

Les bases en JavaScript

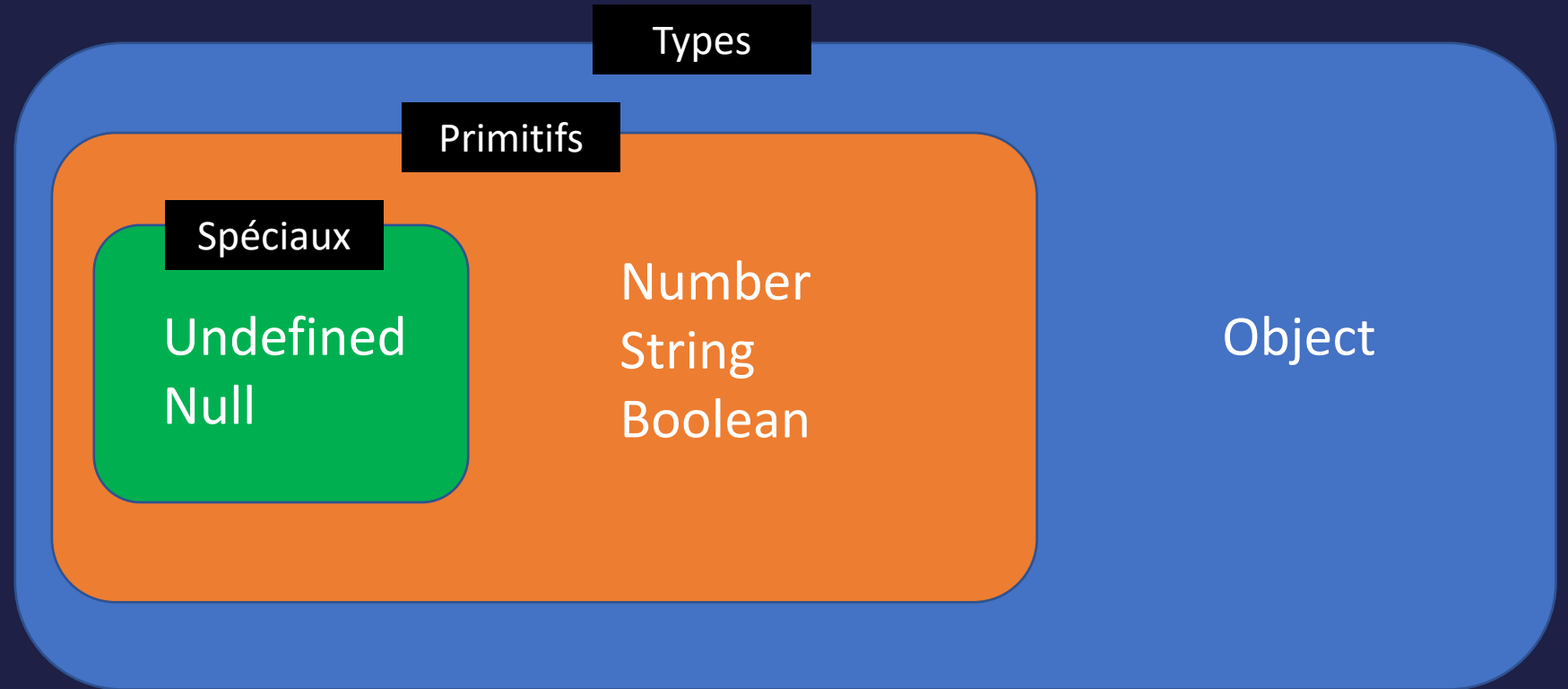
Michael
X NATIS



Compétence visée :
Créer des classes avec leurs
constructeurs et leurs
propriétés

1. Les types
2. La portée statique
3. Le polymorphisme par shadowing
4. Le contexte objet this
5. Les designs pattern en JS
6. L'Instanciation
7. Les classes

Les types



L'opérateur `typeof` renvoie une chaîne qui indique le type de son opérande.

Un objet est un pointeur (*référence*)
alors qu'un contenu d'un type primitif
est une *valeur*

primitive

a	5
---	---

let b = a;

b	5
---	---

indépendant

object

a	0xff3264
---	----------

let b = a;

b	0xff3264
---	----------

age: 14
firstName: 'no'
lastName: 'm'

heap

dépendant

```
const a = 'hello';  
let b = a;  
b = 'ca va';  
console.log(a === b); // prints false
```

```
const obj = {  
  id: 4  
};  
let copie = obj;  
copie.id = 3;  
console.log(obj === copie); // prints true
```


La portée statique

La portée (scope) d'une variable x est la région du programme dans laquelle les utilisations de x font référence à sa déclaration.

L'une des raisons fondamentales du scoping est de garder les variables dans différentes parties du programme distinctes les unes des autres.

La portée (ou scope) concerne la visibilité des variables. Elle détermine la portion de code à partir du moment où la variable naît jusqu'au moment où elle meurt.

Puisqu'il n'y a qu'un petit nombre de noms de variables courts et que les programmeurs partagent des habitudes concernant la dénomination des variables (par exemple, *i* pour un index de tableau), dans tout programme de taille modérée, **les conflits seront évités !**

Avec une **portée dynamique (dynamic scoping)**, un identifiant global fait référence à l'identifiant associé à l'environnement le plus récent, et est rare dans les langues modernes.

En termes plus simples, dans la portée dynamique, l'interpréteur recherche d'abord **le bloc courant** puis successivement **toutes les fonctions appelantes**.

```
function hello() {  
  var x = 1;  
}
```

```
function cava() {  
  console.log(x);  
}
```

```
hello();  
cava();
```



```
function hello() {  
  var x = 1;  
}  
  
function cava() {  
  console.log(x);  
}  
  
hello();  
cava();
```

La portée statique est également appelée portée lexicale. Dans cette portée, une variable fait toujours référence à son environnement de niveau supérieur.

Il s'agit d'une **propriété du texte du programme** et sans rapport avec la pile d'appels d'exécution. La portée statique facilite également la création d'un code modulaire, car un programmeur peut déterminer la portée simplement en regardant le code.

La portée dynamique oblige le
programmeur à anticiper tous les
contextes dynamiques possibles !

Les variables ont 3 portées statiques :

- Global scope (sans var, ni let, ni const)
- Function scope (pour var)
- Block scope (pour let et const)

eternelle

not à la fin de fonction

not à la fin du block *durée de vie*

Nous pouvons ne pas travailler dans le contexte global en mettant notre code dans une fonction et l'exécuter directement : c'est une IIFE (*Immediately Invoked Function Expression*)

encapsuler notre code

V8 Engine

Pour retrouver une variable, l'interpréteur de JS va regarder dans la portée statique courante, et s'il ne la trouve pas, il regardera dans la portée statique externe ... puis ainsi de suite jusqu'au Global scope.

C'est ce qu'on appelle le Scope chain !

Une closure (fermeture) est la paire formée d'une fonction et des références à son état environnant (l'environnement lexical).

En d'autres termes, une closure donne accès à la portée d'une fonction externe à partir d'une fonction interne (on dit aussi que la fonction « capture son environnement »)

En JavaScript, une closure (fermeture) est créée chaque fois qu'une fonction est créée.

Le polymorphisme par shadowing

```
function factory()  
{  
  let x = 'hello';  
  return function sayHello() {  
    console.log(x);  
  }  
}
```

```
let x = 'bye';  
const fun = factory();  
fun();
```

closure
fermeture



```
function factory()  
{  
    return function sayHello() {  
        console.log(x);  
    }  
}
```

```
let x = 'bye';  
const fun = factory();  
fun();
```



Lorsqu'une variable est déclarée dans une certaine portée **ayant le même nom défini sur sa portée externe** et lorsque nous appelons la variable depuis la portée interne, la valeur attribuée à la variable dans la portée interne est la valeur qui sera stockée dans la variable dans l'espace mémoire.

Ceci est connu sous le nom de **Shadowing** ou **Variable Shadowing**.

```
let number = 10;

{
  function double()
  {
    return number * 2;
  }
}

{
  function double()
  {
    return number * 3;
  }
}

console.log(double());
```



```
let number = 10;  
function double()  
{  
    let number = 20;  
    number = number * 2;  
}  
double();  
console.log(number);
```



Le contexte objet 'this'

La portée (ou scope) concerne la visibilité des variables **alors que le contexte** fait référence à **l'objet auquel** appartient une variable ou une fonction.

GEC / Global Execution Context est également appelé exécution de base / par défaut.

Tout code JavaScript qui ne réside dans aucune fonction sera présent dans le contexte d'exécution global.

La raison derrière son nom « contexte d'exécution par défaut » où le code commence son exécution lorsque le fichier se charge pour **la première fois**.

Le GEC découle du flow d'exécution suivant :

1. l'interpréteur crée un objet global pour Node.js et un objet Window pour les navigateurs
2. Il référence l'objet ci-dessus au mot-clé 'this'
3. Il crée une memory heap afin de stocker des variables et des références de fonction

La **moyen standard** d'obtenir l'objet global est
globalThis !

On peut créer un nouveau contexte (et donc un objet) en utilisant l'opérateur `new`.

Cet objet `remplacera le contexte courant` (`this`).

Le contexte courant est en effet dénoté par le mot clé `'this'`.

Il référence majoritairement `l'objet courant` sur lequel les opérations s'appliquent.

La méthode `.apply()` appelle une fonction en lui passant une valeur `this` et des arguments sous forme d'un tableau (ou d'un objet semblable à un tableau).

```
function Person(name) {  
  this.name = name;  
}  
  
const obj = {  
  age: 14  
};  
Person.apply(obj, ['Sophie']);  
console.log(obj);  
console.log(obj instanceof Person);
```



La méthode `.call()` réalise un appel à une fonction avec une valeur `this` donnée et des arguments fournis individuellement.

```
function Person(name) {  
  this.name = name;  
}
```

```
const obj = {  
  age: 14  
};
```

```
Person.call(obj, 'Sophie');  
console.log(obj);  
console.log(obj instanceof Person);
```



Les objets sont constitués de **membres** (propriétés ou **méthodes**) qui permettent de les décrire.

Construire un objet, c'est le faire **physiquement exister en mémoire** pour contenir des données.

Les valeurs des membres d'un objet peuvent être construites à partir de types de données primitifs ou à partir d'autres objets.

Comme des poupées russes ! 😊

```
const obj = {  
  id: 3,  
  firstname: 'Annie',  
  lastname: 'Versaire',  
  email: 'annie.versaire@gmail.com'  
};
```

```
const obj = {  
  id: 3,  
  firstname: 'Annie',  
  lastname: 'Versaire',  
  email: 'annie.versaire@gmail.com',  
  children: [  
    {  
      id: 4,  
      age: 12,  
      firstname: 'Jean',  
      lastname: 'Aimarre'  
    },  
    {  
      id: 6,  
      age: 24,  
      firstname: 'Guy',  
      lastname: 'Tarre'  
    },  
  ],  
};
```

Les designs pattern en JS

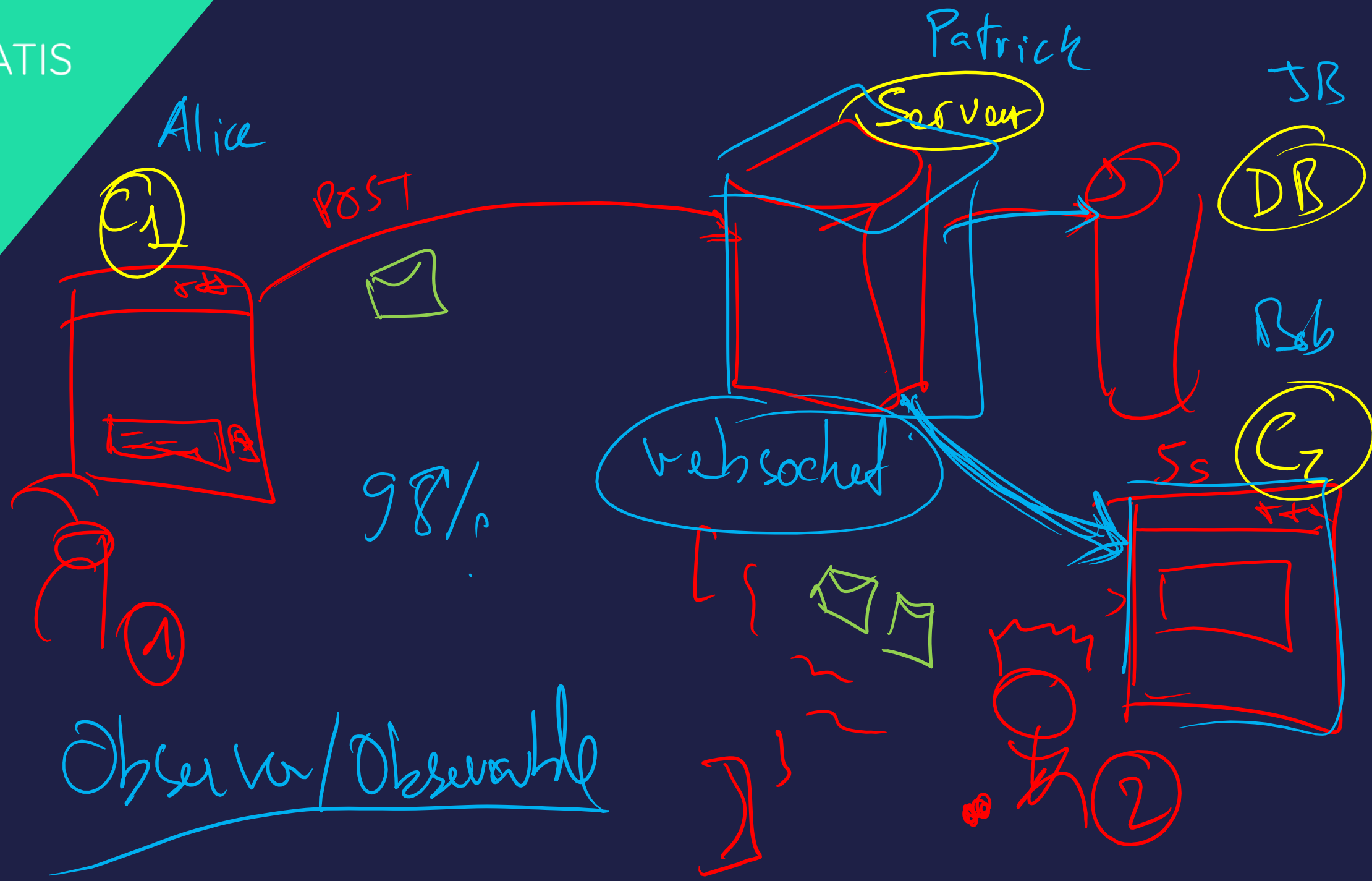
De part la nature de JS, il y a des designs pattern propres à ce langage ou dont l'implémentation est propre à ce langage

Designs pattern (patrons de conception) :

- Factory
- Scope-safe constructor
- Module Pattern
- Revealing Module Pattern
- Singleton

A quoi servent les
designs pattern ?





Instanciación

Consultons les membres de nos objets ! 😊
Avec `Object.getOwnPropertyNames`

Un **constructeur** (ou initialisateur) est une fonction qui vous permet de fournir toute initialisation personnalisée qui doit être effectuée avant qu'une autre méthode puisse être appelée sur un objet instancié.

Lorsqu'une fonction constructeur crée un objet, elle le « marque » :

- Par la propriété : `constructor`
- Par la propriété : `__proto__`

La plupart des constructeurs intégrés, tels que Object, Regex et Array, sont **scope-safe**.

Si new n'est pas utilisé, ils renvoient une instance appropriée de l'objet en appelant à nouveau le constructeur avec new.

```
function Voiture(price) {  
    if (!(this instanceof Voiture)) {  
        return new Voiture(price);  
    }  
    this.price = price;  
}  
  
const obj = Voiture(234);  
console.log(obj instanceof Voiture); // prints true
```




Quel est le type de class ?



```
class Voiture {  
  constructor(brand, model) {  
    this.brand = brand;  
    this.model = model;  
  }  
}  
  
console.log(typeof Voiture);
```



Les classes JavaScript ont été introduites avec ECMAScript 2015. Elles sont un « sucre syntaxique » par rapport à l'héritage prototypique.

En effet, cette syntaxe n'introduit pas de nouveau modèle d'héritage en JavaScript !

```
class Voiture {  
  model = 'Focus';  
  
  constructor(brand) {  
    this.brand = brand;  
  }  
}
```

```
const v = new Voiture('Ford');  
console.log(v);
```



Voiture

{ model: 'Focus', brand: 'Ford' }



```
class Voiture {  
  * model = 'Focus'; // property attached to the object  
  static price = 34; // property attached to the class  
  
  constructor(brand) {  
    * this.brand = brand; // property attached to the object  
  }  
  
  rouler() { // method attached to the prototype  
    console.log('Je roule');  
  }  
  
  static crash() { // method attached to the class  
    console.log('Toutes les voitures crashent');  
  }  
}
```

```
const v = new Voiture('Ford');  
console.log(v); // prints Voiture { model: 'Focus', brand: 'Ford' }  
Voiture.prototype.rouler(); // prints "Je roule"  
console.log(Voiture); // prints [Function: Voiture] { price: 34 }  
Voiture.crash(); // prints Toutes les voitures crashent
```

La retranscription des classes en fonction :

1. Les membres marqués « **static** » sont rattachés **directement à la classe** / fonction
2. Les **méthodes** sont rattachés **au prototype** de la classe / fonction (les objets en héritent)
3. Les **propriétés** sont rattachés à **l'instance** / objet (elles sont propres à l'objet)

Il est possible de
représenter une classe
graphiquement grâce
à l'UML (Unified
Modeling Language)

CHIEN
+ age + groupe sanguin + taille + poids
+ mange() + dort() + aboie()

Pour aller plus loin

<https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/Basics>