Advanced JavaScript

ITM - SDM S9

EPITA

Asynchrony

One of the most important part and also misunderstood in JavaScript is how express and manipulate program behavior spread out over period of time.

What's happens when part of your program runs *now*, and another part of your program runs *later*. There is a gap between *now* and *later* where your program is not actively executing

The relationship between the *now* and *later* parts of your program is at the heart of asynchronous programming.

We'll explore a lot of techniques for async JavaScript programming.

A Program in Chunks

We can write JS program in one *.js* file but this program will have several chunks, only of which is going to execute *now*, and the rest of which will execute *later*. The most common unit of *chunk* is the `function`.

Later chunk of code doesn't happen strictly and immediately after *now*. Tasks cannot complete *now* are, by definition, going to complete asynchronously, and thus we will not have blocking behavior as you might intuitively expect or want.

```
// ajax(..) is some arbitrary Ajax function given by a library
var data = ajax( "http://some.url.1" );

console.log( data );
// Oops! `data` generally won't have the Ajax results
```

Ajax request doesn't complete synchronously, which means the `ajax(..)` function does not yet have any value to return back to be assigned to `data` variable. If `ajax(..)` could block until the response came back, then the `data = ..` the assignment would work fine.

To make it work, we make an asynchronous Ajax request *now*, and we won't get the results back until *later*.

The way of "waiting" from *now* until *later* is to use a function, commonly called a callback function:

```
// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", function myCallbackFunction(data){

    console.log( data ); // Yay, I gots me some `data`!

} );
```

Event Loop

The JS engine does anything more than execute a single chunk of your program at any given moment, when asked to environment.

JavaScript runs inside a *hosting environment*, which is for most developers the typical web browser. But there is not only web browsers, JavaScript has expanded to other environments, such as servers, via things like Node.js.

These environments have a mechanism in them that handles executing multiple chunks of your program *over time*, at each moment invoking the JS engine called the "event loop".

For example, when our JS program makes an Ajax request to fetch some data from a server, we set up the response code in a function callback, and JS engine tells the hosting environment, "Hey, I'm going to suspend execution for now, but whenever you finish with that network request, and you have some data, please *call* this function *back*"

The browser is on pending for the response from the network , and when it has something to give you, it schedule, the callback function to be executed by inserting it into the *event loop*.

Here it is fakish code to represented event loop:

```
// `eventLoop` is an array that acts as a queue (first-in, first-out)
var eventLoop = [ ];
var event;

// keep going "forever"
while (true) {
    // perform a "tick"
    if (eventLoop.length > 0) {
        // get the next event in the queue
        event = eventLoop.shift();

        // now, execute the next event
        try {
            event();
        }
        catch (err) {
            reportError(err);
        }
    }
}
```

As you can see `event loop` is infinite loop `while`, and each iteration is called a "tick". For each tick if event is waiting on the queue, it's taken off and executed. *These events are your function callbacks*.

For function `setTimeout(..)`, your callback function put the event loop only when the timer expires. The environment places your callback into the event loop, such that some future tick will pick it up and execute it.

Let's suppose, there are 20 items in the event loop at that moment, your callback waits. It gets in line behind the others. there's not normally a path for preempting the queue and skipping ahead in line

Our program is generally broken up in lots of small chunks , which happen one after the other in the event loop queue.

Parallel Threading

It's very easy to confuse the terms "async" and "parallel", but they are slightly different.

Async is the gap between *now* and *later*.

Parallel is about things being able to occur simultaneously.

For parallel computing, the most common tools are **processes** and **threads**. They execute independently and may execute simultaneously: on separate processors, or even separate computer

The interleaving of parallel threads of execution and the interleaving of asynchronous events occur at very different levels of granularity.

In single-threaded environment, it really does not matter that items in the thread queue are low-level operations because nothing can interrupt the thread.

But if we have a parallel system, where two different threads are operating in the same, you could very likely have unpredictable behavior.

```
var a = 20;

function foo() {
    a = a + 1;
}

function bar() {
    a = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

In JS's single-threaded behavior, if `foo()` runs before `bar()`, the result is that `a` has `42`, but if `bar()` runs before `foo()` the result in `a` will be `41`.

We consider two threads that represented respectively `foo()` and `bar()`

Thread 1(`X` and `Y` are temporary memory locations):

```
foo():
  a. load value of `a` in `X`
  b. store `1` in `Y`
  c. add `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```

Thread 2(`X` and `Y` are temporary memory locations):

```
bar():
  a. load value of `a` in `X`
  b. store `2` in `Y`
  c. multiply `X` and `Y`, store result in `X`
  d. store value of `X` in `a`
```

What do you happen if two threads run in parallel ?

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
1b  (store `1` in `Y`   ==> `1`)
2b  (store `2` in `Y`   ==> `2`)
1c  (add `X` and `Y`, store result in `X`   ==> `22`)
1d  (store value of `X` in `a`   ==> `22`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `44`)
2d  (store value of `X` in `a`   ==> `44`)
```

The result in a will be 44

```
1a  (load value of `a` in `X`   ==> `20`)
2a  (load value of `a` in `X`   ==> `20`)
2b  (store `2` in `Y`   ==> `2`)
1b  (store `1` in `Y`   ==> `1`)
2c  (multiply `X` and `Y`, store result in `X`   ==> `20`)
1c  (add `X` and `Y`, store result in `X`   ==> `21`)
1d  (store value of `X` in `a`   ==> `21`)
2d  (store value of `X` in `a`   ==> `21`)
```

The result in a will be 21.

Threaded programming is very tricky, because if you don't take special steps to prevent this kind of interruption/interleaving from happening, you can get very surprising, nondeterministic behavior that frequently leads to headaches.

JS never shares data across threads, which means that level of nondeterminism isn't a concern.

But that does not mean JS is always deterministic.

Run-to-completion

```
var a = 1;
var b = 2;

function foo() {
    a++;
    b = b * a;
    a = b + 3;
}

function bar() {
    b--;
    a = 8 + b;
    b = a * 2;
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

In the program above, the function `foo()` (and `bar()`) is atomic because Javascript's run single-threading. So, as soon as function `foo()` starts running, its code will finish before any of the code in `bar()` can run, or vice versa. This behavior is called **run-to-completion** behavior.

Because `foo()` can't interupted `bar()` and `bar()` can't interupted `foo()`, this programs has two outcomes depending on which statement running first. however if threading were present, and the individual statemeents in `foo()` and `bar()` could be interleaved, so the number of possible outcomes would be greatly increased.

We can separate this code in three chunks :

Chunk 1:

```
var a = 1;
var b = 2;
```

Chunk 2 (`foo()`):

```
a++;
b = b * a;
a = b + 3;
```

Chunk 2 (`bar()`):

```
b--;
a = 8 + b;
b = a * 2;
```

Chunk 1 is synchronous, so it happens `now`, but chunks 2 and 3 are asynchronous (happen `later`).

Chunks 2 and 3 may happen in either-first order, so there are two possible outcomes for this program.

Outcome 1:

```
var a = 1;
var b = 2;

// foo()
a++;
b = b * a;
a = b + 3;

// bar()
b--;
a = 8 + b;
b = a * 2;

a; // 11
b; // 22
```

Outcome 2:

```
var a = 1;
var b = 2;

// bar()
b--;
a = 8 + b;
b = a * 2;

// foo()
a++;
b = b * a;
a = b + 3;

a; // 183
b; // 180
```

This code is nondeterminism because there are two different outcomes for the same code.

Function-ordering nondeterminism is commomn term "race condition", as `foo()` and `bar()` are racing against too see which runs first.

Specifically, it's a "race condition" because you cannot predict reliably how a and b will turn out.

Concurrency

Let's imagine a site that displays a list of status updates that progressively loads as the user scrolls down the list. To make such a feature work correctly, two separate "processes" will need to be executing *simultaneously* (i.e., during the same window of time).

The first "process" will respond to `onscroll` events as they fire when the user has scrolled the page further down.

The second "process" will receive Ajax response back.

Concurrency is when two or more "processes" are executing simultaneously over the same period, regardless wheter the individual constituent operations happen in *parallel* or not. You can think of concurrency then as "process"-level parallelism, as opposed to operation-level parallelism.

Let's visualize each independent "process" as a series of events/operations:

"Process" 1 (`onscroll` events):

```
onscroll, request 1
onscroll, request 2
onscroll, request 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
onscroll, request 7
```

"Process" 2 (Ajax response events):

```
response 1
response 2
response 3
response 4
response 5
response 6
response 7
```

It's possible that an `onscroll` event and Ajax response could be executed at exactly the same moment. Let's visualize these events in a timeline:

```
onscroll, request 1
onscroll, request 2          response 1
onscroll, request 3          response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6          response 4
onscroll, request 7
response 6
response 5
response 7
```

But we saw it earlier, the event loop is able to handle one event at a time, so either `onscroll, request 2` is going to happen first or `response 1` is going to happen, but they cannot happen it the same moment.

Let's visualize the interleaving of all these events onto the event loop queue.

```
onscroll, request 1   <--- Process 1 starts
onscroll, request 2
response 1            <--- Process 2 starts
onscroll, request 3
response 2
response 3
onscroll, request 4
onscroll, request 5
onscroll, request 6
response 4
onscroll, request 7   <--- Process 1 finishes
response 6
response 5
response 7            <--- Process 2 finishes
```

"Process 1" and "Process 2" run concurrently, but their individual events run sequentially on the loop queue.

Non interaction

As two or more "processes" are interleaving their events concurrently in the same program. They don't necessarily need to interact with each other if the tasks are unrelated. **If they don't interact, nondeterminism is perfectly acceptable.**

Example:

```
var res = {};

function foo(results) {
    res.foo = results;
}

function bar(results) {
    res.bar = results;
}

//ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

These program is build in such way that it doesn't matter the process they fired in first. The result will always be the same.

Interaction

In lot of case, concurrent "processes" will need to interact with each other. So we need to coordinate these interactions to prevent "race conditions".

Here's a simple example of two concurrent "processes" that interact because of implied ordering, which is only *sometimes broken*:

```
var res = [];

function response(data) {
    res.push( data );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

The concurrent are two `response()` calls that will be made to handle Ajax responses. They can happen in either-first order.

Let's assume the expected behavior is that `res[0]` gets the of result of "http://some.url.1" call , and `res[1]` gets the of result of "http://some.url.1" call.

Sometimes the behavior expected will happen, but sometimes won't the case. It will depend on which call finishes first. There's une strong likelihood that this nondeterminism is a "race condition bug".

To resolve the race condition bug, we can coordinate ordering interaction:

```
var res = [];

function response(data) {
    if (data.url == "http://some.url.1") {
        res[0] = data;
    }
    else if (data.url == "http://some.url.2") {
        res[1] = data;
    }
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

In this program doesn't matter which Ajax response comes back first, we determine which position the response data should occupy in function by `data.url`.

res[0] will always hold the "http://some.url.1" results and res[1] will always hold the "http://some.url.2" results.

Through simple coordination, we eliminated the "race condition" nondeterminism.

Somm concurrency scenarios are *always broken* (not just *sometimes*) without coordinated interaction.

Example:

```
var a, b;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(y) {
    b = y * 2;
    baz();
}

function baz() {
    console.log(a + b);
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

In this example wheter `foo()` or `bar()` fires first, it will always cause `baz()` to run early (neither `a` or `b` will still be `undefined`), but the second call of `baz()` will work, as `a` and `b` will be available.

Here's one simple way to resolve such condition:

```
var a, b;

function foo(x) {
    a = x * 2;
    if (a && b) {
        baz();
    }
}

function bar(y) {
    b = y * 2;
    if (a && b) {
        baz();
    }
}

function baz() {
    console.log( a + b );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

The `if(a && b)` conditional around the `baz()` call is traditionnaly called "gate", we are not sure what order `a` and `b` will arrrive, but we wait for both of them to get there before we proceed to open the gate.

Another concurrency interaction condition you may run into is sometimes called a "race," It's characterized by "only the first one wins" behavior. Here, nondeterminism is acceptable, in that you are explicitly saying it's OK for the "race" to the finish line to have only one winner.

Consider the broken code:

```
var a;

function foo(x) {
    a = x * 2;
    baz();
}

function bar(x) {
    a = x / 2;
    baz();
}

function baz() {
    console.log( a );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

Whichever one (`foo()` or `bar()`) fires last will not only overwrite the assigned a value from the other, but it will also duplicate the call to `baz()` (likely undesired).

So, we can coordinate the interaction like this:

```
var a;

function foo(x) {
    if (a == undefined) {
        a = x * 2;
        baz();
    }
}

function bar(x) {
    if (a == undefined) {
        a = x / 2;
        baz();
    }
}

function baz() {
    console.log( a );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", foo );
ajax( "http://some.url.2", bar );
```

The if `(a == undefined)` conditional allows only the first of `foo()` or `bar()` through, and the second calls would just be ignored. There's just no virtue in coming in second place!

Cooperation

Another expression of concurrency coordination is called "cooperative concurrency". Its goal is take a long-running "process" and break it up into steps so other concurrent "processes" have a chance to interleave their operations into the event loop queue.

For example, we have a Ajax response handler that needs to iterate through a long list of results to transform the values.

```
var res = [];

// `response(..)` receives array of results from the Ajax call
function response(data) {
    // add onto existing `res` array
    res = res.concat(
        // make a new transformed array with all `data` values doubled
        data.map( function(val){
            return val * 2;
        } )
    );
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

Let's imagine, "http://some.url.1" gets its result in first, the entire list will be mapped into `res` all at once. If it's a few thousand or less records, it doesn't matter. But if it's say 10 millions records, that can take a while to run.

While such "process" is running, nothing else in the page can happen, including, no other response, no UI updates, typing, button clicking. It's very problematic.

A solution to resolve this problem, we can process these results in asynchronous steps, after each one "yielding" back to event loop to let other waiting events happen.

```
var res = [];

// `response(..)` receives array of results from the Ajax call
function response(data) {
    // let's just do 1000 at a time
    var chunk = data.splice( 0, 1000 );

    // add onto existing `res` array
    res = res.concat(
        // make a new transformed array with all `chunk` values doubled
        chunk.map( function(val){
            return val * 2;
        } )
    );

    // anything left to process?
    if (data.length > 0) {
        // async schedule next batch
        setTimeout( function(){
            response( data );
        }, 0 );
    }
}

// ajax(..) is some arbitrary Ajax function given by a library
ajax( "http://some.url.1", response );
ajax( "http://some.url.2", response );
```

We process the data set in maximum sized chunks of 1,000 items. We ensure a short running "process" , even if that means many more subsequent "processes", as the interleaving onto the event loop queue will give us a much more performant site/app.

We use the setTimeout(..0) (hack) for async scheduling, which basically just means "stick this function at the end of the current event loop queue."