Callback Hell
(aka Bracket Hell)

Asynchronous Functions in JavaScript with Node.js

Asynchronous Javascript Functions

It is very typical the Javascript environments (Browsers, Node.js) to provide asynchronous functions.

The node.js execution environment provides <u>many</u> utility functions that are asynchronous (and take a call-back function parameter).

Here we look at the motivation behind them and the style of programming that results which is often referred to as "Callback Hell" (as opposed to "Spaghetti Code").

Callback Hell is characterized by a lot of nesting and brackets. (Use editors that will show you matching brackets.)

Asynchronous Javascript Functions

Asynchronous Callback Functions are characterized by defining functions within other function definitions which results in a lot of nested brackets.

```
function (… function(){…} ){…}
```

[It is essential to use an editor the shows matching brackets]

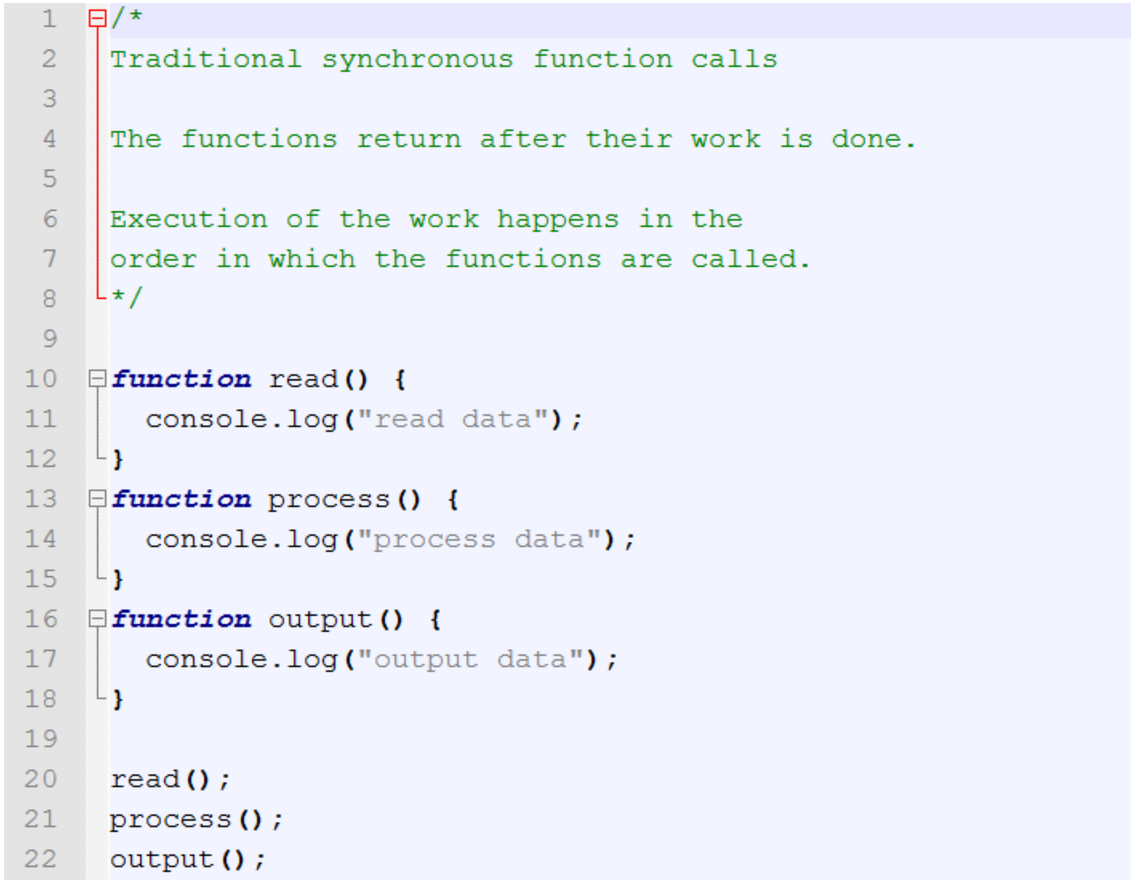
Asynchronous Javascript Functions

- Javascript functions use many bracket syntaxes
- The following are all very common javascript syntaxes involving functions and brackets

```
f(){...}
f()
f(){...g(){...}...g()}
f(){...return g(){...}}
f.g()
f(g(){...}){...g()}
(f(){...})()
(f(){...}())
f()()
(f(){}())()
f(){... f=function(){}}
```

Synchronous Functions

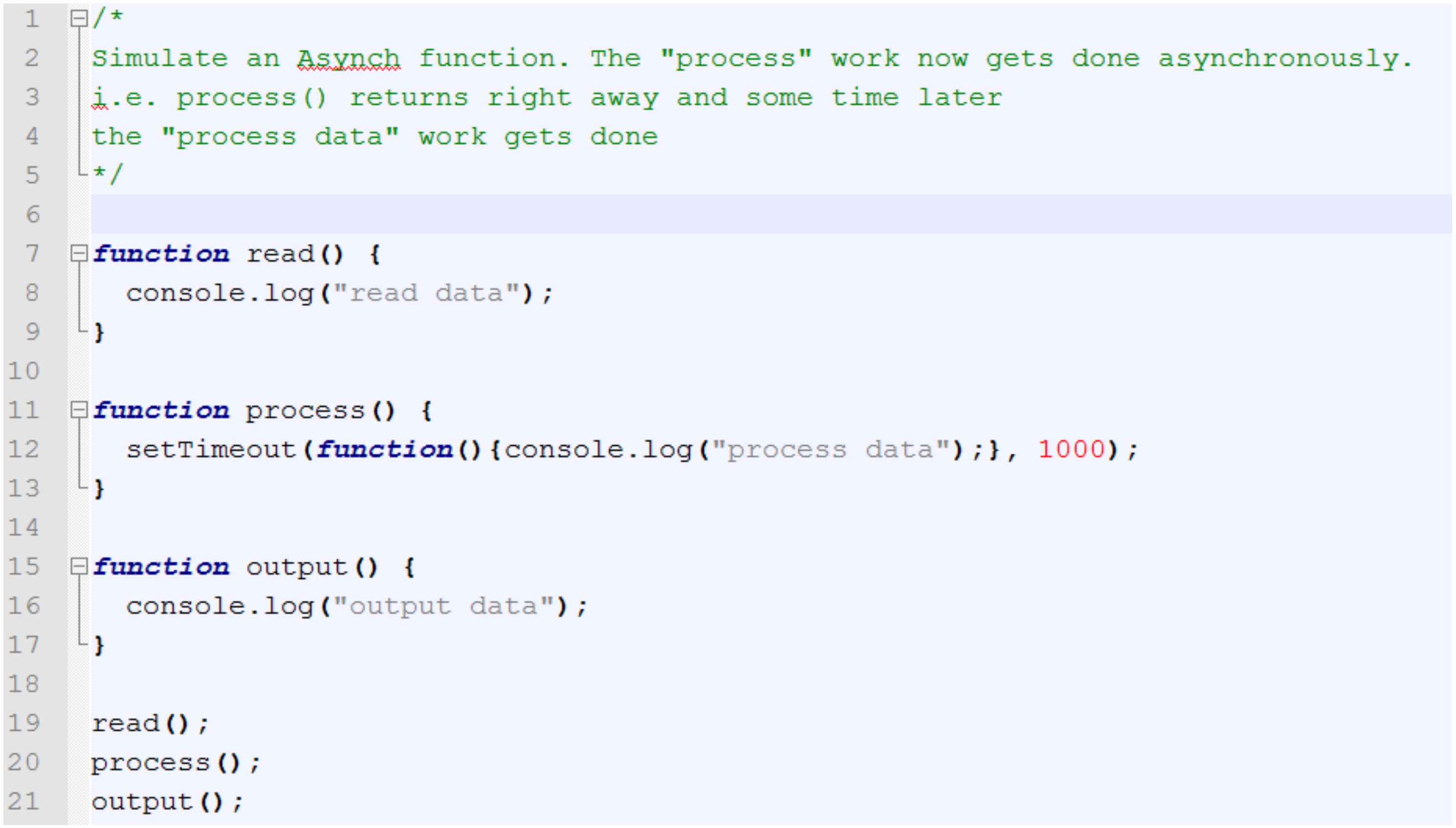
- **Traditional blocking function calls**



```
/*
Traditional synchronous function calls

The functions return after their work is done.

Execution of the work happens in the
order in which the functions are called.
*/

function read() {
   console.log("read data");
}
function process() {
   console.log("process data");
}
function output() {
   console.log("output data");
}

read();
process();
output();
```

Asynchronous Functions

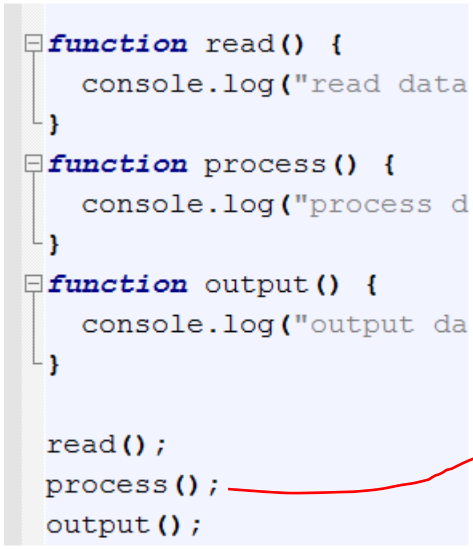
- **Asynchronous function**



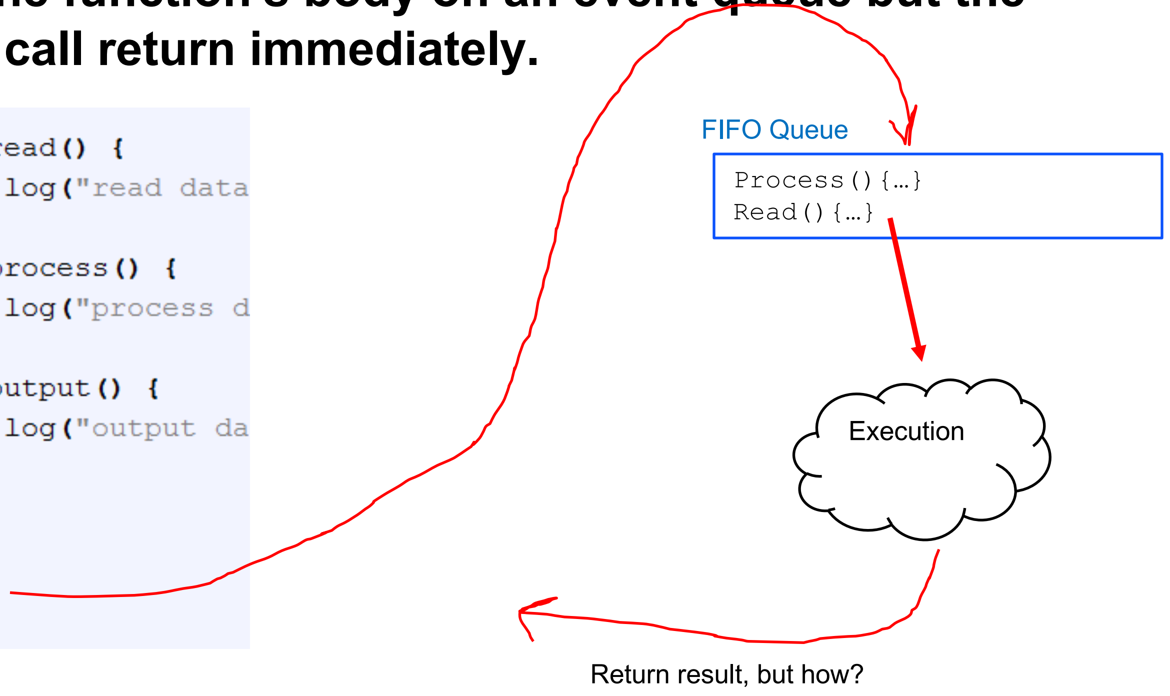
```
/*
Simulate an Asynch function. The "process" work now gets done asynchronously.
i.e. process() returns right away and some time later
the "process data" work gets done
*/

function read() {
  console.log("read data");
}

function process() {
  setTimeout(function(){console.log("process data");}, 1000);
}

function output() {
  console.log("output data");
}

read();
process();
output();
```

Asynchronous Javascript Functions

- The previous examples use asynchronous utility functions (setTimeout) which "condems" the outer function to the effective be asynchronous as well.
- Node.js, and browsers achieve asynchronous behavior by putting the function's body on an event queue but the function call return immediately.



```
function read() {
  console.log("read data
}
function process() {
  console.log("process d
}
function output() {
  console.log("output da
}

read();
process();
output();
```



Motivation for Asynchronous Functions

Accessing Data in memory, disk, network.

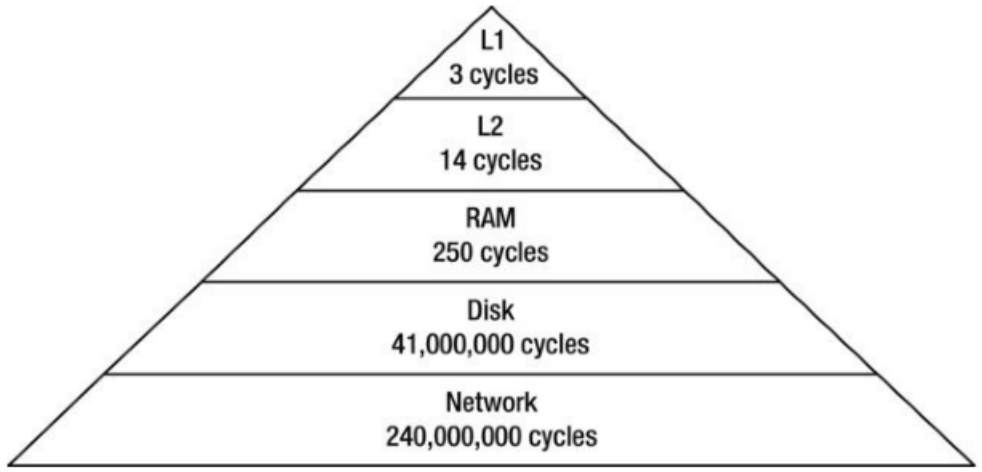


Figure 2-1. Comparing common I/O sources

Beginning Node.js

Copyright © 2014 by Basarat Ali Syed

Motivation for Asynchronous Functions

What do servers and browsers spend most of their time doing?

Node.js provides asynchronous functions for accessing the file system, network requests, streams, timers, and other things.

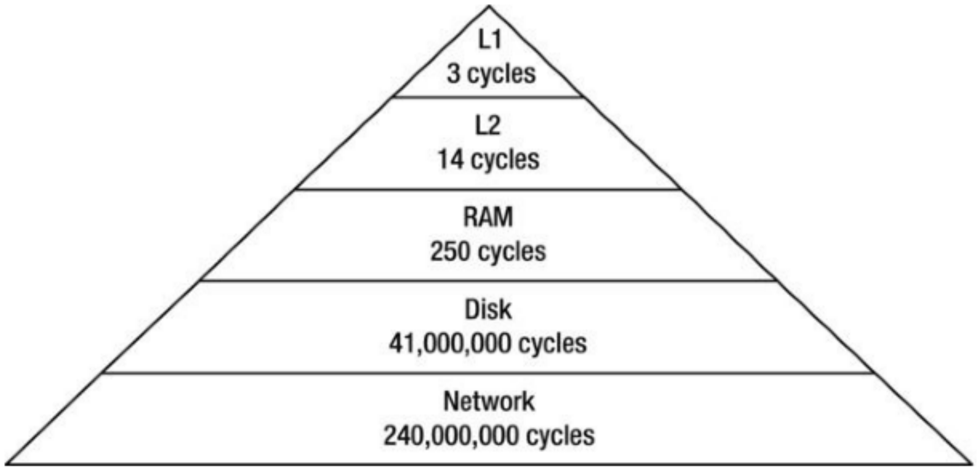


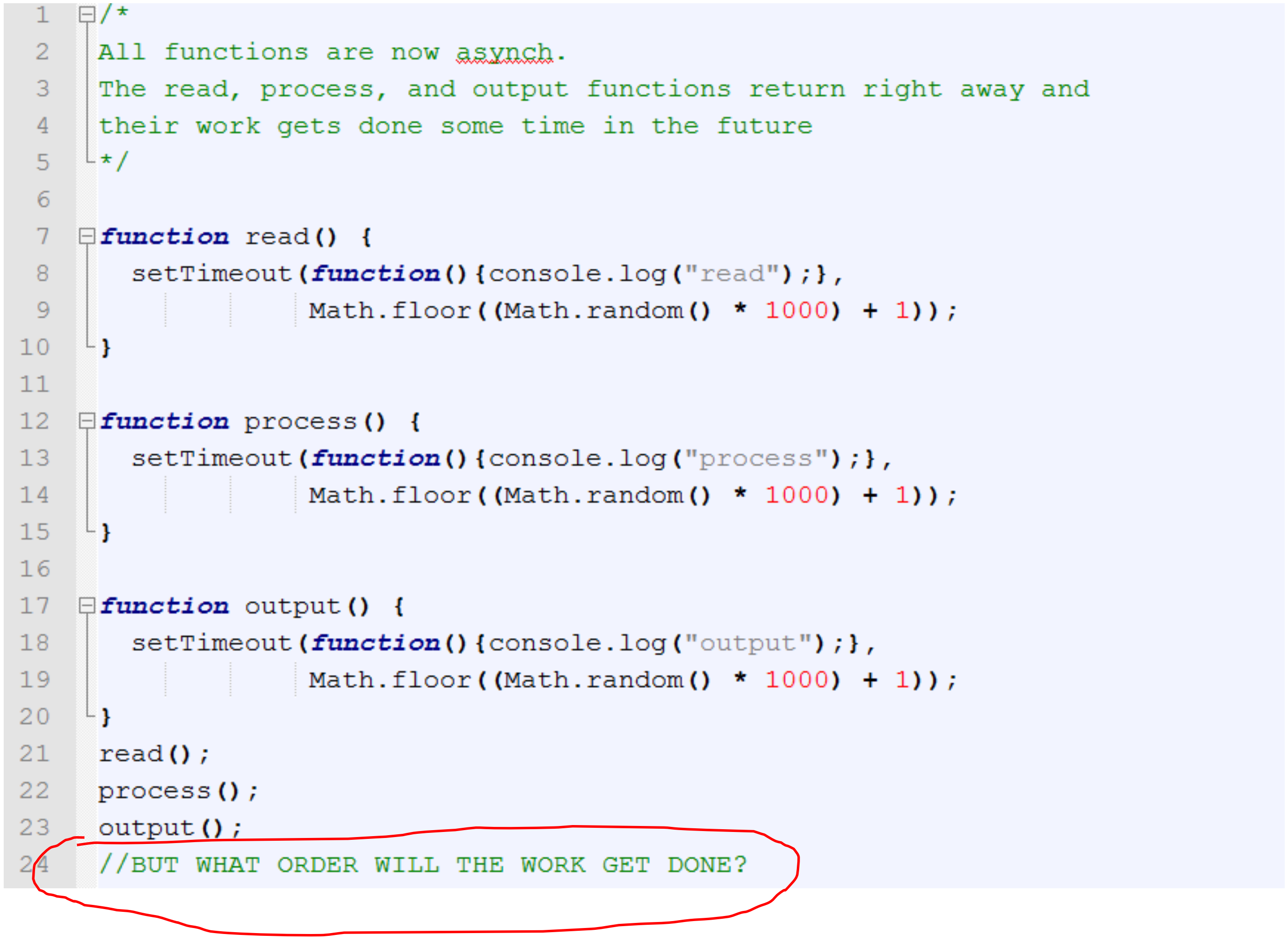
Figure 2-1. Comparing common I/O sources

Beginning Node.js

Copyright © 2014 by Basarat Ali Syed

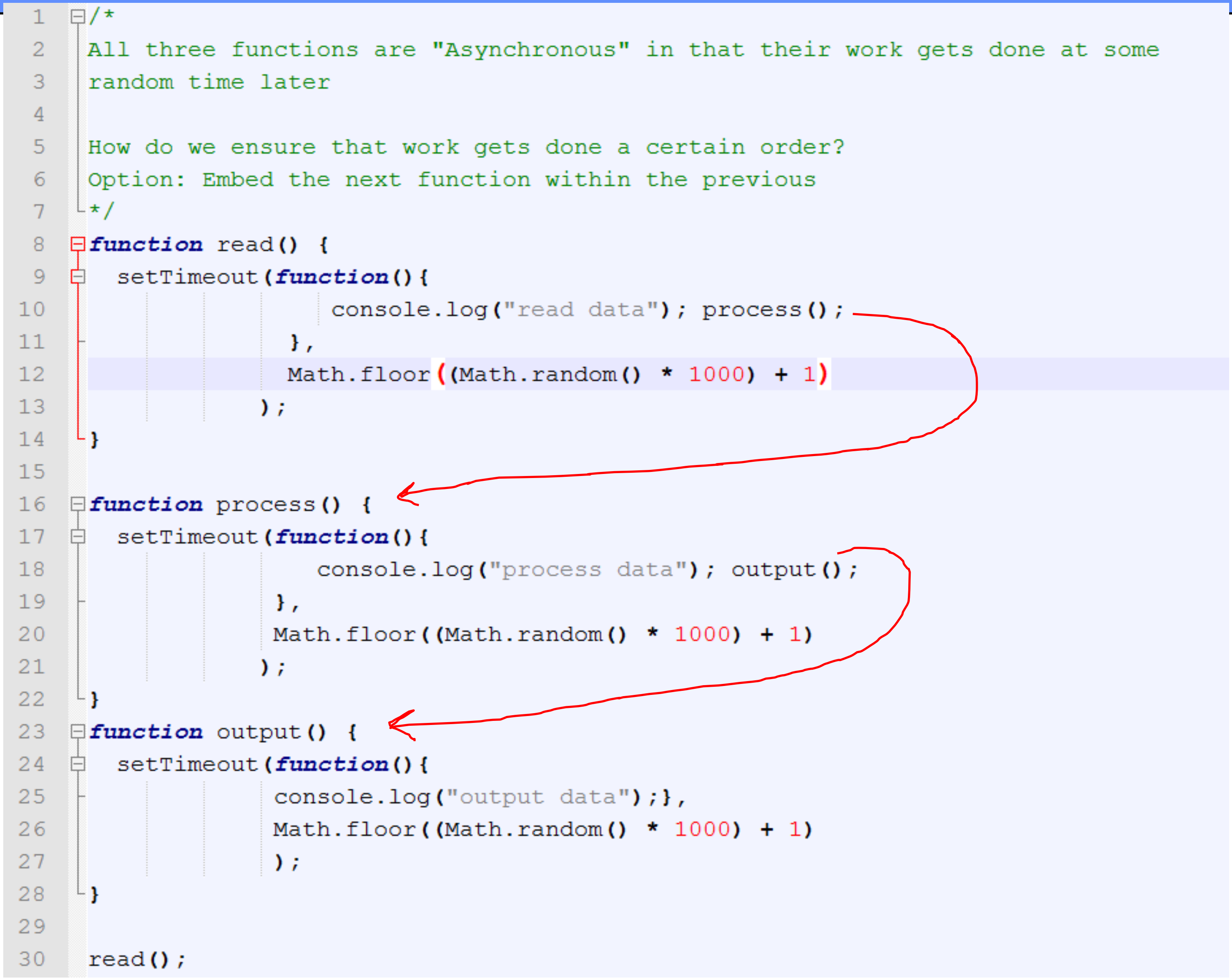
Asynchronous Functions

- **Order or exectution?**



```
/*
All functions are now asynch.
The read, process, and output functions return right away and
their work gets done some time in the future
*/

function read() {
   setTimeout(function(){console.log("read");},
                  Math.floor((Math.random() * 1000) + 1));
}

function process() {
   setTimeout(function(){console.log("process");},
                  Math.floor((Math.random() * 1000) + 1));
}

function output() {
   setTimeout(function(){console.log("output");},
                  Math.floor((Math.random() * 1000) + 1));
}
read();
process();
output();
//BUT WHAT ORDER WILL THE WORK GET DONE?
```

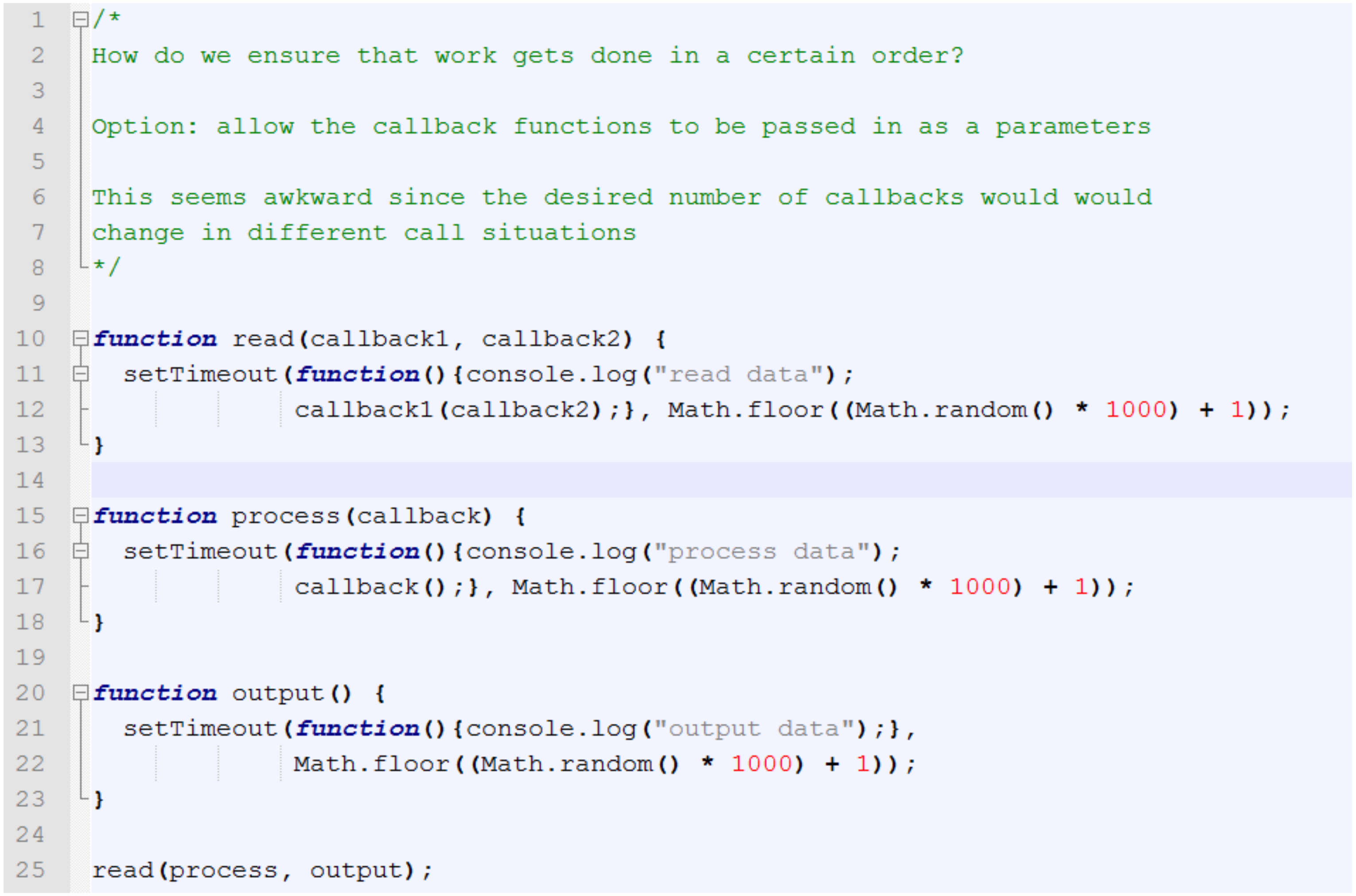
Asynchronous Functions



```
/*
All three functions are "Asynchronous" in that their work gets done at some
random time later

How do we ensure that work gets done a certain order?
Option: Embed the next function within the previous
*/
function read() {
  setTimeout(function(){
                 console.log("read data"); process();
              },
              Math.floor((Math.random() * 1000) + 1)
            );
}

function process() {
  setTimeout(function(){
                console.log("process data"); output();
             },
             Math.floor((Math.random() * 1000) + 1)
            );
}
function output() {
  setTimeout(function(){
             console.log("output data");},
             Math.floor((Math.random() * 1000) + 1)
             );
}

read();
```

Asynchronous Functions

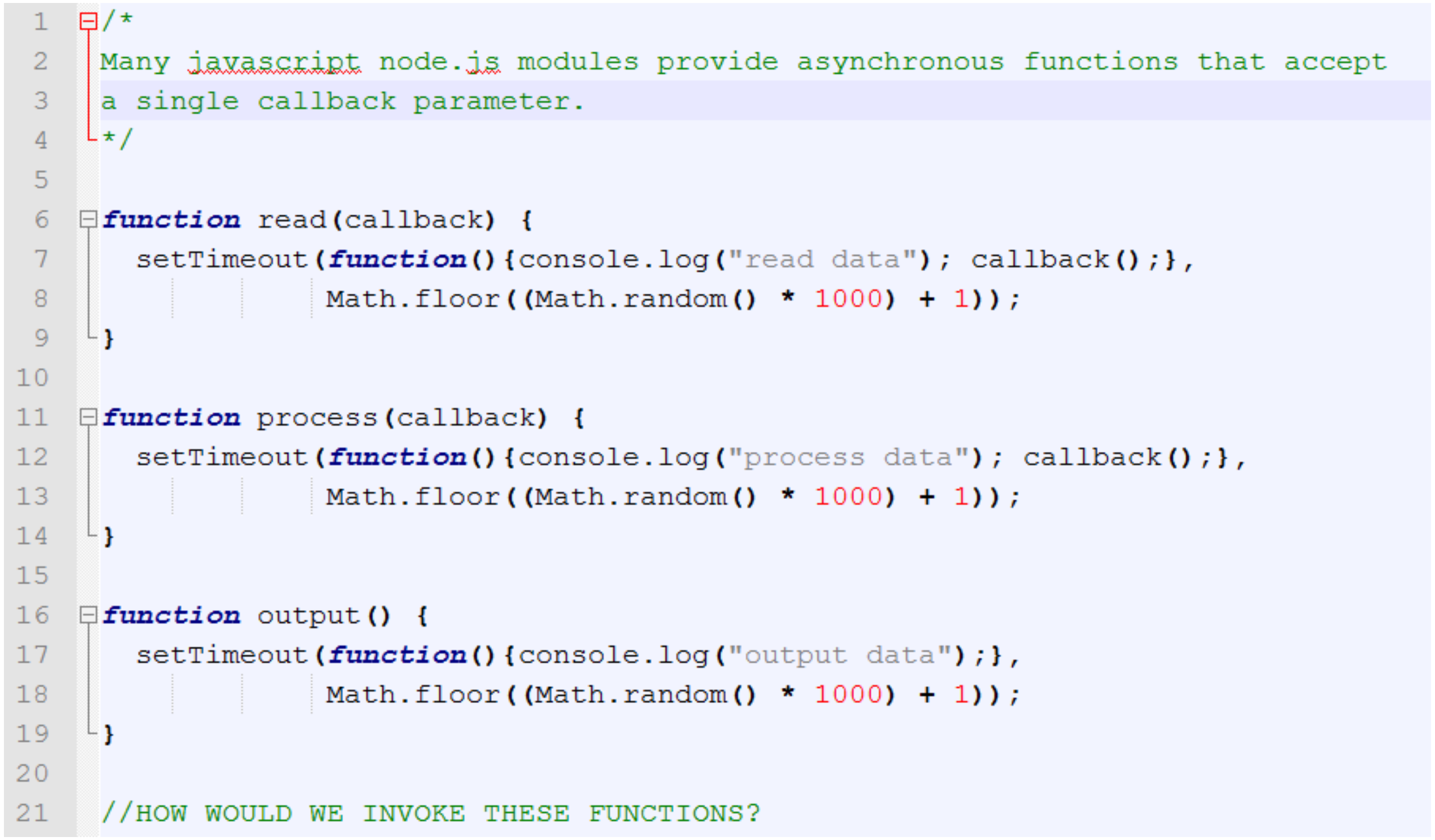
- **Parameterized Callbacks**



```
/*
How do we ensure that work gets done in a certain order?

Option: allow the callback functions to be passed in as a parameters

This seems awkward since the desired number of callbacks would would
change in different call situations
*/

function read(callback1, callback2) {
  setTimeout(function(){console.log("read data");
             callback1(callback2);}, Math.floor((Math.random() * 1000) + 1));
}

function process(callback) {
  setTimeout(function(){console.log("process data");
             callback();}, Math.floor((Math.random() * 1000) + 1));
}

function output() {
  setTimeout(function(){console.log("output data");},
             Math.floor((Math.random() * 1000) + 1));
}

read(process, output);
```

Asynchronous Functions

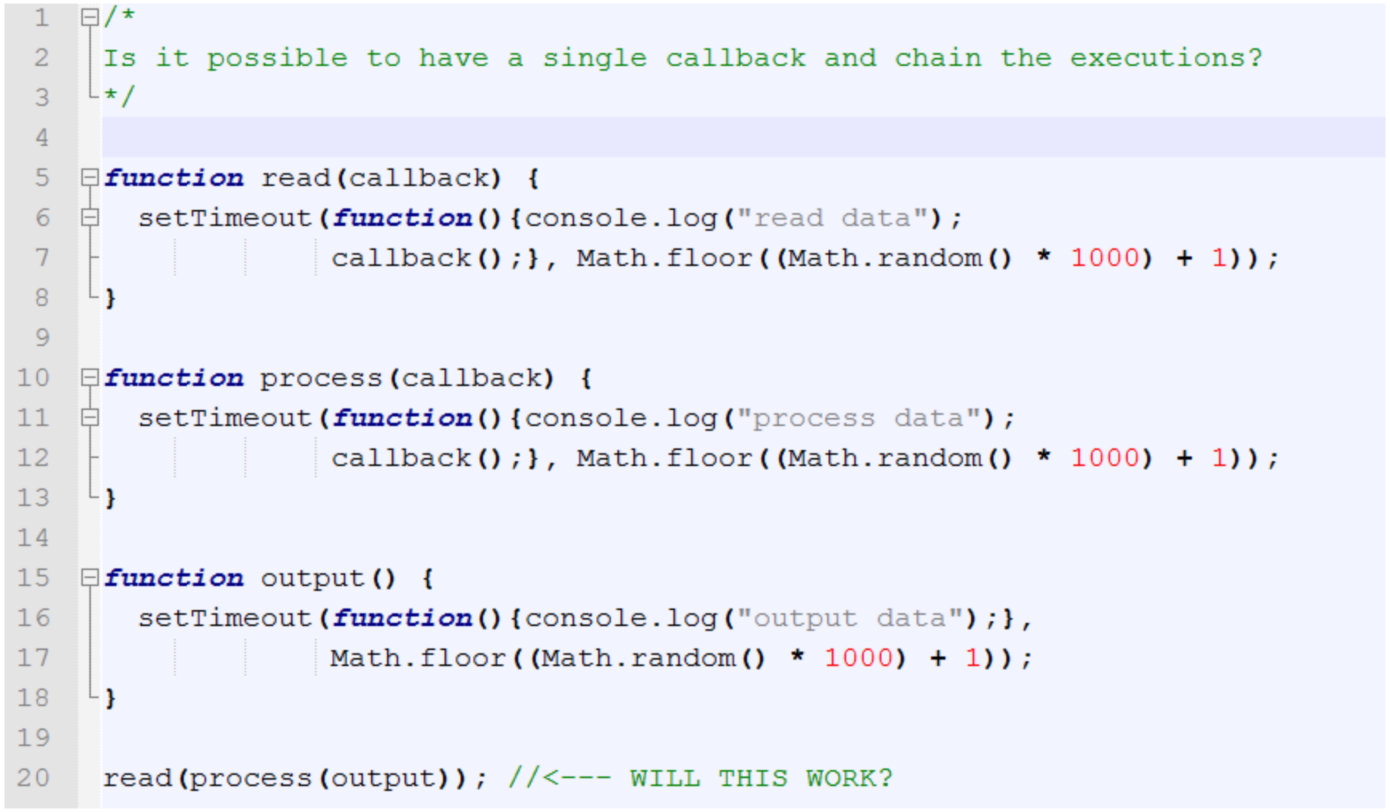
- **Single Callback function parameter**



```
/*
Many javascript node.js modules provide asynchronous functions that accept
a single callback parameter.
*/

function read(callback) {
  setTimeout(function(){console.log("read data"); callback();},
              Math.floor((Math.random() * 1000) + 1));
}

function process(callback) {
  setTimeout(function(){console.log("process data"); callback();},
              Math.floor((Math.random() * 1000) + 1));
}

function output() {
  setTimeout(function(){console.log("output data");},
              Math.floor((Math.random() * 1000) + 1));
}

//HOW WOULD WE INVOKE THESE FUNCTIONS?
```

Asynchronous Functions

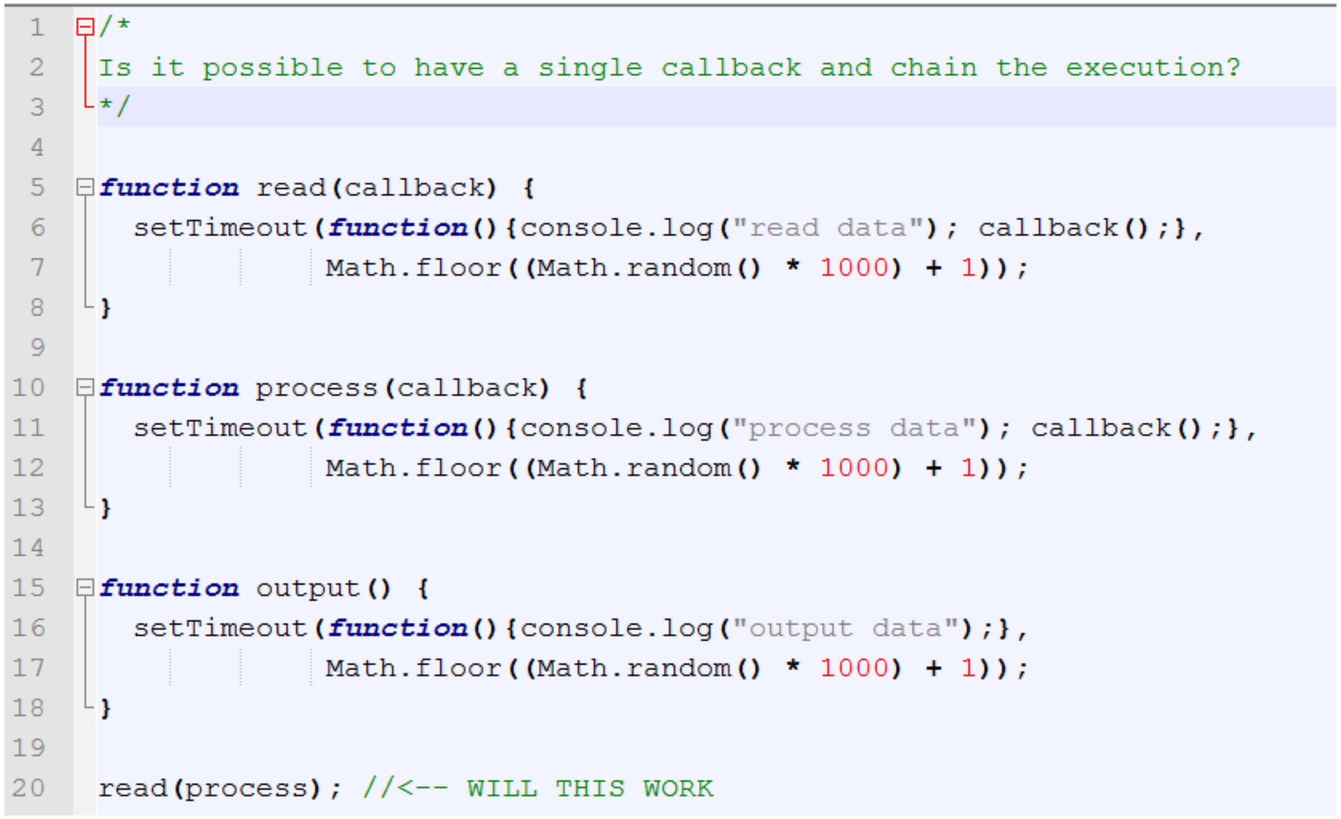
- **Single Callback parameter –chaining executions**



```
/*
Is it possible to have a single callback and chain the executions?
*/

function read(callback) {
  setTimeout(function(){console.log("read data");
          callback();}, Math.floor((Math.random() * 1000) + 1));
}

function process(callback) {
  setTimeout(function(){console.log("process data");
          callback();}, Math.floor((Math.random() * 1000) + 1));
}

function output() {
  setTimeout(function(){console.log("output data");},
          Math.floor((Math.random() * 1000) + 1));
}

read(process(output)); //<--- WILL THIS WORK?
```

Asynchronous Functions

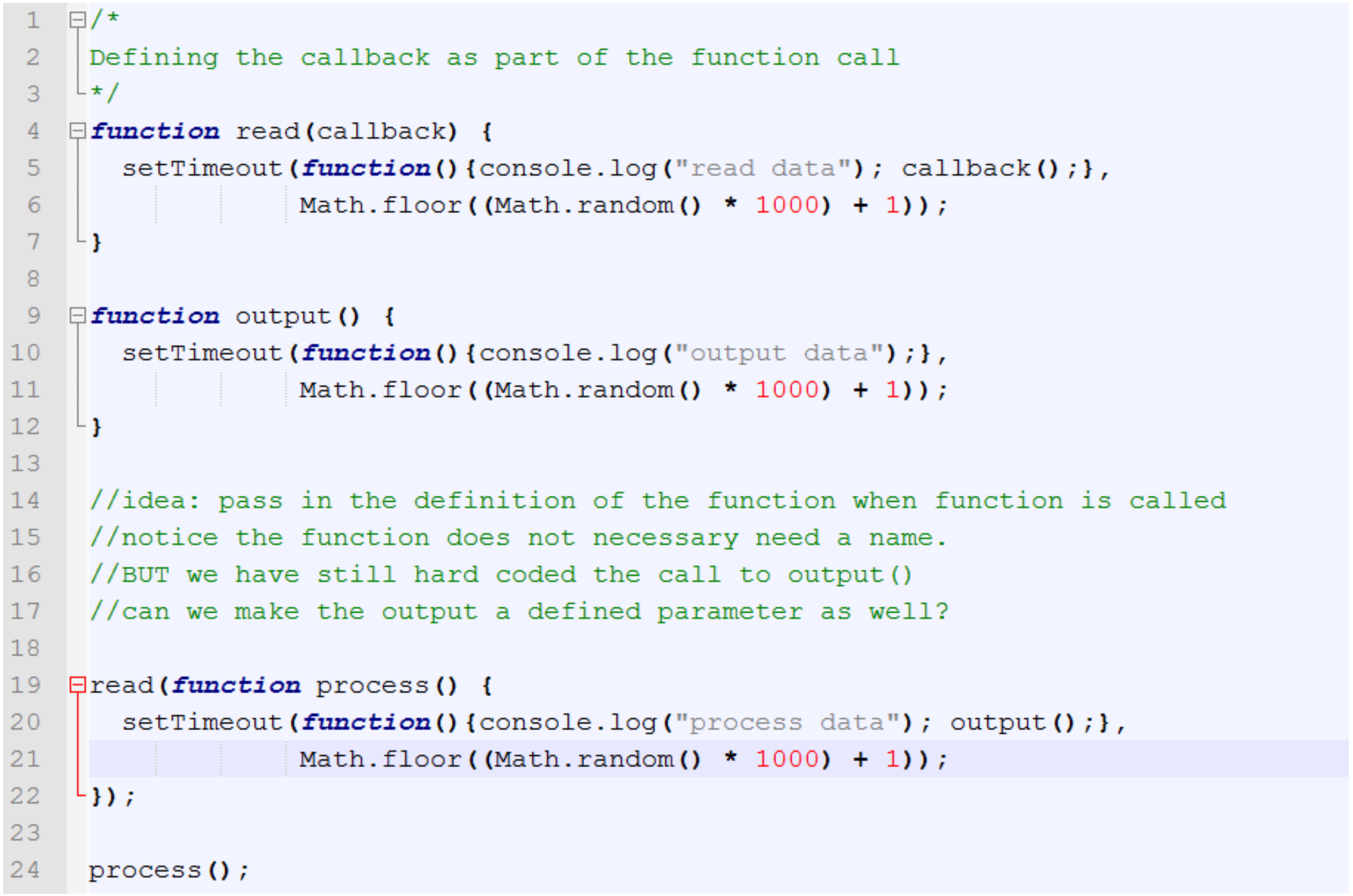
- **Single Callback parameter –chaining executions**



```
/*
Is it possible to have a single callback and chain the execution?
*/

function read(callback) {
  setTimeout(function(){console.log("read data"); callback();},
          Math.floor((Math.random() * 1000) + 1));
}

function process(callback) {
  setTimeout(function(){console.log("process data"); callback();},
          Math.floor((Math.random() * 1000) + 1));
}

function output() {
  setTimeout(function(){console.log("output data");},
          Math.floor((Math.random() * 1000) + 1));
}

read(process); //<-- WILL THIS WORK
```

Asynchronous Functions

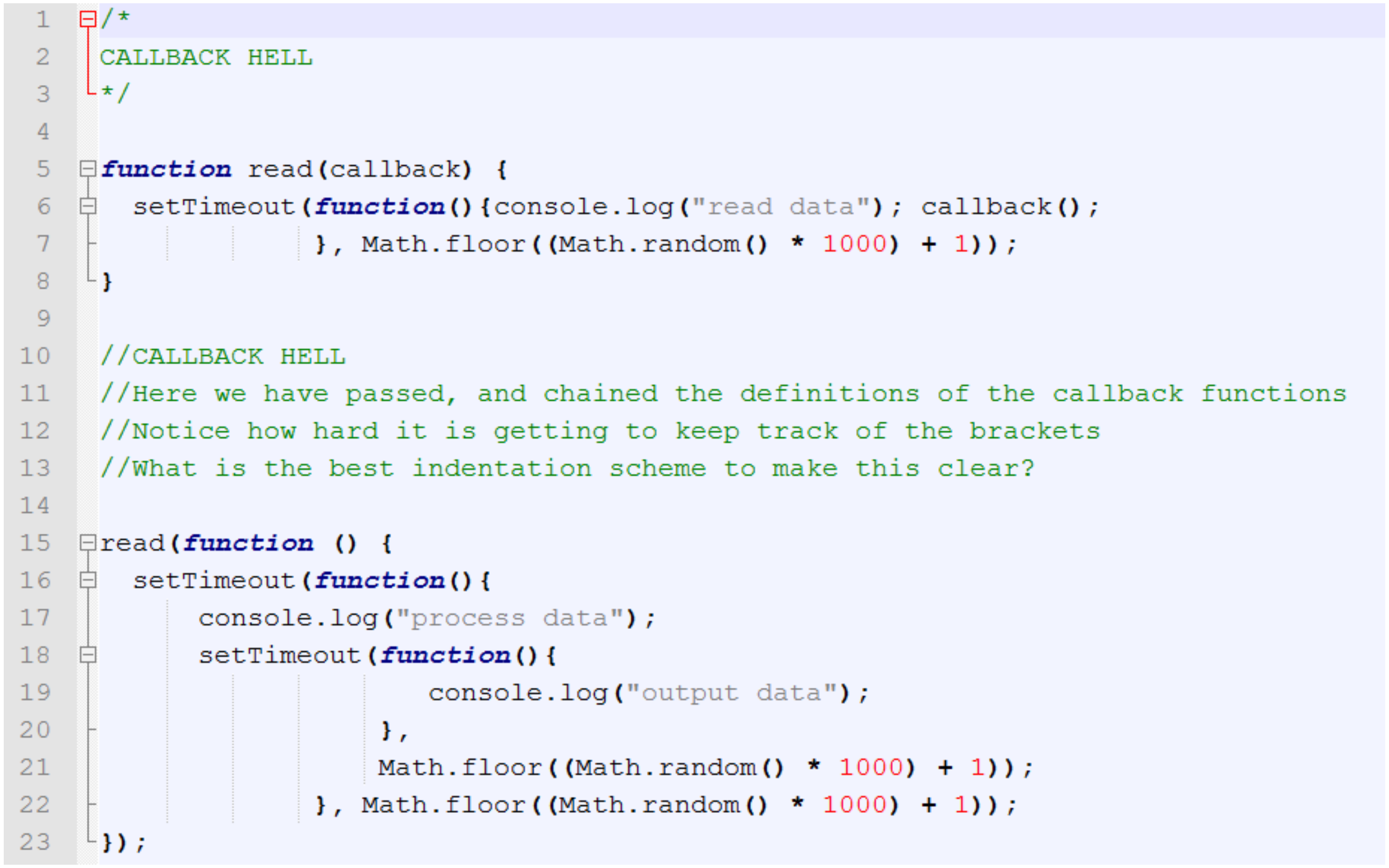
- **Passing function definition as callback**



```
/*
Defining the callback as part of the function call
*/
function read(callback) {
  setTimeout(function(){console.log("read data"); callback();},
             Math.floor((Math.random() * 1000) + 1));
}

function output() {
  setTimeout(function(){console.log("output data");},
             Math.floor((Math.random() * 1000) + 1));
}

//idea: pass in the definition of the function when function is called
//notice the function does not necessary need a name.
//BUT we have still hard coded the call to output()
//can we make the output a defined parameter as well?

read(function process() {
  setTimeout(function(){console.log("process data"); output();},
             Math.floor((Math.random() * 1000) + 1));
});

process();
```

Asynchronous Functions

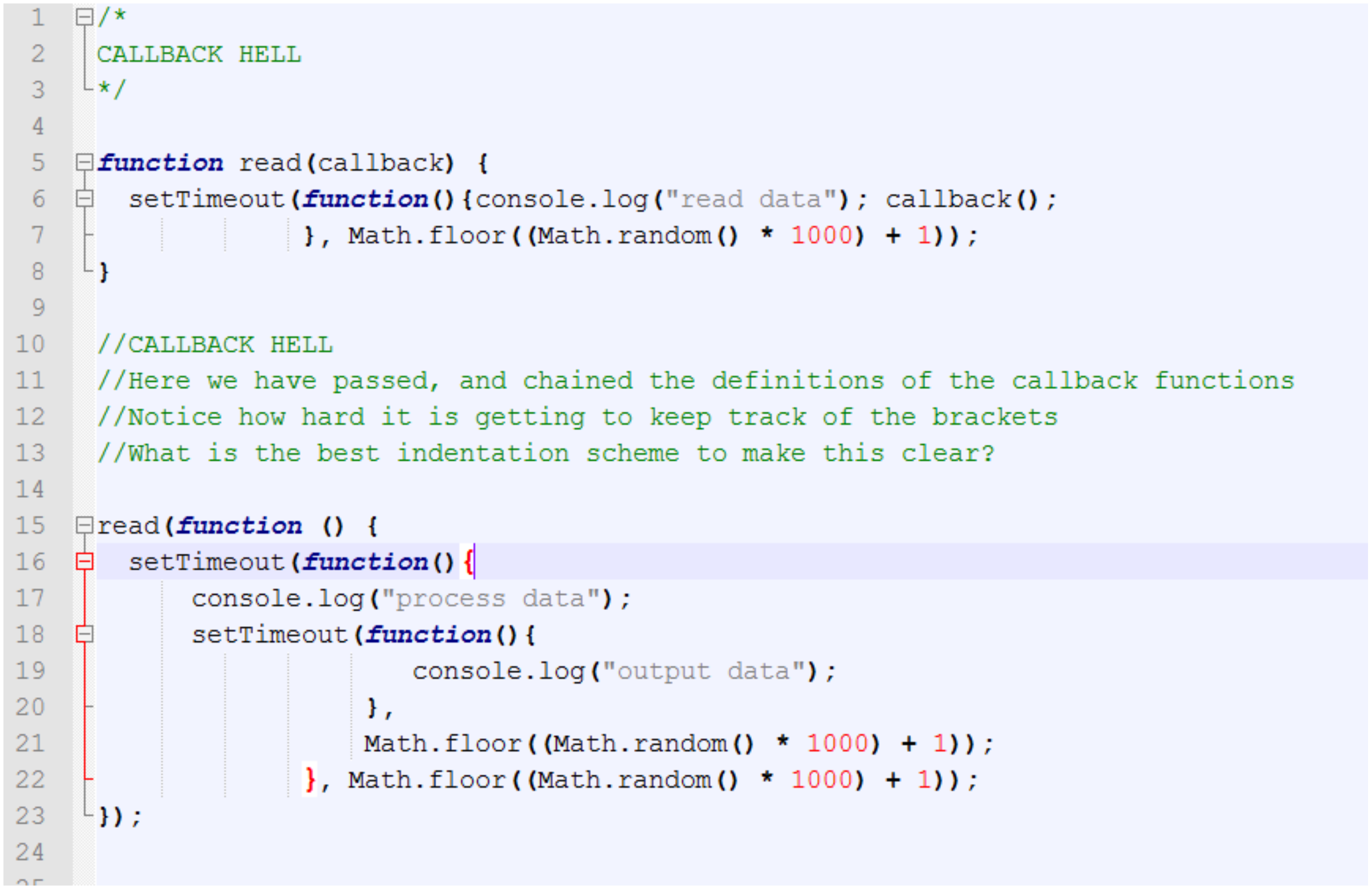
- **"Callback Hell"**



```
/*
CALLBACK HELL
*/

function read(callback) {
  setTimeout(function(){console.log("read data"); callback();
            }, Math.floor((Math.random() * 1000) + 1));
}

//CALLBACK HELL
//Here we have passed, and chained the definitions of the callback functions
//Notice how hard it is getting to keep track of the brackets
//What is the best indentation scheme to make this clear?

read(function () {
  setTimeout(function(){
      console.log("process data");
      setTimeout(function(){
                    console.log("output data");
                 },
                 Math.floor((Math.random() * 1000) + 1));
            }, Math.floor((Math.random() * 1000) + 1));
});
```

matching brackets

- **Suggestion: use a editor that allows you to easily see matching brackets (here Notepad++)**



```
/*
CALLBACK HELL
*/

function read(callback) {
  setTimeout(function(){console.log("read data"); callback();
            }, Math.floor((Math.random() * 1000) + 1));
}

//CALLBACK HELL
//Here we have passed, and chained the definitions of the callback functions
//Notice how hard it is getting to keep track of the brackets
//What is the best indentation scheme to make this clear?

read(function () {
  setTimeout(function() {
      console.log("process data");
      setTimeout(function(){
                   console.log("output data");
                },
                Math.floor((Math.random() * 1000) + 1));
            }, Math.floor((Math.random() * 1000) + 1));
});
```

Battling Callback Hell

There are several techniques used to battle "Callback Hell".

We will encounter them during the course.

These include:

- Event Emitters/Handlers
- Middleware and Routing -Express.js
- Promises –ES6
- Async/Await –ES6

BUT always be able to handle cases that don't use fancy features. That is, be able to deal with "callback hell".