Asynchronous to Synchronous Problem

A real-life interview question asking to convert asynchronous execution to synchronous execution.

Problem

This is an actual interview question asked at Netflix.

Imagine we have an `Executor` class that performs some useful task asynchronously via the method `asynchronousExecution()`. In addition the method accepts a callback object which implements the `Callback` interface. the object's `done()` gets invoked when the asynchronous execution is done. The definition for the involved classes is below:

Executor Class

```
public class Executor {

    public void asynchronousExecution(Callback callback) throws Excep
tion {

        Thread t = new Thread(() -> {
            // Do some useful work
            try {
            // Simulate useful work by sleeping for 5 seconds
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
            }
            callback.done();
        });
        t.start();
    }
}
```

Callback Interface

```
public interface Callback {
```

```
    public void done();
}
```

An example run would be as follows:

```
class Demonstration {
    public static void main( St
        Executor executor = new
        executor.asynchronousEx
            System.out.println(
        });

        System.out.println("ma
    }
}

interface Callback {

    public void done();
}


class Executor {

    public void asynchronousExe

        Thread t = new Thread(
            // Do some useful w
            try {
                Thread.sleep(50
            } catch (Interrupte
            }
            callback.done();
        });
        t.start();
    }
```

Note how the main thread exits before the asynchronous execution is completed.

Your task is to make the execution synchronous without changing the original classes (imagine, you are given the binaries and not the source code) so that main thread waits till asynchronous execution is complete. In other words, the highlighted **line#8** only executes once the asynchronous task is complete.

Solution

The problem here asks us to convert asynchronous code to synchronous code without modifying the original code. The requirement that the main thread should block, till the asynchronous execution is complete hints at using some kind of notification/signalling mechanism. The main thread *waits* on something, which is then *signaled* by the asynchronous execution thread. Semaphore is the first thought that may come to your mind for solving this problem. However, I was told to use primitive Java synchronization constructs i.e. `notify()` and `wait()` methods on the `Object` class.

Since we can't modify the original code, we'll extend a new class `SynchronousExecutor` from the given `Executor` class and override the `asynchronousExecution()` method. The trick here is to invoke the original asynchronous implementation using `super.asynchronousExecution()` inside the overridden method. The overridden method would look like:

```
public void asynchronousExecution(Callback callback) throws Exceptio
n {
    // Pass something to the base class's asynchronous
    // method implementation that the base class can notify on
    // Call the asynchronous executor
    super.asynchronousExecution(callback);
    // Wait on something that the base class's asynchronous
    // method implementation notifies for
}
```

Next, we can create an object for notification and waiting purposes

```
public void asynchronousExecution(Callback callback) throws Exceptio
n {
    Object signal = new Object();
    // Call the asynchronous executor
    super.asynchronousExecution(callback);
    synchronized (signal){
        signal();
    }
}
```

Now we need to pass the `signal` object to the superclass's `asynchronousExecution()` method so that the asynchronous execution thread can `notify()` the `signal` variable once asynchronous execution is complete. We pass in the `callback` object to the super class's method. We can wrap the original callback in another callback object and pass also in our `signal` variable to the super class. Let's see how we can achieve that:

```
public void asynchronousExecution(Callback callback) throws Exceptio
n {

    Object signal = new Object();
    Callback cb = new Callback() {
        @Override
        public void done() {
            callback.done();
            synchronized (signal) {
                signal.notify();
            }
        }
    };
    // Call the asynchronous executor
    super.asynchronousExecution(cb);
    synchronized (signal) {
            signal.wait();
    }
}
```

Note that the variable `signal` gets *captured* in the scope of the new callback that we define. However, the captured variable must be defined `final` or be effectively `final`. Since we are assigning the variable only once, it is effectively `final`. The code so far defines the basic structure of the solution and we need to add a few missing pieces for it to work.

Remember we can't use `wait()` method without enclosing it inside a while loop as supurious wakeups can occur. Let's fix that

```
public void asynchronousExecution(Callback callback) throws Exceptio
n {
    Object signal = new Object();
    boolean isDone = false;
    Callback cb = new Callback() {
```

```
        @Override
        public void done() {
            callback.done();
            synchronized (signal) {
                signal.notify();
                isDone = true;
            }
        }
    };
    // Call the asynchronous executor
    super.asynchronousExecution(cb);
    synchronized (signal) {
        while (!isDone) {
            signal.wait();
        }
    }
}
```

Note that the invariant here is `isDone` which is set to true after the asynchronous execution is complete. The last problem here is that `isDone` isn't `final`. We can't declare it final because `isDone` gets assigned to after initialization. At this a slighly less elegant but workable solution is to use a boolean array of size 1 to represent our boolean. The array can be final because it gets assigned memory at initialization but the contents of the array can be changed later without compromising the finality of the variable.

```
@Override
public void asynchronousExecution(Callback callback) throws Exceptio
n {
    Object signal = new Object();
    final boolean[] isDone = new boolean[1];
    Callback cb = new Callback() {
        @Override
        public void done() {
            callback.done();
            synchronized (signal) {
                signal.notify();
                isDone[0] = true;
            }
```

```
        }
    };
    // Call the asynchronous executor
    super.asynchronousExecution(cb);
    synchronized (signal) {
        while (!isDone[0]) {
            signal.wait();
        }
    }
}
```

The complete code appears below:

```
class Demonstration {
    public static void main( String args[] ) throws Exception {
        SynchronousExecutor executor = new SynchronousExecutor();
        executor.asynchronousExecution(() -> {
            System.out.println("I am done");
        });

        System.out.println("main thread exiting...");
    }
}

interface Callback {

    public void done();
}


class SynchronousExecutor extends Executor {

    @Override
    public void asynchronousExecution(Callback callback) throws Exception {

        Object signal = new Object();
        final boolean[] isDone = new boolean[1];

        Callback cb = new Callback() {

            @Override
            public void done() {
                callback.done();
                synchronized (signal) {
                    signal.notify();
                    isDone[0] = true;
                }
            }
        };

        // Call the asynchronous executor
        super.asynchronousExecution(cb);

        synchronized (signal) {
            while (!isDone[0]) {
```

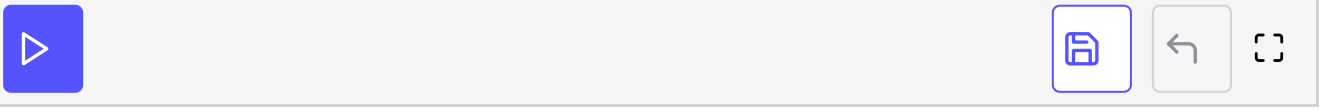
```
                signal.wait();
            }
        }


    }
}

class Executor {

    public void asynchronousExecution(Callback callback) throws Exception {

        Thread t = new Thread(() -> {
            // Do some useful work
            try {
                Thread.sleep(5000);
            } catch (InterruptedException ie) {
            }
            callback.done();
        });
        t.start();
    }
}
```



Note the main thread has its print-statement printed after the asynchronous execution thread print its print-statement verifying that the execution is now synchronous.

Follow-up Question: Is the method `asynchronousExecution()` thread-safe?

The way we have constructed the logic, all the variables in the overridden method will be created on the thread-stack for each thread therefore the method is threadsafe and multiple threads can execute it in parallel.