Asynchronous Programming

> "Who can wait quietly while the mud settles?
> Who can remain still until the moment of action?"
>
> — Laozi, *Tao Te Ching*

The central part of a computer, the part that carries out the individual steps that make up our programs, is called the *processor*. The programs we have seen so far will keep the processor busy until they have finished their work. The speed at which something like a loop that manipulates numbers can be executed depends pretty much entirely on the speed of the computer's processor and memory.

But many programs interact with things outside of the processor. For example, they may communicate over a computer network or request data from the hard disk—which is a lot slower than getting it from memory.

When such a thing is happening, it would be a shame to let the processor sit idle—there might be some other work it could do in the meantime. In part, this is handled by your operating system, which will switch the processor between multiple running programs. But that doesn't help when we want a

single program to be able to make progress while it is waiting for a network request.

Asynchronicity

In a *synchronous* programming model, things happen one at a time. When you call a function that performs a long-running action, it returns only when the action has finished and it can return the result. This stops your program for the time the action takes.

An *asynchronous* model allows multiple things to happen at the same time. When you start an action, your program continues to run. When the action finishes, the program is informed and gets access to the result (for example, the data read from disk).

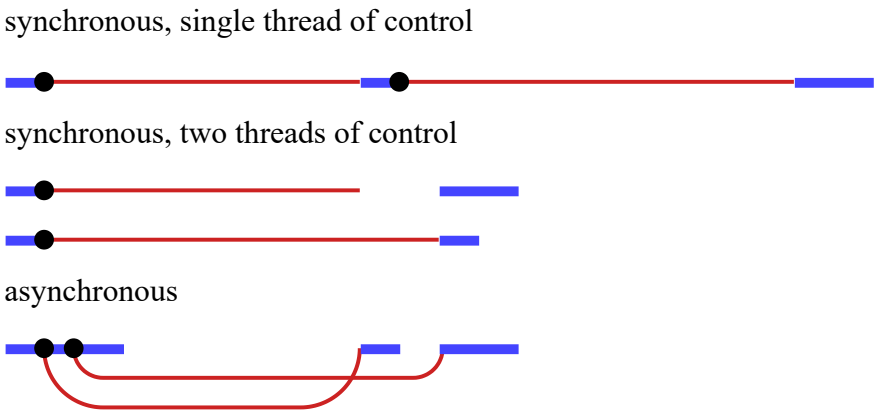
We can compare synchronous and asynchronous programming using a small example: a program that makes two requests over the network and then combines the results.

In a synchronous environment, where the request function returns only after it has done its work, the easiest way to perform this task is to make the requests one after the other. This has the drawback that the second request will be started only when the first has finished. The total time taken will be at least the sum of the two response times.

The solution to this problem, in a synchronous system, is to start additional threads of control. A *thread* is another running program whose execution may be interleaved with other programs by the operating system—since most modern computers contain multiple processors, multiple threads may even run at the same time, on different processors. A second thread could start the second request, and then both threads wait for their results to come back, after which they resynchronize to combine their results.

In the following diagram, the thick lines represent time the program spends running normally, and the thin lines represent time spent waiting for the network. In the synchronous model, the time taken by the network is *part* of the timeline for a given thread of control. In the asynchronous model, starting a network action allows the program to continue running while the network

communication happens alongside it, notifying the program when it is finished.



Another way to describe the difference is that waiting for actions to finish is *implicit* in the synchronous model, while it is *explicit*—under our control—in the asynchronous one.

Asynchronicity cuts both ways. It makes expressing programs that do not fit the straight-line model of control easier, but it can also make expressing programs that do follow a straight line more awkward. We'll see some ways to reduce this awkwardness later in the chapter.

Both prominent JavaScript programming platforms—browsers and Node.js—make operations that might take a while asynchronous, rather than relying on threads. Since programming with threads is notoriously hard (understanding what a program does is much more difficult when it's doing multiple things at once), this is generally considered a good thing.

CALLBACKS

One approach to asynchronous programming is to make functions that need to wait for something take an extra argument, a *callback function*. The asynchronous function starts a process, sets things up so that the callback function is called when the process finishes, and then returns.

As an example, the `setTimeout` function, available both in Node.js and in browsers, waits a given number of milliseconds and then calls a function.

```
setTimeout(() => console.log("Tick"), 500);
```

Waiting is not generally important work, but it can be very useful when you need to arrange for something to happen at a certain time or check whether some action is taking longer than expected.

Another example of a common asynchronous operation is reading a file from a device's storage. Imagine you have a function `readTextFile` that reads a file's content as a string and passes it to a callback function.

```
readTextFile("shopping_list.txt", content => {
  console.log(`Shopping List:\n${content}`);
});
// → Shopping List:
// → Peanut butter
// → Bananas
```

The `readTextFile` function is not part of standard JavaScript. We will see how to read files in the browser and in Node.js in later chapters.

Performing multiple asynchronous actions in a row using callbacks means that you have to keep passing new functions to handle the continuation of the computation after the actions. An asynchronous function that compares two files and produces a boolean indicating whether their content is the same might look like this:

```
function compareFiles(fileA, fileB, callback) {
  readTextFile(fileA, contentA => {
    readTextFile(fileB, contentB => {
      callback(contentA == contentB);
    });
  });
}
```

This style of programming is workable, but the indentation level increases with each asynchronous action because you end up in another function. Doing more complicated things, such as wrapping asynchronous actions in a loop, can get awkward.

In a way, asynchronicity is *contagious*. Any function that calls a function that works asynchronously must itself be asynchronous, using a callback or similar mechanism to deliver its result. Calling a callback is somewhat more involved

and error-prone than simply returning a value, so needing to structure large parts of your program that way is not great.

Promises

A slightly different way to build an asynchronous program is to have asynchronous functions return an object that represents its (future) result instead of passing around callback functions. This way such functions actually return something meaningful, and the shape of the program more closely resembles that of synchronous programs.

This is what the standard class `Promise` is for. A *promise* is a receipt representing a value that may not be available yet. It provides a `then` method that allows you to register a function that should be called when the action for which it is waiting finishes. When the promise is *resolved*, meaning its value becomes available, such functions (there can be multiple) are called with the result value. It is possible to call `then` on a promise that has already resolved—your function will still be called.

The easiest way to create a promise is by calling `Promise.resolve`. This function ensures that the value you give it is wrapped in a promise. If it's already a promise, it is simply returned. Otherwise, you get a new promise that immediately resolves with your value as its result.

```
let fifteen = Promise.resolve(15);
fifteen.then(value => console.log(`Got ${value}`));
// → Got 15
```

To create a promise that does not immediately resolve, you can use `Promise` as a constructor. It has a somewhat odd interface: the constructor expects a function as argument, which it immediately calls, passing it a function that it can use to resolve the promise.

For example, this is how you could create a promise-based interface for the `readTextFile` function:

```
function textFile(filename) {
  return new Promise(resolve => {
    readTextFile(filename, text => resolve(text));
```

```
  });
}

textFile("plans.txt").then(console.log);
```

Note how, in contrast to callback-style functions, this asynchronous function returns a meaningful value—a promise to give you the contents of the file at some point in the future.

A useful thing about the `then` method is that it itself returns another promise. This one resolves to the value returned by the callback function or, if that returned value is a promise, to the value that promise resolves to. Thus, you can "chain" multiple calls to `then` together to set up a sequence of asynchronous actions.

This function, which reads a file full of filenames, and returns the content of a random file in that list, shows this kind of asynchronous promise pipeline:

```
function randomFile(listFile) {
  return textFile(listFile)
    .then(content => content.trim().split("\n"))
    .then(ls => ls[Math.floor(Math.random() * ls.length)])
    .then(filename => textFile(filename));
}
```

The function returns the result of this chain of `then` calls. The initial promise fetches the list of files as a string. The first `then` call transforms that string into an array of lines, producing a new promise. The second `then` call picks a random line from that, producing a third promise that yields a single filename. The final `then` call reads this file, so that the result of the function as a whole is a promise that returns the content of a random file.

In this code, the functions used in the first two `then` calls return a regular value that will immediately be passed into the promise returned by `then` when the function returns. The last `then` call returns a promise (`textFile(filename)`), making it an actual asynchronous step.

It would also have been possible to perform all these steps inside a single `then` callback, since only the last step is actually asynchronous. But the kind

of `then` wrappers that only do some synchronous data transformation are often useful, such as when you want to return a promise that produces a processed version of some asynchronous result.

```
function jsonFile(filename) {
  return textFile(filename).then(JSON.parse);
}

jsonFile("package.json").then(console.log);
```

Generally, it is useful to think of a promise as a device that lets code ignore the question of when a value is going to arrive. A normal value has to actually exist before we can reference it. A promised value is a value that *might* already be there or might appear at some point in the future. Computations defined in terms of promises, by wiring them together with `then` calls, are executed asynchronously as their inputs become available.

Failure

Regular JavaScript computations can fail by throwing an exception. Asynchronous computations often need something like that. A network request may fail, a file may not exist, or some code that is part of the asynchronous computation may throw an exception.

One of the most pressing problems with the callback style of asynchronous programming is that it makes it extremely difficult to ensure failures are properly reported to the callbacks.

A common convention is to use the first argument to the callback to indicate that the action failed, and the second to pass the value produced by the action when it was successful.

```
someAsyncFunction((error, value) => {
  if (error) handleError(error);
  else processValue(value);
});
```

Such callback functions must always check whether they received an exception and make sure that any problems they cause, including exceptions

thrown by functions they call, are caught and given to the right function.

Promises make this easier. They can be either resolved (the action finished successfully) or rejected (it failed). Resolve handlers (as registered with `then`) are called only when the action is successful, and rejections are propagated to the new promise returned by `then`. When a handler throws an exception, this automatically causes the promise produced by its `then` call to be rejected. If any element in a chain of asynchronous actions fails, the outcome of the whole chain is marked as rejected, and no success handlers are called beyond the point where it failed.

Much like resolving a promise provides a value, rejecting one also provides a value, usually called the *reason* of the rejection. When an exception in a handler function causes the rejection, the exception value is used as the reason. Similarly, when a handler returns a promise that is rejected, that rejection flows into the next promise. There's a `Promise.reject` function that creates a new, immediately rejected promise.

To explicitly handle such rejections, promises have a `catch` method that registers a handler to be called when the promise is rejected, similar to how `then` handlers handle normal resolution. It's also very much like `then` in that it returns a new promise, which resolves to the original promise's value when that resolves normally and to the result of the `catch` handler otherwise. If a `catch` handler throws an error, the new promise is also rejected.

As a shorthand, `then` also accepts a rejection handler as a second argument, so you can install both types of handlers in a single method call: `.then(acceptHandler, rejectHandler)`.

A function passed to the `Promise` constructor receives a second argument, alongside the resolve function, which it can use to reject the new promise.

When our `readTextFile` function encounters a problem, it passes the error to its callback function as a second argument. Our `textFile` wrapper should actually check that argument, so that a failure causes the promise it returns to be rejected.

```
function textFile(filename) {
  return new Promise((resolve, reject) => {
    readTextFile(filename, (text, error) => {
      if (error) reject(error);
      else resolve(text);
    });
  });
}
```

The chains of promise values created by calls to `then` and `catch` thus form a pipeline through which asynchronous values or failures move. Since such chains are created by registering handlers, each link has a success handler or a rejection handler (or both) associated with it. Handlers that don't match the type of outcome (success or failure) are ignored. Handlers that do match are called, and their outcome determines what kind of value comes next—success when they return a non-promise value, rejection when they throw an exception, and the outcome of the promise when they return a promise.

```
new Promise((_, reject) => reject(new Error("Fail")))
  .then(value => console.log("Handler 1:", value))
  .catch(reason => {
    console.log("Caught failure " + reason);
    return "nothing";
  })
  .then(value => console.log("Handler 2:", value));
// → Caught failure Error: Fail
// → Handler 2: nothing
```

The first `then` handler function isn't called, because at that point of the pipeline the promise holds a rejection. The `catch` handler handles that rejection and returns a value, which is given to the second `then` handler function.

Much like an uncaught exception is handled by the environment, JavaScript environments can detect when a promise rejection isn't handled and will report this as an error.

Carla

It's a sunny day in Berlin. The runway of the old, decommissioned airport is teeming with cyclists and inline skaters. In the grass near a garbage container, a flock of crows noisily mills about, trying to convince a group of tourists to part with their sandwiches.

One of the crows stands out—a large scruffy female with a few white feathers in her right wing. She is baiting people with a skill and confidence that suggest she's been doing this for a long time. When an elderly man is distracted by the antics of another crow, she casually swoops in, snatches his half-eaten bun from his hand, and sails away.

Contrary to the rest of the group, who look like they are happy to spend the day goofing around here, the large crow looks purposeful. Carrying her loot, she flies straight towards the roof of the hangar building, disappearing into an air vent.

Inside the building, you can hear an odd tapping sound—soft, but persistent. It comes from a narrow space under the roof of an unfinished stairwell. The crow is sitting there, surrounded by her stolen snacks, half a dozen smartphones (several of which are turned on), and a mess of cables. She rapidly taps the screen of one of the phones with her beak. Words are appearing on it. If you didn't know better, you'd think she was typing.

This crow is known to her peers as "cāāw-krö". But since those sounds are poorly suited for human vocal chords, we'll refer to her as Carla.

Carla is a somewhat peculiar crow. In her youth, she was fascinated by human language, eavesdropping on people until she had a good grasp of what they were saying. Later in life, her interest shifted to human technology, and she started stealing phones to study them. Her current project is learning to program. The text she is typing in her hidden lab is, in fact, a piece of asynchronous JavaScript code.

Breaking In

Carla loves the internet. Annoyingly, the phone she is working on is about to run out of prepaid data. The building has a wireless network, but it requires a code to access.

Fortunately, the wireless routers in the building are 20 years old and poorly secured. Doing some research, Carla finds out that the network authentication mechanism has a flaw she can use. When joining the network, a device must send along the correct six-digit passcode. The access point will reply with a success or failure message depending on whether the right code is provided. However, when sending a partial code (say, only three digits), the response is different based on whether those digits are the correct start of the code or not. Sending incorrect numbers immediately returns a failure message. When sending the correct ones, the access point waits for more digits.

This makes it possible to greatly speed up the guessing of the number. Carla can find the first digit by trying each number in turn, until she finds one that doesn't immediately return failure. Having one digit, she can find the second digit in the same way, and so on, until she knows the entire passcode.

Assume Carla has a `joinWifi` function. Given the network name and the passcode (as a string), the function tries to join the network, returning a promise that resolves if successful and rejects if the authentication failed. The first thing she needs is a way to wrap a promise so that it automatically rejects after it takes too much time, to allow the program to quickly move on if the access point doesn't respond.

```
function withTimeout(promise, time) {
  return new Promise((resolve, reject) => {
    promise.then(resolve, reject);
    setTimeout(() => reject("Timed out"), time);
  });
}
```

This makes use of the fact that a promise can only be resolved or rejected once. If the promise given as argument resolves or rejects first, that result will be the result of the promise returned by `withTimeout`. If, on the other hand, the `setTimeout` fires first, rejecting the promise, any further resolve or reject calls are ignored.

To find the whole passcode, the program needs to repeatedly look for the next digit by trying each digit. If authentication succeeds, we know we have found what we are looking for. If it immediately fails, we know that digit was wrong,

and must try the next digit. If the request times out, we have found another correct digit, and must continue by adding another digit.

Because you cannot wait for a promise inside a `for` loop, Carla uses a recursive function to drive this process. On each call, this function gets the code as we know it so far, as well as the next digit to try. Depending on what happens, it may return a finished code or call through to itself, to either start cracking the next position in the code or to try again with another digit.

```
function crackPasscode(networkID) {
  function nextDigit(code, digit) {
    let newCode = code + digit;
    return withTimeout(joinWifi(networkID, newCode), 50)
      .then(() => newCode)
      .catch(failure => {
        if (failure == "Timed out") {
          return nextDigit(newCode, 0);
        } else if (digit < 9) {
          return nextDigit(code, digit + 1);
        } else {
          throw failure;
        }
      });
  }
  return nextDigit("", 0);
}
```

The access point tends to respond to bad authentication requests in about 20 milliseconds, so to be safe, this function waits for 50 milliseconds before timing out a request.

```
crackPasscode("HANGAR 2").then(console.log);
// → 555555
```

Carla tilts her head and sighs. This would have been more satisfying if the code had been a bit harder to guess.

Async functions

Even with promises, this kind of asynchronous code is annoying to write. Promises often need to be tied together in verbose, arbitrary-looking ways. To

create an asynchronous loop, Carla was forced to introduce a recursive function.

The thing the cracking function actually does is completely linear—it always waits for the previous action to complete before starting the next one. In a synchronous programming model, it'd be more straightforward to express.

The good news is that JavaScript allows you to write pseudo-synchronous code to describe asynchronous computation. An `async` function implicitly returns a promise and can, in its body, `await` other promises in a way that *looks* synchronous.

We can rewrite `crackPasscode` like this:

```
async function crackPasscode(networkID) {
  for (let code = "";;) {
    for (let digit = 0;; digit++) {
      let newCode = code + digit;
      try {
        await withTimeout(joinWifi(networkID, newCode), 50);
        return newCode;
      } catch (failure) {
        if (failure == "Timed out") {
          code = newCode;
          break;
        } else if (digit == 9) {
          throw failure;
        }
      }
    }
  }
}
```

This version more clearly shows the double loop structure of the function (the inner loop tries digit 0 to 9, the outer loop adds digits to the passcode).

An `async` function is marked by the word `async` before the `function` keyword. Methods can also be made `async` by writing `async` before their name. When such a function or method is called, it returns a promise. As soon as the function returns something, that promise is resolved. If the body throws an exception, the promise is rejected.

Inside an `async` function, the word `await` can be put in front of an expression to wait for a promise to resolve and only then continue the execution of the function. If the promise rejects, an exception is raised at the point of the `await`.

Such a function no longer runs from start to completion in one go like a regular JavaScript function. Instead, it can be *frozen* at any point that has an `await` and can be resumed at a later time.

For most asynchronous code, this notation is more convenient than directly using promises. You do still need an understanding of promises, since in many cases you'll still interact with them directly. But when wiring them together, `async` functions are generally more pleasant to write than chains of `then` calls.

Generators

This ability of functions to be paused and then resumed again is not exclusive to `async` functions. JavaScript also has a feature called *generator* functions. These are similar, but without the promises.

When you define a function with `function*` (placing an asterisk after the word `function`), it becomes a generator. When you call a generator, it returns an iterator, which we already saw in Chapter 6.

```
function* powers(n) {
  for (let current = n;; current *= n) {
    yield current;
  }
}

for (let power of powers(3)) {
  if (power > 50) break;
  console.log(power);
}
// → 3
// → 9
// → 27
```

Initially, when you call `powers`, the function is frozen at its start. Every time you call `next` on the iterator, the function runs until it hits a `yield` expression, which pauses it and causes the yielded value to become the next value produced by the iterator. When the function returns (the one in the example never does), the iterator is done.

Writing iterators is often much easier when you use generator functions. The iterator for the `Group` class (from the exercise in Chapter 6) can be written with this generator:

```
Group.prototype[Symbol.iterator] = function*() {
  for (let i = 0; i < this.members.length; i++) {
    yield this.members[i];
  }
};
```

There's no longer a need to create an object to hold the iteration state—generators automatically save their local state every time they yield.

Such `yield` expressions may occur only directly in the generator function itself and not in an inner function you define inside of it. The state a generator saves, when yielding, is only its *local* environment and the position where it yielded.

An `async` function is a special type of generator. It produces a promise when called, which is resolved when it returns (finishes) and rejected when it throws an exception. Whenever it yields (awaits) a promise, the result of that promise (value or thrown exception) is the result of the `await` expression.

A Corvid Art Project

One morning, Carla wakes up to unfamiliar noise from the tarmac outside of her hangar. Hopping onto the edge of the roof, she sees the humans are setting up for something. There's a lot of electric cabling, a stage, and some kind of big black wall being built up.

Being a curious crow, Carla takes a closer look at the wall. It appears to consist of a number of large glass-fronted devices wired up to cables. On the back, the devices say "LedTec SIG-5030".

A quick internet search turns up a user's manual for these devices. They appear to be traffic signs, with a programmable matrix of amber LED lights. The intent is of the humans is probably to display some kind of information on them during their event. Interestingly, the screens can be programmed over a wireless network. Could it be they are connected to the building's local network?

Each device on a network gets an *IP address*, which other devices can use to send it messages. We talk more about that in Chapter 13. Carla notices that her own phones all get addresses like `10.0.0.20` or `10.0.0.33`. It might be worth trying to send messages to all such addresses and see if any one of them responds to the interface described in the manual for the signs.

Chapter 18 shows how to make real requests on real networks. In this chapter, we'll use a simplified dummy function called `request` for network communication. This function takes two arguments—a network address and a message, which may be anything that can be sent as JSON—and returns a promise that either resolves to a response from the machine at the given address, or a rejects if there was a problem.

According to the manual, you can change what is displayed on a SIG-5030 sign by sending it a message with content like `{"command": "display", "data": [0, 0, 3, …]}`, where `data` holds one number per LED dot, providing its brightness—0 means off, 3 means maximum brightness. Each sign is 50 lights wide and 30 lights high, so an update command should send 1500 numbers.

This code sends a display update message to all addresses on the local network, to see what sticks. Each of the numbers in an IP address can go from 0 to 255. In the data it sends, it activates a number of lights corresponding to the network address's last number.

```
for (let addr = 1; addr < 256; addr++) {
  let data = [];
  for (let n = 0; n < 1500; n++) {
    data.push(n < addr ? 3 : 0);
  }
  let ip = `10.0.0.${addr}`;
  request(ip, {command: "display", data})
```

```
      .then(() => console.log(`Request to ${ip} accepted`))
      .catch(() => {});
  }
```

Since most of these addresses won't exist or will not accept such messages, the `catch` call makes sure network errors don't crash the program. The requests are all sent out immediately, without waiting for other requests to finish, in order to not waste time when some of the machines don't answer.

Having fired off her network scan, Carla heads back outside to see the result. To her delight, all of the screens are now showing a stripe of light in their top left corners. They *are* on the local network, and they *do* accept commands. She quickly notes the numbers shown on each screen. There are 9 screens, arranged three high and three wide. They have the following network addresses:

```
const screenAddresses = [
  "10.0.0.44", "10.0.0.45", "10.0.0.41",
  "10.0.0.31", "10.0.0.40", "10.0.0.42",
  "10.0.0.48", "10.0.0.47", "10.0.0.46"
];
```

Now this opens up possibilities for all kinds of shenanigans. She could show "crows rule, humans drool" on the wall in giant letters. But that feels a bit crude. Instead, she plans to show a video of a flying crow covering all of the screens at night.

Carla finds a fitting video clip, in which a second and a half of footage can be repeated to create a looping video showing a crow's wingbeat. To fit the nine screens (each of which can show 50×30 pixels), Carla cuts and resizes the videos to get a series of 150×90 images, ten per second. Those are then each cut into nine rectangles, and processed so that the dark spots on the video (where the crow is) show a bright light, and the light spots (no crow) are left dark, which should create the effect of an amber crow flying against a black background.

She has set up the `clipImages` variable to hold an array of frames, where each frame is represented with an array of nine sets of pixels—one for each screen—in the format that the signs expect.

To display a single frame of the video, Carla needs to send a request to all the screens at once. But she also needs to wait for the result of these requests, both in order to not start sending the next frame before the current one has been properly sent, and in order to notice when requests are failing.

`Promise` has a static method `all` that can be used to convert an array of promises into a single promise that resolves to an array of results. This provides a convenient way to have some asynchronous actions happen alongside each other, wait for them all to finish, and then do something with their results (or at least wait for them to make sure they don't fail).

```
function displayFrame(frame) {
  return Promise.all(frame.map((data, i) => {
    return request(screenAddresses[i], {
      command: "display",
      data
    });
  }));
}
```

This maps over the images in `frame` (which is an array of display data arrays) to create an array of request promises. It then returns a promise that combines all of those.

In order to be able to stop a playing video, the process is wrapped in a class. This class has an asynchronous `play` method that returns a promise that only resolves when the playback is stopped again via the `stop` method.

```
function wait(time) {
  return new Promise(accept => setTimeout(accept, time));
}

class VideoPlayer {
  constructor(frames, frameTime) {
    this.frames = frames;
    this.frameTime = frameTime;
    this.stopped = true;
  }

  async play() {
    this.stopped = false;
```

```
      for (let i = 0; !this.stopped; i++) {
        let nextFrame = wait(this.frameTime);
        await displayFrame(this.frames[i % this.frames.length]);
        await nextFrame;
      }
    }

    stop() {
      this.stopped = true;
    }
  }
```

The `wait` function wraps `setTimeout` in a promise that resolves after the given amount of milliseconds. This is useful for controlling the speed of the playback.

```
let video = new VideoPlayer(clipImages, 100);
video.play().catch(e => {
  console.log("Playback failed: " + e);
});
setTimeout(() => video.stop(), 15000);
```

For the entire week that the screen wall stands, every evening, when it is dark, a huge glowing orange bird mysteriously appears on it.

The event loop

An asynchronous program starts by running its main script, which will often set up callbacks to be called later. That main script, as well as the callbacks, run to completion in one piece, uninterrupted. But between them, the program may sit idle, waiting for something to happen.

So callbacks are not directly called by the code that scheduled them. If I call `setTimeout` from within a function, that function will have returned by the time the callback function is called. And when the callback returns, control does not go back to the function that scheduled it.

Asynchronous behavior happens on its own empty function call stack. This is one of the reasons that, without promises, managing exceptions across

asynchronous code is so hard. Since each callback starts with a mostly empty stack, your `catch` handlers won't be on the stack when they throw an exception.

```
try {
  setTimeout(() => {
    throw new Error("Woosh");
  }, 20);
} catch (e) {
  // This will not run
  console.log("Caught", e);
}
```

No matter how closely together events—such as timeouts or incoming requests—happen, a JavaScript environment will run only one program at a time. You can think of this as it running a big loop *around* your program, called the *event loop*. When there's nothing to be done, that loop is paused. But as events come in, they are added to a queue, and their code is executed one after the other. Because no two things run at the same time, slow-running code can delay the handling of other events.

This example sets a timeout but then dallies until after the timeout's intended point of time, causing the timeout to be late.

```
let start = Date.now();
setTimeout(() => {
  console.log("Timeout ran at", Date.now() - start);
}, 20);
while (Date.now() < start + 50) {}
console.log("Wasted time until", Date.now() - start);
// → Wasted time until 50
// → Timeout ran at 55
```

Promises always resolve or reject as a new event. Even if a promise is already resolved, waiting for it will cause your callback to run after the current script finishes, rather than right away.

```
Promise.resolve("Done").then(console.log);
console.log("Me first!");
```

```
// → Me first!
// → Done
```

In later chapters we'll see various other types of events that run on the event loop.

Asynchronous bugs

When your program runs synchronously, in a single go, there are no state changes happening except those that the program itself makes. For asynchronous programs this is different—they may have *gaps* in their execution during which other code can run.

Let's look at an example. This is a function that tries to report the size of each file in an array of files, making sure to read them all at the same time rather than in sequence.

```
async function fileSizes(files) {
  let list = "";
  await Promise.all(files.map(async fileName => {
    list += fileName + ": " +
      (await textFile(fileName)).length + "\n";
  }));
  return list;
}
```

The `async fileName =>` part shows how arrow functions can also be made `async` by putting the word `async` in front of them.

The code doesn't immediately look suspicious... it maps the `async` arrow function over the array of names, creating an array of promises, and then uses `Promise.all` to wait for all of these before returning the list they build up.

But this program is entirely broken. It'll always return only a single line of output, listing the file that took the longest to read.

```
fileSizes(["plans.txt", "shopping_list.txt"])
  .then(console.log);
```

Can you work out why?

The problem lies in the `+=` operator, which takes the *current* value of `list` at the time where the statement starts executing and then, when the `await` finishes, sets the `list` binding to be that value plus the added string.

But between the time where the statement starts executing and the time where it finishes there's an asynchronous gap. The `map` expression runs before anything has been added to the list, so each of the `+=` operators starts from an empty string and ends up, when its storage retrieval finishes, setting `list` to the result of adding its line to the empty string.

This could have easily been avoided by returning the lines from the mapped promises and calling `join` on the result of `Promise.all`, instead of building up the list by changing a binding. As usual, computing new values is less error-prone than changing existing values.

```
async function fileSizes(files) {
  let lines = files.map(async fileName => {
    return fileName + ": " +
      (await textFile(fileName)).length;
  });
  return (await Promise.all(lines)).join("\n");
}
```

Mistakes like this are easy to make, especially when using `await`, and you should be aware of where the gaps in your code occur. An advantage of JavaScript's *explicit* asynchronicity (whether through callbacks, promises, or `await`) is that spotting these gaps is relatively easy.

Summary

Asynchronous programming makes it possible to express waiting for long-running actions without freezing the whole program. JavaScript environments typically implement this style of programming using callbacks, functions that are called when the actions complete. An event loop schedules such callbacks to be called when appropriate, one after the other, so that their execution does not overlap.

Programming asynchronously is made easier by promises, objects that represent actions that might complete in the future, and `async` functions,

which allow you to write an asynchronous program as if it were synchronous.

Exercises

Quiet Times

There's a security camera near Carla's lab that's activated by a motion sensor. It is connected to the network and starts sending out a video stream when it is active. Because she'd rather not be discovered, Carla has set up a system that notices this kind of wireless network traffic and turns on a light in her lair whenever there is activity outside, so she knows when to keep quiet.

She's also been logging the times at which the camera is tripped for a while and wants to use this information to visualize which times, in an average week, tend to be quiet, and which tend to be busy. The log is stored in files holding one time stamp number (as returned by `Date.now()`) per line.

```
1695709940692
1695701068331
1695701189163
```

The `"camera_logs.txt"` file holds a list of log files. Write an asynchronous function `activityTable(day)` that for a given day of the week returns an array of 24 numbers, one for each hour of the day, that hold the number of camera network traffic observations seen in that hour of the day. Days are identified by number using the system used by `Date.getDay`, where Sunday is 0 and Saturday is 6.

The `activityGraph` function, provided by the sandbox, summarizes such a table into a string.

To read the files, use the `textFile` function defined earlier—given a filename, it returns a promise that resolves to the file's content. Remember that `new Date(timestamp)` creates a `Date` object for that time, which has `getDay` and `getHours` methods returning the day of the week and the hour of the day.

Both types of files—the list of log files and the log files themselves—have each piece of data on its own line, separated by newline (`"\n"`) characters.

```
async function activityTable(day) {
  let logFileList = await textFile("camera_logs.txt");
  // Your code here
}

activityTable(1)
  .then(table => console.log(activityGraph(table)));
```

▶ Display hints...

Real Promises

Rewrite the function from the previous exercise without `async`/`await`, using plain `Promise` methods.

```
function activityTable(day) {
  // Your code here
}

activityTable(6)
  .then(table => console.log(activityGraph(table)));
```

In this style, using `Promise.all` will be more convenient than trying to model a loop over the log files. In the `async` function, just using `await` in a loop is simpler. If reading a file takes some time, which of these two approaches will take the least time to run?

If one of the files listed in the file list has a typo, and reading it fails, how does that failure end up in the `Promise` object that your function returns?

▶ Display hints...

Building Promise.all

As we saw, given an array of promises, `Promise.all` returns a promise that waits for all of the promises in the array to finish. It then succeeds, yielding an array of result values. If a promise in the array fails, the promise returned by `all` fails too, passing on the failure reason from the failing promise.

Implement something like this yourself as a regular function called `Promise_all`.

Remember that after a promise has succeeded or failed, it can't succeed or fail again, and further calls to the functions that resolve it are ignored. This can simplify the way you handle failure of your promise.

```
function Promise_all(promises) {
  return new Promise((resolve, reject) => {
    // Your code here.
  });
}

// Test code.
Promise_all([]).then(array => {
  console.log("This should be []:", array);
});
function soon(val) {
  return new Promise(resolve => {
    setTimeout(() => resolve(val), Math.random() * 500);
  });
}
Promise_all([soon(1), soon(2), soon(3)]).then(array => {
  console.log("This should be [1, 2, 3]:", array);
});
Promise_all([soon(1), Promise.reject("X"), soon(3)])
  .then(array => {
    console.log("We should not get here");
  })
  .catch(error => {
    if (error != "X") {
      console.log("Unexpected failure:", error);
    }
  });
```

▶ Display hints...

◂ ● ▸ ?