220

# Concurrency Patterns

Dr Robert Chatley - rbc@imperial.ac.uk

@rchatley #doc220

## Threads

```
public class ThreadExample {

    public static void main(String[] args) {
        new MyThread("A").start();
        new MyThread("B").start();
    }
}

class MyThread extends Thread {

    private final String name;

    public MyThread(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        int i = 0;
        while(true) {
            System.out.println("This is " + name + ". Counting: " + i);
            sleepForRandomTime();
            i++;
        }
    }
}
```

#doc220

The most basic way of running code concurrently in Java (and some other languages) is to create a new thread, and to specify the way it should behave. In Java a familiar way of doing this is by writing a class that extends the built-in Thread class, overriding its `run()` method, instantiating it and calling `start()`. Some other languages also have similar mechanisms.

From our previous experience, we should be a little hesitant of the use of inheritance here. What if we want to re-use this computation in another context where we aren't using threads?

## Runnables

```
public class RunnableExample {

    public static void main(String[] args) {
        new Thread(new MyTask("A")).start();
        new Thread(new MyTask("B")).start();
    }
}

class MyTask implements Runnable {

    private final String name;

    public MyTask(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        int i = 0;
        while(true) {
            System.out.println("This is " + name + ". Counting: " + i);
            sleepForRandomTime();
            i++;
        }
    }
}
```

#doc220

We can break the coupling that we get with inheriting from Thread by composing together objects instead, following the strategy pattern. In Java a typical way of doing this is to create a Runnable which encapsulates the behaviour that we want to run in a separate thread. This is just an object that implements a run() method, but it doesn't extend from Thread. We pass the Runnable to a Thread's constructor to create the composition that we want.

```
public class RunnableNowExample {

    public static void main(String[] args) {

        MyTask myTaskA = new MyTask("A");
        MyTask myTaskB = new MyTask("B");

        // run synchronously
        myTaskA.run();
        myTaskB.run();

        // run asynchronously
        new Thread(myTaskA).start();
        new Thread(myTaskB).start();
    }
}

class MyTask implements Runnable {

    private final String name;

    public MyTask(String name) {
        this.name = name;
    }

    ....
```

# Sync or Async

#doc220

A Runnable just encapsulates some behaviour to be run at a particular time. We can either run it now, or later - synchronously or asynchronously. We can use a Runnable without a Thread just by calling its run() method directly.

---

```
public class CallableExample {

    public static void main(String[] args) throws Exception {
        MyCallable myTask = new MyCallable("A");
        Double totalDelay = myTask.call();
        System.out.println("Total delay was: " + totalDelay);
    }
}

class MyCallable implements Callable<Double> {

    @Override
    public Double call() throws Exception {
        int i = 0;
        double totalDelay = 0;

        while(i < 5) {
            System.out.println("Count: " + i);
            try {
                double delay = Math.random() * 5000;
                totalDelay += delay;
                Thread.sleep((long) delay);
                i++;
            } catch (InterruptedException e) {
                // continue
            }
        }
        return totalDelay;
    }
}
```

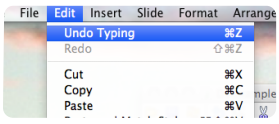
# Callables

#doc220

A Callable is an extension of this idea introduced in more recent versions of the JDK. It is like a Runnable, but allows values to be returned from the call() method, and also allows exceptions to be thrown if necessary.

---

# Command



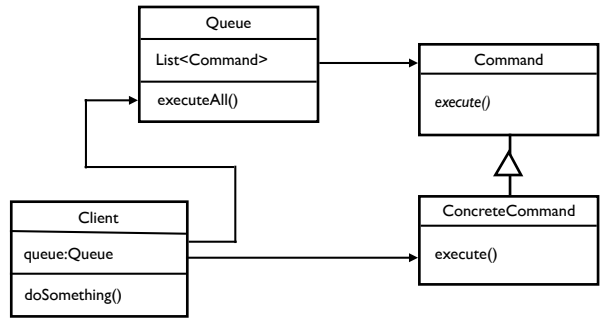

Wrap up a piece of behaviour in a object
to do now, or later, or repeatedly.
Operations may be queued or undone.

A command is an instruction to do something.

#doc220

Both Runnable and Callable are examples of a device where we can wrap up a piece of behaviour in an object, so that we can pass that behaviour around, execute in a different context etc. We can also append command objects to a queue or log, so that they can be executed in a batch, or we can keep a history of past commands executed to replay them later (for example as an undo/redo function).

## Command: Pattern




#doc220

Here is the abstract Command design pattern with its collaborators. A client creates a command, and adds it to a queue of commands, which may be executed later as a batch.

---

## Queuing Up Commands

```
public class QueuingExample {

    public static void main(String[] args) {

        Queue<Runnable> queue = new LinkedList<Runnable>();
        queue.add(new Command("A"));
        queue.add(new Command("B"));

        for (Runnable command : queue) {
            command.run();
        }
    }
}

class Command implements Runnable {

    private final String name;

    public Command(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("This is command " + name + " starting");
        // [snip]
        System.out.println("This is command " + name + " finishing");
    }
}
```

#doc220

Here is an example of doing this in Java, using a Queue<Runnable>. We create a concrete Command class that implements Runnable, instantiate it a couple of times, add these commands to the queue, then when we're ready, iterate over the items in the queue and execute them in succession.

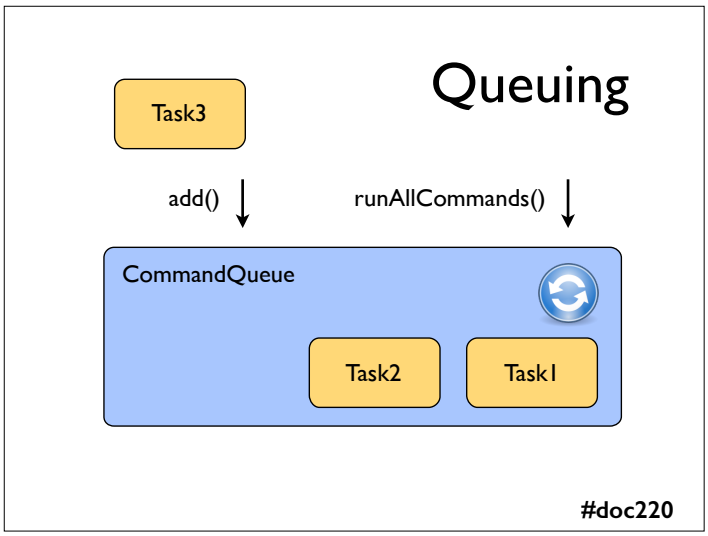
---

## Tell, Don't Ask

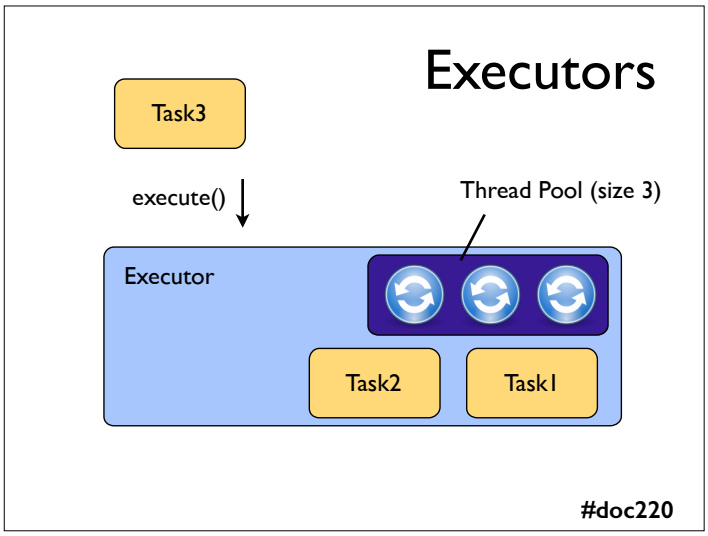
```
public class QueuingExample {

    public static void main(String[] args) {

        CommandQueue queue = new CommandQueue();

        queue.add(new Command("A"));
        queue.add(new Command("B"));

        queue.runAllCommands();

        System.out.println("done");
    }
}

class CommandQueue {

    private Queue<Runnable> queue = new LinkedList<Runnable>();

    public void add(Command command) {
        queue.add(command);
    }

    public void runAllCommands() {
        for (Runnable command : queue) {
            command.run();
        }
    }
}
```

#doc220

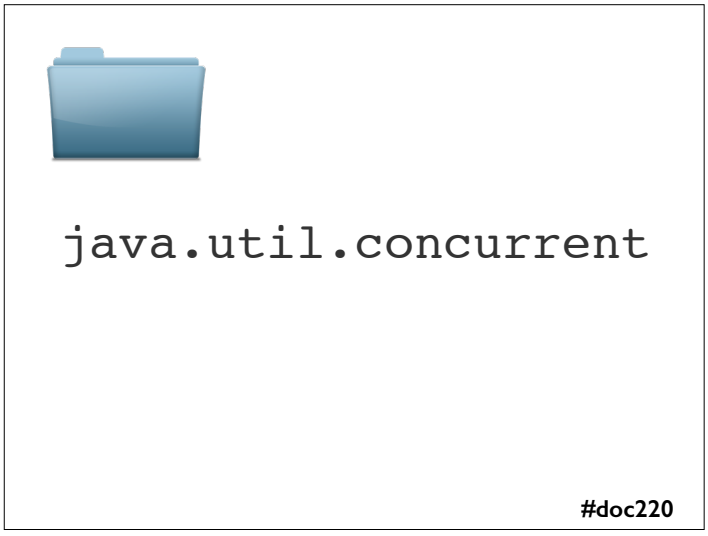
With the previous example, we were using an external iterator, and asking the queue to give us the individual commands. If we follow more of a Tell, Don't Ask style, then we can create our own CommandQueue class that encapsulates the queue and the iteration. Then we just instruct the CommandQueue to execute all the queued commands when we are ready.




This diagram shows a schematic of the way that the previous example executes. All in the context of a single thread, we add tasks to the command queue, and then later tell the CommandQueue to execute all the commands that it is currently holding, in order, one at a time.




Now we are in a position where we can exploit the possibility of parallelism by using more threads, but without it having any effect on the calling code. We don't have to explicitly create or manage any threads. If instead of our CommandQueue, we use an Executor (which is built in to the JDK), we can have exactly the same scheme, but the Executor can manage a pool of threads, and use these to execute our tasks concurrently.




Executor is part of the package java.util.concurrent. There is lots of good stuff in there. This package presents a number of concurrency abstractions that allow you to write programs that run concurrently without having to explicitly create and manage threads.

```java
public class CommandExample {

    public void run() {

        Executor queue = Executors.newFixedThreadPool(3);

        for (int i = 1; i <= 10; i++) {
            queue.execute(new Command(i));
        }
    }

    private static class Command implements Runnable {
        private int n;

        public Command(int n) {
            this.n = n;
        }

        @Override
        public void run() {
            System.out.println("Task " + n + " started");
            sleepSeconds((int) (10 * Math.random()));
            System.out.println("Task " + n + " completed");
        }

        private void sleepSeconds(int seconds) {
          ....
        }
    }
}
```

Java's Executors allow us to run multiple tasks (commands) in parallel.

Runnable gives us a generic interface for commands. You might want something more specific.

#doc220

Here we show an example where we create an Executor that manages a threadpool of size 3. We then submit 10 tasks to our execution queue, and the executor runs them using whichever of the three threads becomes available next. By observing the output of this program we can see that some of the tasks are run in parallel. To vary the number of threads used (and therefore the overall execution time) we only have to vary the size of the thread pool.

---

# Waiting for Results

```java
public static void main(String[] args) throws Exception {

    MyCallable myTask = new MyCallable("A");

    ExecutorService executor = Executors.newFixedThreadPool(2);

    Future<Double> future = executor.submit(myTask);

    doSomeThingElseWhileItsCalculating();

    Double result = future.get();
    System.out.println("Calculation result was: " + result);
}
```

Future

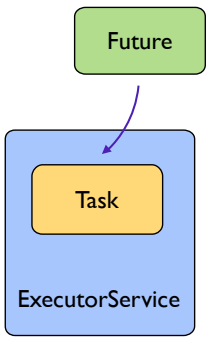
Block until the result is available

#doc220

Sometimes we would like to execute functions in parallel that compute a value. If we use a Callable, then when we submit the task to the Executor, it gives us back a Future, which is something that we can use to track the execution of our task.

---

# Futures




- get()
- get(timeout)
- cancel(mayInterruptIfRunning)
- isDone()
- isCancelled()

#doc220

There are several different tracking methods available on a Future, allowing us to ask whether the task is done yet, to cancel it, or to wait for it to finish and retrieve the return value of the call() method.

# Is Everything Done?

ExecutorService extends Executor with more functionality

```
public static void main(String[] args) {

    ExecutorService executorService = Executors.newFixedThreadPool(2);

    executorService.submit(new MyTask("A"));
    executorService.submit(new MyTask("B"));
    executorService.submit(new MyTask("C"));
    executorService.submit(new MyTask("D"));

    executorService.shutdown();

    try {
        executorService.awaitTermination(120, TimeUnit.SECONDS);
    } catch (InterruptedException e) {
        // interrupted
    }

    System.out.println("All Threads finished");
}
```

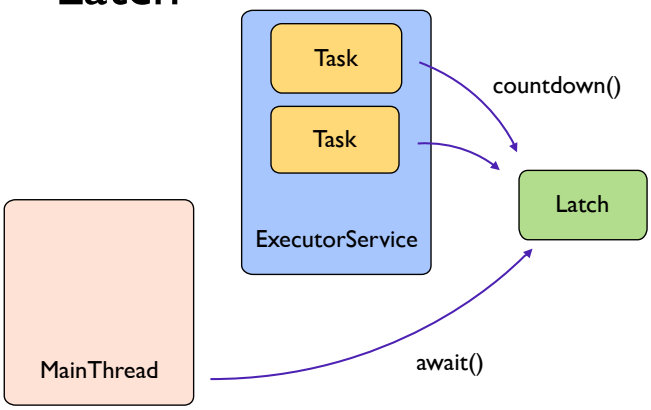
#doc220

What about if we don't care so much about an individual return value, but we want to wait for all tasks to complete - perhaps we want to inform the user, or perhaps we just want to quit the application when everything is done.

We can use an ExecutorService instead of a basic Executor and this gives us access to a bit more functionality. In the code above, after we have submitted all of our tasks, we tell the executor to shutdown, and then we wait for termination. Once this call returns, we know that the executor's queue is empty (or the timeout has expired).

---

# Latch




#doc220

There are alternative ways of doing this. What if we don't want to shut down our ExecutorService (we want to use it again for the next set of tasks), we just want to know when all of the tasks in one group are complete so that we can proceed. We can use a Latch for that. We create the latch in the main thread, with a counter set to the number of tasks to do. We pass the latch to each task, and at the end of the task, they make the latch count down. In the main thread we wait for the latch count to reach zero.

---

```
public class LatchExample {

    public static void main(String[] args) throws Exception {

        ExecutorService executor = Executors.newFixedThreadPool(2);
        CountDownLatch latch = new CountDownLatch(4);

        executor.submit(new LatchedTask("A", latch));
        executor.submit(new LatchedTask("B", latch));
        executor.submit(new LatchedTask("C", latch));
        executor.submit(new LatchedTask("D", latch));

        latch.await();

        System.out.println("All finished");
    }
}

class LatchedTask implements Runnable {

    private final String name;
    private final CountDownLatch latch;

    public LatchedTask(String name, CountDownLatch latch) {
        this.name = name;
        this.latch = latch;
    }

    @Override
    public void run() {
        System.out.println("Starting " + name);
        sleepForRandomTime();
        System.out.println("Finished " + name);

        latch.countDown();
    }
}
```

#doc220

Here is the associated code sample, using the latch to co-ordinate and wait for completion of the four tasks.

# Ruby


Thread.new

pass a block

not so sophisticated

people don't tend to write multi threaded apps

multi process more common

#doc220

Let's briefly consider how this compares to the other languages that we are thinking about in this course. In Ruby there is a Thread abstraction, and you can create new threads, but there isn't such a complex library of concurrency tools and patterns, as people don't tend to write such complex code in Ruby, or if they want to deal with concurrency they more commonly do it by having separate applications running in separate processes, rather than one multi-threaded application.

---

# NodeJS


only one thread

don't block

scheduled

abstractions to effect parallelism

#doc220

In JavaScript we depend on the execution environment to determine the concurrency rules, but when running in the browser, or running server-side using Node.js, we only have one thread. It isn't possible to create any more. Instead, the virtual machine schedules our code and callbacks to run when the thread is free. A tenet of programming in these environments is not to block waiting for external events (for example I/O), because during that time the thread could be running a different part of your application.

---

# async.parallel


```
var async = require('async');

var taskA = function(callback) {
  console.log("Doing task A");
  callback(false);
}

var taskB = function(callback) {
  console.log("Doing task B");
  callback(false);
}

var afterwards = function(err, results) {
   console.log("All done!");
}

async.parallel([taskA, taskB], afterwards);
```

#doc220

There are ways of running things "in parallel", but in fact they are just run interleaved - not truly in parallel. Here we define two tasks as functions. At the end of each task they call a callback. This is a way of saying "when you've finished executing that - call this.

We use the async library with Node.js to run these tasks in parallel. We just pass it an array of functions to run in parallel, and a callback to let us know when everything is done.

# Gathering Results


```
var async = require('async');

var taskA = function(callback) {
  console.log("Doing task A");
  callback(false, "A");
}

var taskB = function(callback) {
  console.log("Doing task B");
  callback(false, "B");
}

var afterwards = function(err, results) {
   console.log("All done!");
   console.log("Results: " + results);
}

async.parallel([taskA, taskB], afterwards);
```

call callback passing result of task

final callback receives array of all task results

#doc220

Again we can gather results from these parallel executions. Instead of returning our calculated value, we pass it as a parameter to the callback. Then async gathers all of these together, and when it calls the final callback, it passes an array of all the results. This is more of a Tell style, instead of an Ask style where we might wait for return values.

---

# Promises


```
fs.readFile("file.json", function (err, val) {
    if (err) {
        console.error("unable to read file");
    }
    else {
        try {
            val = JSON.parse(val);
            console.log(val.success);
        }
        catch (e) {
            console.error("invalid json in file");
        }
    }
});
```

can be rewritten as

```
fs.readFileAsync("file.json").then(JSON.parse).then(function (val) {
    console.log(val.success);
})
.catch(SyntaxError, function (e) {
    console.error("invalid json in file");
})
.catch(function (e) {
    console.error("unable to read file");
});
```

actually executes asynchronously - when it finishes, the *then* function runs

A different technique used in more modern JavaScript is to use *promises* (normally by importing a promise library, of which there are several). Promises act a bit like futures. An asynchronous method can return a promise, on which you can call then() to configure what method should be called once the asynchronous method returns (either successfully or in error). In the example on the slide, then() also returns a promise, so these can be chained. This style tends to lead to more readable code.

---

# Summary

**`new Thread(...)`**

**`executor.submit(...)`**

#doc220

As a summary, managing threads and concurrency yourself can be difficult, and can detract from the clarity of expression of the core of your program. By taking advantage of abstractions like Commands and Executors, you can separate the important logic of your specific program from the code that manages threads and concurrent execution.