which is that there is a method to wrap a value around with a `CompletableFuture`. We call this the `of` method in the context of `Stream` and `Optional`, but in `CompletableFuture`, it is called `completedFuture`. This method creates a `CompletableFuture` that is completed. The `completedFuture` method is useful, for instance, if we want to convert a method below to asynchronous.

```
Integer foo(int x) {
  if (x < 0)
    return 0;
  else
    return doSomething(x);
}
```

With `CompletableFuture`, it becomes:

```
CompletableFuture<Integer> fooAsync(int x) {
  if (x < 0)
    return CompletableFuture.completedFuture(0);
  else
    return CompletableFuture.supplyAsync(() -> doSomething(x));
```

Extra Example

In the class, I got carried away with the question about `completedFuture` and added the following example for ~~`flatMap`~~ `thenCompose` as well:

Original non-async version:

```
int x = bar(z)
int y = foo(x)
```

Async version:

```
y = barAsync(z)
       .thenCompose(i -> fooAsync(i))
       .get();
```

When we discussed about monad, we say that one way to think of a monad as a wrapper of a value in some context. In the case of `Optional`, the context is that the value may or may not be there. In the context of `CompletableFuture`, the context is that the value not be available yet.

Being a functor and a monad, `CompletableFuture` objects can be chained together, just like `Stream` and `Optional`. We can write code like this:

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m1))
    .thenApply(m -> m.add(m2))
    .thenApply(m -> m.transpose)
    .thenAccept(System.out::println);
```

Another example:

```
CompletableFuture left = CompletableFuture
    .supplyAsync(() -> a1.multiply(b1));
CompletableFuture right = CompletableFuture
    .supplyAsync(() -> a2.multiply(b2))
    .thenCombine(left, (m1, m2) -> m1.add(m2));
    .thenAccept(System.out::println);
```

Similar to `Stream`, some of the methods are terminal (e.g., `thenRun`, `thenAccept`), and some are intermediate (`thenApply`).

Variations

- There are variations of methods with name containing the word `Either` or `Both`, taking in another `CompletableFuture`. These methods invoke the given `Function` / `Runnable` / `Consumer` when either one (for `Either`) or both (for `Both`) of the `CompletableFuture` completes.
- There are variations of methods with name ending with the word `Async`. These methods are called asynchronously in another thread

For example, `runAfterBothAsync(future, task)` would run `task` only after `this` and given `future` is completed.

Other features of `CompletableFuture` include:

- Some methods take in additional `Executor` parameter, for cases where running in the default `ForkJoinPool` is not good enough.
- Some methods takes in additional `Throwable` parameter, for cases where earlier calls might throw an exception.

Handling Exceptions

Handling exceptions is non-trivial for asynchronous methods. Remember that, in synchronous method calls, the exceptions are repeatedly thrown to the caller up the call stack, until someone catches the exception. For asynchronous calls, it is not so obvious. For instance, should we put a catch around `fork()` or around `join()`? A `ForkJoinTask` doesn't handle exception with catch, but instead requires us to check for `isCompletedAbnormally` and then call `getException` to get the exception thrown.

As `CompletableFuture` allows chaining, it provides a cleaner way to pass exceptions from one call to the next. The terminal operation `whenComplete` takes in a `BiConsumer` as parameter -- the first argument to the `BiConsumer` is the result from previous chain (or `null` if exception thrown); the second argument is an exception (null if completes normally).

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m))
    .whenComplete((result, exception) -> {
        if (exception) {
            System.err.println(exception);
        } else {
            System.out.print(result);
        }
    }
```

`whenComplete` returns a CompletableFuture, surprisingly, despite it taking in a `BiConsumer` -- in a sense, `whenComplete` is more similar to `peek` rather than `forEach`.

`handle` is similar to `whenComplete`, but takes in a `BiFunction` instead of a `BiConsumer`, thus allowing the result or exception to be transformed.

Finally, `exceptionally` handles exception by replacing a thrown exception with a value, similar to `orElse` in `Optional`.

```
CompletableFuture
    .completedFuture(Matrix.generate(nRows, nCols, rng::nextDouble))
    .thenApply(m -> m.multiply(m))
    .exceptionally(ex -> Matrix.generate(nRows, nCols, () -> 0));
```

Promise

`CompletableFuture` is similar to `Promise` in other languages, notably JavaScript and C++ (`std::promise`).

CompletionStage

In Java, `CompletableFuture` also implements a `CompletionStage` interface. Thus, you will find references to this interface in many places in the Java documentation. I find this name unintuitive and makes an already-confusing java documentation even harder to read.

Exercise

1. Change the following sequence of code so that `f()`, `g()` and `h()` are invoked asynchronously, using `CompletableFuture`.

   (i)

   ```
   B b = f(a);
   C c = g(b);
   D d = h(c);
   ```

   (ii)

   ```
   B b = f();
   C c = g(b);
   h(c); // no return value
   ```

   (iii)

   ```
   B b = f(a);
   C c = g(b);
   D d = h(b);
   E e = i(c, d);
   ```