R4.01-R4.A.10

Compléments javascript

monnerat@u-pec.fr

23 janvier 2026

IUT de Fontainebleau

Sommaire

Variables et types

Déclaration

`var`

```
var a = 0, b = 1
var c //typeof c === 'undefined'
```

- En dehors de toute fonction, la variable est assignée à l'objet global, et visible partout.
- Dans une fonction, elle est assignée à la fonction, visible (uniquement) dans toute la fonction.

`let` : limite la portée (scope) d'une variable au bloc où elle est déclarée.

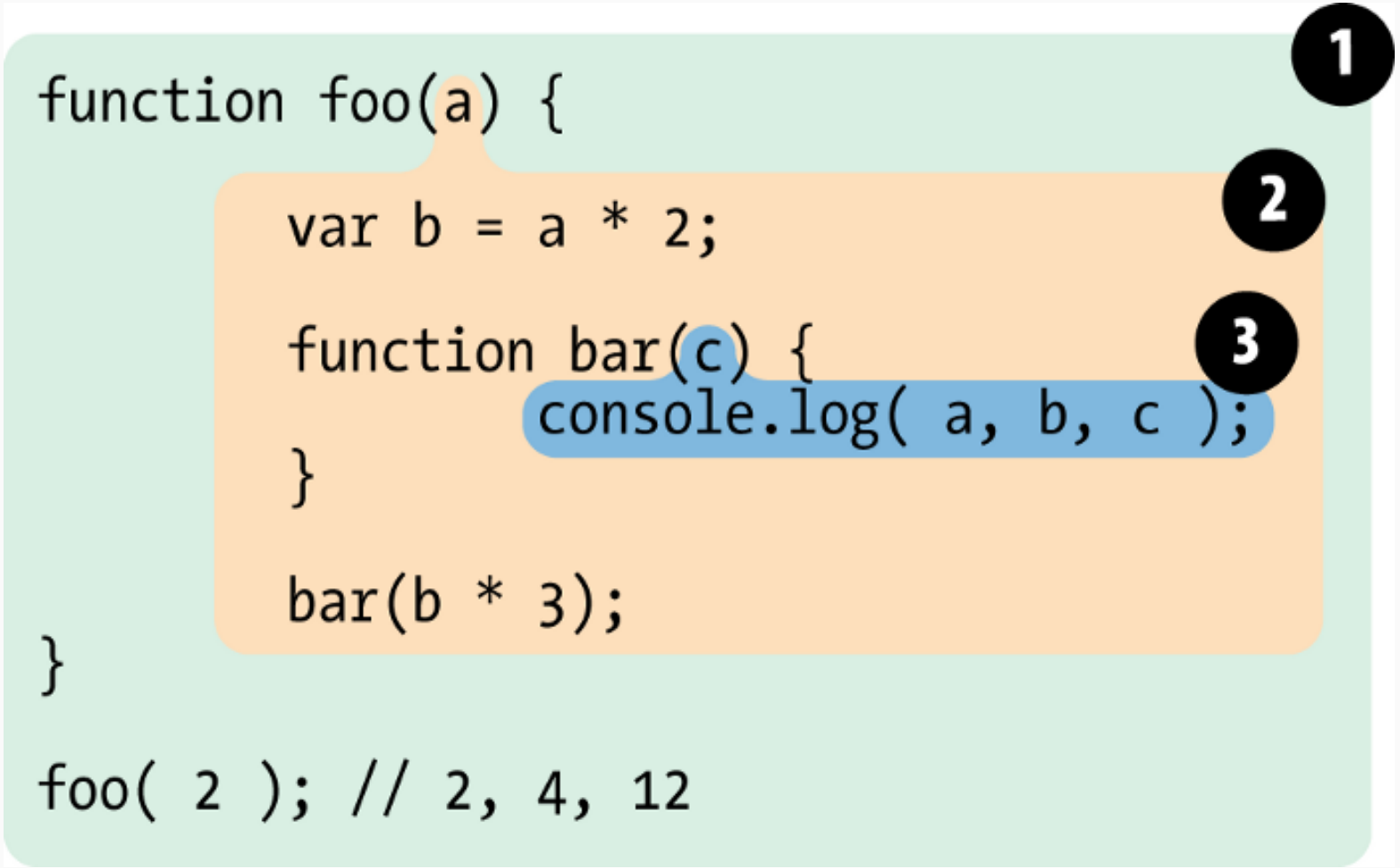
`const` : idem `let`, mais on ne peut plus modifier la valeur de la variable.

```
const a = 'toto'
```

Si la variable `a` est une référence sur un objet, `const` ne rend pas l'objet immutable !

Scope/portée

Portée lexicale (statique)



Pas de portée dynamique en javascript

```
function foo() {
  console.log( a ); // affiche 2
}

function bar() {
  let a = 3;
  foo();
}

let a = 2;
bar();
```

Typage

Le type est déterminé lors de l'inialisation ou d'une affectation. Il peut changer lors d'une affectation.

Types primitifs

`null`	littéral `null` : représente la "nullité" (aucune valeur pour l'objet n'est présente).
`undefined`	propriété de l'objet global, qui vaut undefined. type "undefined"
`boolean`	booléens : `true,false`
`number`	Entier : `102,0xaef,075`. Réel : `3.1415,-6.23e-12`.
`string`	Chaîne de caractères. `"toto",'toto'`. Les caractères d'échappement du C sont reconnus

Tout le reste est constitué d'objets et de fonctions (objet aussi).

- ⇝ `typeof(x)` retourne, sous forme d'une chaîne, le type de `x`. typeof sur une variable non définie renvoie la chaîne "undefined".
- ⇝ A noter la présence de l'opérateur `a===b` qui renvoit true si `a` et `b` sont de même type et de même valeur (et `!==`).
- ⇝ Sur des objets, `a===b` teste directement l'adresse (`a` et `b` doivent correspondre à la même adresse).

Remarques :

- l'opérateur `typeof` ne renvoie pas d'erreur si l'objet n'est pas défini.
- toute variable définie, non initialisé a pour type "undefined".

```
>let x
>x === undefined
true
>typeof x
"undefined"
>typeof y
"undefined"
```

Conversion de type

Le type `String` (immuable) est dominant.

- ⇝ Conversion implicite avec les opérateurs d'égalités faibles (==). JS essaie dans beaucoup de situtations de transtyper vers le type Number (cf algo sur MDN).
- ⇝ Conversion explicite avec `Boolean()`, `Number()`, `String()`
- ⇝ Toutes les valeurs de types primitifs peuvent être transtypés en booléens.
- ⇝ Pas de conversion avec les oprérateurs d'égalités strictes (===).

```
N=12;
T="34";
X=N+T;// X est la chaîne 1234
X=N+Number(T);// X vaut 46
```

Gabarit de chaîne de caractères

- Délimité avec ‘ . . . . ‘, éventuellement sur plusieurs lignes (les sauts de lignes font parties de la chaîne).
- Interpolation d'expressions avec `${...}`.

```
const sanctionMaximale = 5;
let message = `
  <h1>Attention!</h1>
  <p>La pratique non autorisée du hockey peut résulter en
  une sanction maximale de ${sanctionMaximale} minutes.</p>
`;
```

Atention aux xss avec des données "dynamiques" évidemment.

Gabarit étiqueté

Étiquette (fonction) qui "calcule" la valeur finale du gabarit.

```
let personne = 'Michou';
let age = 28;

function monEtiquette(chaines, expPersonne, expAge) {
  let chn0 = chaines[0]; // "ce "
  let chn1 = chaines[1]; // " est un "
  let chnAge;
  if (expAge > 99){
    chnAge = 'centenaire';
  } else {
    chnAge = 'jeunot';
  }
  // On peut tout à fait renvoyer une chaîne construite  avec un gabarit
  return `${chn0}${expPersonne}${chn1}${chnAge}`;
}

let sortie = monEtiquette`ce ${ personne } est un ${ age }`;
console.log(sortie); // ce Michou est un jeunot
```

Booléens et opérateurs logiques

Les (seules) 6 valeurs sont converties en `false`

```
false
undefined
null
NaN
0
"" (empty string)
```

Tout le reste est transtypé en `true`

On peut faire un transtypage vers un booléen avec

```
let a = []
b = !!a
```

`!expr`

Renvoie `false` si expr peut être transtypé en true, sinon `true`.

`expr1 && expr2`

Attention, renvoie `expr1` si elle peut être convertie en `false` et renvoie `expr2` sinon.

`expr1 || expr2`

Attention, renvoie `expr1` si elle peut être convertie en `true` et renvoie `expr2` sinon.

```
const or = '' || 'hi'; // "hi"
const or = [] || 'hi'; // []

const and = '' && 'hi'; // ""
const and = [] && 'hi'; // "hi"
```

null, undefined

```
js> var x=null;
js> typeof(x)
  "object"
js> var y
js> typeof(y)
  "undefined"
js> x==y
  true
js> x===y
  false
js> !x
  true
js> !y
  true
js> z == null
typein:1: ReferenceError: z is not defined
js> z == undefined
typein:2: ReferenceError: z is not defined
js> typeof z
"undefined"
```

```
js> var s=""
js> s==null
  false
js> !s
  true
js> var t={}
js> t==null
  false
js> !t
  false
js> typeof(t)
  "object"
```

Nullish coalescing operator ??

L'opérateur de coalescence des nuls (??), est un opérateur logique qui renvoie son opérande de droite lorsque son opérande de gauche vaut null ou undefined et qui renvoie son opérande de gauche sinon.

```
const foo = null ?? 'default string';
console.log(foo);

const baz = 0 ?? 42;
console.log(baz);
// expected output: 0
```

Value vs référénce

- Les variables de type primitifs sont copiées/passées par valeur.

```
var x = 10;
var a = x;
x=15
console.log(x, a); // -> 15,10
```

- Les variables qui ne sont pas de type primitifs (les objets en gros) sont en fait une référence.

```
var reference = [1];
var refCopy = reference;

reference.push(2);
console.log(reference, refCopy); // -> [1, 2], [1, 2]
```

Réassigner une variable avec un autre objet remplace la référence.

Comparer/assigner par valeur

```
var arr1 = ['Hi!'];
var arr2 = ['Hi!'];
console.log(arr1 === arr2); // -> false
```

Stringifier les objets pour comparer leur contenu :

```
var arr1str = JSON.stringify(arr1);
var arr2str = JSON.stringify(arr2);
console.log(arr1str === arr2str); // true
```

Copier un objet par valeur :

```
b=JSON.parse(JSON.stringify(a))
```

Pas toujours possible. Attention au problème de copie superficiel/profondeur. On a aussi

```
structuredClone()
```

Passage des arguments à une fonction :

- Le passage des arguments de type primitif se fait par valeur.
- Le reste se fait par référence.

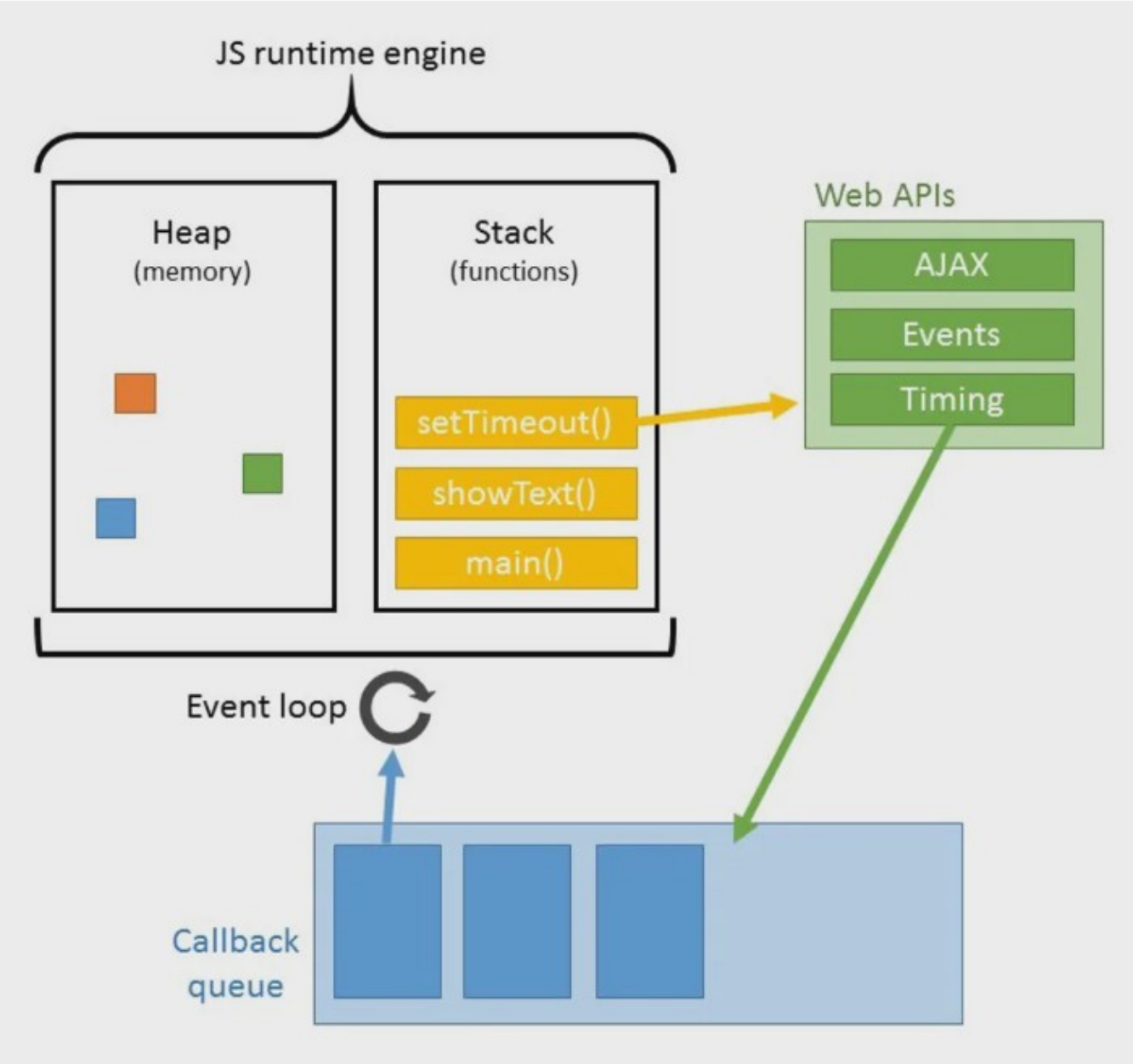
Flow d'exécution en js

Une fenêtre de navigateur utilise un seul thread pour parser l'html, gérer les évenements et exécuter le code javascript.

- le code de "haut niveau" dans les balises `script` est exécuté pendant le chargement de la page.
- les handlers d'évènements à exécuter sont enfilés dans une file de messages (Queue) et consommés (la fonction correspondante est exécutée complétement) au fur et à mesure (Event Loop).

Remarques

- Il existe une api pour exécuter du code dans un thread séparé : Web Workers. Communication par message avec le thread principal.
- La plupart des opérations d'E/S sont asynchrones. Il est possible que le navigateur utilise suivant son implantation un thread dédié ou pas.
- On peut écrire des fonctions qui s'exécutent de manière asynchrone avec les promesses.


JS runtime engine
Heap
(memory)
Stack
(functions)
setTimeout()
showText()
main()
Web APIs
AJAX
Events
Timing
Event loop
Callback
queue

Objets

Les objets

Ils sont traités en interne comme des tableaux associatifs. Pas de vraies classes.

- pas de "vrai" héritage.
- uniquement des créations d'objets et de propriétés prototypes.
- les méthodes statiques existent.
- notation pointée.

Remarques

- Certains objets sont justes des agrégateurs de propriétés, d'autres peuvent être exécutés (fonctions).
- La méthode `Object.assign()` permet de copier les propriétés "directes" d'un objet dans un autre.

Déclaration d'**un** objet avec la syntaxe JSON

```
var obj=
{
  x:2,
  y:3,
  somme(){
    return this.x+this.y;
  }
};
alert(obj.x);
alert(obj['x']);
alert(obj.somme());
```

Déclaration d'une "classe" par la définition de son constructeur.

Opérateur de chaînage optionnel

L'opérateur `?.` (en cours de support) permet de lire une propriété d'un objet. Si la propriété n'existe pas, l'expression ne provoque pas une erreur, mais est évaluée en `undefined`.

```
const adventurer = {
  name: 'Alice',
  cat: {
    name: 'Dinah'
  }
};
const dogName = adventurer.dog?.name; // undefined
```

On peut les emplier

```
let client = {
  nom: "Carl",
  details: {
    age: 82,
    localisation: "Paradise Falls"
    // adresse détaillée inconnue
  }
};
let villeDuClient = client.details?.adresse?.ville;

// Cela fonctionne aussi avec le chaînage optionnel
// sur les appels de fonction
let duree = vacations.trip?.getTime?.();
```

Fonctionne avec les tableaux et les appels de fonctions.

Fonctions

Les fonctions sont des objets !

```
js> var obj = {};
js> var fn = function(){};
js> obj.prop = "some value";
js> fn.prop = "some value";
js> obj.prop == fn.prop
true
```

Pas de problème. Les fonctions ont des propriétés.

Une fonction peut

- être affectée à des variables ou des structures de données.
- être passée comme paramètre.
- être retournée par une fonction.
- être construite lors de l'exécution.

Utilisation d'un cache

```
function isPrime( num ) {
  if ( isPrime.cache[ num ] )
    return isPrime.cache[ num ];
  let prime = num != 1; // Everything but 1 can be prime
  for ( let i = 2; i < num; i++ ) {
    if ( num % i === 0 ) {
      prime = false;
      break;
    }
  }
  isPrime.cache[ num ] = prime
  return prime;
}
isPrime.cache = {};
js> isPrime(5)
true
js> isPrime.cache[5]
true
```

Contexte et `this`

Une fonction s'éxécute dans un "contexte" accessible par le mot clé `this`.

ici l'objet `global`

```
js> this
({})
js> var x=3;this
({x:3})
js> function f(){this.y=4;}
js> f()
js> y
4
js> this
({x:3, f:function f() {this.y = 4;}, y:4})
```

ici l'objet `katana`

```
js> var katana = {
  isSharp: true,
  use (){
   this.isSharp = !this.isSharp;
  }
};
js> katana.use();
js> katana.isSharp
false
```

- `this` dans une fonction est l'objet qui l'appelle.
- En mode strict, (`'use strict'`), en dehors de tout objet, `this` est `undefined`.
- Dans une fonction fléchée, `this` n'est pas défini, mais provient de sa portée lexical (comme une autre variable).

Expliquez

```
var prop = 10
let o = {prop: 37}

function alone() {
  return this.prop
}

o.f = alone

console.log(alone()) // 10
console.log(o.f())   // 37
```

apply, call

On peut le changer avec `apply` ou `call` lors de l'appel

```
js> function S(a){return this.x + a;}
js> x=2
js> S.call(this,2)
4
js> var obj={x:3}
js> S.apply(obj,[2])
5
```

bind

`bind` crée une copie d'une fonction liéé (contexte `this`) à un objet donné.

```
let o = {
  prop: 0,
  getProp(){
    return this.prop
  }
}

let oo = {prop : 1}

o.getProp = o.getProp.bind(oo)
console.log(o.getProp()) // 1
```

Expliquez.

Tableau arguments :

```
function test() {
  alert("Nombre de parametres: " + arguments.length);
  for(var i=0; i<arguments.length; i++) {
    alert("Parametre " + i + ": " + arguments[i]);
  }
}
test("valeur1", "valeur2");
test("valeur1", "valeur2", "valeur3", "valeur4");
```

Obsolète avec ES6 (cf opérateur de décomposition). On préfèrera la forme (compatible avec les fonctions fléchées)

```
function test(...args){
// le tableau args contient tous
// les paramètres d'appel.
}
```

Fonctions

Fonction comme constructeur

Le mot clé `new`

```
function user(prenom,nom){
  this.prenom = prenom;
  this.nom=nom;
  this.changerNom = function (n){
    this.nom=n;
  };
}

js> var Denis = user("denis","monnerat");
js> Denis.prenom
typein:16: TypeError: Denis is undefined
js> var Moi = new user("denis","monnerat");
js> Moi.prenom
Denis
```

L'opérateur new, suivi de la fonction équivaut à :

```
function user(prenom,nom){
  this.prenom = prenom;
  this.nom=nom;
  this.changerNom = function (n){
    this.nom=n;
  };
}

js> var Denis={};
js> user.call(Denis,"Denis","Monnerat");
```

- On peut voir cela comme la définition d'une classe `user`, et d'une instanciation avec `new`.
- Chaque objet garde une trace du "constructeur" avec la propriété (fonction) `constructor`.

Fonctions

Fermetures (closures)

Closures/fermetures

```
function compteur() {
    let count = 0;

    return function() {
        return count++;
    };
}

let plusUn = compteur();
let plusUnBis = compteur();
```

```
console.log(plusUn());
0
console.log(plusUn());
1
console.log(plusUnBis());
0
console.log(plusUn());
2
console.log(plusUnBis());
1
```

Quand une fonction est appelée, elle s'exécute dans le scope défini lors de sa déclaration (scope/portée lexical).

Dans une fonction réflexe :

```
let results = jQuery("#results").html("<li>Loading...</li>");
jQuery.get("test.html", function(html){
  results.html(html);
});
```

Dans un timer :

```
let count = 0;
let timer = setInterval(function(){
  if ( count < 5 ) {
    count++;
  } else {
    clearInterval( timer );
  }
}, 100);
```

Propriété privée avec une fermeture

```
function T(){
  let x = 0;
  this.getX = function(){
    return x;
  };
  this.X = function(){
    x++;
  };
}
js> let t=new T()
js> t.x == undefined
true
js> t.getX()
0
js>t.X()
js>t.getX()
1
```

Modules

```
(function() {

  // declare private variables and/or functions
  return {
    // declare public variables and/or functions
  }
})();
```

Remarque : la notion de modules est maintenant explicite avec `export` et `import`.

```
var counter = (function(){
  let x = 0;
  function _inc(){
    x++;
  }
  function _dec(){
    x--;
  }
  return {
    INC:_inc,
    DEC:_dec,
    GET:function(){
      return x;
    },
    SET:function(a){
      x=a;
    }
  }
})();
```

```
counter.SET(0);
counter.INC();
counter.DEC();
console.log(counter.GET());
```

Chaînage des objets, prototype

Propriété `[[prototype]]`

En ce qui concerne l'héritage, js n'utilise qu'une seule structure : les objets.

- Chaque objet posséde une propriété "privée" qui contient un lien vers un autre objet appelé `prototype`.
- Cet objet a également un prototype, et ainsi de suite, jusqu'à `null`.
- La majorité des objets js sont des "instances" de `Object`, qui est l'avant dernier maillon de la chaîne de prototype.

Quand on accéde à une propriété d'un objet en lecture, celle-ci est cherché dans l'objet lui-même, puis dans son prototype, et ainsi de suite.

Héritage (chaînage) prototypal

[[prototype]] est accessible via la propriété __proto__ (obsolète)

On peut utiliser les accesseurs `Object.getPrototypeOf()` et `Object.setPrototypeOf()`

```
let animal = {
  eats : true,
  walk(){
    return "animal walks"
  }
}

let rabbit ={
  jumps : true
}

let hare = {
  earLength : 10
}
Object.setPrototypeOf(rabbit , animal)
console.log(rabbit.walk()) // animal walk

Object.setPrototypeOf(hare , rabbit)
console.log(hare.walk()) // animal walk
```

Héritage avec fonction constructeur (ou classe)

On a déjà vu qu'il est possible de créer des objets en passant par une fonction constructeur (ou une classe), en utilisant l'opérateur `new`.

```
function Rectangle(l,h){
  this.l = l
  this.h = h
}
let r1 = new Rectangle (10,20)
```

Que vaut `[[prototype]]` de l'objet `r1` ?

L'opérateur `new F(...)` utilise la propriété "publique" (c'est un objet) `F.prototype` de la fonction constructeur pour définir `[[prototype]]` du nouvel objet.

Remarque : la création d'un objet littéral utilise `new Object()`.

Toute fonction a la propriété `prototype`. Par défaut, c'est un objet avec comme seule propriété `constructor` qui renvoie à la fonction elle-même.

On peut donc rajouter de manière dynamique des propriétés à une instance en passant par le prototype de son constructeur.

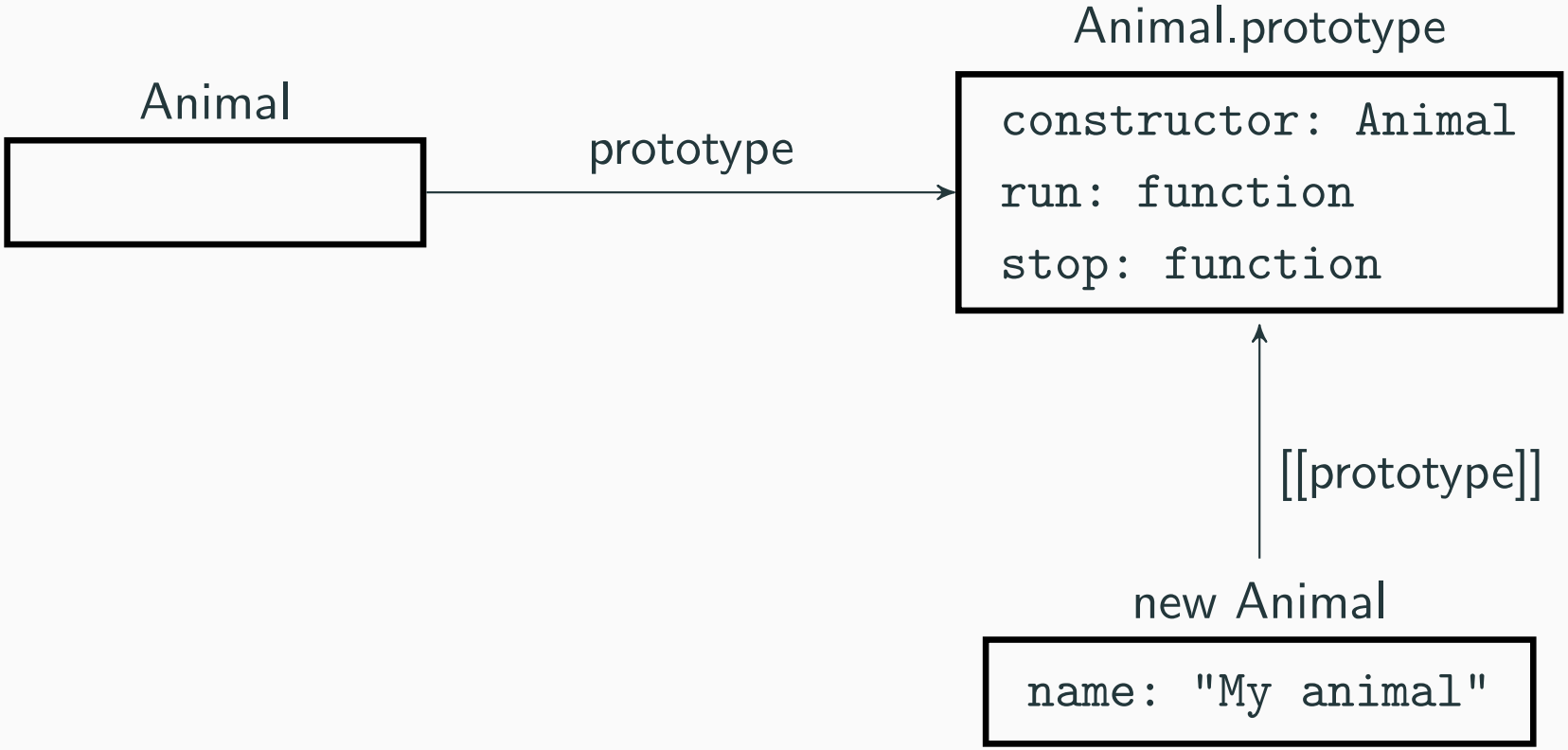
```
r1 = new Rectangle(10,20)
Rectangle.prototype.surface = function (){
  return this.l * this.h
}

console.log(r1.surface())
```

- Expliquez précisemment ce qui se passe au moment de l'appel `r1.surface()` .
- Expliquez la différence si on intégre à priori la méthode surface dans le constructeur.

Classes

Depuis ECMAScript 2015, la notion de classe (sucre syntaxique) a été introduite :

```
class Animal {
  constructor(name) {
    this.speed = 0
    this.name = name
  }
  run(speed) {
    this.speed = speed
    return `${this.name} runs with speed ${this.speed}.`
  }
  stop() {
    this.speed = 0;
    return `${this.name} stands still.`
  }
}
let animal = new Animal("My animal")
```



Remarques :

- les méthodes de la classe `Animal` sont placées dans le prototype.
- exécution en mode strict même sans `"use strict"`.

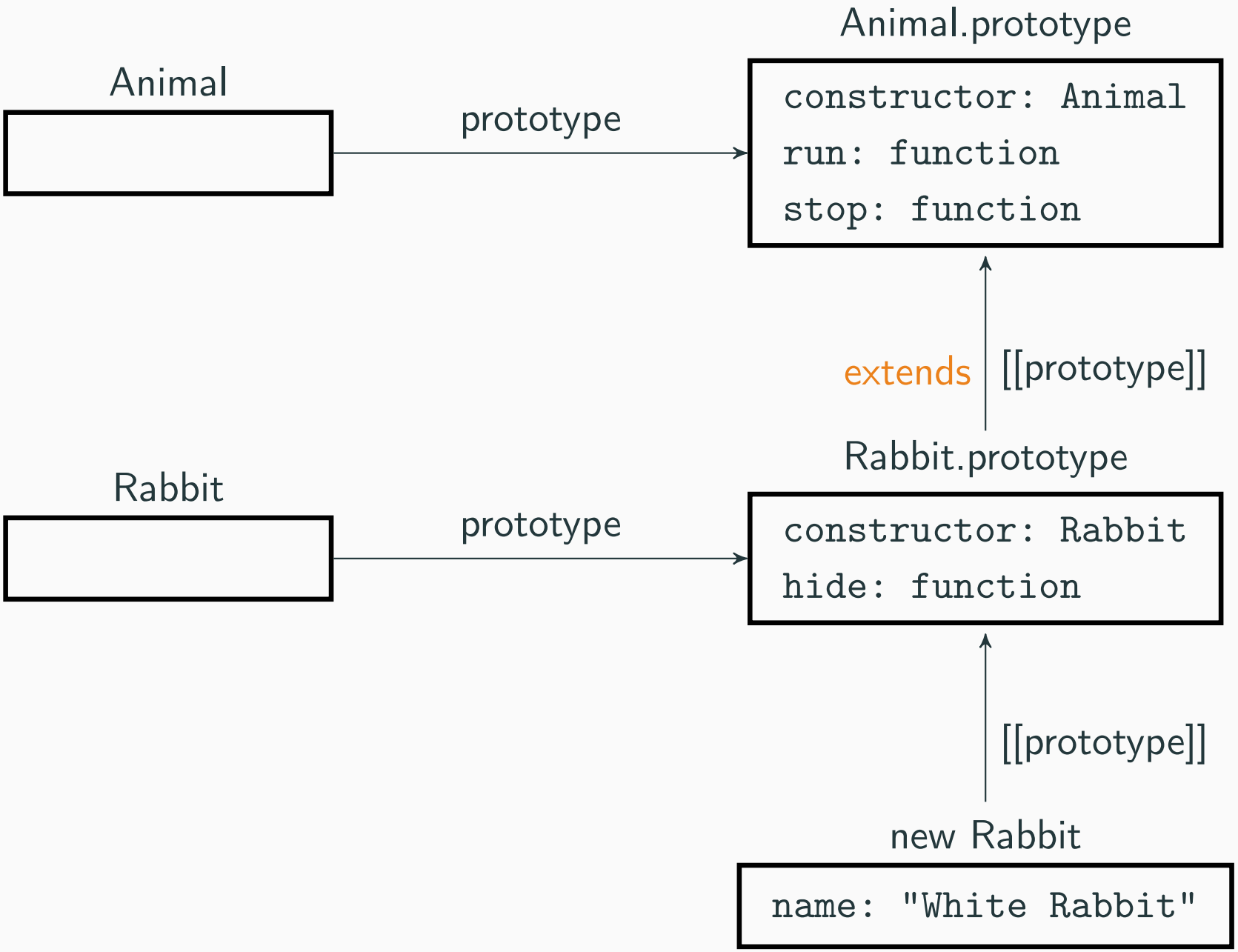
Méthodes statiques

```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }
  static distance(a, b) {
    const dx = a.x - b.x;
    const dy = a.y - b.y;
    return Math.hypot(dx, dy);
  }
}
const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
console.log(Point.distance(p1, p2));
```

Sous-classe

```
class Rabbit extends Animal {
  hide() {
    return `${this.name} hides!`
  }
}

let rabbit = new Rabbit("White Rabbit")

rabbit.run(5) // White Rabbit court à la vitesse 5.
console.log(rabbit.hide()) // White Rabbit se cache!
```

Comment fonctionne en interne le mot clé `extends` ? Il utilise le prototype



Expliquez ce qui se passe lors de l'appel `rabbit.run()`

Décomposition

Opérateur de décomposition

```
// dans un appel de fonction  f(...objetIterable);
function sum(x, y, z) {
  return x + y + z;
}
const numbers = [1, 2, 3];
sum(...numbers);

// Pour les littéraux de tableaux  [...objetIterable, 4, 5, 6]
let arr1 = ['A', 'B', 'C'];
let arr2 = ['X', 'Y', 'Z'];
let result = [...arr1, ...arr2];

// les literraux objets  { ...obj };
let g = { x:{z:3},s:"toto"};
let gg = {h:true,...g};
```

Opérateur de décomposition

Fonction avec un nombre d'arguments variables

```
let sum = function (...args){
return args.reduce(function (a, b) {
  return a + b
}, 0)
}
```

On pourra même écrire avec les fonctions fléchées

```
let sum = (...args) => args.reduce((a, b) => a+b , 0)
```

Affectation par décomposition

Avec les tableaux (itérables)

```
let [a,b] = [1,2]
let [a,...b] = [1,2,3,4]
let [,,b] = [1,2,3]
[a,b] = [b,a]
```

Avec les objets

```
const o = {p: 42, q: true};
const {p, q} = o;

console.log(p); // 42
console.log(q); // true
// Assign new variable names
const {p: toto, q: truc} = o;

console.log(toto); // 42
console.log(truc); // true

let {a, b, ...reste } = {a: 10, b: 20, c: 30, d: 40};
a; // 10
b; // 20
reste; // { c: 30, d: 40 }
```

Dans la déclaration d'une fonction

```
let obj = {x:1,y:2}
function f({x}){
return x + 1
}

f(obj) // ?
```

Fonction fléchées

Expression de fonction fléchée

```
(param1, param2, ... , paramn) => expression
// équivalent à
(param1, param2, ... , paramn) => {
  return expression;
}
// Parenthèses non nécessaires quand
// il n'y a qu'un seul argument
param => expression
// Une fonction sans paramètre peut s'écrire avec un couple
// de parenthèses
() => { instructions }
```

- Syntaxe synthétique.
- Les fonctions fléchées utilisent la valeur `this` de leur portée englobante, pas celle de l'appel.

Expliquez

```
let obj = {
  x:1,
  inc(){
    this.x++
  },
  incBis : (()=> {
    this.x++
  })
}
obj.inc()
obj.incBis()
obj.incBis()
console.log(obj.x) // expliquez ?
```

Expression de fonction fléchée

```
document
   .getElementById("img")
   .onclick=function(){
     this.src="on.png";
     // qui est this ?
   }

document
   .getElementById"img")
   .onclick=(()=>{
     this.setTimeout(
       () => console.log("OK")
     ),1000);
// qui est this ?
```

Itération d'un tableau

Itération

`for, while, do...while` (boucle for classique)

```
const arr = [1, 2, 3];
for (let i = 0; i < arr.length; i++) {
  console.log(arr[i]);
}
```

`for...of`

```
for (const value of arr) {
  console.log(value);
}
```

`Array.prototype.forEach()`

```
arr.forEach((value, index, array) => {
  console.log(value, index);
});
```

Itération

`map()` (transformation)

```
  const doubled = arr.map(value => value * 2);
```

`filter()` (filtrage)

```
  const even = arr.filter(value => value % 2 === 0);
```

`reduce()` (accumulation)

```
  const sum = arr.reduce((acc, value) => acc + value, 0);
```

Itération (recherche)

some(), every()

```
arr.some(v => v > 2);   // true si au moins un élément correspond
arr.every(v => v > 0); // true si tous correspondent
```

find(),findIndex()

```
const numbers = [3, 7, 10, 15];

const result = numbers.find(n => n > 8);
const index = numbers.findIndex(result);
console.log(result); // 10
console.log(index); // 2
```

includes()

```
const fruits = ["pomme", "banane", "orange"];

const hasBanana = fruits.includes("banane");
console.log(hasBanana); // true
```