**Angular 9**

**Angular et ECMAScript 6**

**C'est quoi, ECMAScript 6 ?**

ECMAScript 6 est le nom de la **dernière version standardisé** de JavaScript.

Si vous utilisez JavaScript pour vos développements, vous avez sans doute remarquez que JavaScript est un language un peu à part. Il existe un système d'héritage *"prototypal"*, des fonctions *"anonymes"* ... Le besoin d'une nouvelle standardisation s'est fait sentir pour fournir à JavaScript les moyens de développer des applications web robustes.

Cette standardisation a été approuvé par l'organisme de normalisation en Juin 2015 : cela signifie que ce standard va être supporté de plus en plus par les navigateurs dans les temps à venir. ECMAScript 6 a été annoncé pour la première fois en 2008, et bientôt il deviendra inévitable.

Pour votre information, ECMAScript6, ECMAScript 2015 et ES6 désignent la même chose. Nous utiliserons ES6 comme nom, car c'est le surnom le plus populaire.

**A quoi sert ECMAScript 6 dans une application Angular ?**

ES6, via sa nouvelle syntaxe, apporte plusieurs nouvelles évolutions à JavaScript. Voici les éléments les plus intéressants que nous utiliserons dans le cadre de ce cours.

*PS: Pas besoin de tout retenir, survolez simplement la liste ci-dessous pour vous familiariser avec la syntaxe de ES6.*

**Les classes**

ES6 introduit une nouvelle syntaxe: le mot-clef *class*. C'est le même mot clé que dans d'autres langages, mais sachez que c'est toujours de l'héritage par prototype qui tourne derrière, mais vous n'avez plus à vous en soucier.

1. class Vehicule {
2. constructor(couleur, nombreRoueMotrice) {
3. this.couleur = couleur;
4. this.nombreRoueMotrice = nombreRoueMotrice;
5. this.moteurAllumer = false;
6. }
7. demarrer() {
8. this.moteurAllumer = true;
9. }
10. couperMoteur() {
11. this.moteurAllumer = false;
12. }
13. }

Voilà une classe JavaScript, bien différente de ce que l'on avait précédemment. Si vous avez bien remarqué, on a même droit à un constructeur. C’est pas mal pour du JavaScript ! :)

**Le nouveau mot-clef let**

Le mot-clé *let* permet de déclarer une variable locale, dans le contexte où elle a été assignée. (Un contexte est le terme français pour désigner un le *$scope* dans AngularJS, ou *scope* en anglais).

Par exemple, les instructions que vous écrivez dans le corps d'une fonction ou à l'extérieur n'ont pas le même contexte. Normalement une instruction *if* n'a pas de contexte en soi, mais maintenant si, avec le mot clé *let*. Cela peut être utile quand on veut effectuer beaucoup d'opérations sur une variable et que vous ne voulez pas polluer d'autres contextes des variables qui ne sont pas nécessaires dans d'autres contextes :

1. var x = 1;
3. if(x < 10) {
4. let v = 1;
5. v = v + 21;
6. console.log(v);
7. }
9. console.log(v); // v n'est pas définie, car v a été déclaré avec 'let' et non 'var'.

 En général, garder un contexte global propre est vivement conseillé, et c'est pourquoi ce mot clé *let* est vraiment le bienvenu ! Sachez que *let* a été pensé pour remplacer *var*, alors nous n'allons pas nous en priver dans ce cours.

**Le nouveau mot-clef const**

 Le mot clé const quant à lui, permet de déclarer ... des constantes ! Voyons comment cela fonctionne:

 const PI = 3.141592 ;

 Une déclaration de constante ne peut se faire qu'une fois, une fois définie, vous ne pouvez plus changer sa valeur.

 C'est bien pratique si vous avez besoin de définir des valeurs une bonne fois pour toutes dans votre application.

 *Attention, le comportement est un peu différent pour une constante de tableau ou d'objet. Vous ne pouvez pas modifier la référence vers le tableau ou l'objet, mais vous pouvez continuer à modifier les valeurs à l'intérieur tableau, ou les propriétés de l'objet.*

**Les fonctions fléchées**

 Les fonctions fléchées, ou *arrow functions*, ne sont pas des fonctions classiques, parce qu’**elles ne définissent pas un nouveau contexte** comme les fonctions traditionnelles. Nous les utiliserons beaucoup dans le cadre de la programmation asynchrone qui nécessite beaucoup de fonctions anonymes.

 La structure d'une fonction fléchée est toujours la même: **(paramètre) => { corps de la fonction }**.

 La fonction suivante:

1. bouton.onclick = function() {
2. envoyerMail(this.email);
3. }

 Correspond donc à la fonction fléchée suivante:

 bouton.onclick = () => { envoyerEmail(this.email); }

 Je vous conseille de vous habituer dès maintenant à cette nouvelle syntaxe, et nous allons l'utiliser tout le temps !

**Les paramètres de fonctions par défaut**

 En JavaScript ES6, on peut définir facilement des paramètres de fonctions avec une valeur par défaut.

 Imaginons une fonction qui multiplie deux nombres passés en paramètres, mais le deuxième paramètre est facultatif, et il vaut 1 par défaut:

1. function multiplier(a, b = 1) {
2. return a \* b;
3. }

 Avec ES6, il suffit de définir une valeur par défaut dans la signature même de la fonction. C'est très pratique !

**La syntaxe *template string***

 La concaténation de chaîne de caractère a toujours été pénible avec JavaScript.

Mais avec ES6, on peut utiliser des *templates strings*, qui commencent et se terminent par un *backtick* (`). Il s'agit d'un symbole particulier qui permet d'insérer facilement des variables dans des chaînes de caractères. On peut insérer des variables dans la chaîne de caractères avec *${variable}*, comme ceci:

1.  let name = 'toto';
2. let email = 'toto@gmail.com';
3. console.info(‘${name} a pour email: ${email}’);

 C'est quand même plus pratique que de concaténer des chaines de caractères entre elles !

**Conclusion**

Nous avons vu plusieurs changements majeurs apporté par ES6 à Javascript. Sachez que ES6 est rétro-compatible, donc vous pouvez toujours continuer à développer de la façon dont vous le faites actuellement, puis migrer petit à petit vers la syntaxe d'ES6. Je vous recommande d'adopter ES6 assez rapidement, vous gagnerez en productivité et en lisibilité avec cette nouvelle syntaxe.

Même si les nouveautés de ES6 ne fonctionnent pas encore dans certains navigateurs, beaucoup de développeurs ont commencé à développer avec ES6 et utilisent un transpilateur pour convertir leur code ES6 en du code ES5. Ainsi leur code est lisible par tout les navigateurs.

Un transpilateur est un outil qui permet de publier son code pour les navigateurs qui ne supportent pas encore l'ES6: le rôle du transpilateur est de traduire le code ES6 en code ES5. Comme certaines fonctionnalités d'ES6 ne sont pas disponibles avec ES5, leur comportement est simulé pour permettre aux navigateurs plus anciens de *comprendre* le code.

Dans la prochaine session, nous verrons que TypeScript permet de transpiler notre code ES6 en ES5, et ajoute une surcouche intéressante au JavaScript afin de pouvoir typer nos variables JavaScript. Et ça, c'est assez nouveau dans le monde du JavaScript !

**Angular et TypeScript**

**C'est quoi, TypeScript ?**

Nous allons aborder un des piliers d'Angular : *TypeScript*. Certains d'entre vous en ont peut-être déjà entendu parler, mais les autres ne vous en faites pas, nous allons démystifier tout cela immédiatement !

Avant de voir plus en détail ce que nous allons faire avec ce langage, et parce que je ne suis pas toujours inspiré, je vous propose de commencer par la définition de Wikipédia:

TypeScript est un langage de programmation libre et open-source, développé par Microsoft, qui a pour but d'améliorer et de sécuriser la production de code JavaScript. C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code Javascript correct peut être utilisé avec TypeScript). Le code TypeScript est transcompilé en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.

Je n'aurai pas été plus clair. Mais je vous rassure on va creuser tout ça tout de suite, je ne vais pas vous laisser avec une simple définition de Wikipédia.

Les faiblesses du langage JavaScript font que lorsque l'on commence à écrire beaucoup de code, on arrive vite à un moment où on s'emmêle les pinceaux ! Notre code devient redondant, perd en élégance, et devient de moins en moins lisible : on parle de *code spaghetti*. Vous comprendrez tous seuls la référence aux célèbres pâtes italiennes ! ;)

La communauté des développeurs, ainsi que certaines entreprises, se sont mis alors à développer des méta-langages pour *améliorer* JavaScript : *CoffeeJS*, *Dart* et *TypeScript* en sont les exemples les plus réussis. Ces outils apportent également de nouvelles fonctionnalités qui font défaut au JavaScript natif, avec une nouvelle syntaxe moins verbeuse.

TypeScript permet de *typer* nos variables, ce qui permet d'écrire du code plus robuste. Une fois que vous avez développé votre application, vous devez le compiler, c'est-à-dire le transformer en du code JavaScript que le navigateur pourra interpréter. En effet, votre navigateur ne sait pas interpréter le TypeScript. Vous devez d'abord compilez ce code en JavaScript pour qu'il soit lisible par votre navigateur.

*Attention, l'extension des fichiers TypeScript est .****ts*** *et non .****js*** *!*

**A quoi sert TypeScript dans une application Angular ?**

Il est chaudement recommandé d'utiliser TypeScript pour développer en Angular.  C'est ce que Google recommande explicitement dans la documentation officielle d'Angular.

En bonus, Angular lui-même est développer avec TypeScript. C'est pour cela que nous utiliserons TypeScript dans ce cours !

**Le typage avec TypeScript**

Voici comment typer nos variables avec TypeScript:

1. var pointDeVie: number = 100;
2. var surnom: string = 'Green Lantern';

Comme vous pouvez le constater, il suffit d'ajouter deux points et le type de votre variable pour typer une variable en TypeScript. J'ai jouté en ressource la liste de tous les types de base disponibles avec TypeScript.

**TypeScript et les fonctions**

On peut aussi typer les **paramètres** et la **valeur de retour** d'une fonction, toujours avec la syntaxe des deux points:

1. function creerHeros(pointDeVie: number, surnom: string): Heros {
2. var heros = new Heros();
3. heros.pointDeVie = pointDeVie;
4. heros.surnom = surnom;
5. return heros;
6. }

**Les décorateurs**

Les annotations TypeScript permettent d'ajouter des informations sur nos classes, pour indiquer par exemple que telle classe est un composant de l'application, ou telle autre un service. On utilise @ comme syntaxe:

1. @Component({
2. selector: 'mon-composant',
3. template: 'mon-template.html'
4. })
5. export class MonComposant {}

On utilisera beaucoup cette syntaxe lors de nos développements avec Angular.

**Conclusion**

TypeScript apporte évidemment d'autres fonctionnalités : les valeurs énumérées, les interfaces, la généricité, etc.

TypeScript est donc un méta-langage qui est surtout connu pour apporter le typage à JavaScript, mais il s'agit également d'un transpilateur, capable de générer du code vers ES5 ou ES6 !

L'objectif de cette section était que vous preniez rapidement connaissance de TypeScript, pour voir ensuite son fonctionnement concret dans une application Angular. Si vous voulez pousser plus loin vos connaissances en TypeScript, je vous invite à regarder la documentation officielle que j'ai ajouté en ressource de cette session.

Angular et les Composants Web

**C'est quoi, les Web Components ?**

Les Web Components désignent un standard qui permet aux développeurs de créer des sections complètement autonomes au sein de leurs pages web. On peut par exemple créer un composant web qui gère l'affichage d'articles dans notre application: ce composant web fonctionnera indépendamment du reste de la page, il possèdera son propre code HTML, son propre code CSS et son propre code JavaScript, encapsulé dans le composant web. Il faudra ensuite insérer ce composant web dans la page principale de notre application pour indiquer que, à tel endroit, nous voulons afficher des articles grâce à ce composant web.

*On peut voir les Web Components comme des widgets réutilisables.*

**Angular et les Web Components**

Les Web Components utilisent des capacités standards, nouvelles ou en cours de développements, des navigateurs. Il s'agit de technologies récentes qui ne sont pas encore supportées par tous les navigateurs. Pour les utiliser dès aujourd'hui, nous devrons utiliser des *polyfills* pour combler les lacunes de couverture des navigateurs.

*Un polyfill est un ensemble de fonction, souvent sous forme de scripts JavaScript, permettant de simuler sur un navigateur web ancien des fonctionnalités qui ne sont pas nativement disponibles.*

**Les Web Components sont composés de quatre technologies différentes**, qui peuvent chacune être utilisées séparément, mais qui une fois assemblées forme le standard des Web Components :

1. Les éléments personnalisés (*Custom Elements*) permettent de créer ses propres éléments HTML valides.
2. Le DOM de l'ombre (*Shadow DOM*) permet d'encapsuler du code HTML, CSS et JavaScript qui n'interfère pas avec le DOM principal de la page web.
3. Les templates HTML (*HTML Templates*) permettent de développer des morceaux de code HTML qui ne sont pas interprétés au chargement de la page.
4. Les imports HTML (*HTML Imports*) permettent d'importer du HTML dans une autre page HTML.

*Pour ceux qui ne s'en souviennent plus, le DOM est une représentation de votre code HTML sous forme d'arbre. Ainsi un élément <ul>  a des éléments fils <li> . L'élément racine du DOM est donc la balise <html> .*

**Conclusion**

Je ne vais pas vous présenter chacune de ces technologies pour le moment, le plus important est que vous compreniez pourquoi telle ou telle technologie a été créer, et à quoi elle sert. Retenez que les composants web permettent d'encapsuler une partie de votre page dans un composant, et que les composants Angular repose sur ce standard des Composants Web.

**En résumé**

1. Les sites web deviennent de plus en plus de véritables applications, et une utilisation intensive du langage JavaScript devient nécessaire.
2. Angular est un Framework orienté composant, car votre application entière est un assemblage de composants.
3. Angular est conçu pour le web de demain et intègre déjà la norme *ECMAScript6* (ou ES6) et les *Web Components*.
4. Enfin, retenez que tout cet écosystème est bourgeonnant et change (très) vite. Soyez persévérant lors de votre apprentissage et ne vous découragez pas. Vous prendrez vite plaisir à utiliser Angular !

**Avant de continuer : Angular 6**

Dans la suite de ce cours, tous les [extraits de code](https://awesome-angular.com/ebook/#first-title) et la [correction de l'application](https://github.com/moumoune/ng6-pokemons-app) sont à jour pour Angular 6.

Cependant, la plupart des vidéos ont été tournées pour la version 5 d'Angular. Il y a donc quelques vidéos dans la session *Premiers pas avec Angular* et *Effectuer des requêtes HTTP* où j'utilise du code adapté uniquement pour la version 5 d'Angular.

Je vous invite donc simplement à récupérer la version plus récente des fichiers concernés, qui sont indiqués en ressource de chaque vidéo de la formation. Je mettrai prochainement les vidéos concernées à jour, et je vous tiendrai informé grâce aux annonces Udemy.

Installer NodeJS et NPM

Pour installer les dépendances de notre projet Angular que nous avons renseigné dans le fichier package.json, nous allons utiliser un petit outil bien pratique pour cela, nommé NPM (Node Package Manager).

Pour cela, il est nécessaire d'installer NodeJS, qui permet d'exécuter du code javaScript côté serveur. NodeJS ne nous sert pas directement dans cette formation, mais il permet d'installer NPM, qui va nous permettre d'exécuter des commandes "npm ..." sur notre machine. 👨🏻‍💻

Voici le lien pour télécharger NodeJS sur Windows, Mac ou Linux :

👉 [https://nodejs.org/fr/download](https://nodejs.org/fr/download/)

Dès que l'installation est terminée sur votre poste, vous êtes prêt à passer à la suite !

Cas où votre application ne démarre pas

#### Si l'application ne démarre pas...

Si votre application ne démarre pas correctement lorsque vous lancer la commande npm start, essayez les solutions suivantes :

- Placer le fichier main.ts dans le dossier src.

- Vérifier votre fichier [package.json](https://gist.github.com/moumoune/4823306afa00b6b873243fc18dd247e0#file-package-6-0-0-json).

- Supprimer le dossier node\_modules, et réinstaller vos dépendances avec la commande npm install.

- Lancer la commande npm run build:watch, puis dans un autre terminal, lancer la commande npm run serve.

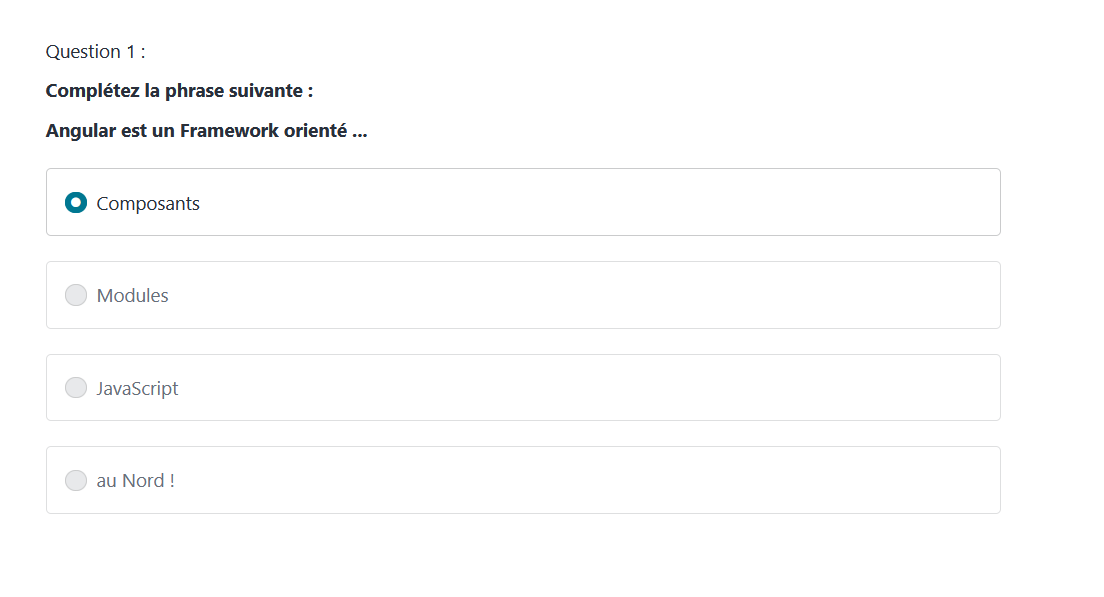
J'espère que cela corrigera votre erreur.

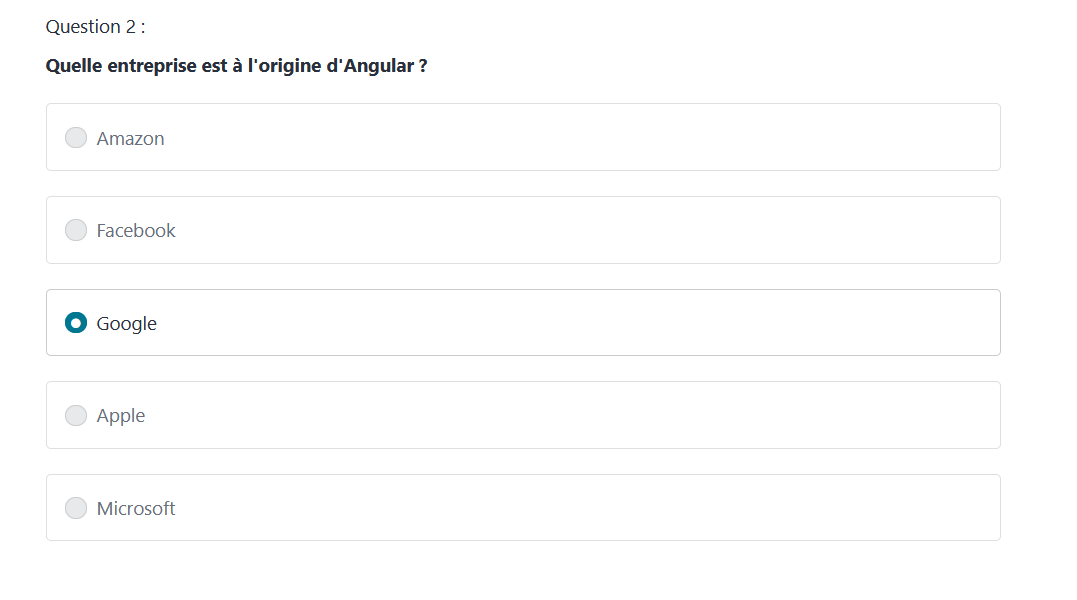
Si ça n'est toujours pas le cas, vous pouvez récupérer la correction dans les ressources de la correction. Ensuite lancez les commandes **npm install** et **npm start**, et vous devriez avoir un environnement de travail prêt pour continuer la formation. :)

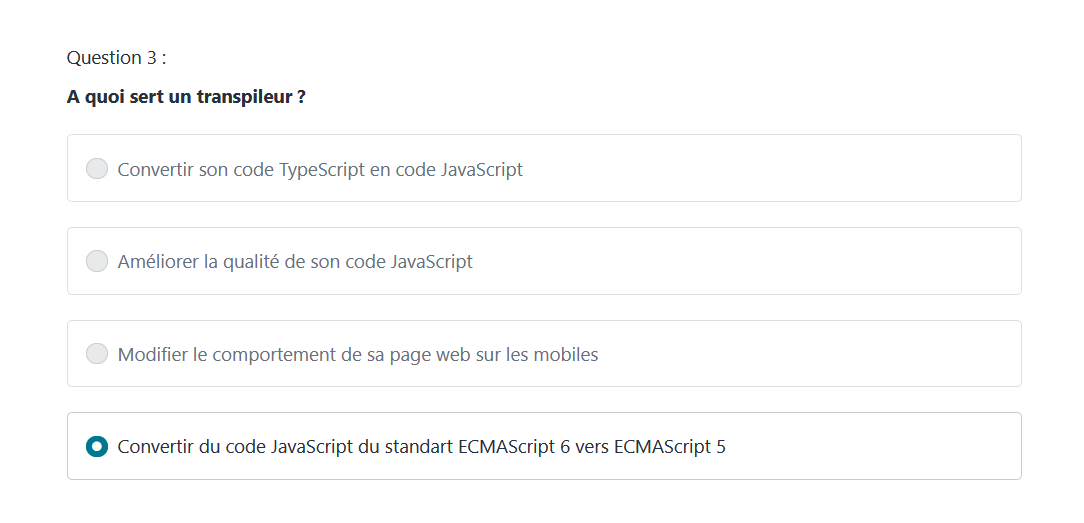
**En résumé**

1. SystemJS est la bibliothèque par défaut choisie par Angular pour charger les modules.
2. On a besoin au minimum d'un module racine et d'un composant racine par application.
3. Le module racine se nomme par convention *AppModule*.
4. Le composant racine se nomme par convention *AppComponent*.
5. L'ordre de chargement de l'application est le suivant : *index.html* > *main.ts* > *app.module.ts* > *app.component.ts*.
6. Le fichier *package.json* initial est fourni avec des commandes prêtes à l'emploi comme la commande *npm start*, qui nous permet de démarrer notre application web.

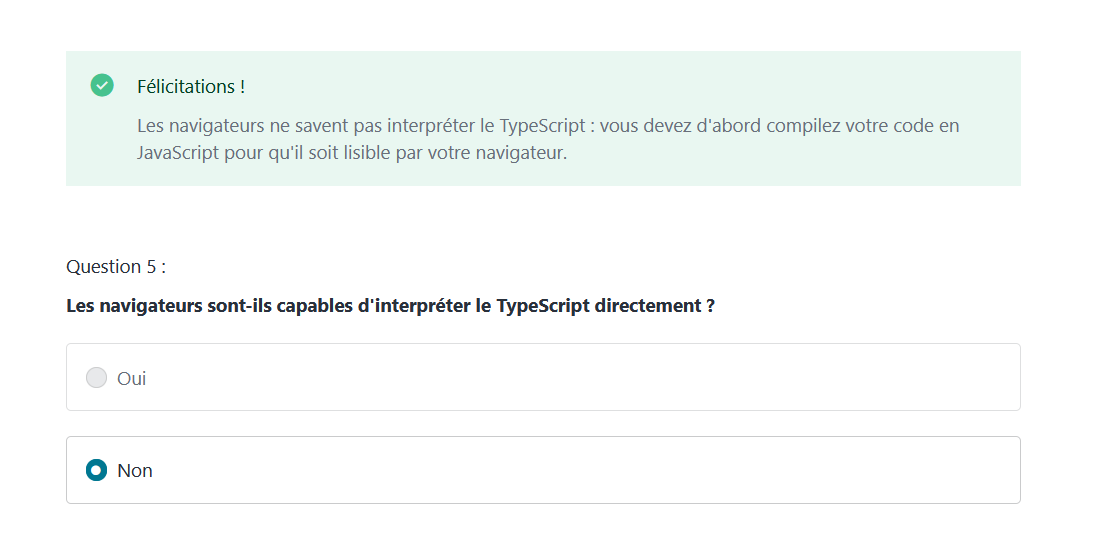
**Quizz**

****

****

****

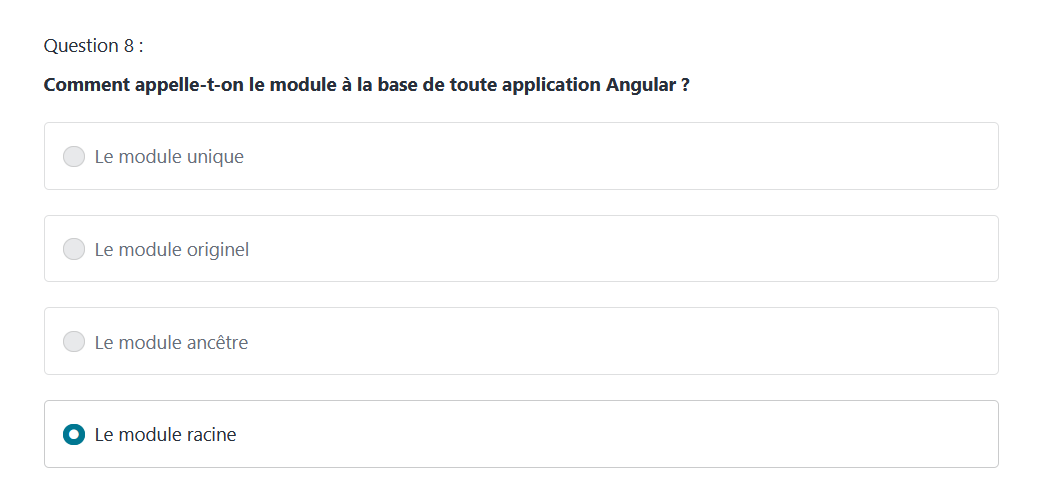
****

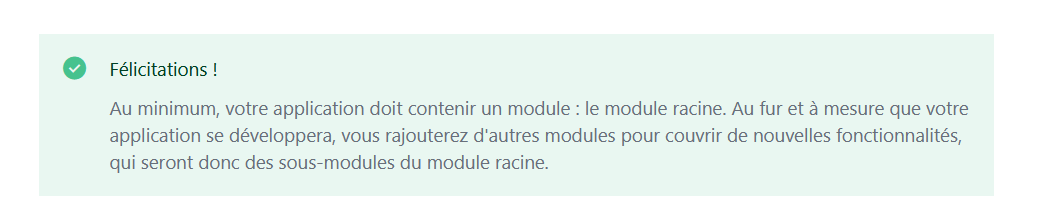
****

****

****

****

****

****

Ce que vous savez

* [Quelle entreprise est à l'origine d'Angular ?](javascript:void(0))
* [Quel est le langage recommandé pour le développement d'applications Angular ?](javascript:void(0))
* [Quel est le standard ci-dessous qui ne fait pas partie du standard des WebComponents ?](javascript:void(0))
* [Comment appelle-t-on le module à la base de toute application Angular ?](javascript:void(0))

Ce que vous devriez revoir

* [Complétez la phrase suivante :Angular est un Framework orienté ...](javascript:void(0))
* [A quoi sert un transpileur ?](javascript:void(0))
* [Les navigateurs sont-ils capables d'interpréter le TypeScript directement ?](javascript:void(0))
* [Est-il possible d'avoir plusieurs DOM de l'ombre sur une même page web ?](javascript:void(0))

**Les cycles de vie d'un composant**

Chaque composant a un cycle de vie géré par Angular lui-même. Angular crée le composant, s'occupe de l'afficher, crée et affiche ses composants fils, vérifie quand les données des propriétés changent, et détruit les composants, avant de les retirer du DOM quand cela est nécessaire. Pratique, non ?

Angular nous offre la possibilité d'agir sur ces moments clefs quand ils se produisent, en implémentant une ou plusieurs interfaces, pour chacun des événements disponibles. Ces interfaces sont disponibles dans la librairie *core* d'Angular.

Chaque interface permettant d'interagir sur le cycle de vie d'un composant fournit une et une seule méthode, dont le nom est le nom de l'interface préfixé par *ng*. Par exemple, l'interface *OnInit* fournit la méthode *ngOnInit*, et permet de définir un comportement lorsque le composant est initialisé.

Voici ci-dessous la liste des méthodes disponibles pour interagir avec le cycle de vie d'un composant, dans l'ordre chronologique du moment où elles sont appelées par Angular.

1. **ngOnChanges**: *C*'est la méthode appelée en premier lors de la création d'un composant, avant même *ngOnInit*, et à chaque fois que Angular détecte que les valeurs d'une propriété du composant sont modifiées. La méthode reçoit en paramètre un objet représentant les valeurs actuelles et les valeurs précédentes disponibles pour ce composant.
2. **ngOnInit**: Cette méthode est appelée juste après le premier appel à *ngOnChanges*, et elle initialise le composant après qu’Angular ait initialisé les propriétés du composant.
3. **ngDoCheck**: On peut implémenter cette interface pour étendre le comportement par défaut de la méthode *ngOnChanges*, afin de pouvoir détecter et agir sur des changements qu’Angular ne peut pas détecter par lui-même.
4. **ngAfterViewInit**: Cette méthode est appelée juste après la mise en place de la vue d'un composant, et des vues de ses composants fils s'il en a.
5. **ngOnDestroy**: Appelée en dernier, cette méthode est appelée avant qu’Angular ne détruise et ne retire du DOM le composant. Cela peut se produire lorsqu'un utilisateur navigue d'un composant à un autre par exemple. Afin d'éviter les fuites de mémoire, c'est dans cette méthode que nous effectuerons un certain nombre d'opérations afin de laisser l'application "propre" (nous détacherons les gestionnaires d'événements par exemple).

Même si vous ne définissez pas explicitement des méthodes de cycle de vie dans votre composant, sachez que ces méthodes sont appelées en interne par Angular pour chaque composant. Lorsqu'on utilise ces méthodes, on vient donc juste surcharger tel ou tel événement du cycle de vie d'un composant.

Les méthodes que vous utiliserez le plus seront certainement *ngOnInit* et *ngOnDestroy*, qui permettent d'initialiser un composant, et de le nettoyer proprement par la suite lorsqu'il est détruit.

Cas où votre application ne démarre pas

#### Si l'application ne démarre pas...

Si votre application ne démarre pas correctement lorsque vous lancer la commande npm start, essayez les solutions suivantes :

- Placer le fichier main.ts dans le dossier src.

- Vérifier votre fichier [package.json](https://gist.github.com/moumoune/4823306afa00b6b873243fc18dd247e0#file-package-6-0-0-json).

- Supprimer le dossier node\_modules, et réinstaller vos dépendances avec la commande npm install.

- Lancer la commande npm run build:watch, puis dans un autre terminal, lancer la commande npm run serve.

J'espère que cela corrigera votre erreur.

Si ça n'est toujours pas le cas, vous pouvez récupérer la correction dans les ressources de la correction. Ensuite lancez les commandes **npm install** et **npm start**, et vous devriez avoir un environnement de travail prêt pour continuer la formation. :)

En résumé

**En résumé**

1. SystemJS est la bibliothèque par défaut choisie par Angular pour charger les modules.
2. On a besoin au minimum d'un module racine et d'un composant racine par application.
3. Le module racine se nomme par convention *AppModule*.
4. Le composant racine se nomme par convention *AppComponent*.
5. L'ordre de chargement de l'application est le suivant : *index.html* > *main.ts* > *app.module.ts* > *app.component.ts*.
6. Le fichier *package.json* initial est fourni avec des commandes prêtes à l'emploi comme la commande *npm start*, qui nous permet de démarrer notre application web.

Plein écran

29. Conclusion

Quiz 1 : Questionnaire n°1

Les cycles de vie d'un composant

**Les cycles de vie d'un composant**

Chaque composant a un cycle de vie géré par Angular lui-même. Angular crée le composant, s'occupe de l'afficher, crée et affiche ses composants fils, vérifie quand les données des propriétés changent, et détruit les composants, avant de les retirer du DOM quand cela est nécessaire. Pratique, non ?

Angular nous offre la possibilité d'agir sur ces moments clefs quand ils se produisent, en implémentant une ou plusieurs interfaces, pour chacun des événements disponibles. Ces interfaces sont disponibles dans la librairie *core* d'Angular.

Chaque interface permettant d'interagir sur le cycle de vie d'un composant fournit une et une seule méthode, dont le nom est le nom de l'interface préfixé par *ng*. Par exemple, l'interface *OnInit* fournit la méthode *ngOnInit*, et permet de définir un comportement lorsque le composant est initialisé.

Voici ci-dessous la liste des méthodes disponibles pour interagir avec le cycle de vie d'un composant, dans l'ordre chronologique du moment où elles sont appelées par Angular.

1. **ngOnChanges**: *C*'est la méthode appelée en premier lors de la création d'un composant, avant même *ngOnInit*, et à chaque fois que Angular détecte que les valeurs d'une propriété du composant sont modifiées. La méthode reçoit en paramètre un objet représentant les valeurs actuelles et les valeurs précédentes disponibles pour ce composant.
2. **ngOnInit**: Cette méthode est appelée juste après le premier appel à *ngOnChanges*, et elle initialise le composant après qu’Angular ait initialisé les propriétés du composant.
3. **ngDoCheck**: On peut implémenter cette interface pour étendre le comportement par défaut de la méthode *ngOnChanges*, afin de pouvoir détecter et agir sur des changements qu’Angular ne peut pas détecter par lui-même.
4. **ngAfterViewInit**: Cette méthode est appelée juste après la mise en place de la vue d'un composant, et des vues de ses composants fils s'il en a.
5. **ngOnDestroy**: Appelée en dernier, cette méthode est appelée avant qu’Angular ne détruise et ne retire du DOM le composant. Cela peut se produire lorsqu'un utilisateur navigue d'un composant à un autre par exemple. Afin d'éviter les fuites de mémoire, c'est dans cette méthode que nous effectuerons un certain nombre d'opérations afin de laisser l'application "propre" (nous détacherons les gestionnaires d'événements par exemple).

Même si vous ne définissez pas explicitement des méthodes de cycle de vie dans votre composant, sachez que ces méthodes sont appelées en interne par Angular pour chaque composant. Lorsqu'on utilise ces méthodes, on vient donc juste surcharger tel ou tel événement du cycle de vie d'un composant.

Les méthodes que vous utiliserez le plus seront certainement *ngOnInit* et *ngOnDestroy*, qui permettent d'initialiser un composant, et de le nettoyer proprement par la suite lorsqu'il est détruit.

Exercice

#### Un peu de pratique !

Bon passons à la pratique, et essayons d'enrichir quelque peu le composant app.component.ts de notre application. Je propose d'ajouter deux fonctionnalités que nous venons de voir:

1. Injecter une liste de Pokémons dans le composant.
2. Préparer une méthode qui devra gérer l'action lorsque l'utilisateur cliquera sur un Pokémon de cette liste.

Mais avant de vous laisser chercher tout seul, je me dois de vous donner deux choses :

1. Une classe permettant de modéliser un Pokémon.
2. Un fichier contenant quelques Pokémons à injecter dans notre composant, à titre d'exemple.

#### Le modèle

Créer un fichier pokemon.ts dans le dossier app de notre projet, avec le code suivant :

1. export class Pokemon {
2. id: number;
3. hp: number;
4. cp: number;
5. name: string;
6. picture: string;
7. types: Array<string>;
8. created: Date;
9. }

Cette classe a pour rôle de représenter un Pokémon dans notre application. Chaque Pokémon aura donc :

1. **Id**: un identifiant unique sous forme de nombre.
2. **Hp**: le nombre de points de vie du Pokémon.
3. **Cp**: le nombre de dégâts d'un Pokémon.
4. **Name**: un nom.
5. **Picture**: l'url d'une image représentant le Pokémon.
6. **Types**: un tableau contenant les types du Pokémon (Eau, Feu, Vol, etc...).
7. **Created**: la date de création du Pokémon.

#### Les données

Maintenant, créons un fichier mock-pokemons.ts qui contiendra les données de plusieurs Pokémons. Ce fichier doit être placé dans le dossier src/app (le fichier est un peu long, mais il s'agit simplement de données mises à disposition pour notre application). Etant donné que nous n’allons pas modifier ce fichier dans la suite du cours, je vous propose de le copier directement à partir de [l’adresse suivante](https://gist.github.com/moumoune/112d6e83544a8fb237e0e0a30b42f58a#file-mock-pokemons-ts).

Comme vous pouvez le constater, ce fichier ne fait qu'exporter la constante POKEMONS, qui contient des données nécessaires pour notre application.

Maintenant, vous avez tous les éléments nécessaires pour améliorer le composant app.component.ts de notre application. Pour rappel, voici ce que vous devez faire :

1. Injecter une liste de Pokémons dans le composant.
2. Préparer une méthode qui devra gérer l'action lorsque l'utilisateur cliquera sur un Pokémon de cette liste.

A vous de jouer maintenant !

Syntaxe de liaison des données

#### Syntaxe de liaison des données

L'interpolation est très pratique pour lier notre template et notre composant. Cependant, il existe d'autres manières de créer des liaisons entre le composant et le template. L'interpolation a une particularité : elle pousse les données depuis le composant vers le template, mais ce n'est pas le seul moyen de faire. Nous allons voir les autres types de liaisons possibles.

#### Du composant vers le template

Nous pouvons pousser plusieurs données depuis le composant vers le template. Voici les liaisons que nous pouvons mettre en place :

**La liaison sur une propriété d'élément**

On utilise les crochets pour lier directement la source de l'image à la propriété someImageUrl du composant :

*<img [src]="someImageUrl">*

**La liaison sur une propriété d'attribut**

On lie l'attribut for de l'élément label avec la propriété de notre composant someLabelId :

<label [attr.for]="someLabelId">...</label>

**La liaison sur une propriété de la classe**

Fonctionnement similaire, pour attribuer ou non la classe special à l'élément div ci-dessous :

<div [class.special]="isSpecial">Special</div>

**La liaison sur une propriété de style**

On peut également définir un style pour nos éléments de manière dynamique : ici on définit la couleur de notre bouton en fonction de la propriété isSpecial, soit rouge, soit vert. C'est un [opérateur ternaire](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Op%C3%A9rateurs/L_op%C3%A9rateur_conditionnel) que l'on utilise comme expression :

<button [style.color]="isSpecial ? 'red' : 'green'">

Note : Nous verrons les liaisons bidirectionnelles dans le chapitre dédié sur les Formulaires, où nous en auront vraiment besoin. Cela permet de décharger un peu ce chapitre.

En résumé

**En résumé**

1. L'interpolation permet d'afficher les propriétés de nos composants dans les templates, via la syntaxe *{‌{ }}*.
2. On peut lier une propriété d'élément, d'attribut, de classe ou de style d'un composant vers le template.
3. Si nos templates sont trop long, on peut utiliser le *backtick* d'ES6, ou définir nos templates dans des fichiers séparés.
4. La directive *NgIf* permet de conditionner l'affichage d'un template.
5. La directive *NgFor* permet d'afficher une propriété de type tableau dans un template.
6. On peut gérer les interactions d'un utilisateur avec un élément de la page grâce à la syntaxe : *'(' + 'nom de l'événement' + ')'*.
7. On peut référencer des variables directement dans le template plutôt que de manipuler l'objet *$event*.
8. Les variables référencées dans le template sont accessibles pour tous les éléments fils et frères de l'élément dans lequel elle ont été déclarées.
9. Essayez d'éviter de mettre la logique de votre application dans vos templates. Gardez-les le plus simple possible !

En résumé

**En résumé**

1. On utilise l'annotation *@Directive* pour déclarer une directive dans notre application.
2. Il existe trois types de directives différentes : les composants, les directives d'attributs et les directives structurelles (*ngFor* et *ngIf* par exemple).
3. Une directive d'attribut permet d'agir avec les éléments HTML d'une page, en leur attachant un comportement spécifique.
4. Une directive utilise un sélecteur CSS pour cibler les éléments HTML sur lesquels elle s’applique.
5. Il est recommandé de préfixer le nom de ses directives pour éviter les problèmes de collisions.
6. Angular crée une nouvelle instance de notre directive à chaque fois qu'il détecte un élément HTML avec l'attribut correspondant. Il injecte alors dans le constructeur de la directive l'élément du DOM *ElementRef*.
7. Il faut déclarer notre directive pour pouvoir l’utiliser.
8. On utilise l'annotation *@HostListener* pour gérer les interactions de l'utilisateur au sein d'une directive.

**En résumé**

1. Les *pipes* permettent de formater les données affichées dans nos templates.
2. L'opérateur des pipes est « *|* ».
3. Angular fournit des pipes prêts à l'emploi, disponibles dans tous les templates de notre application : *DatePipe*, *UpperCasePipe*, *LowerCasePipe*, etc.
4. Les *pipes* peuvent avoir des paramètres, mais tous les paramètres sont facultatifs.
5. On peut créer des *pipes* personnalisés pour les besoins de notre application avec l'annotation *@Pipe*.
6. Les *pipes* personnalisés doivent être déclarés avant de pouvoir être utilisés dans les templates de composants.

En résumé

**En résumé**

1. Il existe deux types de modules : le *module racine* et les *modules de fonctionnalité*, appelés également sous-modules.
2. On déclare un module avec l'annotation *@NgModule*, quel que soit le type du module.
3. On peut créer des applications complexes en ajoutant des modules de fonctionnalité au module racine.
4. Chaque module regroupe tous les composants, directives, pipes, services, ... liés au développement d'une fonctionnalité donnée, dans un dossier à part.
5. Chaque module peut disposer de ses propres routes également.
6. On définit les routes de nos sous-modules avec *forChild*, et *forRoot* pour le module racine.
7. Modifier l'architecture d'une application existante peut être pénible, il est préférable de prévoir l'architecture des modules de votre application à l'avance.