Asynchronous

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

Asynchronous calls to a method allows execution to continue immediately after calling the method, so that we can continue executing the rest of our code.

`Asynchronous calls must be handled properly.`

`Your program can exit without even having the result completed if you are not careful.`

`Check on this.`

A `RecursiveTask` has a `isDone()` method that it implements as part of the `Future` interface.

With `isDone()`, we can do something like this.

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

Future

`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.

We should occupy the CPU while waiting for a result, instead of occupying the CPU by doing pointless busy waiting. Busy waiting should be avoided at all costs

This can get problematic as situations like this can arise:

```
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();
}
```

It's getting out of hand!

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

To do the callback, Java introduces the class `CompletableFuture<V>` which implements the `Future<V>` interface. This class is more powerful as we can specify an asynchronous task and an action to perform when the task completes.

The notion of "complete" is important for `CompletableFuture`. If the `CompletableFuture` object 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.

Creating a CompletableFuture object

Call one of its static methods.

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

`future` completes when m1.multiply(m2) returns.

Doing something when the CompletableFuture object completes.

We can use the `thenAccept` method.

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

Waiting for Completion

`join()` can be called.

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

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

On the other hand, there is `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 and Monad.

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)
```

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

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`.

Monad

The `thenCompose` is analogous 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.

Using that property, synchronous methods can be converted into asynchronous methods easily.

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

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

When we talked about monad, we say that one way to think of a monad is 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 may not be available yet.

Being a functor and a monad, `CompletableFuture` objects can be chained together, just like `Stream` and `Optional`.

For 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 methods are terminal (e.g. `thenRun`, `thenAccept`), and some are intermediate (`thenApply`).

Variations

There are other variations of methods. Explore the java 9 API.

Handling Exceptions for asynchronous methods

These exceptions are non-trivial. It is not obvious where to catch the error pokemon. Should we catch around fork or 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.

Completable future has a way of handling exceptions by chaining.

One is `handle` which takes in a `BiFunction`, the first being the result, the second being the exception, thus allowing the result or exception to be transformed.

// this is the blue box in mkdocs. (!!!) !!! note "Promise" `CompletableFuture` is similar to `Promise` in other languages, notably JavaScript and C++(std::promise).