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.

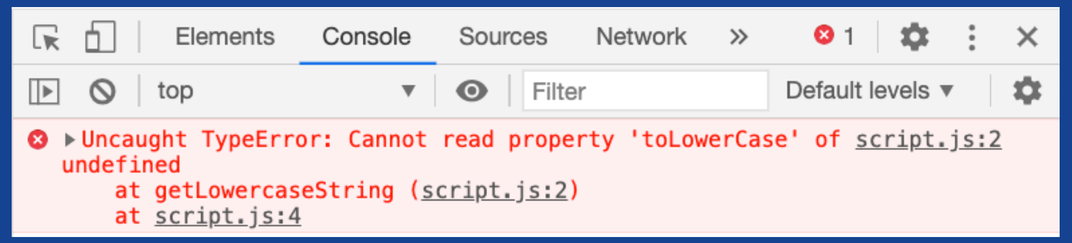
?

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

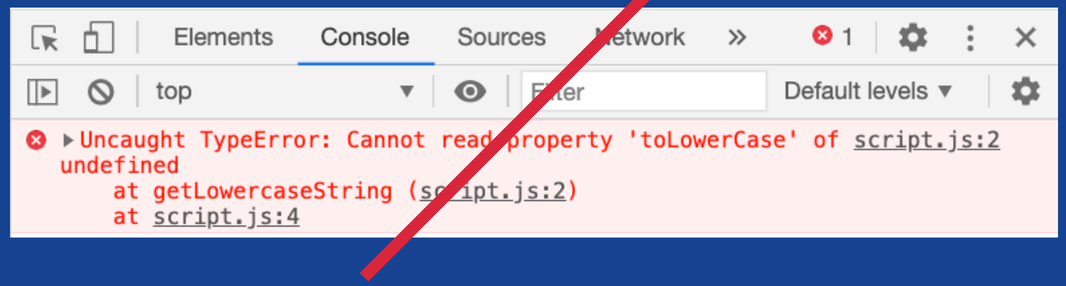
```
getLowercaseString( );
```

Cela va nous causer un problème!

On constatera l'erreur une fois le code lancé !!!!



Solution TypeScript




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

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

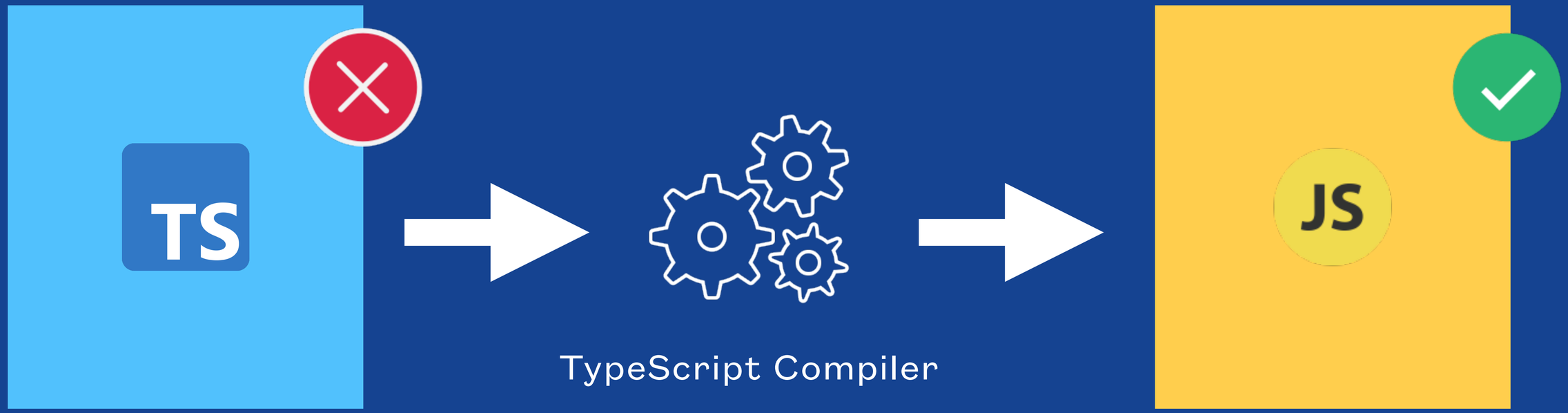
```
getLowercaseString( 200 );
```

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 ?

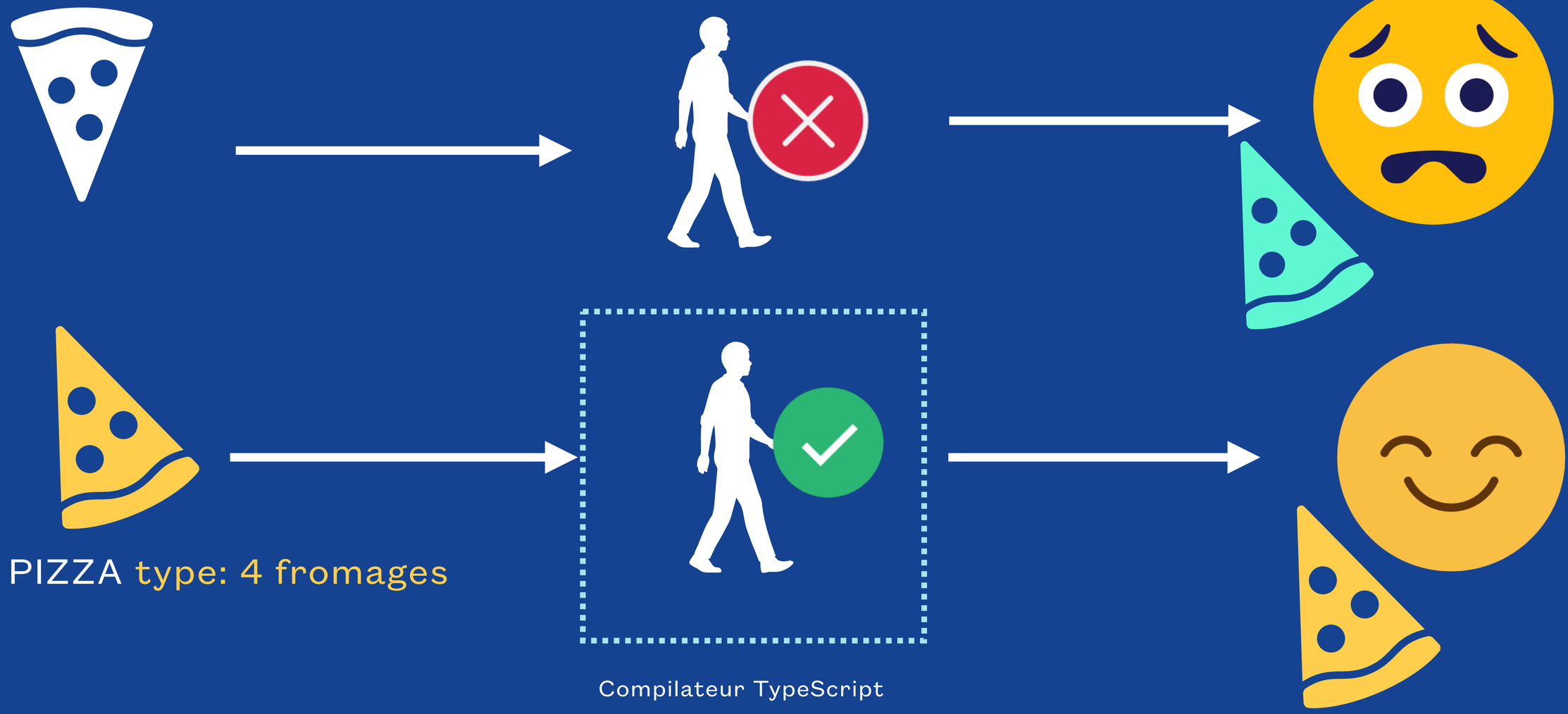
Le code TypeScript n’est pas reconnu par les navigateurs web, il faut donc le compiler en langage JavaScript



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)

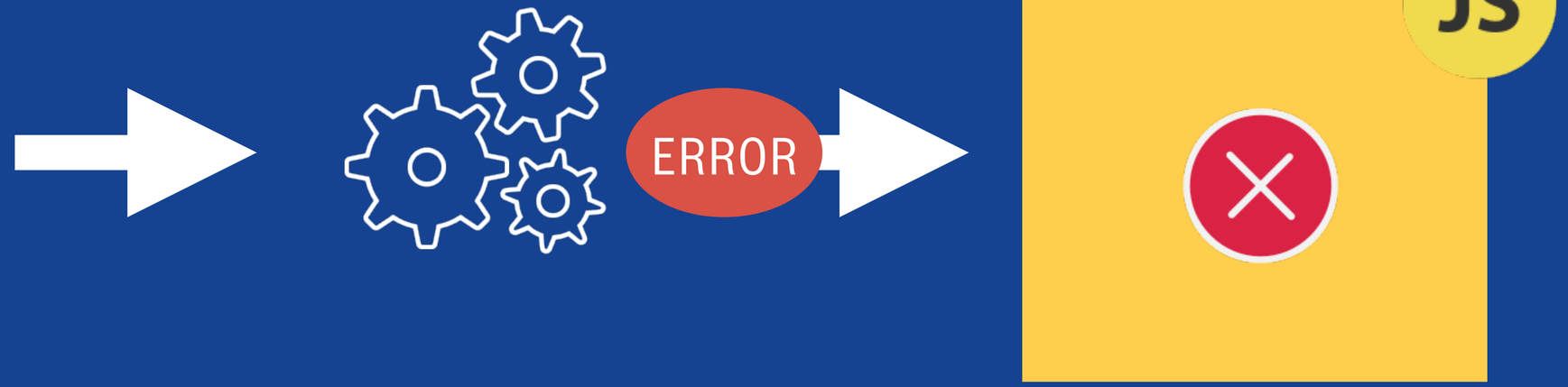
- Vérifie que notre code fonctionne bien (constater les erreurs durant le dev)

- 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

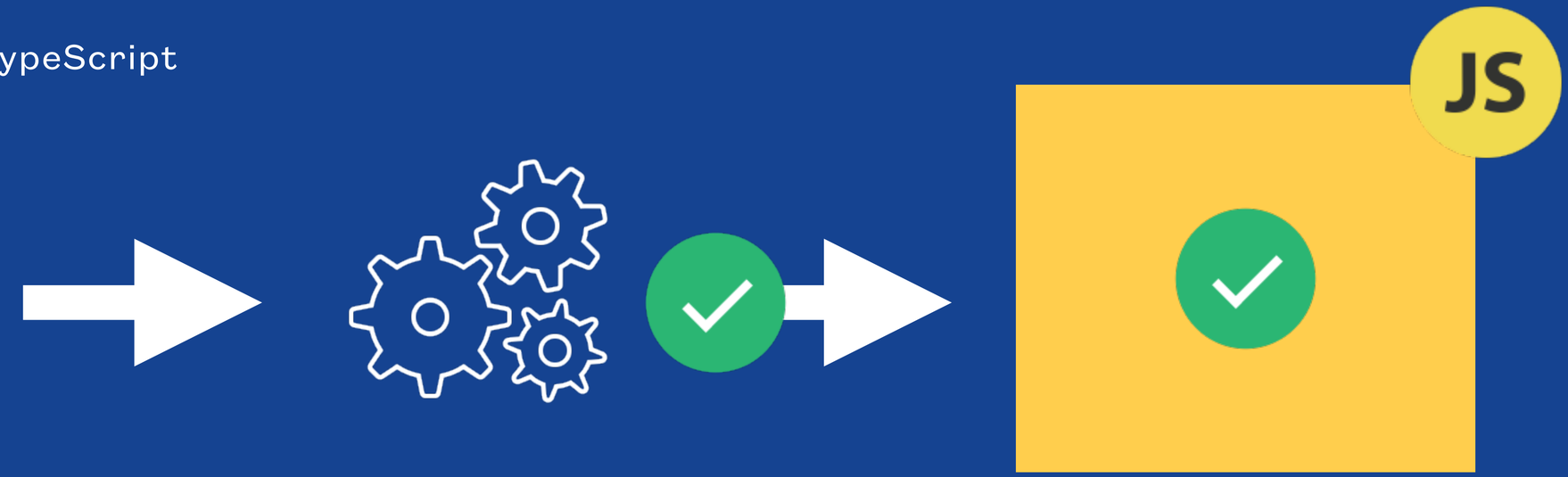
```
Let firstName = ‘toto’;
firstName = 12345;
```



Types stricts en TypeScript

TS

```
Let firstName = ‘toto’;
firstName = ‘tata’;
```



Outils

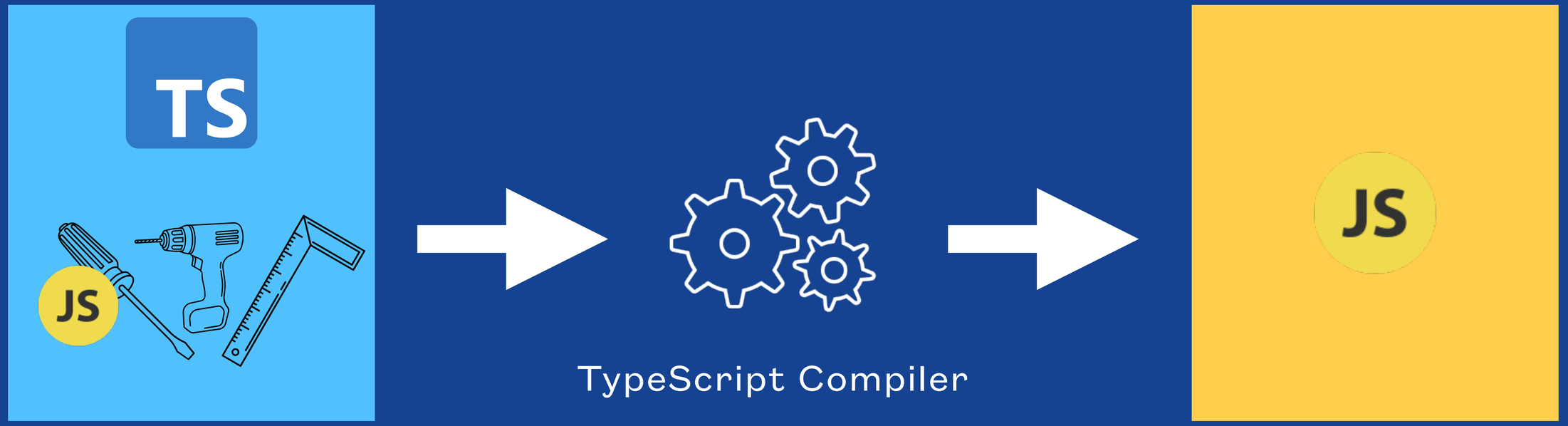
Accès à Internet (La DOC + package NPM)

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

Éditeur de code - Visual Studio Code (Intellisense)

On aura besoin de NPM pour installer TypeScript

Compiler du TypeScript en JavaScript



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
```

Déclarations de Variables

- TypeScript comprend JavaScript ES6
- 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

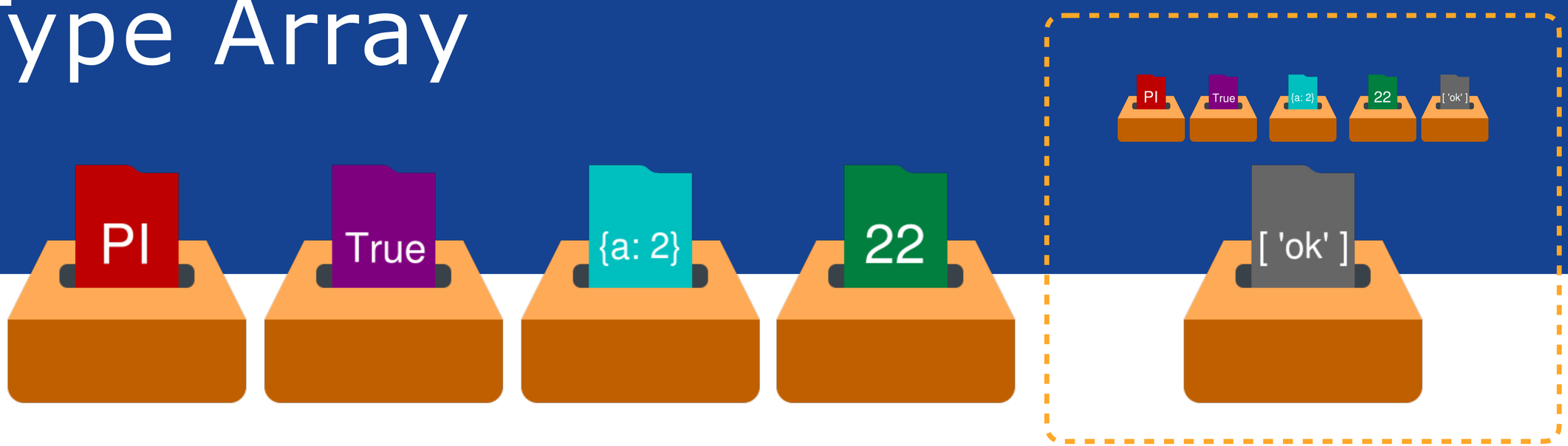
IntelliSense

- Quels sont ces types ?

- Sont-ils obligatoires en TypeScript?

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

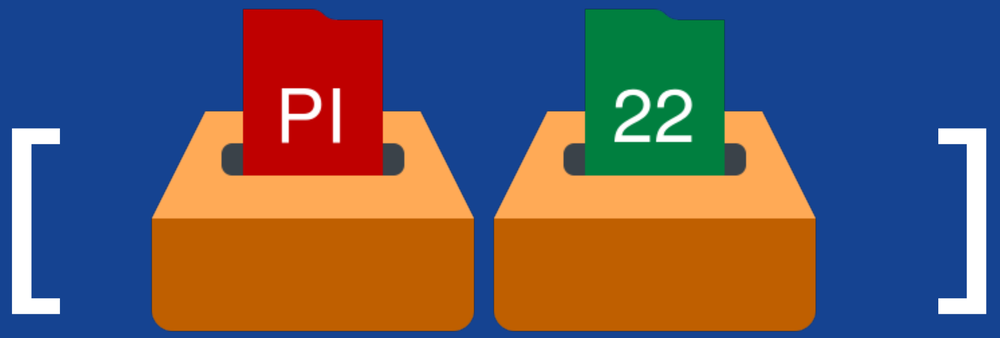
Type Array



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

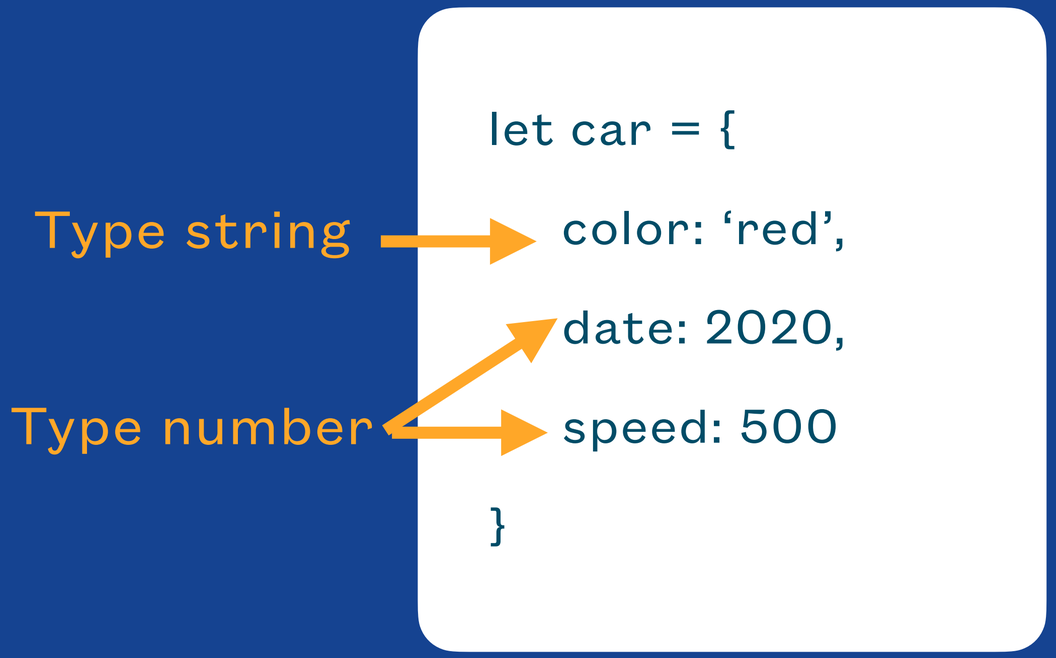
- Number, Boolean, Object, String, Array ...
- Mélanger toutes ces contenus ensemble dans un même Array

Tuple Type



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

Type Object



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.

Révisions

1 - créer une variable « names » qui n’accepte qu’un type String

2 - créer une variable « speed » qui n’accepte que les numbers et l’initialiser avec la valeur 25

3 - créer une variable « isLoading » qui n’accepte que les types booleans

4 - créer une variable « age » qui n’accepte que les types number ou string

Révisions

5 - créer une variable « colors » qui est un Array de strings. Ensuite, effectuer un push de la valeur « red » + console.log de la variable « colors »

6 - créer une variable « infos » qui ne peut accepter qu'un Array de strings ou numbers ou booleans. Ensuite, faire un push du string « dupont » + true + 40

Révisions

7 - créer une variable « number » qui ne peut être qu’un objet.

- Via l’indicateur de valeur « object », sans définir le contenu de l’objet. Ensuite, avec les propriétés suivantes:

    firstName: « Dupont »

    age: 20

    isLoggedIn: true

- Afficher la valeur age via console.log

Révisions

8 - créer une variable « number » qui ne peut être qu’un objet.

- Préciser le type objet via attribution (assignation) tout en indiquant les propriétés suivantes:

  firstName: « Dupont »

  age: 20

  isLoggedIn: true

- Afficher la valeur age via console.log

Révisions

9 - créer une variable « number » qui ne peut être qu'un objet.

- Laisser TypeScript définir le type objet via inférence tout en indiquant les propriétés suivantes:

  firstName: « Dupont »

  age: 20

  isLoggedIn: true

- Afficher la valeur age via console.log

Icons from https://www.iconfinder.com/pancaza

Révisions

10: 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.

Révisions

11 - Créer une variable « infos » qui ne peut être qu’un Tuple.

de seulement 2 valeurs. string et number dans cet ordre-là!

Question: Peut-on changer la valeur infos[1] = « salut » ?

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)

Icon from: https://www.iconfinder.com/iconic_hub

Unknown Type

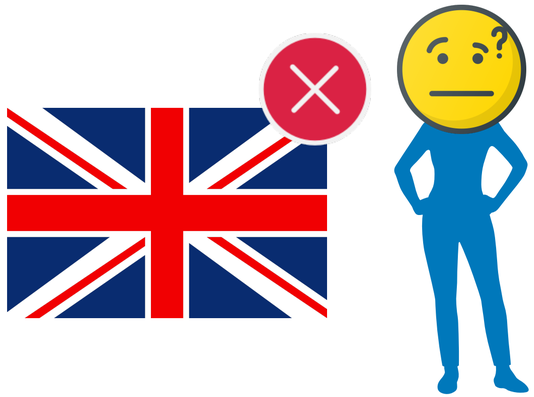
Via « Unknown », le TypeScript définit un type comme étant inconnu!

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

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

Exemple

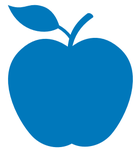
Variable



Datas types Objets

Pomme

Livre



Pomme

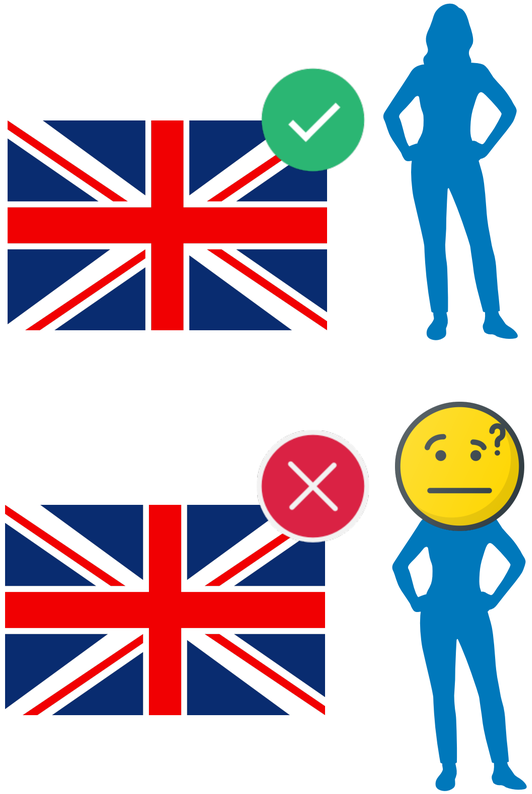
Livre

TYPES: COMESTIBLES

Flag icons from: https://www.iconfinder.com/iconsets/flags-free

Exemple avec vérification

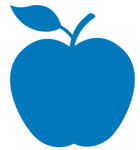
Variable



Datas types Objets

Pomme

Livre



Pomme

Livre

TYPES: COMESTIBLES

Flag icons from: https://www.iconfinder.com/iconsets/flags-free

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 êtres très explicite : (param: type) => return type

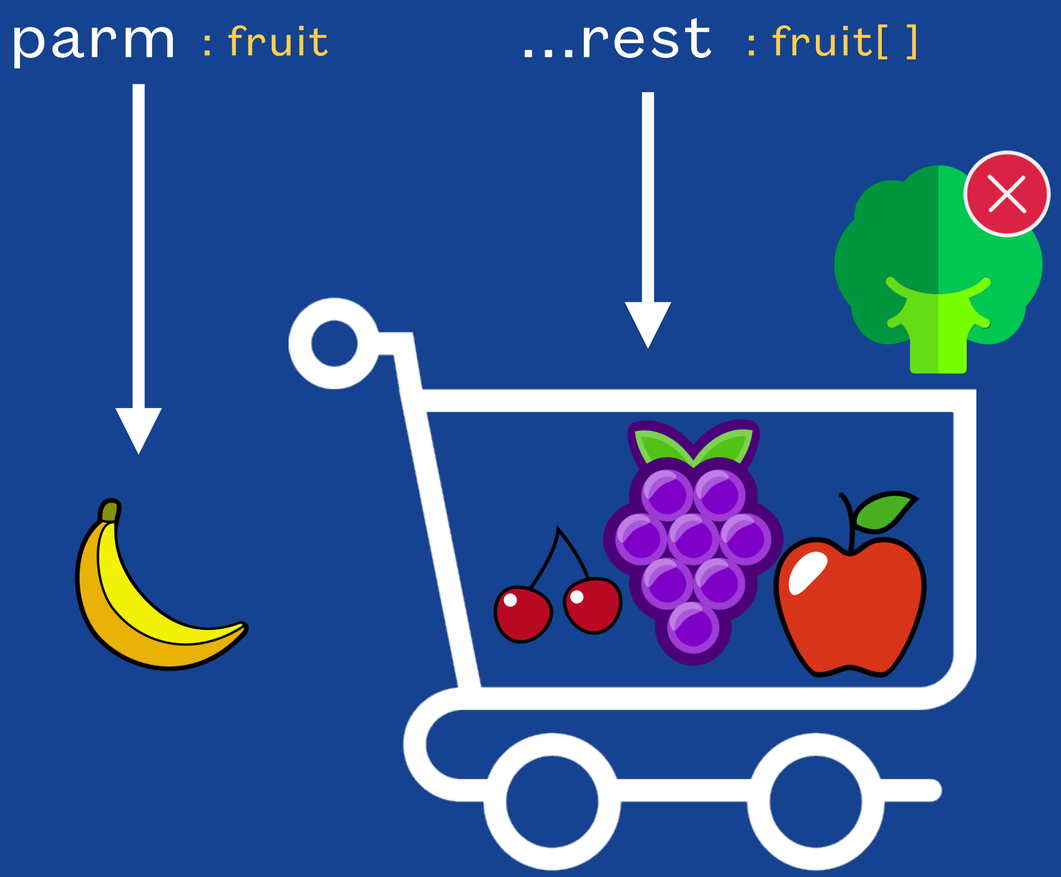
Function Types

- Paramètres facultatifs
- Paramètres par défaut

Paramètre Rest

- Rest dans les paramètres de fonction - Type

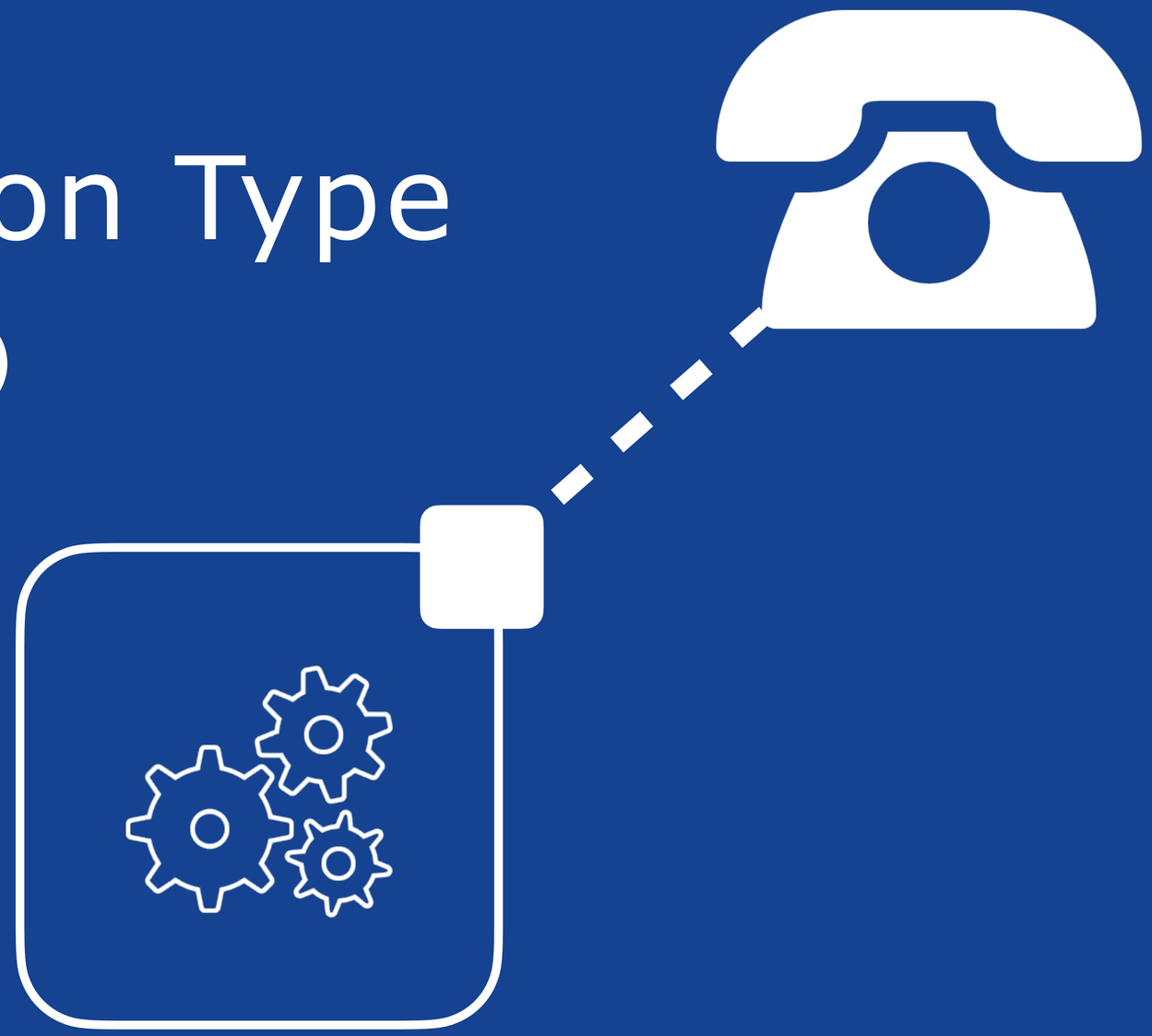
- Rest dans le Type Function



Icons: fruits: https://www.iconfinder.com/search/?q=fruits&price=free - broccoli : https://www.iconfinder.com/search/?q=salad&price=free

CallBack Function Type

(Fonction de rappel)



Union Type

```
string | 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’ ✓

Type Never (cas rares)

Le type « never » est relativement nouveau dans TypeScript. Il représente le type de valeurs qui ne se produisent jamais.

Le type « never » est assignable à chaque type. Mais aucun type n'est assignable à un type « never », même le type « any » !

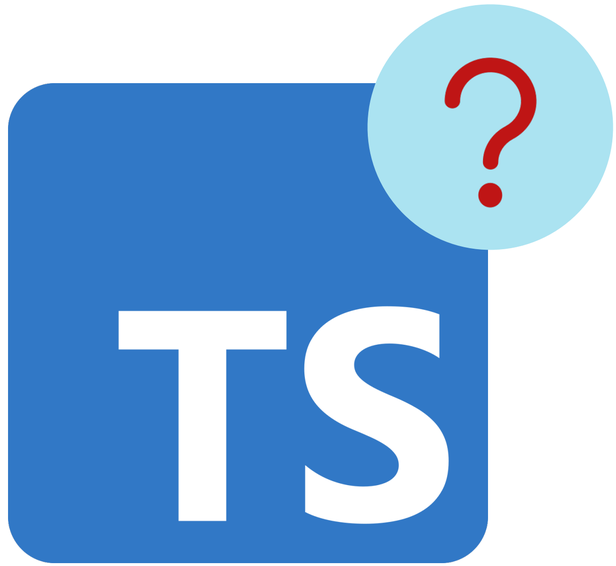
En effet, contrairement au type "void" qui s'applique dans les cas d’une fonction qui retourne "undefined", le type « Never », lui, indique qu’une fonction ne retourne jamais rien. Même pas "undefined". Le type « Never » peut donc s’avérer utile si on souhaite capturer une erreur via le throw ou dans le cas de boucles infinies..

Null

Undefined

Icon Basket: https://www.iconfinder.com/iconsets/Basket_icons_by_Asher

Type Assertions (Affirmatif)



C’est un Number !!!!

Fais moi confiance, je sais ce que je fais :-)

Type Assertions (Affirmatif)

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 »

Les Classes

Héritage et Polymorphisme en TypeScript

Personne

Propriétés

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

Méthodes

- Parler - Rire ..etc

Héritage

Les Classes

Access Modifiers

Personne

Propriétés

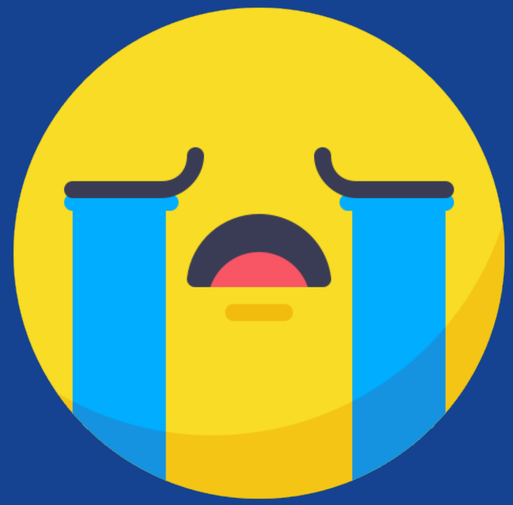
- Nom et prénom
- Cheveux châtain
- Yeux: 2
- Oreilles: 2 ... etc

Méthodes

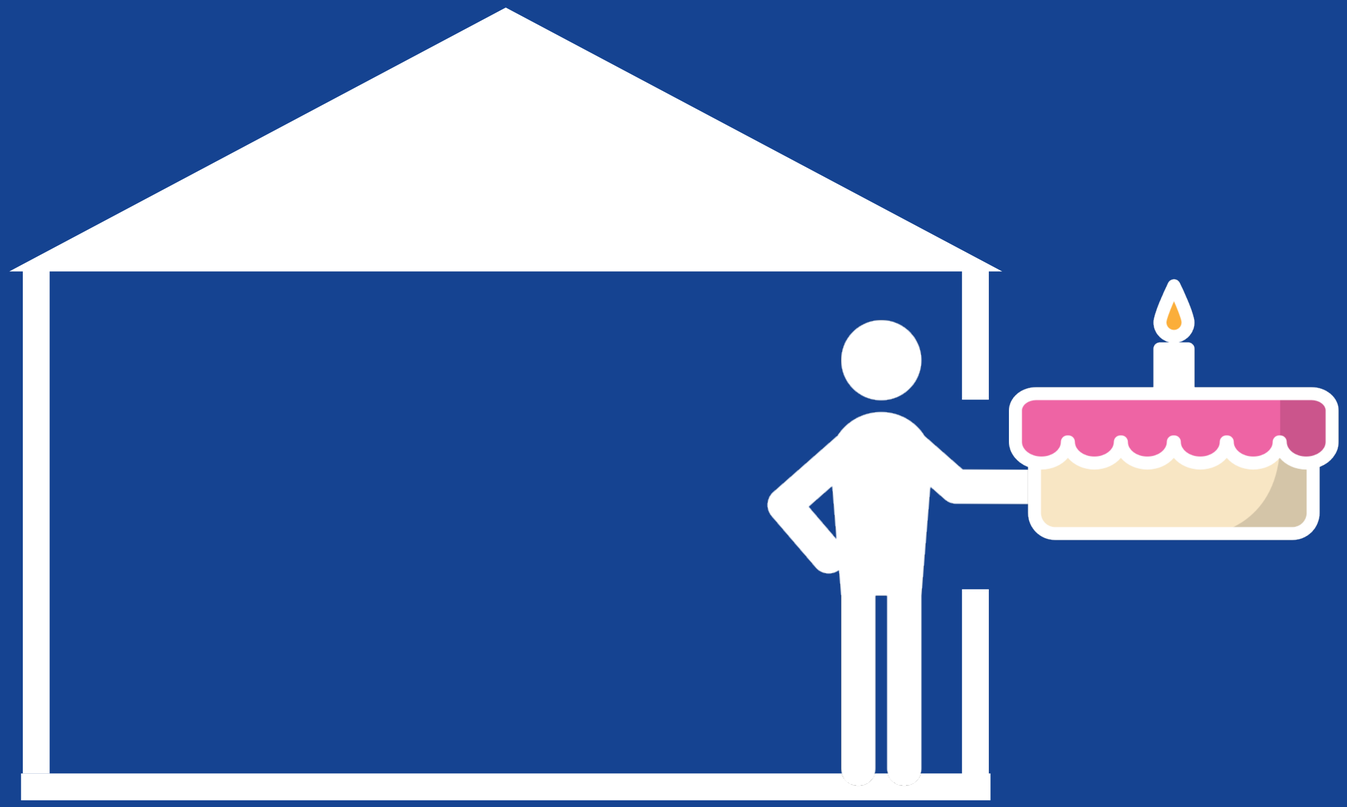
- Parler - Rire ..etc

Héritage

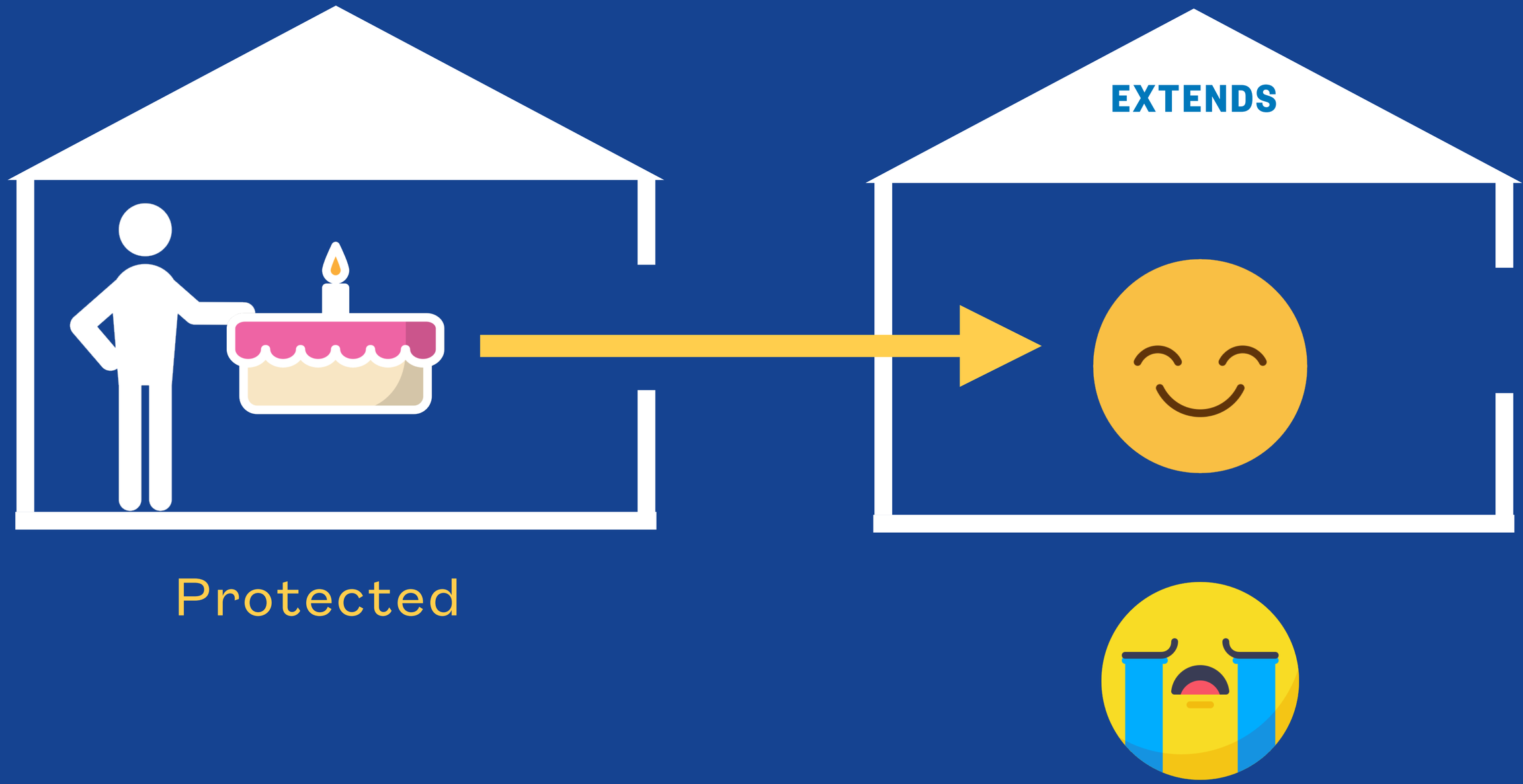
Public (accessible à l’extérieur)



Private (Accessible seulement dans la classe)



Getters & Setters


EXTENDS
Protected

readonly (Peut être modifié que via le constructor)

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..

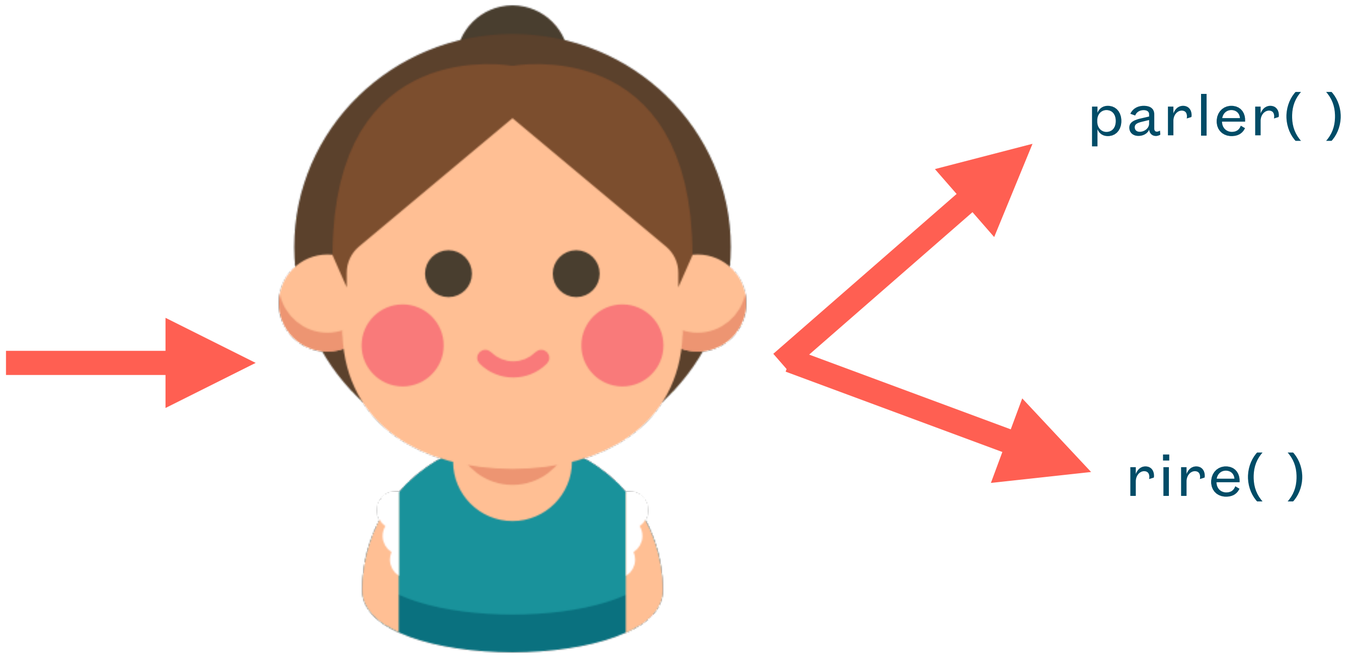
Les Classes (static)

class Person

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

- Parler - Rire ..etc

Object



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

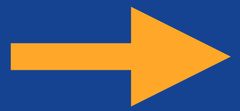
Type Any

Accepte tous les types mais ne les détecte pas

Type *Gerenic*

- 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

Partial<T>

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

Nouveau type Partial avec

des propriétés facultatives

```
Partial<Todo> {
    title?: string;
    description?: string
}
```

Readonly<T>

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

Readonly prends un paramètre de type « Generics » qui doit être une « Interface ».

```
Readonly<Todo> = {
  readonly title: string;
  readonly description: string;
}
```

Cela nous permet de générer un nouveau type dont les propriétés de l'interface « T » seront en mode lecture seulement.

Record<keys, Type>

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

KEY: STRING, NUMBER, STRING LITERAL...ETC

```
Record<Key, Todo> = {
  Key: { title: string, description: string },
  Key: { title: string, description: string }
}
```

Permet de construire un Type avec un ensemble de « propriétés Clés de type Type ».

Cet utilitaire peut être utilisé pour mapper les propriétés d'un type à un autre type.

Pick<Type, Keys>

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

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

Le Generic Utility Type "Pick" permet de construire un nouveau Type en sélectionnant les propriétés Keys (clés) dans le Type utilisé (Interface).

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
```

NonNullable<Type>

Le Generic Utility Type "NonNullable" nécessite un seul paramètre « Type » qui doit être un « Union Type » et permet de construire un nouveau type en excluant les types « NULL » et « UNDEFINED »

En gros, c’est un type qui interdit les types « NULL » et « UNDEFINED »

```
type A = string | string[ ] | undefined | null
```

```
NonNullable<A> = string | string[ ]
```

Parameters<Type>

Le Generic Utility Type "Parameters" permet de construire un nouveau type sous forme de TUPLE à partir des types utilisés dans les paramètres function Type.

```
type Parameters<T extends (...args: any) => any>
= T extends (...args: infer P) => any ? P : never;
```

ReturnType<Type>

Le Generic Utility Type "ReturnType" permet de construire un nouveau type en fonction du type retourné par une fonction.

```
function  myFunction {
    return {
        age : 25, name: « toto »
    }
}
```

```
ReturnType<typeof myFunction>
```

Nouveau Type object avec
2 propriétés « age, name »

ReturnType<Type>

Le Generic Utility Type "ReturnType" permet de construire un nouveau type en fonction du type retourné par une fonction.

```
function  myFunction {
    return {
        age : 25, name: « toto »
    }
}
```

```
ReturnType<typeof myFunction>
```

Nouveau Type object avec
2 propriétés « age, name »

TypeScript Compiler

Installer le TypeScript Compiler

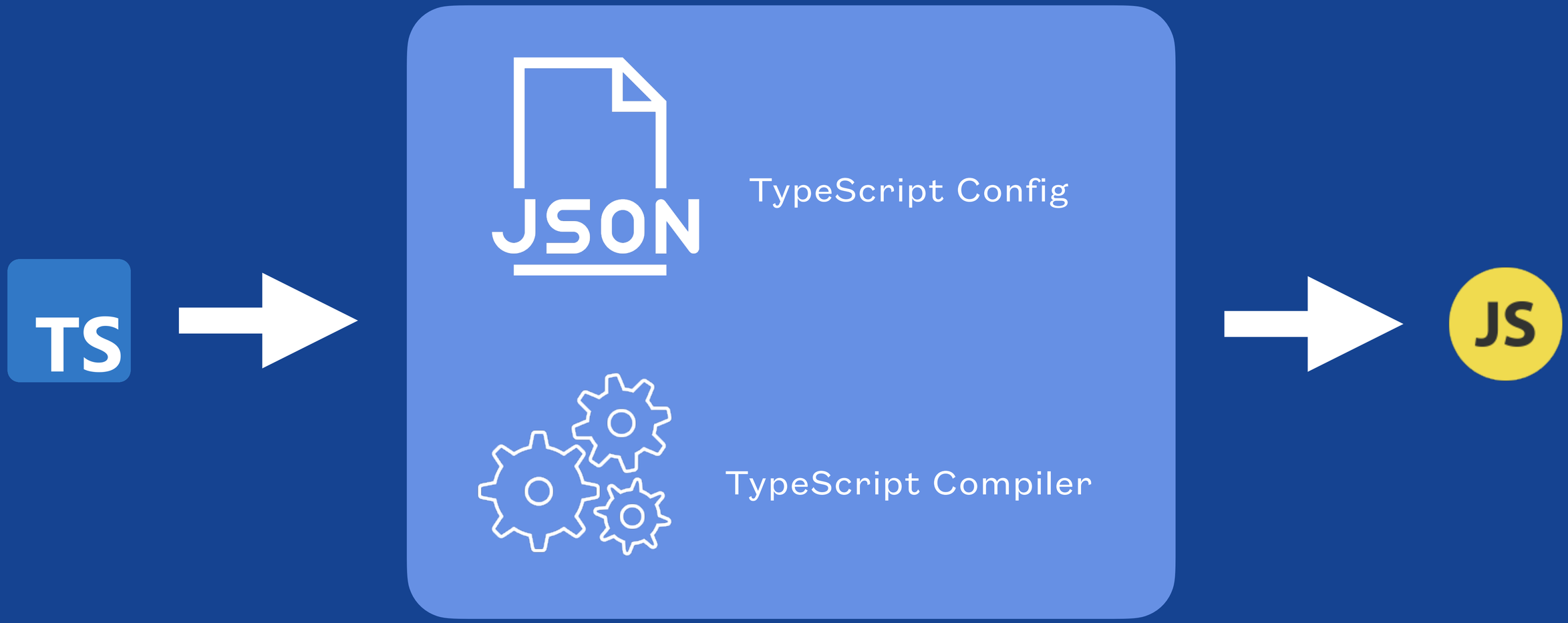
Compiler les fichiers `tsc file.ts`

Compiler les fichiers avec des arguments `tsc file.ts —out output-file.js` (tsc -help)

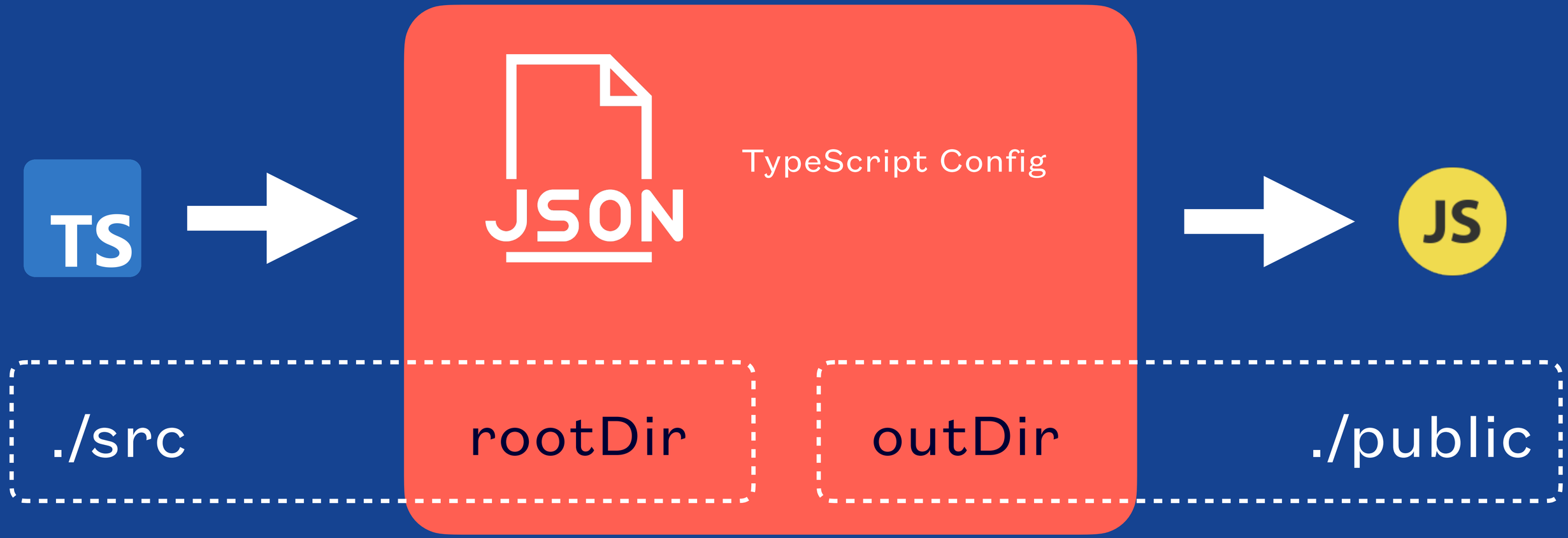
Compiler les fichiers automatiquement après validation `tsc —watch` `tsc -w`

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

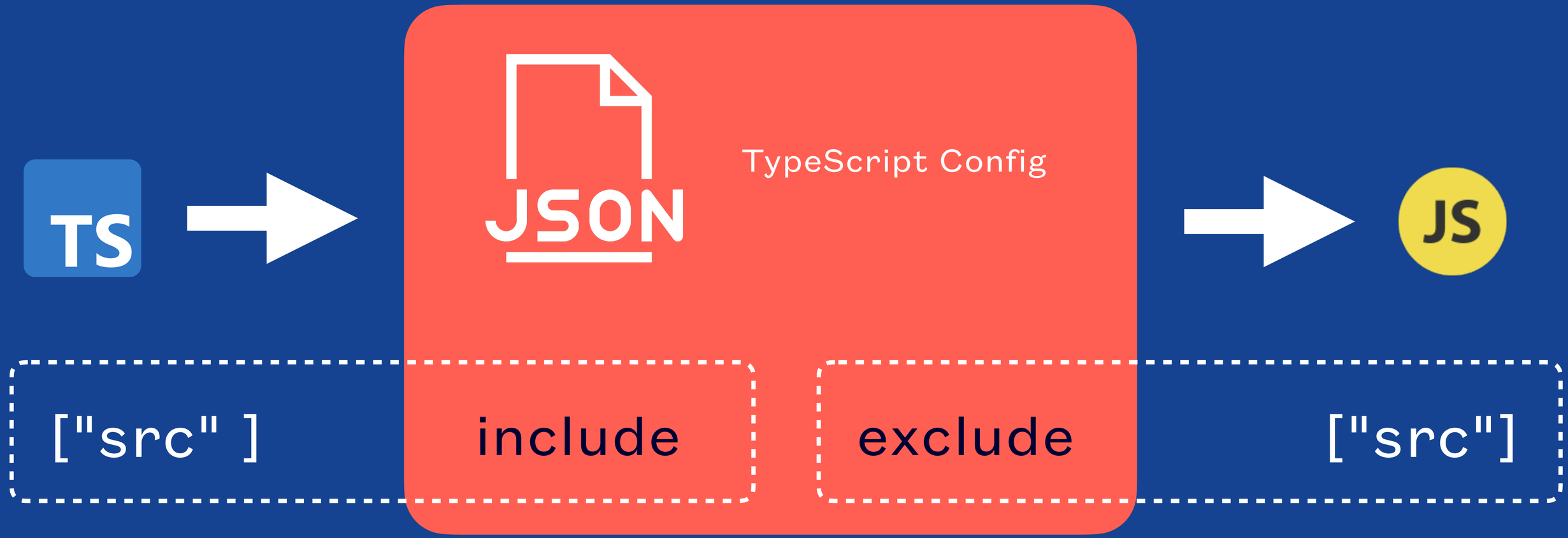
TypeScript Compiler - tsconfig.json



TypeScript Compiler - tsconfig.json



TypeScript Compiler - tsconfig.json



TypeScript Compiler - tsconfig.json

target: "es5"

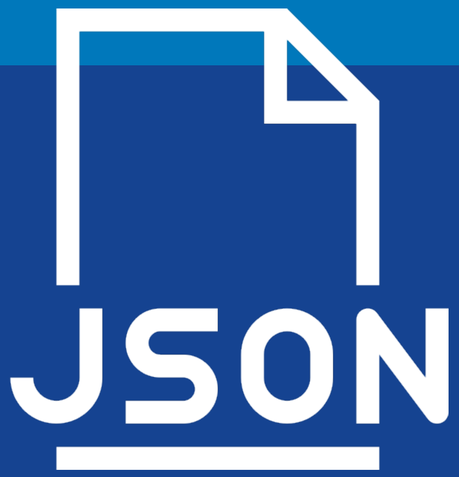
module

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 êtres utile si travaillez coté 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*

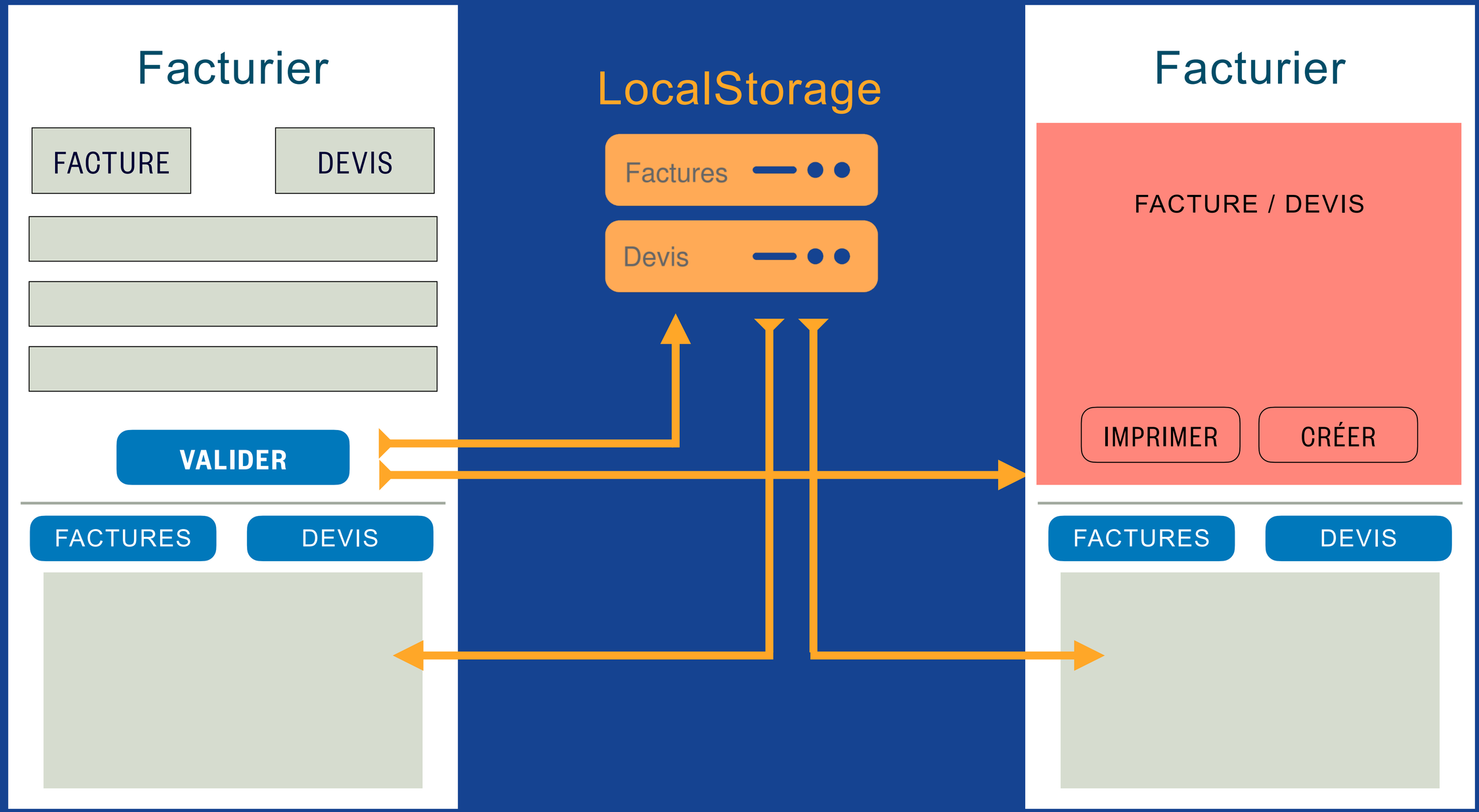
Le rest des options

Les Modules

Namespaces

Modules ES6

Reconnu dans TypeScript


Facturier
FACTURE
DEVIS
VALIDER
FACTURES
DEVIS
LocalStorage
Factures
Devis
Facturier
FACTURE / DEVIS
IMPRIMER
CRÉER
FACTURES
DEVIS

Créer les dossiers de notre application et paramétrer le fichier **tsconfig.json**

Vidéo 2

Récupérer les éléments du formulaire

Valider du formulaire

Afficher les valeurs validées via un console.log( )

Refaire la validation du formulaire via l’instanciation d’une class ‘ FormInput ’

Vérification du formulaire (Valeurs numériques > 0)

Une fois les valeurs validées, elles seront retournées dans un TUPLE

Affichage des données via un console.log( )

Vidéo 3

Créer un dossier /classes pour nos classes

Transférer notre code dans le dossier class

Exporter et l’importer dans la classe dans script.ts pour l’instanciation

---

Continuer avec notre console.log et tester les valeurs négatives

Utiliser les valeurs validées pour générer notre document

Créer une class ‘ Datas ’ et lui implémenter une Interface ‘ HasHtmlFormat ’

Créer un dossier /interfaces avec l’interface ‘ HasHtmlFormat ’

Vidéo 4

Maintenant que nous avons pu générer notre Template, il faut y injecter les données

- Les données du client
- Les données du produit / service
- Calculer le prix TTC en fonction du taux de TVA indiqué
- Tester pour visualiser le template du document via un console.log( )

Vidéo 5

Afficher le template du document généré dans le DOM

- Créer une class Display qui implémente l’interface HasRender

- Cacher le formulaire pour afficher le document

- Voir comment invoquer une méthode public via l’objet instancié et lui passé un objet instancié d’une autre class en tant que argument tout en respectant l’interface définie pour cet objet.

Vidéo 6

- Travailler le text du bouton « Imprimer la facture » « Imprimer le devis »

- Passer le bouton lors de l’instanciation de la class Display pour pouvoir y accéder et y injecter le texte

Vidéo 7

- Mettre en place un listener pour l’événement ‘click’ sur le bouton ‘ print ’

- Créer la class « Print » qui implémente l’interface ‘ HasPrint ’

- via l’objet instancié de la class ‘ Print ’, j’accède à la méthode print( )

- Une fois le document imprimé, je recharge la page pour pouvoir créer d’autres documents.

Vidéo 8

- Mettre en place un listener pour l’événement ‘click’ sur le bouton ‘ reload ’

Vidéo 9

- Initialiser deux array dans le localStorage pour y enregistrer les documents créés
    - Item INVOICE [ ]
    - Item ESTIMATE [ ]

- Créer une class ‘ Storage ’ avec une méthode STATIC qui va se charger d’initialiser ces deux arrays

Vidéo 10

- Créer une Interface ‘ HasSetItem ’ et l’implémenter pour la class ‘ Storage ’

- Créer une méthode public ‘setItem’ qui va se charger de vérifier et d’enregistrer le document dans le localStorage

Vidéo 11

Afficher les documents enregistrés dans le localStorage en cliquant sur les 2 boutons en bas de la page.

Decorators

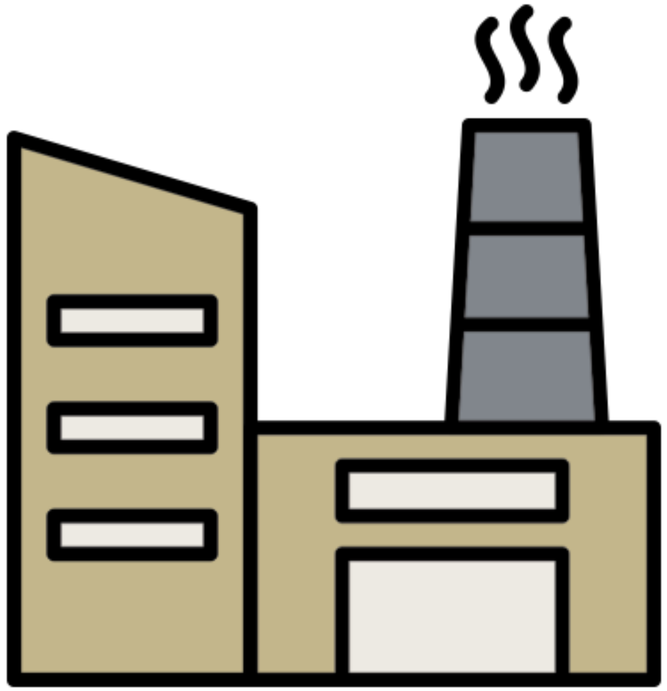
Les decorators sont actuellement en phase expérimentale et peuvent subir des changements dans les prochaines versions JavaScript. Néanmoins, je les aborde dans cette formation car c’est un pattern assez important adopté largement dans les récents frameworks tel que Angular

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

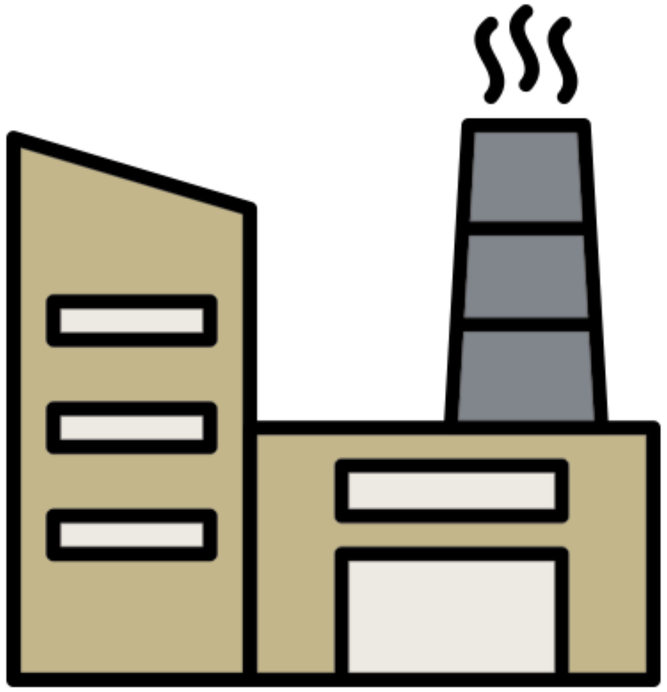


Function(param)

Anonymous Function

Decorator Factories

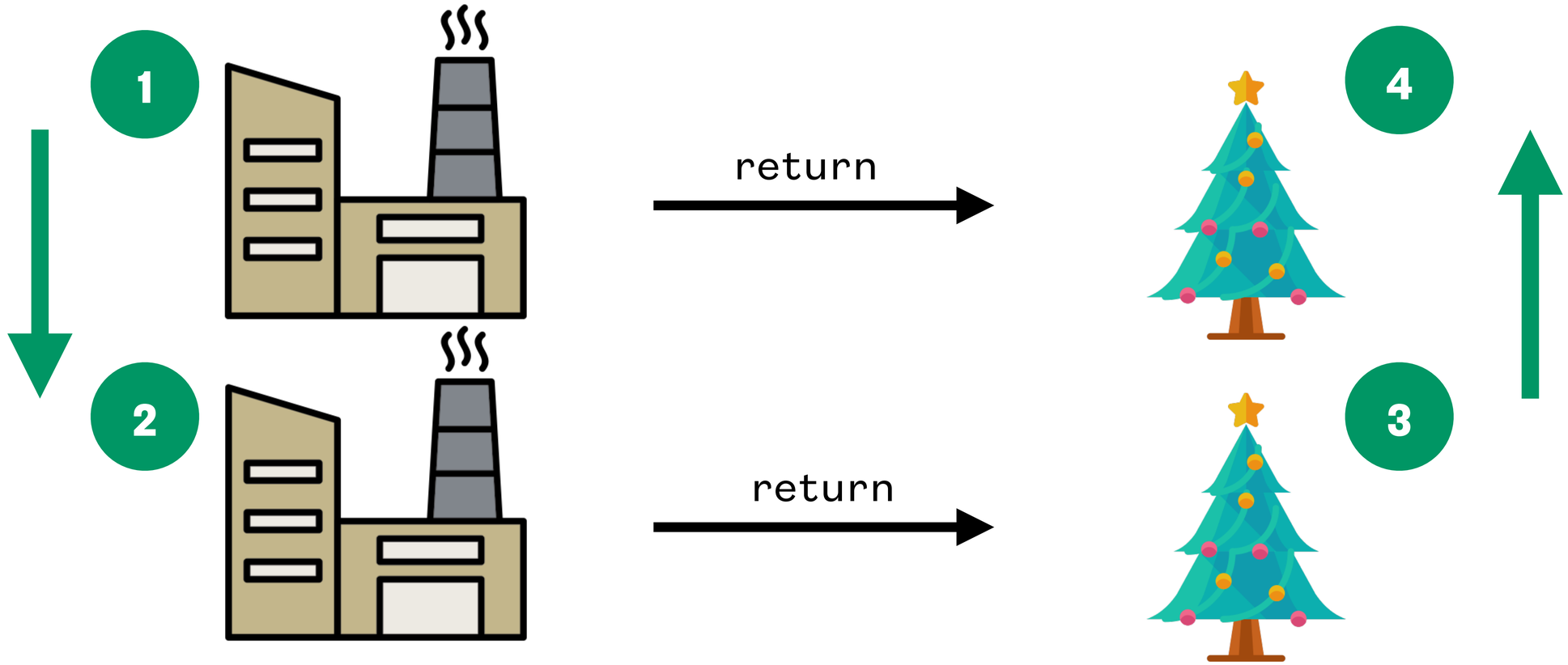
Injecter du contenu dans le DOM à la manière Angular via un decorator @component



Function(param)

Anonymous Function

Multiple Decorator Factories



Class Decorator

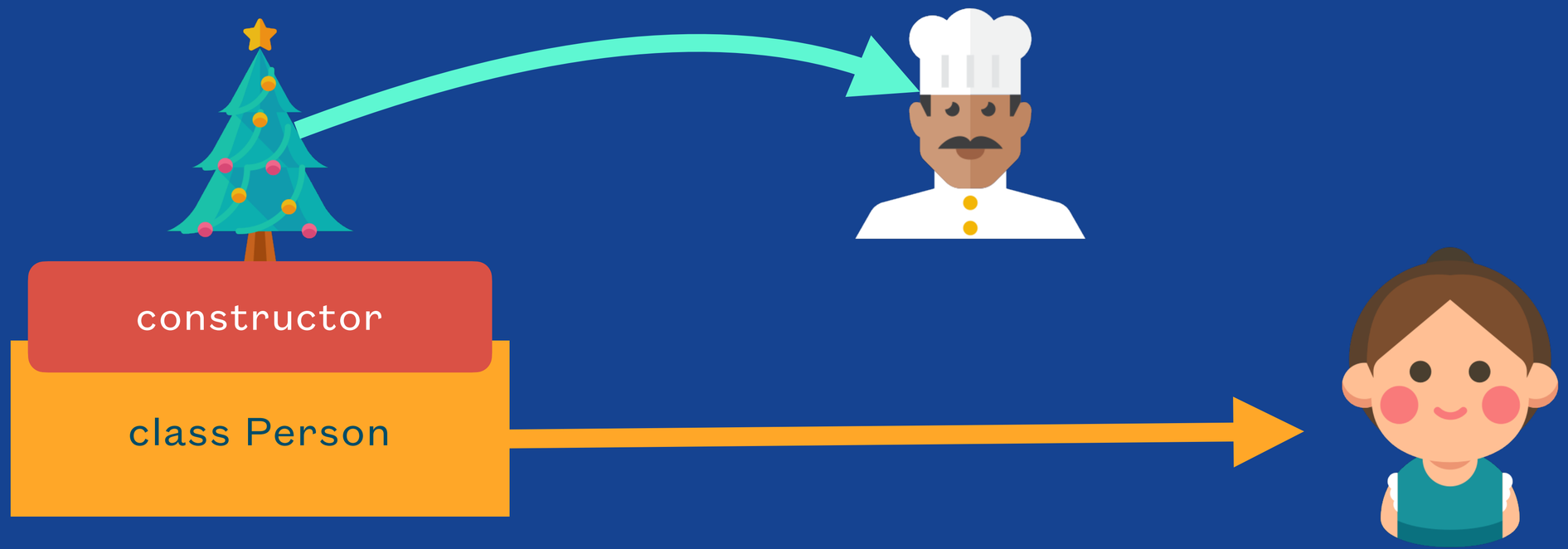
- Propriétés
- Accessors (getters - setters)
- Méthodes
- Paramètres

Class Decorator

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

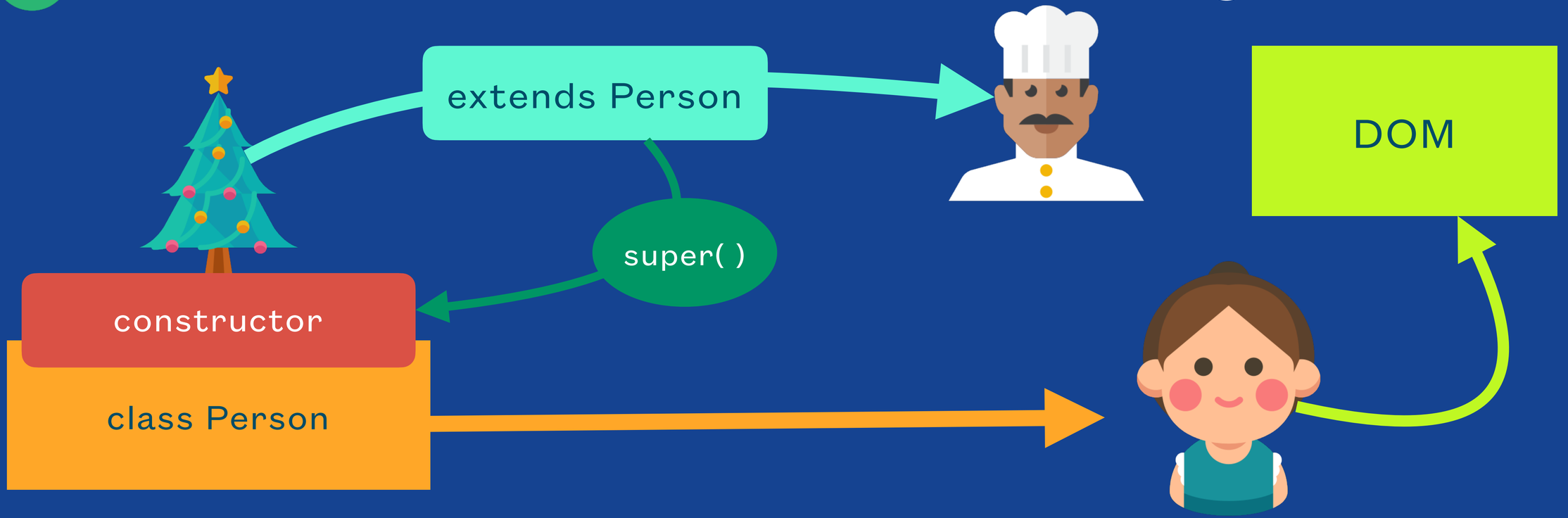
Class Decorator

- class décorateur peut réécrire la fonction constructor



Class Decorator

- Réécrire la fonction constructor via l’héritage



Exercice

Utiliser un decorator de méthode comme alternative au bind(this) que nous avons vu dans notre petite application « Facturier »

Nous allons donc faire un petit exercice sur un événement « click » pour invoquer une méthode d’un objet instancié d’une classe.

Button

Exercice (Suite)

Appliquer le décorateur de l’exercice précédent à notre petite application « Facturier » au niveau de la class FormInputs

Form Submit

handleFormSubmit

Conditional Types

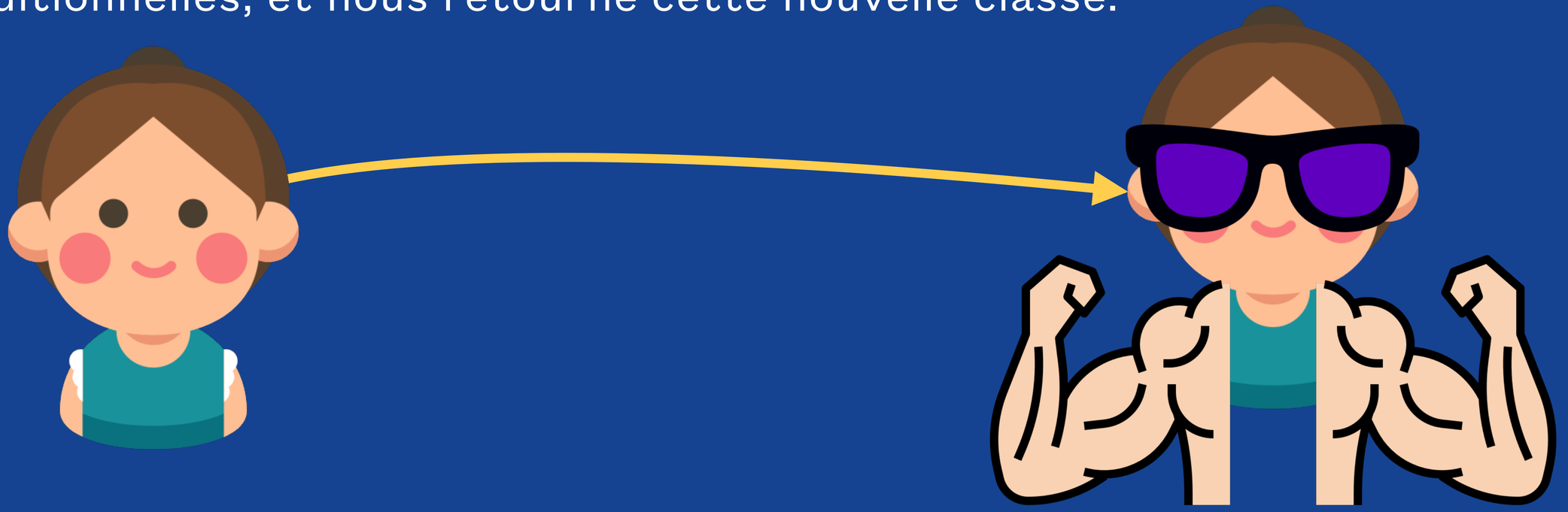
Un type conditionnel sélectionne, via un opérateur conditionnel, l'un des deux types possibles en fonction d'une condition exprimée sous la forme d'un test de relation de type

Mapped Types

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

Les Mixins

Un Mixin est une fonction qui prends un constructor afin de nous créer une nouvelle classe qui hérite de ce constructor, avec des fonctionnalités additionnelles, et nous retourne cette nouvelle classe.



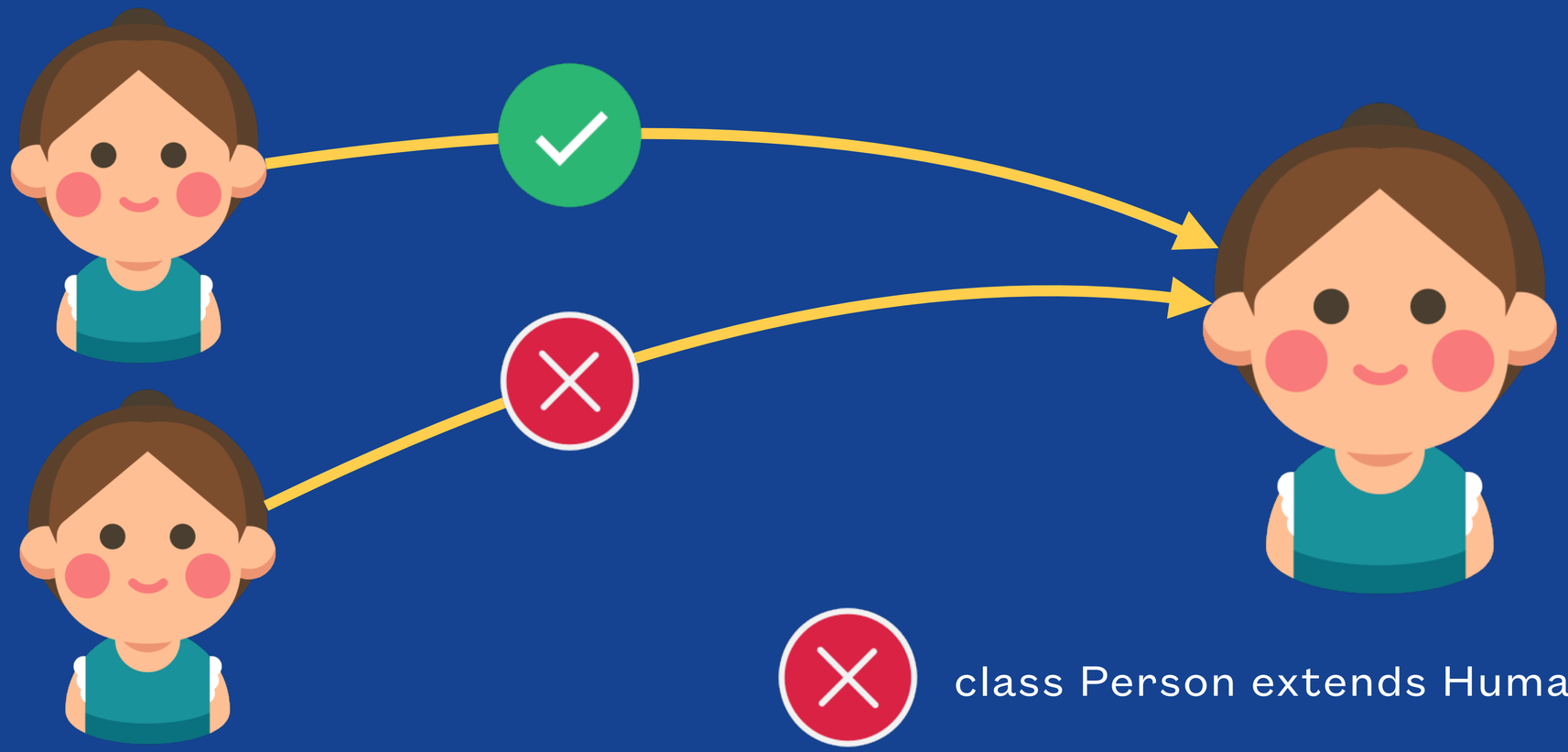
icons from: https://www.iconfinder.com

Mixins

Une class ne peut hériter que d’une seule classe

Class Human

Class Robot



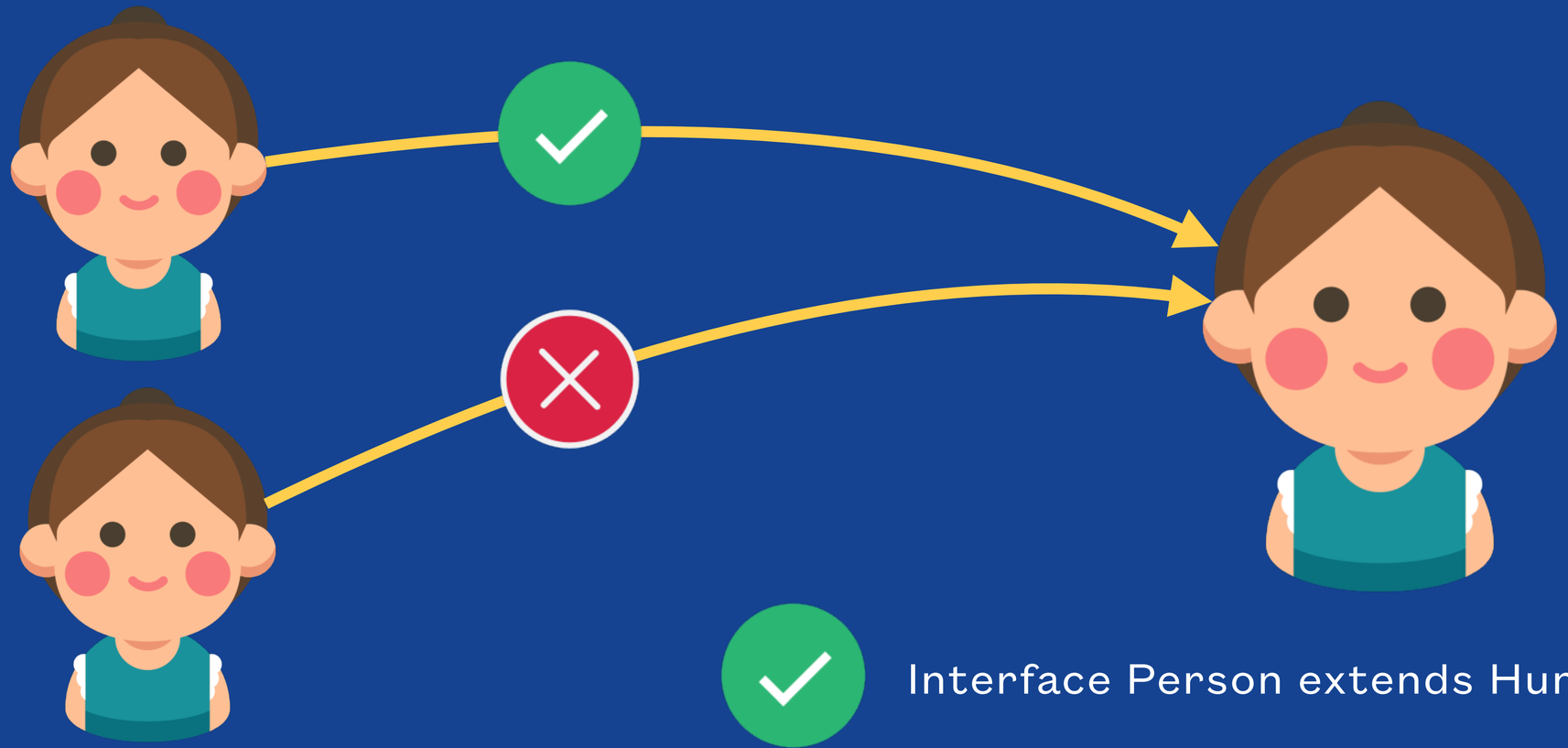
class Person extends Human, Robot

Mixins

Une interface peut hériter de plusieurs classes

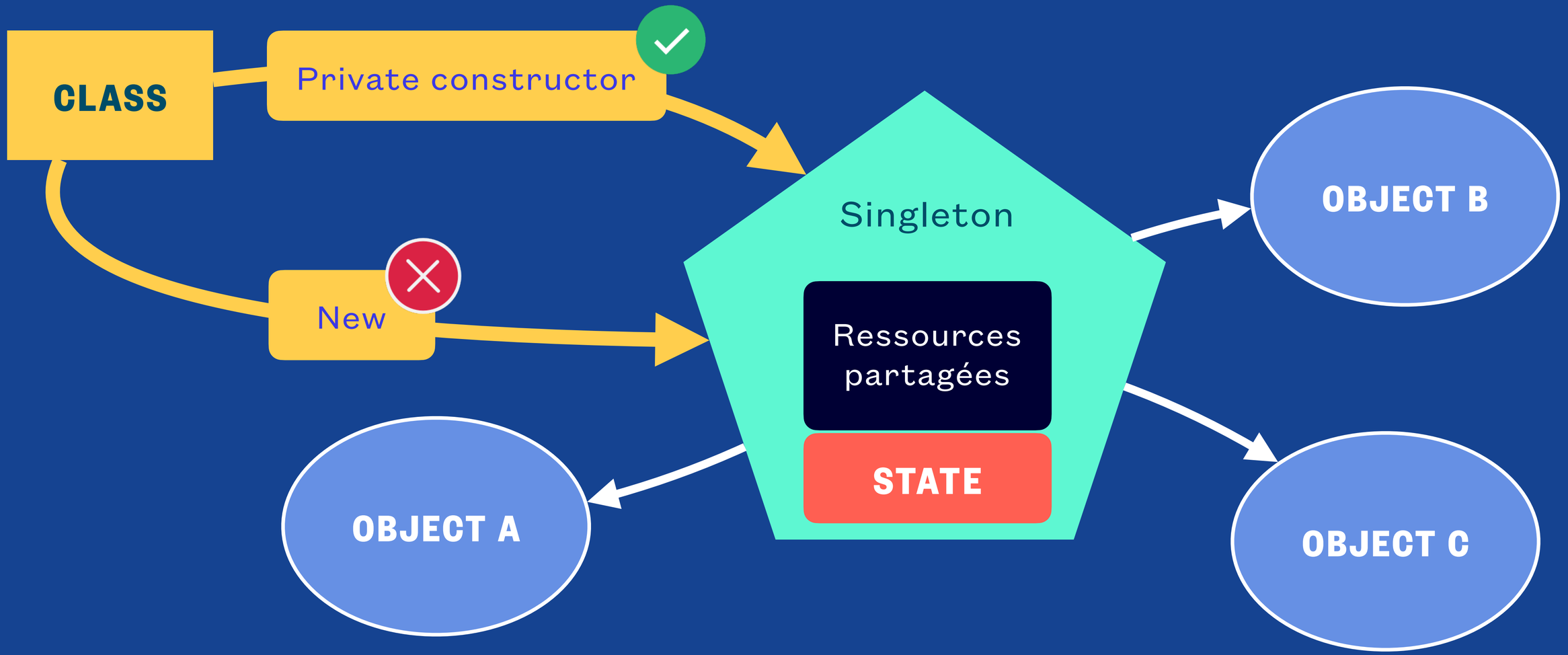
Class Human

Class Robot



Interface Person extends Human, Robot

Singleton Design Pattern



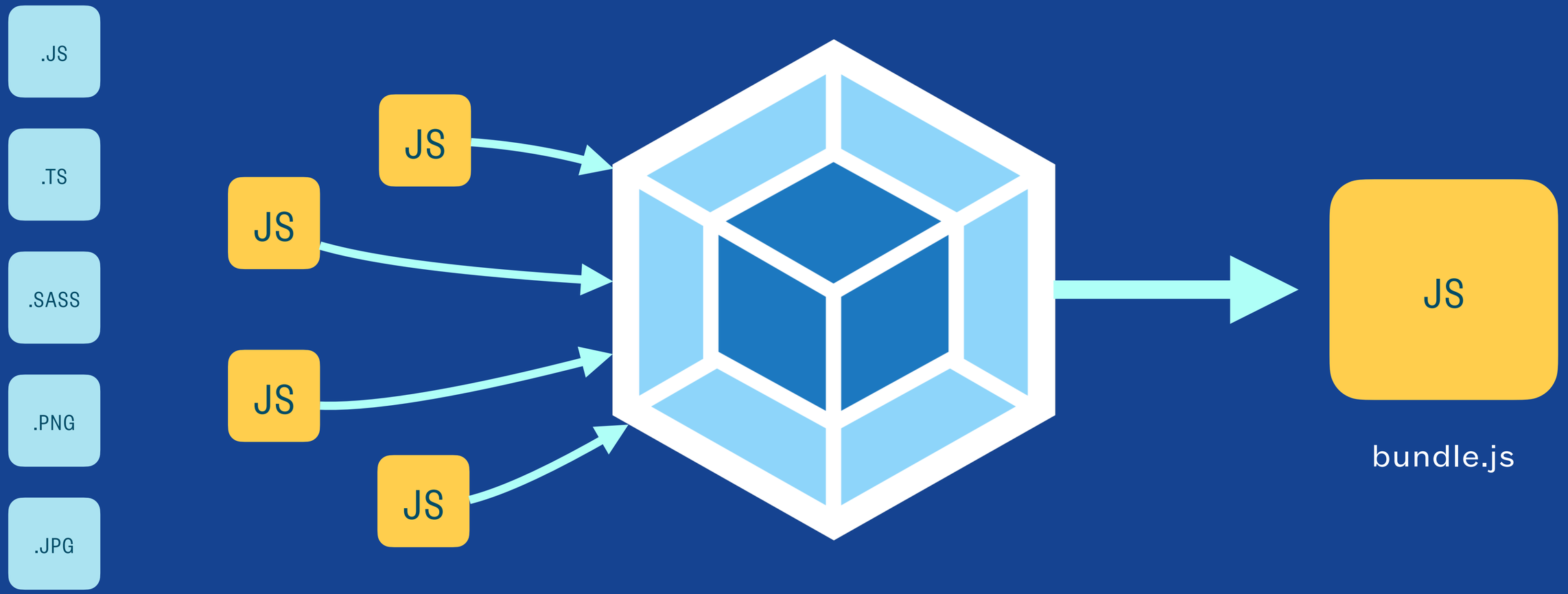
Créer un projet NPM

```
npm -v
```

- Initialiser le fichier package.json `npm init`
- Renommer le fichier ‘ script.ts ’ en ‘ index.ts ’

`npm install ***`

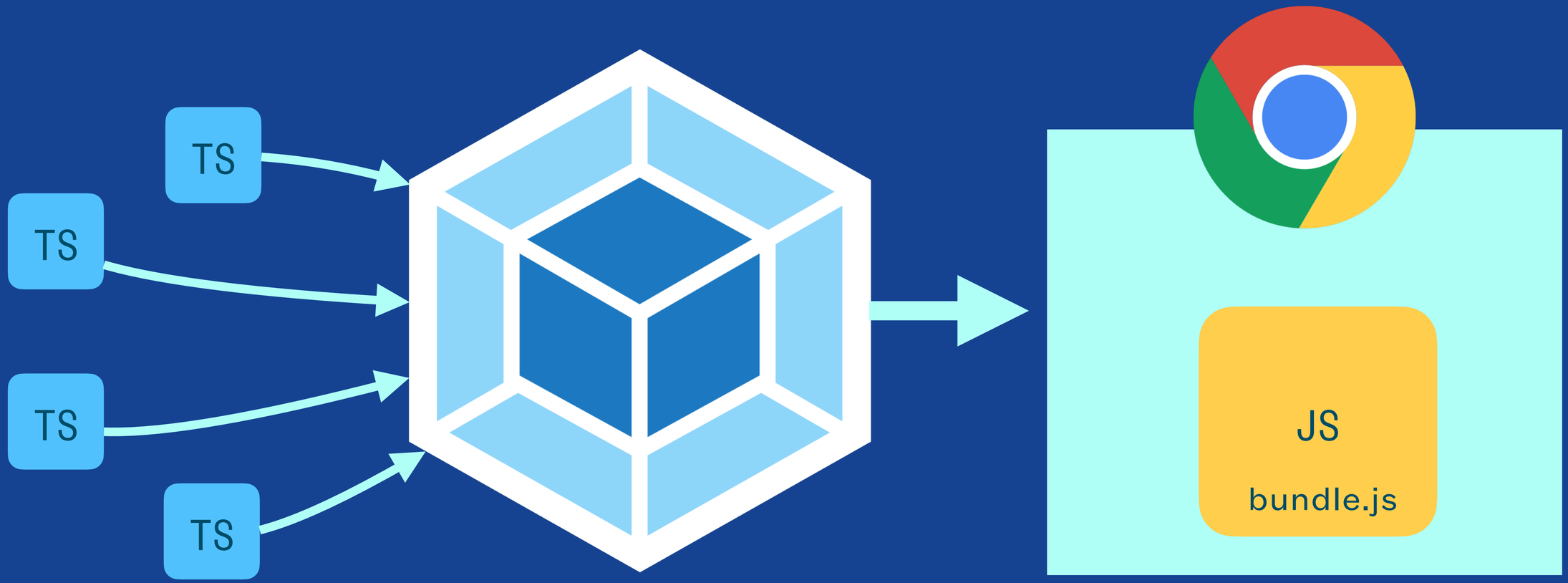
Webpack (bundler)



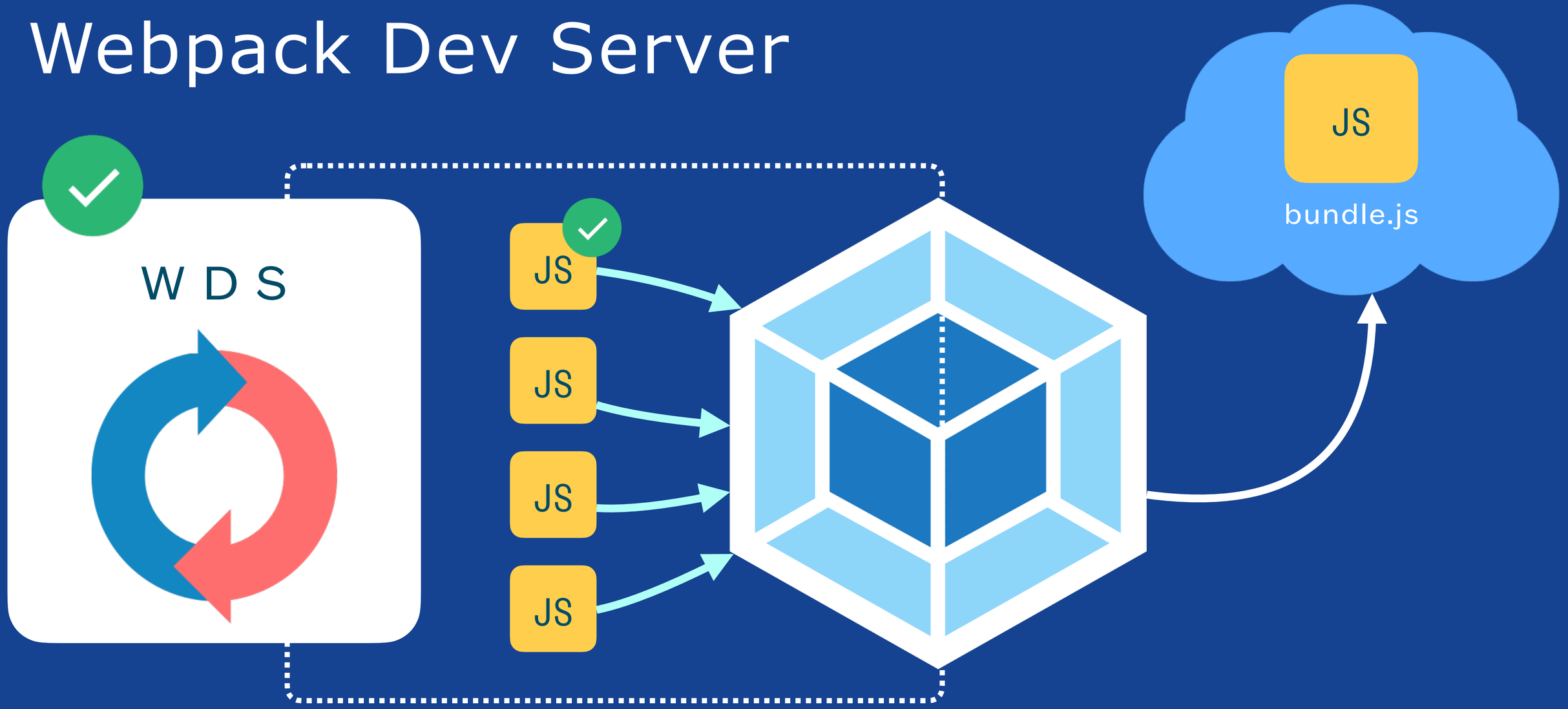
npm install webpack webpack-cli typescript ts-loader - -save-dev

webpack.config.js

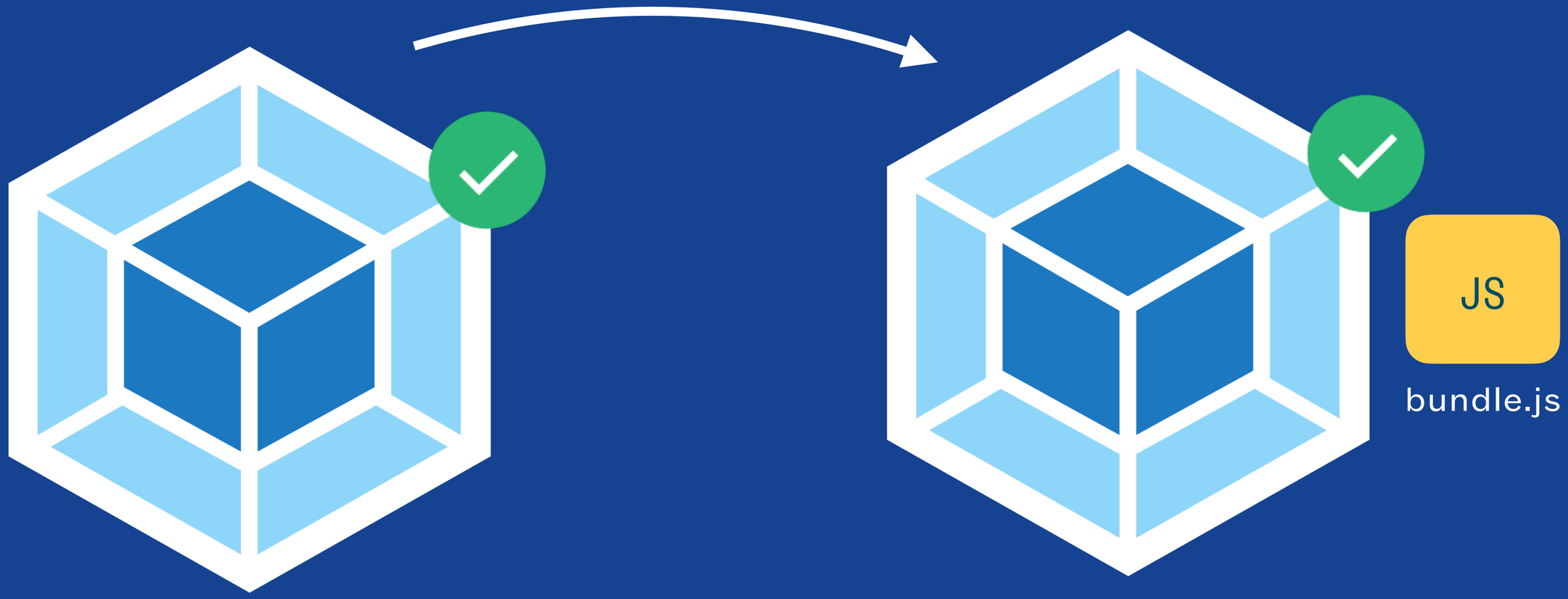
Webpack (TypeScript Debug - SourceMaps)



Webpack Dev Server



npm install webpack-dev-server - -save-dev



Webpack coté Dev

Webpack coté Production

a) Paramètres *«CleanWebpackPlugin»*

b) Type Definitions