

# Asynchrony

## [Objectives](#)

## [Introductory example](#)

## [What is asynchronous code?](#)

## [Why is asynchrony useful?](#)

## [How does one write asynchronous code?](#)

### [Thread Pools](#)

### [Fork/join](#)

### [Callbacks](#)

#### [Lambdas / closures](#)

#### [Functional interfaces / anonymous instances](#)

#### [Callback hell](#)

### [Futures](#)

#### [Producing Futures](#)

#### [Consuming Futures](#)

#### [Chaining](#)

#### [What the thread?](#)

#### [Common usage errors with Futures](#)

##### [About Future.get, Future.isDone and Future.isCancelled](#)

##### [Storing data into synchronous structures](#)

##### [Callback hell: Electric Boogaloo](#)

### [Coroutines](#)

### [Examples](#)

## [SOA / microservices / edge computing](#)

### [Example: refactoring a monolithic application into microservices](#)

## [How to implement asynchrony](#)

### [Example: Building Futures using Thread](#)

## [Synchronous vs asynchronous: A case study](#)

## Objectives

Modern software is rarely self-contained: it communicates with its environment, such as by reading files or sending data via the Internet. The traditional way to implement such features is to “block” the current thread of execution until the operation finishes. But this provides for a poor user experience: the program appears unresponsive for an amount of time outside the control of the developer. Furthermore, blocking operations are inefficient at scale: leaving threads unused because they are blocked waiting for some operation to complete is poor resource management. The answer to these problems is asynchrony: architecting software such that it can perform other tasks while waiting for responses. As a software engineer, you will frequently use asynchronous methods, and thus you will need to understand how they differ from synchronous methods. You will also occasionally have to implement your own asynchronous operations on top of code that implements asynchrony in a different way, such as by explicitly using threads.

In this module, you will learn:

- How to use asynchronous operations
- How to implement your own asynchronous operations

We are assuming here that you have understood very well the material taught in EPFL's [CS-206 course "Parallelism and Concurrency"](#). Before reading this week's material, please review particularly the three lectures on Concurrency in CS-206. Without a good understanding of that required material, it will be very difficult to understand the material of this week.

After having reviewed that material, we encourage you to re-read the material on [Reactive programming](#) in the "Good Code" lecture notes.

## Introductory example

Below is the source code of a client-server system communicating using [socket.io](#) in NodeJS. The server code is on the left, client on the right:


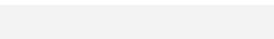
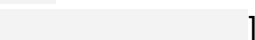

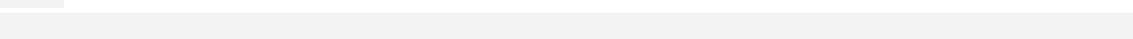
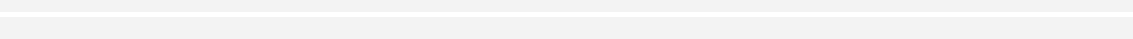

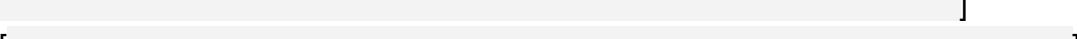
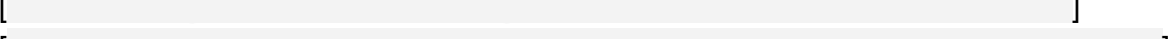
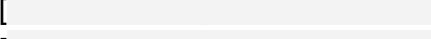
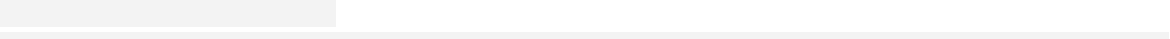
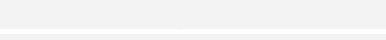
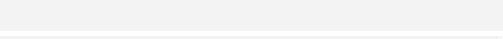

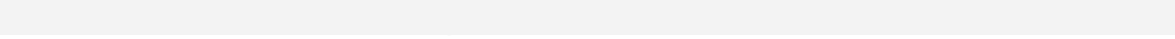

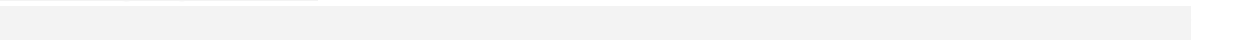
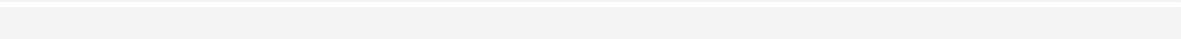
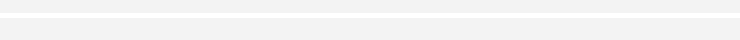


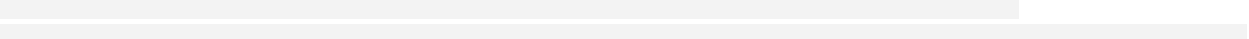
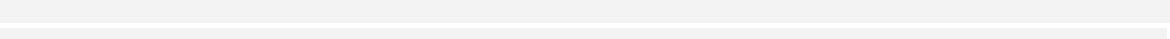
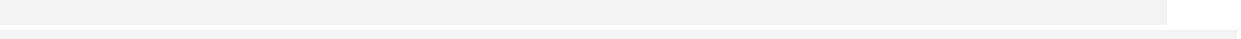
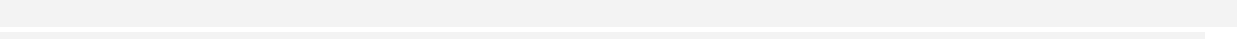

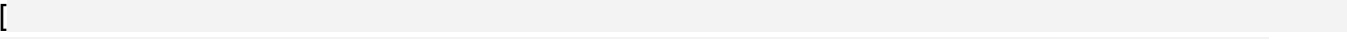
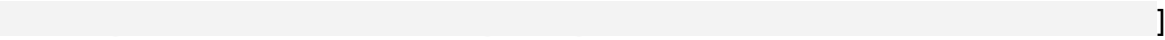
```
1. const express = require('express')()
2. const http = require('http')
   .createServer(express)
3. const server = require('socket.io')(http)
4.
5. server.on('connection', (socket) => {
6.   console.log('new client connected')
7.
8.   const clock = setInterval(() => {
9.     socket.emit('time', Date.now())
10.  }, 3000)
11.
12.  socket.on('disconnect', () => {
13.    console.log('client disconnected')
14.    clearInterval(clock)
15.  })
16.
17.  socket.on('ping', () => {
18.    console.log('ping')
19.    socket.emit('pong')
20.  })
21.})
22.
23.http.listen(3000, () => {
24.  console.log('server started')
25.})
```

```
1. const io =
   require('socket.io-client')
2. const client =
   io('http://localhost:3000')
3.
4. let counter = 0
5.
6. console.log('client started')
7. client.emit('ping')
8.
9. client.on('pong', () => {
10.  console.log('pong')
11.})
12.
13.client.on('time', (time) => {
14.  console.log('Server time: ' +
15.    new Date(time))
16.  counter = counter + 1
17.  if (counter === 5) {
18.    console.log('bye')
19.    client.disconnect()
20.  }
21.})
```

Can you answer to the following questions:

- What does the server print and in which order?
- What does the client print and in which order?
- When does the client program end?
- When does the server program end?
- What is the program flow, i.e. the order in which each function call or closure is executed?
- Can you give a guaranteed lower bound on the execution time of the client program?

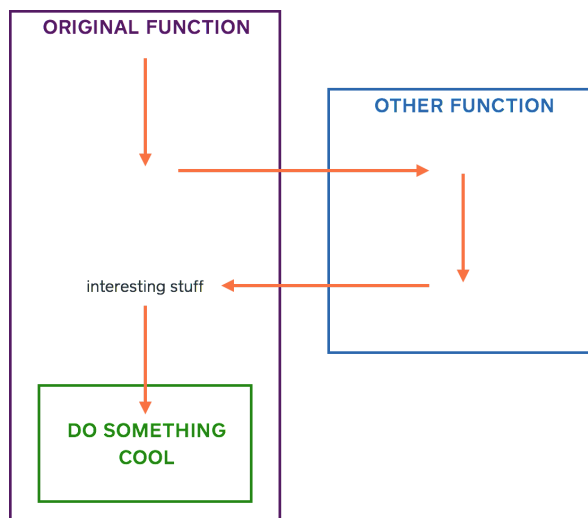
**Answers:** (highlight the text between each pair of brackets)

- [     ]
- [          ]
- [  ]
- [  ]
- [ 
  1.  
  2.  
  3.   
  4.        
  5.  
  6. 
  7.        ]
- [    ]

As you can see, we have changed our programming paradigm: the code does not run in the order in which it is written, rather we express it in such a way that its execution depends on so-called *events*. This lecture will cover why such a property is desirable, how to program in an *event-driven* way, and peek under the hood of asynchronous engines to get a general overview of how the event-driven paradigm can be implemented.

# What is asynchronous code?

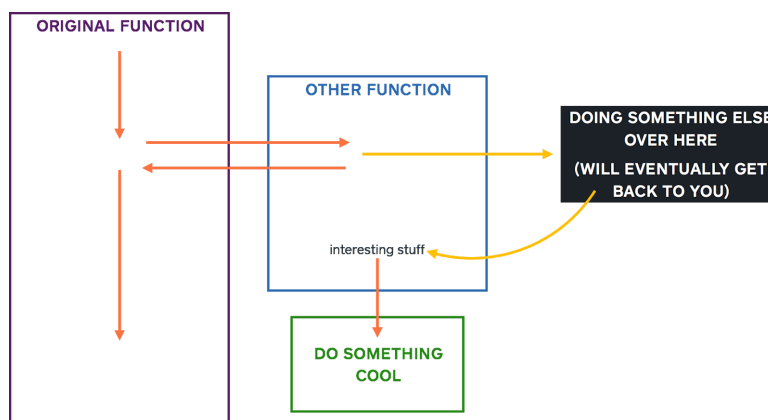
Up until now, the code we have seen throughout the course was **synchronous**: instructions ran in a sequential order with only one event happening at a time. This means that the program's execution is halted while waiting for a method call to return (e.g. a network request).



This way of dealing with “secondary” events can get very tedious for long-running tasks. In the context of a weather app, waiting for the weather service to serve the request implies blocking the main UI thread therefore rendering the whole application unresponsive. In addition to being an unpleasant experience, this isn't a good way of using computing resources: why halt and wait while other instructions can be executed?

Asynchronous code execution allows some code to execute independently from the “main” execution thread. This code is referred to as **asynchronous code**. It allows multiple actions to happen at the same time by separating the main program's execution from independent events like I/O operations. For example, when a request to a server is made, the request is sent and the rest of the program keeps running. In a synchronous fashion, the program would've waited until the server's reply. At some point, the server will answer the request: as the program went on, there would be no way to get the result back to the program.

Instead, an extra parameter is provided to the asynchronous function: it specifies to the function what to do with the result once it completes. This parameter, a function, is referred to as a **callback**: as its name suggests, it “calls back” the calling program with the asynchronous result. This callback parameter is **registered** (i.e. stored) to the result event at runtime, and only executed later, when it actually occurs. The callback registration call itself usually **immediately returns**. The program will obtain the value later inside the **callback body only**.



In an asynchronous regime with multiple independent asynchronous tasks, the execution order is not defined: tasks can run simultaneously, sequentially, or in an interleaved order, each on a separate thread. That behavior

is called **concurrent**. Note that, often, the execution order of such tasks matters. Most modern applications operate in a concurrent fashion. Let's reconsider the weather app: the app's main thread is the UI thread on which the app's layout is drawn and displayed. Asynchronous calls to the weather service and the map service (to display weather info on it) will all be concurrent to the main thread.

A key point to remember when dealing with asynchrony is that we never **want to block while waiting for a computation to finish**. Usually, mixing asynchrony with synchronous constructs is a recipe for disaster, opening the door to undefined behaviors or unnecessary blocking which hurts performance. Once some part a program must deal with asynchrony (such as imposed by a SDK), then callbacks must be propagated from the data entry point to its use site.

## Why is asynchrony useful?

Why would you care about writing asynchronous code, when synchronous code would be much easier to deal with? The main advantages of asynchronous statements is that they will make your software much more efficient and more responsive.

To understand these benefits, let's consider the synchronous and asynchronous version of a function that downloads an image from a web server based on a given ID. We will first consider the synchronous version, *syncGetImage(id)*, that sends a request to the web server and returns an *Image* when the download is completed. Now let's say that you are developing a social media app. One of the features you want to implement is a page where the user can see all his friends' names alongside with their profile picture. To code this feature, you first retrieve all information about these friends, and store the id of their profile picture in the Array *profilePicturesId*. Now if you want to synchronously download these profile pictures and store them in an Array of Image to be displayed, you will probably write something like this:

```
1. for(int i = 0; i < profilePicturesId.size(); i++){
2.     profilePictures[i] = syncGetImage(profilePicturesId[i])
3. }
4. displayProfilePictures(profilePictures)
```

This code will do the job and you will get all the pictures displayed. However, downloading an image over the network can be a very long process relatively to the execution of our software. For this example, let's assume that it takes 1 second to download an image. In the above code snippet, because *syncGetImage(id)* is a blocking call, one iteration of the for-loop will take approximately 1 second to execute. This means that if a user only has 20 friends (which you can arguably display all at once), the page will take 20 seconds to load ! Meanwhile, the program is blocked, and the user is not able to interact with the page at all. This is of course not efficient at all.

The main flaw of this code is obviously that it doesn't make sense to wait for a profile picture to be downloaded to send the request to download the next one. Why not send the next request regardless of the status of the previous one? This is what asynchrony seeks to solve.

Let's now try to replace this synchronous function with its asynchronous counterpart, namely *asyncGetImage(id)*. Note that this function doesn't return an *Image*, but a *Future<Image>* (if you don't remember what a Future is, have a look at the next section). This means that *asyncGetImage(id)* is non blocking and that the execution of the code will continue once the function has sent its request to the web server. Your code may now look like this :

```
1. for(int i = 0; i < imagesId.length(); i++){
2.     futureProfilePictures[i] = asyncGetImage(profilePicturesId[i])
3. }
```

Now, even though it takes 1 second to download a profile picture, one iteration of this for-loop only takes some milliseconds to complete. **You should however beware that once your for-loop has ended, it is not guaranteed (and it is most likely not the case) that the pictures are all completely downloaded.** What you have are Futures, which means that one picture may be fully downloaded, another one may be currently receiving data, and another one may still be awaiting a response from the server.

However, this means that it now takes 1 second to retrieve all profile pictures in parallel, and for a user with 20 friends, you have effectively reduced the waiting time by 19 seconds.

One question remains: how does one know when the images have downloaded? How do we display the profile pictures afterwards? What happens after the for loop? To answer these questions, we need a paradigm shift: one should not be concerned about the images themselves anymore, but rather the events that surround the availability of the data. In the next section, we will explore language constructs to write and reason about asynchronous programs. **In essence, we do not decide when our code is executed anymore:** rather we provide code snippets that react to specific events (such as a completed download), and let the runtime bind them and execute them at the appropriate time. As a sneak peek, one may want to implement the for-loop like this instead:

```
1. for(int i = 0; i < imagesId.length(); i++){
2.     CompletableFuture<Image> dl = asyncGetImage(profilePicturesId[i])
3.     dl.whenComplete((img, err) -> {
4.         if (err != null) displaySingleProfilePicture(img)
5.     })
6. }
```

## How does one write asynchronous code?

The most basic way to implement asynchrony is to use threads. However, creating and managing the state of threads is a task that tends to require a lot of memory and computation. Creating many threads (e.g., one per request to a server) may actually hurt performance. Let's also remember that the CPU has a fixed amount of threads which is usually way less than the number of threads created by the program. Let's look at a few ways that Java can handle this problem:

### Thread Pools

The [thread pool pattern](#) reduces the load of creating new threads for each task by managing a pool of threads that can process incoming tasks. A producer can submit a large number of tasks that will be enqueued and processed as soon as a thread in the pool is available. This reduces the number of threads to a degree that can actually be handled by the OS and the CPU. This is particularly useful in applications such as TCP/IP servers.

### Fork/join

In the case where we have to deal with a large number of small tasks that depend on each other (e.g., recursive tasks), the [fork/join framework](#) allows the program to create many (more than the number of threads on the CPU) lightweight threads that perform small tasks. This is known as the divide-and-conquer approach in algorithms, where the divide step is handled by “fork” which creates subtasks and the combine step is handled by “join” which collects the subresults to produce the final value.

In practice, these patterns are often abstracted away in higher level frameworks and the programmer does not have to worry about how threads are handled. But it is useful to know about them in the case where a custom

solution to a concurrency problem is required. In barebone Java, we will most often use the [Executor abstraction](#), which provides constructs to decide on which thread a computation should be run: often we will use such an instance with [ForkJoinPool.commonPool\(\)](#), a basic thread pool that scales with the number of CPU cores of the underlying physical machine. In many frameworks and libraries, asynchronous operations will already make use of threads: make sure to read the relevant documentation to correctly schedule your own computations and implementations.

## Callbacks

In asynchronous programming models, long-running computations are conceptually detached from the main program flow and executed in “the background”, and the language must provide some way for these diverging paths to either yield their result back to the main logic, or continue with some other calculation. Hence some language syntax constructs are required: a natural way to represent the generic continuations of these computations is through functions. Indeed, we usually want to perform some action when the result is available on its value. It could in turn return a new value (for instance, filtering some search results after a query over the network) or perform some side-effects (such as writing to disk). Such functions are named “callbacks” which will be “called back” when the event to which they have been registered occurs. One may want to register multiple callbacks or perform very different responses to the same event, thus functions provide a reusable, generic abstraction over such actions.

## Lambdas / closures

Functional programming (and languages that support functions as first-class citizens in general) are well-suited for this usage: one can pass an anonymous function (lambda) as an argument.

*Reminder: Lambdas are closures if they allow capturing the current environment in scope: for instance, in Scala, a function parameter is a closure because it can access the class members in which it is defined, while in Python such definitions are not visible to the body of a lambda construct.*

User events are a common use case of asynchronous callbacks: the OS will notify the UI thread of an application through its runtime (browser, ART, ...) when a hardware event occurs, such as a mouse move triggering a hardware interrupt, or a gamepad joystick being polled with a new value. It would be impossible to statically place the code that handles this specific user interaction ahead of time such that the instruction pointer lands on it at the exact time the event is raised: however, what we can do is register a function in the runtime such that it gets executed whenever the event occurs.

An example with Android in Java: inside an Activity screen (which lives on the main UI thread), we register a click listener callback on some “Go” button: the argument is a function which takes as input the view being clicked (i.e. the button) and its execution will change the current screen being displayed to the GreetingActivity.

```
1. Button goButton = findViewById(R.id.mainGoButton);
2. goButton.setOnClickListener(view -> {
3.     Intent greeting = new Intent(MainActivity.this, GreetingActivity.class);
4.     startActivity(greeting);
5. });
```

Note that when line 2 is reached, the callback parameter (`view -> { ... }`) is not executed immediately, instead it is **only saved** into the runtime and the `setOnClickListener` call immediately returns afterwards. The callback is executed later, when the OS actually detects a finger click on the corresponding button. The OS will pass the event to the app runtime, which in turn will lookup the memory location in which the callback was

registered, point to its code and execute it. In this scenario (and in most UI applications), the callback is executed on the main UI thread again, however callbacks are not necessarily always run on the same thread on which they were defined.

I/O operations such as network requests and disk / database persistence are other good candidates for asynchronous programming, as they usually are lengthy (compared to CPU speed), unpredictable (as they depend on physical mediums which may fail) and independent from the execution of the main program (for instance, one generally doesn't *need* to wait for a file to be written).

Below is an example of an asynchronous file read I/O using NodeJS. `readFile` takes a file path as first parameter, and a callback which is executed when the OS has finished copying the file into memory or when a read error occurs (such as a file not found).

```
1. const fs = require('fs')
2. fs.readFile('/etc/passwd', (err, data) => {
3.     if (err) {
4.         console.log("uh oh... an error occurred " + err)
5.     } else {
6.         console.log(data.toString())
7.     }
8. })
9. console.log("When do I print?")
```

Again, line 2 returns immediately after registering the callback. *Notice the last `console.log` statement on line 9, when does it print? In which part of the code can you have guarantees about the content of the file?*

```
[
  // ...
]
```

## Functional interfaces / anonymous instances

Not all languages support functions as values: for instance Java before version 8 did not have any of the functional goodness such as lambda expressions and stream APIs. Some other limitations in the language may also exist: for instance, even Java  $\geq 8$  only supports the `Function<T,R>` and `BiFunction<S,T,R>` types for 1 and 2 lambda arguments respectively, but not for higher argument counts; function names are lost between the supplier and consumer of a lambda parameter, and clearly-defined sets of grouped functions with precise semantics are hard to express using lambdas only.

Fortunately, functional interfaces and anonymous instances solve all the above problems by allowing the programmer to define an interface with a fixed set of named abstract methods to implement. In fact, Java lambda expressions desugar into anonymous instances of functional interfaces.

The above Android example would be written as follows before Java 8 (and the modern implementation also desugars to):

```
1. Button goButton = findViewById(R.id.mainGoButton);
2. goButton.setOnClickListener(new View.OnClickListener() {
```



```

3.     @Override
4.     public void onClick(View v) {
5.         Intent greeting = new Intent(MainActivity.this,
6.             GreetingActivity.class);
7.         startActivity(greeting);
8.     }
9. });

```

This code is strictly equivalent to the code shown before in the “Lambdas / closures” first example, we simply explicitly expanded the lambda into an anonymous instance of `View.OnClickListener`, which is a functional interface with a single abstract `onClick` method which takes the same `View` argument as the previous lambda, and must be overridden by the anonymous declaration. The source code of the interface in the Android library is

```

1. /**
2.  * Interface definition for a callback to be invoked when a view is clicked.
3.  */
4. public interface OnClickListener {
5.     /**
6.      * Called when a view has been clicked.
7.      *
8.      * @param v The view that was clicked.
9.      */
10.    void onClick(View v);
11. }

```

Anonymous instances of interfaces are also useful when more than one callback is required: one could pass many different callback parameters (one for each execution outcome), but an interface allows the programmer to group them together semantically. For instance, the Cronet network library in Java / Kotlin does exactly that: a request to an URL may trigger up to 5 events while processing its response:

```

URLRequest request = cronetEngine.newUrlRequestBuilder(
    "https://www.example.com",
    new URLRequest.Callback() {
        @Override
        public void onRedirectReceived(URLRequest request, URLResponseInfo info,
            String newLocationUrl) {
            request.followRedirect();
        }

        @Override
        public void onResponseStarted(URLRequest request, URLResponseInfo info) {
            request.read(ByteBuffer.allocateDirect(102400));
        }

        @Override
        public void onReadCompleted(URLRequest request, URLResponseInfo info,
            ByteBuffer byteBuffer) {
            request.read(ByteBuffer.allocateDirect(102400));
        }

        @Override
        public void onSuccess(URLRequest request, URLResponseInfo info) {
            Log.i("INFO", "request succeeded");
        }
    }
)

```

```

    }

    @Override
    public void onFailed(UrlRequest request, UrlResponseInfo info,
        CronetException error) {
        Log.wtf("ERROR", "request failed");
    }
},
Executors.newSingleThreadExecutor()
).build();
request.start();

```

When designing your own asynchronous systems in Java, think about using functional interfaces and anonymous instances if you need to deal with more than 2 lambda parameters, or more than 2 event callbacks. If you need to handle a success / failure scenario, you may also want to use a Future or Promise (read below).

## Callback hell

Since it is very common for callbacks to take the result of the previous callbacks as argument, poorly written asynchronous code often ends up in a pyramid shape, like the following snippet:

```

1. User.authenticate(credentials, (err, authResult) => {
2.     if (err) {
3.         console.log("Login error")
4.         throw err
5.     } else if (authResult == AuthResult.SUCCESS) {
6.         contentDatabase.getPosts(authResult.user, (err, posts) => {
7.             if (err) {
8.                 console.log("Content fetch error")
9.                 throw err
10.            }
11.            const postsLayout = findView("posts_container")
12.            posts.forEach(post => {
13.                const postView = createPostView(post)
14.                postsLayout.add(postView)
15.                postView.setClickListener(view => {
16.                    localStorage.save(post, (err, storageKey) => {
17.                        if (err) {
18.                            console.log("Persistence error")
19.                            throw err
20.                        }
21.                        User.addFavorite(post.id, (err, res) => {
22.                            if (err) {
23.                                console.log("Network error")
24.                                throw err
25.                            }
26.                        })
27.                    })
28.                })
29.            })
30.        })
31.    }
32. })

```

In this example, the user must first authenticate. If the login is successful, the content (some user posts) is then fetched from a database through the network. Finally, each post is displayed and a click listener is registered, which once triggered will save the post to local storage and add the post to the user's favorites. Note that each

operation is asynchronous, so the delay between each callback is unknown. Also, each layer deals with I/O, which may fail and provide an error to its respective callback instead.

As you can see, the issue lies in the fact that these asynchronous computations are interdependent. A first solution to flatten the code is to name each callback and hoist them all at the top of the file:

```
1. function authenticate() {
2.   User.authenticate(credentials, (err, authResult) => {
3.     if (err) {
4.       console.log("Login error")
5.       throw err
6.     } else if (authResult == AuthResult.SUCCESS) {
7.       fetchContent(authResult.user)
8.     }
9.   })
10. }
11.
12. function fetchContent(user) {
13.   contentDatabase.getPosts(user, (err, posts) => {
14.     if (err) {
15.       console.log("Content fetch error")
16.       throw err
17.     }
18.     displayPosts(posts)
19.   })
20. }
21.
22. function displayPosts(posts) {
23.   const postsLayout = findView("posts_container")
24.   posts.forEach(post => {
25.     const postView = createPostView(post)
26.     postsLayout.add(postView)
27.     postView.setOnClickListener(view => {
28.       savePost(post)
29.     })
30.   })
31. }
32.
33. function savePost(post) {
34.   localStorage.save(post, (err, storageKey) => {
35.     if (err) {
36.       console.log("Persistence error")
37.       throw err
38.     }
39.     addFavorite(post.id)
40.   })
41. }
42.
43. function addFavorite(postId) {
44.   User.addFavorite(postId, (err, res) => {
45.     if (err) {
46.       console.log("Network error")
47.       throw err
48.     }
49.   })
50. }
51.
52. authenticate()
```

Notice however that with this syntax, we have no way to enforce the temporal linearity of each callback in the code structure (we could list the callbacks in any order). We also lost the semantics of the complete chain of

operations; we could use extremely long function names instead, which wouldn't be convenient either, like `authenticateThenFetchContentThenDisplayPostsAndSaveFavoriteOnPostViewClick`.

In the next chapter, we explore another solution to callback hell, which consists in wrapping asynchronous computations and their associated callbacks into objects called Futures.

## Futures

A Future (also called Promise, Task, Deferred or Delay in other languages and frameworks) is a proxy object for a value that will be computed eventually or which may fail with an error. It wraps asynchronous operations of any kind and provides a unified interface to deal with them (as opposed to scattered or nested callbacks as seen above). A Future is a fancy object that handles passing these callbacks around and calling them at the right time, without the need to explicitly pass the callbacks as arguments at every call of the chain.

In this chapter, we will cover the usage and implementation of the [Future / Promise construct](#), which corresponds to [Scala's Futures and Promises](#), [JavaScript's Promise](#) and [Java's CompletableFuture](#). Note that [Java's Future interface](#) **does not provide asynchronous capabilities out-of-the-box**, as such we will always use **CompletableFuture** in Java instead. We will use the name Future for brevity purposes.

A Future can be in any of 3 states:

- Pending: the computation is being performed, and the result is not available yet
- Fulfilled: the computation has finished successfully, and a result value is available
- Rejected: the computation failed (e.g. I/O error), and a rejection reason is provided

There are 2 sides when dealing with Futures: the producer side which creates the Future at the computation source (which you use when creating a library, API or data source in your application such as a database connector) and the consumer side that uses the Future, i.e the client code that depends on the termination of the computation (such as some UI awaiting the data to be displayed).

## Producing Futures

To create a Future, we first need to examine the computation we wish to wrap: does it already run in a separate thread, or should it be put into one?

In the former case (for instance when dealing with an existing library which already handles threads under the hood), one simply needs to instantiate a Future object and fulfill it when the value is available. For instance, let's take back the example of the authentication library, and assume that it handles connections on another thread by itself. Transforming a callback-based asynchronous operation in Java into a CompletableFuture would look like this:

```
1. public CompletableFuture<AuthResult> authenticate(Credentials credentials) {
2.     CompletableFuture<AuthResult> future = new CompletableFuture();
3.     User.authenticate(credentials) (err, authResult) -> {
4.         if (err != null) {
5.             future.completeExceptionally(err);
6.         } else {
7.             future.complete(authResult);
8.         }
9.     });
10.    return future;
11.}
```

This takes care of calling the already asynchronous `User.authenticate` method, and fulfilling the `CompletableFuture` with the authentication result if the call succeeds using `future.complete(result)`, or rejecting it with the exception parameter of the callback if it exists otherwise with `future.completeExceptionally(exception)`. Note that we did not handle nor throw the error, and that we did not process the successful result either: we simply wrapped them in the `CompletableFuture` once they were available.

In other cases, the computation is not intrinsically asynchronous (for instance, a huge computation function that does not execute on another thread by itself). Then one can use the `supplyAsync` factory which takes a `Supplier` functional interface instance as argument: the factory will create a new `Future` which computation will be run asynchronously on the [ForkJoinPool.commonPool\(\)](#) thread pool (or on the second `Executor` parameter if provided):

```
1. public int bigComputation() {
2.     // ... some long-running computation that is not threaded
3. }
4.
5. public CompletableFuture<Integer> bigComputationAsync() {
6.     return CompletableFuture.supplyAsync(() -> bigComputation())
7. }
```

The `bigComputationAsync` method will thus execute `bigComputation` on a separate thread, and the result will be available through the returned `CompletableFuture`. `CompletableFuture` thus provides an easy way for programmers to add efficient concurrency and asynchrony to their program.

## Consuming Futures

From the consumer side, 2 families of functions are used to handle computation success and error respectively. The `then` functions allow consumers to register a callback applied on the successful value when the `Future` completes. The `exceptionally` (Java) / `catch` (JS) methods are used to handle the provided error if the computation fails, again by registering a callback which is executed only when a failure occurs. Java also provides `handle` which takes both arguments in a mutually exclusive fashion, and lets the programmer deal with both cases simultaneously.

For instance, one may handle the result of the previous authentication as follows:

```
1. authenticate(credentials)
2.     .thenApply(authResult -> {
3.         if (authResult == AuthResult.SUCCESS) {
4.             System.out.println("yay I'm logged in");
5.         }
6.         return authResult;
7.     })
8.     .exceptionally(throwable -> {
9.         /* handle failure ... */
10.    })
```

## Chaining

The strength of Futures lies in the return value of their handlers (then, exceptionally, catch, handle, ...) which **always return another Future**, enabling powerful composition of dependent asynchronous operations.

To declaratively describe a chain of dependent computations, one simply keeps calling handlers on the returned futures: each callback will be executed in a cascading fashion, where each previous result resolves the next future in the callback chain: data flows asynchronously from one handler to the next. In the following example, we assume that the functions invoked in each callback return subsequent futures, which we can then chain with `thenCompose`:

```
1. authenticate()
2.   .thenCompose(authResult -> fetchContent(authResult.user))
3.   .thenCompose(posts -> filterPosts(posts))
4.   .thenCompose(results -> displayPosts(results))
5.   .thenApply(posts -> writeToDisk(posts))
6.   .exceptionally(reason -> {
7.       /* handle failure ... */
8.   })
```

Error handling is also greatly simplified: if any of the `CompletableFuture` in a chain fails, then the associated `Throwable` cascades down until it reaches an error handler, as described by the interface documentation:

*In all other cases, if a stage's computation terminates abruptly with an (unchecked) exception or error, then all dependent stages requiring its completion complete exceptionally as well, with a [CompletionException](#) holding the exception as its cause.*

Thus one can provide either a single error handler at the end of the chain, or more granular checks with additional error handlers in-between compositions in the chain:

```
1. futureA
2.   .handle((res, err) -> /* handle error of future A if err != null */)
3.   .thenCompose(a -> createFutureB(a))
4.   .handle((res, err) -> /* handle error of future B if err != null */)
5.   .thenRun(() -> System.out.println("end"));
```

Many composition methods are available: [see the Javadoc of CompletableFuture](#) through inherited methods of `CompletionStage`. `thenApply` is used to transform the value into another type, `thenCompose` is used to invoke and await another dependent `CompletableFuture`, `thenRun` is used to perform side-effects, `thenCombine` is used to await another independent `CompletableFuture` and then continue with both values simultaneously, ...

`CompletableFuture` also provides helpers to deal with multiple concurrent Futures at once such as `anyOf`, which returns a `CompletableFuture` from a variadic number of `CompletableFuture` parameters and which resolves if any of the argument `CompletableFuture`s resolves, as well as `allOf`, which also returns a `CompletableFuture` from a variadic number of `CompletableFuture` arguments but which resolves once all of them resolve. This allows programmers to declaratively describe any combination of dependencies between multiple Futures.

Let us take the example of a cluster of nodes in which we want to perform a search for a given document identifier. A coordinator node will trigger the search recursively on each of its partition nodes, and we would like to be informed when any of the partitions finds the corresponding document. Of course, since the search will involve

I/O across the network for inter-node communication and disk usage for the search, all operations should happen in an asynchronous fashion to maximize throughput. One example implementation using `CompletableFutures` could be as follows:

```
1. class Node {
2.     ...
3.     public CompletableFuture<Document> searchDocumentById(String id) {
4.         // search implementation which scans the disk asynchronously
5.         ...
6.     }
7. }
8.
9. class Coordinator extends Node {
10.    private Collection<Node> partitions = ... // get the partition nodes somehow
11.    ...
12.    public CompletableFuture<Document> searchDocumentById(String id) {
13.        // map each partition to a search future recursively
14.        CompletableFuture<Document>[] partitionsSearchFutures = partitions
15.            .stream()
16.            .map(p -> p.searchDocumentById(id))
17.            .toArray(CompletableFuture<Document>[]::new)
18.        // combine all search futures using anyOf
19.        return CompletableFuture.anyOf(partitionsSearchFutures)
20.    }
21. }
```

The `CompletableFuture` obtained by invoking `Coordinator.searchDocumentById` will resolve as soon as any single one of the mapped `CompletableFuture` resolves, and the result value will be the same as the particular resolved Future. Note that only the first resolved value will be considered, and `anyOf` will also fail on the first Future that fails, if any (e.g. disk failure).

## What the thread?

If you take a look again at the [list of composition methods on a CompletableFuture](#), you will notice that most of them come in 3 flavors: for instance [thenApply\(Function<? super T,? extends U> fn\)](#), [thenApplyAsync\(Function<? super T,? extends U> fn\)](#) and [thenApplyAsync\(Function<? super T,? extends U> fn, Executor executor\)](#).

Although they provide the same functionality (continuing with a provided function applied to the result value of the Future), their difference lies in the execution context of their callback parameter, more specifically the thread on which they are run.

- The first runs the callback function on the thread executor defined by the underlying `CompletableFuture` object. In the general case, you cannot know where the callback is executed, and it may execute immediately if the result is already available.
- The single-argument method suffixed by `Async` will run the callback function on the environment executor. In general, it will submit the job to the default thread pool [ForkJoinPool.commonPool\(\)](#)
- The two-arguments method suffixed by `Async` will run the callback on the `Executor` provided as a second parameter (for instance a specific thread, or another thread pool).

## Common usage errors with Futures

### About Future.get, Future.isDone and Future.isCancelled

A common mistake for programmers new to Futures is to use the synchronous methods of the class, such as Future.get, Future.isDone or Future.isCancelled. It may be very tempting to force the Future to terminate using get, which blocks the current thread until the result is available, or poll isDone to wait until the condition is true or isCancelled to check for errors.

```
1. CompletableFuture<String> future = CompletableFuture
2.    .supplyAsync(() -> longComputationReturningAString())
3.
4. /* ! Don't do this ! */
5. // blocks the current thread until the result is available
6. String s = future.get();
7.
8. /* ! Don't do this ! */
9. // poll isDone until result is available -> blocks the thread
10. while(!future.isDone());
11.
12. /* ! Don't do this ! */
13. // the future may finish at any time, it could finish later! No guarantees
14. boolean ready = future.isDone();
15.
16. /* ! Don't do this ! */
17. // the future may cancel at any time, it could terminate later! No guarantees
18. boolean error = future.isCancelled();
```

There are many issues with these usages:

- Blocking the thread, which defeats the whole purpose of asynchrony: the whole thread is unresponsive and cannot process other computations in the meantime! For instance, a blocked UI thread would freeze the app.
- Mixing asynchronous constructs with synchronous calls: no guarantees about the availability of the data can be made! You should let the runtime execute your callbacks when the result is actually available.
- Race conditions, if the future is executed on a different thread

In general, you should **never use these synchronous constructs** in actual code: if you do, it probably means that you are either using asynchrony wrong, or that asynchrony doesn't solve your problem. Such methods should only be used in specific tests (for instance to validate an asynchronous result, or avoid a concurrent environment in tests) or when implementing your own asynchronous engine.

Instead, let the CompletableFuture help you: use its asynchronous constructs, which already execute your callbacks at the right time without blocking the thread. You should be using handlers such as thenApply, thenCompose, handle, exceptionally, and provide them with a callback which the runtime will execute appropriately. When creating futures, use supplyAsync if you need to offload the job to another thread.

### Storing data into synchronous structures

A corollary to the above point is to not store asynchronous results into synchronous data structures, for the same reasons as previously stated. In general, your code will simply not work as expected:



```

1. public List<Object> fetchDataFromServer() {
2.     /* ! Don't do this ! */
3.     ArrayList<Object> results = new ArrayList<>();
4.     server.fetch().thenApply(data -> {
5.         /* ! Don't do this ! */
6.         for (Object element : data) {
7.             results.add(element);
8.         }
9.     })
10.    /* ! Don't do this ! */
11.    return results;
12.}

```

The above code *looks like* it runs from top to bottom but **the callback provided to thenApply can be executed at any later time!** In fact, the return statement will nondeterministically run before the callback, in the midst of its execution or afterwards: the resulting array may be empty or partially filled. Instead, you should return the `CompletableFuture` directly and handle it at the usage site.

### Callback hell: Electric Boogaloo

Finally, using `CompletableFuture` doesn't make your code immune to the callback hell: a poor usage of its handlers will lead to the same unreadable result:

```

1. /* ! Don't do this ! */
2. futureA.thenApply(a -> {
3.     getFutureB(a).thenApply(b -> {
4.         getFutureC(b).thenApply(c -> {
5.             getFutureD(c).thenApply(d -> {
6.                 ... // yup, we're deep in callback hell again
7.             })
8.         })
9.     })
10.})

```

The issue in the above code is that each future is handled inside the callback body of the previous handler. However, `CompletableFuture` provides the appropriate `thenCompose` method which takes a callback which must return another `CompletableFuture` which resolves after the previous one, allowing them to be chained together:

```

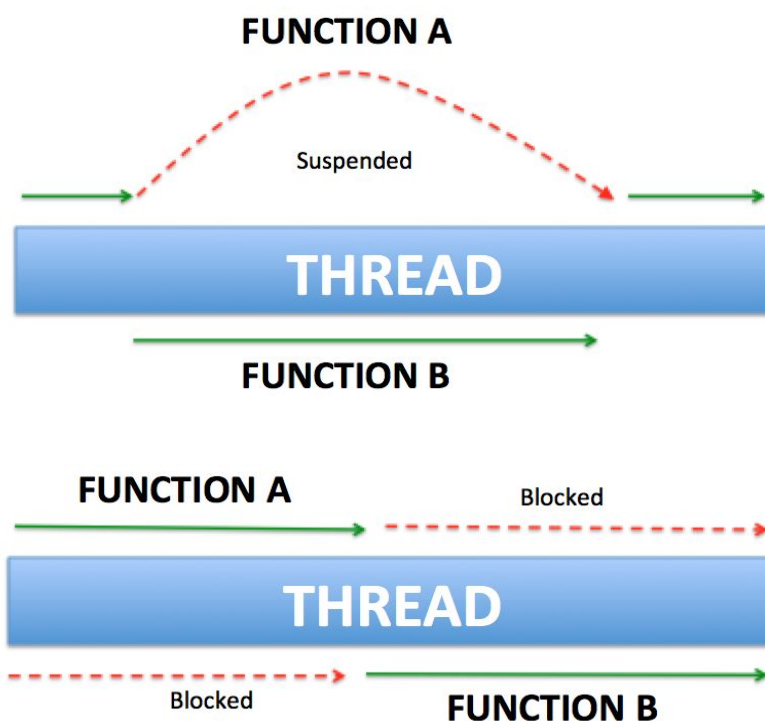
1. futureA
2.     .thenCompose(a -> getFutureB(a))
3.     .thenCompose(b -> getFutureC(b))
4.     .thenCompose(c -> getFutureD(c))
5.     .thenCompose(d -> ...) // yay I'm readable

```

`CompletableFuture` thus provides a robust toolbox and self-contained ecosystem of objects, methods, patterns and constructs to manage asynchrony in Java. Many async-oriented languages will provide similar implementations such as `Future` in Scala and `Promise` in JavaScript.

# Coroutines

**Coroutines** can be seen as lightweight threads (wrapping computations) that are cooperatively multitasked (while threads are preemptively multitasked); this means that coroutines voluntarily yield back control in order to enable **concurrency**. As such, many coroutines can run **without blocking** through *suspension* on one or more threads. Essentially, coroutines are multiplexed onto a set of threads. When a coroutine blocks, such as by calling a blocking system call for a networking operation, the run-time automatically suspends it and schedules another coroutine on the same OS thread for execution.



A suspended coroutine can later resume execution. This is the main difference (and advantage) over threads: switching between coroutines does not involve any (blocking) system call nor context switching. Other advantages are:

- No need for synchronization primitives (such as locks) to guard critical sections;
- No need for support from the OS: they can be implemented entirely in software;
- Very lightweight: since coroutines use much less memory compared to threads, you can create thousands of them.

Notice, however, that coroutines running on the same thread are not executed in parallel; you need multiple threads for parallelism.

Coroutines are used to simplify asynchronous code in many mainstream programming languages: Ruby, Go, Kotlin, Python, Rust, etc.

Since its adoption as an official Android language, Kotlin has risen in popularity, and Kotlin's coroutines have become the de-facto standard for writing asynchronous Android apps. In Kotlin, coroutines are run, by default, on a shared pool of threads. Threads still exist in a program based on coroutines, but one thread can run many coroutines, so the amount of threads used is quite low.

Let's see a simple example:

```

import kotlinx.coroutines.*

fun main(args: Array<String>) {
    println("Start")

    // Start a coroutine
    GlobalScope.launch {
        delay(1000)
        println("Hello")
    }

    // The main threads continues execution
    println("Stop")

    Thread.sleep(2000) // wait for 2 seconds
}

```

The program prints, in order “Start”, “Stop” and finally “Hello”. This is the case because we launch a coroutine in which we delay execution for 1 second by suspending the coroutine itself, and not by blocking the thread: thus, “Stop” is printed immediately after “Start”, and “Hello” is only printed after 1 second, once the coroutine is resumed.

## Examples

Let’s now apply some asynchronous patterns in Java to accelerate some synchronous code. The most basic example would be a task that takes a considerable amount of time to run but does not return a result (saving a file to disk, sending a network packet, ...). Let’s consider the function:

```

public void longRunningTask() {
    // Do some work
    Thread.sleep(5000);
}

```

Calling this function would make the calling thread busy for five seconds. Let’s make this asynchronous using the very primitive Thread class:

```

Thread t = new Thread(() -> {
    longRunningTask();
});
t.start();

```

Now a new thread is created with the task of running this function. The calling thread will no longer be blocked and the task will run asynchronously. However, the interface of Runnable (which represents a task run by a thread) does not allow to return a value. Thus, if our task must return a value, this simple pattern will not work. Let’s consider the second function:

```

public double bigComputation() {
    // Do some work
    Thread.sleep(5000);
    return Math.PI;
}

```

```
}
```

Let's try to use the Thread class to deal with this problem:

```
volatile double result = 0;
Thread t = new Thread(() -> {
    result = bigComputation();
});
t.start();
// Do some other work
t.join(); // Waits for task completion
System.out.println("Value is " + result);
```

Now this is a lot of boilerplate code, and we have to make sure to synchronize the threads before reading the value of the result variable. Let's instead use the CompletableFuture abstraction:

```
CompletableFuture<Double> future = CompletableFuture.supplyAsync(() -> bigComputation());
// Do some other work
double result = future.get(); // Waits for task completion
System.out.println("Value is " + result);
```

Now this is a lot cleaner. We can also clarify the intent of printing the value once it is computed by chaining multiple futures:

```
CompletableFuture<Void> future = CompletableFuture.supplyAsync(() ->
bigComputation()).thenAccept(r -> System.out.println("Value is " + r));
// Do some other work
future.get(); // Waits for task completion.
```

Be wary of the `future.get()` method. It is used to synchronize the asynchronous task with the current thread (i.e. it will block the current thread until the future completion) which can, if used incorrectly, make your asynchronous code synchronous again. It is recommended to use the composition methods (`thenAccept`, `thenRun`, ...) of the `CompletableFuture` class instead. If your code requires a call to `future.get()` right after the creation of the `CompletableFuture`, this is a sign that asynchrony may not be the correct solution to the problem.

Let's now finally look at the more intricate example of a recursive function:

```
public int factorial(int n) {
    if(n <= 1) return 1;

    // Do some more work
    Thread.sleep(1000);
    return n * factorial(n - 1);
}
```

```
}
```

A simple `CompletableFuture` would be good enough here but the body of the function takes a long time to run and running this function for a parameter of `n = 10` would take nine seconds. Let's use the `RecursiveTask` interface to deal with this problem:

```
class FactorialCalculator extends RecursiveTask<Integer> {
    private Integer n;
    public FactorialCalculator(Integer n) {
        this.n = n;
    }

    @Override
    protected Integer compute() {
        if (n <= 1) {
            return n;
        }
        FactorialCalculator c = new FactorialCalculator(n - 1);
        c.fork();

        // Do some more work
        Thread.sleep(1000);
        return n * c.join();
    }
}
```

This is a divide-and-conquer approach to the factorial function. Threads are created recursively and individually compute their value locally before combining their results into the final value. This reduces the execution time to only two seconds.

Here are some links with more examples on the usage of [CompletableFuture](#), [RecursiveTask](#) and [Thread](#).

## SOA / microservices / edge computing

By combining extreme modularity with distributed computing we obtain a paradigm called **Service Oriented Architecture** (SOA). The main idea here is to split the system into *services* that communicate with each other only via messages over the network. A service is a self-contained blackbox that provides some kind of functionality through a network API and can contact subsequent services to do so. A service must more or less logically correspond to a business activity. These services must be stateless, that means that they either return the requested value or an exception to keep the resource use low. The messages can be implemented by various protocols such as REST or SOAP but they must be exchanged via platforms and programming languages independent technologies (e.g., HTTP). In this way services can be implemented in many different languages, run on different hardware but still be compatible with each other. An example of a service as described here can be a service that provides the shopping cart of customers of a website.

Amazon is known for applying rigorous SOA for all their development. Each service corresponds to a service and must implement the service with an API as if this API will be used by developers from all around the world. This means that they do not “trust” other teams but treat their inputs and the results of their services as outside world<sup>1</sup>. For this to work, we need to have a central service to which other services will register. Customers of the services will then reach this central service to receive the list of available services as well as their APIs.

We now discuss the benefits of the SOA. Firstly, the scalability is greatly improved: if the number of users grows, it is far easier to duplicate services that we need as they are self-contained and stateless. Secondly, if our services are well defined in the first place, they are highly reusable. If the API is the same, it is also easy to swap implementations with others which make the improvements easier.

One of the main downsides of the SOA is the performance. As services communicate only via messages over the network, there is an overhead. If the functionality implemented by your services is too small, this overhead can represent the majority of the latency of your system, which is bad for the overall performance.

Microservices is a variant of SOA. The difference is that microservices refers to the paradigm of implementing an application as small independent and self-contained services communicating via a network (can be message passing between processes on the same OS). SOA relates more to the division of business activities where microservices is more fine-grained inside an application. We can use microservices to implement SOA.

We will now discuss *edge computing* and the relationship between this concept and the SOA we saw before. Edge computing is a paradigm that dictates that the computing power and the data should be close to the customer, hence the name “edge” that means that the “server” should be at the edge of the network rather than at the center (aka in big data centers). Edge computing is applicable when the processing of instant data is required in real-time. The IoT (Internet of Things) is a great example of such applications, that produces a lot of data that needs to be processed quickly. This becomes difficult to perform using data centers, as the bandwidth becomes the bottleneck. Edge computing proposes to use these smart objects to provide services instead of data centers. This joins SOA as services implemented as standalone black boxes communicating via messages over the network can easily be executed on these devices. Another application of edge computing is cloud gaming, in which the latency is key.

## Example: refactoring a monolithic application into microservices

Here we provide you with a well illustrated [example](#) about refactoring a monolithic application into a service based application or microservices. It starts from a monolithic code of a restaurant’s order-delivery management system. After some code extraction steps, the code is refactored into microservices, where each of the order management and delivery management are separate services sharing an API Gateway to interact with them.

## How to implement asynchrony

In this section, we provide some insights on how asynchronous systems are implemented. Note that the material is somewhat advanced and absolutely non-exhaustive, the idea is to give you a taste of what happens under the hood.

## Example: Building Futures using Thread

In this small example, we build a simple implementation of Futures on top of barebone Threads. Note that you should use the existing standard library (or popular available libraries with richer features set) when dealing with asynchrony, however we will use this example to cover a few essential concepts related to implementation details.

---

<sup>1</sup> <https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/>

Let's implement `Future<V>` in Java using plain Java threads. This standalone Future implementation can be constructed from a `Callable` which will asynchronously compute a result of type `V`. The user can also get a new `Future<T>` by chaining from an existing `Future<V>`, using the `andThen` function. This basic example demonstrates callback registration and Future chaining. In particular, it does not support cancellation. Otherwise, it mostly follows a subset of the specifications of the `Java CompletableFuture`.

We represent the state of the Future in an enum: pending, success and failure. Callbacks for success and failure are stored in a list. The result of the computation if successful, and the exception raised if failed, are stored as members.

```
public class Future<V> {

    private enum State {
        PENDING,
        SUCCESS,
        FAILURE,
    }

    private State state = State.PENDING;
    private V result;
    private Exception exception;
    // Note: in practice, static thread pools are used (see Java ExecutionContext)
    private Thread executingThread;

    private List<Consumer<V>> onSuccessCallbacks;
    private List<Consumer<Exception>> onFailureCallbacks;

    public boolean isDone() {
        return hasSucceeded() || hasFailed();
    }

    public boolean hasSucceeded() { return state == State.SUCCESS; }
    public boolean hasFailed() { return state == State.FAILURE; }
}
```

Let's start defining private methods `complete` and `fail`.

```
private void complete(final V completionResult) {
    if (isDone()) throw new IllegalStateException("Cannot complete an already finished Future");
    state = State.SUCCESS;
    result = completionResult;
    for (var callback : onSuccessCallbacks) {
        callback.accept(completionResult);
    }
}

private void fail(final Exception e) {
    if (isDone()) throw new IllegalStateException("Cannot fail an already finished Future");
    state = State.FAILURE;
    exception = e;
    for (var callback : onFailureCallbacks) {
        callback.accept(e);
    }
}
```

The constructor taking a `Callable<V>` immediately creates and starts a `Thread` that executes the function, and completes or fails accordingly.

```

public Future(final Callable<V> computation) {
    onSuccessCallbacks = new ArrayList<>();
    onFailureCallbacks = new ArrayList<>();

    executingThread = new Thread(() -> {
        try {
            complete(computation.call());
        } catch (Exception e) {
            fail(e);
        }
    });
    executingThread.start();
}

```

Let's allow the user to register callbacks for success and failure. We will always allow a user to add a callback. A success callback will be executed immediately if the future is already successful, and never executed if the future is already failed (resp. for failure callbacks).

```

public void addOnSuccessListener(final Consumer<V> callback) {
    onSuccessCallbacks.add(callback);
    if (hasSucceeded()) {
        callback.accept(result);
    }
}

public void addOnFailureListener(final Consumer<Exception> callback) {
    onFailureCallbacks.add(callback);
    if (hasFailed()) {
        callback.accept(exception);
    }
}

```

We are missing a way to let users forcibly wait and get the result of the Future. On failure, the raised exception is wrapped in an ExecutionException and thrown.

```

public V get() throws ExecutionException, InterruptedException {
    executingThread.join();
    if (hasFailed()) {
        throw new ExecutionException(exception);
    } else {
        return result;
    }
}

```

The last thing to implement in our basic Future class is chaining. Chained futures do not have their own execution thread ; rather, they piggyback on an existing future, and are completed when the existing future completes. So is the member `executingThread` useless for them ? It's used in `get()` to join the execution thread, and for a chained future, we should wait for the thread of the piggybacked future to complete, since it will complete the chained future as well. So let's define a private constructor which takes an existing Thread object. The function `andThen()` creates a new future with the current future's own `executingThread`, and registers callbacks on the current future, so that it also completes the new future when it's done.

```

private Future(final Thread thread) {
    this.executingThread = thread;

    onSuccessCallbacks = new ArrayList<>();
    onFailureCallbacks = new ArrayList<>();
}

```



```

public <T> Future<T> andThen(final Function<V, T> then) {
    var futureT = new Future<T>(executingThread);
    addOnSuccessListener(v -> {
        try {
            futureT.complete(then.apply(v));
        } catch (Exception e) {
            futureT.fail(e);
        }
    });
    addOnFailureListener(futureT::fail);
    return futureT;
}

```

Note that the mapping function `then()` may throw, so we take care of catching exceptions and failing the piggybacking future. If we won't, the execution thread of the current future might stop, some callbacks may not be executed, and some futures may get left pending infinitely.

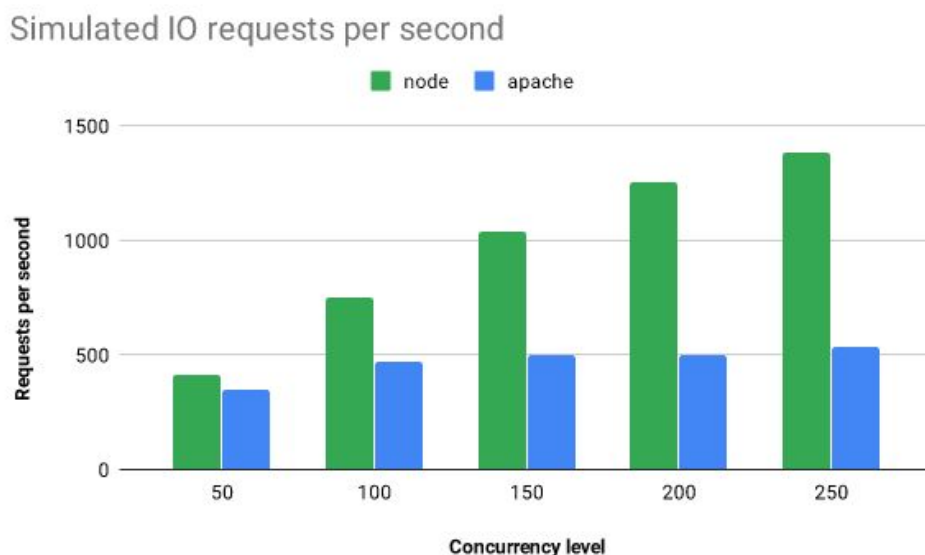
## Synchronous vs asynchronous: A case study

A legitimate question may still linger: is all of this asynchronous stuff actually useful? In other words, does performance scale better with asynchronous implementations rather than equivalent synchronous systems?

First, we define performance: asynchrony allows the total throughput of operations to grow, as operations are not blocked by pending results. The second measurement of interest is the total runtime of one operation, from its entry in the asynchronous system (i.e. the request) until its exit (the last resulting value).

Web systems are perfect candidates for asynchrony: they must deal with a lot of network exchanges, read files from the filesystem and persist data using databases. Traditionally, web servers used blocking implementations to handle incoming client requests, process them and gather the required information, and send an appropriate response back. This was usually done by forking a process and assigning it for each incoming client request. Modern web servers on the other hand are often implemented in an asynchronous fashion, where a single main thread (called the event loop) manages the concurrent requests, and delegates long-running IO tasks to background threads or coroutines (examples NodeJS http, Go http, ...).

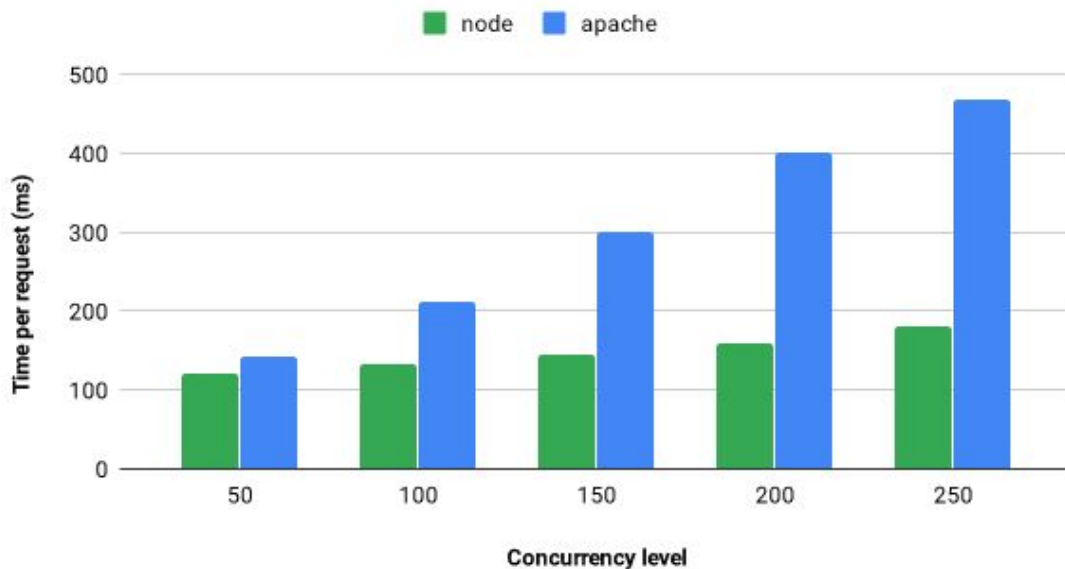
User [emilioSp compared the performance of NodeJS and Apache](#) by simulating an IO operation and a CPU intensive task and observed the following results:



On the x-axis, the concurrency level represents the number of requests sent simultaneously at a time. On the y-axis, each bar represents the number of requests per second that the server was able to serve (higher is

better). We can see that NodeJS was able to scale much better with the concurrency level, as it is well suited for asynchronous IO: offloaded workloads are balanced between threads and no wait is involved. Apache on the other hand struggles after a few hundred connections, since a thread is created for each new request and each of them blocks on the IO computation: the total amount of threads that may coexist on the CPU is the bottleneck.

Simulated IO time per request



The mean time spent per request is also much lower using an asynchronous server compared to a blocking implementation. Since NodeJS never needs to wait on computation results and dispatches callbacks appropriately at the time when asynchronous operations terminate, the accumulated total overhead is much lower.

*Advanced note: actually, it's more complicated™. Apache, Nginx, ... are often considered to be slower under heavy concurrent connection loads (especially since NodeJS' v8 engine is so optimized at execution time and the whole runtime is designed around asynchrony), however nowadays Nginx and the likes are often used in conjunction with asynchronous servers as reverse proxies or dispatchers, and some other runtime will actually handle the request (note that each of the components may have also async variants, like PHP Zend). It is quite common to have either Nginx in front for stability, which blocks / forwards requests to async servers, or sometimes the other way around, for instance using NodeJS in the front as a load balancer for performance.*