COESIONE ITALIA 21-27 PIEMONTE
Cofinanziato dall'Unione europea
REGIONE PIEMONTE

Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

Intervento realizzato da ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

Modern JavaScript

ES6+

Shadi Lahham - Web development

History

History of Javascript

- ES6 (ECMAScript 2015) ha introdotto una revisione importante del linguaggio, semplificando la sintassi e aggiungendo nuove funzionalità.

- ES6+ include tutte le versioni successive a ES5, come ES7, ES8, ecc.

- ECMAScript 2015 or ES6 was the second major revision to JavaScript
- It added a lot of features that change and simplify Javascript syntax

- ES6 and beyond, or ES6+, refers to all versions after ES5

ES6 - ECMAScript 6

Javascript version history

Let and const

Let vs var

Let e Const sono due nuovi modi per dichiarare variabili in JavaScript, rispetto al tradizionale var.

```
for (let i = 0; i < 10; i++) {
 let t = i;
 console.log('inside i = ', i);
 console.log('inside t = ', t);
}

console.log('outside i = ', i); // i not defined
console.log('outside t = ', t); // t not defined
```

let: Block-scoped

Access restricted to nearest enclosing block

è "block-scoped", ovvero la sua visibilità è limitata al blocco di codice in cui è definita.

```
for (var i = 0; i < 10; i++) {
 var t = i;
 console.log('inside i = ', i);
 console.log('inside t = ', t);
}

console.log('outside i = ', i); // output?
console.log('outside t = ', t); // output?
```

var: Function-scoped

Access restricted to nearest enclosing function
Common in older Javascript code

Const

```
let x = 88;
const y = 77;
x = 9;
console.log('x = ', x);
y = 17;  // TypeError: Assignment to constant variable.
console.log('y = ', y);
const y = 55; // SyntaxError: Identifier 'y' has already been declared
```

const: Block-scoped, like **let**

Values of const variables cannot be reassignment
Const variables cannot be redeclared

Let bug in IE11

```
for (let i = 0; i < 3; ++i) {
   setTimeout(function() {
       console.log(i);
    }, i * 100);
}

// output on chrome 0,1,2
// output on IE11 3,3,3



// let variables not bound separately to each iteration of for loops
```

Support table
https://caniuse.com/#feat=let

Arrow Functions

Arrow Functions: Syntax

Le arrow functions sono una sintassi abbreviata per scrivere funzioni

- A function shorthand
- Use the => syntax
- Share the same lexical **this** as their surrounding code particolarmente utili per le funzioni callback.

Syntax

```
(x, y, z) => { statements }
(x, y, z) => expression // same as: (x, y, z) => { return expression; }
```

Optional parentheses

```
(x) => { statements }
x => { statements }
```

No parameters syntax

```
() => { statements }
```

Arrow Functions: Variants

FUNZIONE NORMALE!!!

```
function square(a) {
 return a * a;
}

let square = (a) => {
 return a * a;
};

// equivalent
let square = (a) => a * a;

// equivalent
let square = a => a * a;
```

Arrow functions are functions

```
let add = (x, y) => { return x + y; };

console.log(typeof add); // function

console.log(add instanceof Function); // true
```

Useful for callbacks

```
const f = () => {
 console.log('no return value');
};

setTimeout(() => {
 console.log('before calling f');
 f();
 console.log('after calling f');
}, 500);

setTimeout(f, 1500);
```

Questo esempio mostra come possiamo organizzare il flusso di esecuzione in modo ordinato usando setTimeout.

Ad esempio, possiamo voler eseguire un'operazione subito dopo un'altra, ma con un breve ritardo per dare il tempo ad altre operazioni di completarsi.

Shorter code

```
const result = [ 1, 2, 3, 4 ]
 .filter(n => n % 2 !== 0)
 .map(n => n * 2);

console.log('result = ', result);
```

```
let result = [ 1, 2, 3, 4 ]
 .filter(function(number) {
   return number % 2 !== 0;
 })
 .map(function(number) {
   return number * 2;
 });

console.log('result = ', result);
```

Returning object literals

```
const setColor = (color) => {value: color;};
const color = setColor('green').value;
console.log(color);


// err: Cannot read property 'value' of undefined
```

```
const  setColor = (color) => ({ value: color });
const color = setColor('green').value;
console.log(color);


// all OK: output is 'green'
```

This operator in arrow functions

Il problema è legato al comportamento di this nelle funzioni JavaScript.

Le arrow functions sono una soluzione moderna ed elegante per mantenere il contesto di this, mentre l'uso di una variabile come self è un metodo tradizionale.

```
// in methods, context is sometimes lost, e.g. when using setTimeout

let person = {
 name: 'james',
 talk: function() {
   console.log('I am', this.name);
 },
 talkLater: function() {
   setTimeout(function() { funzione passata a setTimeout, this non si riferisce più all'oggetto person
     console.log('I am still', this.name);
   }, 1000);
 },
};


person.talk(); // I am james
person.talkLater(); // I am still
```

This operator in arrow functions

```
// old style solution

let person = {
 name: 'james',
 talk: function() {
   console.log('I am', this.name);
 },
 talkLaterFix: function() {
   let self = this;
   setTimeout(function() {
     console.log('I am still', self.name);
   }, 1000);
 },
};

person.talk(); // I am james
person.talkLaterFix(); // I am still james
```

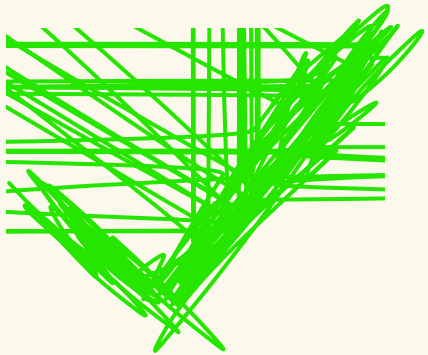
This operator in arrow functions

SOLUZIONE AL problema riguardante al contesto di this, l'uso di un arrow function dentro il setTimeout si riferisce al contesto dove è stato creato

```
// arrow functions solution because they use the same lexical this as surrounding code

let person = {
 name: 'james',
 talk: function() {
   console.log('I am', this.name);
 },
 talkArrow: function() {
   setTimeout(() => {
     console.log('I am still', this.name);
   }, 3000);
 }
};


person.talk(); // I am james
person.talkArrow(); // I am still james
```



Primitive types

New primitive types

BigInt and Symbol were added to JavaScript as new primitive types in ECMAScript 2015 (ES6)
BigInts are used to represent integers of arbitrary precision, while Symbols are unique and immutable data types primarily used as property keys in objects to prevent name collisions

Template Strings

Template strings

Le template strings permettono di scrivere stringhe che possono includere variabili ed espressioni all'interno di backticks (`)

```
const title = `Template strings are syntactic sugar`;

const message = `Can be
on multiple
lines`;

console.log(`Used almost anywhere strings are used, more or less`);
```

Template strings

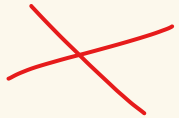
```
const name = 'james';
const age = 25;

// interpolate variable bindings
console.log(`My name is ${name} I am ${age + 10}
years old (lie)`);
```

molto + corto e funzionale!

```
let name = 'james';
let age = 25;

// without using template strings
console.log('My name is '.concat(name, ' I am
').concat(age + 10, ' years old (lie)'));
```



Template strings

```
// may include complex expressions but this reduces code readability
const randInt = n => Math.floor(Math.random() * n);
const inventName = () => [ 'paul', 'adam', 'han' ][randInt(3)];

// template string with complex expressions
console.log(`My name is ${inventName()} I am ${randInt(60) + 21} years old (completely lie)`);
```

Destructuring

Il destructuring consente di estrarre facilmente valori da oggetti o array.

Destructuring objects

```
const person = {
 firstName: 'james',
 lastName: 'smith',
 teacher: true,
 age: 33
};

// quickly get property values from an object
const { firstName, age } = person;
console.log(`My name is ${firstName} I am ${age + 10} years old (lie)`);

const { firstName: name, lastName, age: years } = person;
console.log(`Name is ${name}, Lastname is ${lastName}, age is ${years}`);
```

prendere i valori dall'oggetto

Destructuring nested objects

```
const person = {
 firstName: 'adam',
 lastName: 'jensen',
 work: {
   title: 'chief of security',
   experience: 10
 },
 age: 43
};

// also for nested objects; but increases code complexity
const { firstName: name, age, work: { title: job, experience } } = person;
console.log(`I am ${name}, ${age} years old, I have been a ${job} for ${experience} years`);

const printPerson = ({ firstName: name, age, work: { title: job, experience } }) =>
 console.log(`name:${name}, age:${age}, job:${job}, experience: ${experience}`);

printPerson(person);
```

Destructuring with default values

```
const person = {
 firstName: 'sam',
 job: 'teacher'
};

const person2 = {
 firstName: 'mike'
};

// can assign default values while destructing
const printPerson = ({ firstName: name, job = 'unknown' }) => console.log(`name:${name}, job:${job}`);

printPerson(person);
printPerson(person2);
```

Destructuring arrays

```
const cast = [ 'Gomez', 'Morticia', 'Pugsley', 'Wednesday', 'Uncle Fester' ];
const [ , second, , , fifth, sixth = 'missing' ] = cast;

console.log('second =', second);
console.log('fifth =', fifth);
console.log('sixth =', sixth);

const printCast = ([ , a, b, , c ]) => console.log(a, b, c);
printCast(cast);
```

Default and rest parameters

Default function parameters

```
function composeName(first = 'John', last = 'Smith') {
 return `Mr ${first} ${last}`;
}

const compose = (first = 'John', last = 'Smith') => `Mr ${first} ${last}`;
console.log(compose('Mike'));
console.log(compose('Jack', 'Harkness'));
console.log(compose('Hugh', ''));
console.log(compose('Peter', null));
console.log(compose('Sam', undefined));
console.log(compose(undefined, 'Song'));
// Mr Mike Smith
// Mr Jack Harkness
// Mr Hugh
// Mr Peter null
// Mr Sam Smith
// Mr John Song
```

Rest parameter

```
function printActors(star, guest, ...rest) {
 // rest is an array
 console.log(`Film cast
   Staring: ${star}
   Guest star: ${guest}
   ----
   less important actors: ${rest.sort()}`);
}

printActors('Gomez', 'Morticia', 'Pugsley', 'Wednesday', 'Uncle Fester');

// Film cast
// Staring: Gomez
// Guest star: Morticia
// ----
// less important actors: Pugsley,Uncle Fester,Wednesday
```

Rest parameter

```
function getActorsList(...args) {
 // args is an array
 return args.join(':');
}

const getActors = (...args) => args.map(x => ('' + x).toLowerCase()).join('-');

const res = getActors('Gomez', 'Morticia', 'Pugsley', 99, 'Wednesday', 'Uncle Fester');
console.log(res);
```

// the rest parameter can be freely named, but should be meaningful e.g. ...rest or ...args

Spread operator

Spread operator

- Useful for
  - merging arrays or objects
  - making shallow copies of arrays or objects
  - passing arguments to functions
- Don't confuse the spread operator with the rest operator
  - might look similar
  - very different things!

Spreading arrays

```
const bonds = [ 'Pierce Brosnan', 'Daniel Craig' ];
const oldBonds = [ 'Sean Connery', 'Roger Moore', 'Timothy Dalton' ];

// copy
const clones = [ ...bonds ];

// copy and add
const mixed = [ 'Mark Hamill', ...bonds, 'Harrison Ford' ];
// Mark Hamill,Pierce Brosnan,Daniel Craig,Harrison Ford

// merge
const allBonds = [ ...bonds, ...oldBonds ];
// Pierce Brosnan,Daniel Craig,Sean Connery,Roger Moore,Timothy Dalton

const tooManyBonds = [ ...bonds, ...oldBonds, ...bonds ];
// Pierce Brosnan,Daniel Craig,Sean Connery,Roger Moore,Timothy Dalton,Pierce Brosnan,Daniel Craig
```

Spreading objects

```
// objects
const sam = { name: 'sam', age: 42 };
const clone = { ...sam };

const mike = { ...sam, name: 'mike', hobby: 'fishing' };
console.log(JSON.stringify(mike));
// {"name":"mike","age":42,"hobby":"fishing"}

const monster = { ...mike, ...sam, category: 'chimera' };
console.log(JSON.stringify(monster));
// {"name":"sam","age":42,"hobby":"fishing","category":"chimera"}
```

Spreading strings

```
// strings
const password = 'Abracadabra';
const letters = [...password];
console.log(letters);
// ["A", "b", "r", "a", "c", "a", "d", "a", "b", "r", "a"]

const result = letters.join('||');
console.log(`result:`, result);
// result: A||b||r||a||c||a||d||a||b||r||a
```

Spreading arrays - shallow copy

```
const numbers = [ 1, [ 2, 3 ], 4, 5 ];

const clone = [ ...numbers ]; // shallow copy

console.log(clone.toString()); // 1,2,3,4,5

clone[1][0] = 8;
console.log(clone.toString()); // 1,8,3,4,5
console.log(numbers.toString()); // 1,8,3,4,5
```

```
// the spread operator creates a shallow copy
// need to write custom code if a deep copy is required; ideas?
```

Spreading objects - shallow copy

```
let carl = {
 name: 'carl',
 job: {
   title: 'hunter',
   salary: 1200
 },
 speak: (word = 'nothing') => console.log(`I say ${word}`)
};

let sam = { ...carl, name: 'sam', age: 28 };
sam.speak();
sam.speak('hello');
sam.job.title = 'teacher';
console.log(carl.job.title); // copy is shallow



// the spread operator creates a shallow copy also when using objects
```

Spreading as function arguments

```
const print = (title = 'Staring', actor1, actor2, separator = '&') =>
 console.log(`${title} : ${actor1} ${separator} ${actor2}`);

const actors = [ 'Mark Hamill', 'Harrison Ford' ];
print('In this film', ...actors);

const args = [ 'Staring', ...actors.reverse(), 'and' ];
print(...args);
```

For..of

Iteration - for .. of

```
const countries = [ 'Italy', 'France', 'Germany' ];

for (const country of countries) {
 console.log(country);
}

for (const [ index, value ] of countries.entries()) {
 console.log(`item-${index}: ${value}`);
}

// remember don't use for..of on objects
const sam = { name: 'sam', age: 42 };
for (const property of sam) {
 // TypeError: sam is not iterable
 console.log(property);
}
```

Operators

Logical OR ||

The OR || operator using non-boolean values

1. Evaluates operands from left to right
2. For each operand, if it is truthy, stops and returns the original value of the operand
3. If all operands are falsy, returns the last operand

```
let name = '';
let userName = name || 'default'; // default

let name2 = 'james';
let userName2 = name2 || 'default'; // james
```

Logical OR ||

Logical AND &&

The AND && operator using non-boolean values

1. Evaluates operands from left to right
2. For each operand, if it is falsy, stops and returns the original value of that operand
3. If all operands are truthy, returns the last operand

```
let userName = person && person.name; // undefined

let person = {};
userName = person && person.name; // undefined

person = { name: 'james' };
userName = person && person.name; // james
```

Logical AND &&

Optional chaining ?.

Optional chaining ?. operator

1. The ?. operator is like . chaining but short-circuits with undefined
2. It returns undefined if a reference is nullish (null or undefined)
3. With functions, it returns undefined if the function doesn't exist

```
let userName = person?.name; // undefined

let person = {};
userName = person?.name; // undefined

person = { name: 'james' };
userName = person?.name; // james
```

Optional chaining ?. can also be used with functions and expressions

Nullish coalescing ??

```
let name = ''; // falsy but not nullish
console.log(name || 'default'); // 'default'
console.log(name ?? 'default'); // ''

let age = 0; // falsy but not nullish
console.log(age || 18); // 18
console.log(age ?? 18); // 0
let distance = NaN; // falsy but not nullish
console.log(distance || 0); // 0
console.log(distance ?? 0); // NaN

let weight = null; // null is falsy and nullish
console.log(weight || 70); // 70
console.log(weight ?? 70); // 70
let blank; // undefined is falsy and nullish
console.log(blank ?? 70);
```

Nullish coalescing ??

Promise

Le promesse (Promises) in JavaScript sono un meccanismo che permette di gestire operazioni asincrone, ovvero operazioni che non avvengono immediatamente ma richiedono del tempo per essere completate, come le richieste a un server o la lettura di file.

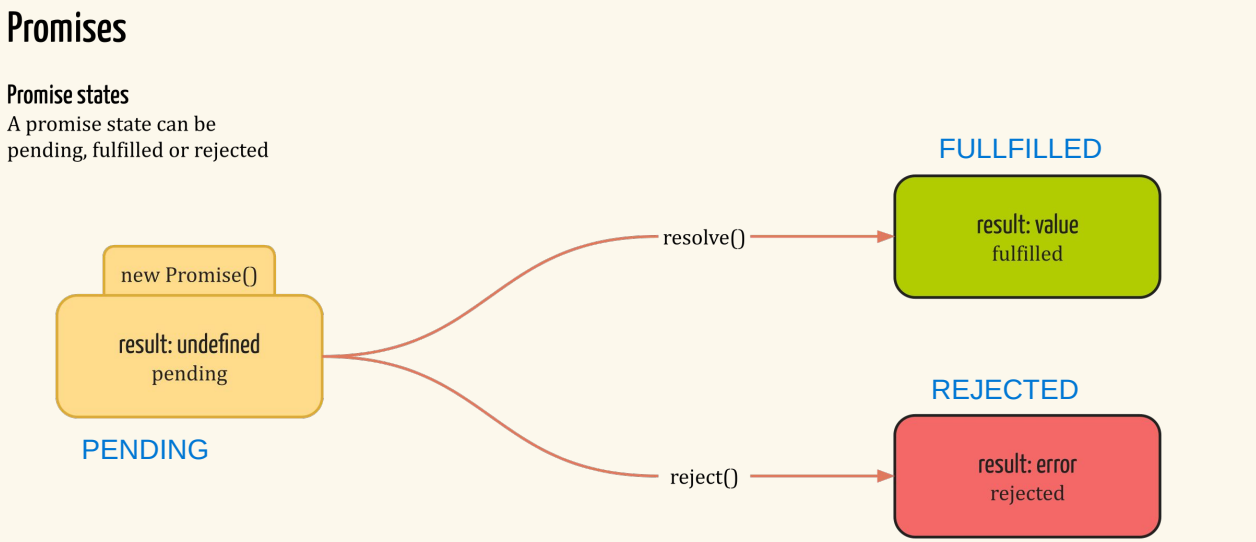
Una Promise rappresenta il risultato di un'operazione asincrona che potrebbe non essere ancora disponibile quando viene restituito l'oggetto. In pratica, è un "promemoria" che il risultato sarà disponibile in futuro, o che l'operazione potrebbe fallire.

What's a promise

- An object that may produce a single value in the future
  - a resolved value
  - or a reason that it’s not resolved: an error
- Can have 3 possible states
  - fulfilled
  - rejected
  - pending
- Promise users attach callbacks to handle the fulfilled value or the rejection

How a promise works

non bloccherà il mio codice anche se non otterà la risposta subito!



Coin toss promise

una promise riceve come primo parametro una funzione (di tipo function)

che avrà 2 parametri (resolve reject) i 2 valori sono di tipo function

```
console.log(`main starts`);

new Promise((resolve, reject) => {
  setTimeout(() => {
    Math.random() > 0.5 ? resolve('won toss') : reject(new Error('failed toss'));
  }, 500);
})
  // handles the result
  .then(result => console.log(result))
  // handles the error
  .catch(err => console.error(err.message))
  // runs when promise resolves or rejects
  .finally(() => console.log('end of game'));

console.log(`main continues`);
```

.finally() -> VERRA ESEGUITO SEMPRE! viene usato per fare pulizie e chiudere il collegamento

Using promises

- Without handlers a promise will not have much effect
- handlers are registered using the methods
  - .then
  - .catch
  - .finally
- Promises are useful and used for many asynchronous actions
  - waiting for the result of a request to a server
  - waiting for a connection a service
  - waiting for some module to initialize

Instant promises

```
// promises can resolve or reject after a certain time, or instantly
// the mechanism works the same way

// instantly resolved
new Promise((resolve, reject) => resolve())
  .then(result => console.log('promise resolved'))
  .catch(err => console.error('promise rejected'));

// instantly rejected
new Promise((resolve, reject) => reject())
  .then(result => console.log('promise resolved'))
  .catch(err => console.error('promise rejected'));
```

Chaining handlers

```
new Promise((resolve, reject) => resolve('I did it'))
  .then(result => {
    console.log(`We got a result: ${result}`);
    return result;
  })
  .then(result => console.log(`Still got a result: ${result}`))
  .then(result => console.log(`Resolved but lost result: ${result}`));

// note: if you don't return a result the next handler gets undefined
```

Success and fail callbacks

```
// can use individual success and fail callback functions

const onSuccess = result => console.log('promise resolved');
const onFail = err => console.error('promise rejected');

new Promise((resolve, reject) => reject()).then(onSuccess, onFail);
// note: onFail handles a promise reject but doesn't handle errors thrown by onSuccess
```

Promise API

```
// Promise has useful methods such as .all
// waits for multiple promises to resolve
// takes an array of promises and returns an array of results

Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 400)),
  new Promise(resolve => setTimeout(() => resolve(2), 200)),
  new Promise(resolve => setTimeout(() => resolve(3), 100))
]).then(results => console.log(results)); // output [ 1, 2, 3 ]
```

Promise.all()

```
// remember to handle any errors
Promise.all([
  new Promise(resolve => setTimeout(() => resolve(1), 200)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error(2)), 300)),
  new Promise(resolve => setTimeout(() => resolve(3), 100))
])
  .then(results => console.log(results))
  .catch(err => console.log(`error: ${err.message}`));
```

Promise.all()

```
// a more useful example
// assume this function actually loads urls asynchronously
const load = url => new Promise(resolve => resolve(`slow code gets ${url}`));

const requests = [
  'https://www.bbc.com/',
  'https://www.cnn.com/',
  'https://www.amazon.it/'
].map(url => load(url));

// the handler runs when all urls have been loaded successfully
Promise.all(requests).then(responses => console.log(responses));
```

Promise API

Useful Promise API methods

- Promise.all()
- Promise.allSettled()
- Promise.race()
- Promise.any()

Async and await

L'uso di async e await in JavaScript permette di lavorare con il codice asincrono in modo più leggibile e simile al codice sincrono.

Async/await

```
async function main() {
  const promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve('done!'), 400);
  });

  console.log(`before await`);
  let result = await promise; // wait until promise resolves
  console.log(`after await. Result is: ${result}`);
}

main();
console.log(`main is async so this code will not wait for the promise to resolve`);



// async/await is elegant, makes promises simpler and code easier to read
// could have written the same code using .then()
```

Await needs an async

```
// await can't be used in a function that is not async
function doSomething() {
  let promise = new Promise(resolve => setTimeout(resolve, 300));
  let result = await promise; // Syntax error function not async
}

// await can't be used in top-level code.
let promise = new Promise(resolve => setTimeout(resolve, 300));
let result = await promise; // Syntax error top-level code
```

note: after ES2022 await can be used at the top level of a module
[ECMAScript 2022 Features]

Await needs an async

```
// can use an IIFE as a solution
(async () => {
  let promise = new Promise(resolve => setTimeout(resolve, 3000));
  console.log(`early`);
  let result = await promise; // this works
  console.log(`later`);
})();
```

References:

IIFE

Async/await error handling

```
async function getUser(id) {
  // assume this function actually handles DB communication
  const getFromDB = id =>
    new Promise((resolve, reject) => reject(new Error(`User ${id} not found`)));

  try {
    let userData = await getFromDB(id);
    // do something with userData then return it
    return userData;
  } catch (err) {
    console.error(`Error: ${err.message}`);
  }
}

getUser(42);
```

Async/await error handling

- There are many styles of handling errors with promises
  - See Async/Await without Try/Catch Block in JavaScript for examples
- Best to go with the standard try/catch implementation because most programmers are more familiar with it

Extras

Additional modern features

Important modern Javascript features

- The Fetch API
- Map
- Set

Your turn

1.Delay

- Use promises to implement a delay function that can be used like in the code below
- Your implementation should work for any type of Javascript function such as
  - regular functions
  - arrow functions
  - anonymous functions

```
delay(300).then(myFunction);
```

2.Roulette

- Write a function called round that returns a promise with a 50/50 probability of resolving or rejecting
- The function should take 2 optional parameters:
  - label, a label for the round, otherwise the default is "round"
  - delay, a delay in which to resolve the promise, otherwise 500ms
- Call the function 3 times and use the Promise API to create an output as in the following page
- Remember to handle any possible errors cleanly

2.Roulette

When any round is lost (and terminate)

round x: lost!
Game over

When all rounds are won (and terminate)

round 1:won!
round 2:won!
round 3:won!
Game over

Bonus

3.Greatest hits

- Rewrite some previous exercises in modern JS syntax
  - Credit Card Validation
  - Advanced Arrivals
  - Reduce All
- Try to use as many modern features as you can
- In readme.md document any important changes
- **Bonus:**
  - Use webpack, make your code compatible with older browsers

References

Let

Const

Arrow function expressions

Template strings

References

Destructuring assignment

Default parameters

Rest parameters

Spread syntax (...)

For...of - JavaScript

References

Promise
Async and await
javascript.info Promise Guide