Software Engineering Concepts (COMP3003)

3. Multithreading Design Structures

CRICOS Provide Code: 00301J.
Updated: 10/12/2019.

In theory, monitors, synchronisation and other simple-ish threading mechanisms let us do whatever we want.

But, in the best tradition of software engineering, "whatever we want" is not good enough. To a certain extent, we need to be saved from ourselves, from our own propensity to make horrible mistakes. And let's be clear about this: between race conditions, deadlocks, unsynchronised memory, uninterruptible blocking and incomprehensible code, multithreading lets us make some pretty bad mistakes.

Fortunately, modern APIs gives us higher-level ideas to side-step the chaos:

- **Blocking queues**, to more easily and safely transfer data between threads;
- **Thread confinement**, where we "confine" a particular resource to a particular thread, GUIs being the most notable example of this;
- **Futures** and **promises**, to represent values that are going to be produced by tasks not yet complete, so you don't need a complex notification system for when the tasks actually do complete;
- **Thread pools**, to manage performance issues while running many tasks simultaneously.

1 Producers, Consumers and Blocking Queues

The **Producer Consumer pattern** provides an intuitive way of thinking about thread communication.[3] In most cases, you can classify communicating threads as either **producers** or **consumers**. Producers progressively supply data at a particular rate (depending on the task), and consumers use that data to perform their own work, at a particular rate.

You already know how to use monitors for thread communication, by calling `wait()` and `notify()` (or `notifyAll()`). But there are a lot of parts to this, as we saw, and they have to go in a very specific sequence. Put something in the wrong order, and you can create very strange, unpredictable problems.

Blocking queues help sort this out. In Java, there exists a `BlockingQueue` interface, and in .NET a `BlockingCollection` class. As you know, a queue has two basic operations: you can add data to the start, and remove it from the end ("first-in-first-out", or FIFO). In an ordinary queue, either operation can fail. If the queue is full, data cannot be added, and if the queue is empty, data cannot be removed.

In a blocking queue, these operations *block* instead of failing. If you read (`take()`) from an empty blocking queue, the queue will simply wait until another thread adds something. Likewise, if you attempt to write (`put()`) to a full blocking queue, the queue will first wait until another thread removes something. Blocking queues have thread safety guarantees built-in, since they

are expressly intended for multithreading[a].

Let's adapt the fireworks example to illustrate:

Listing 1: Using a blocking queue to pass objects from a producer to a consumer.

```
public class FireworksFactory implements Runnable
{
    private BlockingQueue<Firework> queue = new ArrayBlockingQueue<>(25);

    @Override
    public void run()
    {
        try
        {
            while(true)
            {
                // Produce objects
                Firework f = new Firework();
                f.addGunpowder();
                f.addRocket();
                f.addPackaging();

                // Add to blocking queue
                queue.put(f);
            }
        }
        catch(InterruptedException e) {...} // Producer finished
    }

    public Firework getNextFirework() throws InterruptedException
    {
        // Consume firework -- take from blocking queue.
        return queue.take();
    }
}
```

As before, there's a producer thread executing the code inside `run()`, and one (or more) consumer threads simultaneously calling `getNextFirework()`. `put()` puts `Firework` objects into the queue, blocking if the queue reaches its maximum size of 25. Meanwhile, `take()` takes objects out of the queue, blocking if the queue becomes empty. If either `put()` or `take()` blocks, it will un-block as soon as the *other* one is called.

Notice how much complexity we've saved over using monitors. We don't need a `synchronized` statement, any waiting, notifying or condition checking, nor a field to store the data temporarily. The blocking queue handles all of that, as well as letting us store multiple temporary objects. This results in much simpler (and therefore harder to mess up) code.

We do still need to handle `InterruptedException` though. Like other blocking methods, `put()` and `take()` both throw this when the thread's interrupted status is set to `true`.

Figure 1 more abstractly illustrates the simplest arrangement, where one producer thread supplies data to a blocking queue, which is then removed by a consumer thread. However, in general, there could be any number of producers and consumers accessing the same queue, as shown in Figure 2.

[a] It makes no sense to use a blocking queue with just a single thread. It will simply freeze your program forever. If you need a conventional, single-threaded queue, just use a `LinkedList`.

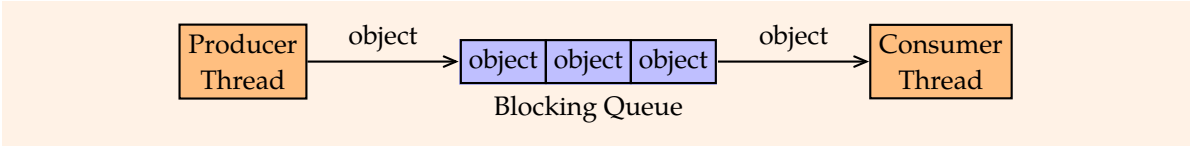

Figure 1: A one-to-one producer-consumer relationship.

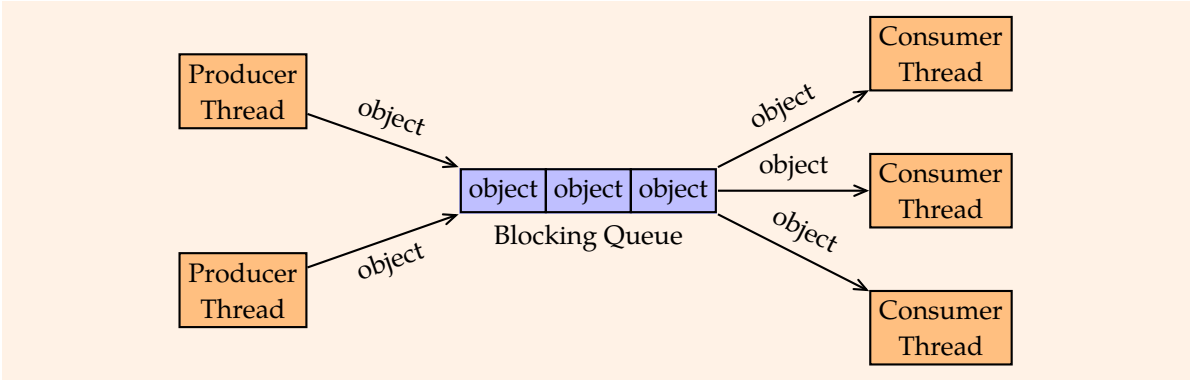

Figure 2: A many-to-many producer-consumer relationship.

However, as Listing 1 illustrates, *structurally* all the communication should happen *within* a class. We still want to hide the details of thread communication from the outside.

1.1 Types of Blocking Queues

Table 1 describes several kinds of blocking queues available in the Java API.

The simplest one is `SynchronousQueue`[b]. In a sense, this isn't a queue at all, because it doesn't store anything, but it fulfils the `BlockingQueue` contract. `ArrayBlockingQueue` is the simplest non-trivial implementation, for the more-usual case where you do actually want some buffering of elements.

Why use `ArrayBlockingQueue` over `SynchronousQueue`? Essentially, performance. Say you have producer and consumer tasks that can produce/consume about the same number of objects per second, but where there's also moment-to-moment variation. A *real* queue can deal with this quite efficiently.

For instance, say the fireworks producer (for whatever reason) is able to speed up production for a while. With an `ArrayBlockingQueue`, it can stockpile fireworks in the queue. Later on, maybe the consumer will become faster than the producer, and this is fine too because it can just process the existing stockpile without waiting. With a `SynchronousQueue`, we can only run at the speed of whichever task is slower, because one must always wait for the other.

So, would you ever choose `SynchronousQueue`? Yes, if the producer task is *much* faster than the consumer, or the other way around. In this case, any actual queue is either going to be always-empty or always-full, so one thread will end up waiting all the time anyway. So you may as well choose the option that minimises memory usage.

1.2 Timeouts and Non-Blocking Operations

`put()` and `take()` are the two most notable blocking queue operations, but there are others:

- `offer(obj)` – puts an object into the queue without blocking. It returns true if it was successful, or false if there's not enough space *right now*.
- `offer(obj, timeout, unit)` – puts an object into the queue, and *does* block if there's not enough space, but only for a limited time. (See below.)

[b] Don't confuse `SynchronousQueue` with a "synchron*ised*" (thread-safe) queue. All types blocking queues are thread-safe.

Table 1: Types of blocking queues in the Java API.

API Class	Functionality
`SynchronousQueue`	A zero-size queue. Whichever of `put()` and `take()` is called first, it blocks until the other one is called (by a different thread). Then the element is transferred directly from one thread to the other.
`ArrayBlockingQueue`	A queue backed by an array. It's **bounded**, having an upper limit on the number of elements (whatever number you like, but it can't change once created). `put()` only blocks when the queue is full, `take()` only blocks when it's empty.
`LinkedBlockingQueue`	A queue backed by a linked list, either bounded or **unbounded**. When bounded, it's basically equivalent to `ArrayBlockingQueue`, but with slightly different memory usage and performance. When unbounded, `put()` *never* blocks, but you might have to worry about the amount of memory the queue uses.
`LinkedBlockingDeque`	Similar to above, but also implements deque (double-ended queue) functionality, where elements can be added and removed at both ends. This has a very specific application, in "fork-join pools".
`PriorityBlockingQueue`	An unbounded queue backed by a heap. When the queue contains multiple elements, `take()` will retrieve whichever element has the "highest priority", which you can define however you like using a `Comparator` object.

- `poll()` – takes an object from the queue without blocking. It returns an object if there's one there, or `null` if there isn't.
- `poll(timeout, unit)` – takes an object from the queue, and blocks for a limited time if it's empty.
- `peek()` – retrieves an object from the queue without removing it (and without blocking).

In the "timeout" versions of `offer()` and `poll()`, one of three things can happen:

1. The method completes successfully right away; or
2. The method blocks, but completes successfully within the timeout period; or
3. The method "times out", blocking for the maximum specified time, but then fails and returns `false` or `null`.

Timeouts are specified using two parameters: a `long` number and a `TimeUnit` constant; e.g.:

Listing 2

```
boolean success = queue.offer(firework, 10L, TimeUnit.SECONDS);
```

This attempts to add an object to the queue, waiting up to 10 seconds for space to become available if it isn't initially. The value of `success` will indicate whether the object was actually added.

1.3 Class Design and Message Passing

We established a rule in lecture 2:

Note 1: Encapsulate thread communication.

> A thread communication mechanism should be hidden (encapsulated) a single class, and not exposed to other classes. This includes blocking queues.

A blocking queue object should be constructed within the class that's going to use it. It should not

be passed as a parameter to any method or constructor. It should not be returned by any method, and (like all class fields) it should be private. No other class should be aware of its existence.

If we follow this rule, each of our classes can have complete control over its own communication. Even with the relative simplicity of blocking queues compared to monitors, there's still a level of coordination required between the producer and consumer threads. Things will be easier if we can manage that coordination in one place, and not have it spread out over several classes.

But the rule still leaves us with options for how our producers and consumers communicate.

In our running example (last seen in Listing 1), we gave the producer class (`FireworksFactory`) a `getNextFirework()` method, to be called by the consumer thread. `FireworksFactory` has to deal with both threads, because it controls the blocking queue, but it's primarily responsible for the producer thread.

Somewhere else there is another class primarily responsible for the consumer thread. Let's see what it might be doing:

Listing 3: Consumer example.

```
public class Festival implements Runnable
{
    private FireworksFactory producer;

    public Festival(FireworksFactory producer)
    {
        this.producer = producer;
    }

    @Override
    public void run()
    {
        try
        {
            while(true)
            {
                // Obtain firework (might throw InterruptedException)
                Firework f = producer.getNextFirework();

                // Do something with it
                f.launch();
            }
        }
        catch(InterruptedException e) {...} // Consumer finished
    }
}
```

Here, the consumer class `Festival` has a reference to the producer, and calls it repeatedly to obtain a piece of information. Note that we don't see a blocking queue here. `Festival` doesn't know about the blocking queue; that's not its responsibility.

But we *could* reverse the situation if we wanted. Instead of the consumer calling the producer to retrieve information, the producer could call the consumer to provide information. It would be a mirror image of what we have so far:

- `Festival` (the consumer) would have the blocking queue, and it would also have a public method for receiving the next firework; e.g., a `putNextFirework()` method, taking a `Firework` parameter.

- `FireworksFactory` (the producer) would have a reference to `Festival`. It would explicitly supply each successive `Firework` object to the consumer by calling `putNextFirework()`.

Which approach is better? Should the producer class have the blocking queue, or the consumer class? As with many things, it depends.

In particular, the "thing" that consumes objects might not be just one thread and class, but multiple threads and/or multiple classes. If so, we'd probably have the producer class controlling the blocking queue, because otherwise it would have to know about all that complexity, in order to know where and when to send its objects. It would be much simpler to let the consumers simply call `getNextFirework()` when each of them is ready.

But we could also face the opposite situation. There could be a complicated set of producer threads/classes, all of which need to send things to the same place. In that case, we would probably let the consumer class control the blocking queue, for an equivalent reason.

In other words, choose the approach that will give you the simplest design in your particular situation!

In the worst case, if there are multiple/complicated producers *and* consumers, you might consider putting the blocking queue in a new class, which will act as a coordinator:

Listing 4: A class for coordinating multiple producers and consumers.

```
public class MarketPlace
{
    private BlockingQueue<Firework> queue = new ArrayBlockingQueue<>(25);

    public void putNextFirework(Firework f) throws InterruptedException
    {
        queue.put(f);
    }

    public Firework getNextFirework() throws InterruptedException
    {
        return queue.take();
    }
}
```

The coordinator doesn't need its own thread. The producers and consumers would simply use it a central organising point. However, don't use this if you don't need to. Creating a new class should generally be avoided unless it solves a specific problem.

1.4 Ending the Queue

The producer and consumer won't necessarily want to run forever, but you have to be careful about how to end them. In particular, if you stop them both at once (say, from some other thread), the blocking queue might still contain unprocessed data. In some cases, this may be ignorable, but in other cases this data might be valuable.

If you want to make sure all the blocking queue data gets processed, you have to let the consumer finish processing it. One solution, an imperfect one, is to set a flag indicating that the queue is in the process of ending. For instance:

Listing 5:
A cancellation flag.

```
public class FireworksFactory implements Runnable
{
    private BlockingQueue<Firework> queue = new ArrayBlockingQueue<>(25);
    private Object mutex = new Object();
    private boolean shutdown = false;

    @Override
    public void run()
    {
        try
        {
            while(true) { ... }
        }
        catch(InterruptedException e) {...}
        synchronized(mutex)
        {
            shutdown = true;
        }
    }

    public Firework getNextFirework() throws InterruptedException
    {
        synchronized(mutex)
        {
            if(shutdown)
            {
                return queue.poll();
            }
            else
            {
                return queue.take();
            }
        }
    }
}
```

In this case, we set the shutdown flag once the producer stops, and then in `getNextFirework()` we "switch" to using the non-blocking `poll()` method, which returns `null` if the queue is empty. And so, in effect, `getNextFirework()` returns `null` if the producer has stopped and there are no more fireworks. The consumer can then shut itself down too once it sees the `null` value.

The issue with this approach is that we're bypassing the queue, and so we need to reintroduce a mutex and `synchronized` statements. Our code is more complex and difficult to understand.

What we can do instead is send the signal through the queue itself. We call this a **poison** value.

Listing 6:
A poison value.

```
public class FireworksFactory implements Runnable
{
    private BlockingQueue<Firework> queue = new ArrayBlockingQueue<>(25);
    private static final Firework POISON = new Firework();

    @Override
    public void run()
    {
        try
        {
            try
            {
                while(true) { ... }
            }
            finally
            {
                queue.put(POISON)
            }
        }
        catch(InterruptedException e) {...}
    }

    public Firework getNextFirework() throws InterruptedException
    {
        Firework result = queue.take();
        if(result == POISON)
        {
            result = null;
        }
        return result;
    }
}
```

Unfortunately, the poison value cannot simply be `null` (at least not in Java). Java's blocking queue classes don't allow `null` values, so we have to create a specific `POISON` object of the same type as the queue accepts.

We put this object into the queue when the producer is interrupted. We've used a `try-finally` block inside the `try-catch`, since `queue.put(POISON)` could throw *another* `InterruptedException`. (There are possible variations on this.)

We then check for the `POISON` object inside `getNextFirework()`. Note the use of `==` rather than `.equals()`, to check for identity rather than equality. This is important. The poison object has to be distinguishable from any valid values that might be in the queue. You might think that `new Firework()` is not going to fulfil this requirement, because a new firework in general ought to be a legitimate non-poison value. However, *every* new object has a unique identity. If we keep this particular one specifically for use as a poison object, then we'll know it when it comes out of the queue again.

2 GUIs and Thread Confinement

One key advantage of blocking queues is that we can eliminate other locking mechanisms, because there's no longer any "shared state". There is no mutable (changeable) information that two threads can access at the same time.

While programming languages give us many ways to manage shared state, its elimination is often desirable for the simplicity it brings. We call this **thread confinement**, where something is only ever accessed by the same thread. It creates a little area of single-threading in your application, where you don't have to worry (too much) about what other threads are doing.

GUIs are the key example of this.

However simple they might seem to the user, GUIs turn out to be exceedingly complex underneath. They are so complex, in fact, that they *cannot* be made multithreaded safely and reliably. Experts have tried and failed, and now believe they understand why.[2]

Note 2: What actually is a GUI?

Let's be clear about our terms. We create graphical user interfaces (GUIs) by using the classes and functions made available by GUI *libraries*. Such libraries include JavaFX, Swing, Windows Presentation Foundation (WPF), Qt, GTK+, UIKit, and hundreds of others. Sometimes they're part of a much larger, all-encompassing application framework.

These libraries let you make a user interface by arranging "widgets" on the screen, like buttons, text fields, lists, etc. GUI libraries display these widgets by communicating with the underlying platform/OS. The underlying platform may actually be another (slightly simpler) GUI library in its own right, with its own (simpler) buttons and text fields, in which case the top-level library that *you're* using may use and build on those simpler constructs. Alternatively, it may stand on its own, and simply tell the operating system what colour each pixel in the window needs to be.

Then there's input, going in the other direction. The operating system notifies the library when a mouse or keyboard button is pressed or released, and also gives the library the coordinates of the mouse cursor and any screen-touch event. The GUI library then works out (by comparing (x, y) coordinates) what widgets these events are related to, and tells your application about it.

2.1 Why Are GUIs Single-Threaded?

The problem is deadlocking. Deadlocks can occur when threads try to lock multiple resources in different orders. The general principle for avoiding deadlocks is this: always lock resources in a consistent order.

For instance, say one thread locks resource A, then (while A is still locked) locks resource B, then (while A and B are still locked) locks resource C. All other threads, if they need the same resources, must also *always* lock them in that same order. If another thread only needs some of the resources, say, A and C, it must still follow the ordering; i.e., it must lock A first, then C.

That way, if you've locked A and B and are waiting on resource C, you know that whichever thread has locked C *is not* in turn waiting for A or B. The other thread couldn't have gotten to the point where it needed resource C if it first needed A or B.

(It's also important to unlock resources in the exact reverse order they were locked in, but the `synchronized` statement takes care of that.)

In GUIs, it turns out to be impossible to keep to this rule, because there is a natural tendency for input (keyboard/mouse/touch events) and output (display/update) operations to access resources in the opposite order. User input "moves upwards", from the operating system, which first detects it, to the lower levels of the GUI library, to individual GUI widgets, and then to your

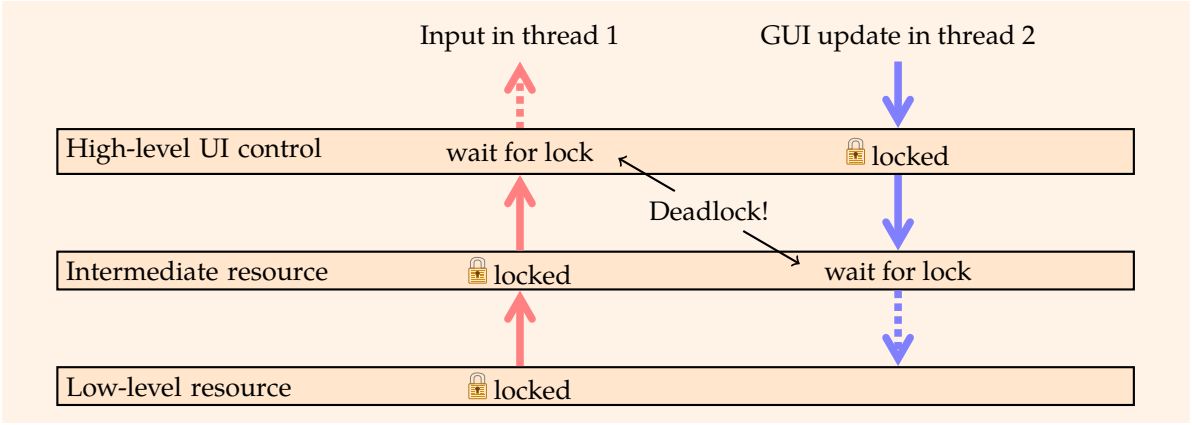

Figure 3: GUI deadlock problem, in a hypothetical multithreaded GUI. Thread 1 performs input while thread 2 performs output, locking resources in opposite orders.

application. Output "moves downwards", following a similar path but in reverse.

If these operations happen in different threads at the same time, then they'll need to lock the resources they use. GUIs are highly mutable, and contain a lot of state information. The input and output threads would likely lock GUI resources in different orders, thereby deadlocking each other. Figure 3 illustrates this.

2.2 The GUI Thread

Thus, in a GUI application (which includes mobile applications), there is always exactly one thread for controlling all user input and output. Different GUI libraries have different names for it. In JavaFX, it's the "JavaFX Application Thread", for instance. The GUI thread will typically be created when you first load the library, or attempt to display your first GUI window.

If your application is simple and boring, the GUI thread may be the only thread you need, and your life will be simple and boring too!

However, if you need to perform any long-running tasks, you need to be aware of two things:

1. You must perform them in another thread, starting one if necessary. Otherwise the GUI will freeze while your task completes (since you'd be hijacking the GUI's only capability for getting input and displaying output).
2. You cannot directly access the GUI from that other thread. It is thread-confined, intended to be accessed only from a single thread. As such, the GUI is *not thread-safe*. If you try to access it from outside its own thread, you'll create race conditions and hence unpredictable failures.

For this purpose, a "long-running task" is anything that takes more than a fraction of a second; anything noticeable to the user. This definitely includes any blocking operations.

To understand the problems here, first consider a very simple task. Say (in JavaFX), we have a `Button` that increments a number in a `Text` field. The `Text` and `Button` are both objects that we setup beforehand. We use `Button.setOnAction()` to register an event handler[c]. When the button is pressed, we do our work:

[c] This is the Observer pattern, in case you're interested!

Listing 7: Okay: fast, single-threaded GUI event handler.

```
// GUI objects:
Text theText = ...;
Button theButton = ...;

// Set up an event handler using Java's lambda syntax:
theButton.setOnAction((event) ->
{
    // When the button is pressed, do this:
    int number = Integer.parseInt(theText.getText()) + 1;
    theText.setText(String.valueOf(number));
});

// OR, using an anonymous class (equivalent but more verbose):
theButton.setOnAction(new EventHandler<ActionEvent>()
{
    @Override
    public void handle(ActionEvent event)
    {
        int number = Integer.parseInt(theText.getText()) + 1;
        theText.setText(String.valueOf(number));
    }
});

// This could also be done in another way using the XML layout file (if you have
// one), but that doesn't help illustrate the next point.
```

From within the event handler, we access `theText.getText()` and `setText()`, to get input and provide output. This all runs inside the GUI thread (basically, it's single-threaded), and while it's running the GUI can't do anything else. That's fine, though, because this task will take *nanoseconds* to run. It's simple and quick enough that the user won't notice the time it takes.

But what if we do have a long running task? Say, attempting to download a webpage and display it:

Listing 8: Bad: long-running, single-threaded GUI event handler.

```
Text theText = ...;
Button theButton = ...;

theButton.setOnAction((event) ->
{
    try
    {   // This will cause the GUI to freeze!
        String content = IOUtils.toString(new URL(theText.getText()), "UTF-8");
        theText.setText(content);
    }
    catch(IOException e) {...}
});
```

`IOUtils.toString()` comes from a third-party library called Apache Commons IO. It takes a URL, and downloads it and returns the contents as a string. This won't always take *that* long, but it could easily be long enough that the user will notice the GUI freezing.

We need to take the downloading out of the GUI thread, by starting a new one:

Listing 9: Solving one problem and creating another: putting the long-running task into a new thread (shown using a blue box).

```
private volatile Thread downloadThread = null;
...

Text theText = ...;
Button theButton = ...;

theButton.setOnAction((event) ->
{
    // The event handler itself runs in the GUI thread
    if(downloadThread == null)
    {
        Runnable downloadTask = () ->
        {                       // This code runs in a new thread, outside the GUI
            try
            {
                String c = IOUtils.toString(new URL(theText.getText()), "UTF-8");
                theText.setText(c); // This will create race conditions!
            }
            catch(IOException e) {...}
            downloadThread = null;
        };

        downloadThread = new Thread(downloadTask, "downloader");
        downloadThread.start();
    }
});
```

There's a bit to take in in Listing 9. The actual downloading code is now in a `Runnable` object (`downloadTask`). The event handler will launch a new thread to run this task when the `Button` is pressed, if one is not already running (i.e., `if(downloadThread == null)`).

We're keeping a reference to the thread (`downloadThread`) in a field, because it needs to be kept track of, both for interruption purposes (if we wanted a "Cancel download" button), and to ensure we don't have multiple simultaneous download threads[d].

(Why `volatile`? I could have had a mutex to lock the `downloadThread` field, because it's accessed by two different threads, but only using operations that are inherently atomic, and hence not susceptible to race conditions. So we need synchronisation but not locking.)

So, having hopefully convinced you that this is all necessary, let's confront the bit where it all goes wrong: `theText.setText()`. This is not thread safe, there isn't any way to make it thread-safe, and we're not in the right thread. The GUI thread could be accessing the same resource for its own reasons at any moment.

We need to somehow run `setText()` from within the GUI thread. Fortunately, every GUI library has a way to do this. In JavaFX it's called `Platform.runLater()`. This takes a `Runnable` as a parameter, and tells the GUI thread to run it. The method name says "later", because the GUI thread could be doing something else right now, but in practice it will run your code *more-or-less* right away.

[d] You'll notice that Java lets us access `theText` and `theButton` from within the task and the event handler, and they're only local variables. But `downloadThread` needs to be a field, because it needs to be modifiable. One of Java's seemingly arbitrary rules is that, if you create a lambda or anonymous class inside a method, it can only *read* local variables from that method (and in fact only variables that are never modified anywhere once assigned). That restriction doesn't apply to fields.

Here's the updated code:

Listing 10: Putting a long-running task into a new thread, *and* using `Platform.runLater()` to access the GUI.

```
private volatile Thread downloadThread = null;
...

Text theText = ...;
Button theButton = ...;

theButton.setOnAction((event) ->
{
    // Inside the GUI thread
    if(downloadThread == null)
    {
        String urlText = theText.getText();
        Runnable downloadTask = () ->
        {                                                  // Outside the GUI thread
            try
            {
                String c = IOUtils.toString(new URL(urlText), "UTF-8");
                Platform.runLater(() ->  // Runnable
                { // Back in the GUI thread again
                    theText.setText(c);
                });
            }
            catch(IOException e) {...}
            downloadThread = null;
        };

        downloadThread = new Thread(downloadTask, "downloader");
        downloadThread.start();
    }
});
```

We've put `text.setText()` inside yet another lambda, which becomes a `Runnable` object when passed to `runLater()`. We've also extracted the call to `getText()` and placed it inside the event handler (so, right after the button is pressed), but not in the new thread.

Thus, success! We have a long-running task, stared by the GUI but which runs in its own thread, but where all interaction with the GUI is still done in the GUI thread.

The JavaFX specification doesn't say exactly how `runLater()` works, but imagine that your task is being added to a blocking queue, specifically a `BlockingQueue<Runnable>`. The GUI thread then spends its time polling that queue, in between handling input events.

You may begin to see why we distinguish between a thread and a task. Up until now, there's been a one-to-one relationship between them. Whenever we've created a `Runnable` (task), we've used it to define everything the thread does. But here, with `runLater()`, that's not true anymore. The GUI thread typically runs a whole sequence of tasks, one after another.

2.3 Shared State vs Thread Confinement

GUIs need thread confinement to avoid deadlocks, but thread confinement can also let you sidestep a lot of design complexity. There's nothing especially wrong with shared state, where fields are accessed from multiple threads – the opposite of thread confinement. It can be a perfectly

valid solution, but only if you're careful to use `synchronized()` properly. Software engineering is all about choosing the simplest and most robust solution.

(It's worth noting that, if you use shared state and you *don't* use `synchronized()` properly, either by mistake or through naivety, it's very unlikely that you'll get any direct message about it, especially not at compile time. You'll get race conditions and/or deadlocks, which may trigger seemingly-unrelated error messages, or simply freezes, or weirdly-incorrect results, and you need to be prepared to recognise these when they happen.)

However, to implement thread confinement – confining objects to a particular thread – you need to *have* a specific thread to confine them to. This might be a thread that wouldn't otherwise exist.

Say you're writing a social media messaging application, and you have a `PostHistory` class to represent past conversation(s). `PostHistory` objects will need to be updated by both incoming posts (i.e., that other people send to the user), and outgoing posts (composed by the user to others). Its contents will also need to be displayed on the screen.

Internally, `PostHistory` has a `List<Post>` field that stores all the messages. In a shared-state situation, you would simply lock a mutex whenever something wanted to access or update this list:

Listing 11: Shared state example.

```
public class PostHistory // Thread-safe (shared state)
{
    private List<Post> posts = new ArrayList<>();
    private Object mutex = new Object();

    public void addPost(Post msg)
    {
        synchronized(mutex)
        {
            posts.add(msg);
        }
    }

    public List<Post> getPosts()
    {
        synchronized(mutex)
        {
            return Collections.unmodifiableList(posts);
        }
    }
}
// For the sake of the simplicity, say that 'Post' objects are immutable.
```

If we wanted to implement thread confinement here, we'd need, well, a thread. In Listing 12, we have the same basic functionality, but the object starts a new thread, and the `posts` list is only accessed from that thread. The new thread executes `queue.take().run()` indefinitely, taking each sub-task out of the queue and running it.

What are these sub-tasks? The `addPost()` and `getPosts()` methods, which are called from various other threads, are not allowed to directly access the `posts` field. Instead, they must create sub-tasks to do that, and add them to the queue.

The implementation of `getPosts()` is slightly awkward (because we haven't discussed "futures" yet – see below), but we can replace the return value with a callback. So, any outside code that wants to retrieve the posts would do the following:

Listing 12: Thread confinement example.

```
public class PostHistory // Thread-confined
{
    private List<Post> posts = new ArrayList<>();
    private BlockingQueue<Runnable> queue = new ArrayBlockingQueue<>(5);
    private Thread thread = null;

    public void start()        // For simplicity, assume start() and stop() will
    {                          // only be called from a single other thread.
        Runnable task = () ->
        {
            try
            {   // Simply run all the Runnables in the queue, one after another.
                while(true) { queue.take().run(); }
            }
            catch(InterruptedException e) {...}
        };
        thread = new Thread(task, "post-history-thread");
        thread.start();
    }

    public void stop() { ... } // We'll gloss over the stopping logic for now.

    public void addPost(Post p) throws InterruptedException
    {                                  // Don't directly add a post, but
        queue.put(() -> posts.add(p)); // queue a task to do it.
    }

    public void getPosts(PostCallback cb)
        throws InterruptedException    // Don't directly return posts, but
    {                                  // queue a task to call a callback.
        queue.put(() -> cb.provide(Collections.unmodifiableList(posts)));
    }
} // We assume interface 'PostCallback' has a 'provide(List<Post>)' method.
```

Listing 13: Thread confinement: retrieving confined information.

```
PostHistory ph = ...;
ph.getPosts((posts) -> // Instance of PostCallback, called in PostHistory's own
{                      // thread.
    ... // Now do something important with 'posts' (which is a copy of
});     // PostHistory's own 'posts' list).
```

Although we've now achieved thread confinement, there are also more elaborate ways of *enforcing* it, to avoid accidentally sharing a confined resource:

- We can (with some design work) implement **stack confinement**, where we arrange for the confined resource to be accessed via the stack, as a local variable, rather than as an object field. It is impossible for two threads to access the same local variable.

  In our example in Listing 12, we would have to refactor our code to declare `posts` as a local variable within the `Runnable` task. We'd have to rethink the way that we pass messages via the queue, because now `addPost()` and `getPosts()` cannot even define a lambda that refers

to `posts`.

- We can (somewhat more easily) use Java's `ThreadLocal<T>` container. This has `get()` and `set()` methods for a single value, but it holds a *different* value for each thread that accesses it. If the wrong thread tries to access the resource, it will simply get `null`, and you'll discover that very quickly.

  In our example, we would have "`private ThreadLocal<List<Post>> posts`", and (when accessing it) write `posts.get().add(...)` or just `posts.get()`.

We've actually designed `PostHistory` to work a little bit like an **actor** in the **actor model**. This is a multithreading concept that avoids shared state, and even avoids explicit usage of threads.[5] In the actor model, your key objects are "actors"[e], which all work concurrently, though the actual threads they use are managed for them, and they communicate via message passing. Each actor has a "mailbox" (a blocking queue) for receiving messages, and can put messages into other mailboxes. Java has no direct support for the actor model, but other languages like Erlang and Scala do.

3 Asynchronicity and Futures

We've already performed some **asynchronous operations** – those that return immediately, and then later (asynchronously) do their actual job. For instance, `thread.start()`, `button.setOnAction()` and `Platform.runLater()` all set up asynchronous operations. The first creates a new thread, the second occurs entirely within the GUI thread, and the second communicates across threads to the GUI thread. In each case, you supply code that is executed at some (slightly) later point.

"Asynchronous" is the opposite of "synchronous", but be careful *not* to jump to conclusions about this. We're not talking about `synchronized` or `SynchronousQueue` here, but something broader. In the general sense, a **synchronous operation** is one that does its job before returning; i.e., a normal function/method call. You've been using them all along.

Asynchronous operations tend to break up your code structurally and visually, with parts happening inside various lambdas and anonymous classes. Having algorithmic steps appear out of sequence, as tends to happen with lambdas and anonymous classes, makes for difficult reading.

Moreover, say we have an algorithm in one thread (let's call it the coordinating thread) that must wait for multiple simultaneous tasks to complete, because it needs all their results together. We could use a monitor and shared state, with each asynchronous task notifying the monitor, and the coordinating thread waiting, and for each notification checking whether it has all the information yet. We could alternatively use a `ArrayBlockingQueue` of size one, to store the single result, which is a bit simpler. (We could technically use `SynchronousQueue`, but that would tie up threads unnecessarily.)

To get a more concrete sense of the issue, consider Listing 14. The goal here is to download a file (from some URL) and save it under a name chosen by the user. For efficiency, we'd like to have the download start *right away*, at the same time as the user is selecting a name for it. We save the file under a temporary name, and then at the end rename it to the name chosen by the user. We use the blocking queue to transfer the asynchronous result (the user's filename) back to the original thread.

(Note: `File` represents a filename, which may or may not correspond to an existing file on disk. `URL.getStream()` gives you an `InputStream` that, when read, will download data from that URL.

[e] In requirements analysis, an "actor" usually means an external entity, like the user or a payment system, but that's *not* what we mean here.

Listing 14: Retrieving an asynchronous result, using a blocking queue.

```
BlockingQueue<File> result = new ArrayBlockingQueue<>(1);
Runnable nameInputter = () -> // Asynchronous task.
{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter filename: ");
    result.put(new File(sc.nextLine()));
};

File tempFile = ...;
URL downloadUrl = ...;

// Asynchronously ask for a new filename (for the file-to-be-downloaded).
new Thread(nameInputter, "name-inputter-thread").start();

try(OutputStream out = new FileOutputStream(tempFile)) // Open temporary file.
{
    // Start downloading to tempFile right away, while a name is chosen.
    IOUtils.copy(downloadUrl.openStream(), out);
    out.close(); // Close to force the OS to finish writing the file.

    // Retrieve asynchronous result, and rename the downloaded file accordingly.
    tempFile.renameTo(result.take());
}
catch(...) {...}
```

`IOUtils.copy()` is again part of Apache Commons IO, and simply reads everything from a given input stream, and transfers it to a given output stream.)

Conceptually, what we're doing is making efficient use of two resources: (1) the network bandwidth, and (2) the user's attention. These are available simultaneously, so we may as well use them simultaneously, and then integrate the result from each.

3.1 Futures and Promises

What we did in Listing 14 is basically what **futures** and **promises** are for.

A future/promise object represents the end-result of an asynchronous operation. By comparison, blocking queues are designed for ongoing tasks that produce/consume many objects over time. You obtain a future object at the *start* of the operation, before the value has actually been produced, and you get its value when the operation finishes.

The relationship between futures and promises can get a bit murky, depending on which language you're using. *In theory,* the difference is that a future provides a way to eventually retrieve an asynchronously-computed value, whereas a promise is a place for the asynchronous task to insert its result.[4] A future is held by the consumer, which cannot change it, and a promise by the producer, which can set its value *once*[f]. We say that a promise **completes** a future when the asynchronous task puts its result into the promise object, and it becomes available to whoever has the future.

Some languages adhere to this dual concept more closely than others. For instance, the Scala documentation[1] says the following:

[f] Just to be clear, there's no queue – just a single end result.

> While futures are defined as a type of read-only placeholder object created for a result which doesn't yet exist, a promise can be thought of as a writeable, single-assignment container, which completes a future. That is, a promise can be used to successfully complete a future with a value (by "completing" the promise) using the `success` method. Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the `failure` method.

C# and other .NET-based languages have futures and promises too, but use different names: `Task<T>` is a future, and `TaskCompletionSource<T>` is a promise.

JavaScript uses the term "promise" (and sometimes the less-standard term "deferred") to mean something that's really both a future and a promise at the same time, as well as providing a chaining callback mechanism. A lot of potential confusion over the terminology arises from this, so beware.

As for Java, brace yourself, because the language designers have made *two* attempts at implementing futures/promises (so far), and we'll discuss both of them.

3.2 Java Futures, Round 1: `FutureTask` and `Callable`

Java's first attempt at implementing futures came in version 5, with a combination of the `Future` and `Callable` interfaces and the `FutureTask` class[g]. We don't really have a proper "promise" here. Rather, when you write an asynchronous task, you simply write a method that returns a value, and this is made available to whoever has the `Future` object.

When you have a `Future` object, you can do the following:

- `get()` – Retrieve the result of the operation, blocking until it's available (optionally with a timeout, just like for the `poll()` and `offer()` blocking queue operations);
- `cancel(false)` – Prevent the operation from starting, if it hasn't already;
- `cancel(true)` – Prevent the operation from starting, or interrupt it if it has already started (without needing to refer to the `Thread` object);
- `isCancelled()` – Check if an operation has been cancelled;
- `isDone()` – Check if an operation is finished, whether complete or cancelled.

Listing 15 shows how we might use one in practice. Here, we obtain a `Future<File>` object right away, though notice that it's really a `FutureTask`, with a `Callable` passed to its constructor. `FutureTask` has a dual nature. It is a placeholder for some future result, *and also* represents the task that produces that result, inheriting from both `Future` and `Runnable`.

The `Callable` object defines what the task actually does. `FutureTask` is just a wrapper around it. (We could also have used a lambda to define the `Callable`, rather than the more verbose anonymous class syntax.)

`Callable.call()` is comparable to `Runnable.run()`. However, `call()` returns a result, and also potentially throws checked exceptions, whereas `run()` cannot do either. Thus, `Callable` is a natural way to define any asynchronous operation that produces a result. The `FutureTask` retrieves the result, and (when asked) passes it over to the consumer thread.

(If the task fails due to an exception, `FutureTask` will transfer the exception object back to the consumer thread. This is a nice trick, because exceptions don't normally cross thread boundaries. If `call()` throws an any exception, `FutureTask` records it, wraps it inside an `ExecutionException`, and throws the `ExecutionException` when you call `get()`. Then you have to "unwrap" the original exception, by calling the exception's `getCause()` method.)

Figure 4 shows a high-level overview of the key classes and interfaces involved in the use of futures. You could argue that Java has overcomplicated the API a bit, but it is what it is. Compro-

[g] And others, but those three will suffice for an introductory explanation.

Listing 15: Retrieving an asynchronous result, with `FutureTask` and `Callable`.

```
Future<File> nameInputter = new FutureTask<>(new Callable<File>()
{                                                    // Asynchronous task.
    @Override
    public File call()
    {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter filename: ");
        return new File(sc.nextLine());
    }
});

// Asynchronously ask for a new filename (for the file-to-be-downloaded).
new Thread(nameInputter, "name-inputter-thread").start();

File tempFile = ...;
URL downloadUrl = ...;

try(OutputStream out = new FileOutputStream(tempFile)) // Open temporary file.
{
    // Start downloading to tempFile right away, while a name is chosen.
    IOUtils.copy(downloadUrl.openStream(), out);
    out.close(); // Close to force the OS to finish writing the file.

    // Retrieve asynchronous result, and rename the downloaded file accordingly.
    tempFile.renameTo(nameInputter.get());
}
catch(...) {...}
```

mises often need to be made to preserve backwards compatibility when things like futures get added to the language.

To summarise the relationships: each `Thread` needs to run a `Runnable` object. `FutureTask` is a kind of `Runnable`, and in turn needs a `Callable` (which you create) to actually do something useful and produce a result. *Then*, the algorithm running in the consumer thread needs a `Future` to retrieve the result. The same `FutureTask` object is also a kind of `Future`, and so can supply the result.

3.3 Java Futures, Round 2: `CompletableFuture`

In version 8, Java introduced `CompletableFuture`, which is both a future and a promise (similar to JavaScript's `Promises`). Some of the methods are designed for use by the producer thread, and some by the consumer thread. It does *not* represent the task itself per se.

`CompletionFuture` implements the `Future` interface, but mostly for backwards compatibility. It also implements the new `CompletionStage` interface, though this isn't hugely important either because no other class does so. It essentially supercedes `FutureTask`, although you should still expect to see code that uses the "old" approach.

Listing 16 shows how to use `CompletableFuture` as a promise. Instead of returning a value, we pass it to `CompletableFuture.complete()`. We can use `get()` in the same way as in Listing 15.

So why did the language designers go to the trouble of making `CompletableFuture`?

First, having a proper *promise* object, where you explicitly store a value, is more flexible than using a return value. Say, instead of console input, we want to pop up a GUI window with a box

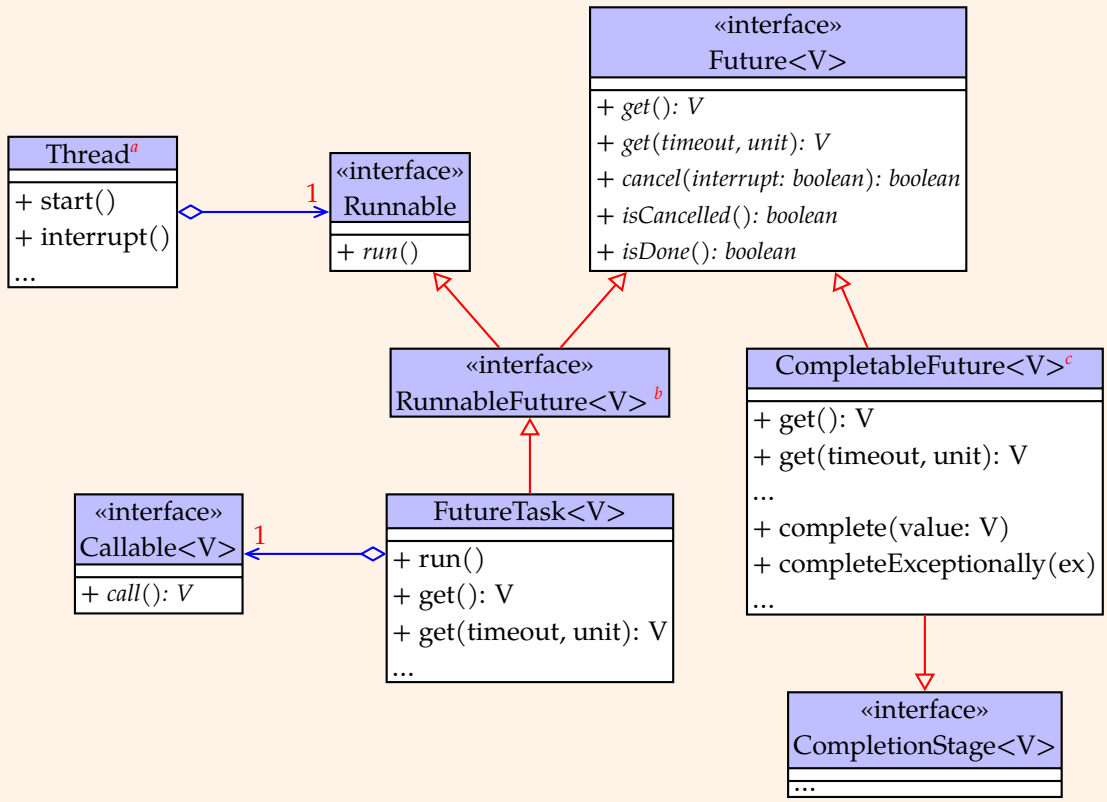

Figure 4: Relationships between key future-related classes in the Java API.

[a] `Thread` also technically inherits from `Runnable`, but this is really just for backwards compatibility.

[b] `RunnableFuture` is not that important for understanding these relationships. Things would mostly be the same if `FutureTask` simply inherited directly from `Runnable` and `Future`.

[c] There are *a lot* of additional methods here which we can't fit on the diagram. See the API documentation!

Listing 16: Setting an asynchronous result, with `CompletableFuture`.

```
CompletableFuture<File> nameInputter = new CompletableFuture<>();
Runnable task = () ->
{
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter filename: ");
    nameInputter.complete(new File(sc.nextLine()));
};
new Thread(task, "name-inputter-thread").start();
... // Continue on as in Listing 15.
```

for the user to enter a filename and an "Ok" button. The problem (for `FutureTask`) is that you cannot easily design a method that actually *returns* this result, while maintaining asynchronicity. There are *two* separate asynchronous operations: one to create and display the window itself, and another to handle the button press. The first returns before the user has inputted anything, and the second is forced to return `void` by the GUI event handling system (i.e., it must implement a GUI-specific interface, `EventHandler` in the case of in JavaFX, not `Callable`).

It is considerably easier to use a promise:

Listing 17: Using `CompletableFuture` to handle a doubly-asynchronous result.

```
CompletableFuture<File> nameInputter = new CompletableFuture<>();
Platform.runLater(() -> // First asynchronous operation
{
    ... // Set up GUI window
    TextField filenameField = ...;
    Button okButton = ...;

    okButton.setOnAction((event) -> // Second asynchronous operation
    {
        nameInputter.complete(new File(filenameField.getText())); // Set result
        ... // Close window
    });
}); // Continue on as in Listing 15 (but don't start a new thread, because we
... // now have the GUI thread handling the input).
```

Second, there is a better way than the blocking method `Future.get()` for retrieving the result. Threads are a kind of resource – it takes time and memory to create and destroy them. We'd like to avoid creating them just to have them sit idle. But what's the alternative?

`CompletableFuture` supports callbacks, for when the task completes. More generally, it allows you to chain both synchronous and asynchronous tasks together, so that when one ends, the next one starts, and (optionally) uses results from the completed one. For instance:

Listing 18: `CompletableFuture` callback.

```
CompletableFuture<String> c1 = ...;
...
c1.thenRun(() ->
{
    System.out.println("Result is " + c1.get());
});
```

In this example, our `println()` will run once `c1` is complete, and therefore we can guarantee that calling `get()` won't block.

This may seem at odds with the idea that `CompletableFuture` doesn't represent a task, and certainly different from how we've used it so far. There is a subtlety here: it's the `thenRun()` method on the *original* `CompletableFuture` that arranges for the new task to start.

In the process of doing that, `thenRun()` also creates and returns a second `CompletableFuture`. We're discarding that here, because this task is the final thing we want to do.

`CompletableFuture` actually has more than *fifty* methods, most of which are designed to chain tasks together in various different ways. For instance:

In Listing 19, we chain together five tasks, each running after the previous one is complete, though `exceptionally()` is slightly different. Each of these methods does two things: (1) starts a task

Listing 19: Chaining `CompletableFutures` together.

```
CompletableFuture
    .supplyAsync(() -> // Task 1
    {
        ... // Return an object.
        return ...;
    })
    .thenApply((obj) -> // Task 2
    {
        ... // Take in Task 1's result, and return a new object.
        return ...;
    })
    .thenCompose((obj) -> // Task 3
    {
        CompletableFuture<MyClass> cf = new CompletableFuture<>();
        ...               // Take in Task 2's result, and return a
        cf.complete(...); // CompletableFuture, which in turn provides another
        ...               // object.
        return cf;
    })
    .thenAccept((obj) -> // Task 4
    {       // Take in Task 3's result - not the intermediate CompletableFuture,
        ... // but the final 'MyClass' object. Return nothing.
    })
    .exceptionally((throwable) -> // Task 5, only done if an error occurs
    {
        ... // Handle error
        return null;
    });
```

once the previous result is known (except the first one, which starts right away), and (2) returns a new `CompletableFuture` representing the result of the new task.

If you count them, there are *six* `CompletableFuture` objects being created here: one for each task, and one intermediate one in task 3.

The chaining methods differ in how they let you define tasks, taking advantage of Java 8's assortment of different "functional interfaces"[h] from the `java.util.function` package. It's worth mentioning a few of these first:

- `Supplier<T>` represents a task with no parameters, but that returns a result (of some type `T`). It's conceptually like `Callable`, but intended for more generic uses.
- `Consumer<T>` represents a task that takes a parameter and returns nothing.
- `Function<S,T>` represents a task that takes a parameter and returns a result.
- `BiConsumer<S,T>` and `BiFunction<R,S,T>` are like `Consumer` and `Function`, but take two parameters.

[h] Functional interfaces were designed for use with various functional-programming constructs introduced in Java 8, such as lambdas. A functional interface in Java is *any* interface that has only one method (strictly speaking, one *non-default* method). Every time you write a lambda, the compiler interprets it as implementing a particular functional interface, depending on the context. These include `Runnable` and `Callable`, even though they originate from previous Java versions.

So, how are these used by the chaining methods? Here are some examples:

- `supplyAsync()` (a static method) starts a new `Supplier` task right away, in another thread. The new `CompletableFuture` represent's the task's return value. (This is superficially a bit like `FutureTask`).
- `thenRun()` starts a new `Runnable` task. The new `CompletableFuture` represents the task's *end*, but there's no value. (Specifically, `thenRun()` returns `CompletableFuture<Void>`, and the value is always `null`.)
- `thenApply()` starts a new `Function` task, where the parameter is the result of the previous task. As with `supplyAsync()`, the new `CompletableFuture` represent's the tasks's return value.
- `thenCompose()` starts a new `Function` task which itself returns a `CompletableFuture`. This gives us the flexibility to handle more complex cases, as in Listing 17, where the task needs to refer to an actual promise object, not just return a value.
- `thenAccept()` starts a new `Consumer` task, where (like `thenApply()`) the parameter is the result of the previous task. Like `thenRun()`, the new `CompletableFuture<Void>` represent's the task's end, but there's no value.
- `exceptionally()` starts a new `Function` task, *only if* the previous task(s) produced an error. The parameter to the task is a `Throwable` (exception) object representing the error. The return value is a substitute value for whatever *should* have been produced. (If `exceptionally()` is the last in the chain, the return value can just be `null` because it has no real purpose.)

These methods let us define tasks in a relatively natural way: using parameters and return values, while allowing for more flexibility when needed. In most cases, therefore, the tasks don't actually need to interact directly with `CompletableFuture` (though they can).

There are still a couple more tricks here.

There are alternate versions of the chaining methods ending in "...`Async`"; e.g., `thenRunAsync()`, `thenApplyAsync()`, etc. (`supplyAsync()` is one too, though there's no `supply()` method). The *non*-`Async` methods simply run their tasks in the same thread as the previous task. The `Async` methods use a different thread. There's a bit of magic to this that won't become clear until we discuss thread pools in the next section[i]. For the time being, imagine that the `Async` methods launch a new thread for each task.

Given this, it makes sense to ask if/how we can use `CompletableFuture` for *parallel* tasks, since that's what we originally had. Indeed we can!

Back in Listing 15, we were asking for user input at the same time as downloading a file. Only once *both* of those had completed did we perform the last simple step: renaming the file.

---

[i] If you've read ahead, then here it is: `CompletionFuture`'s `Async` methods submit their tasks to the executor obtained by calling `ForkJoinPool.commonPool()`. Alternatively, there are overloaded versions of these methods that let you substitute your own executor.

Listing 20: Parallel tasks using `CompletableFuture`.

```
CompletableFuture<File> askFilename = new CompletableFuture<>();
Platform.runLater(() -> { ... }); // User input task - basically what we had in
                                  // Listing 17.
File tempFile = ...;
URL downloadUrl = ...;

CompletableFuture
    .runAsync(() -> // Download task - run in parallel with the user input task.
    {
        try(OutputStream out = new FileOutputStream(tempFile))
        {
            IOUtils.copy(downloadUrl.openStream(), out); // Download file.
        }
        catch(IOException e)
        {
            throw new UncheckedIOException(e);
        }
    })
    .thenAcceptBoth(askFilename, (download, filename) -> // Run when the
    {                                                    // download AND user
        tempFile.renameTo(filename);                     // input are complete.
    })
    .exceptionally((t) -> // Handle all errors here (including from the user
    {                     // input task).
        Platform.runLater(() ->
        {
            new Alert( // Display error dialog box.
                Alert.AlertType.ERROR, t.getMessage(), ButtonType.OK).show();
        });
        return null;
    });
```

`Platform.runLater()` and `CompletableFuture.runAsync()` will start two parallel tasks: waiting for user input, and downloading the file. `CompletableFuture.thenAcceptBoth()` takes in a second `CompletableFuture` as well as a `BiConsumer` task, which has two parameters, one for each task it depends on. This final task will be run once the first two have completed, irrespective of which order they finish in.

(In this particular case, the first parameter (`download`) is not meaningful, because it's of type `Void` and so guaranteed to be `null`. We have to put it there, because the interface demands it, but we just ignore it.)

What if we had more than two parallel tasks? We could use `CompletableFuture.allOf()`, which takes any number of `CompletableFutures`, and creates and returns another one that is instantly completed once all the passed-in ones are complete. There's also `anyOf()`, which creates a new `CompletableFuture` that is instantly completed once *any* of the passed-in futures completes. There are more combinations still, but you'll need to look them up in the API documentation.

Finally, let's briefly discuss how error handling actually works.

`CompletableFuture` can "complete exceptionally" (fail), because something called `cancel()` or `completeExceptionally()`, or because the task threw an exception, specifically one that extends

`RuntimeException`[j].

Most chaining methods *don't* run their task if the previous task fails, but they pass on the failure to the next chaining method. The `exceptionally()` method is the other way around. It *only* runs its task in the event of a failure. It's a bit like a `catch` construct (though you can't specify which exception types it handles – it must simply handle all of them). If you put it at the end of the chain, you can handle all errors generated across all the tasks in one place. You can put one in the middle of the chain, *if* there's a way to recover from an error and keep going.

There are other chaining methods too – `handle()`, `whenComplete()`, and their `Async` versions – that will handle both normal and exceptional cases via a single task, if you need that flexibility.

4 Thread Pools

Threads don't come for free. The operating system expends time and memory to create them. Threads themselves let us utilise hardware resources in parallel, but these resources are limited. If you're already making maximum use of the CPU, network bandwidth, etc., creating a new thread cannot make anything faster. In fact, adding new threads in this case will (very slightly) slow down your application. So, sometimes it's better to wait for an existing thread to become free.

We use **thread pools** to limit the number of threads we create, and also to recycle threads rather than redundantly destroying and re-creating them. A thread pool contains one or more threads[k], as well as a way to submit tasks to run on them. Using thread pools means we no longer know exactly which thread each task is running on, which may take some getting used to.

4.1 Basic Usage

In Java, thread pools are created and accessed via an **executor**. This can mean either the `Executor` or `ExecutorService` interfaces, the latter inheriting from the former. Figure 5 shows the various interfaces and implementing classes. You can create an `ExecutorService` by

- Constructing one of the concrete classes directly (e.g., `ThreadPoolExecutor`);
- Calling one of the static factory methods in `Executors` (note the "s").

The basic `Executor` interface can also be implemented in more creative ways; e.g., a method reference to a method that accepts a `Runnable` and returns `void`. `Platform.runLater()` fulfils this requirement.

---

[j] The functional interfaces generally don't allow checked exceptions to be thrown, but anything inheriting from `RuntimeException` is unchecked.

[k] In the trivial case it could have zero, but then it can't actually do anything.

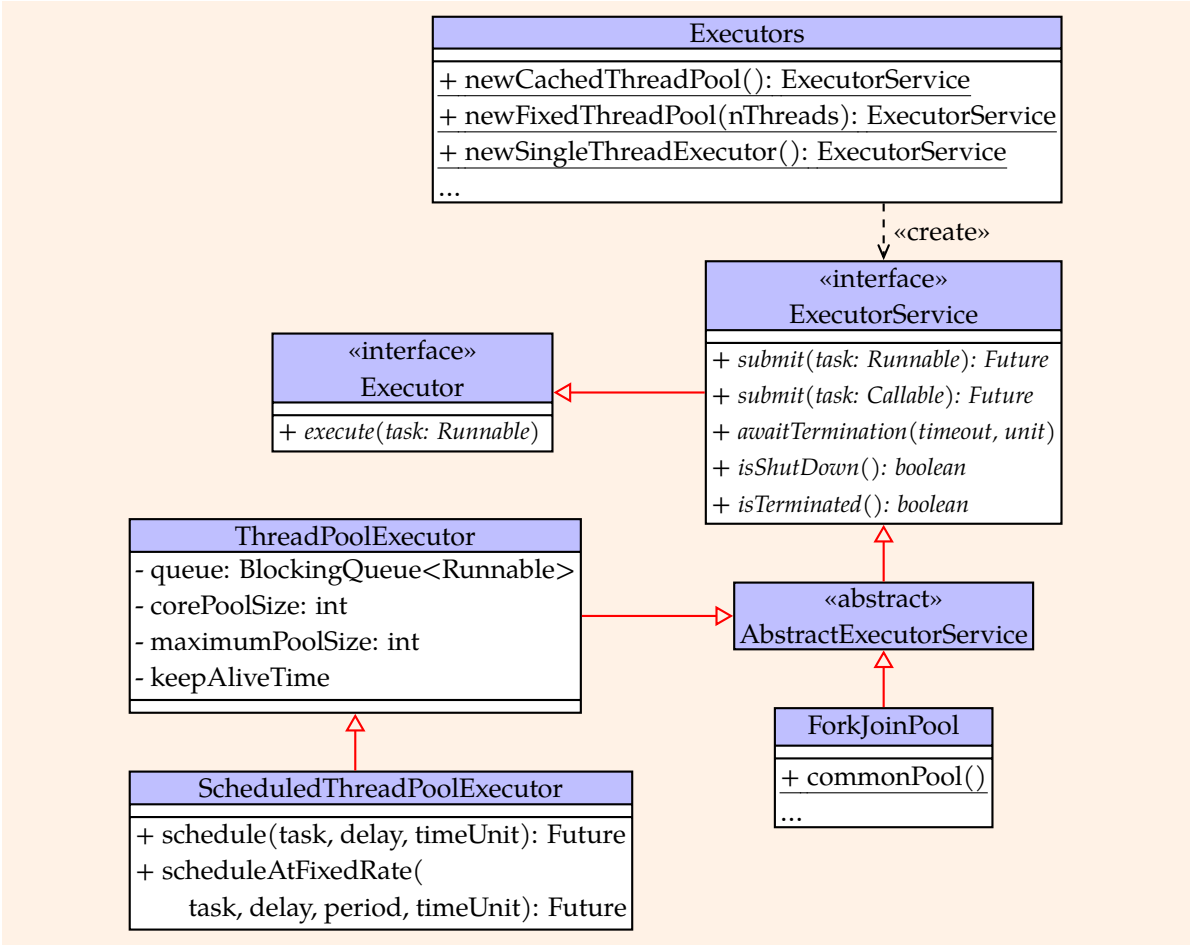

Figure 5: The `Executor` hierarchy in the Java API.

Listing 21: Some ways to obtain executors.

```
ExecutorService es1 = Executors.newCachedThreadPool(); // Unlimited threads.

ExecutorService es2 = Executors.newFixedThreadPool( // As many threads as there
    Runtime.getRuntime().availableProcessors());    // are CPU threads.

ExecutorService es3 = new ThreadPoolExecutor(
    4, 8,                    // Minimum 4 threads, maximum 8.
    15, TimeUnit.SECONDS,    // Destroy excess idle threads after 15 seconds.
    new SynchronousQueue<>() // Used to deliver new tasks to the threads.
);

ExecutorService es4 = ForkJoinPool.commonPool(); // Pre-configured executor.

Executor es5 = Platform::runLater; // Treat the GUI thread as a pool of 1.
```

In any case, once we have an executor, we can use it (at a very simple level) as in Listing 22.

Of course, most of the time you actually have an `ExecutorService`, which is a bit more sophisticated than `Executor`. `ExecutorService` provides a `Future` when you submit a task, via one of the `submit()` methods as Listing 23 shows. Unfortunately this isn't a `CompletableFuture`. It's not guaranteed to be a `FutureTask` either, but in any case it has the same limitations as `FutureTask`.

In fact, we can use `CompletableFuture` with executors after all. For every `…Async` method in `CompletableFuture`, there's an overloaded version that accepts an `Executor` parameter after the task. The method will use this executor to run the task, as in Listing 24.

Listing 22: Using a plain vanilla `Executor`.

```
private Executor exec = ...;
...
exec.execute(() -> // This is a Runnable.
{
    ... // Define your task here
});

// OR
exec.execute(this::myMethod); // myMethod() must be void and have no parameters.
```

Listing 23: Submitting a `Callable` to an `ExecutorService` to obtain a `Future`.

```
private ExecutorService es = ...;
...
Future<MyClass> future = es.submit(() -> // This is a Callable.
{
    ... // Define your task here
    return new MyClass();
});

// OR
future = exec.submit(this::myMethod); // myMethod() must return MyClass.
```

Listing 24: Using `Executors` with `CompletableFutures`.

```
private Executor exec = ...;
...
CompletableFuture
    .supplyAsync(() ->
    {
        ... // Run task 1 on 'exec'
        return new MyClass();
    }, exec)
    .thenAcceptAsync((obj) ->
    {
        ... // Run task 2 on 'exec'
    }, exec);
```

This raises a question that we glossed over before: what happens when you *don't* specify the executor? Despite what we mentioned at the time, the `Async` methods do not actually create a new thread, but call `ForkJoinPool.commonPool()` to get access to a pre-existing executor. This particular executor is a convenience, in case you don't need to adjust the thread pool's settings.

We won't go into detail on exactly how to use `ForkJoinPool` or `ScheduledThreadPoolExecutor`. Briefly, though:

- `ScheduledThreadPoolExecutor` lets you *schedule* tasks to run periodically, or after a certain delay.
- `ForkJoinPool` lets you run tasks that recursively divide themselves into smaller tasks to be run in parallel, and then recombine their results. It may not always be possible to arrange for a task to do this, but where it is possible, it can help make more efficient use of computing resources. "Forking"[l] refers to breaking up a task into smaller, independent tasks, and "joining" to the combining of their eventual results. Fork-join pools also employ a blocking queue for each thread, to deliver tasks to that thread, and idle threads are able to "steal" tasks from other threads' queues.

  The static method `ForkJoinPool.commonPool()` returns a global `ForkJoinPool` instance. You can easily use it (or any other `ForkJoinPool` instance) as a "normal" thread pool too without having to do any forking or joining.

4.2 ThreadPoolExecutor

If you do need to define your own thread pool, `ThreadPoolExecutor` is the traditional choice. Many of the factory methods in `Executors` are likely shortcuts for making `ThreadPoolExecutor` objects.

As shown in Figure 5 and Listing 21, a `ThreadPoolExecutor` has:

- A "core size". It will always keep this many threads running, unless it's starting up or shutting down.
- A "maximum size". If this is larger than the core size, then the thread pool will sometimes grow above the core size when processing a lot of tasks at once, but never larger than the maximum.
- A "keep-alive time". In quieter times, when there are more threads than tasks, some threads will sit idle. If the thread pool is currently larger than the core size, then idle threads will be stopped after a certain period of time – the keep-alive time.
- A blocking queue[m]. The thread that calls `submit()` acts as a producer of tasks. The threads within the pool are acting as consumers of tasks. So, to transfer tasks from one to the other, `ThreadPoolExecutor` uses a blocking queue (`BlockingQueue<Runnable>`), and makes you specify what type it should be.

When you submit a task, `ThreadPoolExecutor` works as follows:

- If it hasn't yet created all the core threads, it will start a new one to run the new task;
- Otherwise, if there are one or more idle threads, one of them will immediately start the new task (since the queue will be empty);
- Otherwise, if all threads are occupied but there's space in the queue, it will add the task there temporarily, and one of the threads will get to it later;

---

[l] Not to be confused with the Unix `fork()` function that creates a child process.

[m] `ForkJoinPool` has a blocking queue for each thread, while `ThreadPoolExecutor` has only one blocking queue for the whole pool.

- Otherwise, if all threads are occupied and the queue is full, but the thread pool is less than the maximum size, it will create a new thread to run the task;
- Otherwise, it will "reject" the task, although this still isn't the end of the matter. You can supply a `RejectedExecutionHandler` (as in Listing 25) to handle rejected tasks, and possibly still find a way to run them. There are several pre-defined ones, including:
  - `ThreadPoolExecutor.AbortPolicy` will, when a task is rejected, make `submit()` throw `RejectedExecutionException`. In this case, the task doesn't run.
  - `ThreadPoolExecutor.CallerRunsPolicy` will, when a task is rejected, make `submit()` run the task in the *current* thread instead of in the thread pool. In other words, the task will run synchronously, not asynchronously.

    You *should not* use `CallerRunsPolicy` if you're calling `submit()` from the GUI thread. However, it could be useful in other situations, where it can naturally slow down the rate at which new tasks are being generated.

  There are other pre-defined options too, and you can create your own by implementing the `RejectedExecutionException` interface.

Listing 25: Supplying a `RejectedExecutionHandler`.

```
ThreadPoolExecutor exec = new ThreadPoolExecutor(...);
exec.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());

// OR, make your own RejectedExecutionHandler:
exec.setRejectedExecutionHandler((runnable, executor) ->
{
    ... // Take whatever action you like here.
});
```

4.3 How Many Threads Can/Should I Have?

This is an important and complicated question, now that we have the means to enforce a limit.

First, let's bust one simple myth: the number of "CPU threads", or CPU cores, has nothing to do with the number of software threads that you can have. It's just one factor in how many you *should* have.

Note 3:
What are CPU threads and cores?

CPU threads and cores are hardware constructs. The number of them is fixed for any given CPU, with (typically) two CPU threads per core. At the time of writing, modern retail/consumer CPUs typically have 4–32 cores, and thus 8–64 CPU threads.

A core contains a complete copy of all the hardware logic needed to execute machine instructions (compiled code). Each core can only do one thing at a time. However, often the code running on a core periodically causes it to wait for the retrieval of data. This presents an opportunity to have the core do something else in the mean time.

So, CPUs pretend that they actually have two "logical cores" (CPU threads) for each physical core. This gives each core something to do during any wait periods, so it doesn't sit unused[a]. Two CPU threads sharing the same core *do not* work as fast as two separate cores, but they can be faster than performing tasks sequentially.

However, the number of CPU threads does not limit the number of *software* threads. If there are more software threads than CPU threads, the operating system will work out how to schedule the CPU's resources to run all the software threads. Each CPU thread will just flip back and forth between different software threads, as directed by the OS.

[a] The CPU manufacturer Intel calls this "Hyper-Threading".

The actual number of (software) threads that you *can* have is well into the thousands (perhaps hundreds of thousands). The operating system may impose an arbitrary limit, and the reason it can't be unlimited comes down to memory. Each thread needs its own growable stack space.

If a task is computationally-intensive, where CPU power is the limiting factor, we say it's **CPU-bound**. On the other hand, if a task is constantly transferring data across a network, or constantly reading/writing files, we say it's **I/O-bound**. Short tasks that only run periodically are neither CPU-bound nor I/O-bound. Non-CPU-bound tasks do also need to use the CPU, of course, but their demands of it are (by definition) very small.

For *anything*-bound tasks, it makes sense to run them in parallel, if the nature of the tasks allows it (i.e., if one doesn't depend on the result of the other). You can make better use of the available resources that way, so the tasks will collectively finish faster. However, threads come with a cost, and resources are always limited, so there's a limit to the number of parallel tasks you can usefully have:

- For CPU-bound tasks, it's simple. There's no benefit from running more tasks in parallel than you have actual CPU (hardware) threads. You can't use more CPU resources if there aren't any left. (You can try, and your tasks *will* run, but not any faster.)

  In Java (as we saw before), `Runtime.getRuntime().availableProcessors()`[n] will report the number of CPU threads. You can use this as the maximum thread pool size, for instance.

- For storage-I/O-bound tasks (reading/writing files), you *may* only want one task at a time. There's an overall upper limit to the data rate for any given storage device. One task could easily occupy the entire data rate for itself if it's reading/writing a single large file. However, if you're reading/writing many smaller files, there could be other effects introduced by caching and the filesystem.

- For network-I/O-bound tasks (sending/receiving network data), the limits are harder to pin down. Running these tasks in parallel still makes sense, because any given transfer of data to/from another machine, across the internet, probably won't make maximum use of the local bandwidth (your own direct connection to the internet). The data rate for any given

[n] The name is slightly misleading, and dates from a time when CPUs had a single core/thread, though you could still have multiple CPUs.

transmission is limited by the slowest link in the route, and that can vary from time to time. Moreover, connecting to different machines across the internet may entail a different set of network links – a somewhat different set of resources to be utilised.

- For other tasks, it depends! The GUI works fine with all its tasks operating sequentially (i.e., nothing in parallel at all). But you may be using other external resources too: running other programs and waiting for their results, performing console input, running calculations on the GPU (graphics processing unit), etc. You may want to do these things in parallel.

If you have CPU-bound *and* I/O-bound tasks, it probably makes sense to assign them to two different thread pools. You'll want to have different limits for each. Remember that, even if you have as many parallel CPU-bound tasks as there are CPU threads, that *doesn't* prevent other threads from running.

There isn't always a simple theoretical answer, though. What if you have tasks performing some combination of network communication, CPU computations, and simply waiting? It's not going to be easy to analyse this on paper.

The surest way to see how many threads you need is to *measure the actual performance!* Get some solid empirical data. Try out different thread pool sizes, on a range of different machines, and measure the time your application takes to do something important, and the amount of memory and network bandwidth it uses. In fact, measure the effect of varying all the different things you can configure: the core size, the maximum size, the keep-alive time, and the queue type/size.

5 Further Reading

This document has tried to address some of the key tools for designing multithreaded systems, but it's an expansive topic, not least because there are different multithreading structures (or variations of them) in different languages.

- Brian Goetz, et al. (2006), *Java Concurrency in Practice*, Addison-Wesley.

  This is now somewhat outdated, being published prior to the release of Java 6 (let alone Java 8, discussed here). However, given Java's multithreading features at the time, it is extremely in-depth. In particular, it is still a good resource for blocking queues and the use of shared state.

- Brian Goetz (2003), *Java theory and practice: Concurrent collection classes*, http://www.ibm.com/developerworks/java/library/j-jtp07233/index.html
- Ted Neward (2010), *5 things you didn't know about ...java.util.concurrent, Part 1*, http://www.ibm.com/developerworks/java/library/j-5things4/index.html.
- Ted Neward (2010), *5 things you didn't know about ...java.util.concurrent, Part 2*, http://www.ibm.com/developerworks/java/library/j-5things5/index.html.

References

[1] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. SIP-14 – futures and promises. https://docs.scala-lang.org/sips/completed/futures-promises.html, 1 2012. Accessed: 2019-06-30. [**See section 3.1.**]

[2] Graham Hamilton. Multithreaded toolkits: A failed dream? https://community.oracle.com/blogs/kgh/2004/10/19/multithreaded-toolkits-failed-

dream, 2004. Accessed: 2017-07-30. [**See section 2.**]

[3] Roger Hughes. The producer consumer pattern. https://dzone.com/articles/producer-consumer-pattern, 2013. Accessed: 2019-07-05. [**See section 1.**]

[4] Kisalaya Prasad, Avanti Patil, and Heather Miller. Futures and promises. In *Programming Models for Distributed Computing*. 2017. Accessed: 2019-06-30. [**See section 3.1.**]

[5] Brian Storti. The actor model in 10 minutes. https://www.brianstorti.com/the-actor-model/, 2015. Accessed: 2019-07-05. [**See section 2.3.**]