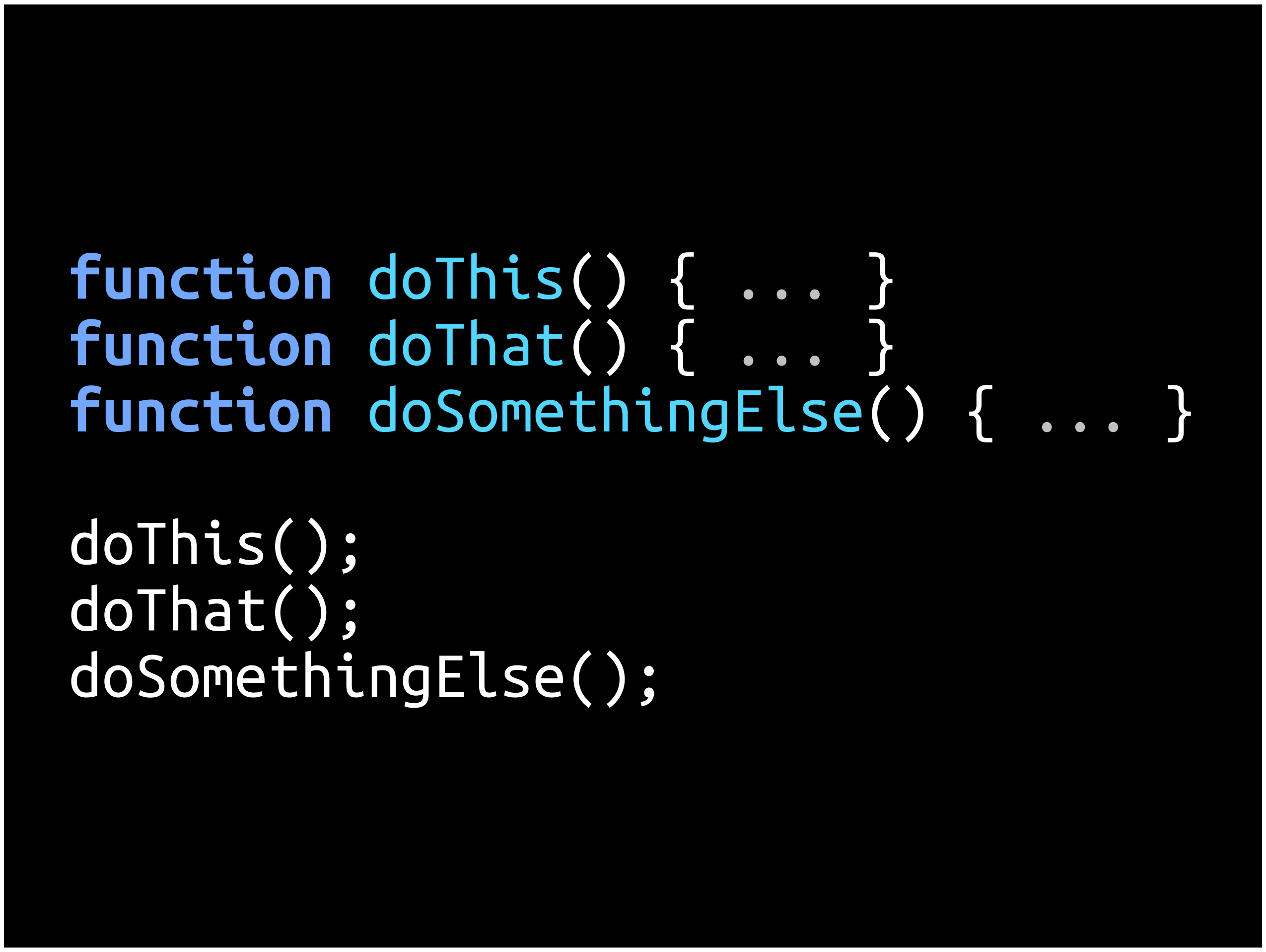
Asynchronicity

An Introduction to Asynchronous Programming

A Lecture by
Matt Patenaude

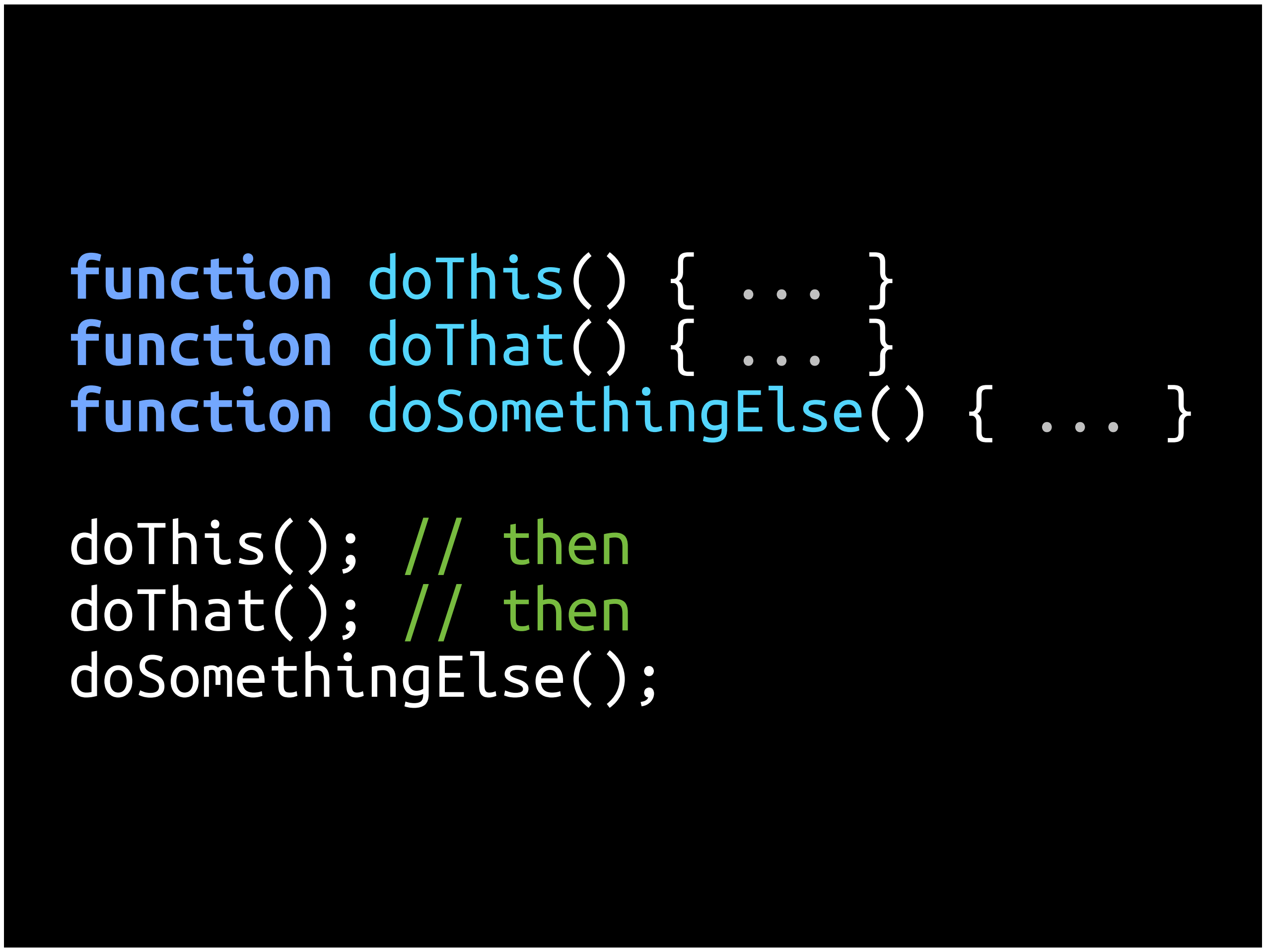
@mattpat

It's a Police joke. Get it? Get it?



```
function doThis() { ... }
function doThat() { ... }
function doSomethingElse() { ... }

doThis();
doThat();
doSomethingElse();
```

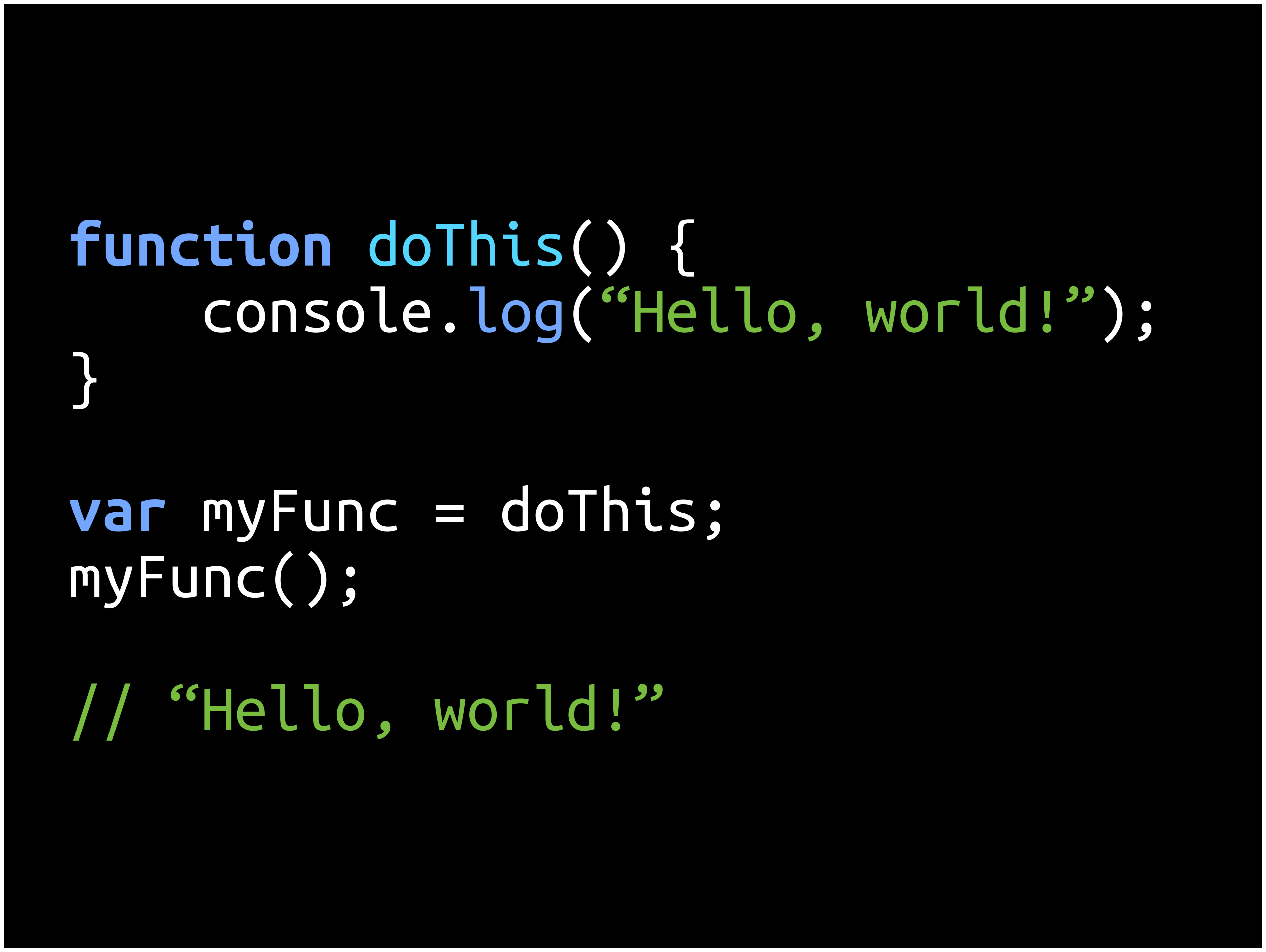
A simple JavaScript example with three functions (doThis, doThat, and doSomethingElse), which we then call in order. A very important, subtle point about this code, though, is that it actually says…



```
function doThis() { ... }
function doThat() { ... }
function doSomethingElse() { ... }

doThis(); // then
doThat(); // then
doSomethingElse();
```

… this: doThis, THEN doThat, THEN doSomethingElse. There's something nice and comforting about this programming model: it's predictable, it's simple, and it's intuitive. This is what's known as synchronous programming. It does, however, have its obstacles.

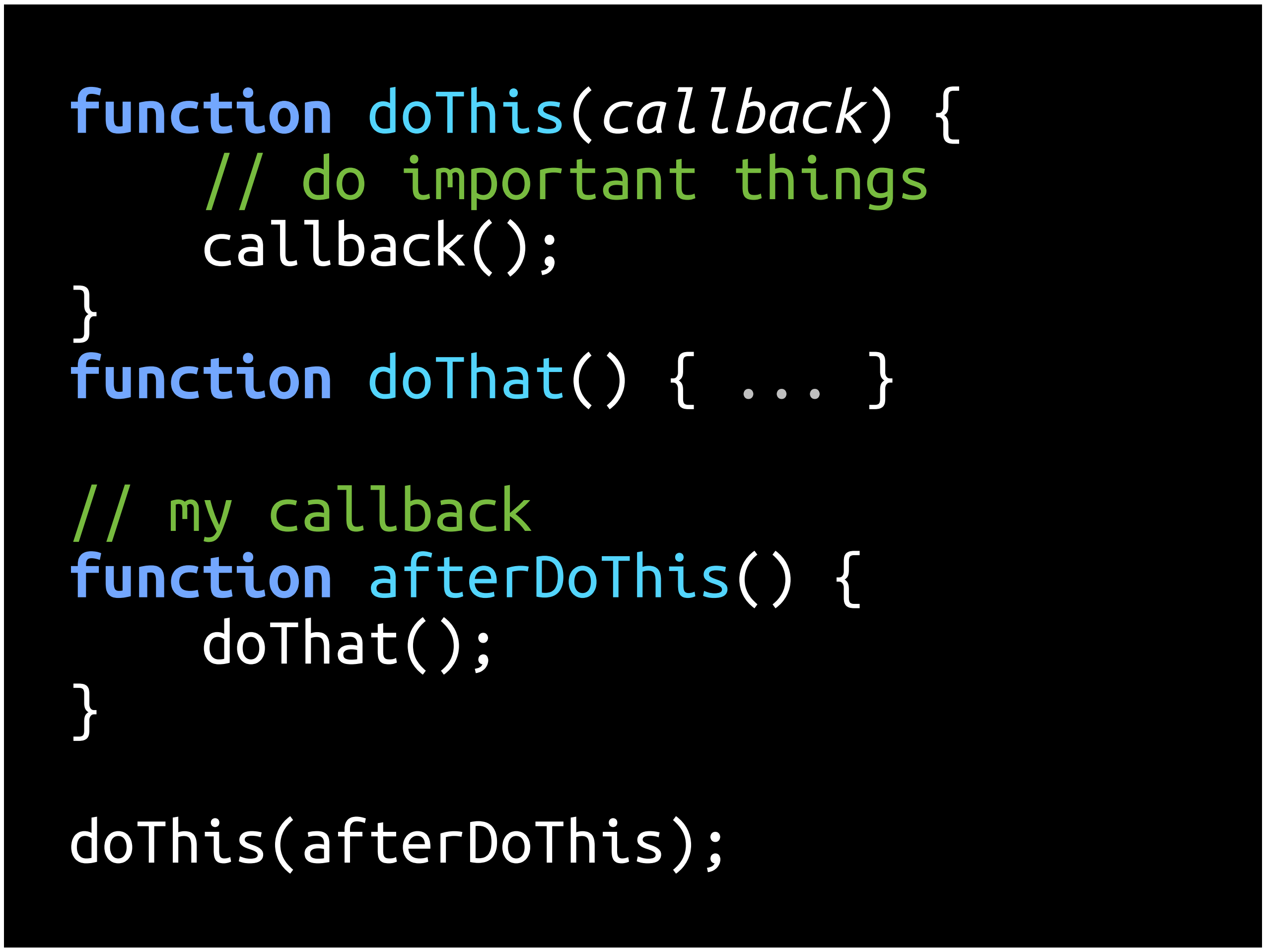
At this point in the presentation, I asked Sam to go get me a glass of water, and then stopped talking. I didn't resume the presentation until Sam came back with the glass of water. The problem with synchronous programming is that if one operation takes a long time, you're stuck waiting doing absolutely nothing until it finishes. It's a waste of everyone's time and resources, and there's no reason I had to just sit there while I waited for the operation (Sam getting water) to finish. Asynchronous programming provides the answer.



```
function doThis() {
    console.log(“Hello, world!”);
}

var myFunc = doThis;
myFunc();

// “Hello, world!”
```

Our ability to program asynchronously in JavaScript comes from this language feature: I can assign the name of a function (NOTE: I did not use parentheses at the end) to a variable, and that variable now becomes a new "alias" for the original function. If I then call my new variable as a function, it does exactly what the original function did.

This is analogous to the concept of function pointers in C: the name of a function always acts as a pointer to it.



```
function doThis(callback) {
    // do important things
    callback();
}
function doThat() { ... }

// my callback
function afterDoThis() {
    doThat();
}

doThis(afterDoThis);
```

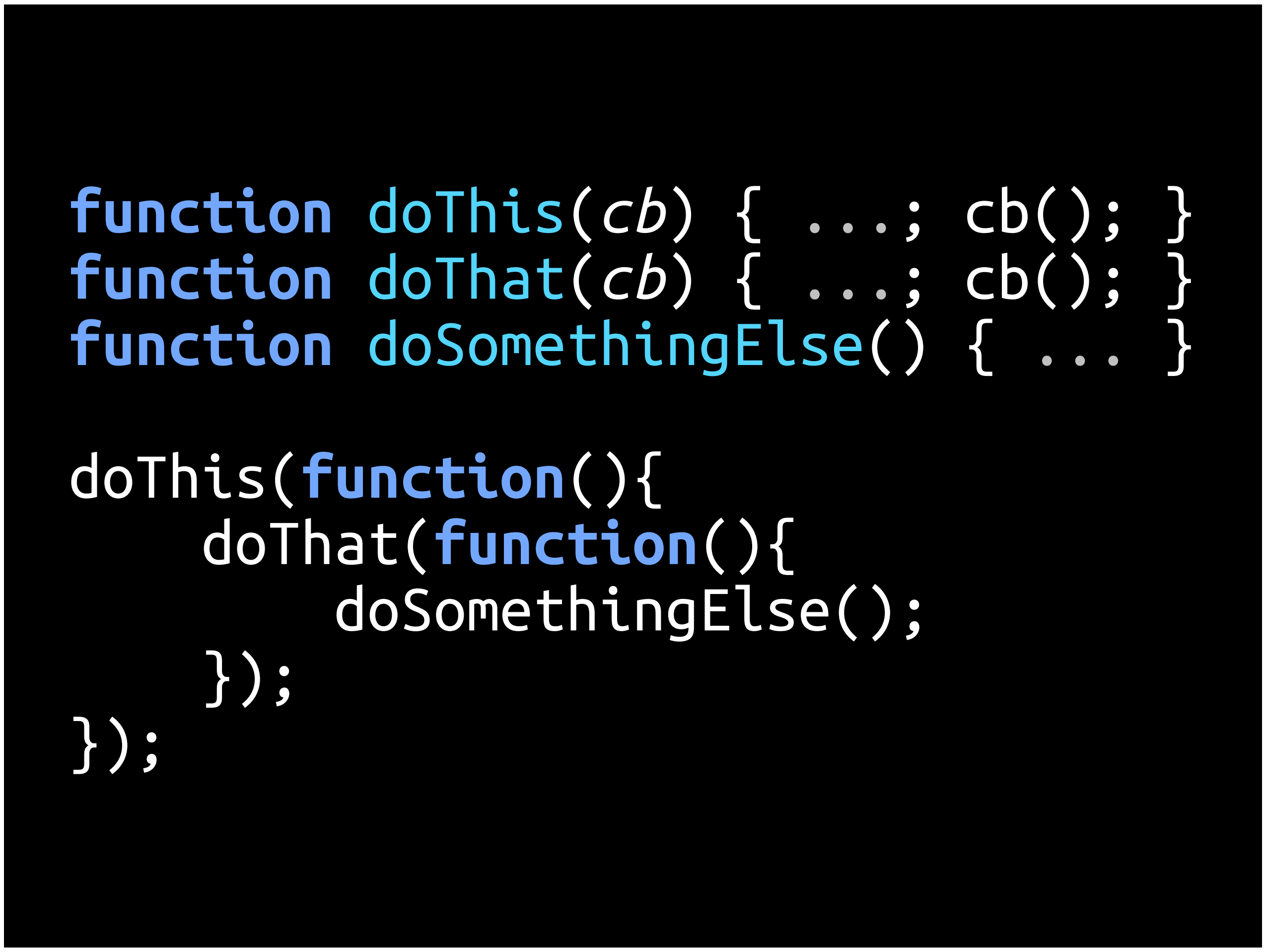
With that in mind, we find a solution to asynchronous programming using “callbacks.” Here, we rewrite our original doThat function to take one parameter — callback — which we then call at the end of the function to signal we’re done. We then create a new function, called “afterDoThis” that will be our doThis callback — all it does is call doThat. Finally, we call doThis passing in afterDoThis as its callback.

The idea here is that this is effectively no different than if we just called doThis(); doThat(); — doThat will happen after doThis finishes. The difference, though, is that we’ve now freed things up to happen asynchronously. doThis now has the capability to tell us when it’s done, so our process can actually do other things in the meantime.

The problem with the code we just wrote, though, is that it’s pretty confusing and requires a lot of work. Writing a whole new set of callback functions and then delicately weaving them together takes a lot of care, and hinders productivity. The solution to this is a new coding style that allows us to “pass the baton” between functions.

Continuation Passing Style

```
foo(^{
    bar(^{
        baz(^{
            // done!
        });
    });
});
```



```
function doThis(cb) { ...; cb(); }
function doThat(cb) { ...; cb(); }
function doSomethingElse() { ... }

doThis(function(){
    doThat(function(){
        doSomethingElse();
    });
});
```

Here, we have the same code as before, but written asynchronously. Note that we've modified doThis and doThat to take callbacks (named cb), which they call at the end of their work. Then, we use JavaScript "anonymous functions" — which can be created by just using the word function without providing a name — and "nest" them into our function calls. The effect is that each new level of "indentation" represents the next place where execution will resume after each operation completes.

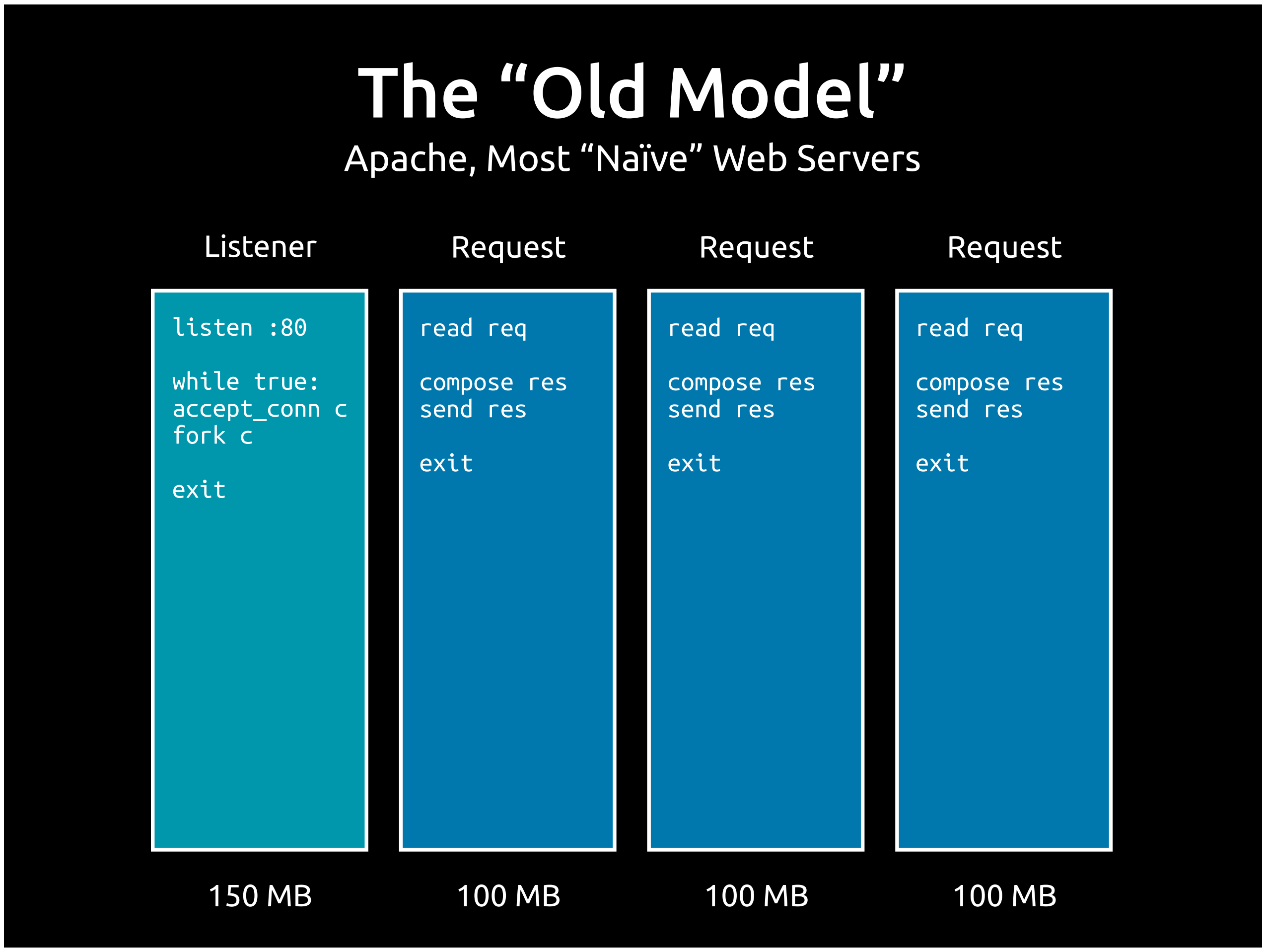
Continuation passing style lets us write code like it's synchronous, but preserves the new asynchronous nature of the callbacks.

Demo

In the code provided with these slides is a file called demo.js. Take any large file, rename it “bigfile.dat”, and put it in the same folder as demo.js, then run the script (node demo.js). If you look at the code, you can generally see what’s happening: we request to read bigfile.dat into memory, but meanwhile, we start counting up from 0, stopping when the file finishes loading.

You’ll notice that, even though it only takes on the order of a few hundred milliseconds to read the file into memory, we’re still able to count into the tens of thousands in the meantime. While 500ms is an instant for a human, it’s an eternity for a computer, which is why it’s important to write code asynchronously (that way, we always maximize all the time that’s given to us).

Also provided in the folder is a C version of the code using Blocks and GCD (two features available only on the Mac). If you have Xcode installed on your machine, you can compile the code by running “make”, and then execute it with “./demo”



The "Old Model"

Apache, Most "Naïve" Web Servers

```
listen :80

while true:
accept_conn c
fork c

exit
```

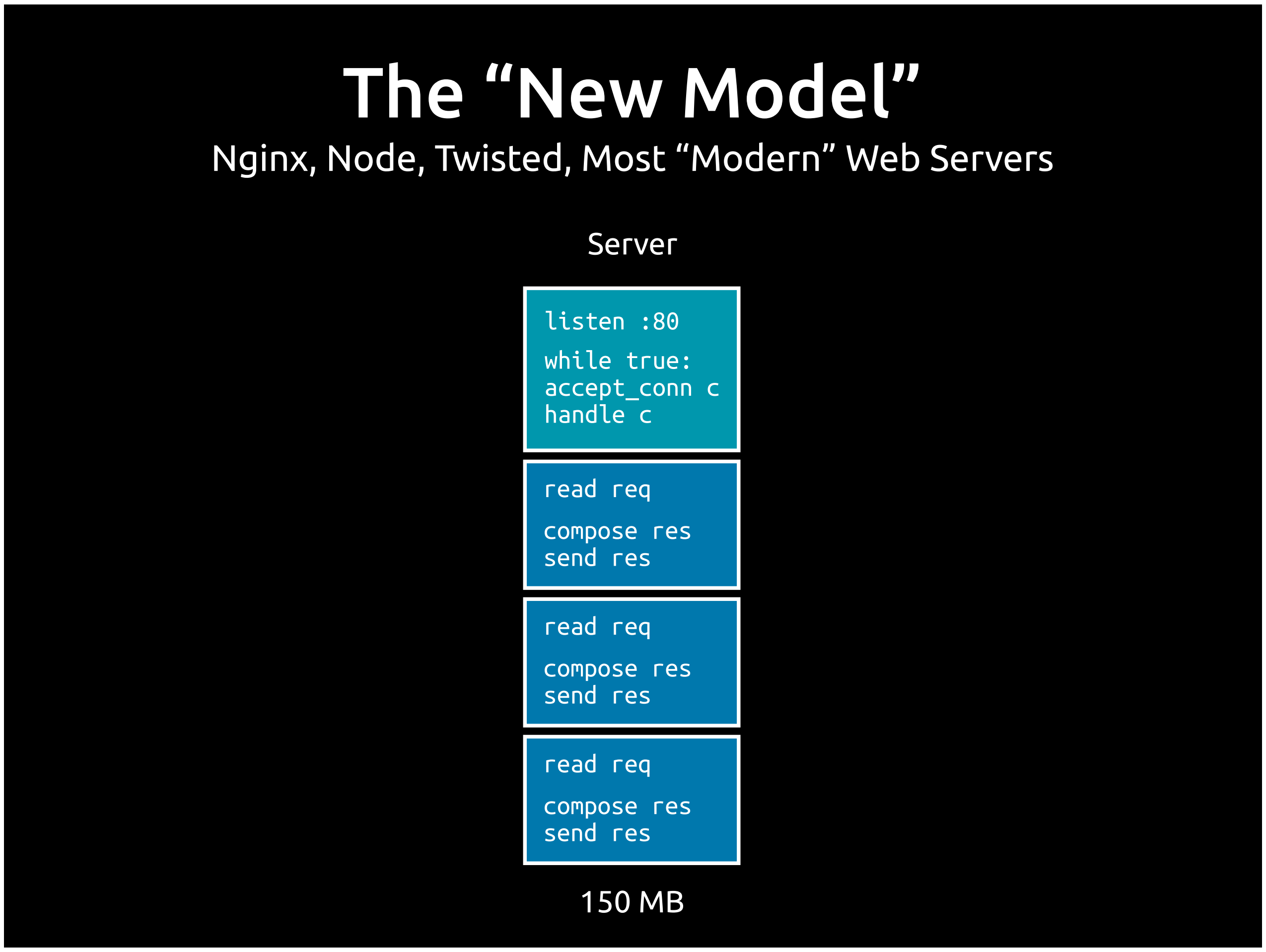
```
read req

compose res
send res

exit
```

```
read req

compose res
send res

exit
```

```
read req

compose res
send res

exit
```

Here we show an example of how web servers used to work. The idea is that old web servers, like Apache, use threads to handle large numbers of clients. They start one thread initially, which listens on a port (like 80), then just waits until clients try to connect. Because they are written synchronously though, once a client does connect, they can't respond to it in the same thread, because that will prevent other clients from making new connections (think about it: if I need to send a large file back, which will take a few seconds, other clients will have to wait that few seconds before I get back around to checking if new clients are waiting to connect). In order to solve that problem, then, Apache forks a new thread for every incoming request, which is responsible for handling *just* that request.

Here's a scenario, though: imagine you pay $20 p/month for web hosting, and you get 512MB of RAM for that price. In the Apache model, each thread has its own copy of PHP in memory, which means each thread weighs in at ~150MB of RAM. If three users try to access your site simultaneously, you have effectively exhausted the capacity of your server, and other users will have to wait.



The “New Model”

Nginx, Node, Twisted, Most “Modern” Web Servers

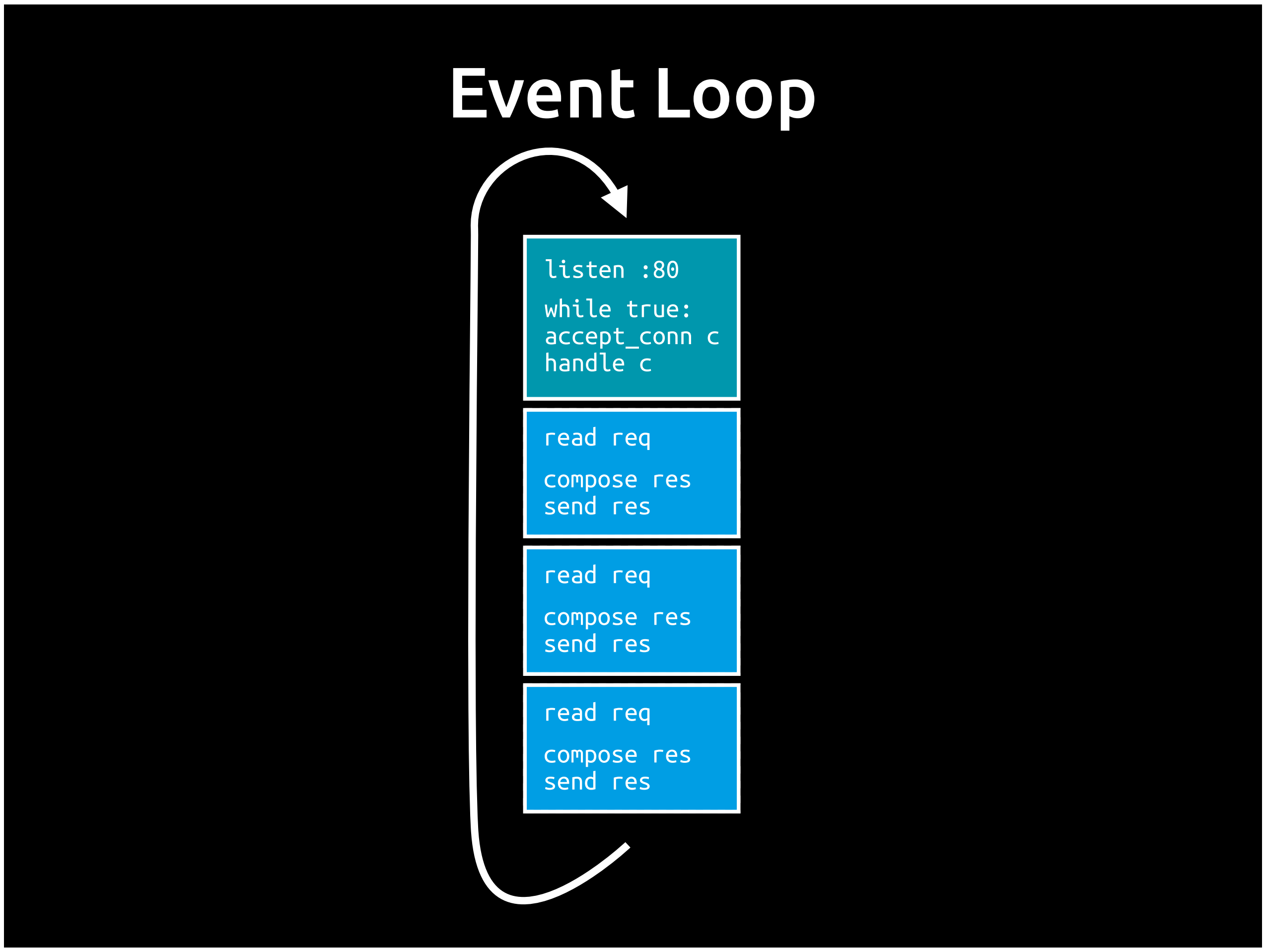
```
listen :80
while true:
accept_conn c
handle c
```

```
read req
compose res
send res
```

```
read req
compose res
send res
```

```
read req
compose res
send res
```

If we instead write our code asynchronously, though, new options are opened to us. For instance, most modern web servers now use just a single thread, which means their memory usage is relatively low.



Event Loop

```
listen :80
while true:
accept_conn c
handle c
```

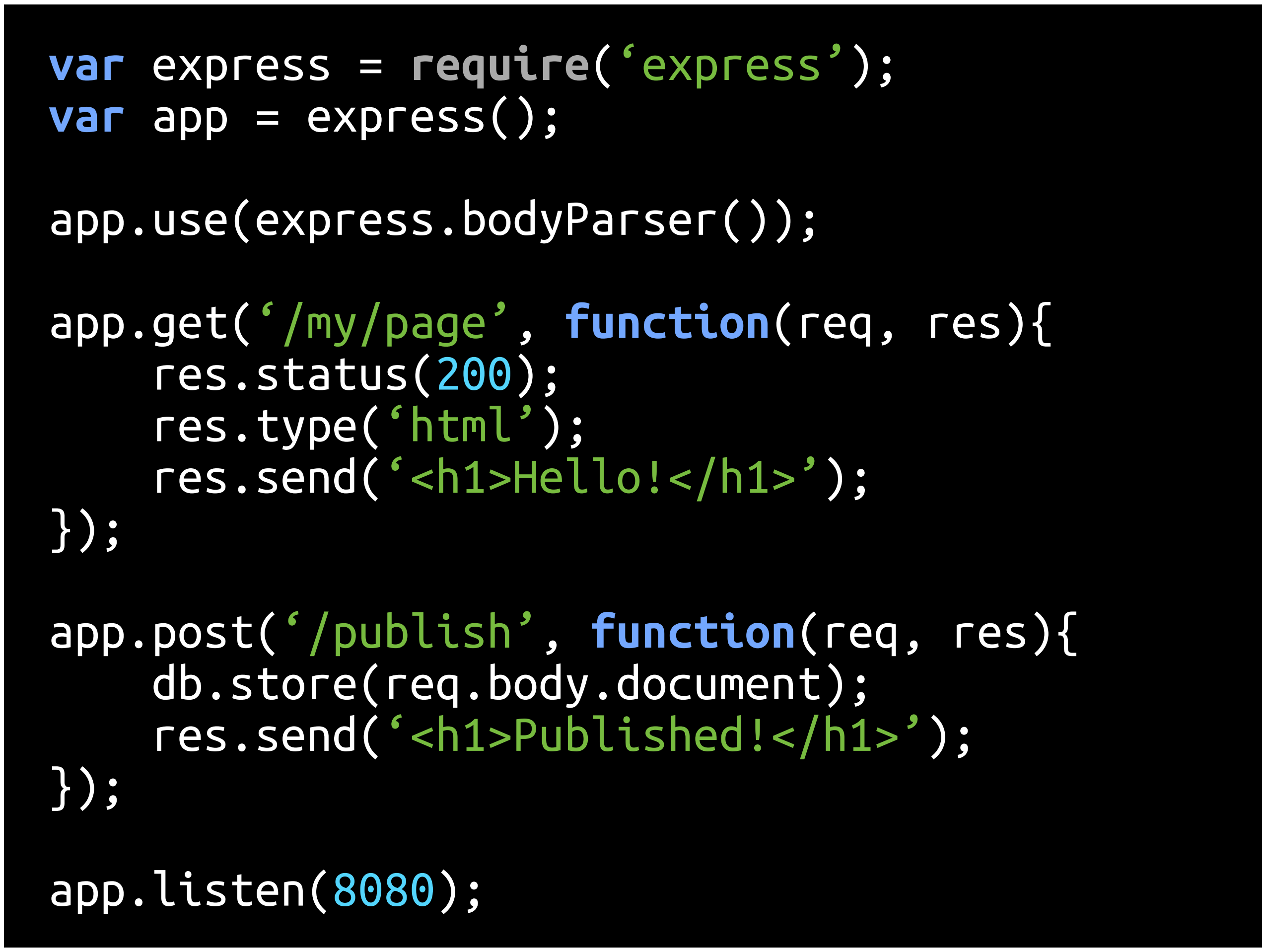
```
read req
compose res
send res
```

```
read req
compose res
send res
```

```
read req
compose res
send res
```

This works by structuring code in an “event loop.” The idea is that, initially, there’s only one event in the loop, which is to check for new connections. Once connections come in, new events get added to the loop to handle each request.

Now that many of our important operations (like reading files and sending responses) are asynchronous though, thanks to Node’s callback-friendly APIs, I no longer have to wait for one request to finish before moving onto the next one. Consider, for example, the large file scenario: if I get a request for a large file, I can start reading it in, but because it will let me know when it’s done, I no longer have to wait — I can move on and start handling the next request. Eventually, the response to the first request will get sent once the file finishes loading. This maximizes the utility of our one thread, rather than having lots of heavyweight threads that don't do all that much individually.

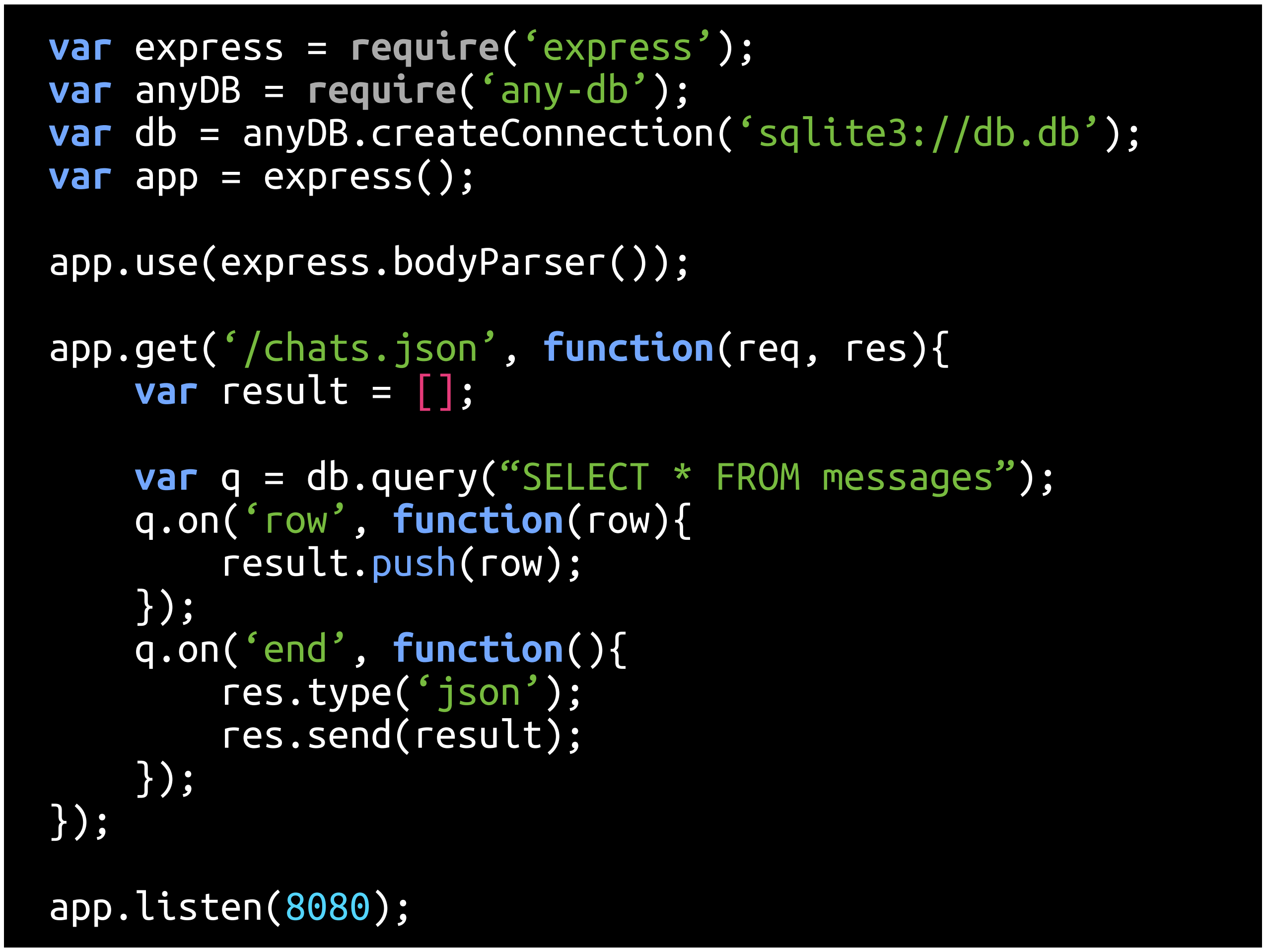


```
var express = require(‘express’);
var app = express();

app.use(express.bodyParser());

app.get(‘/my/page’, function(req, res){
    res.status(200);
    res.type(‘html’);
    res.send(‘<h1>Hello!</h1>’);
});

app.post(‘/publish’, function(req, res){
    db.store(req.body.document);
    res.send(‘<h1>Published!</h1>’);
});

app.listen(8080);
```

Let’s apply this to a typical example: an Express app. Here, we require Express, setup the app, tell it to use the bodyParser middleware, and then start listening on port 8080. All of these options happen synchronously — we don’t provide them callbacks, nor should we, because they’re just part of our server start-up time.

However, the interesting things here are our app.get and app.post calls. First of all, it’s important to note that app.get(…) doesn’t mean “get something,” it means “I’m going to tell you what to do when a *web browser* tries to get something.” This is how you associate callbacks with particular URLs, and this is the reason we use Express — it affords us the convenience of linking arbitrary callbacks to URLs.

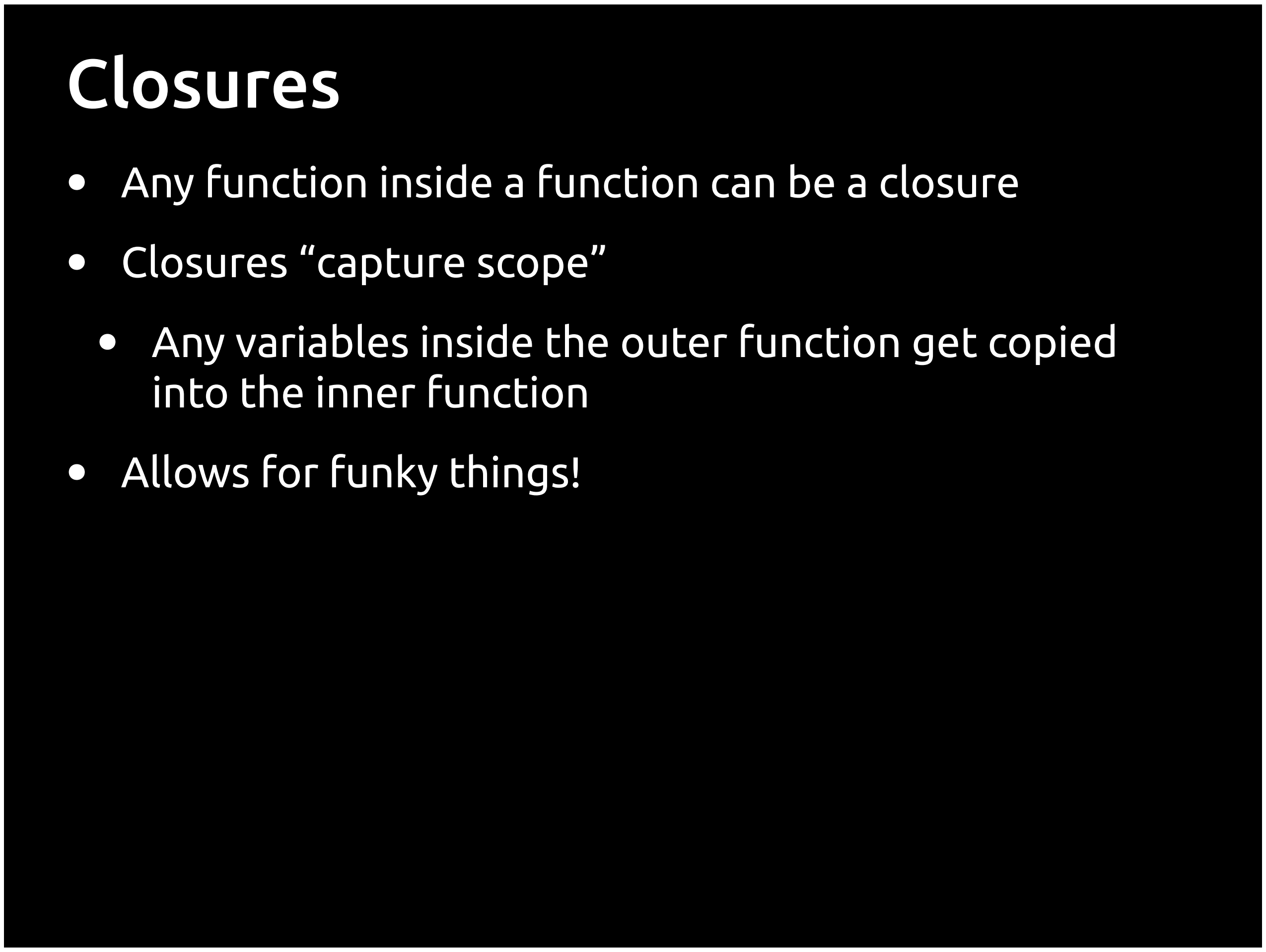
Inside each URL callback, we do any work we need to do, and then send a response. Note that if we don’t call res.send(…) (or something similar like res.end(…), res.render(…), or res.redirect(…)), the browser will just sit there forever. This is a consequence of asynchronous programming: rather than just sending back an empty response at the end of your callback, Express assumes that you might be waiting on some *other* callback before you want to send a response.



```
var express = require('express');
var anyDB = require('any-db');
var db = anyDB.createConnection('sqlite3://db.db');
var app = express();

app.use(express.bodyParser());

app.get('/chats.json', function(req, res){
    var result = [];

    var q = db.query("SELECT * FROM messages");
    q.on('row', function(row){
        result.push(row);
    });
    q.on('end', function(){
        res.type('json');
        res.send(result);
    });
});

app.listen(8080);
```

Here's an example of that with a database. When a request comes in for chats.json (which, incidentally, isn't a real file: it's a "file" we're going to generate on-the-fly), we start off a request to our database for a list of all the messages. We associate callbacks with the "row" event (which gets fired every time a row comes back from the database), and the "end" event (which gets fired when the database has given us all of our results). Once we get told the database is done, *then* we are free to send the response to the web browser.

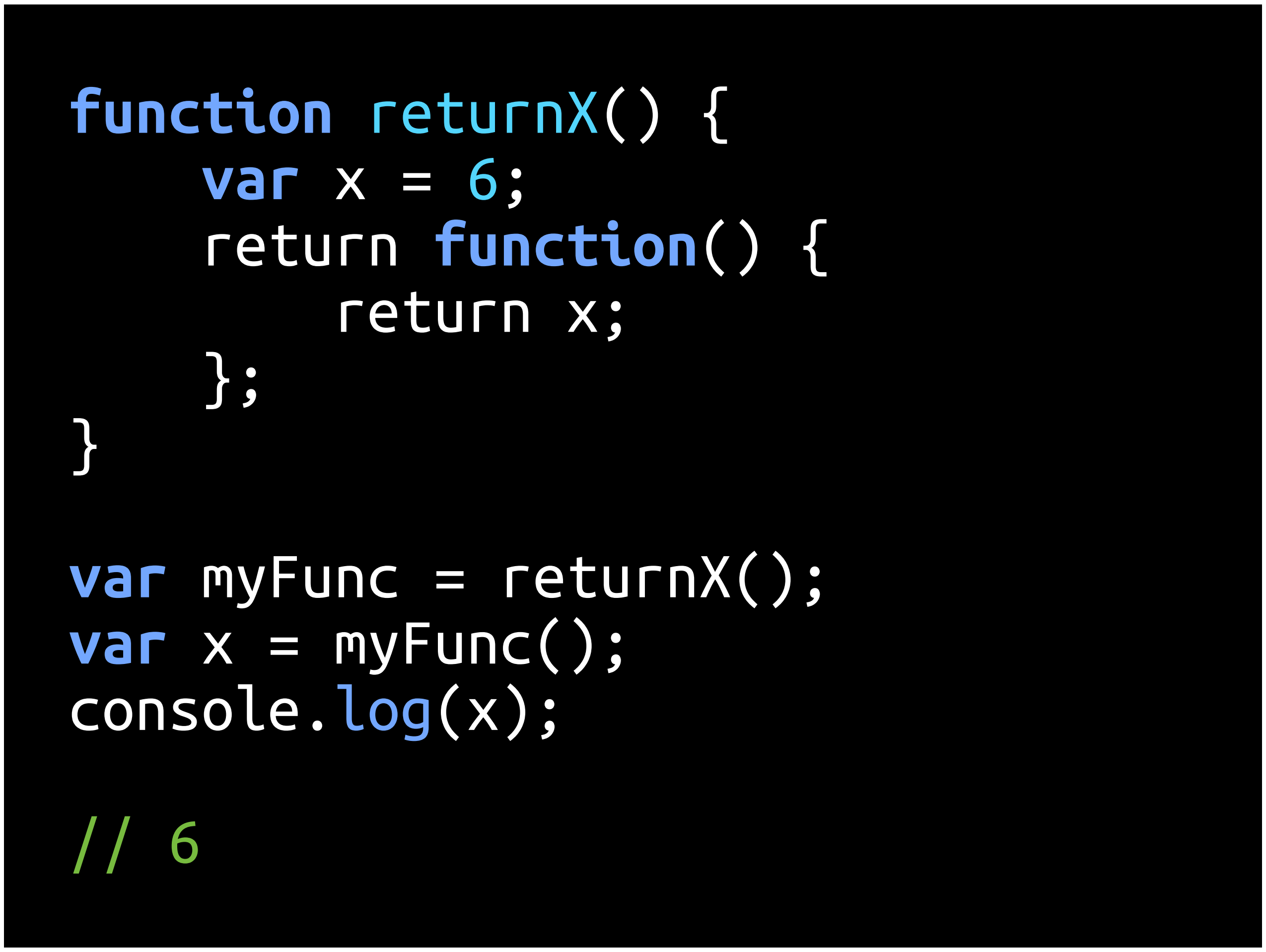
One important point to note here, though, is that both of our callbacks use variables that were defined elsewhere: they use the result array, which we defined in the handler callback for chats.json. The reason they're able to do this is because of a feature of functional languages (like JavaScript) called closures.



Closures

- Any function inside a function can be a closure
- Closures “capture scope”
  - Any variables inside the outer function get copied into the inner function
- Allows for funky things!

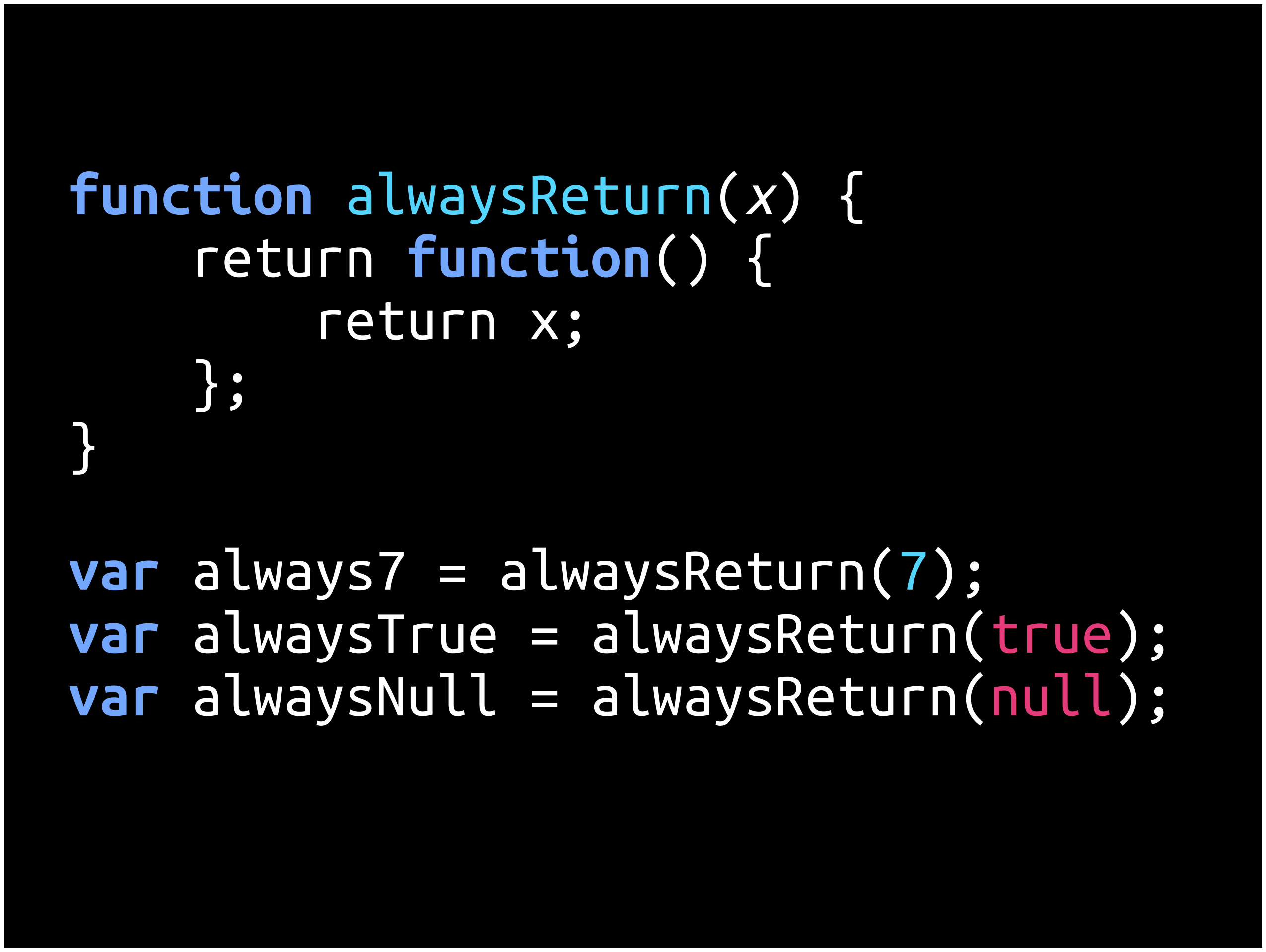
This slide is self-explanatory.



```
function returnX() {
    var x = 6;
    return function() {
        return x;
    };
}

var myFunc = returnX();
var x = myFunc();
console.log(x);

// 6
```

Here’s a simple example of a closure. We have a variable inside the function returnX which is set to 6. Every time we call returnX, we return a new anonymous function which returns that value of x.

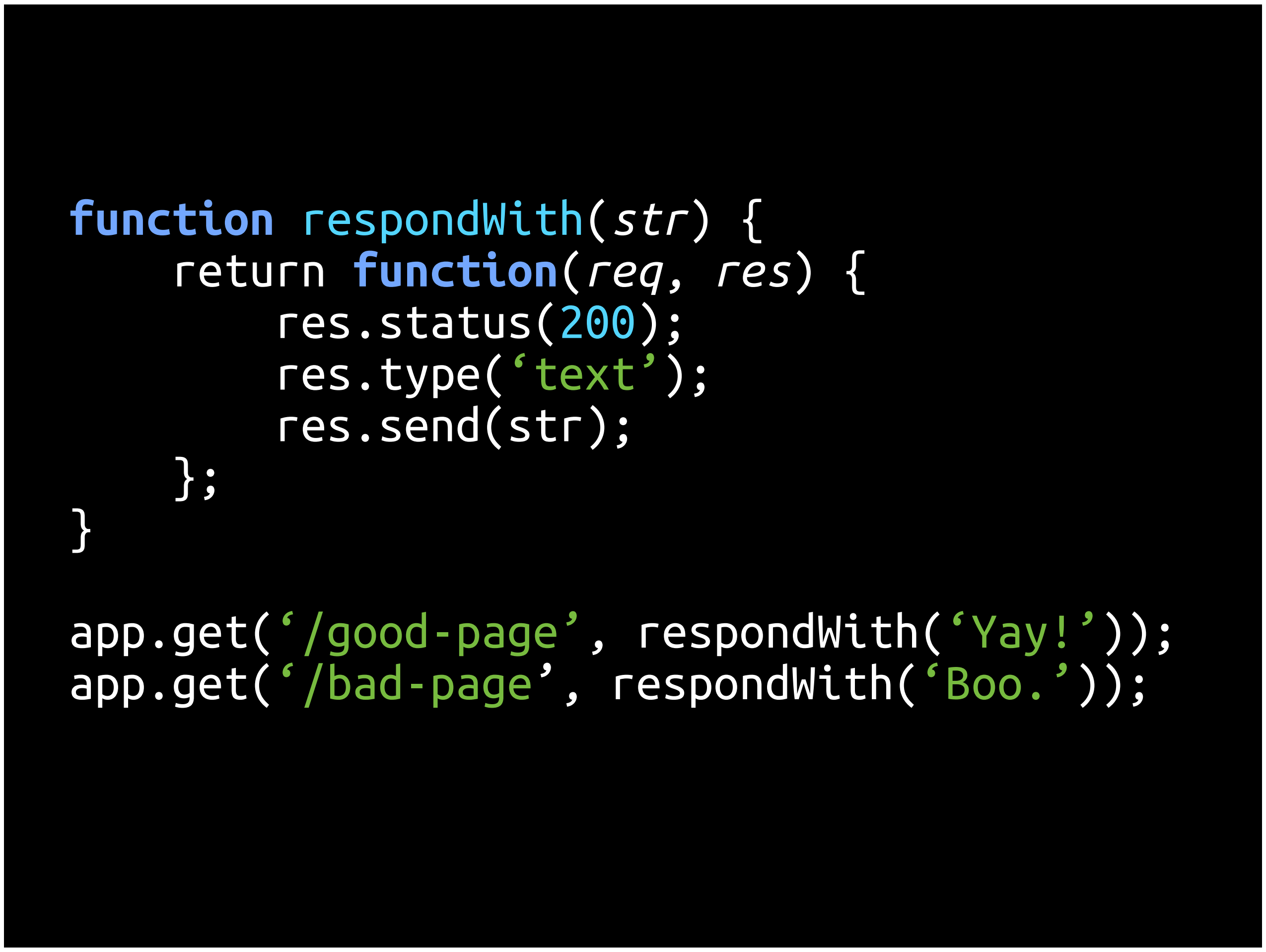
When we call returnX, it’s important to notice that we get *another function back*. So, in the example above, myFunc now becomes a new function that has “closed over” the value of x. When I call myFunc, it returns 6, even though that variable technically belonged to its “parent function.”



```
function alwaysReturn(x) {
    return function() {
        return x;
    };
}

var always7 = alwaysReturn(7);
var alwaysTrue = alwaysReturn(true);
var alwaysNull = alwaysReturn(null);
```

Let’s take a look at some more interesting examples: closures also apply to arguments passed to a function. So, here, we’ve created a function called alwaysReturn, which returns a new function that will always return the value originally passed into alwaysReturn.

So, using the alwaysReturn function, we create three brand new functions: always7, alwaysTrue, and alwaysNull. If we subsequently call, for example, alwaysTrue(), it will always return “true”. If we call always7(), it will always return the number 7, and so on.

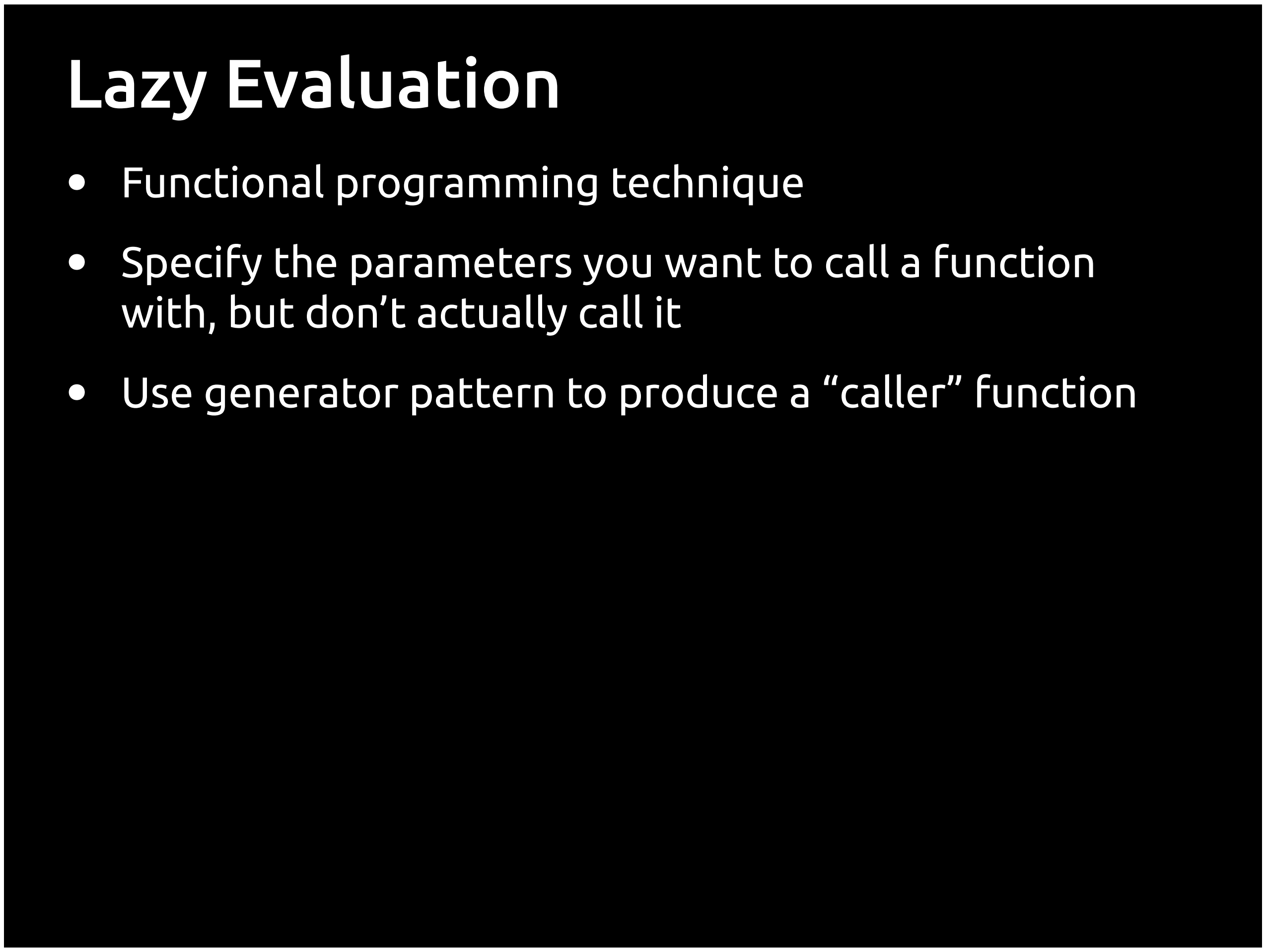


```
function respondWith(str) {
    return function(req, res) {
        res.status(200);
        res.type('text');
        res.send(str);
    };
}

app.get('/good-page', respondWith('Yay!'));
app.get('/bad-page', respondWith('Boo.'));
```

This gets more interesting in our new example: a respondWith function. Here, we pass a string into the respondWith function, which in turn returns a new function that follows the standard pattern of an Express callback. Inside that callback, we send 200 OK, set the response type to plaintext, and send the original string we passed in as a response.

The utility here is that we can very easily generate new callbacks for Express. Here, we setup /good-page to respond with "Yay!", and /bad-page to respond with "Boo." While ordinarily we would have to duplicate the anonymous function code to create handlers, we use respondWith to make our code short.

It's important to notice here that respondWith *does not get called when we go to, e.g., /good-page*: respondWith is called *immediately* when we setup the callback. The return value of respondWith, a new function, is what gets called when we visit /good-page.

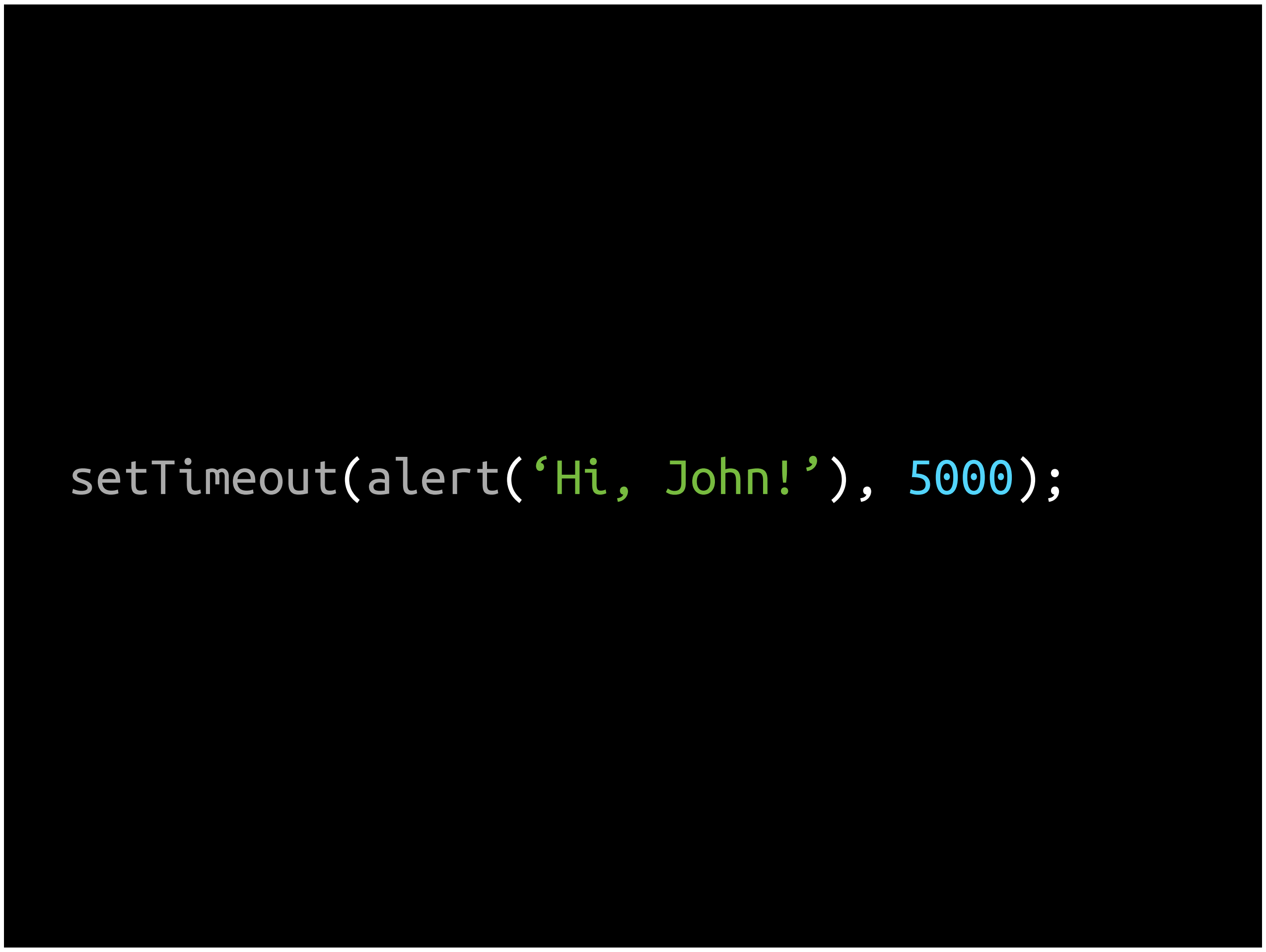


Lazy Evaluation

- Functional programming technique
- Specify the parameters you want to call a function with, but don’t actually call it
- Use generator pattern to produce a “caller” function

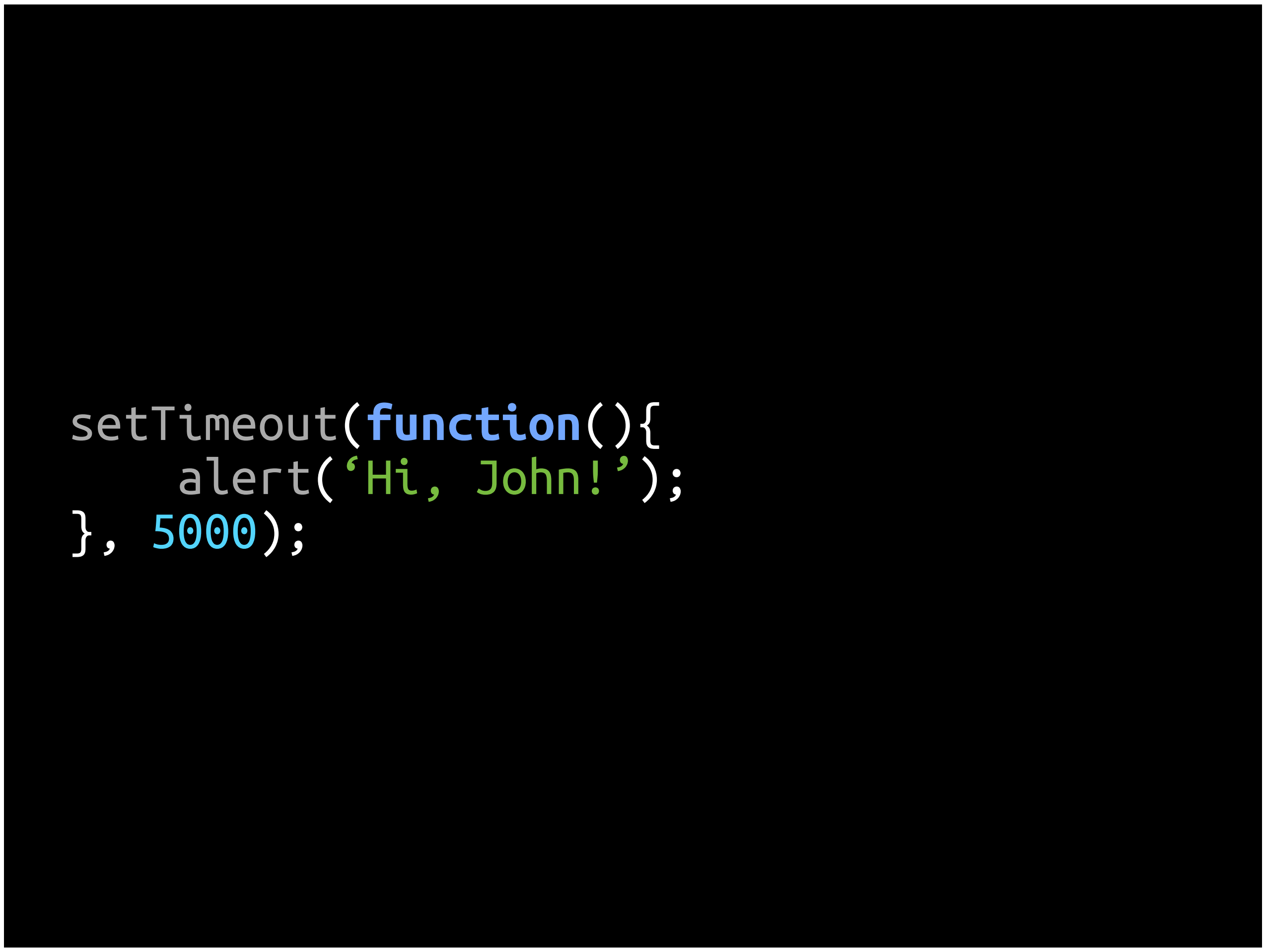
The code we’ve been working with is a prelude to lazy evaluation, the idea that we can call a function but not have it actually execute until some time in the future when we want it. The standard way of doing this is using something called the “generator pattern.”

Note: none of the rest of this is necessarily applicable to the Chatroom assignment. While the code is cool and nice to know, do not feel like you’re doing something wrong if you don’t find a way to make use of it in Chatroom.



```
setTimeout(alert(‘Hi, John!’), 5000);
```

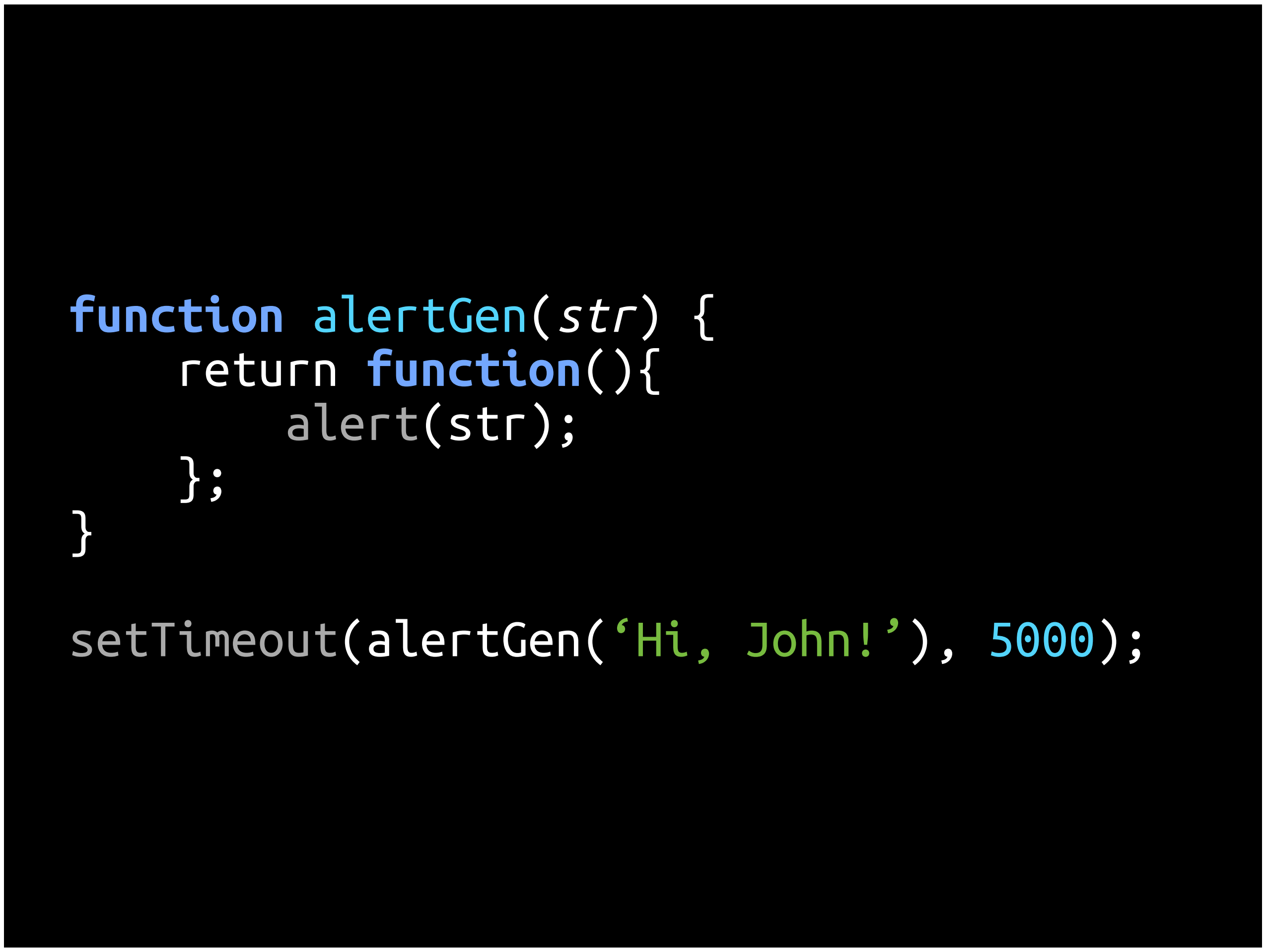
Here we have some sample code for the client (running in the web browser, not in Node). What we want to do is show an alert with “Hi, John!” after 5 seconds (5000ms). Unfortunately this code doesn’t work. Instead, “Hi, John!” shows immediately, and in 5 seconds, we get an error. why?



```
setTimeout(function(){
    alert(‘Hi, John!’);
}, 5000);
```

The reason the previous code didn’t work is because it said to run alert(…) immediately. The return value of alert(…) was then used as the callback for setTimeout, but since alert returns null, setTimeout couldn’t make sense of it.

One solution to this is to use an anonymous function like we’ve been doing with Express — we pass in an actual *function* that we want to execute after 5 seconds, and that function calls alert. This works, but the code is ugly, especially if we want to do this kind of thing a lot. The solution is the generator pattern.

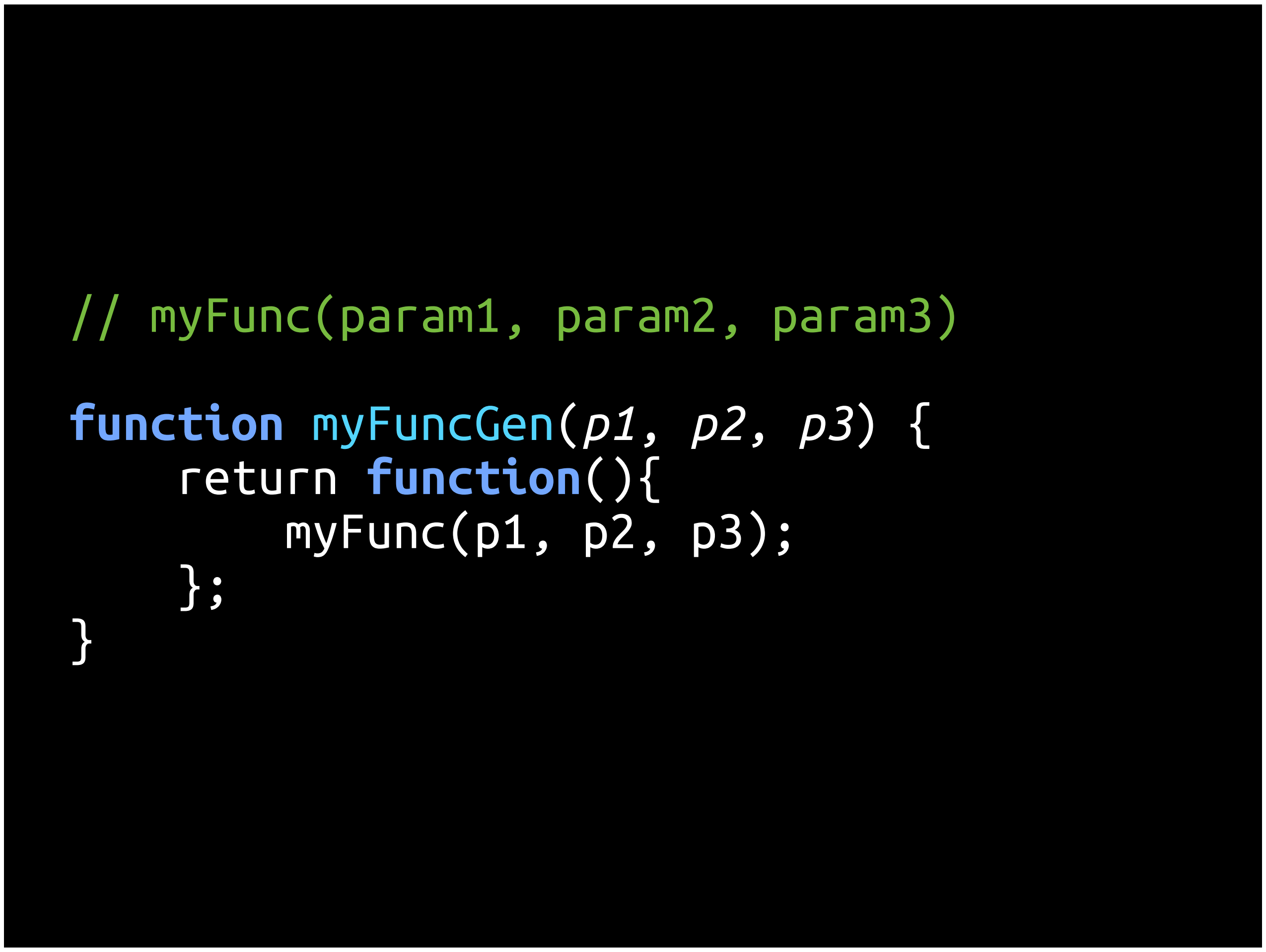


```
function alertGen(str) {
    return function(){
        alert(str);
    };
}

setTimeout(alertGen(‘Hi, John!’), 5000);
```

Here, we define a new function called alertGen (which takes a string), that returns a new function that just calls alert with the given string (that inner function is the same as the callback we used on the previous slide). Now, when we go to use setTimeout, we call alertGen instead of alert. Everything works as expected.

What’s important to notice here is that we now have the exact same semantics as our first example (that didn’t work): the only difference in our call to setTimeout here is the three letters “Gen” in alertGen. Otherwise, it looks as though we’re calling the alert function, just delayed by 5 seconds.

The reason this works is that alertGen actually *does* execute immediately, like alert did on our first slide. The difference is that alertGen doesn’t do anything itself, it just returns a new function that actually does the real work of showing the alert.

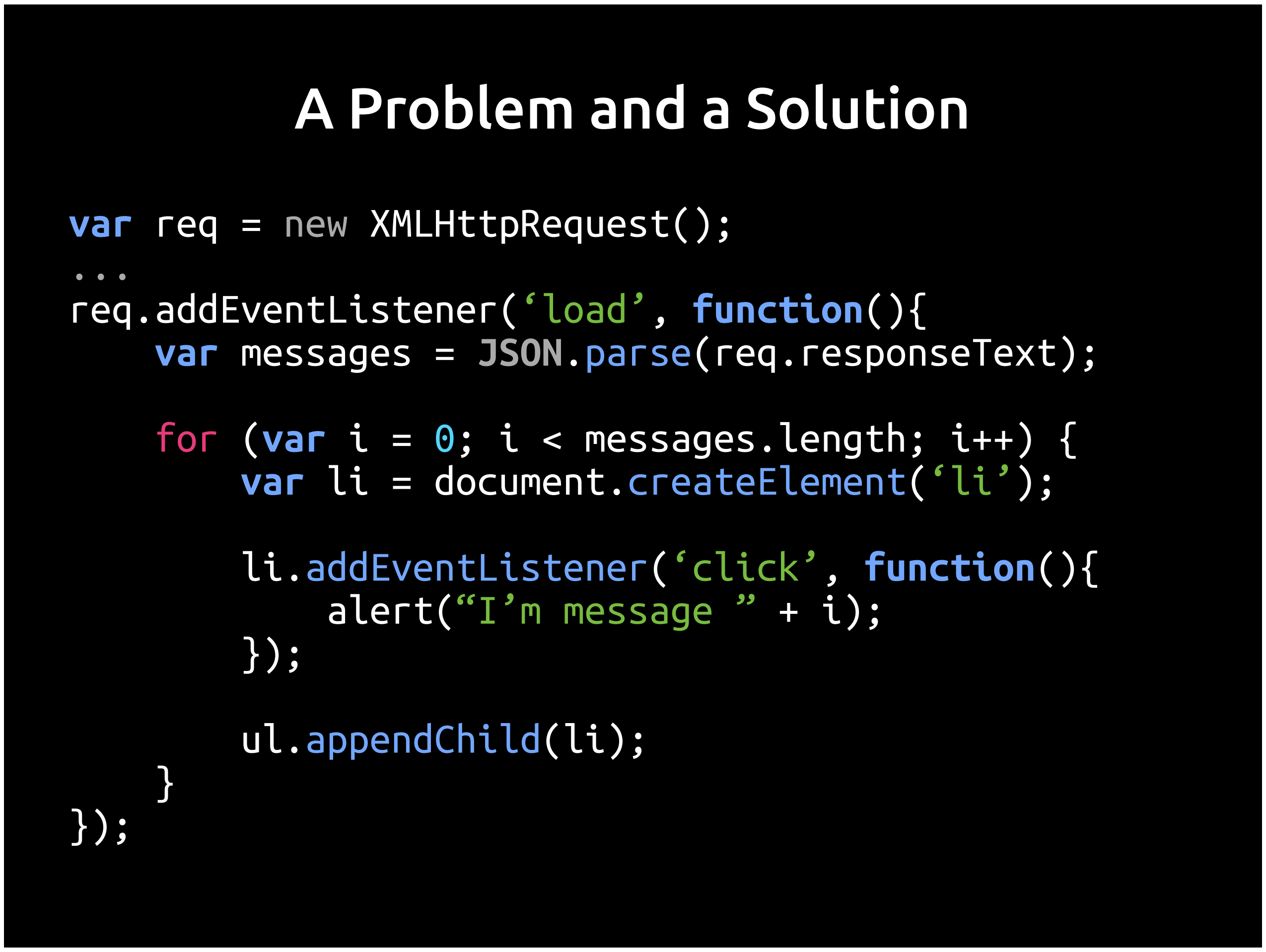


```
// myFunc(param1, param2, param3)

function myFuncGen(p1, p2, p3) {
    return function(){
        myFunc(p1, p2, p3);
    };
}
```

The code on the previous slide can be made very generic, which is called the generator pattern. Imagine you want to call the function myFunc with parameters p1, p2, and p3, but you instead want to do it in such a way that you can use it as a callback — i.e., you don't want to call it *immediately*, you want to give it to someone else (like setTimeout or app.get) to call at a later time.

To do that, you can pretty much copy the above code verbatim. Create a new function called myFuncGen (or whatever really) that takes the parameters of your original function. Then, return an anonymous function (that takes no parameters) that just calls the original function with the parameters passed to the generator.

If I now did "var foo = myFuncGen(a, b, c)", and then later did "foo()", it would have the same effect as calling "myFunc(a, b, c)". You create a new function that's "bound" to a set of parameters you want to use later, and that new function can either be called directly, or used as a callback.



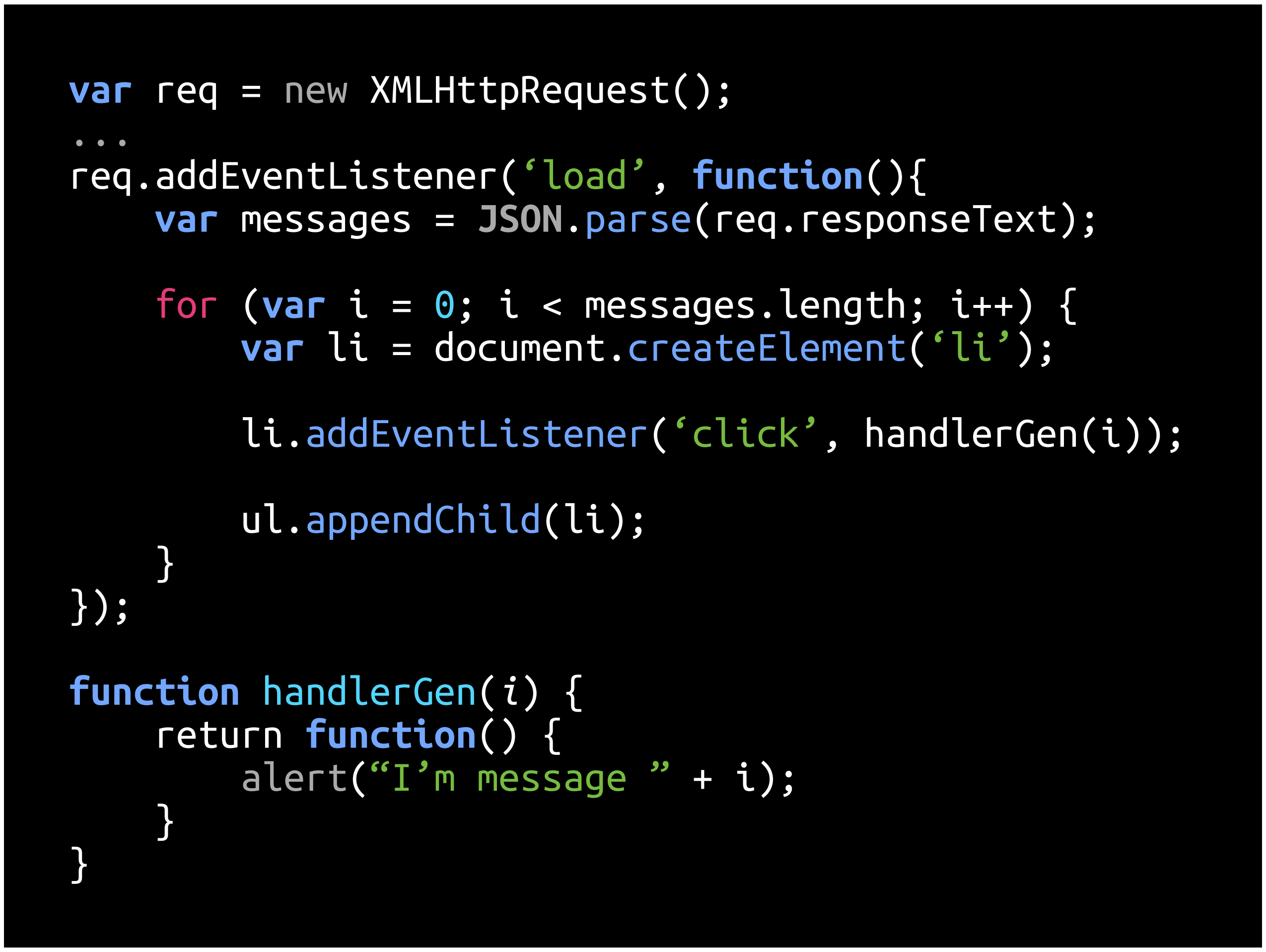
A Problem and a Solution

```
var req = new XMLHttpRequest();
...
req.addEventListener('load', function(){
    var messages = JSON.parse(req.responseText);

    for (var i = 0; i < messages.length; i++) {
        var li = document.createElement('li');

        li.addEventListener('click', function(){
            alert("I'm message " + i);
        });

        ul.appendChild(li);
    }
});
```

While lazy evaluation and the generator pattern is a little esoteric in and of itself, it actually has a common application. Consider the above code, running in a web browser, where you make an Ajax request, then create a bunch of li's with messages retrieved from the server. For each message, you add a callback so that when someone clicks on the message, it pops up an alert with the number of the message that you clicked.

Unfortunately, this code doesn't work. If you run this, and you have (e.g.) 10 messages, clicking *any* message will pop up an alert that says "I'm message 10". Why is that?

The reason is that our anonymous callback functions that we create in the loop all close over the same "scope" (set of variables a function can "see" at a given time), which means they all share the same i variable. Thus, as i keeps increasing with each iteration of the loop, it increases in *all* functions that closed over it.



```
var req = new XMLHttpRequest();
...
req.addEventListener('load', function(){
    var messages = JSON.parse(req.responseText);

    for (var i = 0; i < messages.length; i++) {
        var li = document.createElement('li');

        li.addEventListener('click', handlerGen(i));

        ul.appendChild(li);
    }
});

function handlerGen(i) {
    return function() {
        alert("I'm message " + i);
    }
}
```

The solution is to use the generator pattern. Here, instead of writing our callback inline, we moved it (unchanged) into a new function called handlerGen, that takes an i parameter. Then, in our loop, we now call handlerGen with the current value of i. handlerGen then returns the callback that we want to use.

The reason this works is because handlerGen, being a separate function, has its *own* value for i — it doesn't use the same one that the loop uses. Thus, each call to handlerGen returns a *different* callback bound to the value of i during that particular iteration of the loop.

You don't have to fully understand how this works (though it's pretty cool). The takeaway here, though, is that if you have this kind of loop, and you find somehow that every item in your list is doing something that only the last item in your list should do, you can usually fix it using the generator pattern.

Q & A

If you have any questions about the content contained in these slides, you can always ask the TAs during office hours, or else email us at cs132tas@cs.brown.edu. Thanks!