UNIVERSITÀ DI TORINO

Async Programming in Javascript

Prof. Fabio Ciravegna
Dipartimento di Informatica
Università di Torino
fabio.ciravegna@unito.it

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing

UNIVERSITÀ DI TORINO

Whys and Whats

- Async programming is a strategy whereby the program is able to control occurrences of events outside the normal flow of the main thread
  - i.e. some code is executed when an event occurs rather than when the flow of programming requires it

- Typical example is the click of a button
  - this raises an event (buttonClick) which is intercepted by the program
    - to execute some code

UNIVERSITÀ DI TORINO

<HTML>
<head>…</head>
<body>
…
<button id='button'>
…
</body>
<script>

```
const btn = document.querySelector('button');
btn.addEventListener('click', () => {
  alert('You clicked me!');

});
```

</script>

never insert javascript code directly into an html file, always use separate javascript files!!!

The code running the alert is run every time the button is clicked

UNIVERSITÀ DI TORINO

Async Javascript

- Many Web API features use asynchronous code
  - especially those that access or fetch resources from external devices,
    - e.g. files from the network, data from databases, video streams from a web cam, etc.
- You should learn how to use synch Javascript because
  - computational resources are scarce,
  - battery power is scarce
  - computing is very expensive

UNIVERSITÀ DI TORINO

4 ways in Js

- Async operations are stored into an event queue
- The queue is dealt with after the main thread has finished processing
  - not in parallel as Js is single threaded!!
- The queued operations are completed as soon as possible
  - They return their results to the JavaScript environment
- 4 methods
  - Callbacks (classic)
  - Timeouts and Intervals (delayed and repeated execution)
  - Promises (new)
  - Await/async (newer)

Callbacks and Associated Hell

http://callbackhell.com/

UNIVERSITÀ DI TORINO

What are callbacks?

- Callbacks are used in asynchronous functions
  - These are functions that take some time to execute
- They are also used in event capturing
- When you call a normal function you can use its return value immediately:

```
var result = multiplyTwoNumbers(5, 10)
console.log(result)
// 50 gets printed out
```

  - But in async functions this cannot be done

```
var photo = downloadPhoto('http://coolcats.com/cat.gif')
// photo is 'undefined'!
```

  - the var photo is undefined until the file has finished downloading

ctd

- So instead you store the code that you want to execute at the end of the downloading as a callback function

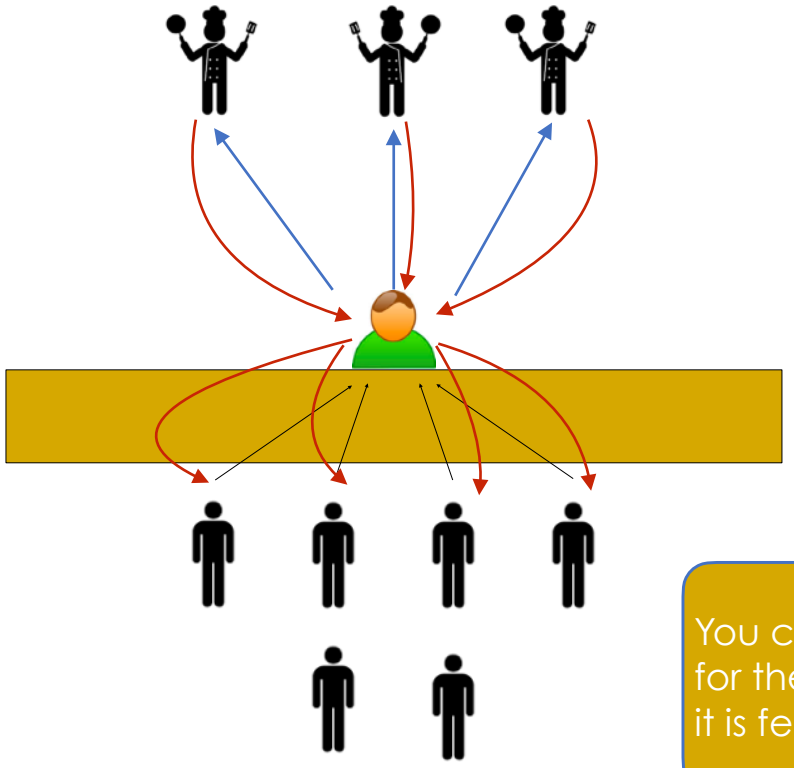
```
downloadPhoto('http://coolcats.com/cat.gif', handlePhoto)

function handlePhoto (error, photo) {
  if (error) console.error('Download error!', error)
  else console.log('Download finished', photo)
}

console.log('Download started')
```

- What you see on the log on console may seem strange:
  - **`Download started`**
  - …
  - **`Download finished`**
- but that is the point of async functions. They are executed later!

Event Driven and non-blocking

Remember the last lecture?

- This is very similar to a fast food outlet organisation



- Requests are posted to the till (the node.js server) which will direct them to the right cook (e.g. a database, file system, etc.)
- When a cook has prepared the food (the data), the counter (node.js server) will return it to the client
- While the food is being prepared, the till can serve other requests

You cannot start eating immediately after ordering. You must wait for the food to arrive first - equally you cannot use a photo before it is fetched

Nesting callbacks

- Issue is:
  - callbacks are often nested
  - example:
    - server is requested to get a list of images
    - actions:
      - retrieve the list of images path from the database (callback)
      - for each image path retrieve the image file (nested callback)
    - this will require nested callbacks

```
db.getImages(<fetch parameters>, function (req, res, error, data){
        for (image in data)
                fetch (image, function (req, res error, file){
                        …}
```

A very common error

- Some people will do this

This will not do what you expect (aka the code is wrong!!)
Also, I have not checked if the Javascript is completely correct - it is the spirit that is important

```
var listofphotos= [‘cat.gif’, ‘dog.gif’, ‘giraffe.gif’];
for (var ph as listofphotos)
   downloadPhoto(ph, function handlePhoto (error, photo) {
               if (error) console.error('Download error!', error)
               else photolist.push (photo);
}

zipPhotos(aCallbackofChoice, photolist);    // we suppose there is a callback
                                                  // function of choice defined
```

ziPhotos definition

```
var photolist= [];
// function that gets a list of photos taken from the file system and zips it
// and returns some metadata (e.g. average image size and number of photos) */
function zipPhotos(functionDoingStuff, photolist){
  var numOfPhotos=photolist.length;
  var totSize=0;
  for (var photo as photolist)
    totalSize+=getFileSize(photo);  // we suppose to have defined somewhere a
                                    // function getting the size of a file
  var averageSize=totalSize/numOfPhotos;
  var zippedPhotos= zipFiles(photolist); // we suppose to have defined
                                         // somewhere a
                                              // function zipping a set of files
  functionDoingStuff(null,{numOfPhotos: numOfPhotos, averageSize: averageSize
                           zippedPhotos: zippedPhotos});
  }
```

UNIVERSITÀ DI TORINO

Why?

- This code is wrong
  - zipPhotos will receive an empty list of files
  - zipPhotos will be called before all the callbacks have finished
- Even worse:
  - this code will seem to work
  - that is because sometimes - the download of the photos is fast enough for zipPhotos to find some files in the list, while the loop in zip photos is run, some photos are added to the variable list-photos
  - So the user thinks this is working but it will be by chance: very often lots of photos will be missed.
    - It is the worst that can happen to you as a programmer:
      - an inexplicably temperamental program with bugs you are unlikely to be able to recreate in testing conditions

But this is equivalent

- To go to the burger stall and thinking that you can start eating immediately after paying
  - Now, sometimes you could be able to do just that because maybe the fries are quick to come and after you have finished inputting your credit card pin code, they are already on the tray
  - So
    - you eat the fries
    - see the tray is empty
    - you leave
    - when the burger comes you have already left
      - (and you are still hungry!)
  - So remember to make sure that you wait for all your food before leaving

More clever people will do this!

This will not do what you expect (aka the code is wrong!!)

```
for (var ph as listofphotos)
   downloadPhoto(ph, function handlePhoto (error, photo) {
              if (error) console.error('Download error!', error)
              else {
                     photolist.push (photo);
                     zipPhotos(photolist, aCallbackofChoice);
```

- This is even worse
  - Your callback will be called several times, you will get a number of zipped files each containing one photo
    - if your callback returns to the client, it will always just return the zipped cat (and any followup callbacks will crash the system)

So how do you do it?

- Exactly as the burger joint does
- You keep a list of all the items you have to receive
- then you keep track of the items you have received
- when you have received them all, you leave

So how do you do that?

So how do you do that?

- You keep a list of all the items you have to receive
- When the callback realises that all the elements have been processed,
  - it will call the final function

```
for (var ph as listofphotos)
   downloadPhoto(ph, function handlePhoto (error, photo) {
            if (error) console.error('Download error!', error)
            else {
                  photolist.push (photo);
                  if (photolist.length==listofphotos.length)
                    zipPhotos(photolist, aCallbackofChoice);
```

now you are sure that photos contains all the pictures required

Actually that is not correct

```
for (var ph as listofphotos)
   downloadPhoto(ph, function handlePhoto (error, photo) {
            if (error) {
                  console.error('Download error!', error);
                  photolist.push (null);
            } else {
                  photolist.push (photo);
                  if (photolist.length==listofphotos.length)
                     zipPhotos(photolist, aCallbackofChoice);
```

Because otherwise, if there is an error, the server will never return

the function zipPhotos will have to filter out the null elements

And that was simple!

imagine you had to nest several layers of callback in callbacks…

```
chooseToppings(function(toppings) {
  placeOrder(toppings, function(order) {
    collectOrder(order, function(pizza) {
      eatPizza(pizza);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```


https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Promises

UNIVERSITÀ DI TORINO

How do I fix callback hell?

- Keep your code shallow

```
var form = document.querySelector('form')
form.onsubmit = function (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

This code has two anonymous functions. Let's give em names!

```
var form = document.querySelector('form')
form.onsubmit = function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, function postResponse (err, response, body) {
    var statusMessage = document.querySelector('.status')
    if (err) return statusMessage.value = err
    statusMessage.value = body
  })
}
```

UNIVERSITÀ DI TORINO

shallow (ctd)

As you can see naming functions is super easy and has some immediate benefits:

- makes code easier to read thanks to the descriptive function names
- when exceptions happen you will get stacktraces that reference actual function names instead of "anonymous"
- allows you to move the functions and reference them by their names

Now we can move the functions to the top level of our program:

```
document.querySelector('form').onsubmit = formSubmit

function formSubmit (submitEvent) {
  var name = document.querySelector('input').value
  request({
    uri: "http://example.com/upload",
    body: name,
    method: "POST"
  }, postResponse)
}

function postResponse (err, response, body) {
  var statusMessage = document.querySelector('.status')
  if (err) return statusMessage.value = err
  statusMessage.value = body
}
```

UNIVERSITÀ DI TORINO

Always handle errors

The first two rules are primarily about making your code readable, but this one is about making your code stable. When dealing with callbacks you are by definition dealing with tasks that get dispatched, go off and do something in the background, and then complete successfully or abort due to failure. Any experienced developer will tell you that you can never know when these errors happen, so you have to plan on them always happening.

With callbacks the most popular way to handle errors is the Node.js style where the first argument to the callback is always reserved for an error.

```
var fs = require('fs')

fs.readFile('/Does/not/exist', handleFile)

function handleFile (error, file) {
  if (error) return console.error('Uhoh, there was an error', error)
  // otherwise, continue on and use `file` in your code
}
```

Having the first argument be the `error` is a simple convention that encourages you to remember to handle your errors. If it was the second argument you could write code like `function handleFile (file) { }` and more easily ignore the error.

Code linters can also be configured to help you remember to handle callback errors. The simplest one to use is called **standard**. All you have to do is run `$ standard` in your code folder and it will show you every callback in your code with an unhandled error.

UNIVERSITÀ DI TORINO

Callbacks in callbacks

- The callback hell does not stop there
- You can understand the callback function and forget about it when you are inside a callback itself
- Example:
  - create a server that
    - retrieves a set of photos from another server
    - return the whole set of photos as one zipped file
- I am sure you know what to do by now
  - you will have two functions: one to retrieve the photos and one to zip them

UNIVERSITÀ DI TORINO

You must understand how to get out of the callback hell

So that you can sleep peacefully at night

UNIVERSITÀ DI TORINO

But you must avoid to find yourself in one at all cost

Although that is not always possible, we will now see a mechanism to avoid most (although not all) cases

UNIVERSITÀ DI TORINO

Questions?

© Prof. Fabio Cavegna, Università di Torino