Modern JavaScript

ES6+

Shadi Lahham - Programmazione web - Frontend - Javascript

History

History of Javascript

- 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

```
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
```

```
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?
```

let: Block-scoped

Access restricted to nearest enclosing block

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

- A function shorthand
- Use the => syntax
- Share the same lexical **this** as their surrounding code

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

```
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);
```

Shorter code

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

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

```javascript
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

```
// 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() {
     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

```
// 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
```

Template Strings

Template strings

```
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)`);
```

```
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

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}`);
```

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

```javascript
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);
```

```javascript
// 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);
}
```

Promise

What's a promise

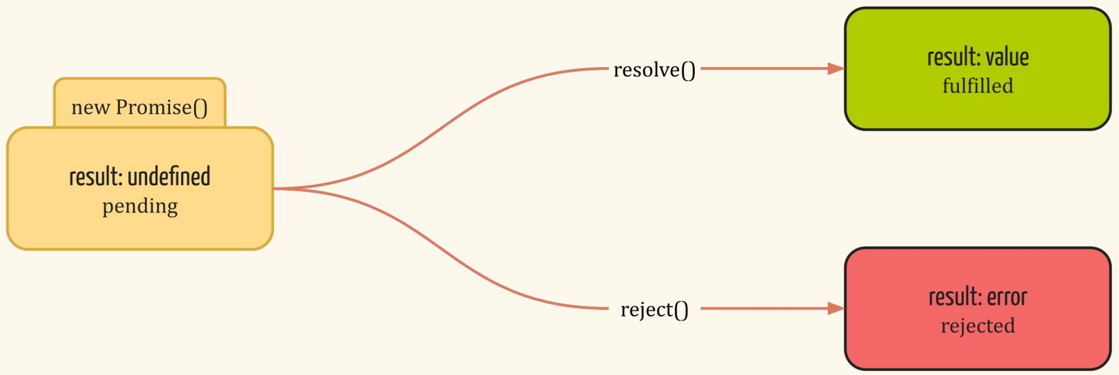
- An object that may produce a single value in the future
  - a resolved value
  - or a reason that it’s not resolved: an error
- Can have 3 possible states
  - fulfilled
  - rejected
  - pending
- Promise users attach callbacks to handle the fulfilled value or the rejection

How a promise works

Promises

Promise states

A promise state can be
pending, fulfilled or rejected



Coin toss promise

```javascript
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`);
```

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

Async/await

```javascript
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
```

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 arguments:
    - 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 and polyfills, 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