Async programming

Asynchrony

> It’s easier to understand "asynchrony" if you first understand what “synchrony”, the opposite, means.

Synchronous code

> Synchronous code as “a bunch of statements in sequence”; so each statement in your code is executed one after the other.

```
console.log('First');
console.log('Second');
console.log('Third');
// OUTPUT: "First" "Second" "Third"
```

Asynchronous code

> Asynchronous code takes statements outside of the main program flow, allowing the code after the asynchronous call to be executed immediately without waiting.

Let's see it in action

Consider:

```
var users = jQuery.get('//jsonplaceholder.typicode.com/users')
  .done(function(response){
    console.log('first log: ' + response.length)
  })

console.log('second log: ' + users.length)
```

Output:

```
"second log: undefined"
"first log: 10"
```

- Logs are not made by order of code lines
- The result of `getUsers` is not an array of users

Code executed now:

```
var users = jQuery.get('//jsonplaceholder.typicode.com/users')
  .done( ... )
console.log('second log: ' + users.length)
```

Code executed later:

```
function(response){
  console.log('first log: ' + response.length)
}
```

This is a callback that will be executed only when `jsonplaceholder.typicode.com` responds with data.

> Asynchrony is essential for activities that are potentially blocking.
>
> While browser is waiting for a response from the web service, code execution is not blocked and keeps executing the rest of the lines.

Going further

Consider:

```
function log (content) {
  console.log(content)
}
function printing() {
   log(1);
```

```
    setTimeout(function callback1() { log(2); }, 0);
    setTimeout(function callback2() { log(3); }, 1000);
    log(4);
}
printing();
```

Output:

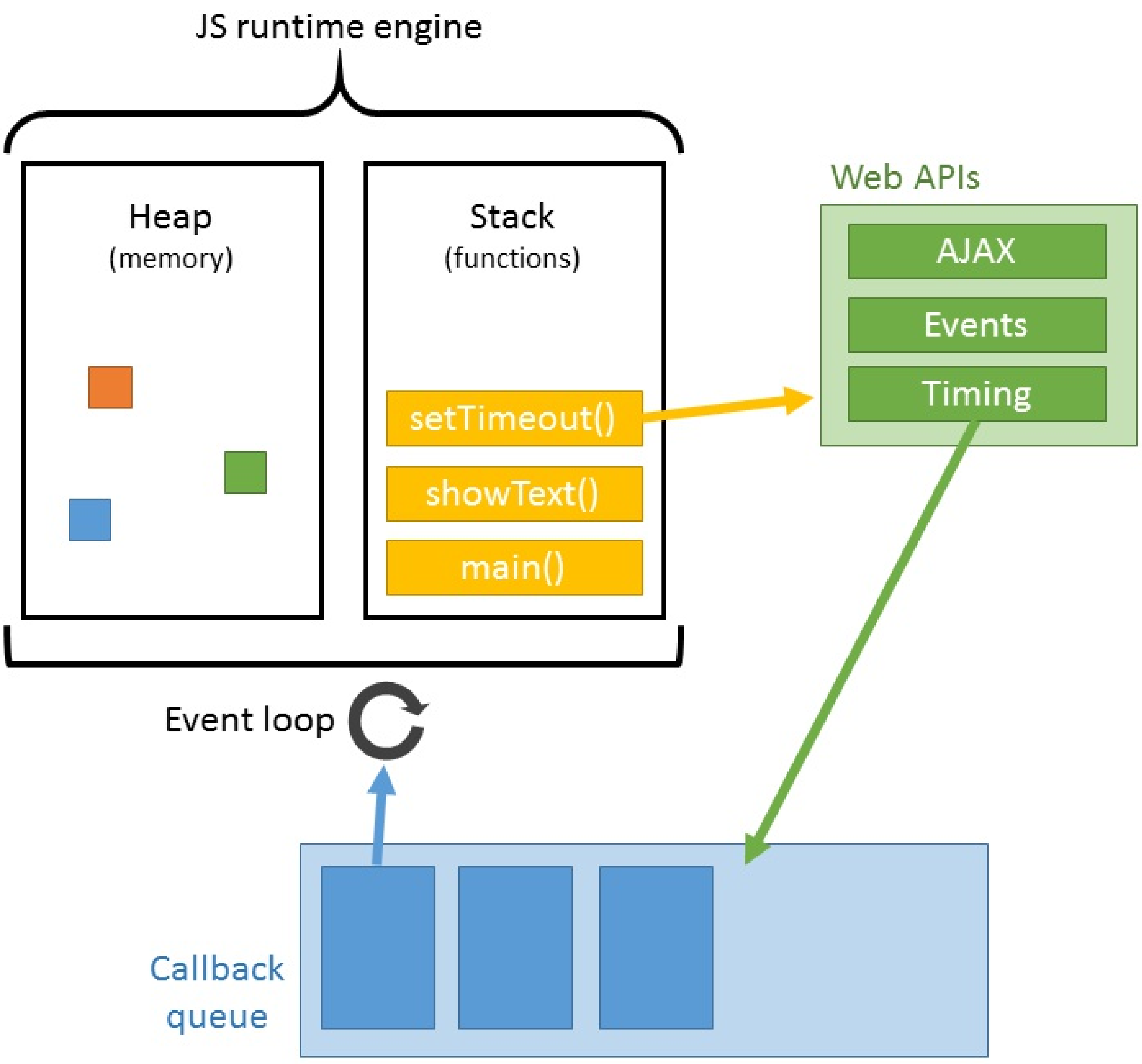
```
1
4
2
3
```

Why is `3` after `4` if the timeout has no time to wait?

To get it, we need to get into runtime concepts.

Javascript engine is composed of:

- **Stack**: Function calls form a stack of frames.
- **Heap**: Objects are allocated in a heap which is just a name to denote a large mostly unstructured region of memory.
- **Queue**: A JavaScript runtime contains a message queue, which is a list of messages to be processed. **A function is associated with each message.** When the stack is empty, a message is taken out of the queue and processed.



Event loop

> The event loop got its name because of how it's usually implemented, which usually resembles:

```
while(queue.waitForMessage()){
  queue.processNextMessage();
}
```

`queue.waitForMessage` waits synchronously for a message to arrive if there is none currently.

Check the docs

Let's how it works with our code

```
function log (content) {
  console.log(content)
}
function printing() {
   log(1);
   setTimeout(function callback1() { log(2); }, 0);
   setTimeout(function callback2() { log(3); }, 1000);
   log(4);
}
printing();
```

http://latentflip.com/loupe/

Making the synchronous, asynchronous

> Sometimes, synchronous heavy tasks blocks runtime until the end. It can really damage user's experience.

But now that you understand asynchony in javascript, you can take advantage of it...

Consider:

```
function traverseRecursion (current, depth) {
  var children = current.childNodes
  for (var i = 0, len = children.length; i < len; i++) {
    traverseRecursion(children[i], depth + 1)
  }
}
```

Make it asynchronous, non blocking:

```
function traverseRecursion (current, depth) {
  var children = current.childNodes
  for (var i = 0, len = children.length; i < len; i++) {
    setTimeout(
      (function(current, depth) {
        traverseRecursion(current, depth)
      })(children[i], depth + 1)
    , 0)
    }
  }
```

https://jsbin.com/fupuveh/9/edit?js,console

Callbacks

> Callbacks are the fundamental unit of asynchrony in JS.
>
> Instead of immediately returning some result like most functions, functions that use callbacks take some time to produce a result.

Practice

Implement `getUsersPhotos()` to retrieve photos of all users.

```
var getUsers = function (callback, limit) { ... }
var getUserAlbums = function (userId, callback, limit) { ... }
var getAlbumPhotos = function (albumId, callback, limit) { ... }

function getUsersPhotos(callback, limit) {
  // YOUR CODE GOES HERE
}
```

https://jsbin.com/rojomaf/edit?js,console,output

You've probably ended up with that kind of code

```
function getUsersPhotos(callback, limit) {
  var result = []
  var albumsLeftToProcess = 0
  getUsers(function(users){
    users.forEach(function(user){
      getUserAlbums(user.id, function(albums){
        albums.forEach(function(album){
          albumsLeftToProcess++
          getAlbumPhotos(album.id, function(photos){
            photos.forEach(function(photo){
              result.push(photo)
            })
            --albumsLeftToProcess || callback(result)
          }, limit)
        })
      }, limit)
    })
  }, limit)
}
```

Note: We call that the "Pyramid of Doom"

The problem with callbacks

Callback hell | Pyramid of doom

> The cause of callback hell is when people try to write JavaScript in a way where execution happens visually from top to bottom.

http://callbackhell.com/

Inversion of control

> Callbacks suffer from inversion of control in that they implicitly give control over to another party.
>
> This control transfer leads us to a troubling list of trust issues, such as whether the callback is called more times than we expect.

Promises

> Promises are now the official way to provide async return values in both JavaScript and the DOM.
>
> The Promise object is used for asynchronous computations. A Promise represents a value which may be available now, or in the future, or never.

Check the docs

Basics

Let's see how it looks like:

```
var promise = getUsers()
promise
  .then(function(users){
    console.log(users)
  }, function(e){
    console.error(e.message)
  })
```

Note: `getUsers` returns a promise. Promise always provides functions `then` and `catch`. `then` handler will be called with the value if the promise is resolved `catch` handler will be called with the value if the promise is rejected explicitly or any error occurs during the execution

---

States of a promise

- **pending** - Hasn't resolved or rejected yet
- **resolved** - The action relating to the promise succeeded
- **rejected** - The action relating to the promise failed
- **settled** - Has resolved or rejected

```
var promise = getUsers() // Promise is pending
promise
  .then(function(users){ // Promise is resolved and settled
    console.log(users)
  },
  function(e){ // Promise is rejected and settled
    console.error(e.message)
  })
```

Immutability of settled promises

- A promise is resolved with a **value**, passed to **resolution handler**

- A promise is rejected for a **reason** or **a thrown exception**, passed to the **rejection handler**

- Once a promise is settled (resolved or rejected), it's immutable and can't be **resolved** with a different value or **rejected** afterwards

Check full specs

Absence of race conditions

> **Race condition**: output is dependent on the sequence or timing of other uncontrollable events.

If the promise is settled when a corresponding handler is attached, the handler will be called.

Then, so there is no race condition between an asynchronous operation completing and its handlers being attached.

---

http get with callbacks

```
function reqListener () {
  console.log(this.responseText);
}

var oReq = new XMLHttpRequest();
oReq.addEventListener("load", reqListener);
oReq.open("GET", "http://www.example.org/example.txt");
oReq.send();
```

If `load` handler is attached **after** the response, the handler **will not** be executed

---

http get with promises

```
function reqListener (response) {
  console.log(response.text());
}

var oReq = fetch('flowers.jpg');

oReq.then(reqListener)
```

If `then` handler is attached **after** the response, the handler **will** be executed

Consume Promises

`Promise.prototype.then( onresolved, onRejected )`

> `promise.then` accepts both resolution and rejection handlers

```
getUsers() // returns a promise
  .then(function(users){
    console.log(users)
  }, function(e){
    console.error(e.message)
  })
```

Resolution handling

> If the promise returned by `getUsers` is resolved, resolution handler will be called with the value.

```
getUsers()
  .then(function(users){
    console.log(users)
  }, function(e){
    console.error(e.message)
  })
```

Rejection handling

> If the promise returned by `getUsers` is rejected, rejection handler will be called with a reason or `Error`.

```
getUsers()
  .then(function(users){
    console.log(users)
  }, function(e){
    console.error(e.message)
  })
```

Chaining

> `promise.then` always returns a new promise.
>
> The returned value in the attached handler will resolved a newly created promise.

```
getUsers() // promise 1
  .then(function(users){
    return users.filter( user => !!user.active )
  }) // promise 2
  .then(function(activeUsers){
    console.log(activeUsers)
  }) // promise 3
```

Rejection cascade

> As each `then` returns a **new independant promise**, the rejection handler is only triggered if something happens is the 'previous' promise.
>
> Also, the resolution handler will not be called on the next promise if first promise is rejected

---

```
getUsers() // promise 1
  .then(
    function onResolved1(users){
      throw new Error( 'No users' )
      return users.filter( user => !!user.active )
    },
    function onReject1(e) { console.error(e)} )
  .then(
    function onResolved2(activeUsers){
      console.log(activeUsers)
    },
    function onReject2(e) { console.error(e) })

// Only `onResolved1` and `onRejection2` will be called
```

Casting

> In a resolution handler, you can either return a plain value or **a new promise**

```
getUsers() // promise 1
```

```
    .then(function(users){
      return users.filter( user => !!user.active )
    }) // promise 2
    .then(function(activeUsers){
      return getAlbums( activeUsers[0].id ) // promise 3
    }) // promise 3
    .then(function(firstActiveUserAlbums){
      console.log(firstActiveUserAlbums)
    })
```

Create Promises

---

new Promise(function(resolve, reject){})

> In ES6, `Promise` is a new Object can instantiate
>
> A `resolve` and `reject` function are provided to either resolve or reject the promise.

```
var promise = new Promise(function(resolve, reject){
  if (true) {
    resolve(true)
  } else {
    reject('It\'s false ')
  }
})
```

Getting rid of callbacks with promises

This code:

```
var getUsers = function (callback, limit) {
  jQuery.get('//jsonplaceholder.typicode.com/users')
    .done(function (response) {
      callback(response.slice(0, limit))
    })
}

getUsers(function (users)
  console.log(users)
}, 5)
```

Becomes:

```
var getUsers = function (limit) {
  return new Promise(function(resolve, reject){
    jQuery.get('//jsonplaceholder.typicode.com/users')
      .done(function (response) {
        resolve(response.slice(0, limit))
      })
  })
}

getUsers(5)
  .then( function(users){
    console.log(users)
  })
```

Practice

Print the number of photos of a user.

```
var getOneUser = function () { ... }
var getUserAlbum = function (userId) { ... }
var getAlbumPhotos = function (albumId) { ... }

var printUserFirstPhotos = function(){
  // YOUR CODE GOES HERE
}

printUserFirstPhotos() // 50
```

https://jsbin.com/xofebas/8/edit?js,console

If you ended up with a code like that:

```
var printUserFirstPhotos = function(){
  getOneUser()
    .then(function(user){
      return getUserAlbum(user.id)
        .then(function(album){
          return getAlbumPhotos(album.id)
            .then(function(photos){
              console.log(photos.length)
            })
        })
    })
}
```

You are still stuck with pyramides and not understanding promises...

What about that ?

```
var printUserFirstPhotos = function(){
  getOneUser()
    .then( user => getUserAlbum(user.id) )
    .then( album => getAlbumPhotos(album.id) )
    .then( photos => console.log(photos.length) )
}
```

Or that ?

```
var printUserFirstPhotos = function(){
  getOneUser()
    .then( user => user.id )
    .then( getUserAlbum )
    .then( album => album.id )
    .then( getAlbumPhotos )
    .then( photos => photos.length )
    .then( console.log )
}
```

`Promise` static methods

- Promise.all()
- Promise.race()
- Promise.reject()
- Promise.resolve()

Promise.resolve(value)

> A static method to create a **promise resolved** with the given value (or another promise)

This code

```
var promise = Promise.resolve(5)
```

is the same as

```
var promise = new Promise(function(resolve){
  resolve(5)
})
```

```
var getSquare = function(x) {
  return Promise.resolve(x*x)
}

getSquare(4)
  .then( num => console.log(num) )
```

Promise.reject(reason)

> A static method to create a **promise reject** with the given reason

This code

```
var promise = Promise.reject('Some error happened')
```

is the same as

```
var promise = new Promise(function(resolve, reject){
  reject('Some error happened')
})
```

Promise.all(iterable)

> A static method that returns a promise that will be resolved in an array of values of all given promises in the array.

```
var promise = Promise.all([
  Promise.resolve(4),
  Promise.resolve(5),
  Promise.resolve("a"),
  Promise.resolve({})
])

promise.then( values => console.log(values) ) // [4,5,"6",{}]
```

`Promise.all` executes promises **in parallel**, not sequentially

```
var getUserPhotos = function (userId) {
  return getUser(userId)
    .then( user => getUserAlbum(user.id) )
    .then( album => getAlbumPhotos(album.id) )
    .then( photos => console.log(photos) )
}

Promise.all([
  getUserPhotos(2), getUserPhotos(4),
  getUserPhotos(5), getUserPhotos(8)
])
```

Promise.race(iterable)

> A static method that returns a promise that resolves or rejects as soon as **one of the promises** in the iterable resolves or rejects, with the value or reason from that promise.

```
var getTimeoutPromise = function(time) {
  return new Promise(function(resolve, reject){
    setTimeout(function(){
      reject('Timeout')
    }, time)
  })
}

var promise = Promise.race([
  getUserPhotos(),
  getTimeoutPromise(3000)
])
```

If `getUserPhotos()` lasts more than 3 seconds, `promise` will be rejected with reason "Timeout"

Remember: promises are... asynchronous

```
console.log('###### case 1 #####')
Promise.resolve(1)
  .then( x => console.log('then 1.0') )
  .then( x => console.log('then 1.1') )

console.log('###### case 2 #####')
Promise.resolve(1)
  .then( x => console.log('then 2.0') )
  .then( x => console.log('then 2.1') )

// OUTPUT
// "###### case 1 #####"
// "###### case 2 #####"
// "then 1.0"
// "then 2.0"
```

https://jsbin.com/kecemib/edit?js,console,output

Note: handlers are callbacks. Then, they get involved in the event loop.

Practice

Let's redo our practice about callbacks to play with promises.

```
var getUsers = function (limit) { /* Promise */ }
var getUserAlbums = function (userId, limit) { /* Promise */ }
var getAlbumPhotos = function (albumId, limit) { /* Promise */ }

function getUsersPhotos(limit) {
  // YOUR CODE GOES HERE
}

getUsersPhotos(6)
  .then( photos => console.log('Number of photos: ' + photos.length ) )
// OUTPUT : "Number of photos: 216"
```

https://jsbin.com/wemevaj/1/edit?js,console,output

Possible solution

```
function getUsersPhotos(limit) {
  return getUsers(limit)
    .then( users => users.map( user => getUserAlbums(user.id, limit) ) )
    .then( albumsPromises => Promise.all( albumsPromises ) )
    .then( usersAlbums => [].concat(...usersAlbums) )
    .then( albums => albums.map( album => getAlbumPhotos(album.id, limit) ) )
    .then( photosPromises => Promise.all( photosPromises ) )
    .then( albumsPhotos => [].concat(...albumsPhotos) )
}
```

Catching Rejections

`Promise.prototype.catch`

> The `catch()` method returns a Promise and deals with rejected cases only. It behaves the same as calling `Promise.prototype.then(undefined, onRejected)`.

```
getUsersPhotos(6)
  .then( photos => console.log('Number of photos: ' + photos.length ) )
  .catch( e => console.log('getUsersPhotos call failed') )
```

Check the docs

> `catch()` always returns **a promise**, like `then`

Let's see it in action:

https://jsbin.com/cibopuq/edit?js,console

> Be responsible. Catch your own errors and control your output.

```
function getUsersPhotos(limit) {
  return getUsers(limit)
    .then( users => users.map( user => getUserAlbums(user.id, limit) ) )
    .then( albumsPromises => Promise.all( albumsPromises ) )
    .then( usersAlbums => [].concat(...usersAlbums) )
    .then( albums => albums.map( album => getAlbumPhotos(album.id, limit) ) )
    .then( photosPromises => Promise.all( photosPromises ) )
    .then( albumsPhotos => [].concat(...albumsPhotos) )
```

```
        .catch( => return [])
    }
```

A thing to remember

> `catch()` is just sugar for `then(null, onRejection)``

This snippet...

```
getUsersPhotos(limit)
  .catch(onRejected)
```

... is **exactly THE SAME** as

```
getUsersPhotos(limit)
  .then (null, onRejected })
```

On the other hand. This snippet...

```
getUsersPhotos(limit)
  .then(onresolved)
  .catch(onRejected)
```

... is **NOT** the same as:

```
getUsersPhotos(limit)
  .then(onresolved, onRejected)
```

It's **exactly THE SAME** as

```
getUsersPhotos(limit)
  .then(onresolved)
  .then(null, onRejected)
```

Remember that the `onRejected` catches errors from **'previous'** promise that has not a **rejection handler**.

Master Promises

Old promises patterns you must avoid

> Promises have a long and storied history, and it took the JavaScript community a long time to get them right.

`finally` handler

> Handler exexuted whatever the promise is resolved or rejected.

```
showLoadingSpinner()
getUsersPhotos(6)
  .then( photos => console.log('Number of photos: ' + photos.length ) )
  .catch( e => console.log('getUsersPhotos call failed') )
  .finally( => hideLoadingSpinner() )
```

`progress` handler

> Handler to notify of value resolution progress.

```
getJSON().then(function(){ // resolution handler
  console.log('JSON loaded !')
},function(e){ // rejection handler
  console.log('Error !')
},function(progress){ // progress handler
  console.log( progress + '% loaded !')
})
```

The deferred pattern (deferred objects)

```
var deferred = Q.defer();
```

```
FS.readFile("foo.txt", "utf-8", function (error, text) {
    if (error) {
        deferred.reject(new Error(error));
    } else {
        deferred.resolve(text);
    }
});
return deferred.promise;
```

More promise treats

Convert callback functions to promises

```
FS.readFile("foo.txt", "utf-8", function (error, text) {
  /* ... */
});
```

```
var readFile = Q.denodeify(FS.readFile);
readFile("foo.txt", "utf-8")
  .then(onresolved)
  .catch(onRejected)
```

```
Q.nfcall(FS.readFile, "foo.txt", "utf-8")
  .then(onresolved)
  .catch(onRejected)
```

Q.js

Convert promises to callbacks based libs

```
var getUsers = function (callback, limit) {
  window.fetch('//jsonplaceholder.typicode.com/users')
    .asCallback(callback)
}

getUsers(function(err, result){ /* ... */ }, 5)
```

Bluebird

Delay with promises

```
Promise
  .delay(1000)
  .then( => getUsers() )
```

Bluebird

Further info on promises

Promise libraries

- RSVP.js
- Q.js
- Bluebird

Promise based new APIs

- Fetch API
- Service Worker API
- Battery Status API

Must Read/Watch

- Promise/A+ Specification

- Nolan Lawson - We have a problem with promises

- Fun Fun Function - Promises

- Jake Archivald - Tasks, microtasks, queues and schedules

- You-Dont-Know-JS - async & performance

Practice

Write `getFirstCharNumber` body code so it would return a promise resolved with the char number of the uppercase first letter of a given string.

```
function upper(text) { ... }

function firstChar(text) { ... }

function getChartCode(text) { ... }

getFirstCharNumber('abcde').then( console.log )

function getFirstCharNumber(text) {
  // YOUR CODE GOES HERE
}
```

https://jsbin.com/zidohun/edit?js,console,output

Possible solutions

```
function getFirstCharNumber(text) {
  return firstChar(text)
    .then( upper )
    .then( text => getChartCode(text)() )
}
```

```
function getFirstCharNumber(text) {
  return Promise.resolve( upper(text) )
    .then( text => getChartCode(text)() )
}
```

Implement `Promise.delay(ms)`

Returns a promise that will resolved after given milliseconds.

```
Promise.delay = function(ms){
  // YOUR CODE GOES HERE
}

Promise.delay(1000)
  .then(function(){
    console.log('delayed 1000ms')
  })

setTimeout(function(){
  console.log('delayed 500ms')
}, 500)

// OUTPUT "delayed 500ms" "delayed 1000ms"
```

https://jsbin.com/qidokig/edit?js,console,output

Solution

```
Promise.delay = function(ms){
  return new Promise(function(resolve){
    setTimeout(resolve, ms)
  })
}
```

Implement `Promise.series(iterable)`

Works like `Promise.all`, but executes the promises **sequentially** instead of in parallel.

```
var getDelayed = function(ms, name){ ... }

Promise.series = function(promises) {
  // YOUR CODE GOES HERE
}

Promise.series([
  getDelayed(500, 'promise 1'),
  getDelayed(400, 'promise 2'),
  getDelayed(300, 'promise 3')
])

// OUTPUT: "promise 1" "promise 2" "promise 3"
```

https://jsbin.com/mecalow/edit?js,console,output

Solution

NONE. You can't make promises change their execution order.

Once a promises is pending, it's in progress...