

ROADS MODULE

Write a CommonJS module, based on the example from [Chapter 7](#), that contains the array of roads and exports the graph data structure representing them as `roadGraph`. It should depend on a module `./graph`, which exports a function `buildGraph` that is used to build the graph. This function expects an array of two-element arrays (the start and end points of the roads).

CIRCULAR DEPENDENCIES

A circular dependency is a situation where module A depends on B, and B also, directly or indirectly, depends on A. Many module systems simply forbid this because whichever order you choose for loading such modules, you cannot make sure that each module's dependencies have been loaded before it runs.

CommonJS modules allow a limited form of cyclic dependencies. As long as the modules do not replace their default `exports` object and don't access each other's interface until after they finish loading, cyclic dependencies are okay.

The `require` function given [earlier in this chapter](#) supports this type of dependency cycle. Can you see how it handles cycles? What would go wrong when a module in a cycle *does* replace its default `exports` object?

*“Who can wait quietly while the mud settles?
Who can remain still until the moment of action?”*

—Laozi, Tao Te Ching

CHAPTER 11

ASYNCHRONOUS PROGRAMMING

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 are things that 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 processor.

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).

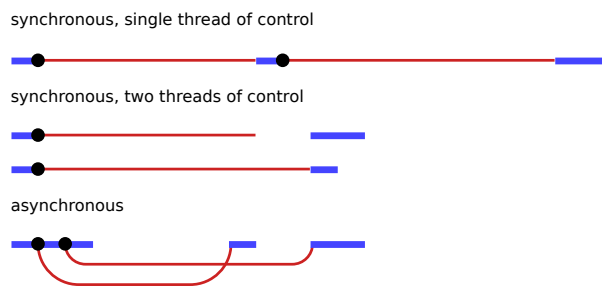
We can compare synchronous and asynchronous programming using a small example: a program that fetches two resources from the network and then combines results.

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 conceptually causes a *split* in the timeline. The program that initiated the action continues running, and the action 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 address this awkwardness later in the chapter.

Both of the important 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.

CROW TECH

Most people are aware of the fact that crows are very smart birds. They can use tools, plan ahead, remember things, and even communicate these things among themselves.

What most people don't know is that they are capable of many things that they keep well hidden from us. I've been told by a reputable (if somewhat eccentric) expert on corvids that crow technology is not far behind human technology, and they are catching up.

For example, many crow cultures have the ability to construct computing devices. These are not electronic, as human computing devices are, but operate through the actions of tiny insects, a species closely related to the termite, which has developed a symbiotic relationship with the crows. The birds provide them with food, and in return the insects build and operate their complex colonies that, with the help of the living creatures inside them, perform computations.

Such colonies are usually located in big, long-lived nests. The birds and insects work together to build a network of bulbous clay structures, hidden between the twigs of the nest, in which the insects live and work.

To communicate with other devices, these machines use light signals. The crows embed pieces of reflective material in special communication stalks, and the insects aim these to reflect light at another nest, encoding data as a sequence of quick flashes. This means that only nests that have an unbroken visual connection can communicate.

Our friend the corvid expert has mapped the network of crow nests in the village of Hières-sur-Amby, on the banks of the river Rhône. This map shows the nests and their connections:



In an astounding example of convergent evolution, crow computers run JavaScript. In this chapter we'll write some basic networking functions for them.

CALLBACKS

One approach to asynchronous programming is to make functions that perform a slow action take an extra argument, a *callback function*. The action is started, and when it finishes, the callback function is called with the result.

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

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

Waiting is not generally a very important type of work, but it can be useful when doing something like updating an animation or checking whether something is taking longer than a given amount of time.

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.

Most crow nest computers have a long-term data storage bulb, where pieces of information are etched into twigs so that they can be retrieved later. Etching, or finding a piece of data, takes a moment, so the interface to long-term storage is asynchronous and uses callback functions.

Storage bulbs store pieces of JSON-encodable data under names. A crow might store information about the places where it's hidden food under the name "food caches", which could hold an array of names that point at other pieces of data, describing the actual cache. To look up a food cache in the storage bulbs of the *Big Oak* nest, a crow could run code like this:

```
import {bigOak} from "./crow-tech";

bigOak.readStorage("food caches", caches => {
  let firstCache = caches[0];
  bigOak.readStorage(firstCache, info => {
    console.log(info);
  });
});
```

(All binding names and strings have been translated from crow language to English.)

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 running multiple actions at the same time,

can get a little awkward.

Crow nest computers are built to communicate using request-response pairs. That means one nest sends a message to another nest, which then immediately sends a message back, confirming receipt and possibly including a reply to a question asked in the message.

Each message is tagged with a *type*, which determines how it is handled. Our code can define handlers for specific request types, and when such a request comes in, the handler is called to produce a response.

The interface exported by the `./crow-tech` module provides callback-based functions for communication. Nests have a `send` method that sends off a request. It expects the name of the target nest, the type of the request, and the content of the request as its first three arguments, and it expects a function to call when a response comes in as its fourth and last argument.

```
bigOak.send("Cow Pasture", "note", "Let's caw loudly at 7PM",
  () => console.log("Note delivered."));
```

But to make nests capable of receiving that request, we first have to define a request type named `"note"`. The code that handles the requests has to run not just on this nest-computer but on all nests that can receive messages of this type. We'll just assume that a crow flies over and installs our handler code on all the nests.

```
import {defineRequestType} from "./crow-tech";

defineRequestType("note", (nest, content, source, done) => {
  console.log(`${nest.name} received note: ${content}`);
  done();
});
```

The `defineRequestType` function defines a new type of request. The example adds support for `"note"` requests, which just sends a note to a given nest. Our implementation calls `console.log` so that we can verify that the request arrived. Nests have a `name` property that holds their name.

The fourth argument given to the handler, `done`, is a callback function that it must call when it is done with the request. If we had used the handler's return value as the response value, that would mean that a request handler can't itself perform asynchronous actions. A function doing asynchronous work typically returns before the work is done, having arranged for a callback to be called when it completes. So we need some asynchronous mechanism—in this case,

another callback function—to signal when a response is available.

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

Working with abstract concepts is often easier when those concepts can be represented by values. In the case of asynchronous actions, you could, instead of arranging for a function to be called at some point in the future, return an object that represents this future event.

This is what the standard class `Promise` is for. A *promise* is an asynchronous action that may complete at some point and produce a value. It is able to notify anyone who is interested when its value is available.

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 finishes with your value as its result.

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

To get the result of a promise, you can use its `then` method. This registers a callback function to be called when the promise resolves and produces a value. You can add multiple callbacks to a single promise, and they will be called, even if you add them after the promise has already *resolved* (finished).

But that's not all the `then` method does. It returns another promise, which resolves to the value that the handler function returns or, if that returns a promise, waits for that promise and then resolves to its result.

It is useful to think of promises as a device to move values into an asynchronous reality. A normal value is simply there. 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 act on such wrapped values and are executed asynchronously as the values become available.

To create a promise, 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. It works this way, instead of for example with a `resolve` method, so that only the code that created the promise can resolve it.

This is how you'd create a promise-based interface for the `readStorage` function:

```
function storage(nest, name) {
  return new Promise(resolve => {
    nest.readStorage(name, result => resolve(result));
  });
}

storage(bigOak, "enemies")
  .then(value => console.log("Got", value));
```

This asynchronous function returns a meaningful value. This is the main advantage of promises—they simplify the use of asynchronous functions. Instead of having to pass around callbacks, promise-based functions look similar to regular ones: they take input as arguments and return their output. The only difference is that the output may not be available yet.

FAILURE

Regular JavaScript computations can fail by throwing an exception. Asynchronous computations often need something like that. A network request may fail, 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 make sure failures are properly reported to the callbacks.

A widely used convention is that the first argument to the callback is used to indicate that the action failed, and the second contains the value produced by the action when it was successful. 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 automatically propagated to the new promise that is returned by `then`. And when a handler

throws an exception, this automatically causes the promise produced by its `then` call to be rejected. So 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 one, 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 if it 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.

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

The chains of promise values created by calls to `then` and `catch` can be seen as 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. But those that do match are called, and their outcome determines what kind of value comes next—success when it returns a non-promise value, rejection when it throws an exception, and the outcome of a promise when it returns one of those.

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

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.

NETWORKS ARE HARD

Occasionally, there isn't enough light for the crows' mirror systems to transmit a signal or something is blocking the path of the signal. It is possible for a signal to be sent but never received.

As it is, that will just cause the callback given to `send` to never be called, which will probably cause the program to stop without even noticing there is a problem. It would be nice if, after a given period of not getting a response, a request would *time out* and report failure.

Often, transmission failures are random accidents, like a car's headlight interfering with the light signals, and simply retrying the request may cause it to succeed. So while we're at it, let's make our request function automatically retry the sending of the request a few times before it gives up.

And, since we've established that promises are a good thing, we'll also make our request function return a promise. In terms of what they can express, callbacks and promises are equivalent. Callback-based functions can be wrapped to expose a promise-based interface, and vice versa.

Even when a request and its response are successfully delivered, the response may indicate failure—for example, if the request tries to use a request type that hasn't been defined or the handler throws an error. To support this, `send` and `defineRequestType` follow the convention mentioned before, where the first argument passed to callbacks is the failure reason, if any, and the second is the actual result.

These can be translated to promise resolution and rejection by our wrapper.

```
class Timeout extends Error {}

function request(nest, target, type, content) {
  return new Promise((resolve, reject) => {
    let done = false;
    function attempt(n) {
      nest.send(target, type, content, (failed, value) => {
        done = true;
        if (failed) reject(failed);
        else resolve(value);
      });
      setTimeout(() => {
        if (done) return;
        else if (n < 3) attempt(n + 1);
        else reject(new Timeout("Timed out"));
      }, 1000);
    }
    attempt(0);
  });
}
```

```

        }, 250);
    }
    attempt(1);
  });
}

```

Because promises can be resolved (or rejected) only once, this will work. The first time `resolve` or `reject` is called determines the outcome of the promise, and further calls caused by a request coming back after another request finished are ignored.

To build an asynchronous loop, for the retries, we need to use a recursive function—a regular loop doesn’t allow us to stop and wait for an asynchronous action. The `attempt` function makes a single attempt to send a request. It also sets a timeout that, if no response has come back after 250 milliseconds, either starts the next attempt or, if this was the third attempt, rejects the promise with an instance of `Timeout` as the reason.

Retrying every quarter-second and giving up when no response has come in after three-quarter second is definitely somewhat arbitrary. It is even possible, if the request did come through but the handler is just taking a bit longer, for requests to be delivered multiple times. We’ll write our handlers with that problem in mind—duplicate messages should be harmless.

In general, we will not be building a world-class, robust network today. But that’s okay—crows don’t have very high expectations yet when it comes to computing.

To isolate ourselves from callbacks altogether, we’ll go ahead and also define a wrapper for `defineRequestType` that allows the handler function to return a promise or plain value and wires that up to the callback for us.

```

function requestType(name, handler) {
  defineRequestType(name, (nest, content, source,
    callback) => {
    try {
      Promise.resolve(handler(nest, content, source))
        .then(response => callback(null, response),
          failure => callback(failure));
    } catch (exception) {
      callback(exception);
    }
  });
}

```

`Promise.resolve` is used to convert the value returned by handler to a promise if it isn't already.

Note that the call to handler had to be wrapped in a `try` block to make sure any exception it raises directly is given to the callback. This nicely illustrates the difficulty of properly handling errors with raw callbacks—it is easy to forget to properly route exceptions like that, and if you don't do it, failures won't get reported to the right callback. Promises make this mostly automatic and thus less error-prone.

COLLECTIONS OF PROMISES

Each nest computer keeps an array of other nests within transmission distance in its `neighbors` property. To check which of those are currently reachable, you could write a function that tries to send a "ping" request (a request that simply asks for a response) to each of them and see which ones come back.

When working with collections of promises running at the same time, the `Promise.all` function can be useful. It returns a promise that waits for all of the promises in the array to resolve and then resolves to an array of the values that these promises produced (in the same order as the original array). If any promise is rejected, the result of `Promise.all` is itself rejected.

```
requestType("ping", () => "pong");

function availableNeighbors(nest) {
  let requests = nest.neighbors.map(neighbor => {
    return request(nest, neighbor, "ping")
      .then(() => true, () => false);
  });
  return Promise.all(requests).then(result => {
    return nest.neighbors.filter((_, i) => result[i]);
  });
}
```

When a neighbor isn't available, we don't want the entire combined promise to fail since then we still wouldn't know anything. So the function that is mapped over the set of neighbors to turn them into request promises attaches handlers that make successful requests produce `true` and rejected ones produce `false`.

In the handler for the combined promise, `filter` is used to remove those elements from the `neighbors` array whose corresponding value is `false`. This makes use of the fact that `filter` passes the array index of the current element

as a second argument to its filtering function (`map`, `some`, and similar higher-order array methods do the same).

NETWORK FLOODING

The fact that nests can talk only to their neighbors greatly inhibits the usefulness of this network.

For broadcasting information to the whole network, one solution is to set up a type of request that is automatically forwarded to neighbors. These neighbors then in turn forward it to their neighbors, until the whole network has received the message.

```
import {everywhere} from "../crow-tech";

everywhere(nest => {
  nest.state.gossip = [];
});

function sendGossip(nest, message, exceptFor = null) {
  nest.state.gossip.push(message);
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "gossip", message);
  }
}

requestType("gossip", (nest, message, source) => {
  if (nest.state.gossip.includes(message)) return;
  console.log(`${nest.name} received gossip '${message}' from ${source}`);
  sendGossip(nest, message, source);
});
```

To avoid sending the same message around the network forever, each nest keeps an array of gossip strings that it has already seen. To define this array, we use the `everywhere` function—which runs code on every nest—to add a property to the nest’s `state` object, which is where we’ll keep nest-local state.

When a nest receives a duplicate gossip message, which is very likely to happen with everybody blindly resending them, it ignores it. But when it receives a new message, it excitedly tells all its neighbors except for the one who sent it the message.

This will cause a new piece of gossip to spread through the network like an

ink stain in water. Even when some connections aren't currently working, if there is an alternative route to a given nest, the gossip will reach it through there.

This style of network communication is called *flooding*—it floods the network with a piece of information until all nodes have it.

MESSAGE ROUTING

If a given node wants to talk to a single other node, flooding is not a very efficient approach. Especially when the network is big, that would lead to a lot of useless data transfers.

An alternative approach is to set up a way for messages to hop from node to node until they reach their destination. The difficulty with that is it requires knowledge about the layout of the network. To send a request in the direction of a faraway nest, it is necessary to know which neighboring nest gets it closer to its destination. Sending it in the wrong direction will not do much good.

Since each nest knows only about its direct neighbors, it doesn't have the information it needs to compute a route. We must somehow spread the information about these connections to all nests, preferably in a way that allows it to change over time, when nests are abandoned or new nests are built.

We can use flooding again, but instead of checking whether a given message has already been received, we now check whether the new set of neighbors for a given nest matches the current set we have for it.

```
requestType("connections", (nest, {name, neighbors},
                             source) => {
  let connections = nest.state.connections;
  if (JSON.stringify(connections.get(name)) ==
      JSON.stringify(neighbors)) return;
  connections.set(name, neighbors);
  broadcastConnections(nest, name, source);
});

function broadcastConnections(nest, name, exceptFor = null) {
  for (let neighbor of nest.neighbors) {
    if (neighbor == exceptFor) continue;
    request(nest, neighbor, "connections", {
      name,
      neighbors: nest.state.connections.get(name)
    });
  }
}
```

```

everywhere(nest => {
  nest.state.connections = new Map();
  nest.state.connections.set(nest.name, nest.neighbors);
  broadcastConnections(nest, nest.name);
});

```

The comparison uses `JSON.stringify` because `==`, on objects or arrays, will return true only when the two are the exact same value, which is not what we need here. Comparing the JSON strings is a crude but effective way to compare their content.

The nodes immediately start broadcasting their connections, which should, unless some nests are completely unreachable, quickly give every nest a map of the current network graph.

A thing you can do with graphs is find routes in them, as we saw in [Chapter 7](#). If we have a route toward a message's destination, we know which direction to send it in.

This `findRoute` function, which greatly resembles the `findRoute` from [Chapter 7](#), searches for a way to reach a given node in the network. But instead of returning the whole route, it just returns the next step. That next nest will itself, using its current information about the network, decide where *it* sends the message.

```

function findRoute(from, to, connections) {
  let work = [{at: from, via: null}];
  for (let i = 0; i < work.length; i++) {
    let {at, via} = work[i];
    for (let next of connections.get(at) || []) {
      if (next == to) return via;
      if (!work.some(w => w.at == next)) {
        work.push({at: next, via: via || next});
      }
    }
  }
  return null;
}

```

Now we can build a function that can send long-distance messages. If the message is addressed to a direct neighbor, it is delivered as usual. If not, it is packaged in an object and sent to a neighbor that is closer to the target, using the "route" request type, which will cause that neighbor to repeat the same

behavior.

```
function routeRequest(nest, target, type, content) {
  if (nest.neighbors.includes(target)) {
    return request(nest, target, type, content);
  } else {
    let via = findRoute(nest.name, target,
                        nest.state.connections);
    if (!via) throw new Error(`No route to ${target}`);
    return request(nest, via, "route",
                  {target, type, content});
  }
}

requestType("route", (nest, {target, type, content}) => {
  return routeRequest(nest, target, type, content);
});
```

We've constructed several layers of functionality on top of a primitive communication system to make it convenient to use. This is a nice (though simplified) model of how real computer networks work.

A distinguishing property of computer networks is that they aren't reliable—abstractions built on top of them can help, but you can't abstract away network failure. So network programming is typically very much about anticipating and dealing with failures.

ASYNC FUNCTIONS

To store important information, crows are known to duplicate it across nests. That way, when a hawk destroys a nest, the information isn't lost.

To retrieve a given piece of information that it doesn't have in its own storage bulb, a nest computer might consult random other nests in the network until it finds one that has it.

```
requestType("storage", (nest, name) => storage(nest, name));

function findInStorage(nest, name) {
  return storage(nest, name).then(found => {
    if (found != null) return found;
    else return findInRemoteStorage(nest, name);
  });
}
```



```

function network(nest) {
  return Array.from(nest.state.connections.keys());
}

function findInRemoteStorage(nest, name) {
  let sources = network(nest).filter(n => n !== nest.name);
  function next() {
    if (sources.length === 0) {
      return Promise.reject(new Error("Not found"));
    } else {
      let source = sources[Math.floor(Math.random() *
        sources.length)];
      sources = sources.filter(n => n !== source);
      return routeRequest(nest, source, "storage", name)
        .then(value => value !== null ? value : next(),
          next);
    }
  }
  return next();
}

```

Because `connections` is a `Map`, `Object.keys` doesn't work on it. It has a `keys` *method*, but that returns an iterator rather than an array. An iterator (or iterable value) can be converted to an array with the `Array.from` function.

Even with promises this is some rather awkward code. Multiple asynchronous actions are chained together in non-obvious ways. We again need a recursive function (`next`) to model looping through the nests.

And the thing the code 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 simpler to express.

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

We can rewrite `findInStorage` like this:

```

async function findInStorage(nest, name) {
  let local = await storage(nest, name);
  if (local !== null) return local;

  let sources = network(nest).filter(n => n !== nest.name);
  while (sources.length > 0) {
    let source = sources[Math.floor(Math.random() *

```

```

        sources.length]);
sources = sources.filter(n => n !== source);
try {
    let found = await routeRequest(nest, source, "storage",
                                   name);
    if (found !== null) return found;
} catch (_) {}
}
throw new Error("Not found");
}

```

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 body returns something, that promise is resolved. If it 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.

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

For non-trivial asynchronous code, this notation is usually more convenient than directly using promises. Even if you need to do something that doesn't fit the synchronous model, such as perform multiple actions at the same time, it is easy to combine `await` with the direct use of promises.

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.

THE EVENT LOOP

Asynchronous programs are executed piece by piece. Each piece may start some actions and schedule code to be executed when the action finishes or fails. In between these pieces, the program sits idle, waiting for the next action.

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 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 (_) {
  // This will not run
  console.log("Caught!");
}
```

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 stopped. 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 might 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. One of the hobbies of our crows is to count the number of chicks that hatch throughout the village every year. Nests store this count in their storage bulbs. The following code tries to enumerate the counts from all the nests for a given year:

```
function anyStorage(nest, source, name) {  
  if (source == nest.name) return storage(nest, name);  
  else return routeRequest(nest, source, "storage", name);  
}  
  
async function chicks(nest, year) {  
  let list = "";  
  await Promise.all(network(nest).map(async name => {  
    list += `${name}: ${  
      await anyStorage(nest, name, `chicks in ${year}`)  
    }\\n`;  
  }));  
  return list;  
}
```

The `async name =>` part shows that 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 set of nests, creating an array of promises, and then uses `Promise.all` to wait for all of these before returning the list they build up.

But it is seriously broken. It'll always return only a single line of output, listing the nest that was slowest to respond.

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 a single-line list—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 chicks(nest, year) {
  let lines = network(nest).map(async name => {
    return name + ": " +
      await anyStorage(nest, name, `chicks in ${year}`);
  });
  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 program during these actions. 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

TRACKING THE SCALPEL

The village crows own an old scalpel that they occasionally use on special missions—say, to cut through screen doors or packaging. To be able to quickly track it down, every time the scalpel is moved to another nest, an entry is added to the storage of both the nest that had it and the nest that took it, under the name "scalpel", with its new location as the value.

This means that finding the scalpel is a matter of following the breadcrumb trail of storage entries, until you find a nest where that points at the nest itself.

Write an `async` function `locateScalpel` that does this, starting at the nest on which it runs. You can use the `anyStorage` function defined earlier to access storage in arbitrary nests. The scalpel has been going around long enough that you may assume that every nest has a "scalpel" entry in its data storage.

Next, write the same function again without using `async` and `await`.

Do request failures properly show up as rejections of the returned promise in both versions? How?

BUILDING PROMISE.ALL

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, with 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.

“The evaluator, which determines the meaning of expressions in a programming language, is just another program.”

—Hal Abelson and Gerald Sussman, *Structure and Interpretation of Computer Programs*

CHAPTER 12

PROJECT: A PROGRAMMING LANGUAGE

Building your own programming language is surprisingly easy (as long as you do not aim too high) and very enlightening.

The main thing I want to show in this chapter is that there is no magic involved in building your own language. I’ve often felt that some human inventions were so immensely clever and complicated that I’d never be able to understand them. But with a little reading and experimenting, they often turn out to be quite mundane.

We will build a programming language called Egg. It will be a tiny, simple language—but one that is powerful enough to express any computation you can think of. It will allow simple abstraction based on functions.

PARSING

The most immediately visible part of a programming language is its *syntax*, or notation. A *parser* is a program that reads a piece of text and produces a data structure that reflects the structure of the program contained in that text. If the text does not form a valid program, the parser should point out the error.

Our language will have a simple and uniform syntax. Everything in Egg is an expression. An expression can be the name of a binding, a number, a string, or an *application*. Applications are used for function calls but also for constructs such as `if` or `while`.

To keep the parser simple, strings in Egg do not support anything like backslash escapes. A string is simply a sequence of characters that are not double quotes, wrapped in double quotes. A number is a sequence of digits. Binding names can consist of any character that is not whitespace and that does not have a special meaning in the syntax.

Applications are written the way they are in JavaScript, by putting parentheses after an expression and having any number of arguments between those parentheses, separated by commas.

```
do(define(x, 10),
```



```

if(>(x, 5),
    print("large"),
    print("small"))

```

The uniformity of the Egg language means that things that are operators in JavaScript (such as `>`) are normal bindings in this language, applied just like other functions. And since the syntax has no concept of a block, we need a `do` construct to represent doing multiple things in sequence.

The data structure that the parser will use to describe a program consists of expression objects, each of which has a `type` property indicating the kind of expression it is and other properties to describe its content.

Expressions of type `"value"` represent literal strings or numbers. Their `value` property contains the string or number value that they represent. Expressions of type `"word"` are used for identifiers (names). Such objects have a `name` property that holds the identifier's name as a string. Finally, `"apply"` expressions represent applications. They have an `operator` property that refers to the expression that is being applied, as well as an `args` property that holds an array of argument expressions.

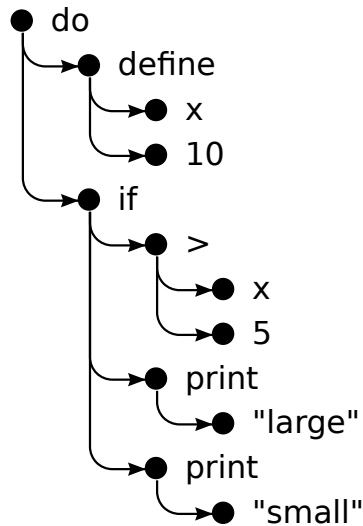
The `>(x, 5)` part of the previous program would be represented like this:

```

{
  type: "apply",
  operator: {type: "word", name: ">"},
  args: [
    {type: "word", name: "x"},
    {type: "value", value: 5}
  ]
}

```

Such a data structure is called a *syntax tree*. If you imagine the objects as dots and the links between them as lines between those dots, it has a treelike shape. The fact that expressions contain other expressions, which in turn might contain more expressions, is similar to the way tree branches split and split again.



Contrast this to the parser we wrote for the configuration file format in [Chapter 9](#), which had a simple structure: it split the input into lines and handled those lines one at a time. There were only a few simple forms that a line was allowed to have.

Here we must find a different approach. Expressions are not separated into lines, and they have a recursive structure. Application expressions *contain* other expressions.

Fortunately, this problem can be solved very well by writing a parser function that is recursive in a way that reflects the recursive nature of the language.

We define a function `parseExpression`, which takes a string as input and returns an object containing the data structure for the expression at the start of the string, along with the part of the string left after parsing this expression. When parsing subexpressions (the argument to an application, for example), this function can be called again, yielding the argument expression as well as the text that remains. This text may in turn contain more arguments or may be the closing parenthesis that ends the list of arguments.

This is the first part of the parser:

```

function parseExpression(program) {
  program = skipSpace(program);
  let match, expr;
  if (match = /^[^"]*)"/.exec(program)) {
    expr = {type: "value", value: match[1]};
  } else if (match = /^[\d+\b]/.exec(program)) {
    expr = {type: "value", value: Number(match[0])};
  } else if (match = /^[^\s(),#"]+/\.exec(program)) {
    expr = {type: "word", name: match[0]};
  } else {

```

```

    throw new SyntaxError("Unexpected syntax: " + program);
  }

  return parseApply(expr, program.slice(match[0].length));
}

function skipSpace(string) {
  let first = string.search(/\S/);
  if (first == -1) return "";
  return string.slice(first);
}

```

Because Egg, like JavaScript, allows any amount of whitespace between its elements, we have to repeatedly cut the whitespace off the start of the program string. That is what the `skipSpace` function helps with.

After skipping any leading space, `parseExpression` uses three regular expressions to spot the three atomic elements that Egg supports: strings, numbers, and words. The parser constructs a different kind of data structure depending on which one matches. If the input does not match one of these three forms, it is not a valid expression, and the parser throws an error. We use `SyntaxError` instead of `Error` as the exception constructor, which is another standard error type, because it is a little more specific—it is also the error type thrown when an attempt is made to run an invalid JavaScript program.

We then cut off the part that was matched from the program string and pass that, along with the object for the expression, to `parseApply`, which checks whether the expression is an application. If so, it parses a parenthesized list of arguments.

```

function parseApply(expr, program) {
  program = skipSpace(program);
  if (program[0] != "(") {
    return {expr: expr, rest: program};
  }

  program = skipSpace(program.slice(1));
  expr = {type: "apply", operator: expr, args: []};
  while (program[0] != ")") {
    let arg = parseExpression(program);
    expr.args.push(arg.expr);
    program = skipSpace(arg.rest);
    if (program[0] == ",") {
      program = skipSpace(program.slice(1));
    } else if (program[0] != ")") {

```

```

        throw new SyntaxError("Expected ',', ' or ')'");
    }
}
return parseApply(expr, program.slice(1));
}

```

If the next character in the program is not an opening parenthesis, this is not an application, and `parseApply` returns the expression it was given.

Otherwise, it skips the opening parenthesis and creates the syntax tree object for this application expression. It then recursively calls `parseExpression` to parse each argument until a closing parenthesis is found. The recursion is indirect, through `parseApply` and `parseExpression` calling each other.

Because an application expression can itself be applied (such as in `multiplier(2)(1)`), `parseApply` must, after it has parsed an application, call itself again to check whether another pair of parentheses follows.

This is all we need to parse Egg. We wrap it in a convenient `parse` function that verifies that it has reached the end of the input string after parsing the expression (an Egg program is a single expression), and that gives us the program's data structure.

```

function parse(program) {
  let {expr, rest} = parseExpression(program);
  if (skipSpace(rest).length > 0) {
    throw new SyntaxError("Unexpected text after program");
  }
  return expr;
}

console.log(parse("(+(a, 10)"));
// → {type: "apply",
//   operator: {type: "word", name: "+"},
//   args: [{type: "word", name: "a"},
//          {type: "value", value: 10}]}

```

It works! It doesn't give us very helpful information when it fails and doesn't store the line and column on which each expression starts, which might be helpful when reporting errors later, but it's good enough for our purposes.

THE EVALUATOR

What can we do with the syntax tree for a program? Run it, of course! And that is what the evaluator does. You give it a syntax tree and a scope object that associates names with values, and it will evaluate the expression that the tree represents and return the value that this produces.

```
const specialForms = Object.create(null);

function evaluate(expr, scope) {
  if (expr.type == "value") {
    return expr.value;
  } else if (expr.type == "word") {
    if (expr.name in scope) {
      return scope[expr.name];
    } else {
      throw new ReferenceError(
        `Undefined binding: ${expr.name}`);
    }
  } else if (expr.type == "apply") {
    let {operator, args} = expr;
    if (operator.type == "word" &&
        operator.name in specialForms) {
      return specialForms[operator.name](expr.args, scope);
    } else {
      let op = evaluate(operator, scope);
      if (typeof op == "function") {
        return op(...args.map(arg => evaluate(arg, scope)));
      } else {
        throw new TypeError("Applying a non-function.");
      }
    }
  }
}
```

The evaluator has code for each of the expression types. A literal value expression produces its value. (For example, the expression `100` just evaluates to the number 100.) For a binding, we must check whether it is actually defined in the scope and, if it is, fetch the binding's value.

Applications are more involved. If they are a special form, like `if`, we do not evaluate anything and pass the argument expressions, along with the scope, to the function that handles this form. If it is a normal call, we evaluate the operator, verify that it is a function, and call it with the evaluated arguments.

We use plain JavaScript function values to represent Egg's function values. We will come back to this [later](#), when the special form called `fun` is defined.

The recursive structure of `evaluate` resembles the similar structure of the parser, and both mirror the structure of the language itself. It would also be possible to integrate the parser with the evaluator and evaluate during parsing, but splitting them up this way makes the program clearer.

This is really all that is needed to interpret Egg. It is that simple. But without defining a few special forms and adding some useful values to the environment, you can't do much with this language yet.

SPECIAL FORMS

The `specialForms` object is used to define special syntax in Egg. It associates words with functions that evaluate such forms. It is currently empty. Let's add `if`.

```
specialForms.if = (args, scope) => {
  if (args.length !== 3) {
    throw new SyntaxError("Wrong number of args to if");
  } else if (evaluate(args[0], scope) !== false) {
    return evaluate(args[1], scope);
  } else {
    return evaluate(args[2], scope);
  }
};
```

Egg's `if` construct expects exactly three arguments. It will evaluate the first, and if the result isn't the value `false`, it will evaluate the second. Otherwise, the third gets evaluated. This `if` form is more similar to JavaScript's ternary `?:` operator than to JavaScript's `if`. It is an expression, not a statement, and it produces a value, namely, the result of the second or third argument.

Egg also differs from JavaScript in how it handles the condition value to `if`. It will not treat things like zero or the empty string as `false`, only the precise value `false`.

The reason we need to represent `if` as a special form, rather than a regular function, is that all arguments to functions are evaluated before the function is called, whereas `if` should evaluate only *either* its second or its third argument, depending on the value of the first.

The `while` form is similar.

```
specialForms.while = (args, scope) => {
```

```

    if (args.length !== 2) {
      throw new SyntaxError("Wrong number of args to while");
    }
    while (evaluate(args[0], scope) !== false) {
      evaluate(args[1], scope);
    }

    // Since undefined does not exist in Egg, we return false,
    // for lack of a meaningful result.
    return false;
  };

```

Another basic building block is `do`, which executes all its arguments from top to bottom. Its value is the value produced by the last argument.

```

specialForms.do = (args, scope) => {
  let value = false;
  for (let arg of args) {
    value = evaluate(arg, scope);
  }
  return value;
};

```

To be able to create bindings and give them new values, we also create a form called `define`. It expects a word as its first argument and an expression producing the value to assign to that word as its second argument. Since `define`, like everything, is an expression, it must return a value. We'll make it return the value that was assigned (just like JavaScript's `=` operator).

```

specialForms.define = (args, scope) => {
  if (args.length !== 2 || args[0].type !== "word") {
    throw new SyntaxError("Incorrect use of define");
  }
  let value = evaluate(args[1], scope);
  scope[args[0].name] = value;
  return value;
};

```

THE ENVIRONMENT

The scope accepted by `evaluate` is an object with properties whose names correspond to binding names and whose values correspond to the values those bindings are bound to. Let's define an object to represent the global scope.

To be able to use the `if` construct we just defined, we must have access to Boolean values. Since there are only two Boolean values, we do not need special syntax for them. We simply bind two names to the values `true` and `false` and use them.

```
const topScope = Object.create(null);

topScope.true = true;
topScope.false = false;
```

We can now evaluate a simple expression that negates a Boolean value.

```
let prog = parse(`if(true, false, true)`);
console.log(evaluate(prog, topScope));
// → false
```

To supply basic arithmetic and comparison operators, we will also add some function values to the scope. In the interest of keeping the code short, we'll use `Function` to synthesize a bunch of operator functions in a loop, instead of defining them individually.

```
for (let op of ["+", "-", "*", "/", "==", "<", ">"]) {
  topScope[op] = Function("a, b", `return a ${op} b;`);
}
```

A way to output values is also useful, so we'll wrap `console.log` in a function and call it `print`.

```
topScope.print = value => {
  console.log(value);
  return value;
};
```

That gives us enough elementary tools to write simple programs. The following function provides a convenient way to parse a program and run it in a fresh scope:


```
function run(program) {
  return evaluate(parse(program), Object.create(topScope));
}
```

We'll use object prototype chains to represent nested scopes so that the program can add bindings to its local scope without changing the top-level scope.

```
run(`
do(define(total, 0),
  define(count, 1),
  while(<(count, 11),
    do(define(total, +(total, count)),
      define(count, +(count, 1)))),
  print(total))
`);
// → 55
```

This is the program we've seen several times before, which computes the sum of the numbers 1 to 10, expressed in Egg. It is clearly uglier than the equivalent JavaScript program—but not bad for a language implemented in less than 150 lines of code.

FUNCTIONS

A programming language without functions is a poor programming language indeed.

Fortunately, it isn't hard to add a `fun` construct, which treats its last argument as the function's body and uses all arguments before that as the names of the function's parameters.

```
specialForms.fun = (args, scope) => {
  if (!args.length) {
    throw new SyntaxError("Functions need a body");
  }
  let body = args[args.length - 1];
  let params = args.slice(0, args.length - 1).map(expr => {
    if (expr.type !== "word") {
      throw new SyntaxError("Parameter names must be words");
    }
  });
  return expr.name;
});
```

```

    return function() {
      if (arguments.length !== params.length) {
        throw new TypeError("Wrong number of arguments");
      }
      let localScope = Object.create(scope);
      for (let i = 0; i < arguments.length; i++) {
        localScope[params[i]] = arguments[i];
      }
      return evaluate(body, localScope);
    };
  };
};

```

Functions in Egg get their own local scope. The function produced by the `fun` form creates this local scope and adds the argument bindings to it. It then evaluates the function body in this scope and returns the result.

```

run(`
do(define(plusOne, fun(a, +(a, 1))),
  print(plusOne(10)))
`);
// → 11

run(`
do(define(pow, fun(base, exp,
  if(==(exp, 0),
    1,
    *(base, pow(base, -(exp, 1)))))),
  print(pow(2, 10)))
`);
// → 1024

```

COMPILATION

What we have built is an interpreter. During evaluation, it acts directly on the representation of the program produced by the parser.

Compilation is the process of adding another step between the parsing and the running of a program, which transforms the program into something that can be evaluated more efficiently by doing as much work as possible in advance. For example, in well-designed languages it is obvious, for each use of a binding, which binding is being referred to, without actually running the program. This

can be used to avoid looking up the binding by name every time it is accessed, instead directly fetching it from some predetermined memory location.

Traditionally, compilation involves converting the program to machine code, the raw format that a computer's processor can execute. But any process that converts a program to a different representation can be thought of as compilation.

It would be possible to write an alternative evaluation strategy for Egg, one that first converts the program to a JavaScript program, uses `Function` to invoke the JavaScript compiler on it, and then runs the result. When done right, this would make Egg run very fast while still being quite simple to implement.

If you are interested in this topic and willing to spend some time on it, I encourage you to try to implement such a compiler as an exercise.

CHEATING

When we defined `if` and `while`, you probably noticed that they were more or less trivial wrappers around JavaScript's own `if` and `while`. Similarly, the values in Egg are just regular old JavaScript values.

If you compare the implementation of Egg, built on top of JavaScript, with the amount of work and complexity required to build a programming language directly on the raw functionality provided by a machine, the difference is huge. Regardless, this example ideally gave you an impression of the way programming languages work.

And when it comes to getting something done, cheating is more effective than doing everything yourself. Though the toy language in this chapter doesn't do anything that couldn't be done better in JavaScript, there *are* situations where writing small languages helps get real work done.

Such a language does not have to resemble a typical programming language. If JavaScript didn't come equipped with regular expressions, for example, you could write your own parser and evaluator for regular expressions.

Or imagine you are building a giant robotic dinosaur and need to program its behavior. JavaScript might not be the most effective way to do this. You might instead opt for a language that looks like this:

```
behavior walk
  perform when
    destination ahead
  actions
    move left-foot
```

```

    move right-foot

behavior attack
  perform when
    Godzilla in-view
  actions
    fire laser-eyes
    launch arm-rockets

```

This is what is usually called a *domain-specific language*, a language tailored to express a narrow domain of knowledge. Such a language can be more expressive than a general-purpose language because it is designed to describe exactly the things that need to be described in its domain, and nothing else.

EXERCISES

ARRAYS

Add support for arrays to Egg by adding the following three functions to the top scope: `array(...values)` to construct an array containing the argument values, `length(array)` to get an array's length, and `element(array, n)` to fetch the n^{th} element from an array.

CLOSURE

The way we have defined `fun` allows functions in Egg to reference the surrounding scope, allowing the function's body to use local values that were visible at the time the function was defined, just like JavaScript functions do.

The following program illustrates this: function `f` returns a function that adds its argument to `f`'s argument, meaning that it needs access to the local scope inside `f` to be able to use binding `a`.

```

run(`
do(define(f, fun(a, fun(b, +(a, b)))),
  print(f(4)(5)))
`);
// → 9

```

Go back to the definition of the `fun` form and explain which mechanism causes this to work.

COMMENTS

It would be nice if we could write comments in Egg. For example, whenever we find a hash sign (`#`), we could treat the rest of the line as a comment and ignore it, similar to `//` in JavaScript.

We do not have to make any big changes to the parser to support this. We can simply change `skipSpace` to skip comments as if they are whitespace so that all the points where `skipSpace` is called will now also skip comments. Make this change.

FIXING SCOPE

Currently, the only way to assign a binding a value is `define`. This construct acts as a way both to define new bindings and to give existing ones a new value.

This ambiguity causes a problem. When you try to give a nonlocal binding a new value, you will end up defining a local one with the same name instead. Some languages work like this by design, but I've always found it an awkward way to handle scope.

Add a special form `set`, similar to `define`, which gives a binding a new value, updating the binding in an outer scope if it doesn't already exist in the inner scope. If the binding is not defined at all, throw a `ReferenceError` (another standard error type).

The technique of representing scopes as simple objects, which has made things convenient so far, will get in your way a little at this point. You might want to use the `Object.getPrototypeOf` function, which returns the prototype of an object. Also remember that scopes do not derive from `Object.prototype`, so if you want to call `hasOwnProperty` on them, you have to use this clumsy expression:

```
Object.prototype.hasOwnProperty.call(scope, name);
```

“The dream behind the Web is of a common information space in which we communicate by sharing information. Its universality is essential: the fact that a hypertext link can point to anything, be it personal, local or global, be it draft or highly polished.”

—Tim Berners-Lee, *The World Wide Web: A very short personal history*

CHAPTER 13

JAVASCRIPT AND THE BROWSER

The next chapters of this book will talk about web browsers. Without web browsers, there would be no JavaScript. Or even if there were, no one would ever have paid any attention to it.

Web technology has been decentralized from the start, not just technically but also in the way it evolved. Various browser vendors have added new functionality in ad hoc and sometimes poorly thought-out ways, which then, sometimes, ended up being adopted by others—and finally set down as in standards.

This is both a blessing and a curse. On the one hand, it is empowering to not have a central party control a system but have it be improved by various parties working in loose collaboration (or occasionally open hostility). On the other hand, the haphazard way in which the Web was developed means that the resulting system is not exactly a shining example of internal consistency. Some parts of it are downright confusing and poorly conceived.

NETWORKS AND THE INTERNET

Computer networks have been around since the 1950s. If you put cables between two or more computers and allow them to send data back and forth through these cables, you can do all kinds of wonderful things.

And if connecting two machines in the same building allows us to do wonderful things, connecting machines all over the planet should be even better. The technology to start implementing this vision was developed in the 1980s, and the resulting network is called the *Internet*. It has lived up to its promise.

A computer can use this network to shoot bits at another computer. For any effective communication to arise out of this bit-shooting, the computers on both ends must know what the bits are supposed to represent. The meaning of any given sequence of bits depends entirely on the kind of thing that it is trying to express and on the encoding mechanism used.

A *network protocol* describes a style of communication over a network. There are protocols for sending email, for fetching email, for sharing files, and even

for controlling computers that happen to be infected by malicious software.

For example, the *Hypertext Transfer Protocol* (HTTP) is a protocol for retrieving named resources (chunks of information, such as web pages or pictures). It specifies that the side making the request should start with a line like this, naming the resource and the version of the protocol that it is trying to use:

```
GET /index.html HTTP/1.1
```

There are a lot more rules about the way the requester can include more information in the request and the way the other side, which returns the resource, packages up its content. We'll look at HTTP in a little more detail in [Chapter 18](#).

Most protocols are built on top of other protocols. HTTP treats the network as a streamlike device into which you can put bits and have them arrive at the correct destination in the correct order. As we saw in [Chapter 11](#), ensuring those things is already a rather difficult problem.

The *Transmission Control Protocol* (TCP) is a protocol that addresses this problem. All Internet-connected devices “speak” it, and most communication on the Internet is built on top of it.

A TCP connection works as follows: one computer must be waiting, or *listening*, for other computers to start talking to it. To be able to listen for different kinds of communication at the same time on a single machine, each listener has a number (called a *port*) associated with it. Most protocols specify which port should be used by default. For example, when we want to send an email using the SMTP protocol, the machine through which we send it is expected to be listening on port 25.

Another computer can then establish a connection by connecting to the target machine using the correct port number. If the target machine can be reached and is listening on that port, the connection is successfully created. The listening computer is called the *server*, and the connecting computer is called the *client*.

Such a connection acts as a two-way pipe through which bits can flow—the machines on both ends can put data into it. Once the bits are successfully transmitted, they can be read out again by the machine on the other side. This is a convenient model. You could say that TCP provides an abstraction of the network.

THE WEB

The *World Wide Web* (not to be confused with the Internet as a whole) is a set of protocols and formats that allow us to visit web pages in a browser. The “Web” part in the name refers to the fact that such pages can easily link to each other, thus connecting into a huge mesh that users can move through.

To become part of the Web, all you need to do is connect a machine to the Internet and have it listen on port 80 with the HTTP protocol so that other computers can ask it for documents.

Each document on the Web is named by a *Uniform Resource Locator* (URL), which looks something like this:

http://eloquentjavascript.net/13_browser.html

protocol	server	path

The first part tells us that this URL uses the HTTP protocol (as opposed to, for example, encrypted HTTP, which would be *https://*). Then comes the part that identifies which server we are requesting the document from. Last is a path string that identifies the specific document (or *resource*) we are interested in.

Machines connected to the Internet get an *IP address*, which is a number that can be used to send messages to that machine, and looks something like 149.210.142.219 or 2001:4860:4860::8888. But lists of more or less random numbers are hard to remember and awkward to type, so you can instead register a *domain name* for a specific address or set of addresses. I registered *eloquentjavascript.net* to point at the IP address of a machine I control and can thus use that domain name to serve web pages.

If you type this URL into your browser's address bar, the browser will try to retrieve and display the document at that URL. First, your browser has to find out what address *eloquentjavascript.net* refers to. Then, using the HTTP protocol, it will make a connection to the server at that address and ask for the resource */13_browser.html*. If all goes well, the server sends back a document, which your browser then displays on your screen.

HTML

HTML, which stands for *Hypertext Markup Language*, is the document format used for web pages. An HTML document contains text, as well as *tags* that

give structure to the text, describing things such as links, paragraphs, and headings.

A short HTML document might look like this:

```
<!doctype html>
<html>
  <head>
    <meta charset="utf-8">
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
    </body>
  </html>
```

This is what such a document would look like in the browser:

My home page

Hello, I am Marijn and this is my home page.

I also wrote a book! Read it [here](http://eloquentjavascript.net).

The tags, wrapped in angle brackets (< and >, the symbols for *less than* and *greater than*), provide information about the structure of the document. The other text is just plain text.

The document starts with <!doctype html>, which tells the browser to interpret the page as *modern* HTML, as opposed to various dialects that were in use in the past.

HTML documents have a head and a body. The head contains information *about* the document, and the body contains the document itself. In this case, the head declares that the title of this document is “My home page” and that it uses the UTF-8 encoding, which is a way to encode Unicode text as binary data. The document’s body contains a heading (<h1>, meaning “heading 1”—<h2> to <h6> produce subheadings) and two paragraphs (<p>).

Tags come in several forms. An element, such as the body, a paragraph, or a link, is started by an *opening tag* like <p> and ended by a *closing tag* like </p>. Some opening tags, such as the one for the link (<a>), contain extra information in the form of name="value" pairs. These are called *attributes*. In this case,

the destination of the link is indicated with `href="http://eloquentjavascript.net"`, where `href` stands for “hypertext reference”.

Some kinds of tags do not enclose anything and thus do not need to be closed. The metadata tag `<meta charset="utf-8">` is an example of this.

To be able to include angle brackets in the text of a document, even though they have a special meaning in HTML, yet another form of special notation has to be introduced. A plain opening angle bracket is written as `<` (“less than”), and a closing bracket is written as `>` (“greater than”). In HTML, an ampersand (&) character followed by a name or character code and a semicolon (;) is called an *entity* and will be replaced by the character it encodes.

This is analogous to the way backslashes are used in JavaScript strings. Since this mechanism gives ampersand characters a special meaning, too, they need to be escaped as `&`. Inside attribute values, which are wrapped in double quotes, `"` can be used to insert an actual quote character.

HTML is parsed in a remarkably error-tolerant way. When tags that should be there are missing, the browser reconstructs them. The way in which this is done has been standardized, and you can rely on all modern browsers to do it in the same way.

The following document will be treated just like the one shown previously:

```
<!doctype html>

<meta charset=utf-8>
<title>My home page</title>

<h1>My home page</h1>
<p>Hello, I am Marijn and this is my home page.
<p>I also wrote a book! Read it
  <a href=http://eloquentjavascript.net>here</a>.
```

The `<html>`, `<head>`, and `<body>` tags are gone completely. The browser knows that `<meta>` and `<title>` belong in the head and that `<h1>` means the body has started. Furthermore, I am no longer explicitly closing the paragraphs since opening a new paragraph or ending the document will close them implicitly. The quotes around the attribute values are also gone.

This book will usually omit the `<html>`, `<head>`, and `<body>` tags from examples to keep them short and free of clutter. But I *will* close tags and include quotes around attributes.

I will also usually omit the `doctype` and `charset` declaration. This is not to be taken as an encouragement to drop these from HTML documents. Browsers will often do ridiculous things when you forget them. You should consider the

doctype and the `charset` metadata to be implicitly present in examples, even when they are not actually shown in the text.

HTML AND JAVASCRIPT

In the context of this book, the most important HTML tag is `<script>`. This tag allows us to include a piece of JavaScript in a document.

```
<h1>Testing alert</h1>
<script>alert("hello!");</script>
```

Such a script will run as soon as its `<script>` tag is encountered while the browser reads the HTML. This page will pop up a dialog when opened—the `alert` function resembles `prompt`, in that it pops up a little window, but only shows a message without asking for input.

Including large programs directly in HTML documents is often impractical. The `<script>` tag can be given an `src` attribute to fetch a script file (a text file containing a JavaScript program) from a URL.

```
<h1>Testing alert</h1>
<script src="code/hello.js"></script>
```

The `code/hello.js` file included here contains the same program—`alert("hello!")`. When an HTML page references other URLs as part of itself—for example, an image file or a script—web browsers will retrieve them immediately and include them in the page.

A script tag must always be closed with `</script>`, even if it refers to a script file and doesn't contain any code. If you forget this, the rest of the page will be interpreted as part of the script.

You can load ES modules (see [Chapter 10](#)) in the browser by giving your script tag a `type="module"` attribute. Such modules can depend on other modules by using URLs relative to themselves as module names in `import` declarations.

Some attributes can also contain a JavaScript program. The `<button>` tag shown next (which shows up as a button) has an `onclick` attribute. The attribute's value will be run whenever the button is clicked.

```
<button onclick="alert('Boom!');">DO NOT PRESS</button>
```

Note that I had to use single quotes for the string in the `onclick` attribute because double quotes are already used to quote the whole attribute. I could also have used `"`;

IN THE SANDBOX

Running programs downloaded from the Internet is potentially dangerous. You do not know much about the people behind most sites you visit, and they do not necessarily mean well. Running programs by people who do not mean well is how you get your computer infected by viruses, your data stolen, and your accounts hacked.

Yet the attraction of the Web is that you can browse it without necessarily trusting all the pages you visit. This is why browsers severely limit the things a JavaScript program may do: it can't look at the files on your computer or modify anything not related to the web page it was embedded in.

Isolating a programming environment in this way is called *sandboxing*, the idea being that the program is harmlessly playing in a sandbox. But you should imagine this particular kind of sandbox as having a cage of thick steel bars over it so that the programs playing in it can't actually get out.

The hard part of sandboxing is allowing the programs enough room to be useful yet at the same time restricting them from doing anything dangerous. Lots of useful functionality, such as communicating with other servers or reading the content of the copy-paste clipboard, can also be used to do problematic, privacy-invading things.

Every now and then, someone comes up with a new way to circumvent the limitations of a browser and do something harmful, ranging from leaking minor private information to taking over the whole machine that the browser runs on. The browser developers respond by fixing the hole, and all is well again—until the next problem is discovered, and hopefully publicized, rather than secretly exploited by some government agency or mafia.

COMPATIBILITY AND THE BROWSER WARS

In the early stages of the Web, a browser called Mosaic dominated the market. After a few years, the balance shifted to Netscape, which was then, in turn, largely supplanted by Microsoft's Internet Explorer. At any point where a single browser was dominant, that browser's vendor would feel entitled to unilaterally invent new features for the Web. Since most users used the most

popular browser, websites would simply start using those features—never mind the other browsers.

This was the dark age of compatibility, often called the *browser wars*. Web developers were left with not one unified Web but two or three incompatible platforms. To make things worse, the browsers in use around 2003 were all full of bugs, and of course the bugs were different for each browser. Life was hard for people writing web pages.

Mozilla Firefox, a not-for-profit offshoot of Netscape, challenged Internet Explorer's position in the late 2000s. Because Microsoft was not particularly interested in staying competitive at the time, Firefox took a lot of market share away from it. Around the same time, Google introduced its Chrome browser, and Apple's Safari browser gained popularity, leading to a situation where there were four major players, rather than one.

The new players had a more serious attitude toward standards and better engineering practices, giving us less incompatibility and fewer bugs. Microsoft, seeing its market share crumble, came around and adopted these attitudes in its Edge browser, which replaces Internet Explorer. If you are starting to learn web development today, consider yourself lucky. The latest versions of the major browsers behave quite uniformly and have relatively few bugs.

“Too bad! Same old story! Once you’ve finished building your house you notice you’ve accidentally learned something that you really should have known—before you started.”

—Friedrich Nietzsche, *Beyond Good and Evil*

CHAPTER 14

THE DOCUMENT OBJECT MODEL

When you open a web page in your browser, the browser retrieves the page’s HTML text and parses it, much like the way our parser from [Chapter 12](#) parsed programs. The browser builds up a model of the document’s structure and uses this model to draw the page on the screen.

This representation of the document is one of the toys that a JavaScript program has available in its sandbox. It is a data structure that you can read or modify. It acts as a *live* data structure: when it’s modified, the page on the screen is updated to reflect the changes.

DOCUMENT STRUCTURE

You can imagine an HTML document as a nested set of boxes. Tags such as `<body>` and `</body>` enclose other tags, which in turn contain other tags or text. Here’s the example document from the [previous chapter](#):

```
<!doctype html>
<html>
  <head>
    <title>My home page</title>
  </head>
  <body>
    <h1>My home page</h1>
    <p>Hello, I am Marijn and this is my home page.</p>
    <p>I also wrote a book! Read it
      <a href="http://eloquentjavascript.net">here</a>.</p>
  </body>
</html>
```

This page has the following structure:



The data structure the browser uses to represent the document follows this shape. For each box, there is an object, which we can interact with to find out things such as what HTML tag it represents and which boxes and text it contains. This representation is called the *Document Object Model*, or DOM for short.

The global binding `document` gives us access to these objects. Its `documentElement` property refers to the object representing the `<html>` tag. Since every HTML document has a head and a body, it also has `head` and `body` properties, pointing at those elements.

TREES

Think back to the syntax trees from [Chapter 12](#) for a moment. Their structures are strikingly similar to the structure of a browser's document. Each *node* may refer to other nodes, *children*, which in turn may have their own children. This shape is typical of nested structures where elements can contain subelements that are similar to themselves.

We call a data structure a *tree* when it has a branching structure, has no cycles (a node may not contain itself, directly or indirectly), and has a single, well-defined *root*. In the case of the DOM, `document.documentElement` serves as the root.

Trees come up a lot in computer science. In addition to representing recursive structures such as HTML documents or programs, they are often used to maintain sorted sets of data because elements can usually be found or inserted

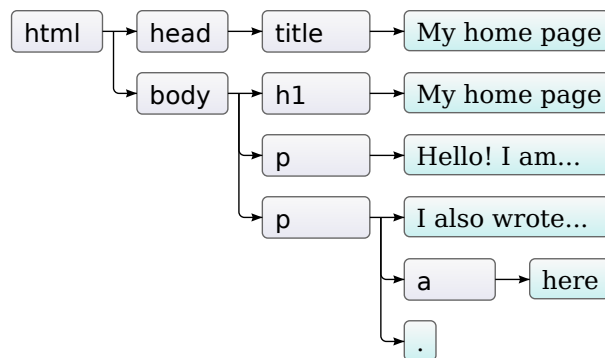
more efficiently in a tree than in a flat array.

A typical tree has different kinds of nodes. The syntax tree for [the Egg language](#) had identifiers, values, and application nodes. Application nodes may have children, whereas identifiers and values are *leaves*, or nodes without children.

The same goes for the DOM. Nodes for *elements*, which represent HTML tags, determine the structure of the document. These can have child nodes. An example of such a node is `document.body`. Some of these children can be leaf nodes, such as pieces of text or comment nodes.

Each DOM node object has a `nodeType` property, which contains a code (number) that identifies the type of node. Elements have code 1, which is also defined as the constant property `Node.ELEMENT_NODE`. Text nodes, representing a section of text in the document, get code 3 (`Node.TEXT_NODE`). Comments have code 8 (`Node.COMMENT_NODE`).

Another way to visualize our document tree is as follows:



The leaves are text nodes, and the arrows indicate parent-child relationships between nodes.

THE STANDARD

Using cryptic numeric codes to represent node types is not a very JavaScript-like thing to do. Later in this chapter, we'll see that other parts of the DOM interface also feel cumbersome and alien. The reason for this is that the DOM wasn't designed for just JavaScript. Rather, it tries to be a language-neutral interface that can be used in other systems as well—not just for HTML but also for XML, which is a generic data format with an HTML-like syntax.

This is unfortunate. Standards are often useful. But in this case, the advantage (cross-language consistency) isn't all that compelling. Having an interface that is properly integrated with the language you are using will save you more

time than having a familiar interface across languages.

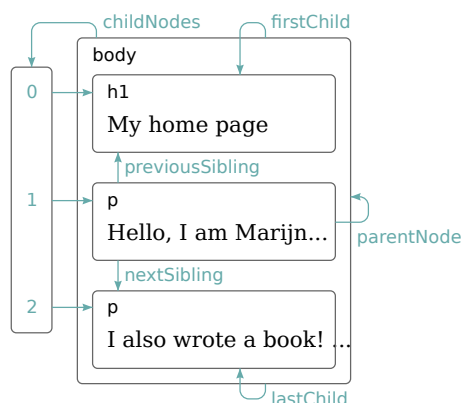
As an example of this poor integration, consider the `childNodes` property that element nodes in the DOM have. This property holds an array-like object, with a `length` property and properties labeled by numbers to access the child nodes. But it is an instance of the `NodeList` type, not a real array, so it does not have methods such as `slice` and `map`.

Then there are issues that are simply poor design. For example, there is no way to create a new node and immediately add children or attributes to it. Instead, you have to first create it and then add the children and attributes one by one, using side effects. Code that interacts heavily with the DOM tends to get long, repetitive, and ugly.

But these flaws aren't fatal. Since JavaScript allows us to create our own abstractions, it is possible to design improved ways to express the operations you are performing. Many libraries intended for browser programming come with such tools.

MOVING THROUGH THE TREE

DOM nodes contain a wealth of links to other nearby nodes. The following diagram illustrates these:



Although the diagram shows only one link of each type, every node has a `parentNode` property that points to the node it is part of, if any. Likewise, every element node (node type 1) has a `childNodes` property that points to an array-like object holding its children.

In theory, you could move anywhere in the tree using just these parent and child links. But JavaScript also gives you access to a number of additional convenience links. The `firstChild` and `lastChild` properties point to the first and last child elements or have the value `null` for nodes without children.

Similarly, `previousSibling` and `nextSibling` point to adjacent nodes, which are nodes with the same parent that appear immediately before or after the node itself. For a first child, `previousSibling` will be null, and for a last child, `nextSibling` will be null.

There's also the `children` property, which is like `childNodes` but contains only element (type 1) children, not other types of child nodes. This can be useful when you aren't interested in text nodes.

When dealing with a nested data structure like this one, recursive functions are often useful. The following function scans a document for text nodes containing a given string and returns true when it has found one:

```
function talksAbout(node, string) {
  if (node.nodeType == Node.ELEMENT_NODE) {
    for (let i = 0; i < node.childNodes.length; i++) {
      if (talksAbout(node.childNodes[i], string)) {
        return true;
      }
    }
    return false;
  } else if (node.nodeType == Node.TEXT_NODE) {
    return node.nodeValue.indexOf(string) > -1;
  }
}

console.log(talksAbout(document.body, "book"));
// → true
```

Because `childNodes` is not a real array, we cannot loop over it with `for/of` and have to run over the index range using a regular `for` loop or use `Array.from`.

The `nodeValue` property of a text node holds the string of text that it represents.

FINDING ELEMENTS

Navigating these links among parents, children, and siblings is often useful. But if we want to find a specific node in the document, reaching it by starting at `document.body` and following a fixed path of properties is a bad idea. Doing so bakes assumptions into our program about the precise structure of the document—a structure you might want to change later. Another complicating factor is that text nodes are created even for the whitespace between nodes. The example document's `<body>` tag does not have just three children (`<h1>`

and two `<p>` elements) but actually has seven: those three, plus the spaces before, after, and between them.

So if we want to get the `href` attribute of the link in that document, we don't want to say something like "Get the second child of the sixth child of the document body". It'd be better if we could say "Get the first link in the document". And we can.

```
let link = document.body.getElementsByTagName("a")[0];
console.log(link.href);
```

All element nodes have a `getElementsByTagName` method, which collects all elements with the given tag name that are descendants (direct or indirect children) of that node and returns them as an array-like object.

To find a specific *single* node, you can give it an `id` attribute and use `document.getElementById` instead.

```
<p>My ostrich Gertrude:</p>
<p></p>

<script>
  let ostrich = document.getElementById("gertrude");
  console.log(ostrich.src);
</script>
```

A third, similar method is `getElementsByClassName`, which, like `getElementsByTagName`, searches through the contents of an element node and retrieves all elements that have the given string in their `class` attribute.

CHANGING THE DOCUMENT

Almost everything about the DOM data structure can be changed. The shape of the document tree can be modified by changing parent-child relationships. Nodes have a `remove` method to remove them from their current parent node. To add a child node to an element node, we can use `appendChild`, which puts it at the end of the list of children, or `insertBefore`, which inserts the node given as the first argument before the node given as the second argument.

```
<p>One</p>
<p>Two</p>
<p>Three</p>
```

```

<script>
  let paragraphs = document.body.getElementsByTagName("p");
  document.body.insertBefore(paragraphs[2], paragraphs[0]);
</script>

```

A node can exist in the document in only one place. Thus, inserting paragraph *Three* in front of paragraph *One* will first remove it from the end of the document and then insert it at the front, resulting in *Three/One/Two*. All operations that insert a node somewhere will, as a side effect, cause it to be removed from its current position (if it has one).

The `replaceChild` method is used to replace a child node with another one. It takes as arguments two nodes: a new node and the node to be replaced. The replaced node must be a child of the element the method is called on. Note that both `replaceChild` and `insertBefore` expect the *new* node as their first argument.

CREATING NODES

Say we want to write a script that replaces all images (`` tags) in the document with the text held in their `alt` attributes, which specifies an alternative textual representation of the image.

This involves not only removing the images but adding a new text node to replace them. Text nodes are created with the `document.createTextNode` method.

```

<p>The  in the
  .</p>

<p><button onclick="replaceImages()">Replace</button></p>

<script>
  function replaceImages() {
    let images = document.body.getElementsByTagName("img");
    for (let i = images.length - 1; i >= 0; i--) {
      let image = images[i];
      if (image.alt) {
        let text = document.createTextNode(image.alt);
        image.parentNode.replaceChild(text, image);
      }
    }
  }
</script>

```

Given a string, `createTextNode` gives us a text node that we can insert into the document to make it show up on the screen.

The loop that goes over the images starts at the end of the list. This is necessary because the node list returned by a method like `getElementsByTagName` (or a property like `childNodes`) is *live*. That is, it is updated as the document changes. If we started from the front, removing the first image would cause the list to lose its first element so that the second time the loop repeats, where `i` is 1, it would stop because the length of the collection is now also 1.

If you want a *solid* collection of nodes, as opposed to a live one, you can convert the collection to a real array by calling `Array.from`.

```
let arrayish = {0: "one", 1: "two", length: 2};
let array = Array.from(arrayish);
console.log(array.map(s => s.toUpperCase()));
// → ["ONE", "TWO"]
```

To create element nodes, you can use the `document.createElement` method. This method takes a tag name and returns a new empty node of the given type.

The following example defines a utility `elt`, which creates an element node and treats the rest of its arguments as children to that node. This function is then used to add an attribution to a quote.

```
<blockquote id="quote">
  No book can ever be finished. While working on it we learn
  just enough to find it immature the moment we turn away
  from it.
</blockquote>

<script>
  function elt(type, ...children) {
    let node = document.createElement(type);
    for (let child of children) {
      if (typeof child !== "string") node.appendChild(child);
      else node.appendChild(document.createTextNode(child));
    }
    return node;
  }

  document.getElementById("quote").appendChild(
    elt("footer", "-",
      elt("strong", "Karl Popper"),
      ", preface to the second edition of ",
```

```
elt("em", "The Open Society and Its Enemies"),
", 1950"));
</script>
```

This is what the resulting document looks like:

No book can ever be finished. While working on it we learn
just enough to find it immature the moment we turn away
from it.
—**Karl Popper**, preface to the second edition of *The Open
Society and Its Enemies*, 1950

ATTRIBUTES

Some element attributes, such as `href` for links, can be accessed through a property of the same name on the element's DOM object. This is the case for most commonly used standard attributes.

But HTML allows you to set any attribute you want on nodes. This can be useful because it allows you to store extra information in a document. If you make up your own attribute names, though, such attributes will not be present as properties on the element's node. Instead, you have to use the `getAttribute` and `setAttribute` methods to work with them.

```
<p data-classified="secret">The launch code is 00000000.</p>
<p data-classified="unclassified">I have two feet.</p>

<script>
  let paras = document.getElementsByTagName("p");
  for (let para of Array.from(paras)) {
    if (para.getAttribute("data-classified") == "secret") {
      para.remove();
    }
  }
</script>
```

It is recommended to prefix the names of such made-up attributes with `data-` to ensure they do not conflict with any other attributes.

There is a commonly used attribute, `class`, which is a keyword in the JavaScript language. For historical reasons—some old JavaScript implementations could not handle property names that matched keywords—the property used to access this attribute is called `className`. You can also access it under its real name, `"class"`, by using the `getAttribute` and `setAttribute` methods.

LAYOUT

You may have noticed that different types of elements are laid out differently. Some, such as paragraphs (`<p>`) or headings (`<h1>`), take up the whole width of the document and are rendered on separate lines. These are called *block* elements. Others, such as links (`<a>`) or the `` element, are rendered on the same line with their surrounding text. Such elements are called *inline* elements.

For any given document, browsers are able to compute a layout, which gives each element a size and position based on its type and content. This layout is then used to actually draw the document.


The size and position of an element can be accessed from JavaScript. The `offsetWidth` and `offsetHeight` properties give you the space the element takes up in *pixels*. A pixel is the basic unit of measurement in the browser. It traditionally corresponds to the smallest dot that the screen can draw, but on modern displays, which can draw *very* small dots, that may no longer be the case, and a browser pixel may span multiple display dots.

Similarly, `clientWidth` and `clientHeight` give you the size of the space *inside* the element, ignoring border width.

```
<p style="border: 3px solid red">
  I'm boxed in
</p>

<script>
  let para = document.body.getElementsByTagName("p")[0];
  console.log("clientHeight:", para.clientHeight);
  console.log("offsetHeight:", para.offsetHeight);
</script>
```

Giving a paragraph a border causes a rectangle to be drawn around it.



The most effective way to find the precise position of an element on the screen is the `getBoundingClientRect` method. It returns an object with `top`, `bottom`, `left`, and `right` properties, indicating the pixel positions of the sides of the element relative to the top left of the screen. If you want them relative to the whole document, you must add the current scroll position, which you can find in the `pageXOffset` and `pageYOffset` bindings.

Laying out a document can be quite a lot of work. In the interest of

speed, browser engines do not immediately re-layout a document every time you change it but wait as long as they can. When a JavaScript program that changed the document finishes running, the browser will have to compute a new layout to draw the changed document to the screen. When a program *asks* for the position or size of something by reading properties such as `offsetHeight` or calling `getBoundingClientRect`, providing correct information also requires computing a layout.

A program that repeatedly alternates between reading DOM layout information and changing the DOM forces a lot of layout computations to happen and will consequently run very slowly. The following code is an example of this. It contains two different programs that build up a line of *X* characters 2,000 pixels wide and measures the time each one takes.

```
<p><span id="one"></span></p>
<p><span id="two"></span></p>

<script>
  function time(name, action) {
    let start = Date.now(); // Current time in milliseconds
    action();
    console.log(name, "took", Date.now() - start, "ms");
  }

  time("naive", () => {
    let target = document.getElementById("one");
    while (target.offsetWidth < 2000) {
      target.appendChild(document.createTextNode("X"));
    }
  });
  // → naive took 32 ms

  time("clever", function() {
    let target = document.getElementById("two");
    target.appendChild(document.createTextNode("XXXXX"));
    let total = Math.ceil(2000 / (target.offsetWidth / 5));
    target.firstChild.nodeValue = "X".repeat(total);
  });
  // → clever took 1 ms
</script>
```


STYLING

We have seen that different HTML elements are drawn differently. Some are displayed as blocks, others inline. Some add styling—`` makes its content bold, and `<a>` makes it blue and underlines it.

The way an `` tag shows an image or an `<a>` tag causes a link to be followed when it is clicked is strongly tied to the element type. But we can change the styling associated with an element, such as the text color or underline. Here is an example that uses the `style` property:

```
<p><a href=".">Normal link</a></p>
<p><a href="." style="color: green">Green link</a></p>
```

The second link will be green instead of the default link color.

[Normal link](#)

[Green link](#)

A style attribute may contain one or more *declarations*, which are a property (such as `color`) followed by a colon and a value (such as `green`). When there is more than one declaration, they must be separated by semicolons, as in `"color: red; border: none"`.

A lot of aspects of the document can be influenced by styling. For example, the `display` property controls whether an element is displayed as a block or an inline element.

```
This text is displayed <strong>inline</strong>,
<strong style="display: block">as a block</strong>, and
<strong style="display: none">not at all</strong>.
```

The `block` tag will end up on its own line since block elements are not displayed inline with the text around them. The last tag is not displayed at all—`display: none` prevents an element from showing up on the screen. This is a way to hide elements. It is often preferable to removing them from the document entirely because it makes it easy to reveal them again later.

This text is displayed **inline**,
as a block
, and .

JavaScript code can directly manipulate the style of an element through the

element's `style` property. This property holds an object that has properties for all possible style properties. The values of these properties are strings, which we can write to in order to change a particular aspect of the element's style.

```
<p id="para" style="color: purple">
  Nice text
</p>

<script>
  let para = document.getElementById("para");
  console.log(para.style.color);
  para.style.color = "magenta";
</script>
```

Some style property names contain hyphens, such as `font-family`. Because such property names are awkward to work with in JavaScript (you'd have to say `style["font-family"]`), the property names in the `style` object for such properties have their hyphens removed and the letters after them capitalized (`style.fontFamily`).

CASCADING STYLES

The styling system for HTML is called CSS, for *Cascading Style Sheets*. A *style sheet* is a set of rules for how to style elements in a document. It can be given inside a `<style>` tag.

```
<style>
  strong {
    font-style: italic;
    color: gray;
  }
</style>
<p>Now <strong>strong text</strong> is italic and gray.</p>
```

The *cascading* in the name refers to the fact that multiple such rules are combined to produce the final style for an element. In the example, the default styling for `` tags, which gives them `font-weight: bold`, is overlaid by the rule in the `<style>` tag, which adds `font-style` and `color`.

When multiple rules define a value for the same property, the most recently read rule gets a higher precedence and wins. So if the rule in the `<style>` tag included `font-weight: normal`, contradicting the default `font-weight` rule, the

text would be normal, *not* bold. Styles in a `style` attribute applied directly to the node have the highest precedence and always win.

It is possible to target things other than tag names in CSS rules. A rule for `.abc` applies to all elements with "abc" in their `class` attribute. A rule for `#xyz` applies to the element with an `id` attribute of "xyz" (which should be unique within the document).

```
.subtle {
  color: gray;
  font-size: 80%;
}
#header {
  background: blue;
  color: white;
}
/* p elements with id main and with classes a and b */
p#main.a.b {
  margin-bottom: 20px;
}
```

The precedence rule favoring the most recently defined rule applies only when the rules have the same *specificity*. A rule's specificity is a measure of how precisely it describes matching elements, determined by the number and kind (tag, class, or ID) of element aspects it requires. For example, a rule that targets `p.a` is more specific than rules that target `p` or just `.a` and would thus take precedence over them.

The notation `p > a ...{ }` applies the given styles to all `<a>` tags that are direct children of `<p>` tags. Similarly, `p a ...{ }` applies to all `<a>` tags inside `<p>` tags, whether they are direct or indirect children.

QUERY SELECTORS

We won't be using style sheets all that much in this book. Understanding them is helpful when programming in the browser, but they are complicated enough to warrant a separate book.

The main reason I introduced *selector* syntax—the notation used in style sheets to determine which elements a set of styles apply to—is that we can use this same mini-language as an effective way to find DOM elements.

The `querySelectorAll` method, which is defined both on the document object and on element nodes, takes a selector string and returns a `NodeList` containing all the elements that it matches.

```

<p>And if you go chasing
  <span class="animal">rabbits</span></p>
<p>And you know you're going to fall</p>
<p>Tell 'em a <span class="character">hookah smoking
  <span class="animal">caterpillar</span></span></p>
<p>Has given you the call</p>

<script>
  function count(selector) {
    return document.querySelectorAll(selector).length;
  }
  console.log(count("p"));           // All <p> elements
  // → 4
  console.log(count(".animal"));     // Class animal
  // → 2
  console.log(count("p .animal"));   // Animal inside of <p>
  // → 2
  console.log(count("p > .animal")); // Direct child of <p>
  // → 1
</script>

```

Unlike methods such as `getElementsByTagName`, the object returned by `querySelectorAll` is *not* live. It won't change when you change the document. It is still not a real array, though, so you still need to call `Array.from` if you want to treat it like one.

The `querySelector` method (without the `All` part) works in a similar way. This one is useful if you want a specific, single element. It will return only the first matching element or null when no element matches.

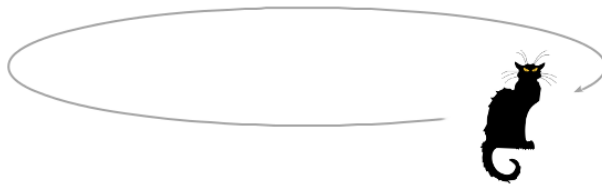
POSITIONING AND ANIMATING

The `position` style property influences layout in a powerful way. By default it has a value of `static`, meaning the element sits in its normal place in the document. When it is set to `relative`, the element still takes up space in the document, but now the `top` and `left` style properties can be used to move it relative to that normal place. When `position` is set to `absolute`, the element is removed from the normal document flow—that is, it no longer takes up space and may overlap with other elements. Also, its `top` and `left` properties can be used to absolutely position it relative to the top-left corner of the nearest enclosing element whose `position` property isn't `static`, or relative to the document if no such enclosing element exists.

We can use this to create an animation. The following document displays a picture of a cat that moves around in an ellipse:

```
<p style="text-align: center">
  
</p>
<script>
  let cat = document.querySelector("img");
  let angle = Math.PI / 2;
  function animate(time, lastTime) {
    if (lastTime != null) {
      angle += (time - lastTime) * 0.001;
    }
    cat.style.top = (Math.sin(angle) * 20) + "px";
    cat.style.left = (Math.cos(angle) * 200) + "px";
    requestAnimationFrame(newTime => animate(newTime, time));
  }
  requestAnimationFrame(animate);
</script>
```

The gray arrow shows the path along which the image moves.



Our picture is centered on the page and given a `position` of `relative`. We'll repeatedly update that picture's `top` and `left` styles to move it.

The script uses `requestAnimationFrame` to schedule the `animate` function to run whenever the browser is ready to repaint the screen. The `animate` function itself again calls `requestAnimationFrame` to schedule the next update. When the browser window (or tab) is active, this will cause updates to happen at a rate of about 60 per second, which tends to produce a good-looking animation.

If we just updated the DOM in a loop, the page would freeze, and nothing would show up on the screen. Browsers do not update their display while a JavaScript program is running, nor do they allow any interaction with the page. This is why we need `requestAnimationFrame`—it lets the browser know that we are done for now, and it can go ahead and do the things that browsers do, such as updating the screen and responding to user actions.

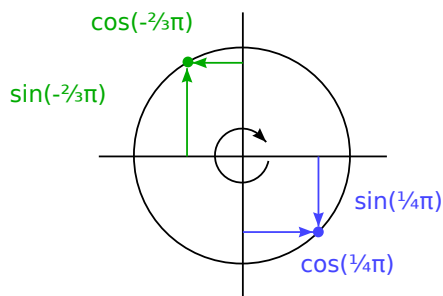
The animation function is passed the current time as an argument. To ensure that the motion of the cat per millisecond is stable, it bases the speed at which

the angle changes on the difference between the current time and the last time the function ran. If it just moved the angle by a fixed amount per step, the motion would stutter if, for example, another heavy task running on the same computer were to prevent the function from running for a fraction of a second.

Moving in circles is done using the trigonometry functions `Math.cos` and `Math.sin`. For those who aren't familiar with these, I'll briefly introduce them since we will occasionally use them in this book.

`Math.cos` and `Math.sin` are useful for finding points that lie on a circle around point (0,0) with a radius of one. Both functions interpret their argument as the position on this circle, with zero denoting the point on the far right of the circle, going clockwise until 2π (about 6.28) has taken us around the whole circle. `Math.cos` tells you the x-coordinate of the point that corresponds to the given position, and `Math.sin` yields the y-coordinate. Positions (or angles) greater than 2π or less than 0 are valid—the rotation repeats so that $a+2\pi$ refers to the same angle as a .

This unit for measuring angles is called radians—a full circle is 2π radians, similar to how it is 360 degrees when measuring in degrees. The constant π is available as `Math.PI` in JavaScript.



The cat animation code keeps a counter, `angle`, for the current angle of the animation and increments it every time the `animate` function is called. It can then use this angle to compute the current position of the image element. The `top` style is computed with `Math.sin` and multiplied by 20, which is the vertical radius of our ellipse. The `left` style is based on `Math.cos` and multiplied by 200 so that the ellipse is much wider than it is high.

Note that styles usually need *units*. In this case, we have to append "`px`" to the number to tell the browser that we are counting in pixels (as opposed to centimeters, "`ems`", or other units). This is easy to forget. Using numbers without units will result in your style being ignored—unless the number is 0, which always means the same thing, regardless of its unit.

SUMMARY

JavaScript programs may inspect and interfere with the document that the browser is displaying through a data structure called the DOM. This data structure represents the browser's model of the document, and a JavaScript program can modify it to change the visible document.

The DOM is organized like a tree, in which elements are arranged hierarchically according to the structure of the document. The objects representing elements have properties such as `parentNode` and `childNodes`, which can be used to navigate through this tree.

The way a document is displayed can be influenced by *styling*, both by attaching styles to nodes directly and by defining rules that match certain nodes. There are many different style properties, such as `color` or `display`. JavaScript code can manipulate an element's style directly through its `style` property.

EXERCISES

BUILD A TABLE

An HTML table is built with the following tag structure:

```
<table>
  <tr>
    <th>name</th>
    <th>height</th>
    <th>place</th>
  </tr>
  <tr>
    <td>Kilimanjaro</td>
    <td>5895</td>
    <td>Tanzania</td>
  </tr>
</table>
```

For each *row*, the `<table>` tag contains a `<tr>` tag. Inside of these `<tr>` tags, we can put cell elements: either heading cells (`<th>`) or regular cells (`<td>`).

Given a data set of mountains, an array of objects with `name`, `height`, and `place` properties, generate the DOM structure for a table that enumerates the objects. It should have one column per key and one row per object, plus a header row with `<th>` elements at the top, listing the column names.

Write this so that the columns are automatically derived from the objects,

by taking the property names of the first object in the data.

Add the resulting table to the element with an `id` attribute of "mountains" so that it becomes visible in the document.

Once you have this working, right-align cells that contain number values by setting their `style.textAlign` property to "right".

ELEMENTS BY TAG NAME

The `document.getElementsByTagName` method returns all child elements with a given tag name. Implement your own version of this as a function that takes a node and a string (the tag name) as arguments and returns an array containing all descendant element nodes with the given tag name.

To find the tag name of an element, use its `nodeName` property. But note that this will return the tag name in all uppercase. Use the `toLowerCase` or `toUpperCase` string methods to compensate for this.

THE CAT'S HAT

Extend the cat animation defined [earlier](#) so that both the cat and his hat (``) orbit at opposite sides of the ellipse.

Or make the hat circle around the cat. Or alter the animation in some other interesting way.

To make positioning multiple objects easier, it is probably a good idea to switch to absolute positioning. This means that `top` and `left` are counted relative to the top left of the document. To avoid using negative coordinates, which would cause the image to move outside of the visible page, you can add a fixed number of pixels to the position values.

“You have power over your mind—not outside events. Realize this, and you will find strength.”

—Marcus Aurelius, *Meditations*

CHAPTER 15

HANDLING EVENTS

Some programs work with direct user input, such as mouse and keyboard actions. That kind of input isn’t available as a well-organized data structure—it comes in piece by piece, in real time, and the program is expected to respond to it as it happens.

EVENT HANDLERS

Imagine an interface where the only way to find out whether a key on the keyboard is being pressed is to read the current state of that key. To be able to react to keypresses, you would have to constantly read the key’s state so that you’d catch it before it’s released again. It would be dangerous to perform other time-intensive computations since you might miss a keypress.

Some primitive machines do handle input like that. A step up from this would be for the hardware or operating system to notice the keypress and put it in a queue. A program can then periodically check the queue for new events and react to what it finds there.

Of course, it has to remember to look at the queue, and to do it often, because any time between the key being pressed and the program noticing the event will cause the software to feel unresponsive. This approach is called *polling*. Most programmers prefer to avoid it.

A better mechanism is for the system to actively notify our code when an event occurs. Browsers do this by allowing us to register functions as *handlers* for specific events.

```
<p>Click this document to activate the handler.</p>
<script>
  window.addEventListener("click", () => {
    console.log("You knocked?");
  });
</script>
```

The window binding refers to a built-in object provided by the browser. It represents the browser window that contains the document. Calling its `addEventListener` method registers the second argument to be called whenever the event described by its first argument occurs.

EVENTS AND DOM NODES

Each browser event handler is registered in a context. In the previous example we called `addEventListener` on the window object to register a handler for the whole window. Such a method can also be found on DOM elements and some other types of objects. Event listeners are called only when the event happens in the context of the object they are registered on.

```
<button>Click me</button>
<p>No handler here.</p>
<script>
  let button = document.querySelector("button");
  button.addEventListener("click", () => {
    console.log("Button clicked.");
  });
</script>
```

That example attaches a handler to the button node. Clicks on the button cause that handler to run, but clicks on the rest of the document do not.

Giving a node an `onclick` attribute has a similar effect. This works for most types of events—you can attach a handler through the attribute whose name is the event name with `on` in front of it.

But a node can have only one `onclick` attribute, so you can register only one handler per node that way. The `addEventListener` method allows you to add any number of handlers so that it is safe to add handlers even if there is already another handler on the element.

The `removeEventListener` method, called with arguments similar to `addEventListener`, removes a handler.

```
<button>Act-once button</button>
<script>
  let button = document.querySelector("button");
  function once() {
    console.log("Done.");
    button.removeEventListener("click", once);
  }
  button.addEventListener("click", once);
</script>
```

```
</script>
```

The function given to `removeEventListener` has to be the same function value that was given to `addEventListener`. So, to unregister a handler, you'll want to give the function a name (once, in the example) to be able to pass the same function value to both methods.

EVENT OBJECTS

Though we have ignored it so far, event handler functions are passed an argument: the *event object*. This object holds additional information about the event. For example, if we want to know *which* mouse button was pressed, we can look at the event object's `button` property.

```
<button>Click me any way you want</button>
<script>
  let button = document.querySelector("button");
  button.addEventListener("mousedown", event => {
    if (event.button == 0) {
      console.log("Left button");
    } else if (event.button == 1) {
      console.log("Middle button");
    } else if (event.button == 2) {
      console.log("Right button");
    }
  });
</script>
```

The information stored in an event object differs per type of event. We'll discuss different types later in the chapter. The object's `type` property always holds a string identifying the event (such as `"click"` or `"mousedown"`).

PROPAGATION

For most event types, handlers registered on nodes with children will also receive events that happen in the children. If a button inside a paragraph is clicked, event handlers on the paragraph will also see the click event.

But if both the paragraph and the button have a handler, the more specific handler—the one on the button—gets to go first. The event is said to *propagate* outward, from the node where it happened to that node's parent node and on

to the root of the document. Finally, after all handlers registered on a specific node have had their turn, handlers registered on the whole window get a chance to respond to the event.

At any point, an event handler can call the `stopPropagation` method on the event object to prevent handlers further up from receiving the event. This can be useful when, for example, you have a button inside another clickable element and you don't want clicks on the button to activate the outer element's click behavior.

The following example registers "mousedown" handlers on both a button and the paragraph around it. When clicked with the right mouse button, the handler for the button calls `stopPropagation`, which will prevent the handler on the paragraph from running. When the button is clicked with another mouse button, both handlers will run.

```
<p>A paragraph with a <button>button</button>.</p>
<script>
  let para = document.querySelector("p");
  let button = document.querySelector("button");
  para.addEventListener("mousedown", () => {
    console.log("Handler for paragraph.");
  });
  button.addEventListener("mousedown", event => {
    console.log("Handler for button.");
    if (event.button == 2) event.stopPropagation();
  });
</script>
```

Most event objects have a `target` property that refers to the node where they originated. You can use this property to ensure that you're not accidentally handling something that propagated up from a node you do not want to handle.

It is also possible to use the `target` property to cast a wide net for a specific type of event. For example, if you have a node containing a long list of buttons, it may be more convenient to register a single click handler on the outer node and have it use the `target` property to figure out whether a button was clicked, rather than register individual handlers on all of the buttons.

```
<button>A</button>
<button>B</button>
<button>C</button>
<script>
  document.body.addEventListener("click", event => {
    if (event.target.nodeName == "BUTTON") {
```

```
        console.log("Clicked", event.target.textContent);
    }
});
</script>
```

DEFAULT ACTIONS

Many events have a default action associated with them. If you click a link, you will be taken to the link's target. If you press the down arrow, the browser will scroll the page down. If you right-click, you'll get a context menu. And so on.

For most types of events, the JavaScript event handlers are called *before* the default behavior takes place. If the handler doesn't want this normal behavior to happen, typically because it has already taken care of handling the event, it can call the `preventDefault` method on the event object.

This can be used to implement your own keyboard shortcuts or context menu. It can also be used to obnoxiously interfere with the behavior that users expect. For example, here is a link that cannot be followed:

```
<a href="https://developer.mozilla.org/">MDN</a>
<script>
    let link = document.querySelector("a");
    link.addEventListener("click", event => {
        console.log("Nope.");
        event.preventDefault();
    });
</script>
```

Try not to do such things unless you have a really good reason to. It'll be unpleasant for people who use your page when expected behavior is broken.

Depending on the browser, some events can't be intercepted at all. On Chrome, for example, the keyboard shortcut to close the current tab (CONTROL-W or COMMAND-W) cannot be handled by JavaScript.

KEY EVENTS

When a key on the keyboard is pressed, your browser fires a "keydown" event. When it is released, you get a "keyup" event.

```

<p>This page turns violet when you hold the V key.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == "v") {
      document.body.style.background = "violet";
    }
  });
  window.addEventListener("keyup", event => {
    if (event.key == "v") {
      document.body.style.background = "";
    }
  });
</script>

```

Despite its name, "keydown" fires not only when the key is physically pushed down. When a key is pressed and held, the event fires again every time the key *repeats*. Sometimes you have to be careful about this. For example, if you add a button to the DOM when a key is pressed and remove it again when the key is released, you might accidentally add hundreds of buttons when the key is held down longer.

The example looked at the `key` property of the event object to see which key the event is about. This property holds a string that, for most keys, corresponds to the thing that pressing that key would type. For special keys such as ENTER, it holds a string that names the key ("Enter", in this case). If you hold SHIFT while pressing a key, that might also influence the name of the key—"v" becomes "V", and "1" may become "!", if that is what pressing SHIFT-1 produces on your keyboard.

Modifier keys such as SHIFT, CONTROL, ALT, and META (COMMAND on Mac) generate key events just like normal keys. But when looking for key combinations, you can also find out whether these keys are held down by looking at the `shiftKey`, `ctrlKey`, `altKey`, and `metaKey` properties of keyboard and mouse events.

```

<p>Press Control-Space to continue.</p>
<script>
  window.addEventListener("keydown", event => {
    if (event.key == " " && event.ctrlKey) {
      console.log("Continuing!");
    }
  });
</script>

```

The DOM node where a key event originates depends on the element that has focus when the key is pressed. Most nodes cannot have focus unless you give them a `tabindex` attribute, but things like links, buttons, and form fields can. We'll come back to form fields in [Chapter 18](#). When nothing in particular has focus, `document.body` acts as the target node of key events.

When the user is typing text, using key events to figure out what is being typed is problematic. Some platforms, most notably the virtual keyboard on Android phones, don't fire key events. But even when you have an old-fashioned keyboard, some types of text input don't match key presses in a straightforward way, such as *input method editor* (IME) software used by people whose scripts don't fit on a keyboard, where multiple key strokes are combined to create characters.

To notice when something was typed, elements that you can type into, such as the `<input>` and `<textarea>` tags, fire "input" events whenever the user changes their content. To get the actual content that was typed, it is best to directly read it from the focused field. [Chapter 18](#) will show how.

POINTER EVENTS

There are currently two widely used ways to point at things on a screen: mice (including devices that act like mice, such as touchpads and trackballs) and touchscreens. These produce different kinds of events.

MOUSE CLICKS

Pressing a mouse button causes a number of events to fire. The "mousedown" and "mouseup" events are similar to "keydown" and "keyup" and fire when the button is pressed and released. These happen on the DOM nodes that are immediately below the mouse pointer when the event occurs.

After the "mouseup" event, a "click" event fires on the most specific node that contained both the press and the release of the button. For example, if I press down the mouse button on one paragraph and then move the pointer to another paragraph and release the button, the "click" event will happen on the element that contains both those paragraphs.

If two clicks happen close together, a "dblclick" (double-click) event also fires, after the second click event.

To get precise information about the place where a mouse event happened, you can look at its `clientX` and `clientY` properties, which contain the event's coordinates (in pixels) relative to the top-left corner of the window, or `pageX`

and `pageY`, which are relative to the top-left corner of the whole document (which may be different when the window has been scrolled).

The following implements a primitive drawing program. Every time you click the document, it adds a dot under your mouse pointer. See [Chapter 19](#) for a less primitive drawing program.

```
<style>
  body {
    height: 200px;
    background: beige;
  }
  .dot {
    height: 8px; width: 8px;
    border-radius: 4px; /* rounds corners */
    background: blue;
    position: absolute;
  }
</style>
<script>
  window.addEventListener("click", event => {
    let dot = document.createElement("div");
    dot.className = "dot";
    dot.style.left = (event.pageX - 4) + "px";
    dot.style.top = (event.pageY - 4) + "px";
    document.body.appendChild(dot);
  });
</script>
```

MOUSE MOTION

Every time the mouse pointer moves, a "mousemove" event is fired. This event can be used to track the position of the mouse. A common situation in which this is useful is when implementing some form of mouse-dragging functionality.

As an example, the following program displays a bar and sets up event handlers so that dragging to the left or right on this bar makes it narrower or wider:

```
<p>Drag the bar to change its width:</p>
<div style="background: orange; width: 60px; height: 20px">
</div>
<script>
  let lastX; // Tracks the last observed mouse X position
  let bar = document.querySelector("div");
```



```

bar.addEventListener("mousedown", event => {
  if (event.button == 0) {
    lastX = event.clientX;
    window.addEventListener("mousemove", moved);
    event.preventDefault(); // Prevent selection
  }
});

function moved(event) {
  if (event.buttons == 0) {
    window.removeEventListener("mousemove", moved);
  } else {
    let dist = event.clientX - lastX;
    let newWidth = Math.max(10, bar.offsetWidth + dist);
    bar.style.width = newWidth + "px";
    lastX = event.clientX;
  }
}
</script>

```

The resulting page looks like this:

Drag the bar to change its width:



Note that the "mousemove" handler is registered on the whole window. Even if the mouse goes outside of the bar during resizing, as long as the button is held we still want to update its size.

We must stop resizing the bar when the mouse button is released. For that, we can use the `buttons` property (note the plural), which tells us about the buttons that are currently held down. When this is zero, no buttons are down. When buttons are held, its value is the sum of the codes for those buttons—the left button has code 1, the right button 2, and the middle one 4. That way, you can check whether a given button is pressed by taking the remainder of the value of `buttons` and its code.

Note that the order of these codes is different from the one used by `button`, where the middle button came before the right one. As mentioned, consistency isn't really a strong point of the browser's programming interface.

TOUCH EVENTS

The style of graphical browser that we use was designed with mouse interfaces in mind, at a time where touchscreens were rare. To make the Web “work” on early touchscreen phones, browsers for those devices pretended, to a certain extent, that touch events were mouse events. If you tap your screen, you’ll get “mousedown”, “mouseup”, and “click” events.

But this illusion isn’t very robust. A touchscreen works differently from a mouse: it doesn’t have multiple buttons, you can’t track the finger when it isn’t on the screen (to simulate “mousemove”), and it allows multiple fingers to be on the screen at the same time.

Mouse events cover touch interaction only in straightforward cases—if you add a “click” handler to a button, touch users will still be able to use it. But something like the resizable bar in the previous example does not work on a touchscreen.

There are specific event types fired by touch interaction. When a finger starts touching the screen, you get a “touchstart” event. When it is moved while touching, “touchmove” events fire. Finally, when it stops touching the screen, you’ll see a “touchend” event.

Because many touchscreens can detect multiple fingers at the same time, these events don’t have a single set of coordinates associated with them. Rather, their event objects have a `touches` property, which holds an array-like object of points, each of which has its own `clientX`, `clientY`, `pageX`, and `pageY` properties.

You could do something like this to show red circles around every touching finger:

```
<style>
  dot { position: absolute; display: block;
        border: 2px solid red; border-radius: 50px;
        height: 100px; width: 100px; }
</style>
<p>Touch this page</p>
<script>
  function update(event) {
    for (let dot; dot = document.querySelector("dot");) {
      dot.remove();
    }
    for (let i = 0; i < event.touches.length; i++) {
      let {pageX, pageY} = event.touches[i];
      let dot = document.createElement("dot");
      dot.style.left = (pageX - 50) + "px";
      dot.style.top = (pageY - 50) + "px";
      document.body.appendChild(dot);
    }
  }
</script>
```

```

    }
  }
  window.addEventListener("touchstart", update);
  window.addEventListener("touchmove", update);
  window.addEventListener("touchend", update);
</script>

```

You'll often want to call `preventDefault` in touch event handlers to override the browser's default behavior (which may include scrolling the page on swiping) and to prevent the mouse events from being fired, for which you may *also* have a handler.

SCROLL EVENTS

Whenever an element is scrolled, a "scroll" event is fired on it. This has various uses, such as knowing what the user is currently looking at (for disabling off-screen animations or sending spy reports to your evil headquarters) or showing some indication of progress (by highlighting part of a table of contents or showing a page number).

The following example draws a progress bar above the document and updates it to fill up as you scroll down:

```

<style>
  #progress {
    border-bottom: 2px solid blue;
    width: 0;
    position: fixed;
    top: 0; left: 0;
  }
</style>
<div id="progress"></div>
<script>
  // Create some content
  document.body.appendChild(document.createTextNode(
    "supercalifragilisticexpialidocious ".repeat(1000)));

  let bar = document.querySelector("#progress");
  window.addEventListener("scroll", () => {
    let max = document.body.scrollHeight - innerHeight;
    bar.style.width = `${(pageYOffset / max) * 100}%`;
  });
</script>

```

Giving an element a `position` of `fixed` acts much like an `absolute` position but also prevents it from scrolling along with the rest of the document. The effect is to make our progress bar stay at the top. Its width is changed to indicate the current progress. We use `%`, rather than `px`, as a unit when setting the width so that the element is sized relative to the page width.

The global `innerHeight` binding gives us the height of the window, which we have to subtract from the total scrollable height—you can't keep scrolling when you hit the bottom of the document. There's also an `innerWidth` for the window width. By dividing `pageYOffset`, the current scroll position, by the maximum scroll position and multiplying by 100, we get the percentage for the progress bar.

Calling `preventDefault` on a scroll event does not prevent the scrolling from happening. In fact, the event handler is called only *after* the scrolling takes place.

FOCUS EVENTS

When an element gains focus, the browser fires a `"focus"` event on it. When it loses focus, the element gets a `"blur"` event.

Unlike the events discussed earlier, these two events do not propagate. A handler on a parent element is not notified when a child element gains or loses focus.

The following example displays help text for the text field that currently has focus:

```
<p>Name: <input type="text" data-help="Your full name"></p>
<p>Age: <input type="text" data-help="Your age in years"></p>
<p id="help"></p>

<script>
  let help = document.querySelector("#help");
  let fields = document.querySelectorAll("input");
  for (let field of Array.from(fields)) {
    field.addEventListener("focus", event => {
      let text = event.target.getAttribute("data-help");
      help.textContent = text;
    });
    field.addEventListener("blur", event => {
      help.textContent = "";
    });
  }
}
```

`</script>`

This screenshot shows the help text for the age field.

Name:

Age:

Age in years

The window object will receive "focus" and "blur" events when the user moves from or to the browser tab or window in which the document is shown.

LOAD EVENT

When a page finishes loading, the "load" event fires on the window and the document body objects. This is often used to schedule initialization actions that require the whole document to have been built. Remember that the content of `<script>` tags is run immediately when the tag is encountered. This may be too soon, for example when the script needs to do something with parts of the document that appear after the `<script>` tag.

Elements such as images and script tags that load an external file also have a "load" event that indicates the files they reference were loaded. Like the focus-related events, loading events do not propagate.

When a page is closed or navigated away from (for example, by following a link), a "beforeunload" event fires. The main use of this event is to prevent the user from accidentally losing work by closing a document. If you prevent the default behavior on this event *and* set the `returnValue` property on the event object to a string, the browser will show the user a dialog asking if they really want to leave the page. That dialog might include your string, but because some malicious sites try to use these dialogs to confuse people into staying on their page to look at dodgy weight loss ads, most browsers no longer display them.

EVENTS AND THE EVENT LOOP

In the context of the event loop, as discussed in [Chapter 11](#), browser event handlers behave like other asynchronous notifications. They are scheduled when the event occurs but must wait for other scripts that are running to finish before they get a chance to run.

The fact that events can be processed only when nothing else is running means that, if the event loop is tied up with other work, any interaction with the page (which happens through events) will be delayed until there's time to process it. So if you schedule too much work, either with long-running event handlers or with lots of short-running ones, the page will become slow and cumbersome to use.

For cases where you *really* do want to do some time-consuming thing in the background without freezing the page, browsers provide something called *web workers*. A worker is a JavaScript process that runs alongside the main script, on its own timeline.

Imagine that squaring a number is a heavy, long-running computation that we want to perform in a separate thread. We could write a file called `code/squareworker.js` that responds to messages by computing a square and sending a message back.

```
addEventListener("message", event => {
  postMessage(event.data * event.data);
});
```

To avoid the problems of having multiple threads touching the same data, workers do not share their global scope or any other data with the main script's environment. Instead, you have to communicate with them by sending messages back and forth.

This code spawns a worker running that script, sends it a few messages, and outputs the responses.

```
let squareWorker = new Worker("code/squareworker.js");
squareWorker.addEventListener("message", event => {
  console.log("The worker responded:", event.data);
});
squareWorker.postMessage(10);
squareWorker.postMessage(24);
```

The `postMessage` function sends a message, which will cause a "message" event to fire in the receiver. The script that created the worker sends and receives messages through the `Worker` object, whereas the worker talks to the script that created it by sending and listening directly on its global scope. Only values that can be represented as JSON can be sent as messages—the other side will receive a *copy* of them, rather than the value itself.

TIMERS

We saw the `setTimeout` function in [Chapter 11](#). It schedules another function to be called later, after a given number of milliseconds.

Sometimes you need to cancel a function you have scheduled. This is done by storing the value returned by `setTimeout` and calling `clearTimeout` on it.

```
let bombTimer = setTimeout(() => {
  console.log("BOOM!");
}, 500);

if (Math.random() < 0.5) { // 50% chance
  console.log("Defused.");
  clearTimeout(bombTimer);
}
```

The `cancelAnimationFrame` function works in the same way as `clearTimeout`—calling it on a value returned by `requestAnimationFrame` will cancel that frame (assuming it hasn't already been called).

A similar set of functions, `setInterval` and `clearInterval`, are used to set timers that should *repeat* every *X* milliseconds.

```
let ticks = 0;
let clock = setInterval(() => {
  console.log("tick", ticks++);
  if (ticks == 10) {
    clearInterval(clock);
    console.log("stop.");
  }
}, 200);
```

DEBOUNCING

Some types of events have the potential to fire rapidly, many times in a row (the "mousemove" and "scroll" events, for example). When handling such events, you must be careful not to do anything too time-consuming or your handler will take up so much time that interaction with the document starts to feel slow.

If you do need to do something nontrivial in such a handler, you can use `setTimeout` to make sure you are not doing it too often. This is usually called

debouncing the event. There are several slightly different approaches to this.

In the first example, we want to react when the user has typed something, but we don't want to do it immediately for every input event. When they are typing quickly, we just want to wait until a pause occurs. Instead of immediately performing an action in the event handler, we set a timeout. We also clear the previous timeout (if any) so that when events occur close together (closer than our timeout delay), the timeout from the previous event will be canceled.

```
<textarea>Type something here...</textarea>
<script>
  let textarea = document.querySelector("textarea");
  let timeout;
  textarea.addEventListener("input", () => {
    clearTimeout(timeout);
    timeout = setTimeout(() => console.log("Typed!"), 500);
  });
</script>
```

Giving an undefined value to `clearTimeout` or calling it on a timeout that has already fired has no effect. Thus, we don't have to be careful about when to call it, and we simply do so for every event.

We can use a slightly different pattern if we want to space responses so that they're separated by at least a certain length of time but want to fire them *during* a series of events, not just afterward. For example, we might want to respond to "mousemove" events by showing the current coordinates of the mouse but only every 250 milliseconds.

```
<script>
  let scheduled = null;
  window.addEventListener("mousemove", event => {
    if (!scheduled) {
      setTimeout(() => {
        document.body.textContent =
          `Mouse at ${scheduled.pageX}, ${scheduled.pageY}`;
        scheduled = null;
      }, 250);
    }
    scheduled = event;
  });
</script>
```


SUMMARY

Event handlers make it possible to detect and react to events happening in our web page. The `addEventListener` method is used to register such a handler.

Each event has a type (`"keydown"`, `"focus"`, and so on) that identifies it. Most events are called on a specific DOM element and then *propagate* to that element's ancestors, allowing handlers associated with those elements to handle them.

When an event handler is called, it is passed an event object with additional information about the event. This object also has methods that allow us to stop further propagation (`stopPropagation`) and prevent the browser's default handling of the event (`preventDefault`).

Pressing a key fires `"keydown"` and `"keyup"` events. Pressing a mouse button fires `"mousedown"`, `"mouseup"`, and `"click"` events. Moving the mouse fires `"mousemove"` events. Touchscreen interaction will result in `"touchstart"`, `"touchmove"`, and `"touchend"` events.

Scrolling can be detected with the `"scroll"` event, and focus changes can be detected with the `"focus"` and `"blur"` events. When the document finishes loading, a `"load"` event fires on the window.

EXERCISES

BALLOON

Write a page that displays a balloon (using the balloon emoji, 🎈). When you press the up arrow, it should inflate (grow) 10 percent, and when you press the down arrow, it should deflate (shrink) 10 percent.

You can control the size of text (emoji are text) by setting the `font-size` CSS property (`style.fontSize`) on its parent element. Remember to include a unit in the value—for example, pixels (`10px`).

The key names of the arrow keys are `"ArrowUp"` and `"ArrowDown"`. Make sure the keys change only the balloon, without scrolling the page.

When that works, add a feature where, if you blow up the balloon past a certain size, it explodes. In this case, exploding means that it is replaced with an 💣 emoji, and the event handler is removed (so that you can't inflate or deflate the explosion).

MOUSE TRAIL

In JavaScript’s early days, which was the high time of gaudy home pages with lots of animated images, people came up with some truly inspiring ways to use the language.

One of these was the *mouse trail*—a series of elements that would follow the mouse pointer as you moved it across the page.

In this exercise, I want you to implement a mouse trail. Use absolutely positioned `<div>` elements with a fixed size and background color (refer to the code in the “Mouse Clicks” section for an example). Create a bunch of such elements and, when the mouse moves, display them in the wake of the mouse pointer.

There are various possible approaches here. You can make your solution as simple or as complex as you want. A simple solution to start with is to keep a fixed number of trail elements and cycle through them, moving the next one to the mouse’s current position every time a “`mousemove`” event occurs.

TABS

Tabbed panels are widely used in user interfaces. They allow you to select an interface panel by choosing from a number of tabs “sticking out” above an element.

In this exercise you must implement a simple tabbed interface. Write a function, `asTabs`, that takes a DOM node and creates a tabbed interface showing the child elements of that node. It should insert a list of `<button>` elements at the top of the node, one for each child element, containing text retrieved from the `data-tabname` attribute of the child. All but one of the original children should be hidden (given a `display` style of `none`). The currently visible node can be selected by clicking the buttons.

When that works, extend it to style the button for the currently selected tab differently so that it is obvious which tab is selected.

“All reality is a game.”

—Iain Banks, *The Player of Games*

CHAPTER 16

PROJECT: A PLATFORM GAME

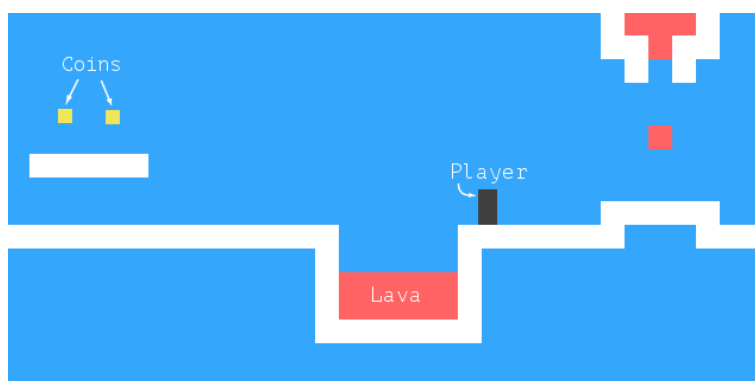
Much of my initial fascination with computers, like that of many nerdy kids, had to do with computer games. I was drawn into the tiny simulated worlds that I could manipulate and in which stories (sort of) unfolded—more, I suppose, because of the way I projected my imagination into them than because of the possibilities they actually offered.

I don’t wish a career in game programming on anyone. Much like the music industry, the discrepancy between the number of eager young people wanting to work in it and the actual demand for such people creates a rather unhealthy environment. But writing games for fun is amusing.

This chapter will walk through the implementation of a small platform game. Platform games (or “jump and run” games) are games that expect the player to move a figure through a world, which is usually two-dimensional and viewed from the side, while jumping over and onto things.

THE GAME

Our game will be roughly based on *Dark Blue* (www.lessmilk.com/games/10) by Thomas Palef. I chose that game because it is both entertaining and minimalist and because it can be built without too much code. It looks like this:



The dark box represents the player, whose task is to collect the yellow boxes

(coins) while avoiding the red stuff (lava). A level is completed when all coins have been collected.

The player can walk around with the left and right arrow keys and can jump with the up arrow. Jumping is a specialty of this game character. It can reach several times its own height and can change direction in midair. This may not be entirely realistic, but it helps give the player the feeling of being in direct control of the on-screen avatar.

The game consists of a static background, laid out like a grid, with the moving elements overlaid on that background. Each field on the grid is either empty, solid, or lava. The moving elements are the player, coins, and certain pieces of lava. The positions of these elements are not constrained to the grid—their coordinates may be fractional, allowing smooth motion.

THE TECHNOLOGY

We will use the browser DOM to display the game, and we'll read user input by handling key events.

The screen- and keyboard-related code is only a small part of the work we need to do to build this game. Since everything looks like colored boxes, drawing is uncomplicated: we create DOM elements and use styling to give them a background color, size, and position.

We can represent the background as a table since it is an unchanging grid of squares. The free-moving elements can be overlaid using absolutely positioned elements.

In games and other programs that should animate graphics and respond to user input without noticeable delay, efficiency is important. Although the DOM was not originally designed for high-performance graphics, it is actually better at this than you would expect. You saw some animations in [Chapter 14](#). On a modern machine, a simple game like this performs well, even if we don't worry about optimization very much.

In the [next chapter](#), we will explore another browser technology, the `<canvas>` tag, which provides a more traditional way to draw graphics, working in terms of shapes and pixels rather than DOM elements.

LEVELS

We'll want a human-readable, human-editable way to specify levels. Since it is okay for everything to start out on a grid, we could use big strings in which

each character represents an element—either a part of the background grid or a moving element.

The plan for a small level might look like this:

```
let simpleLevelPlan = `
.....
..#.....#..
..#.....=#..
..#.....o.o...#..
..#.@.....####...#..
..####.....#..
.....#+++++++#+..
.....#####..
.....`;
```

Periods are empty space, hash (#) characters are walls, and plus signs are lava. The player's starting position is the at sign (@). Every O character is a coin, and the equal sign (=) at the top is a block of lava that moves back and forth horizontally.

We'll support two additional kinds of moving lava: the pipe character (|) creates vertically moving blobs, and v indicates *dripping* lava—vertically moving lava that doesn't bounce back and forth but only moves down, jumping back to its start position when it hits the floor.

A whole game consists of multiple levels that the player must complete. A level is completed when all coins have been collected. If the player touches lava, the current level is restored to its starting position, and the player may try again.

READING A LEVEL

The following class stores a level object. Its argument should be the string that defines the level.

```
class Level {
  constructor(plan) {
    let rows = plan.trim().split("\n").map(l => [...l]);
    this.height = rows.length;
    this.width = rows[0].length;
    this.startActors = [];

    this.rows = rows.map((row, y) => {
      return row.map((ch, x) => {
```

```

        let type = levelChars[ch];
        if (typeof type == "string") return type;
        this.startActors.push(
            type.create(new Vec(x, y), ch));
        return "empty";
    });
});
}
}

```

The `trim` method is used to remove whitespace at the start and end of the plan string. This allows our example plan to start with a newline so that all the lines are directly below each other. The remaining string is split on newline characters, and each line is spread into an array, producing arrays of characters.

So `rows` holds an array of arrays of characters, the rows of the plan. We can derive the level's width and height from these. But we must still separate the moving elements from the background grid. We'll call moving elements *actors*. They'll be stored in an array of objects. The background will be an array of arrays of strings, holding field types such as "empty", "wall", or "lava".

To create these arrays, we map over the rows and then over their content. Remember that `map` passes the array index as a second argument to the mapping function, which tells us the x- and y-coordinates of a given character. Positions in the game will be stored as pairs of coordinates, with the top left being 0,0 and each background square being 1 unit high and wide.

To interpret the characters in the plan, the `Level` constructor uses the `levelChars` object, which maps background elements to strings and actor characters to classes. When `type` is an actor class, its static `create` method is used to create an object, which is added to `startActors`, and the mapping function returns "empty" for this background square.

The position of the actor is stored as a `Vec` object. This is a two-dimensional vector, an object with `x` and `y` properties, as seen in the exercises of [Chapter 6](#).

As the game runs, actors will end up in different places or even disappear entirely (as coins do when collected). We'll use a `State` class to track the state of a running game.

```

class State {
  constructor(level, actors, status) {
    this.level = level;
    this.actors = actors;
    this.status = status;
  }
}

```

```

static start(level) {
  return new State(level, level.startActors, "playing");
}

get player() {
  return this.actors.find(a => a.type == "player");
}
}

```

The `status` property will switch to "lost" or "won" when the game has ended.

This is again a persistent data structure—updating the game state creates a new state and leaves the old one intact.

ACTORS

Actor objects represent the current position and state of a given moving element in our game. All actor objects conform to the same interface. Their `pos` property holds the coordinates of the element's top-left corner, and their `size` property holds its size.

Then they have an `update` method, which is used to compute their new state and position after a given time step. It simulates the thing the actor does—moving in response to the arrow keys for the player and bouncing back and forth for the lava—and returns a new, updated actor object.

A `type` property contains a string that identifies the type of the actor—"player", "coin", or "lava". This is useful when drawing the game—the look of the rectangle drawn for an actor is based on its type.

Actor classes have a static `create` method that is used by the `Level` constructor to create an actor from a character in the level plan. It is given the coordinates of the character and the character itself, which is needed because the `Lava` class handles several different characters.

This is the `Vec` class that we'll use for our two-dimensional values, such as the position and size of actors.

```

class Vec {
  constructor(x, y) {
    this.x = x; this.y = y;
  }
  plus(other) {
    return new Vec(this.x + other.x, this.y + other.y);
  }
}

```

```

    }
    times(factor) {
        return new Vec(this.x * factor, this.y * factor);
    }
}

```

The `times` method scales a vector by a given number. It will be useful when we need to multiply a speed vector by a time interval to get the distance traveled during that time.

The different types of actors get their own classes since their behavior is very different. Let's define these classes. We'll get to their `update` methods later.

The player class has a property `speed` that stores its current speed to simulate momentum and gravity.

```

class Player {
    constructor(pos, speed) {
        this.pos = pos;
        this.speed = speed;
    }

    get type() { return "player"; }

    static create(pos) {
        return new Player(pos.plus(new Vec(0, -0.5)),
                           new Vec(0, 0));
    }
}

Player.prototype.size = new Vec(0.8, 1.5);

```

Because a player is one-and-a-half squares high, its initial position is set to be half a square above the position where the `@` character appeared. This way, its bottom aligns with the bottom of the square it appeared in.

The `size` property is the same for all instances of `Player`, so we store it on the prototype rather than on the instances themselves. We could have used a getter like `type`, but that would create and return a new `Vec` object every time the property is read, which would be wasteful. (Strings, being immutable, don't have to be re-created every time they are evaluated.)

When constructing a Lava actor, we need to initialize the object differently depending on the character it is based on. Dynamic lava moves along at its current speed until it hits an obstacle. At that point, if it has a `reset` property,

it will jump back to its start position (dripping). If it does not, it will invert its speed and continue in the other direction (bouncing).

The `create` method looks at the character that the `Level` constructor passes and creates the appropriate lava actor.

```
class Lava {
  constructor(pos, speed, reset) {
    this.pos = pos;
    this.speed = speed;
    this.reset = reset;
  }

  get type() { return "lava"; }

  static create(pos, ch) {
    if (ch == "=") {
      return new Lava(pos, new Vec(2, 0));
    } else if (ch == "|") {
      return new Lava(pos, new Vec(0, 2));
    } else if (ch == "v") {
      return new Lava(pos, new Vec(0, 3), pos);
    }
  }
}

Lava.prototype.size = new Vec(1, 1);
```

Coin actors are relatively simple. They mostly just sit in their place. But to liven up the game a little, they are given a “wobble”, a slight vertical back-and-forth motion. To track this, a coin object stores a base position as well as a wobble property that tracks the phase of the bouncing motion. Together, these determine the coin’s actual position (stored in the `pos` property).

```
class Coin {
  constructor(pos, basePos, wobble) {
    this.pos = pos;
    this.basePos = basePos;
    this.wobble = wobble;
  }

  get type() { return "coin"; }

  static create(pos) {
    let basePos = pos.plus(new Vec(0.2, 0.1));
```

```

        return new Coin(basePos, basePos,
                        Math.random() * Math.PI * 2);
    }
}

Coin.prototype.size = new Vec(0.6, 0.6);

```

In [Chapter 14](#), we saw that `Math.sin` gives us the y-coordinate of a point on a circle. That coordinate goes back and forth in a smooth waveform as we move along the circle, which makes the sine function useful for modeling a wavy motion.

To avoid a situation where all coins move up and down synchronously, the starting phase of each coin is randomized. The *phase* of `Math.sin`'s wave, the width of a wave it produces, is 2π . We multiply the value returned by `Math.random` by that number to give the coin a random starting position on the wave.

We can now define the `levelChars` object that maps plan characters to either background grid types or actor classes.

```

const levelChars = {
  ".": "empty", "#": "wall", "+": "lava",
  "@": Player, "o": Coin,
  "=": Lava, "|": Lava, "v": Lava
};

```

That gives us all the parts needed to create a `Level` instance.

```

let simpleLevel = new Level(simpleLevelPlan);
console.log(`${simpleLevel.width} by ${simpleLevel.height}`);
// → 22 by 9

```

The task ahead is to display such levels on the screen and to model time and motion inside them.

ENCAPSULATION AS A BURDEN

Most of the code in this chapter does not worry about encapsulation very much for two reasons. First, encapsulation takes extra effort. It makes programs bigger and requires additional concepts and interfaces to be introduced. Since there is only so much code you can throw at a reader before their eyes glaze

over, I've made an effort to keep the program small.

Second, the various elements in this game are so closely tied together that if the behavior of one of them changed, it is unlikely that any of the others would be able to stay the same. Interfaces between the elements would end up encoding a lot of assumptions about the way the game works. This makes them a lot less effective—whenever you change one part of the system, you still have to worry about the way it impacts the other parts because their interfaces wouldn't cover the new situation.

Some *cutting points* in a system lend themselves well to separation through rigorous interfaces, but others don't. Trying to encapsulate something that isn't a suitable boundary is a sure way to waste a lot of energy. When you are making this mistake, you'll usually notice that your interfaces are getting awkwardly large and detailed and that they need to be changed often, as the program evolves.

There is one thing that we *will* encapsulate, and that is the drawing subsystem. The reason for this is that we'll display the same game in a different way in the [next chapter](#). By putting the drawing behind an interface, we can load the same game program there and plug in a new display module.

DRAWING

The encapsulation of the drawing code is done by defining a *display* object, which displays a given level and state. The display type we define in this chapter is called `DOMDisplay` because it uses DOM elements to show the level.

We'll be using a style sheet to set the actual colors and other fixed properties of the elements that make up the game. It would also be possible to directly assign to the elements' `style` property when we create them, but that would produce more verbose programs.

The following helper function provides a succinct way to create an element and give it some attributes and child nodes:

```
function elt(name, attrs, ...children) {
  let dom = document.createElement(name);
  for (let attr of Object.keys(attrs)) {
    dom.setAttribute(attr, attrs[attr]);
  }
  for (let child of children) {
    dom.appendChild(child);
  }
  return dom;
}
```

A display is created by giving it a parent element to which it should append itself and a level object.

```
class DOMDisplay {
  constructor(parent, level) {
    this.dom = elt("div", {class: "game"}, drawGrid(level));
    this.actorLayer = null;
    parent.appendChild(this.dom);
  }

  clear() { this.dom.remove(); }
}
```

The level's background grid, which never changes, is drawn once. Actors are redrawn every time the display is updated with a given state. The `actorLayer` property will be used to track the element that holds the actors so that they can be easily removed and replaced.

Our coordinates and sizes are tracked in grid units, where a size or distance of 1 means one grid block. When setting pixel sizes, we will have to scale these coordinates up—everything in the game would be ridiculously small at a single pixel per square. The `scale` constant gives the number of pixels that a single unit takes up on the screen.

```
const scale = 20;

function drawGrid(level) {
  return elt("table", {
    class: "background",
    style: `width: ${level.width * scale}px`
  }, ...level.rows.map(row =>
    elt("tr", {style: `height: ${scale}px`},
      ...row.map(type => elt("td", {class: type})))
  ));
}
```

As mentioned, the background is drawn as a `<table>` element. This nicely corresponds to the structure of the `rows` property of the level—each row of the grid is turned into a table row (`<tr>` element). The strings in the grid are used as class names for the table cell (`<td>`) elements. The spread (triple dot) operator is used to pass arrays of child nodes to `elt` as separate arguments.

The following CSS makes the table look like the background we want:

```

.background    { background: rgb(52, 166, 251);
                  table-layout: fixed;
                  border-spacing: 0;                }
.background td { padding: 0;                        }
.lava          { background: rgb(255, 100, 100); }
.wall          { background: white;               }

```

Some of these (`table-layout`, `border-spacing`, and `padding`) are used to suppress unwanted default behavior. We don't want the layout of the table to depend upon the contents of its cells, and we don't want space between the table cells or padding inside them.

The `background` rule sets the background color. CSS allows colors to be specified both as words (`white`) or with a format such as `rgb(R, G, B)`, where the red, green, and blue components of the color are separated into three numbers from 0 to 255. So, in `rgb(52, 166, 251)`, the red component is 52, green is 166, and blue is 251. Since the blue component is the largest, the resulting color will be bluish. You can see that in the `.lava` rule, the first number (red) is the largest.

We draw each actor by creating a DOM element for it and setting that element's position and size based on the actor's properties. The values have to be multiplied by `scale` to go from game units to pixels.

```

function drawActors(actors) {
  return elt("div", {}, ...actors.map(actor => {
    let rect = elt("div", {class: `actor ${actor.type}`});
    rect.style.width = `${actor.size.x * scale}px`;
    rect.style.height = `${actor.size.y * scale}px`;
    rect.style.left = `${actor.pos.x * scale}px`;
    rect.style.top = `${actor.pos.y * scale}px`;
    return rect;
  }));
}

```

To give an element more than one class, we separate the class names by spaces. In the CSS code shown next, the `actor` class gives the actors their absolute position. Their type name is used as an extra class to give them a color. We don't have to define the `lava` class again because we're reusing the class for the lava grid squares we defined earlier.

```

.actor { position: absolute; }
.coin  { background: rgb(241, 229, 89); }
.player { background: rgb(64, 64, 64); }

```

The `syncState` method is used to make the display show a given state. It first removes the old actor graphics, if any, and then redraws the actors in their new positions. It may be tempting to try to reuse the DOM elements for actors, but to make that work, we would need a lot of additional bookkeeping to associate actors with DOM elements and to make sure we remove elements when their actors vanish. Since there will typically be only a handful of actors in the game, redrawing all of them is not expensive.

```
DOMDisplay.prototype.syncState = function(state) {  
  if (this.actorLayer) this.actorLayer.remove();  
  this.actorLayer = drawActors(state.actors);  
  this.dom.appendChild(this.actorLayer);  
  this.dom.className = `game ${state.status}`;  
  this.scrollPlayerIntoView(state);  
};
```

By adding the level's current status as a class name to the wrapper, we can style the player actor slightly differently when the game is won or lost by adding a CSS rule that takes effect only when the player has an ancestor element with a given class.

```
.lost .player {  
  background: rgb(160, 64, 64);  
}  
.won .player {  
  box-shadow: -4px -7px 8px white, 4px -7px 8px white;  
}
```

After touching lava, the player's color turns dark red, suggesting scorching. When the last coin has been collected, we add two blurred white shadows—one to the top left and one to the top right—to create a white halo effect.

We can't assume that the level always fits in the *viewport*—the element into which we draw the game. That is why the `scrollPlayerIntoView` call is needed. It ensures that if the level is protruding outside the viewport, we scroll that viewport to make sure the player is near its center. The following CSS gives the game's wrapping DOM element a maximum size and ensures that anything that sticks out of the element's box is not visible. We also give it a relative position so that the actors inside it are positioned relative to the level's top-left corner.

```
.game {
  overflow: hidden;
  max-width: 600px;
  max-height: 450px;
  position: relative;
}
```

In the `scrollPlayerIntoView` method, we find the player's position and update the wrapping element's scroll position. We change the scroll position by manipulating that element's `scrollLeft` and `scrollTop` properties when the player is too close to the edge.

```
DOMDisplay.prototype.scrollPlayerIntoView = function(state) {
  let width = this.dom.clientWidth;
  let height = this.dom.clientHeight;
  let margin = width / 3;

  // The viewport
  let left = this.dom.scrollLeft, right = left + width;
  let top = this.dom.scrollTop, bottom = top + height;

  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5))
    .times(scale);

  if (center.x < left + margin) {
    this.dom.scrollLeft = center.x - margin;
  } else if (center.x > right - margin) {
    this.dom.scrollLeft = center.x + margin - width;
  }
  if (center.y < top + margin) {
    this.dom.scrollTop = center.y - margin;
  } else if (center.y > bottom - margin) {
    this.dom.scrollTop = center.y + margin - height;
  }
};
```

The way the player's center is found shows how the methods on our `Vec` type allow computations with objects to be written in a relatively readable way. To find the actor's center, we add its position (its top-left corner) and half its size. That is the center in level coordinates, but we need it in pixel coordinates, so we then multiply the resulting vector by our display scale.

Next, a series of checks verifies that the player position isn't outside of the