

# Histoire du Javascript

- Création : Mai 1996
- Créateur : Brendan Eich
- Influences : Self, Scheme2, Perl, C, C++, Java, Python
- Standard : Ecma International



# Définition javascript

- JavaScript est un langage de script léger, orienté objet, principalement connu comme le langage de script des pages web. Il est aussi utilisé dans de nombreux environnements extérieurs aux navigateurs web tels que Node.js, Deno, Bun et bien d'autres.
- Le code JavaScript est interprété ou compilé à la volée (JIT). C'est un langage à objets utilisant le concept de prototype, disposant d'un typage faible et dynamique qui permet de programmer suivant plusieurs paradigmes de programmation : fonctionnelle, impérative et orientée objet.

# Variables

Une variable doit être déclarée avant de pouvoir l'utiliser. Il y a 3 façons de le faire, en utilisant **var**, **let** ou **const**, et ces 3 manières diffèrent dans la façon dont vous pouvez interagir avec la variable plus tard.

Jusqu'à ES2015, **var** était le seul constructeur disponible pour définir les variables.

# Variables

- Une variable initialisée avec **var** en dehors de toute fonction est affecté à l'objet global, a une portée globale et est visible partout.
- Une variable initialisée avec **var** à l'intérieur d'une fonction est assignée à cette fonction, elle est locale et n'est visible qu'à l'intérieur, tout comme un paramètre de fonction.

# Variables

Il est important de comprendre qu'un bloc (identifié par une paire d'accolades) ne définit pas une nouvelle portée.

Une nouvelle portée n'est créée que lorsqu'une fonction est créée, car **var** n'a pas de portée de bloc, mais de portée de fonction.

# Variables

**let** est une nouvelle fonctionnalité introduite dans ES2015 et il s'agit essentiellement d'une version à portée de bloc de var. Sa portée est limitée au bloc, à l'instruction ou à l'expression où elle est définie et à tous les blocs internes contenus.

Les développeurs JavaScript modernes peuvent choisir d'utiliser uniquement let et abandonner complètement l'utilisation de var.

Définir **let** en dehors de toute fonction - contrairement à var - ne crée pas de variable globale.

# Variables

- Les variables déclarées avec **var** ou **let** peuvent être modifiées ultérieurement dans le programme et réaffectées. Une fois que **const** est initialisée, sa valeur ne peut plus jamais être modifiée et ne peut pas être réaffectée à une valeur différente.
- **Const** ne fournit pas l'immuabilité, mais s'assure simplement que la référence ne peut pas être modifiée. **const** a une portée de bloc, identique à **let**.
- Les développeurs JavaScript modernes peuvent choisir de toujours utiliser **const** pour les variables qui n'ont pas besoin d'être réaffectées plus tard dans le programme.
- Pourquoi? Parce que nous devrions toujours utiliser la construction la plus simple disponible pour éviter de faire des erreurs sur la route.

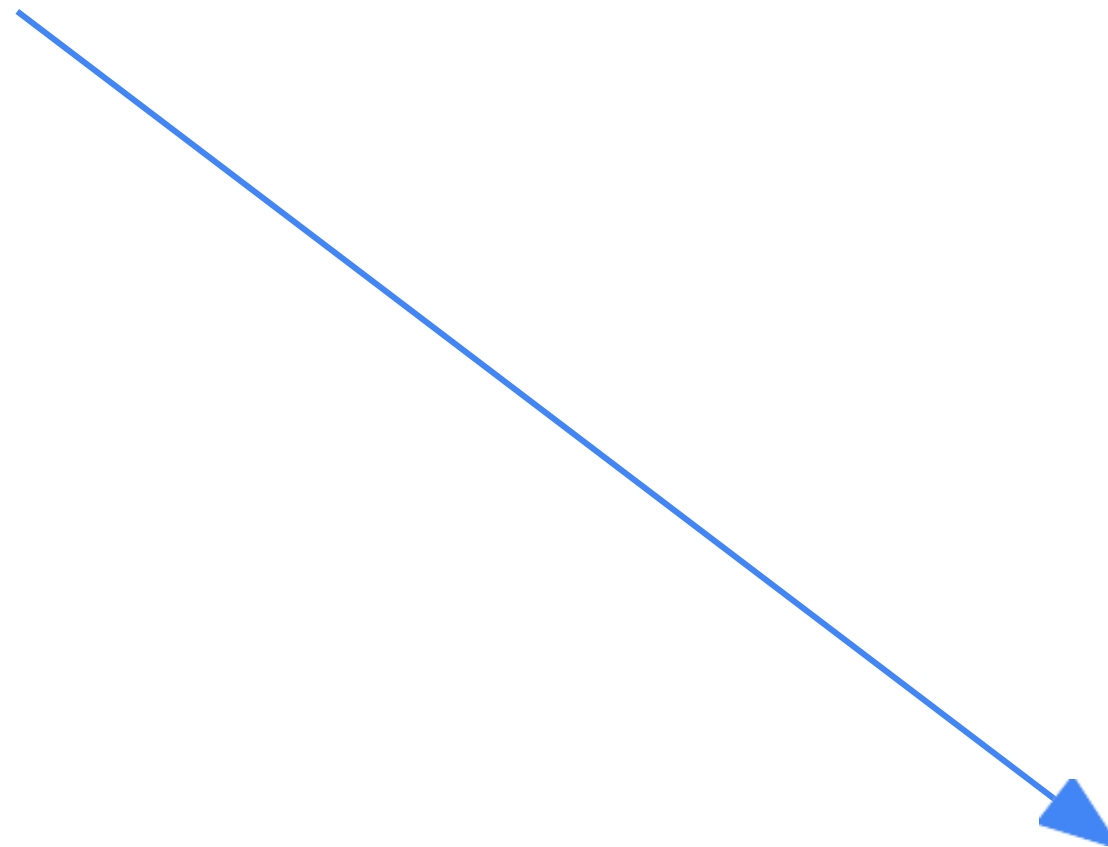
# Fonctions Fléchées

- Les fonctions fléchées ont été introduites dans ES6 / ECMAScript 2015, et depuis leur introduction, elles ont changé à jamais l'apparence (et le fonctionnement) du code JavaScript.
- À mon avis, ce changement était si accueillant que vous voyez maintenant rarement l'utilisation du mot-clé `function` dans les bases de code modernes.
- Visuellement, c'est un changement simple et bienvenu, qui vous permet d'écrire des fonctions avec une syntaxe plus courte



# Fonctions Fléchées

```
const myFunction = function() {  
  //...  
}
```



```
const myFunction = () => {  
  //...  
}
```

# Fonctions Fléchées

Si le corps de la fonction ne contient qu'une seule instruction, vous pouvez omettre les crochets et tout écrire sur une seule ligne :

```
const myFunction = () => doSomething()
```

Les paramètres sont passés entre parenthèses :

```
const myFunction = (param1, param2) => doSomething(param1, param2)
```

# Fonctions Fléchées

S'il n'y a qu'un seul paramètre on peut même oublier les parenthèses :

```
const myFunction = param => doSomething(param)
```

# Retour Implicite

Les fonctions fléchées permettent d'avoir un retour implicite : les valeurs retournées sans avoir à utiliser le mot-clé **return**.

Cela fonctionne lorsqu'il y a dans le corps une instruction d'une ligne de la fonction :

```
const myFunction = () => 'test'
```

```
myFunction() // 'test'
```

# Retour Implicite

Un autre exemple lors du retour d'un objet, n'oubliez pas d'envelopper les accolades entre parenthèses pour éviter qu'il ne soit considéré comme les crochets du corps de la fonction d'enveloppement :

```
const myFunction = () => ({ value: 'test' })  
  
myFunction() //{value: 'test'}
```

# this

**this** est un concept qui peut être compliqué à appréhender, car il varie beaucoup selon le contexte et varie également selon le mode de JavaScript ( mode strict ou non).

Il est important de clarifier ce concept car les fonctions fléchées se comportent très différemment des fonctions normales.

Lorsqu'il est défini comme une méthode d'un objet, dans une fonction régulière **this** fait référence à l'objet, vous pouvez donc faire :

# this

```
const car = {  
  model: 'Fiesta',  
  manufacturer: 'Ford',  
  fullName: function() {  
    return `${this.manufacturer} ${this.model}`  
  }  
}
```

l'appel de `car.fullName()` renverra “Ford Fiesta”

# this

Le **scope** de **this** dans les fonctions fléchées est **hérité** du contexte d'exécution.

Une fonction fléchée ne lie pas du tout **this**, donc sa valeur sera recherchée dans la pile des appels. Ici dans le code suivant, **car.fullName()** ne fonctionnera pas et renverra la chaîne "undefined undefined"



# this

```
const car = {  
  model: 'Fiesta',  
  manufacturer: 'Ford',  
  fullName: () => {  
    return `${this.manufacturer} ${this.model}`  
  }  
}
```

# this

- Pour cette raison, les fonctions fléchées ne sont pas adaptées en tant que méthodes d'objet.
- Les fonctions fléchées ne peuvent pas non plus être utilisées comme constructeurs, lorsque l'instanciation d'un objet lèvera une **TypeError**.
- C'est là que les fonctions normales doivent être utilisées à la place, lorsque le contexte dynamique n'est pas nécessaire.
- C'est également un problème lors de la gestion des événements. Le **DOM Event Listener** établit le **this** comme élément cible et si vous comptez sur **this** dans un gestionnaire d'événement, une fonction régulière est nécessaire:

# this

```
const link = document.querySelector('#link')
link.addEventListener('click', () => {
  // this === window
})
```

```
const link = document.querySelector('#link')
link.addEventListener('click', function() {
  // this === link
})
```

# Rest et spread

Vous pouvez « étendre » un tableau, un objet ou une chaîne de caractère avec l'opérateur spread ...

Exemple avec un array :

```
const a = [1, 2, 3]

const b = [...a, 4, 5, 6] //nouveau tableau

const c = [...a] //copie de tableau
```

# Rest et spread

Cela fonctionne également avec les objets :

```
const newObj = {...oldObj}
```

Avec les String, il crée un array séparant chaque caractère :

```
const hey = 'hey'  
const arrayized = [...hey] // ['h', 'e', 'y']
```

# Rest et spread

Cet opérateur a des applications pratiques. La plus utile est la possibilité d'utiliser un tableau comme argument de fonction d'une manière très simple :

```
const f = (foo, bar) => {}  
const a = [1, 2]  
f(...a)
```

# Rest et spread

L'élément **rest** `...` est utile lorsqu'on destructure les tableaux :

```
const numbers = [1, 2, 3, 4, 5]  
[first, second, ...others] = numbers
```

et lorsqu'on “étale” (spread) les éléments :

```
const numbers = [1, 2, 3, 4, 5]  
const sum = (a, b, c, d, e) => a + b + c + d + e  
const sumOfNumbers = sum(...numbers)
```

# Rest et spread

ES2018 introduit l'utilisation de rest pour les objets :

```
const { first, second, ...others } = {  
  first: 1,  
  second: 2,  
  third: 3,  
  fourth: 4,  
  fifth: 5  
}  
  
first // 1  
second // 2  
others // { third: 3, fourth: 4, fifth: 5 }
```



# Rest et spread

Les propriétés du **spread** permettent de créer un nouvel objet en combinant les propriétés de l'objet passées après l'opérateur de propagation :

```
const items = { first, second, ...others }  
items // { first: 1, second: 2, third: 3, fourth: 4, fifth: 5 }
```

# Déstructuration d'objets et de tableaux

Avec un un objet, en utilisant la syntaxe de déstructuration, vous pouvez extraire quelques valeurs et les mettre dans des variables nommées :

```
const person = {  
  firstName: 'Tom',  
  lastName: 'Cruise',  
  actor: true,  
  age: 54 //made up  
}
```

```
const { firstName: name, age } = person //name: Tom, age: 54
```

Maintenant name et age contiennent ces valeurs

# Déstructuration d'objets et de tableaux

Cette syntaxe fonctionne également sur les tableaux :

```
const a = [1, 2, 3, 4, 5]  
const [first, second] = a
```

Cette instruction crée 3 nouvelles variables en obtenant les éléments d'indice 0, 1, 4 du tableau a:

```
const [first, second, , , fifth] = a
```

# Template literals

Les modèles de littéraux sont une nouvelle fonctionnalité ES2015/ES6 qui vous permet de travailler avec des chaînes d'une manière nouvelle par rapport à ES5 et aux versions antérieures.

La syntaxe à première vue est très simple, il suffit d'utiliser des backticks au lieu de guillemets simples ou doubles :

```
const a_string = `something`
```

# Template literals

- Ils sont uniques car ils offrent de nombreuses fonctionnalités que les chaînes normales construites avec des guillemets n'offrent pas, en particulier :
  - ils offrent une excellente syntaxe pour définir des chaînes multilignes
  - ils fournissent un moyen facile d'interpoler (insérer) des
    - variables et des expressions dans des chaînes
  - ils vous permettent de créer des DSL avec des balises de modèle (DSL signifie Domain Specific Language, et il est par exemple utilisé dans React by Styled Components, pour définir le CSS pour un composant)
- Nous allons les aborder plus en détail.

# Multiline strings

Avant l'ES6, pour créer une chaîne s'étendant sur deux lignes, vous deviez utiliser le caractère \ en fin de ligne :

```
const string =  
  'first part \  
second part'
```

Cela permet de créer une chaîne sur 2 lignes, mais elle est rendue sur une seule ligne : **first part second part**

# Multiline strings

Pour rendre également la chaîne sur plusieurs lignes, vous devez explicitement ajouter `\n` à la fin de chaque ligne, comme ceci :

```
const string =  
    'first line\n \n'  
    'second line'
```

```
const string = 'first line\n' + 'second line'
```

# Multiline strings

- Les littéraux de modèle rendent les chaînes multilignes beaucoup plus simples.
- Une fois qu'un modèle littéral est ouvert avec le backtick, il vous suffit d'appuyer sur Entrée pour créer une nouvelle ligne, sans caractères spéciaux, et elle est rendue telle quelle :

```
const string = `Salut  
tout  
le  
monde!`  
  
const string = `First  
Second`
```



# Multiline strings

Nous avons un petit problème dans le second exemple c'est que l'espace est pris en compte et décale énormément le mot Second à l'affichage.

```
const string =  
`  First  
Second`.trim()
```

# Interpolation

Les littéraux de modèle offrent un moyen simple d'interpoler des variables et des expressions dans des chaînes.

Vous le faites en utilisant la syntaxe **`${variable}`**:

```
const myVariable = 'bleu'  
const string = `Chat ${myVariable}` //Chat bleu
```

# Interpolation

à l'intérieur de `${}` vous pouvez ajouter n'importe quoi, même des expressions :

```
const string = `something ${1 + 2 + 3}`  
const string2 = `something ${foo() ? 'x' : 'y'}`
```

# Les Classes

En 2015, la norme ECMAScript 6 (ES6) a introduit les classes.

JavaScript a un moyen assez rare d'implémenter l'héritage : **l'héritage prototypique**. [L'héritage de prototype](#) est différent de l'implémentation de l'héritage de la plupart des autres langages de programmation populaires, qui est basée sur les classes.

Les personnes venant de Java ou Python ou d'autres langages avaient du mal à comprendre les subtilités de l'héritage prototypique, alors le comité ECMAScript a décidé de saupoudrer de sucre syntaxique sur l'héritage prototypique afin qu'il ressemble à la façon dont l'héritage basé sur les classes fonctionne dans d'autres implémentations populaires.

C'est important : JavaScript sous le capot est toujours le même, et vous pouvez accéder à un prototype d'objet de la manière habituelle.

# Définition de Classe

Voilà à quoi ressemble une classe

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    hello() {  
        return 'Salut je m\'appelle' + this.name + '.'  
    }  
}
```

# Définition de Classe

- Une classe a un identifiant, que nous pouvons utiliser pour créer de nouveaux objets en utilisant **new ClassIdentifier()**.
- Lorsque l'objet est initialisé, la méthode **constructor** est appelée, avec tous les paramètres passés.
- Une classe a aussi autant de méthodes qu'elle en a besoin. Dans ce cas **hello** est une méthode et peut être appelée sur tous les objets dérivés de cette classe :

```
const jojo = new Person('Jojo')  
jojo.hello()
```

# Héritage de Classe

- Une classe peut étendre une autre classe et les objets initialisés à l'aide de cette classe héritent de toutes les méthodes des deux classes.
- Si la classe héritée possède une méthode portant le même nom que l'une des classes supérieures dans la hiérarchie, la méthode la plus proche est prioritaire :

# Héritage de Classe

```
class Person {  
  constructor(name) {  
    this.name = name  
  }  
  
  hello() {  
    return 'Salut je m\'appelle ' +  
this.name + '.'  
  }  
}
```

```
class Programmer extends Person {  
  hello() {  
    return super.hello() + ' Je suis un joueur.'  
  }  
}  
  
const jojo = new Programmer('Jojo')  
jojo.hello()
```



# Méthodes statiques

Normalement, les méthodes sont définies pour une instance, pas pour une classe.

Les méthodes statiques sont exécutées par la classe directement et non pas par une instance:

```
class Person {  
    static personHello() {  
        return 'Hello'  
    }  
}  
  
Person.personHello() //Hello
```

# Getters et setters

Vous pouvez ajouter des méthodes préfixées par **get** ou **set** pour créer un **getter** et un **setter**, qui sont deux morceaux de code différents qui sont exécutés en fonction de ce que vous faites:  
accéder à la variable ou modifier sa valeur.

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    set name(value) {  
        this.name = value  
    }  
  
    get name() {  
        return this.name  
    }  
}
```

# Getters et setters

Si vous n'avez qu'un getter, la propriété ne peut pas être définie et toute tentative de la faire sera ignorée.

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    get name() {  
        return this.name  
    }  
}
```

# Getters et setters

Si vous n'avez qu'un setter, vous pouvez modifier la valeur mais pas y accéder de l'extérieur.

```
class Person {  
    constructor(name) {  
        this.name = name  
    }  
  
    set name() {  
        this.name = value  
    }  
}
```

# Callbacks

Les ordinateurs sont asynchrones par conception.

Asynchrone signifie que les choses peuvent se produire indépendamment du flux principal du programme.

Dans les ordinateurs grand public actuels, chaque programme s'exécute pendant un intervalle de temps spécifique, puis il arrête son exécution pour laisser un autre programme poursuivre son exécution. Cette chose fonctionne dans un cycle si rapide qu'il est impossible de le remarquer, et nous pensons que nos ordinateurs exécutent de nombreux programmes simultanément, mais c'est une illusion (sauf sur les machines multiprocesseurs).

# Callbacks

Les programmes utilisent en interne des interruptions, un signal qui est émis vers le processeur pour attirer l'attention.

Simplement gardez à l'esprit qu'il est normal que les programmes soient asynchrones et arrêtent leur exécution jusqu'à ce qu'ils aient besoin d'attention, et que l'ordinateur puisse exécuter d'autres choses entre-temps. Lorsqu'un programme attend une réponse du réseau, il ne peut pas arrêter le processeur tant que la requête n'est pas terminée.

# Callbacks

- Normalement, les langages de programmation sont synchrones et certains offrent un moyen de gérer l'asynchronisme, dans le langage ou via des bibliothèques. C, Java, C#, PHP, Go, Ruby, Swift, Python, ils sont tous synchrones par défaut. Certains d'entre eux gèrent l'async en utilisant des threads, engendrant un nouveau processus.
- JavaScript est synchrone par défaut et est monothread. Cela signifie que le code ne peut pas créer de nouveaux threads et s'exécuter en parallèle.

# Callbacks

Les lignes de code sont exécutées en série, l'une après l'autre, par exemple :

```
const a = 1  
const b = 2  
const c = a * b  
  
console.log(c)  
doSomething()
```



# Callbacks

- Mais JavaScript est né à l'intérieur du navigateur, son travail principal, au début, était de répondre aux actions des utilisateurs, comme `onClick`, `onMouseOver`, `onChange`, `onSubmit` etc. Comment pourrait-il le faire avec un modèle de programmation synchrone ?
- La réponse était dans son environnement. Le navigateur fournit un moyen de le faire en fournissant un ensemble d'API pouvant gérer ce type de fonctionnalité.
- Plus récemment, Node.js a introduit un environnement non bloquant pour étendre ce concept à l'accès aux fichiers, aux appels réseau, etc.

# Callbacks

Vous ne pouvez pas savoir quand un utilisateur va cliquer sur un bouton, vous définissez donc un gestionnaire d'événements pour l'événement click. Ce gestionnaire d'événements accepte une fonction, qui sera appelée lorsque l'événement est déclenché

```
document.getElementById('button').addEventListener('click', () => {  
    //item clicked  
})
```

C'est ce qu'on appelle un **callback**

# Callbacks

- Un callback est une fonction simple qui est transmise en tant que valeur à une autre fonction et ne sera exécutée que lorsque l'événement se produira. Nous pouvons le faire car JavaScript a des fonctions de première classe, qui peuvent être affectées à des variables et transmises à d'autres fonctions (appelées **fonctions d'ordre supérieur** )
- Il est courant d'encapsuler tout votre code client dans un **load eventListener** sur l'objet **window**, qui exécute la fonction de rappel uniquement lorsque la page est prête :

# Callbacks

```
window.addEventListener('load', () => {  
    //window loaded  
    //fonctions à charger  
})
```

# Callbacks

Les callbacks sont utilisés partout, pas seulement dans les événements DOM.  
Un exemple courant consiste à utiliser des minuteries

```
setTimeout(() => {  
    // se lance après 2 secondes  
}, 2000)
```

# Gestion des erreurs dans les Callbacks

- Comment gérer les erreurs avec les rappels ? Une stratégie très courante consiste à utiliser ce que Node.js a adopté : le premier paramètre de toute fonction de rappel est l'objet d'erreur **error-first callbacks**.
- S'il n'y a pas d'erreur, l'objet est null. S'il y a une erreur, il contient une description de l'erreur et d'autres informations

# Gestion des erreurs dans les Callbacks

```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    //handle error  
    console.log(err)  
    return  
  }  
  
  //no errors, process data  
  console.log(data)  
})
```

# Le problème des Callbacks

Les rappels sont parfaits pour les cas simples !

Cependant chaque callback ajoute un niveau d'imbrication, et quand vous avez beaucoup de callbacks, le code commence à se compliquer très vite :



# Le problème des Callbacks

4 niveaux d'imbrication

```
window.addEventListener('load', () => {  
  document.getElementById('button').addEventListener('click', () => {  
    setTimeout(() => {  
      items.forEach(item => {  
        //your code here  
      })  
    }, 2000)  
  })  
})
```

# Alternative aux Callbacks

À partir de ES6, JavaScript a introduit plusieurs fonctionnalités qui nous aident avec du code asynchrone qui n'implique pas l'utilisation de rappels :

- Promises (ES6)
- Async/Await (ES8)

# Promises

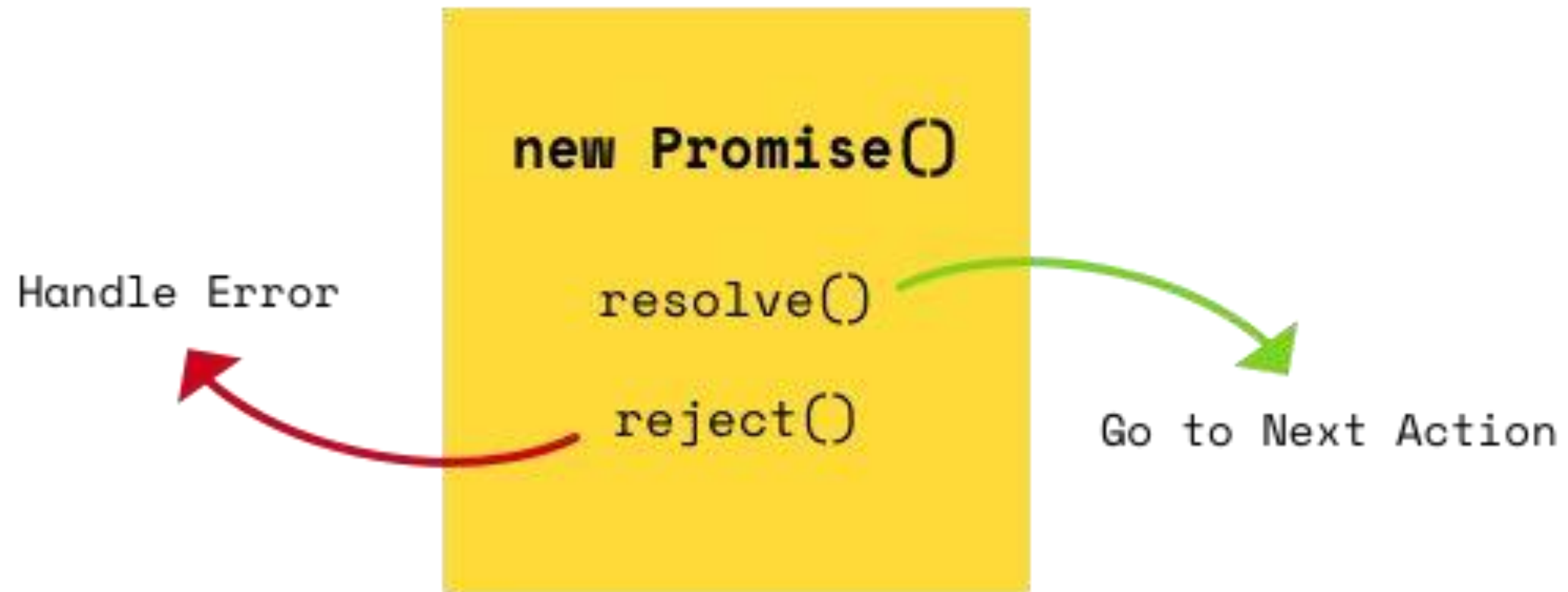
- Les promesses sont un moyen de traiter le code asynchrone, sans écrire trop de rappels dans votre code.
- Bien qu'ils existent depuis des années, ils ont été standardisés et introduits dans ES2015 et maintenant ils sont remplacés dans ES2017 par des fonctions asynchrones.
- Les fonctions asynchrones utilisent l'API des promesses comme bloc de construction, il est donc fondamental de les comprendre même dans un code plus récent, vous utiliserez probablement des fonctions asynchrones au lieu des promesses.

# Comment fonctionnent les Promesses ?

Une fois qu'une promesse a été appelée, elle démarre en état d'attente (pending state). Cela signifie que la fonction appelante continue l'exécution, pendant qu'elle attend la promesse de faire son propre traitement, et donne à la fonction appelante un retour d'informations.

À ce stade, la fonction appelante attend qu'elle renvoie la promesse dans un état résolu, ou dans un état rejeté, mais comme vous le savez, JavaScript est asynchrone, donc la fonction continue son exécution pendant que la promesse fonctionne.

# Comment fonctionnent les Promesses ?



# Quelle API de JS utilise les Promesses ?

En plus de votre propre code et de votre code de bibliothèque, les promesses sont utilisées par les API Web modernes standard telles que Fetch.

Il est peu probable qu'en JavaScript moderne, vous n'utilisiez *pas de* promesses, nous allons pour cela les aborder.

# Créer une Promesse ?

L'API Promise expose un constructeur Promise, que vous initialisez à l'aide de `new Promise()` :

```
let done = true

const isItDoneYet = new Promise((resolve, reject) => { if (done) {
  const workDone = 'Here is the thing I built'
  resolve(workDone)
} else {
  const why = 'Still working on something else' reject(why)
}
})
```

# Créer une Promesse ?

- Comme vous pouvez le voir, la promesse vérifie la constante **done** globale, et si c'est vrai, nous retournons une promesse résolue, sinon une promesse rejetée.
- À l'aide de **resolve** et **reject** nous pouvons communiquer en retour une valeur, dans le cas ci-dessus, nous renvoyons simplement une chaîne, mais cela pourrait également être un objet.



# Utiliser une Promesse ?

```
const isItDoneYet = new Promise()  
//...  
  
const checkIfItsDone = () => {  
  isItDoneYet  
    .then(ok => {  
      console.log(ok)  
    })  
    .catch(err => {  
      console.error(err)  
    })  
}
```

# Utiliser une Promesse ?

Le lancement de **checkIfItsDone()** exécutera la promesse **isItDoneYet()** et attendra qu'elle se résolve, en utilisant le callback **then**, et s'il y a une erreur, il la traitera dans le callback **catch**.

# Enchaîner les Promesses ?

Une promesse peut être retournée à une autre promesse, créant une chaîne de promesses.

Un excellent exemple de chaînage de promesses est donné par l'API Fetch, une couche au-dessus de l'API XMLHttpRequest, que nous pouvons utiliser pour obtenir une ressource et mettre en file d'attente une chaîne de promesses à exécuter lorsque la ressource est récupérée.

L'API Fetch est un mécanisme basé sur la promesse, et l'appel `fetch()` équivaut à définir notre propre promesse en utilisant `new Promise()`.

# Enchaîner les Promesses ?

```
const status = response => {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response)
  }
  return Promise.reject(new Error(response.statusText))
}

const json = response =>

response.json() fetch('/todos.json')
  .then(status)
  .then(json)
  .then(data => {
    console.log('Request succeeded with JSON response', data)
  })
  .catch(error => {
    console.log('Request failed', error)
  })
})
```

# Enchaîner les Promesses ?

Dans cet exemple, nous appelons **fetch()** pour obtenir une liste des éléments TODO du fichier **todos.json** trouvé à la racine du domaine, et nous créons une chaîne de promesses.

Lancer **fetch()** renvoie une [réponse](#), qui a de nombreuses propriétés, et parmi celles que nous référençons :

- **status**, une valeur numérique représentant le code d'état HTTP
- **statusText**, un message d'état, qui est OK si la demande aboutit

# Enchaîner les Promesses ?

- **response** a également une méthode `json()`, qui renvoie une promesse qui se résoudra avec le contenu du corps traité et transformé en JSON.
- Donc, selon ces promesses, voici ce qui se passe : la première promesse de la chaîne est une fonction que nous avons définie, appelée **status()**, qui vérifie l'état de la réponse et s'il ne s'agit pas d'une réponse réussie (entre 200 et 299), il rejette la promesse.
- Cette opération fera en sorte que la chaîne de promesses **ignorera** toutes les promesses enchaînées répertoriées et passera directement au **catch()** en bas, en enregistrant le texte **Request failed** accompagné du message d'erreur.

# Enchaîner les Promesses ?

Si cela réussit, il appelle la fonction **json()** que nous avons définie. Étant donné que la promesse précédente, une fois réussie, renvoyait l'objet `response`, nous l'obtenons en entrée de la deuxième promesse.

Dans ce cas, nous renvoyons les données JSON traitées, donc la troisième promesse reçoit directement le JSON :

```
.then((data) => {  
    console.log('Request succeeded with JSON response', data)  
})
```

# Gestion des erreurs

Dans l'exemple ci-dessus, dans la section précédente, nous avons eu un catch qui était annexé à la chaîne des promesses. Lorsqu'un élément de la chaîne de promesses échoue et génère une erreur ou rejette la promesse, le contrôle passe à la déclaration `catch()` la plus proche en bas de la chaîne



# Gestion des erreurs

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
}).catch(err => {  
  console.error(err)  
})
```

// or

```
new Promise((resolve, reject) => {  
  reject('Error')  
}).catch(err => {  
  console.error(err)  
})
```

# Erreurs en cascade

Si à l'intérieur du `catch()` une erreur est levée, on peut en ajouter une seconde pour gérer le premier, et ainsi de suite.

```
new Promise((resolve, reject) => {  
  throw new Error('Error')  
})  
  .catch(err => {  
    throw new Error('Error')  
  })  
  .catch(err => {  
    console.error(err)  
  })
```

# Orchestrer les promesses avec

**`Promise.all()`**

Si vous devez synchroniser différentes promesses, **`Promise.all()`** aide à définir une liste de promesses et à exécuter quelque chose lorsqu'elles sont toutes résolues.

# Orchestrer les promesses avec

**Promise.all()**

```
const f1 = fetch('/something.json')
const f2 = fetch('/something2.json')

Promise.all([f1, f2])
  .then(res => {
    console.log('Array of results', res)
  })
  .catch(err => {
    console.error(err)
  })
```

# Orchestrer les promesses avec

**`Promise.all()`**

La syntaxe d'affectation de déstructuration ES2015 vous permet également de le faire

```
Promise.all([f1, f2]).then(([res1, res2]) => {  
  console.log('Results', res1, res2)  
})
```

# Orchestrer les promesses avec `Promise.race()`

`Promise.race()` s'exécute dès que l'une des promesses que vous lui transmettez est résolue, et il exécute le callback qui y est joint avec le résultat de la dernière promesse résolue.

# Orchestrer les promesses avec `Promise.race()`

```
const promiseOne = new Promise((resolve, reject) => {
  setTimeout(resolve, 500, 'one')
})
const promiseTwo = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'two')
})

Promise.race([promiseOne, promiseTwo]).then(result => {
  console.log(result) // 'two'
})
```

# Async/Await

- JavaScript a évolué en très peu de temps des rappels aux promesses (ES2015), et depuis ES2017, le JavaScript asynchrone est encore plus simple avec la syntaxe `async/await`.
- Les fonctions asynchrones sont une combinaison de promesses et de générateurs, et fondamentalement, elles constituent une abstraction de niveau supérieur par rapport aux promesses. Attention : **`async/await` est juste construit sur des promesses**



# Pourquoi Async/Await a été introduit ?

- Ils réduisent le passe-partout autour des promesses et la limitation « ne pas rompre la chaîne » des promesses enchaînées.
- Lorsque les promesses ont été introduites dans ES2015, elles étaient destinées à résoudre un problème de code asynchrone, et elles l'ont fait, mais au cours des 2 années qui ont séparé ES2015 et ES2017, il était clair que les *promesses ne pouvaient pas être la solution finale*.
- Des promesses ont été introduites pour résoudre le fameux problème nommé *callback hell*, mais elles ont introduit une complexité par leur nature aussi et leur syntaxe

# Pourquoi Async/Await a été introduit ?

- C'étaient de bonnes primitives autour desquelles une meilleure syntaxe pouvait être exposée aux développeurs, donc quand le moment était venu, nous avons eu des fonctions asynchrones.
- Ils donnent l'impression que le code est synchrone, mais il est asynchrone et non bloquant dans les coulisses.

# Comment ça fonctionne ?

Une fonction async renvoie une promesse :

```
const doSomethingAsync = () => {  
  return new Promise(resolve => {  
    setTimeout(() => resolve('Fin du process'), 3000)  
  })  
}
```

# Comment ça fonctionne ?

Lorsque vous souhaitez appeler cette fonction, vous ajoutez **await**, et le **code appelant s'arrêtera jusqu'à ce que la promesse soit résolue ou rejetée**



Remarque : la fonction client doit être définie comme sync. Voici un exemple :

```
const doSomething = async () => {  
  console.log(await doSomethingAsync())  
}
```

# Testons ce code

```
const doSomethingAsync = () => { return new
  Promise(resolve => {
    setTimeout(() => resolve('I did something'), 3000)
  })
}

const doSomething = async () => { console.log(await
  doSomethingAsync())
}

console.log('Before')
doSomething()
console.log('After')
```

# Le code est plus simple à lire

- Comme vous pouvez le voir dans l'exemple ci-dessus, ce code semble plus simple. Comparez-le au code utilisant des promesses simples, avec des fonctions de chaînage et de rappel.
- Et c'est un exemple très simple, les principaux avantages se produiront lorsque le code est beaucoup plus complexe.
- Par exemple, voici comment obtenir une ressource JSON et l'analyser à l'aide de promesses :

# Le code est plus simple à lire

```
const getFirstUserData = () => {  
  return fetch('/users.json') // get users list  
    .then(response => response.json()) // parse JSON  
    .then(users => users[0]) // pick first user  
    .then(user => fetch(`/users/${user.name}`)) // get user data  
    .then(userResponse => userResponse.json()) // parse JSON  
}
```

```
getFirstUserData()
```

# Le code est plus simple à lire

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json') // get users list  
  const users = await response.json() // parse JSON  
  const user = await users[0] // pick first user  
  const userResponse = await fetch(`/users/${user.name}`) // get  
user data  
  const userData = await userResponse.json() // parse JSON  
  return userData  
}
```

```
getFirstUserData()
```



# Débogage plus facile

Le débogage des promesses est difficile car le débogueur ne franchira pas le code asynchrone.

Async/await rend cela très facile car pour le compilateur, c'est comme du code synchrone.

# Module ES

ES Modules est la norme ECMAScript pour travailler avec des modules.

Alors que Node.js utilise la norme CommonJS depuis des années, le navigateur n'a jamais eu de système de module, car chaque décision importante telle qu'un système de module doit d'abord être standardisée par ECMAScript puis implémentée par le navigateur.

Ce processus de normalisation achevé avec ES6 et les navigateurs, cette norme a été implémentée, fonctionnant de la même manière, et maintenant les modules ES sont pris en charge dans Chrome, Safari, Edge et Firefox (depuis la version 60).

# Module ES

Les modules sont très intéressants, car ils vous permettent d'encapsuler toutes sortes de fonctionnalités et d'exposer ces fonctionnalités à d'autres fichiers JavaScript, en tant que bibliothèques

# Syntaxe des module ES

La syntaxe pour importer un module est :

```
import package from 'module-name'
```

tandis que CommonJS utilise :

```
const package = require('module-name')
```

# Syntaxe des module ES

Un module est un fichier JavaScript qui **exporte** une ou plusieurs valeurs (objets, fonctions ou variables), en utilisant le mot-clé **export**. Par exemple, ce module exporte une fonction qui renvoie une chaîne en majuscule :

***majuscule.js***

```
export default str => str.toUpperCase()
```

# Syntaxe des module ES

Dans cet exemple, le module définit un seul **export par défaut** , il peut donc s'agir d'une fonction anonyme. Sinon, il faudrait un nom pour le distinguer des autres exportations.

Désormais, **tout autre module JavaScript** peut importer la fonctionnalité offerte par majuscule.js en l'important.

Une page HTML peut ajouter un module en utilisant un tag **<script>** avec l'attribut **type="module"** :

# Syntaxe des module ES

Il est important de noter que tout script chargé avec `type="module"` est chargé en **mode strict**.

Dans cet exemple, le module `majuscule.js` définit une **exportation par défaut**, donc lorsque nous l'importons, nous pouvons lui attribuer un nom que nous préférons :

```
import toUpperCase from './uppercase.js'
```

```
toUpperCase('test') // 'TEST'
```

# Syntaxe des module ES

Vous pouvez également utiliser un chemin absolu pour l'import de module, pour référencer des modules définis sur un autre domaine :

```
import toUpperCase from  
'https://stock-of-modules-example.glitch.me/majuscule.js'
```

```
import { foo } from '/majuscule.js'  
import { foo } from '../majuscule.js'
```



# Autres options d'import/export

Nous avons vu cet exemple :

```
export default str => str.toUpperCase()
```

Cela crée une exportation par défaut. Dans un fichier, cependant, vous pouvez exporter plus d'une chose, en utilisant cette syntaxe :

```
const a = 1  
const b = 2  
const c = 3  
  
export { a, b, c }
```

# Autres options d'import/export

Puis importer toutes ces fonctionnalités en utilisant :

```
import * from 'module'
```

Vous pouvez importer seulement quelques-unes de ces exportations, en utilisant l'affectation de déstructuration :

```
import { a } from 'module'  
import { a, b } from 'module'
```

# Autres options d'import/export

Vous pouvez renommer n'importe quelle importation, plus facilement, avec **as** :

```
import { a, b as two } from 'module'
```

Vous pouvez importer l'exportation par défaut et toute exportation qui n'est pas par défaut par son nom, comme dans cette importation React :

```
import React, { Component } from 'react'
```



TypeScript

# Outils



Accès à Internet ( La DOC + package NPM )



Navigateur web (à jour) - La console + ES6+



Éditeur de code - Visual Studio Code (Intellisense)



On aura besoin de NPM pour installer TypeScript

# Installation du compilateur TypeScript via NPM

Le compilateur vient avec un package permettant de comprendre le TypeScript



Vérifier que Node est installé: `node -v` ou `node --version`

```
sudo npm install -g typescript
```

# C'est quoi TypeScript ?

TypeScript est un langage open source développé par Microsoft et devenu très populaire suite à son intégration depuis Angular 2.



C'est un sur-ensemble permettant d'enrichir le JavaScript avec des fonctionnalités supplémentaires. Il s'appuie sur le langage JavaScript (et ses mises à jours) afin de lui apporter d'avantage de capacités.

JavaScript a ses propres limites et qui peuvent causer des problèmes!



Exemple: Envoyer quelqu'un pour nous acheter une pizza





# Limites du JavaScript

JavaScript n'exige pas de passer un argument au moment d'invoquer une fonction.

?

```
function getLowercaseString(arg) {  
    return arg.toLowerCase( );  
}
```

Cela va nous causer un problème!  
On constatera l'erreur une fois  
le code lancé !!!!

```
getLowercaseString( );
```



# Solution TypeScript

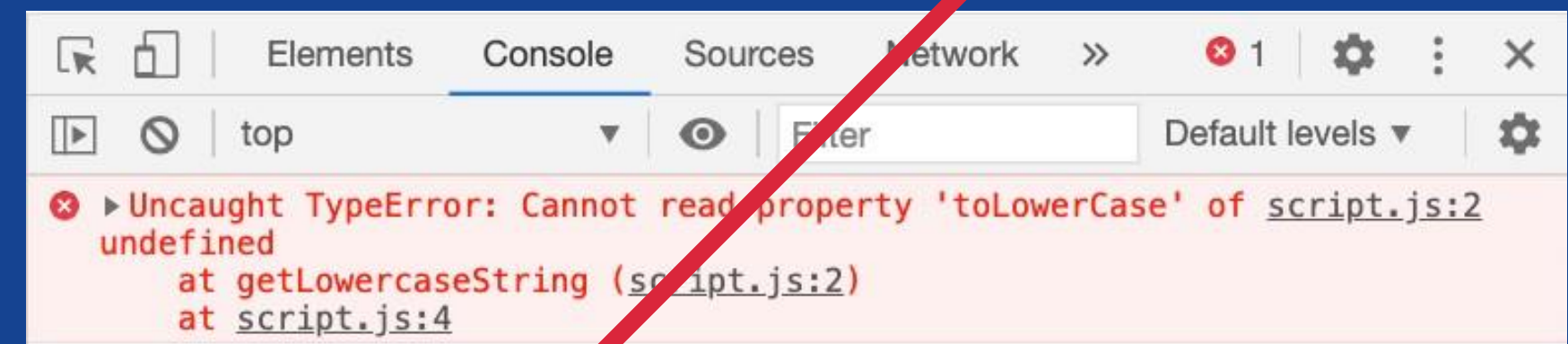


```
function getLowercaseString(arg: string) {  
    return arg.toLowerCase( );  
}
```

```
getLowercaseString( "Hello" );
```



```
getLowercaseString( 200 );
```

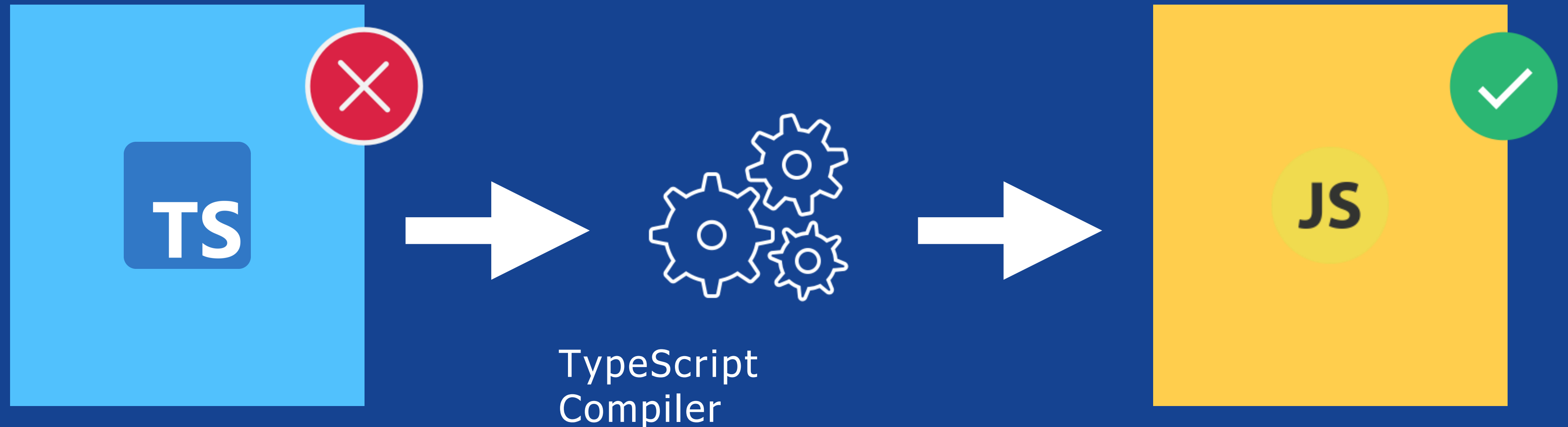


TypeScript exige de préciser le TYPE d'argument à passer dans la fonction.

- TypeScript exige de passer un argument
- correct au moment d'invoquer la fonction.

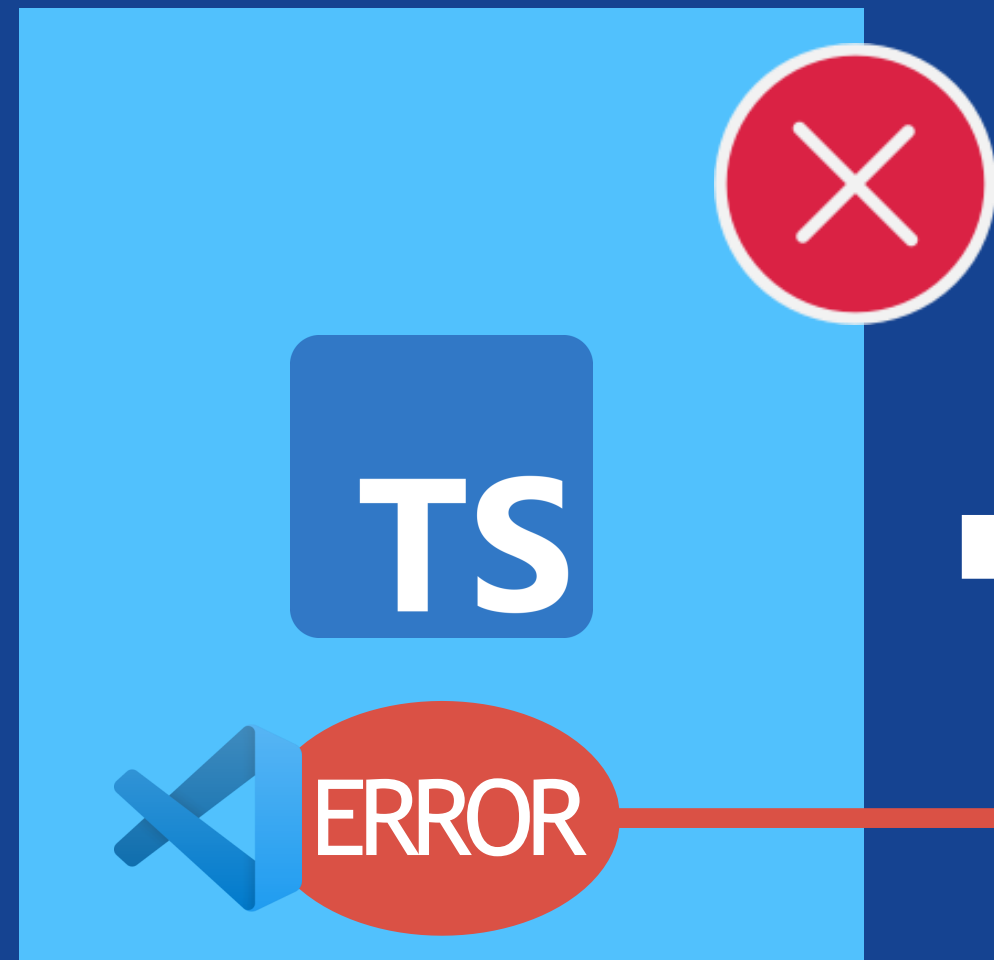
# TypeScript et les navigateurs ?

Le code TypeScript n'est pas reconnu par les navigateurs web, il faut donc le compiler en langage JavaScript



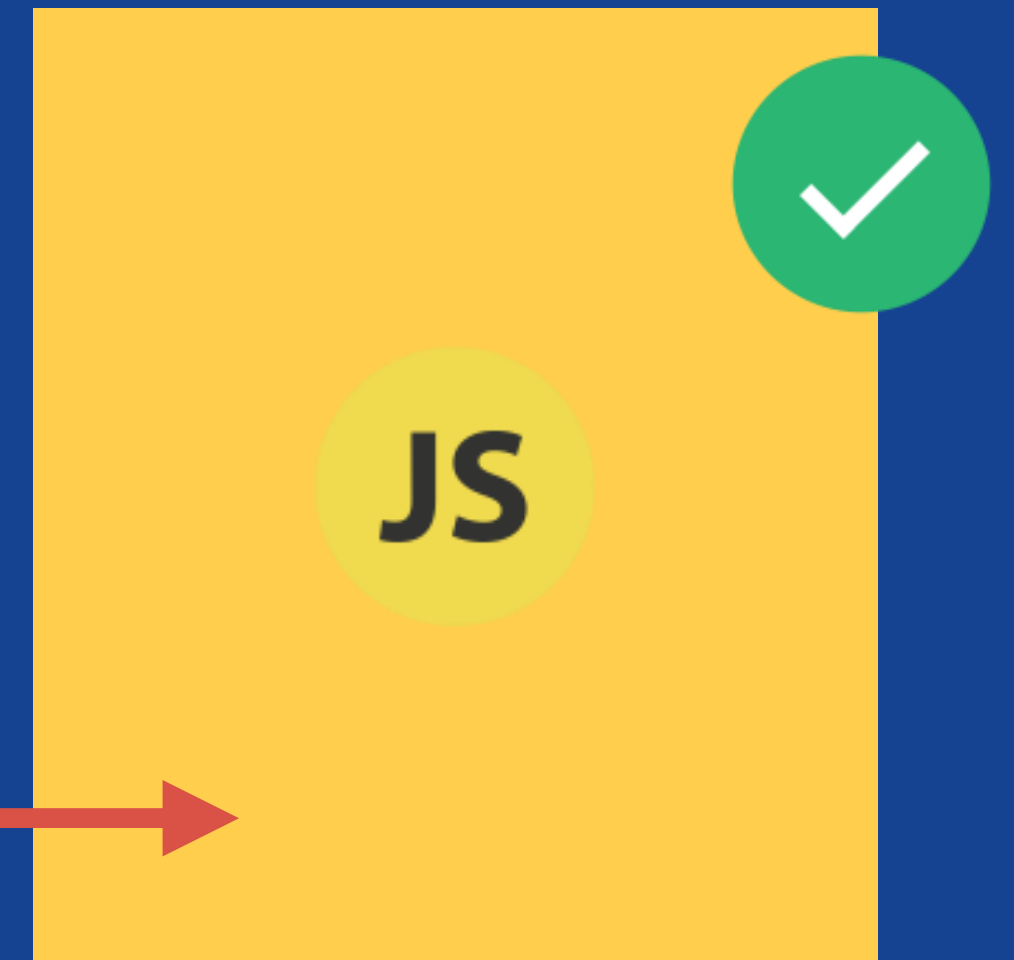
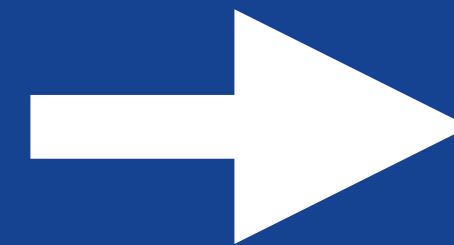
# Pourquoi TS pour obtenir du JS ?

Les fonctionnalités TS seront utiles ici et permettront d'obtenir un meilleur code JavaScript



TypeScript  
Compiler

Et nous alerter  
Des Erreurs





PIZZA type: 4  
fromages

Compilateur TypeScript

# Dois-je utiliser TypeScript dans tous mes projets ?

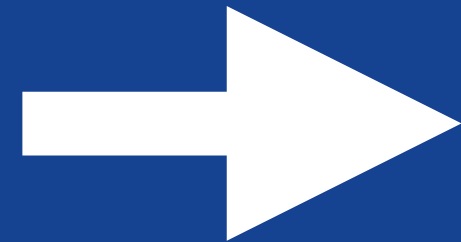
Il est conseillé d'utiliser TypeScript sur les grands projets.

- Permet de définir les types static de nos variables, fonctions ..etc (ce qui n'est pas le cas avec JavaScript)
- Permet une meilleure documentation du code ( facile à comprendre )
- Vérifie que notre code fonctionne bien ( constater les erreurs durant le dev )
- Permet d'écrire un code plus sûr et plus clean. (Cependant, TS n'est pas une raison pour ne pas tester votre code. Tout code qui se respecte doit être testé!)

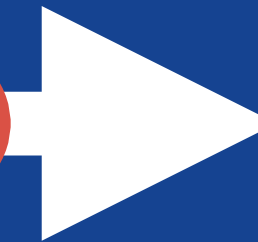
Types dynamiques acceptés en JavaScript mais peut causer des erreurs ou des incohérences

**TS**

```
Let firstName =  
'toto'; firstName =  
12345;
```



**ERROR**

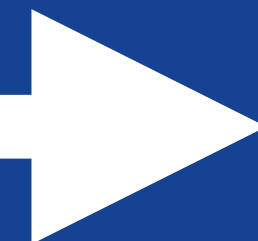
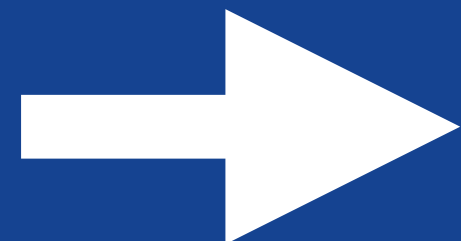


**JS**

Types stricts en TypeScript

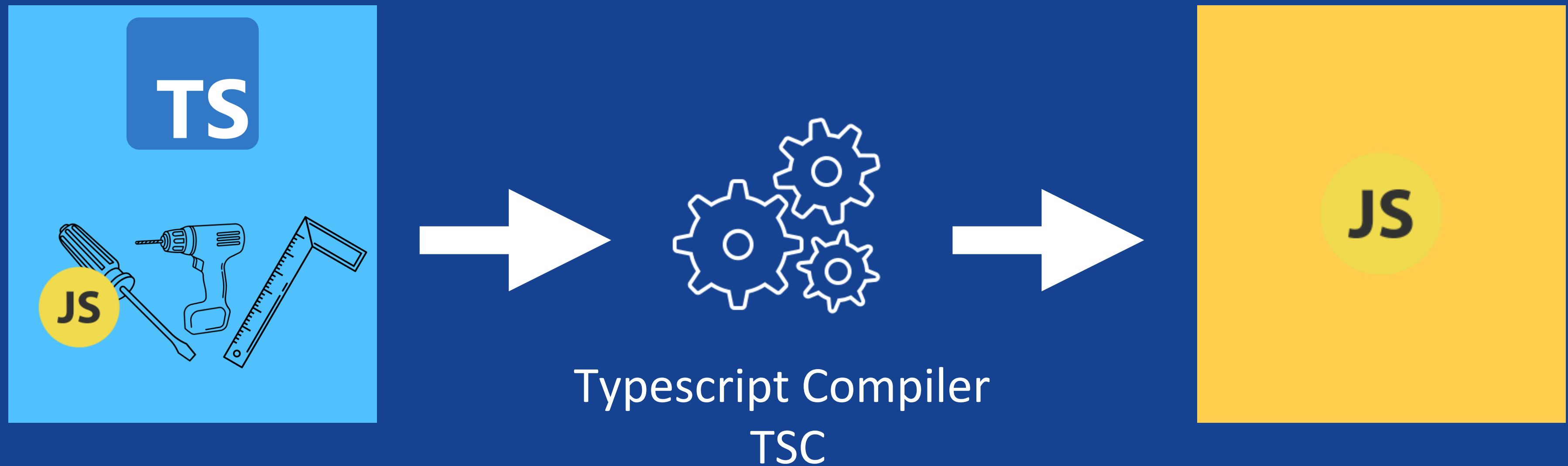
**TS**

```
Let firstName = 'toto';  
firstName = 'tata';
```



**JS**

# Compiler du TypeScript en JavaScript





# TypeScript Compiler

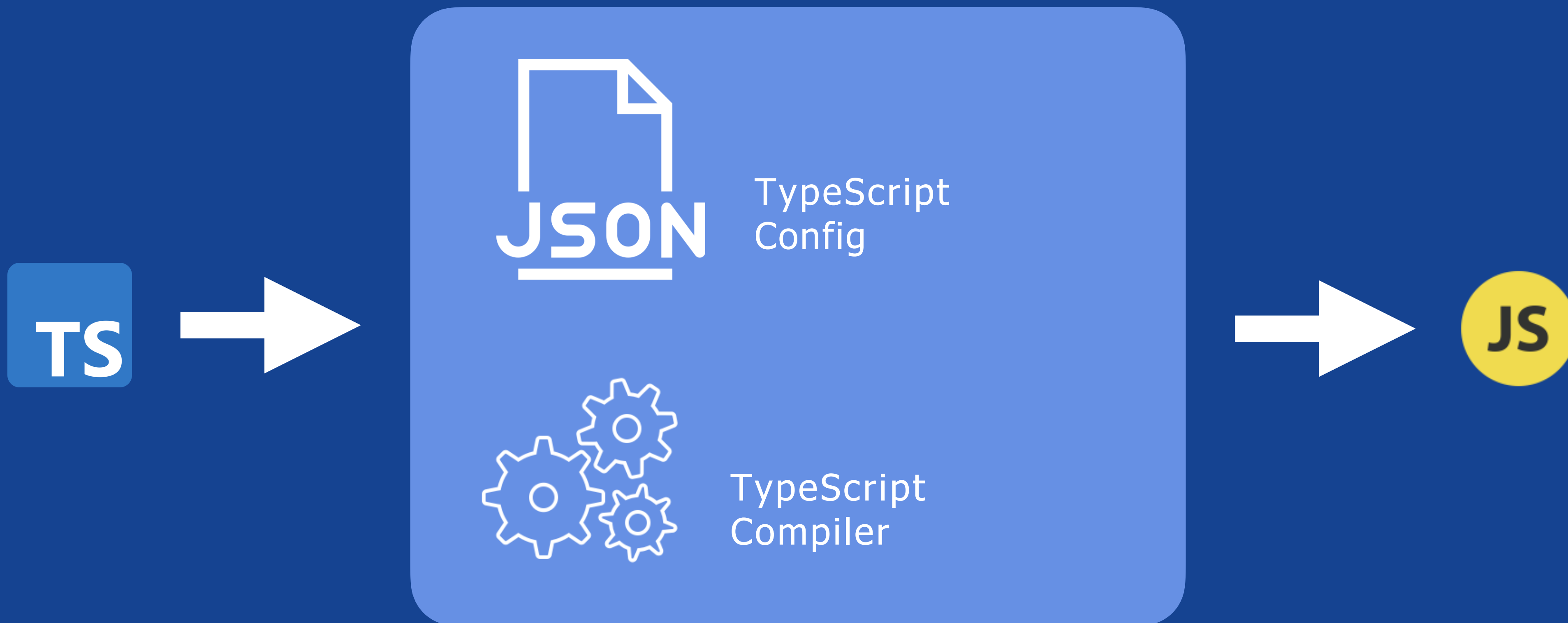


Installer le TypeScript Compiler

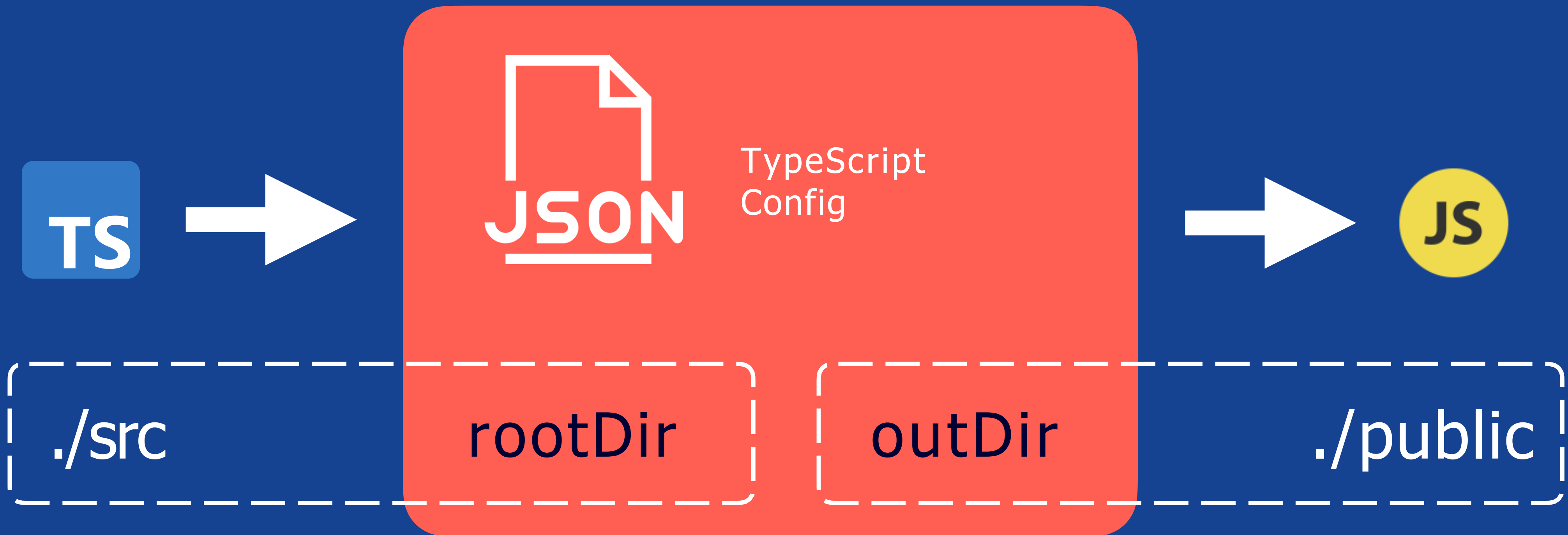
- Compiler les fichiers : **tsc file.ts**
- Compiler les fichiers avec des arguments : **tsc file.ts --out /js**
- Compiler les fichiers automatiquement après validation : **tsc --watch (ou -w)**

Au lieu de préciser les paramètres lors de la compilation, on peut les définir dans un fichier séparé qui va gérer tout cela automatiquement pour nous.

# TypeScript Compiler - `tsconfig.json`



# TypeScript Compiler - `tsconfig.json`



# TypeScript Compiler - `tsconfig.json`

`target: "es5"`

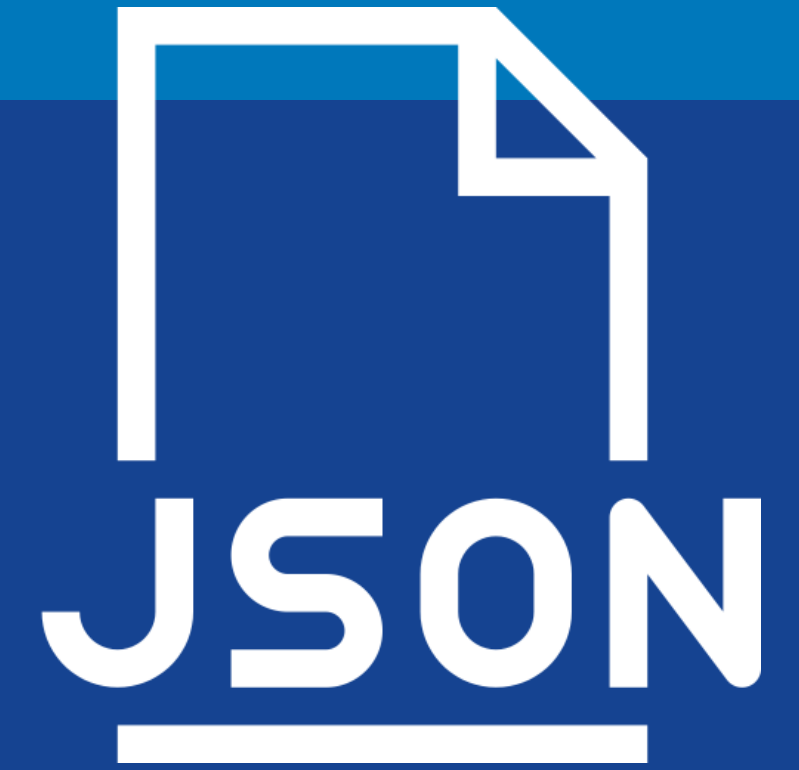
`module`

**lib:** Si désactivé, donc TS accède, par défaut, aux notions globales valides dans la version JS définie dans le « target ». Exemple: le DOM API comme l'objet document et ses méthodes, l'interface Math et ses méthodes, ..etc

Si activé, on aura plus les paramètres par défaut et le compiler ne détectera plus rien. Cela peut être utile si travaillez côté serveur et que vous n'avez pas forcément besoin du DOM types.

Nous allons donc devoir lui définir quelques librairies et les définitions de types spécifiques sous forme de chaînes de caractères dans un array. (Voir la doc)

# TypeScript Compiler - `tsconfig.json`



## Quelques options tsc de base

✓ *allowJS*

✓ *checkJS*

✓ *jsx*

✓ *declaration*

✓ *declarationMap*

✓ *sourceMap*

✓ *removeComments*

✓ *noEmit*

✓ *downLevelIteration*

Le reste des options

# Déclarations de Variables



- TypeScript comprend JavaScript
- Il nous encourage à utiliser les déclarations `let` et `const` (éviter quelques problèmes)



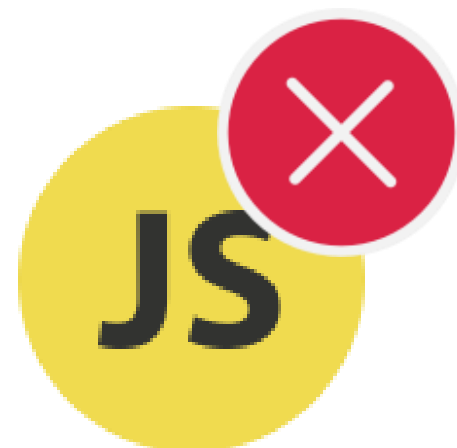
# TypeScript types



TypeScript utilise les types **stricts**  
ce qui n'est pas le cas du JavaScript



Types stricts  
Durant le Dev



Types dynamiques  
Lors de l'exécution

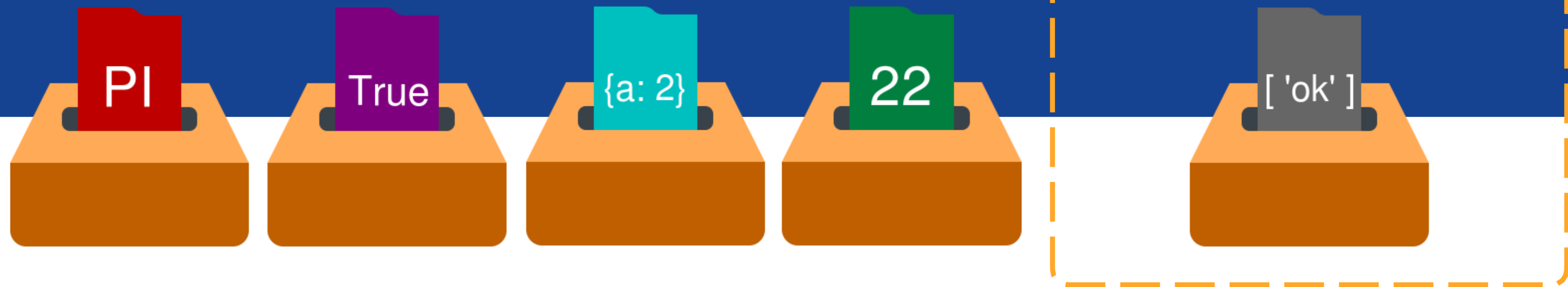


IntelliSense

- Quels sont ces types ?
- Sont-ils obligatoires en TypeScript?

Types par **inférence** Vs types par **Attribution**

# Type Array

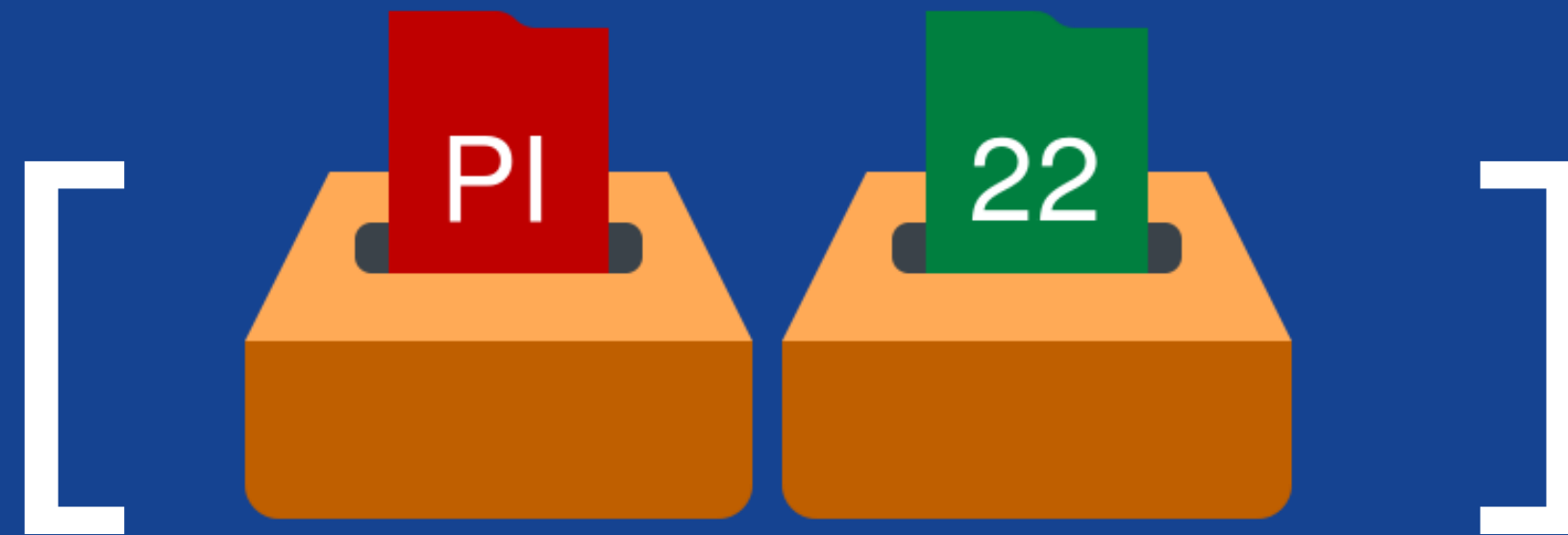
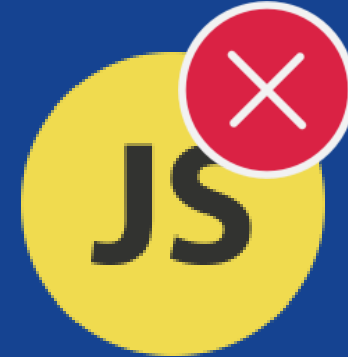


JavaScript et TypeScript permettent de stocker ce qu'on veut dans un Array.

- Number, Boolean, Object, String, Array ...
- Mélanger toutes ces contenus ensemble dans un même Array



# Tuple Type



Définir le nombre d'éléments dans un Array ainsi que leurs types

# Type Object

Type string →

Type number →

```
let car =  
  {  
    color:  
      'red',  
    date: 2020,  
  }  
  speed: 500
```

Un type objet représente un type non primitif, c'est-à-dire tout ce qui n'est pas ( number, string, boolean, symbol, null, or undefined).

Via l'inférence, TypeScript attribut un type aux propriétés d'un objet en fonction des valeurs indiquées.

# Exercices - Typage



# Révisions



Note:

Un Array est également un objet. `{}=[]`

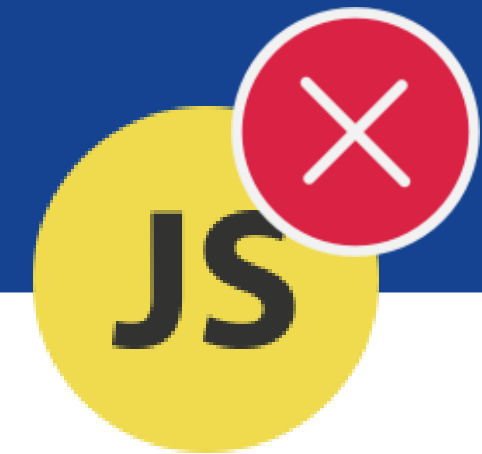
Donc on peut parfaitement assigner une valeur `[ ]` à un object ayant été défini en TypeScript en tant que tel.

# Enums

Enum n'existe pas en JavaScript.

C'est un moyen utilisé dans TypeScript pour nommer des ensembles de valeurs d'une façon numérique.

C'est aussi un moyen qui nous facilite la lecture du code !



```
If ( user.level === 1 )
```

```
If ( user.level === Level.ADMIN )
```

# Any Type

Via « any », le TypeScript n'impose aucun type particulier!

On peut y stocker tous les types



A utiliser avec modération! Vous n'avez aucun contrôle des types et le compilateur TypeScript ne va pas vérifier votre code!

Peut être utile dans certains cas. Exemple: On ne sait pas quel type de data à récupérer d'une API, d'un formulaire, on souhaite modifier le type etc.. (+ vérification, typeof)



# Unknown Type

Via «Unknown », le TypeScript définit un type comme étant inconnu!

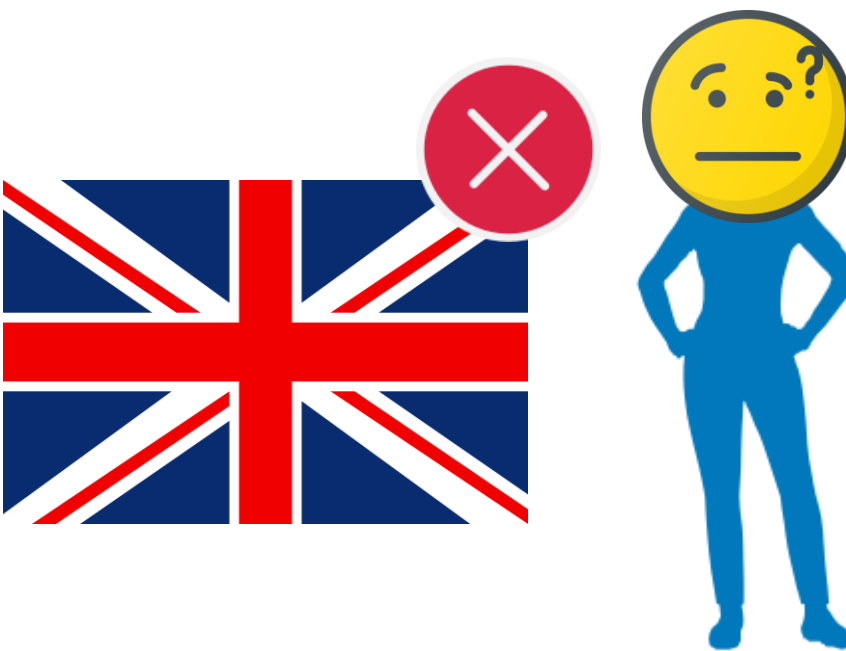
Comme le type «Any », Unknown accepte tous les types



Pour pouvoir utiliser le type Unknown, vous devez d'abord vérifier le type

# Exemple

Variable



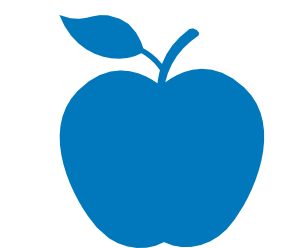
Datas types Objets



Pomme



Livre



Pomme



Livre

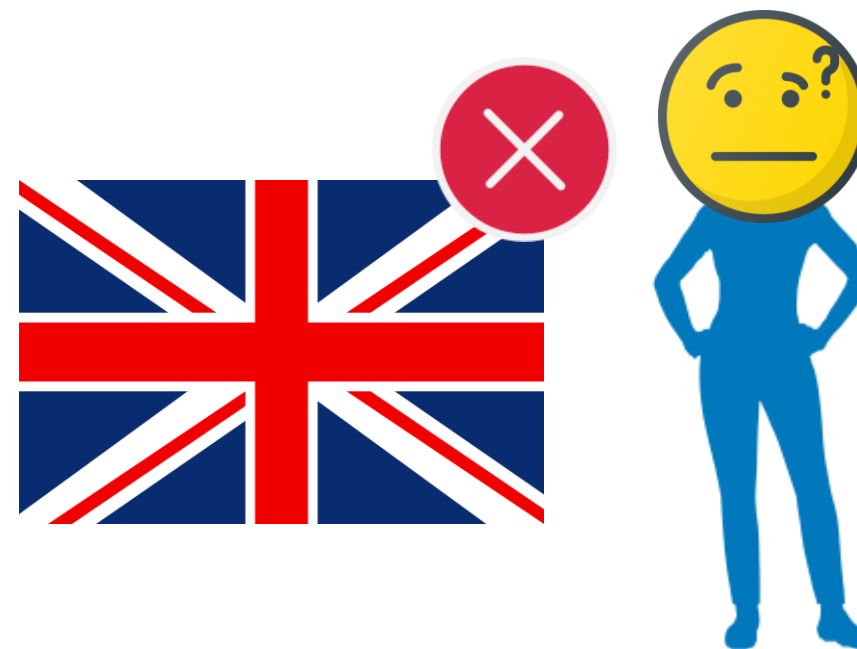
TYPES: COMESTIBLES





# Exemple avec vérification

Variable

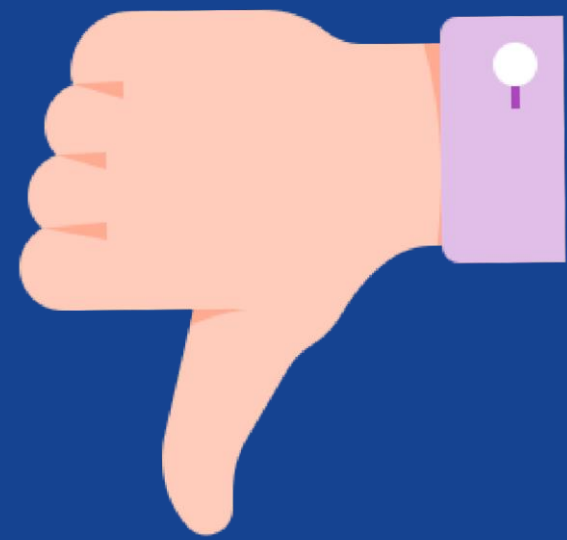


Datas types  
Objets



TYPES: COMESTIBLES





Any

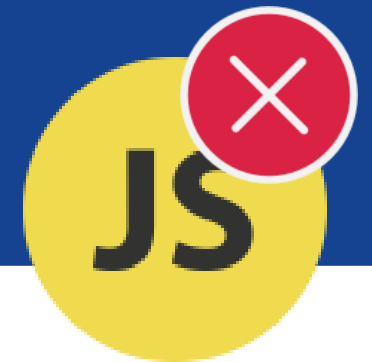
Vs



Unknown

+ Vérification

# Function & Void Type

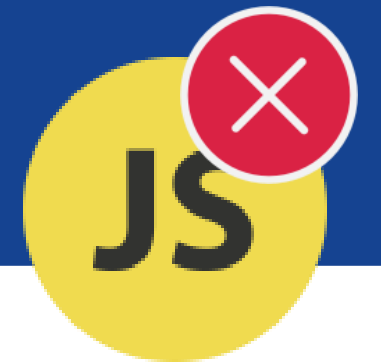


Comme les Enum et les tuple, « void » aussi n'existe pas en JavaScript

De ce fait:

- Une fonction qui retourne une valeur définit le type de la valeur retournée via (via inférence ou Attribution).
- Une fonction qui ne retourne rien définit également un type « void » pour Undefined

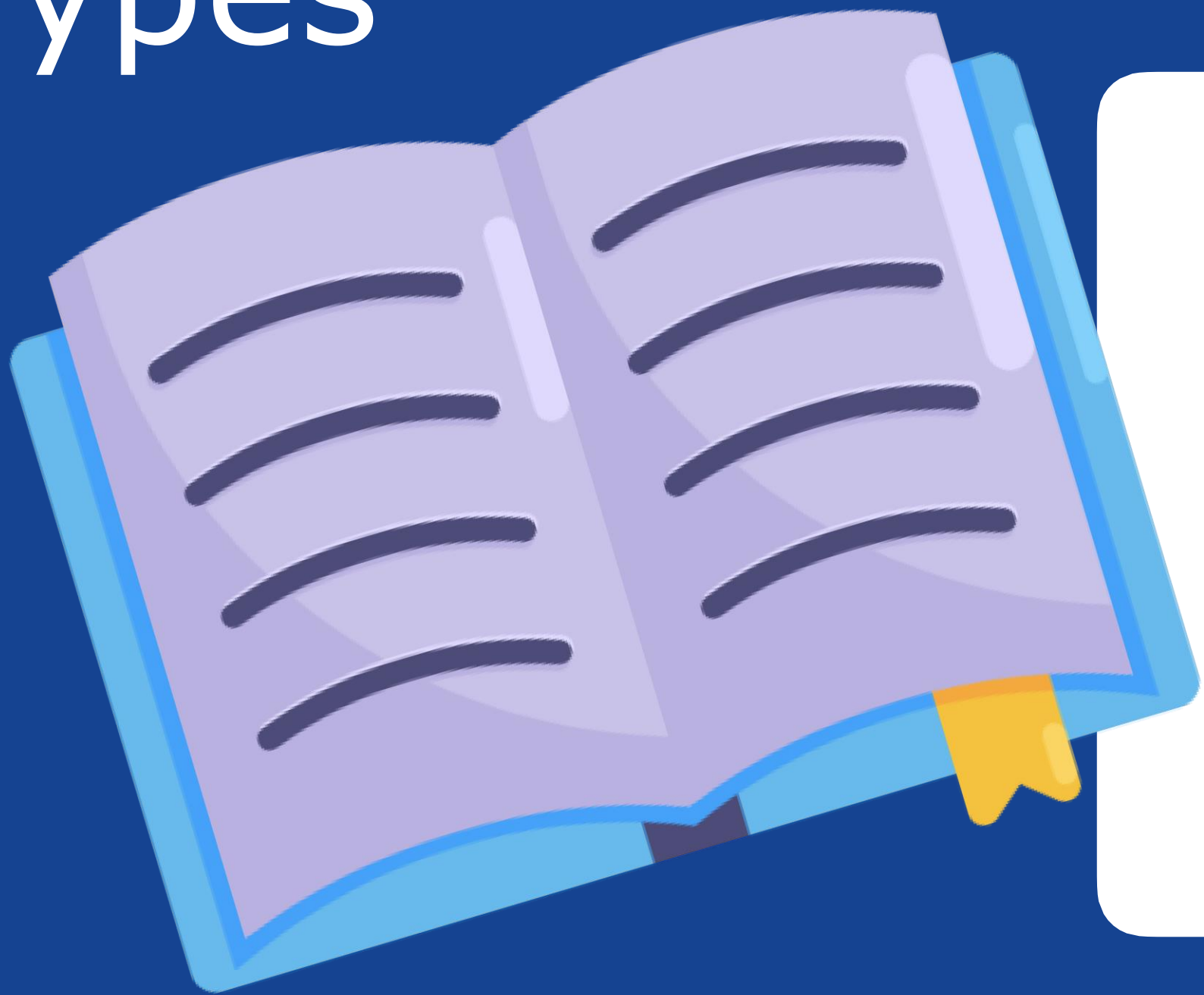
# Function Types



Type Function générique: (Une seule contrainte: le type doit être une fonction)

Types bien spécifiques: Non seulement c'est une fonction mais celle-ci doit être très explicite : `(param: type) => return type`

# Function Types



- Paramètres facultatifs
- Paramètres par défaut

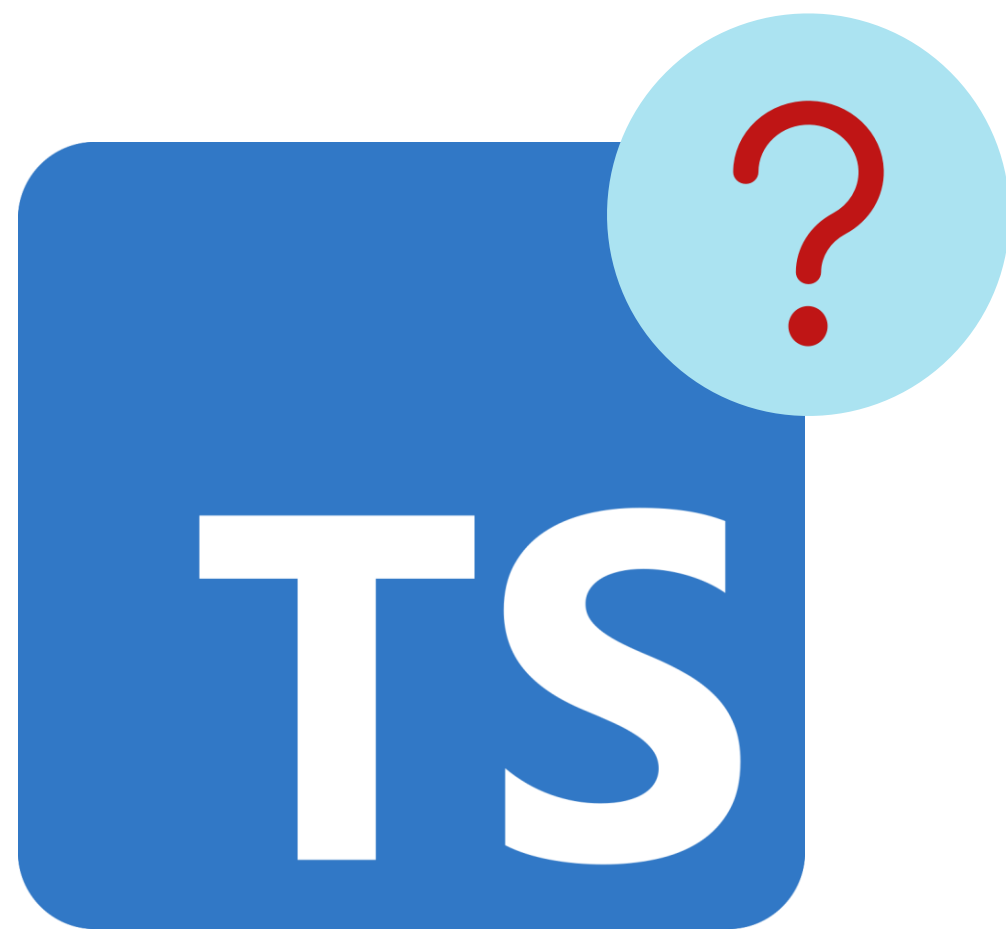
# Null



# Undefined



# Type Assertions (Affirmatif)



C'est un Number !!!!  
Fais moi confiance, je  
sais ce que je fais :-)

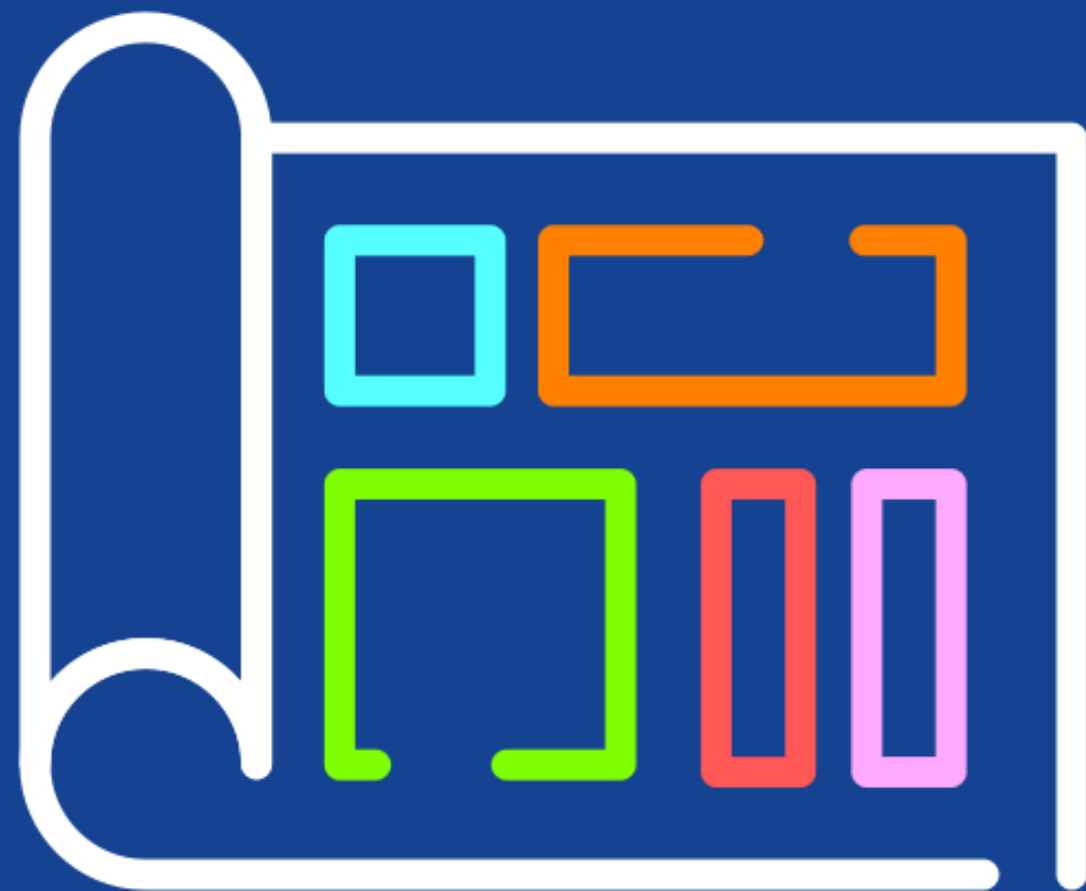
# Type Assertions (Affirmatif)

An illustration of a dark blue rectangular form. At the top center is a green circle with a white checkmark. Below it are two horizontal white input fields. At the bottom is a red rounded rectangle with the white text 'VALIDER' in the center.



# Les Classes

Programmation Orientée Objets en TS



Class -  
Plan



Instanciation



Objet

# Les Classes

: **Invoice**

Respecter les caractéristiques d'un objet

: **Invoice[]**

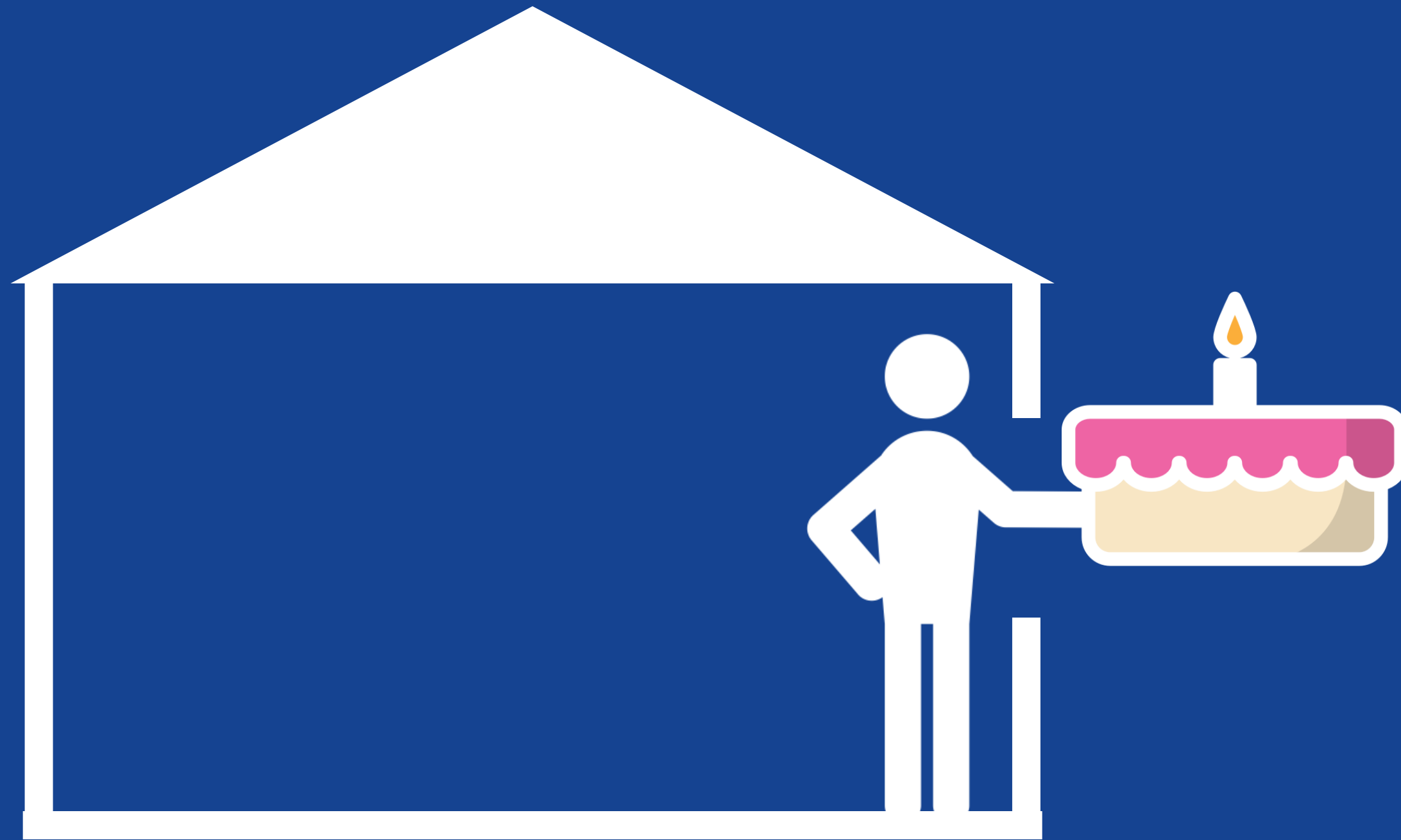
Un array d'objets « Invoice »



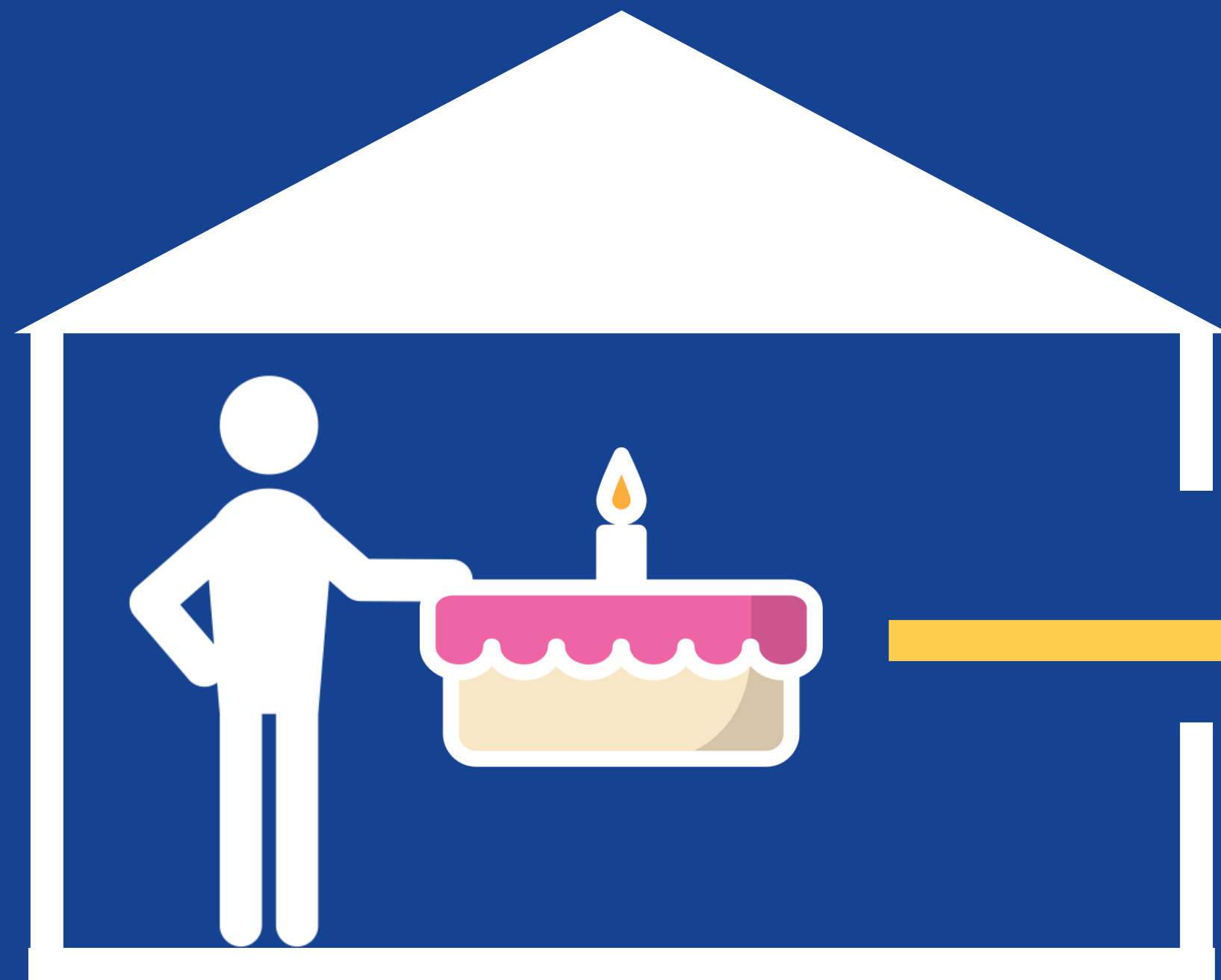
**Public** (accessible à l'extérieur)



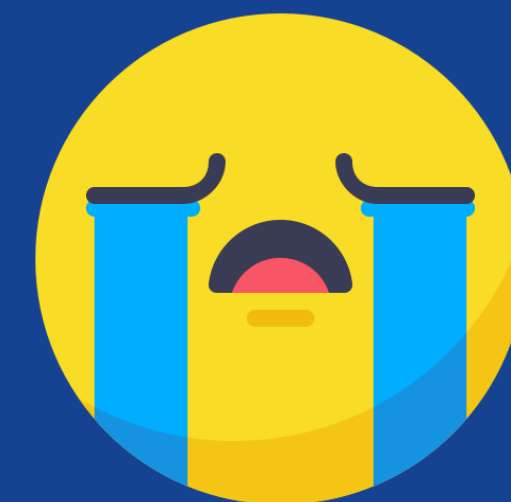
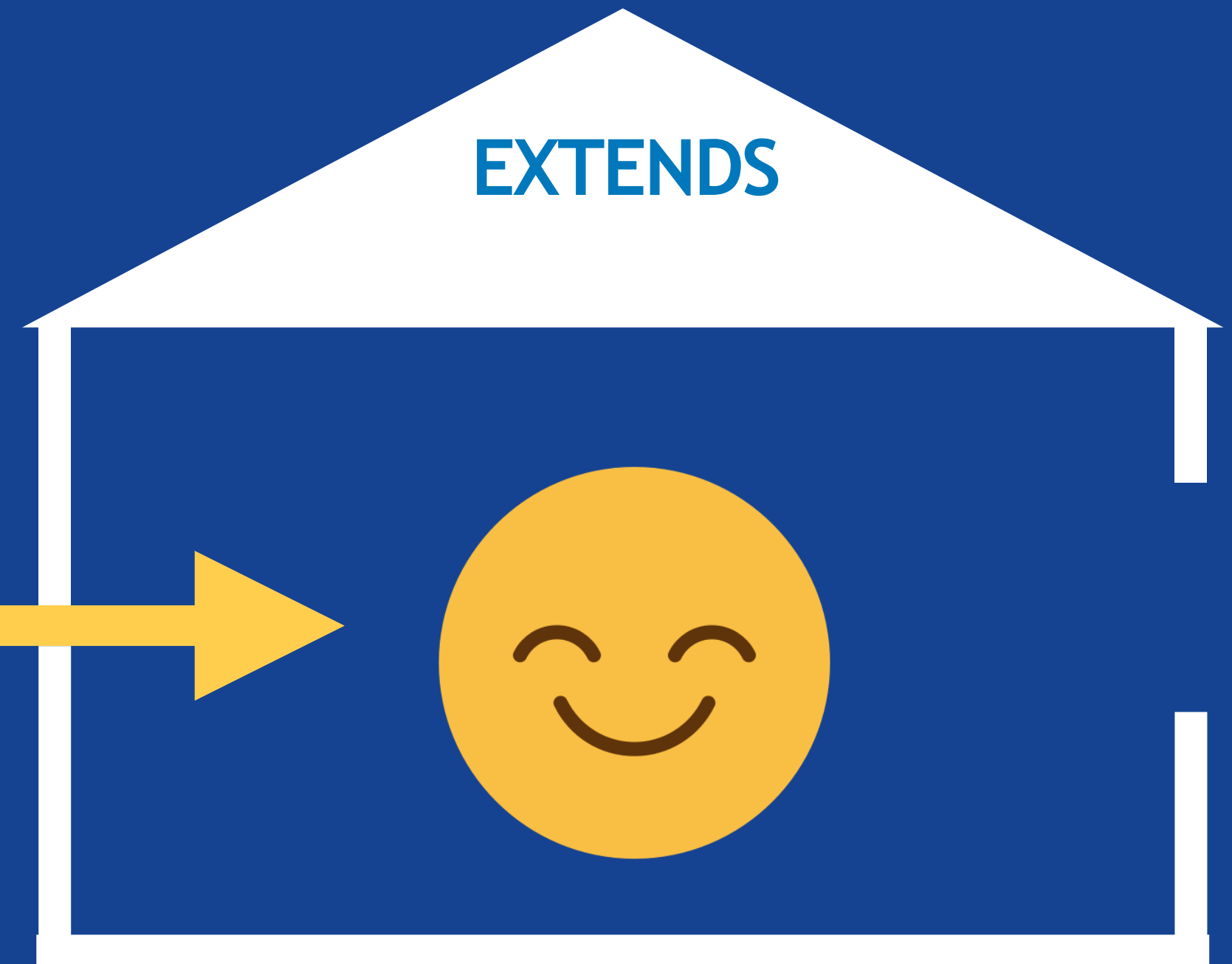
**Private** (Accessible seulement dans la classe)



Getters & Setters



Protected





**readonly** ( Peut être modifié que via le constructor)

# Les Classes

## Héritage et Polymorphisme en TypeScript

Personne



Propriétés

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

Méthode

- Parler
- Rire ..etc

Héritage





# Les Classes

## Access Modifiers

Personne



Propriétés

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

Méthode

- Parler - Rire ..etc

Héritage



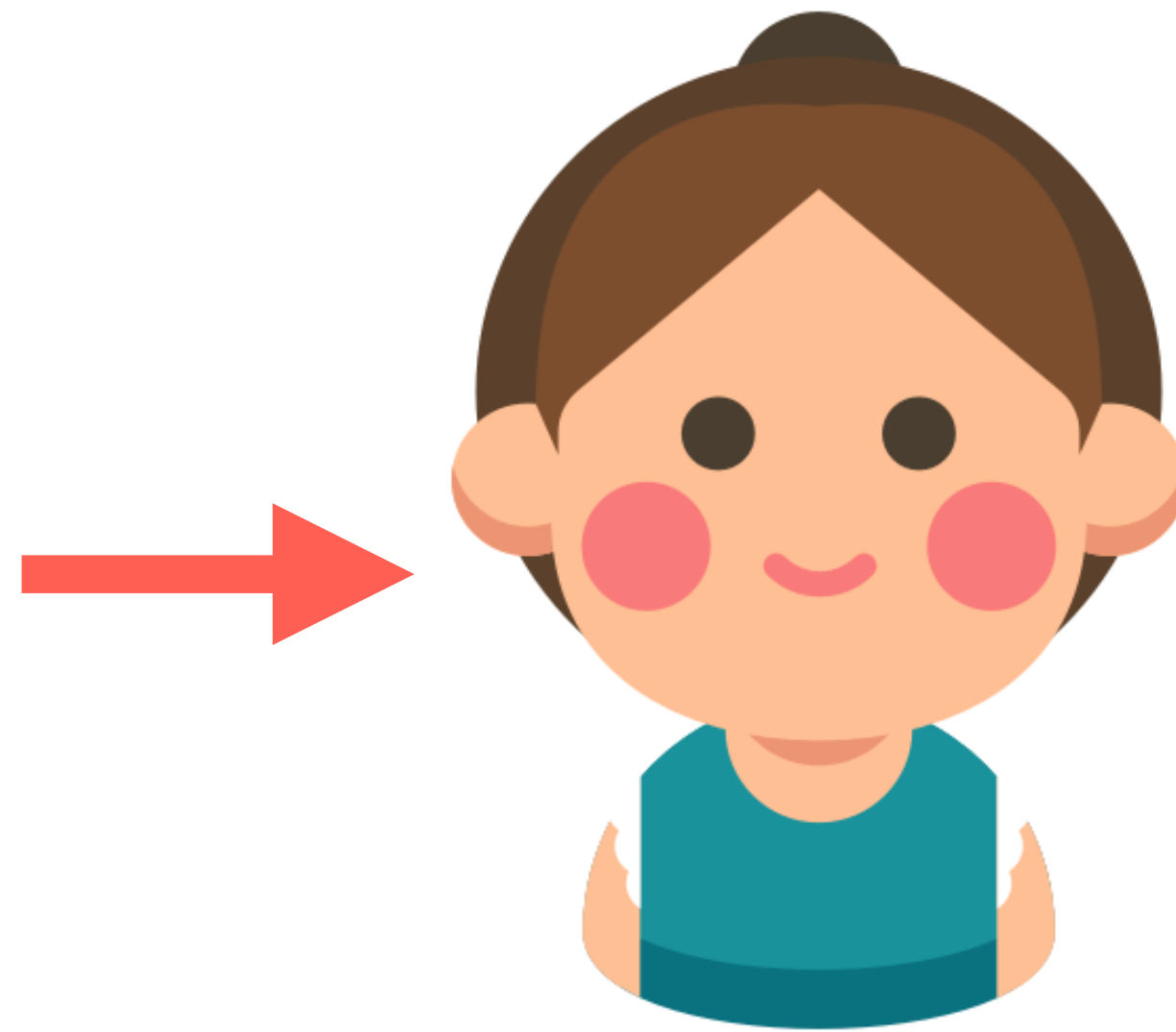
# Les Classes (static)

class Person

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

- Parler - Rire ..etc

Object



parler( )

rire( )

# TP - Annuaire



# Interfaces

class Mother

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

- Parler    - Rire ..etc

Personne: Mother



Interface type

Définir les contraintes d'un objet  
propriétés méthodes

# Interfaces avec les classes

class Mother implements « Interface »

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

- Parler - Rire ..etc

Interface

Définir les contraintes d'un objet  
propriétés méthodes



En implémentant une Interface à une classe, on s'assure que l'objet instancié est conforme aux spécificités définies dans l'interface.

Exemple: Si l'interface exige une méthode speak(), l'objet instancié d'une classe qui implémente cette interface doit pouvoir invoquer cette méthode. Donc il pourra parler..



# Union Type

string | number | boolean



# Exercise – Interface Client



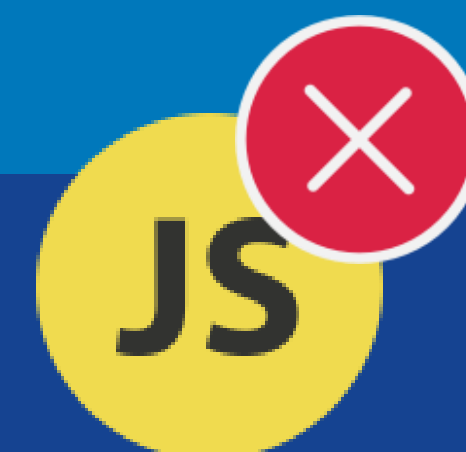


# Intersection Type

Person & number & boolean







# Type Aliases

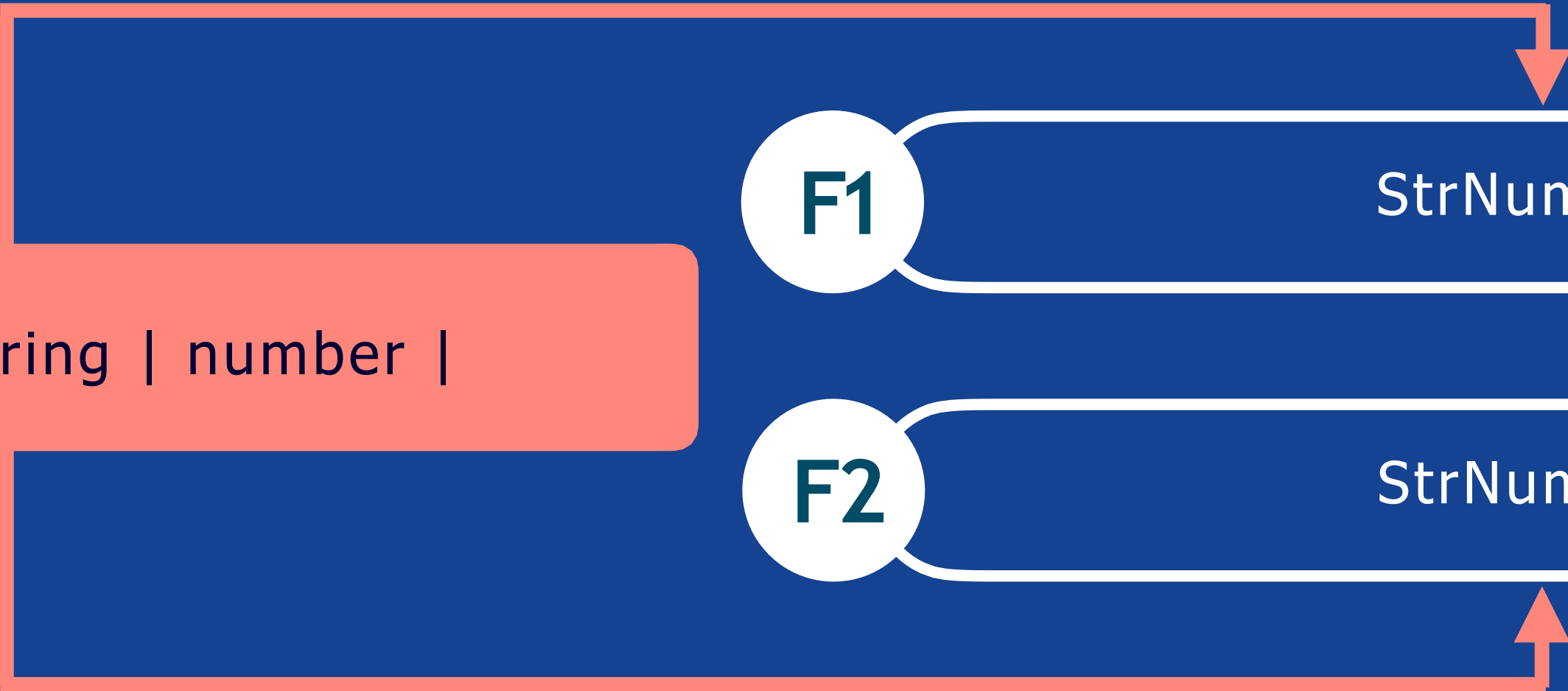
```
type StrNumBoo = string | number | boolean
```

F1

StrNumBoo

F2

StrNumBoo



# Literal Type (Littéral - exact)



## Union Type

**F1**

arg : string | number

'Hello'



200



true



## Literal Type

**F2**

arg : 'toto' | 'number2'

'Hello'



200



true



'toto'



'number2'



# Conditional Types

Un type conditionnel sélectionne, via un opérateur conditionnel, l'un des deux types possibles en fonction d'une condition exprimée sous la forme d'un test de relation de type

 extends  ? "Fruit" : "légume"



# Mapped Types

Les mapped Types nous permettent de créer un nouveau type on nous basant sur des types déjà existants



# Exercices

# Les Classes abstraites

(abstract)

abstract class  
Person

- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

Speak()

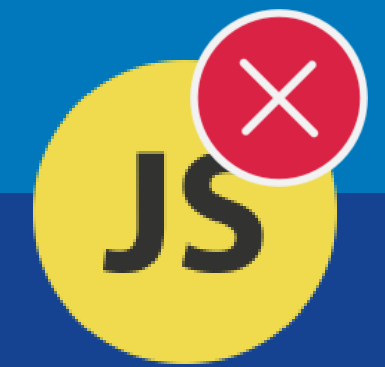
abstract walk() signature

class Mother extends Person


walk() est obligatoires

Object





# Generics

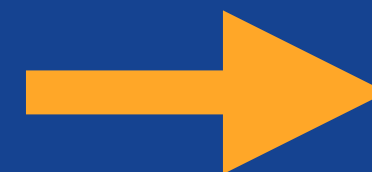
 Type Any

Accepte tous les types mais ne les détecte pas

 Type *Generic*

- Détecter et gérer différents types
- Étant pas spécifique à un type, il peut donc être réutilisable

```
function getData( arg: any ) {  
    return arg  
}
```



```
function getData<T>( arg: T ): T {  
    return arg  
}
```



- Generics avec Interfaces
- Generics avec les classes
- Generic Utility Types



# TP - La Pile



# Omit<Type, Keys>

```
interface Todo {  
    title: string;  
    description: string  
    author: string  
}
```

```
Omit<Todo, "title" | "author" > = {  
    description: string;  
}
```

Le Generic Utility Type "Omit" est le contraire de Pick. Il permet de construire un nouveau Type en sélectionnant les propriétés Keys dans le Type utilisé (Interface) et en retirants certaines autres clés.

# Exclude<Type, ExcludedUnion>

Le Generic Utility Type "Exclude" permet de construire un nouveau Union Type en excluant des types définis dans un autre Union Type.

```
type A = string | string[ ] | boolean
```

```
Exclude<A, boolean> = string | string[ ]
```

# Extract<Type, Union>

Le Generic Utility Type "Extract" est l'opposé de « Exclude ». Il permet de construire un nouveau type en extrayant des « Union membres » qui peuvent être affectés à « Union ».

En gros, on va extraire ce qu'on souhaite utiliser dans notre nouveau type.

```
type A = string | string[ ] | boolean
```

```
Extract<A, boolean> = boolean
```

# Decorators

Les decorators sont actuellement en phase expérimentale et peuvent subir des changements dans les prochaines versions JavaScript. Néanmoins, je les aborde dans cette formation car c'est un pattern assez important adopté largement dans les récents frameworks tel que Angular

C'est simplement un moyen de mettre un morceau de code dans une fonction dans le but d'étendre les fonctionnalités d'une classe par exemple.

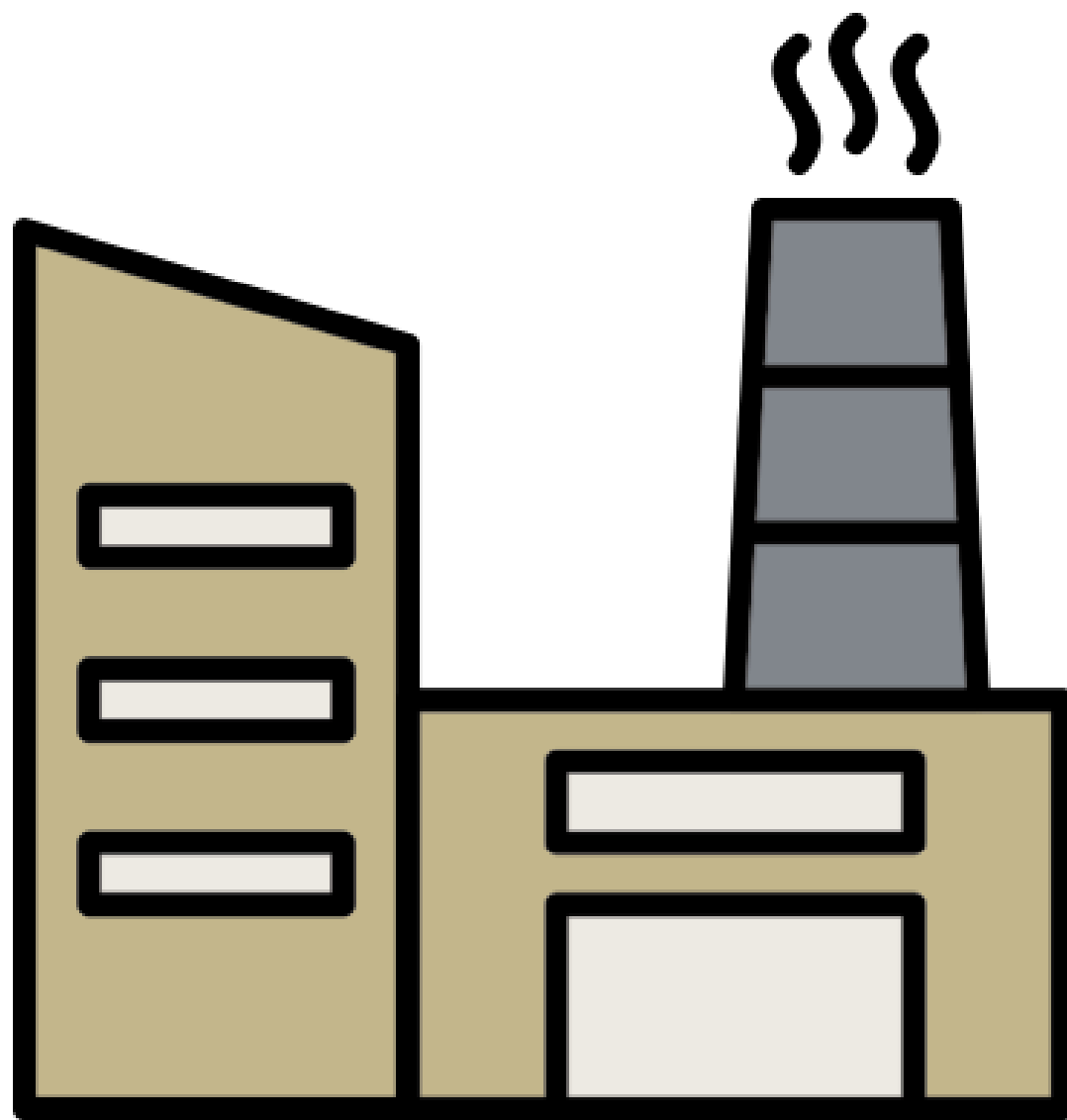
✓ *"experimentalDecorators": true*

✓ *"target": "es6"*



# Decorator Factories

Fonction qui génèrent les decorators



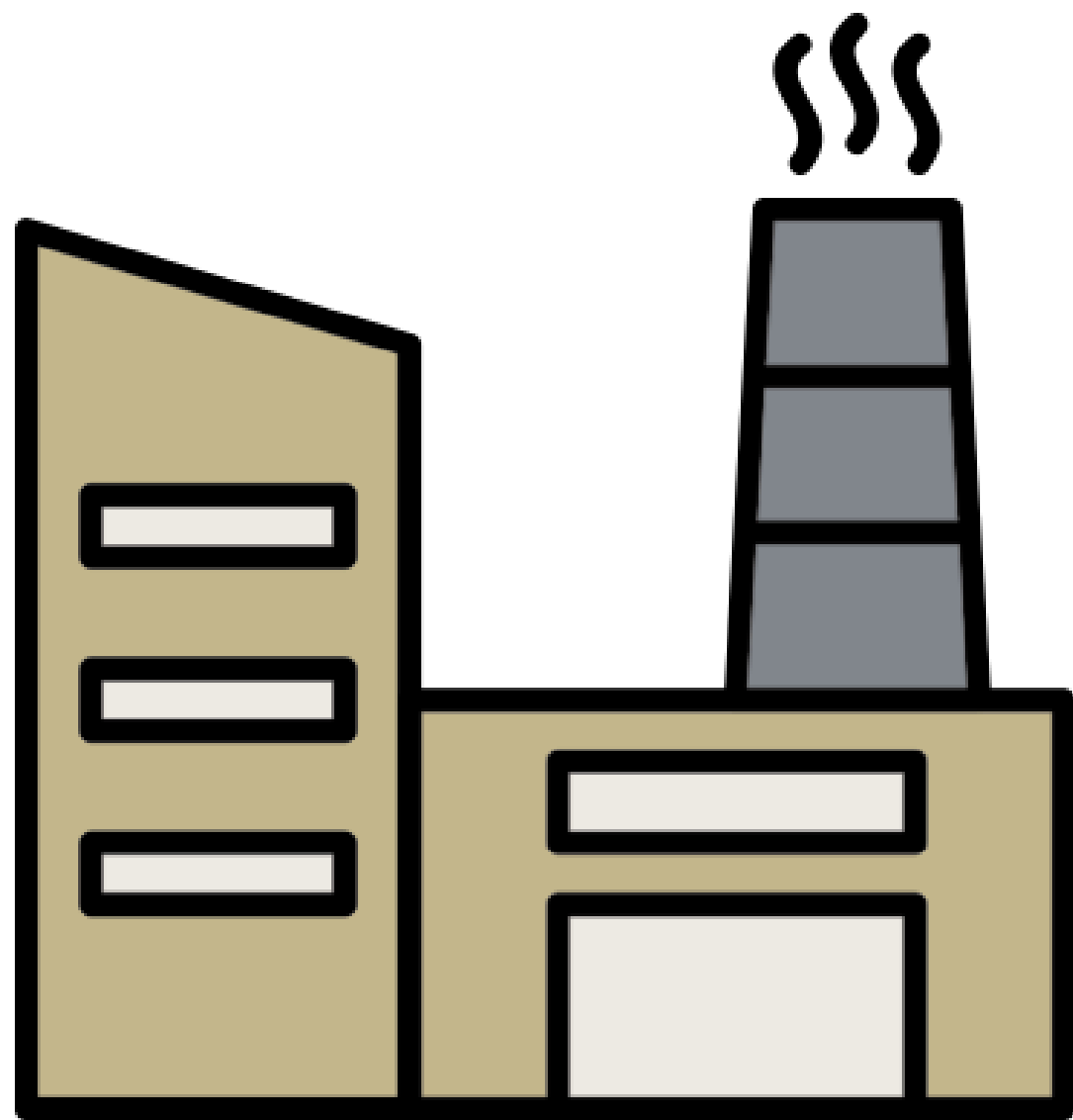
Function( param )



Anonymous Function

# Decorator Factories

Injecter du contenu dans le DOM à la manière Angular via un decorator `@component`



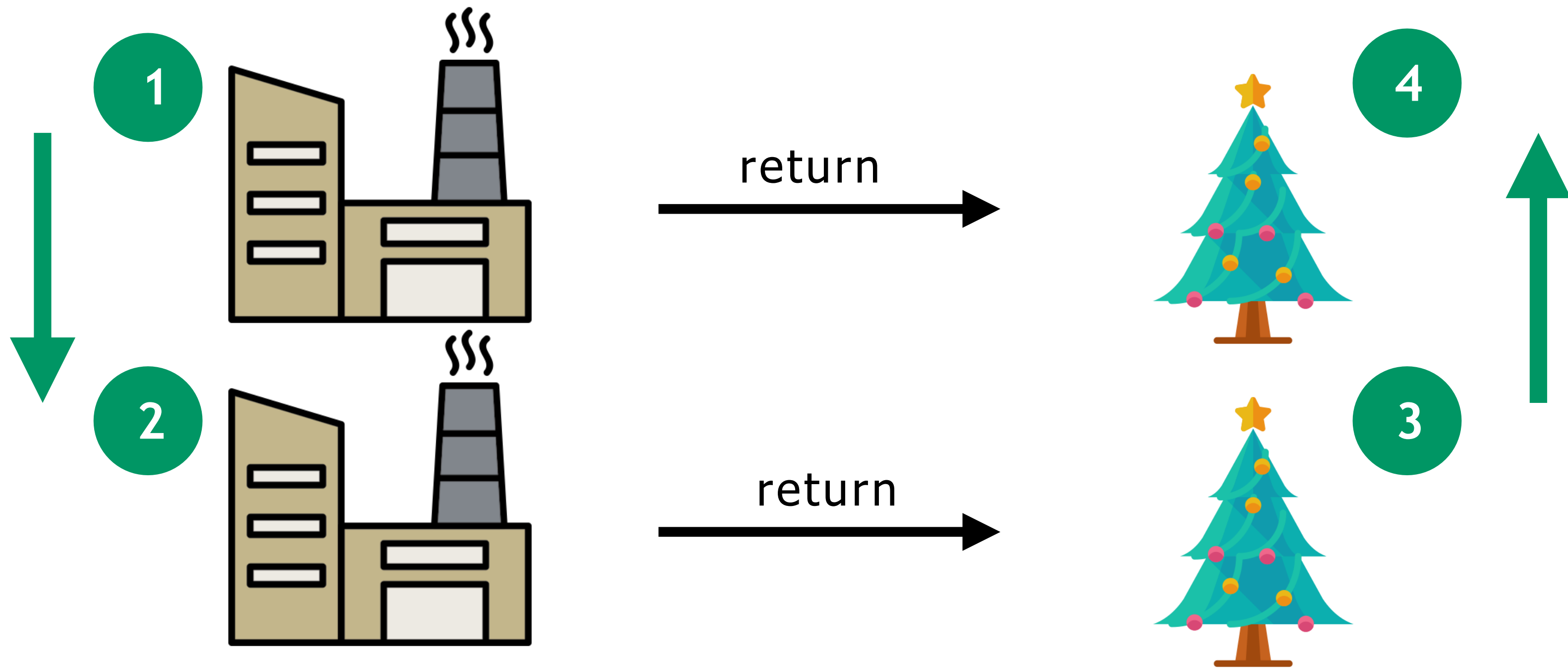
Function( param

)



Anonymous Function

# Multiple Decorator Factories





# Class Decorator

## ✓ Propriétés

Accessors ( getters - setters )

Méthodes

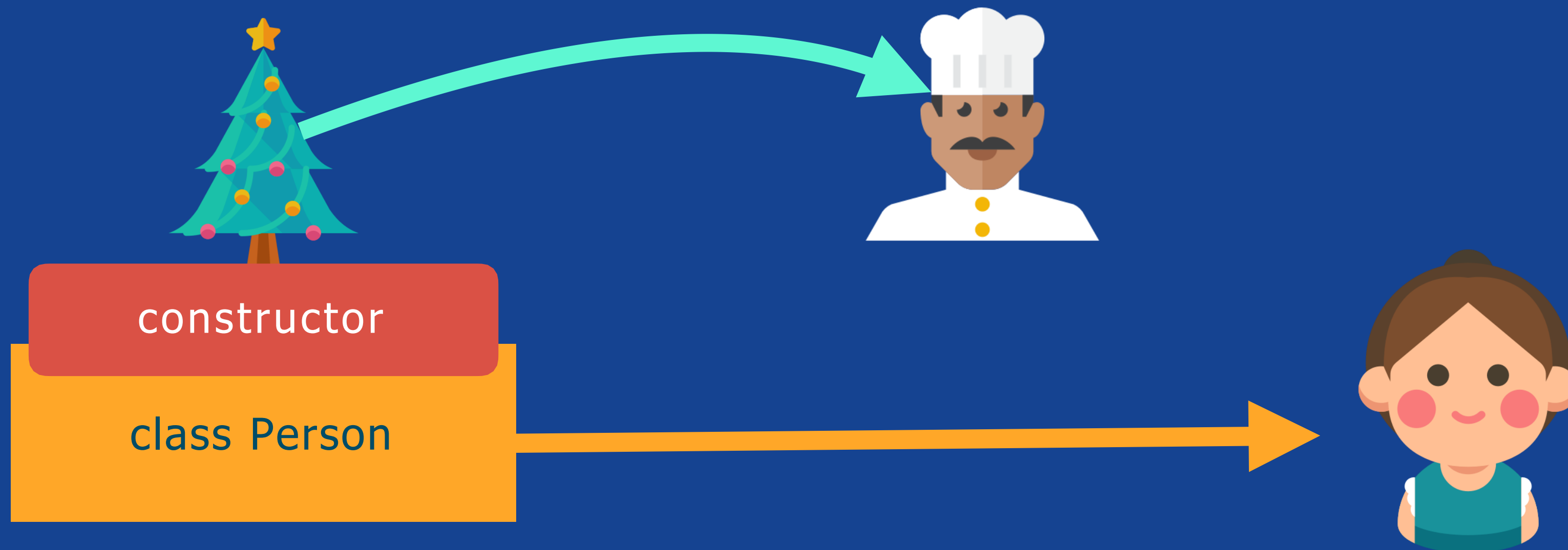
Paramètres

# Class Decorator

- ✓ Propriétés
- ✓ Accessors ( getters - setters )
- ✓ Méthodes
- ✓ Paramètres

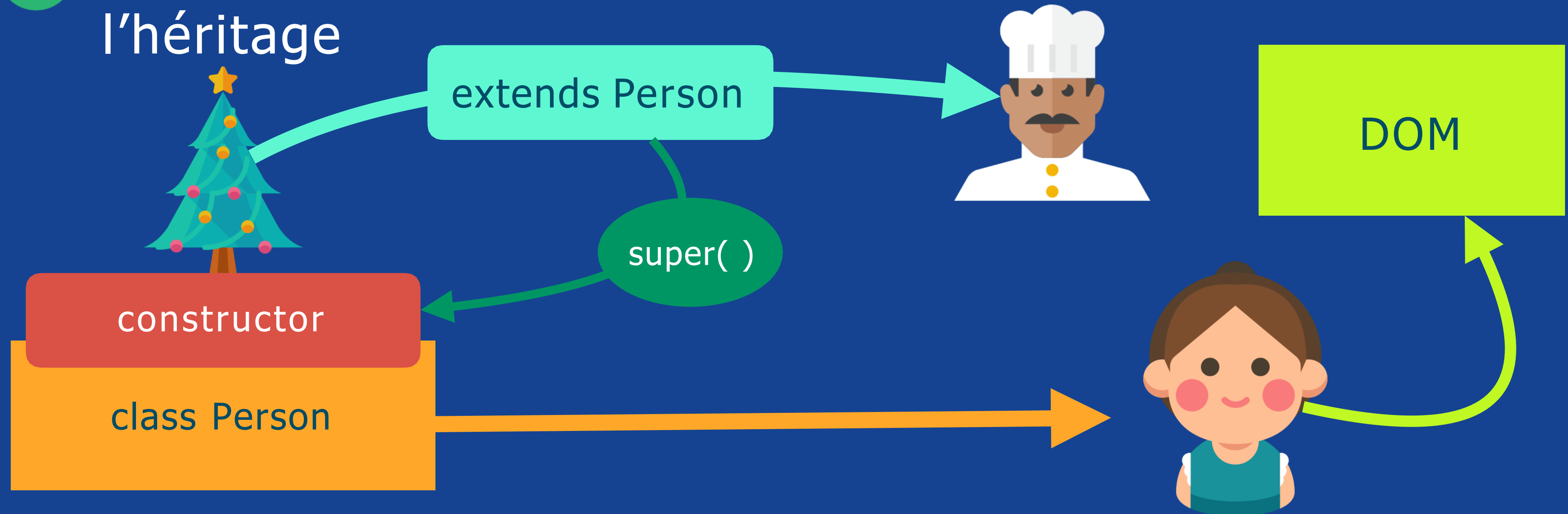
# Class Decorator

- ✓ class décorateur peut réécrire la fonction constructor



# Class Decorator

- ✓ Réécrire la fonction constructor via l'héritage



# TP - Caisse enregistreuse

