

Programmation Web

Tristan Crolard

Laboratoire CEDRIC
Equipe « Systèmes Sûrs »

`tristan.crolard@cnam.fr`

`cedric.cnam.fr/sys/crolard`

Programmation Web mobile

- ▶ Le **DOM** (*Document Object Model*) impose des contraintes :
 - le langage **JavaScript** pour le développement
 - une programmation **événementielle** (pour l'UI) et **asynchrone** (accès réseau)
 - des **mutations** du DOM pour actualiser l'affichage (sans recharger toute la page)

- ▶ Ce style de programmation ne passe pas à l'échelle (trop complexe) :
 - **TypeScript** : extension **statiquement typée** de JavaScript
 - des *frameworks* pour faciliter le développement (comme Angular ou React)
 - des **paradigmes** associés (**fonctionnel** pour React et **réactif** pour Angular)

Les paradigmes de programmation

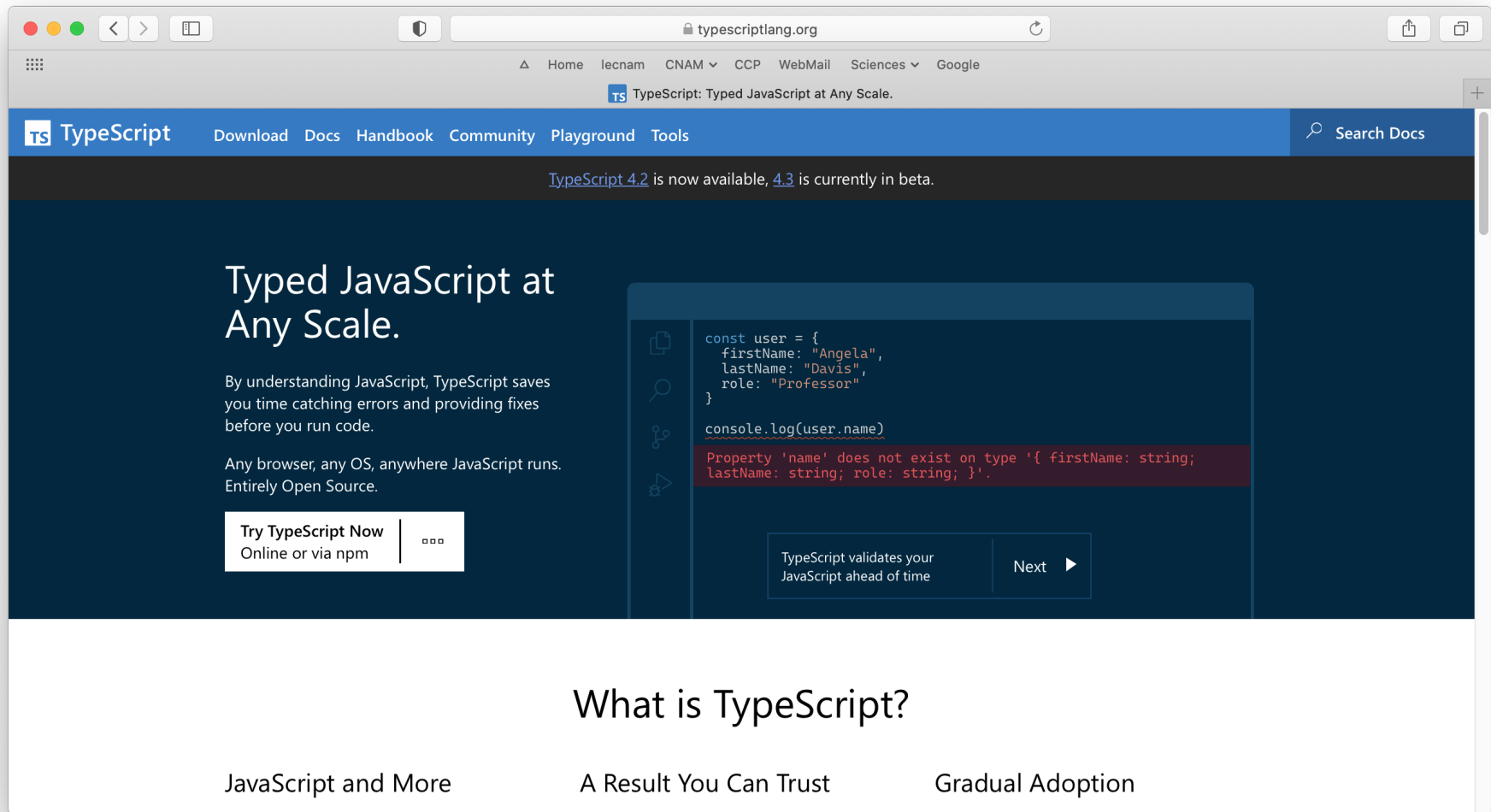
- ▶ **Style impératif** : variables (mutables), procédures, instructions (sequence, boucles, ...)
- ▶ **Style fonctionnel** : constantes, fonctions et expressions

Style	Données	
	mutables	immutables
impératif	idéal	ok
impératif asynchrone	ko	ok
fonctionnel	/	idéal
fonctionnel asynchrone	/	idéal

→ style **impératif** : données **mutables**

→ style **fonctionnel** : données **immutables**

TypeScript



<https://www.typescriptlang.org>

De JavaScript à TypeScript

- ▶ **JavaScript** est un langage **impératif, objet et fonctionnel** à **typage dynamique**.
- ▶ **TypeScript** : extension **statiquement** typée de JavaScript.
 - recommandée, ou imposée, pour les principaux frameworks
- ▶ Types **prédéfinis** : number, string, boolean, ...
- ▶ Définition de **types de données structurés** : enregistrements et tableaux (n -uplets)
- ▶ Possibilité de rendre ces **types de données** immutables (attribut **readonly**)
 - notation *spread* de JavaScript pour les mises-à-jour fonctionnelles
- ▶ **Annotations** des fonctions (arguments et résultat) avec des types
 - annotations **vérifiées** puis **effacées** par le compilateur TypeScript

De JavaScript à TypeScript

The screenshot shows a web browser window with the URL `orta.io`. The page title is "Understanding TypeScript's Popularity | Notes". The main content is a "Major Events Timeline" with the following text: "IMO, these are the big events which enabled TypeScript to keep breaking possible popularity ceilings:". The timeline consists of six bullet points:

- 2014 - TypeScript re-write, TS v1.1** - Post-launch and with an understanding of what TypeScript is, the entire codebase was mostly thrown away and re-written in a functional style (instead of classes with mutation) - this architecture still stands today, and is built with long-running processes and *very rare* mutations in mind. Someone once mentioned that the *precursor* to TypeScript (Strada) was written in C++, not certain on that through.
- 2015 - Angular adopts TypeScript, TS v1.5** - Google were looking at building their own language for Angular, instead opting for using TypeScript. To make this possible, TypeScript broke one of its cardinal rules: Do not implement a TC39 thing early. Thus `experimentalDecorators` support in TypeScript. This technical debt was *totally* worth it for everyone involved, even if a 6 years down the line decorators have not been added to JavaScript.
- 2015 - JSX support in TypeScript, TS v1.6** - React was also growing to be an extremely popular user interface libraries, and React uses JSX: a JS language extension which effectively supports writing HTML inside JavaScript. TypeScript's support for JSX *allowed* others to add support for React (support for React lives in `@types/react`, not inside TypeScript)
- 2016 - Undefined and Control Flow Analysis, TS v2.0** - Building on a feature from 1.4, union types - TypeScript added support for declaring that a type could be there or not. This allowed for a type system which could really model most existing JavaScript code. Coming at the same times was code flow analysis which means that if statements and other user code can affect what the type is on different lines/positions.
- 2016 - Embracing DefinitelyTyped, TS v2.0** - DefinitelyTyped was a side-project handled by volunteers, there were a few different DefinitelyTyped-like systems at the time and the TypeScript team adopted DefinitelyTyped and baked the idea of `@types/x` into the compiler itself. In adopting and taking maintainership of DefinitelyTyped, the team put serious testing and workflow improvements which helped it scale to be one of the most active repos on GitHub. The long-form story of [DT is worth a read here](#).
- 2016 - JavaScript support, TS v2.3** - While there was some existing JavaScript project support in the language tooling, JSDoc support *was added* which allowed JavaScript projects to start getting some of the benefits of TypeScript *without moving to TypeScript*. Creating the start of an incremental migration path to a TypeScript codebase, but also offering a way to give tooling to JavaScript projects which already exist today.

The right sidebar contains navigation links: "Show Graph Visualisation", "Activate dark mode", "Understanding TypeScript's Popularity", "How we got here", "Major Events Timeline", "What were TypeScript's Competitors?", "Future", "Guestimates at the future of TypeScript?", "Current Competitors", "How TypeScript see its position in the Ecosystem", "How does TypeScript think of in terms of its audience??", "How does TypeScript track the JS ecosystem?".

<https://orta.io/notes/js/why-typescript>

Exemples : enregistrements mutables/immuables

```
type Point = { x: number, y: number }  
const p1: Point = { x: 3, y: 7 }  
x1 = p1.x; // ok  
p1.x = 4; // ok
```

```
type Point = { readonly x: number, readonly y: number }  
const p1: Point = { x: 3, y: 7 }  
x1 = p1.x; // ok  
p1.y = 4; // error  
const p2: Point = { x: p1.x, y: 4 } // ok  
const p3: Point = { ...p1, y: 4 } // ok, same point
```

Exemples : enregistrements immutables

```
type Address = {  
  readonly streetNumber: number,  
  readonly streetName: string,  
  readonly postalCode: number,  
  readonly city: string  
}
```

```
type Contact = {  
  readonly firstName: string,  
  readonly lastName: string,  
  readonly phone: string,  
  readonly address: Address  
}
```


Exemples : tableaux mutables/immuables

```
type Points = Point []  
  
const points: Points = [p1, p2, p3];  
const p4: Point = points[2]; // ok  
points[2] = p1; // ok
```

```
type Points = readonly Points []  
  
const points: Points = [p1, p2, p3];  
const p4: Point = points[2]; // ok  
points[2] = p1; // error
```

Tableaux immutables : insertion

- ▶ Cas particuliers : insertion au début (à gauche) ou à la fin (à droite)

```
function addLeft<A>(arr: readonly A[], index: number, value: A):  
A[] {  
    return [value, ...arr];  
}
```

```
function addRight<A>(arr: readonly A[], index: number, value: A):  
A[] {  
    return [...arr, value];  
}
```

- ▶ Cas général :

```
function insert<A>(arr: readonly A[], index: number, value: A): A[]  
{  
    return [...arr.slice(0, index), value, ...arr.slice(index)];  
}
```

Tableaux immutables : mise-à-jour fonctionnelle

- La mise-à-jour fonctionnelle est similaire à l'insertion (seule la seconde tranche change) :

```
function update<A>(arr: readonly A[], index: number, value: A): A[]  
{  
    return [...arr.slice(0, index), value, ...arr.slice(index +  
1)];  
}
```

Remarque. Une nouvelle notation JavaScript pour slice est envisagée, cela donnerait :

```
return [...arr[:index], value, ...arr[index + 1:]];
```

- Il est aussi possible de l'écrire avec la méthode fill :

```
function update<A>(arr: readonly A[], index: number, value: A): A[]  
{  
    return [...arr].fill(value, index, index + 1);  
}
```