Become a

NINJA

with

ANGULAR 2

ninja squad

Deviens un ninja avec Angular2 (extrait gratuit)

Ninja Squad

Table of Contents

Chapter 1. Extrait gratuit

Ce que tu vas lire ici est un extrait gratuit : c'est le début du livre, qui explique son but et son contenu, donne un aperçu d'ECMAScript 6, TypeScript, et des Web Components, décrit la philosophie d'Angular 2, puis te propose de construire ta première application.

Cet extrait ne demande aucune connaissance préliminaire.

Chapter 2. Introduction

Alors comme ça on veut devenir un ninja ?! Ça tombe bien, tu es entre de bonnes mains !

Mais pour y parvenir, nous avons un bon bout de chemin à parcourir ensemble, semé d'embûches et de connaissances à acquérir :).

On vit une époque excitante pour le développement web. Il y a un nouvel Angular, une réécriture complète de ce bon vieil AngularJS. Pourquoi une réécriture complète ? AngularJS 1.x ne suffisait-il donc pas ?

J'adore cet ancien AngularJS. Dans notre petite entreprise, on l'a utilisé pour construire plusieurs projets, on a contribué du code au cœur du framework, on a formé des centaines de développeurs (oui, des centaines, littéralement), et on a même écrit un livre sur le sujet.

AngularJS est incroyablement productif une fois maîtrisé. Mais cela ne nous empêche pas de constater ses faiblesses. AngularJS n'est pas parfait, avec des concepts très difficiles à cerner, et des pièges durs à éviter.

Et qui plus est, le web a bien évolué depuis qu'AngularJS a été conçu. JavaScript a changé. De nouveaux frameworks sont apparus, avec de belles idées, ou de meilleures implémentations. Nous ne sommes pas le genre de développeurs à te conjurer d'utiliser tel outil plutôt que tel autre. Nous connaissons juste très bien quelques outils, et savons ce qui peut correspondre au projet. AngularJS était un de ces outils, qui nous permettait de construire des applications web bien testées, et de les construire vite. On a aussi essayé de le plier quand il n'était pas forcément l'outil idéal. Merci de ne pas nous condamner, ça arrive aux meilleurs d'entre nous, n'est-ce pas ? ;p

Angular 2 sera-t-il l'outil que l'on utilisera sans aucune hésitation dans nos projets futurs ? C'est dur à dire pour le moment, parce que ce framework est encore tout jeune, et que son écosystème est à peine bourgeonnant.

En tout cas, Angular 2 a beaucoup de points positifs, et une vision dont peu de frameworks peuvent se targuer. Il a été conçu pour le web de demain, avec ECMAScript 6, les Web Components, et le mobile en tête. Quand il a été annoncé, j'ai d'abord été triste, comme beaucoup de gens, que cette version 2.0 n'allait pas être une simple évolution (et désolé si tu viens de l'apprendre).

Mais j'étais aussi très curieux de voir quelles idées allait apporter la talentueuse équipe de Google.

Alors j'ai commencé à écrire ce livre, dès les premiers commits, lisant les documents de conception, regardant les vidéos de conférences, et analysant chaque commit depuis le début. J'avais écrit mon premier livre sur AngularJS 1.x quand c'était déjà un animal connu et bien apprivoisé. Ce livre-ci est très différent, commencé quand rien n'était encore clair dans la tête même des concepteurs. Parce que je savais que j'allais apprendre beaucoup, sur Angular évidemment, mais aussi sur les concepts qui allaient définir le futur du développement web, et certains n'ont rien à voir avec Angular. Et ce fut le cas. J'ai du creuser pas mal, et j'espère que tu vas apprécier revivre ces découvertes avec moi, et comprendre comment ces concepts s'articulent avec Angular.

L'ambition de cet ebook est d'évoluer avec Angular. S'il s'avère qu'Angular devient le grand framework qu'on espère, tu en recevras des mises à jour avec des bonnes pratiques et de nouvelles fonctionnalités quand elles émergeront (et avec moins de fautes de frappe, parce qu'il en reste probablement malgré nos nombreuses relectures…). Et j'adorerais avoir tes retours, si certains chapitres ne sont pas assez clairs, si tu as repéré une erreur, ou si tu as une meilleure solution pour certains points.

Je suis cependant assez confiant sur nos exemples de code, parce qu'ils sont extraits d'un vrai projet, et sont couverts par des centaines de tests unitaires. C'était la seule façon d'écrire un livre sur un framework en gestation, et de repérer les problèmes qui arrivaient inévitablement avec chaque release.

Même si au final tu n'es pas convaincu par Angular, je suis à peu près sûr que tu vas apprendre deux-trois trucs en chemin.

Si tu as acheté le "pack pro" (merci !), tu pourras construire une petite application morceau par morceau, tout au long du livre. Cette application s'appelle **PonyRacer**, c'est un site web où tu peux parier sur des courses de poneys. Tu peux même tester cette application ici ! Vas-y, je t'attend.

Cool, non ?

Mais en plus d'être super cool, c'est une application complète. Tu devras écrire des composants, des formulaires, des tests, tu devras utiliser le routeur, appeler une API HTTP (fournie), et même faire des Web Sockets. Elle intègre tous les morceaux dont tu auras besoin pour construire une vraie application.

Chaque exercice viendra avec son squelette, un ensemble d'instructions et quelques tests. Quand tu auras tous les tests en succès, tu auras terminé l'exercice !

Si tu n'as pas acheté le "pack pro" (tu devrais), ne t'inquiète pas : tu apprendras tout ce dont tu auras besoin. Mais tu ne construiras pas cette application incroyable avec de beaux poneys en pixel art. Quel dommage :)!

Tu te rendras vite compte qu'au-delà d'Angular, nous avons essayé d'expliquer les concepts au cœur du framework. Les premiers chapitres ne parlent même pas d'Angular : ce sont ceux que j'appelle les "chapitres conceptuels", ils te permettront de monter en puissance avec les nouveautés intéressantes de notre domaine.

Ensuite, nous construirons progressivement notre connaissance du framework, avec les composants, les templates, les *pipes*, les formulaires, http, le routeur, les tests…

Et enfin, nous nous attaquerons à quelques sujets avancés. Mais c'est une autre histoire.

Passons cette trop longue introduction, et jetons-nous sur un sujet qui va définitivement changer notre façon de coder : ECMAScript 6.

Chapter 3. Une rapide introduction à ECMASCRIPT 6

Si tu lis ce livre, on peut imaginer que tu as déjà entendu parler de JavaScript. Ce qu'on appelle JavaScript (JS) est une des implémentations d'une spécification standardisée, appelée ECMAScript. La version de la spécification que tu connais le plus est probablement la version 5 : c'est celle utilisée depuis de nombreuses années.

Depuis quelques temps, une nouvelle version de cette spécification est en travaux : ECMASCRIPT 6, ES6, ou ECMASCRIPT 2015. Je l'appellerai désormais systématiquement ES6, parce que c'est son petit nom le plus populaire. Elle ajoute une tonne de fonctionnalités à JavaScript, comme les classes, les constantes, les *arrow functions*, les générateurs… Il y a tellement de choses qu'on ne peut pas tout couvrir, sauf à y consacrer entièrement ce livre. Mais Angular 2 a été conçu pour bénéficier de cette nouvelle version de JavaScript. Même si tu peux toujours utiliser ton bon vieux JavaScript, tu auras plein d'avantages à utiliser ES6. Ainsi, nous allons consacrer ce chapitre à découvrir ES6, et voir comment il peut nous être utile pour construire une application Angular 2.

On va laisser beaucoup d'aspects de côté, et on ne sera pas exhaustifs sur ce qu'on verra. Si tu connais déjà ES6, tu peux directement sauter ce chapitre. Sinon, tu vas apprendre des trucs plutôt incroyables qui te serviront à l'avenir même si tu n'utilises finalement pas Angular !

3.1. Transpileur

ES6 vient d'atteindre son état final, il n'est pas encore supporté par tous les navigateurs. Et bien sûr, certains navigateurs vont être en retard (est-il vraiment nécessaire de les nommer ? :p). Ainsi, à quoi bon présenter cela s'il faut être prudent sur ce qu'on peut utiliser ou non ? Et tu as raison, car rares sont les applications qui peuvent se permettre d'ignorer les navigateurs devenus obsolètes. Mais comme tous les développeurs qui ont essayé ES6 ont hâte de l'utiliser dans leurs applications, la communauté a trouvé une solution : un transpileur.

Un transpileur prend du code source ES6 en entrée et génère du code ES5, qui peut tourner dans n'importe quel navigateur. Il génère même les fichiers *source map*, qui permettent de débugger directement le code ES6 depuis le navigateur. Au moment de l'écriture de ces lignes, il y a deux outils principaux pour transpiler de l'ES6 :

- Traceur, un projet Google.
- Babeljs, un projet démarré par Sebastian McKenzie, un jeune développeur de 17 ans (oui, ça fait mal), et qui a reçu beaucoup de contributions extérieures.

Chacun a ses avantages et ses inconvénients. Par exemple Babeljs produit un code source plus lisible que Traceur. Mais Traceur est un projet Google, alors évidemment, Angular et Traceur vont bien ensemble. Le code source d'Angular 2 était d'ailleurs transpilé avec Traceur, avant de basculer en TypeScript.

Pour parler franchement, Babel est biiiiien plus populaire que Traceur, on aurait donc tendance à te le conseiller. Le projet devient peu à peu le standard de-facto.

Si tu veux jouer avec ES6, ou le mettre en place dans un de tes projets, jette un œil à ces transpileurs, et ajoute une étape à la construction de ton projet. Elle prendra tes fichiers sources ES6 et génèrera l'équivalent en ES5. Ça fonctionne très bien mais, évidemment, certaines fonctionnalités nouvelles sont difficiles ou impossibles à transformer, parce qu'elles n'existent tout simplement pas en ES5. Néanmoins l'état d'avancement actuel de ces transpileurs est largement suffisant pour les utiliser sans se faire de souci, alors jetons un coup d'œil à ces nouveautés ES6.

3.2. `let`

Si tu fais du JS depuis quelques temps, tu sais que la déclaration de variable avec `var` peut être délicate. Dans à peu près tous les autres langages, une variable existe à partir de la ligne contenant la déclaration de cette variable. Mais en JS, il y a un concept nommé *hoisting* ("remontée") qui déclare la variable au tout début de la fonction, même si tu l'as écrite plus loin.

Ainsi, déclarer une variable `name` dans le bloc `if` :

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    var name = 'Champion ' + pony.name;
    return name;
  }
  return pony.name;
}
```

est équivalent à la déclarer tout en haut de la fonction :

```
function getPonyFullName(pony) {
  var name;
  if (pony.isChampion) {
    name = 'Champion ' + pony.name;
    return name;
  }
  // name is still accessible here,
  // and can have a value from the if block
  return pony.name;
}
```

ES6 introduit un nouveau mot-clé pour la déclaration de variable, `let`, qui se comporte enfin comme on pourrait s'y attendre :

```
function getPonyFullName(pony) {
  if (pony.isChampion) {
    let name = 'Champion ' + pony.name;
    return name;
  }
  // name is not accessible here
  return pony.name;
}
```

L'accès à la variable `name` est maintenant restreint à son bloc. `let` a été pensé pour remplacer définitivement `var` à long terme, donc tu peux abandonner ce bon vieux `var` au profit de `let`. La bonne nouvelle est que ça doit être indolore, et que si ça ne l'est pas, c'est que tu as mis le doigt sur un défaut de ton code !

3.3. Constantes

Tant qu'on est sur le sujet des nouveaux mot-clés et des variables, il y en a un autre qui peut être intéressant. ES6 introduit aussi `const` pour déclarer des… constantes ! Si tu déclares une variable avec `const`, elle doit obligatoirement être initialisée, et tu ne pourras plus lui affecter de nouvelle valeur par la suite.

```
const PONIES_IN_RACE = 6;
```

```
PONIES_IN_RACE = 7; // SyntaxError
```

J'ai utilisé ici l'écriture en snake_case, en MAJUSCULE, pour nommer la constante, comme on le ferait en Java. Il n'y a aucune obligation à le faire, mais ça semble naturel d'avoir une convention d'écriture pour les constantes : trouve la tienne et tiens-y-toi !

Comme pour les variables déclarées avec `let`, les constantes ne sont pas hoisted ("remontées") et sont bien déclarées dans leur bloc.

Il y a un détail qui peut cependant surprendre le profane. Tu peux initialiser une constante avec un objet et modifier par la suite le contenu de l'objet.

```
const PONY = { };
PONY.color = 'blue'; // works
```

Mais tu ne peux pas assigner à la constante un nouvel objet :

```
const PONY = { };
```

```
PONY = {color: 'blue'}; // SyntaxError
```

Même chose avec les tableaux :

```
const PONIES = [];
PONIES.push({ color: 'blue' }); // works
```

```
PONIES = []; // SyntaxError
```

3.4. Création d'objets

Ce n'est pas un nouveau mot-clé, mais ça peut te faire tiquer en lisant du code ES6. Il y a un nouveau raccourci pour créer des objets, quand la propriété de l'objet que tu veux créer a le même nom que la variable utilisée comme valeur pour l'attribut.

Exemple :

```
function createPony() {
  let name = 'Rainbow Dash';
  let color = 'blue';
  return { name: name, color: color };
}
```

peut être simplifié en :

```
function createPony() {
  let name = 'Rainbow Dash';
  let color = 'blue';
  return { name, color };
}
```

3.5. Affectations déstructurées

Celui-là aussi peut te faire tiquer en lisant du code ES6. Il y a maintenant un raccourci pour affecter des variables à partir d'objets ou de tableaux.

En ES5 :

```
var httpOptions = { timeout: 2000, isCache: true };
// later
var httpTimeout = httpOptions.timeout;
var httpCache = httpOptions.isCache;
```

Maintenant, en ES6, tu peux écrire :

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout: httpTimeout, isCache: httpCache } = httpOptions;
```

Et tu auras le même résultat. Ca peut être perturbant, parce que la clé est la propriété à lire dans l'objet et la valeur est la variable à affecter. Mais ça fonctionne bien ! Et même mieux : si la variable que tu veux affecter a le même nom que la propriété de l'objet à lire, tu peux écrire simplement :

```
let httpOptions = { timeout: 2000, isCache: true };
// later
let { timeout, isCache } = httpOptions;
// you now have a variable named 'timeout'
// and one named 'isCache' with correct values
```

Le truc cool est que ça marche aussi avec des objets imbriqués :

```
let httpOptions = { timeout: 2000, cache: { age: 2 } };
// later
let { cache: { age } } = httpOptions;
// you now have a variable named 'age' with value 2
```

Et la même chose est possible avec des tableaux :

```
let timeouts = [1000, 2000, 3000];
// later
let [shortTimeout, mediumTimeout] = timeouts;
// you now have a variable named 'shortTimeout' with value 1000
// and a variable named 'mediumTimeout' with value 2000
```

Et bien sûr ça fonctionne avec des tableaux de tableaux, des tableaux dans des objets, etc...

Cette fonctionnalité est pratique pour déclarer plusieurs variables à partir d'un objet retourné par une fonction. Imagine une fonction `randomPonyInRace` qui retourne un poney et sa position dans la course.

```
function randomPonyInRace() {
  let pony = { name: 'Rainbow Dash' };
  let position = 2;
  // ...
  return { pony, position };
}
let { position, pony } = randomPonyInRace();
```

Cette nouvelle fonctionnalité de déstructuration assigne la `position` retournée par la méthode à la variable `position`, et le poney à la variable `pony`. Et si tu n'as pas usage de la position, tu peux écrire :

```
function randomPonyInRace() {
  let pony = { name: 'Rainbow Dash' };
  let position = 2;
  // ...
  return { pony, position };
}
let { pony } = randomPonyInRace();
```

Et tu auras seulement une variable `pony`.

3.6. Paramètres optionnels et valeurs par défaut

JS a la particularité de permettre aux développeurs d'appeler une fonction avec un nombre d'arguments variable :

- si tu passes plus d'arguments que déclarés par la fonction, les arguments supplémentaires sont tout simplement ignorés (pour être tout à fait exact, tu peux quand même les utiliser dans la fonction avec la variable spéciale `arguments`).
- si tu passes moins d'arguments que déclarés par la fonction, les paramètres manquants auront la valeur `undefined`.

Ce dernier cas est celui qui nous intéresse. Souvent, on passe moins d'arguments quand les paramètres sont optionnels, comme dans l'exemple suivant :

```
function getPonies(size, page) {
  size = size || 10;
  page = page || 1;
  // ...
  server.get(size, page);
}
```

Les paramètres optionnels ont la plupart du temps une valeur par défaut. L'opérateur OR (`||`) va

retourner l'opérande de droite si celui de gauche est `undefined`, comme ça serait le cas si le paramètre n'a pas été fourni par l'appelant (pour être précis, si l'opérande de gauche est *falsy*, c'est-à-dire `undefined`, `0`, `false`, `""`, etc...). Avec cette astuce, la fonction `getPonies` peut ainsi être invoquée :

```
getPonies(20, 2);
getPonies(); // same as getPonies(10, 1);
getPonies(15); // same as getPonies(15, 1);
```

Cela fonctionnait, mais ce n'était pas évident de savoir que les paramètres étaient optionnels, sauf à lire le corps de la fonction. ES6 offre désormais une façon plus formelle de déclarer des paramètres optionnels, dès la déclaration de la fonction :

```
function getPonies(size = 10, page = 1) {
  // ...
  server.get(size, page);
}
```

Maintenant il est limpide que la valeur par défaut de `size` sera 10 et celle de `page` sera `1` s'ils ne sont pas fournis.

NOTE

Il y a cependant une subtile différence, car maintenant `0` ou `""` sont des valeurs valides, et ne seront pas remplacées par les valeurs par défaut, comme `size = size || 10` l'aurait fait. C'est donc plutôt équivalent à `size = size === undefined ? 10: size;`.

La valeur par défaut peut aussi être un appel de fonction :

```
function getPonies(size = defaultSize(), page = 1) {
  // the defaultSize method will be called if size is not provided
  // ...
  server.get(size, page);
}
```

ou même d'autres variables, d'autres variables globales, ou d'autres paramètres de la même fonction :

```
function getPonies(size = defaultSize(), page = size - 1) {
  // if page is not provided, it will be set to the value
  // of the size parameter minus one.
  // ...
  server.get(size, page);
}
```

Note que si tu essayes d'utiliser des paramètres sur la droite, leur valeur sera toujours `undefined` :

```
function getPonies(size = page, page = 1) {
  // size will always be undefined, as the page parameter is on its right.
  server.get(size, page);
}
```

Ce mécanisme de valeur par défaut ne s'applique pas qu'aux paramètres de fonction, mais aussi aux valeurs de variables, par exemple dans le cas d'une affectation déstructurée :

```
let { timeout = 1000 } = httpOptions;
// you now have a variable named 'timeout',
// with the value of 'httpOptions.timeout' if it exists
// or 1000 if not
```

3.7. *Rest operator*

ES6 introduit aussi une nouvelle syntaxe pour déclarer un nombre variable de paramètres dans une fonction. Comme on le disait précédemment, tu peux toujours passer des arguments supplémentaires à un appel de fonction, et y accéder avec la variable spéciale `arguments`. Tu peux faire quelque chose comme :

```
function addPonies(ponies) {
  for (var i = 0; i < arguments.length; i++) {
    poniesInRace.push(arguments[i]);
  }
}
addPonies('Rainbow Dash', 'Pinkie Pie');
```

Mais tu seras d'accord pour dire que ce n'est ni élégant, ni évident : le paramètre `ponies` n'est jamais utilisé, et rien n'indique qu'on peut fournir plusieurs poneys.

ES6 propose une syntaxe bien meilleure, grâce au *rest operator* ("opérateur de reste").

```
function addPonies(...ponies) {
  for (let pony of ponies) {
    poniesInRace.push(pony);
  }
}
```

`ponies` est désormais un véritable tableau, sur lequel on peut itérer. La boucle `for ⋯ of` utilisée pour l'itération est aussi une nouveauté d'ES6. Elle permet d'être sûr de n'itérer que sur les valeurs de la collection, et non pas sur ses propriétés comme `for ⋯ in`. Ne trouves-tu pas que notre code est

maintenant bien plus beau et lisible ?

Le *rest operator* peut aussi fonctionner avec des affectations déstructurées :

```
let [winner, ...losers] = poniesInRace;
// assuming 'poniesInRace' is an array containing several ponies
// 'winner' will have the first pony,
// and 'losers' will be an array of the other ones
```

Le *rest operator* ne doit pas être confondu avec le *spread operator* ("opérateur d'étalement"), même si, on te l'accorde, ils se ressemblent dangeureusement ! Le *spread operator* est son opposé : il prend un tableau, et l'étale en arguments variables. Le seul cas d'utilisation qui me vient à l'esprit serait pour les fonctions comme `min` ou `max`, qui peuvent recevoir des arguments variables, et que tu voudrais appeler avec un tableau :

```
let ponyPrices = [12, 3, 4];
let minPrice = Math.min(...ponyPrices);
```

3.8. Classes

Une des fonctionnalités les plus emblématiques, et qui va largement être utilisée dans l'écriture d'applications Angular : ES6 introduit les classes en JavaScript ! Tu pourras désormais facilement faire de l'héritage de classes en JavaScript. Tu pouvais déjà, avec l'héritage prototypal, mais ce n'était pas chose facile, surtout pour les débutants...

Maintenant c'est les doigts dans le nez, regarde :

```
class Pony {
  constructor(color) {
    this.color = color;
  }
  toString() {
    return `${this.color} pony`;
    // see that? It is another cool feature of ES6, called template literals
    // we'll talk about these quickly!
  }
}
let bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Les déclarations de classes, contrairement aux déclarations de fonctions, ne sont pas *hoisted* ("remontées"), donc tu dois déclarer une classe avant de l'utiliser. Tu as probablement remarqué la fonction spéciale `constructor`. C'est le constructeur, la fonction appelée à la création d'un nouvel objet

avec le mot-clé `new`. Dans l'exemple, il requiert une couleur, et nous créons une nouvelle instance de la classe `Pony` avec la couleur "blue". Une classe peut aussi avoir des méthodes, appelables sur une instance, comme la méthode `toString()` dans l'exemple.

Une classe peut aussi avoir des attributs et des méthodes statiques :

```
class Pony {
  static defaultSpeed() {
    return 10;
  }
}
```

Ces méthodes statiques ne peuvent être appelées que sur la classe directement :

```
let speed = Pony.defaultSpeed();
```

Une classe peut avoir des accesseurs (*getters*, *setters*), si tu veux implémenter du code sur ces opérations :

```
class Pony {
  get color() {
    console.log('get color');
    return this._color;
  }
  set color(newColor) {
    console.log(`set color ${newColor}`);
    this._color = newColor;
  }
}
let pony = new Pony();
pony.color = 'red'; // set color red
console.log(pony.color); // get color
```

Et bien évidemment, si tu as des classes, l'héritage est possible en ES6.

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {

}
let pony = new Pony();
console.log(pony.speed()); // 10, as Pony inherits the parent method
```

Animal est appelée la classe de base, et Pony la classe dérivée. Comme tu l'as vu, la classe dérivée possède toutes les méthodes de la classe de base. Mais elle peut aussi les redéfinir :

```
class Animal {
  speed() {
    return 10;
  }
}
class Pony extends Animal {
  speed() {
    return super.speed() + 10;
  }
}
let pony = new Pony();
console.log(pony.speed()); // 20, as Pony overrides the parent method
```

Comme tu le vois, le mot-clé `super` permet d'invoquer la méthode de la classe de base, avec `super.speed()` par exemple.

Ce mot-clé `super` peut aussi êre utilisé dans les constructeurs, pour invoquer le constructeur de la classe de base :

```
class Animal {
  constructor(speed) {
    this.speed = speed;
  }
}
class Pony extends Animal {
  constructor(speed, color) {
    super(speed);
    this.color = color;
  }
}
let pony = new Pony(20, 'blue');
console.log(pony.speed); // 20
```

3.9. *Promises*

Les *promises* ("promesses") ne sont pas si nouvelles, et tu les connais ou les utilises peut-être déjà, parce qu'elles tenaient une place importante dans AngularJS 1.x. Mais comme nous les utiliserons beaucoup avec Angular 2, et même si tu n'utilises que du pur JS sans Angular, on pense que c'est important de s'y attarder un peu.

L'objectif des *promises* est de simplifier la programmation asynchrone. Notre code JS est plein d'asynchronisme, comme des requêtes AJAX, et en général on utilise des *callbacks* pour gérer le résultat et l'erreur. Mais le code devient vite confus, avec des *callbacks* dans des *callbacks*, qui le rendent illisible et peu maintenable. Les *promises* sont plus pratiques que les *callbacks*, parce qu'elles permettent d'écrire du code à plat, et le rendent ainsi plus simple à comprendre. Prenons un cas d'utilisation simple, où on doit récupérer un utilisateur, puis ses droits, puis mettre à jour un menu quand on a récupéré tout ça .

Avec des *callbacks* :

```
getUser(login, function(user) {
  getRights(user, function(rights) {
    updateMenu(rights);
  });
});
```

Avec des *promises* :

```
getUser()
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    updateMenu(rights);
  })
```

J'aime cette version, parce qu'elle s'exécute comme elle se lit : je veux récupérer un utilisateur, puis ses droits, puis mettre à jour le menu.

Une *promise* est un objet *thenable*, ce qui signifie simplement qu'il a une méthode `then`. Cette méthode prend deux arguments : un callback de succès et un callback d'erreur. Une *promise* a trois états :

- *pending* ("en cours") : quand la *promise* n'est pas réalisée, par exemple quand l'appel serveur n'est pas encore terminé.
- *fulfilled* ("réalisée") : quand la *promise* s'est réalisée avec succès, par exemple quand l'appel HTTP serveur a retourné un status 200-OK.
- *rejected* ("rejetée") : quand la *promise* a échoué, par exemple si l'appel HTTP serveur a retourné un status 404-NotFound.

Quand la promesse est réalisée (*fulfilled*), alors le callback de succès est invoqué, avec le résultat en argument. Si la promesse est rejetée (*rejected*), alors le callback d'erreur est invoqué, avec la valeur rejetée ou une erreur en argument.

Alors, comment crée-t-on une *promise* ? C'est simple, il y a une nouvelle classe `Promise`, dont le constructeur attend une fonction avec deux paramètres, `resolve` et `reject`.

```
let getUser = function() {
  return new Promise(function(resolve, reject) {
    // async stuff, like fetching users from server, returning a response
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

Une fois la *promise* créée, tu peux enregistrer des *callbacks*, via la méthode `then`. Cette méthode peut recevoir deux arguments, les deux *callbacks* que tu veux voir invoqués en cas de succès ou en cas d'échec. Dans l'exemple suivant, nous passons simplement un seul *callback* de succès, ignorant ainsi une erreur potentielle :

```
getUser()
  .then(function(user) {
    console.log(user);
  })
```

Quand la promesse sera réalisée, le callback de succès (qui se contente ici de tracer l'utilisateur en console) sera invoqué.

La partie la plus cool c'est que le code peut s'écrire à plat. Si par exemple ton *callback* de succès retourne lui aussi une *promise*, tu peux écrire :

```
getUser()
  .then(function(user) {
    return getRights(user) // getRights is returning a promise
      .then(function(rights) {
        return updateMenu(rights);
      });
  })
```

ou plus élégamment :

```
getUser()
  .then(function(user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
```

Un autre truc cool est la gestion d'erreur : tu peux définir une gestion d'erreur par *promise*, ou globale à toute la chaîne.

Une gestion d'erreur par *promise* :

```
getUser()
  .then(function(user) {
    return getRights(user);
  }, function(error) {
    console.log(error); // will be called if getUser fails
    return Promise.reject(error);
  })
  .then(function(rights) {
    return updateMenu(rights);
  }, function(error) {
    console.log(error); // will be called if getRights fails
    return Promise.reject(error);
  })
```

Une gestion d'erreur globale pour toute la chaîne :

```
getUser()
  .then(function(user) {
    return getRights(user);
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
  .catch(function(error) {
    console.log(error); // will be called if getUser or getRights fails
  })
```

Tu devrais sérieusement t'ntéresser aux *promises*, parce que ça va devenir la nouvelle façon d'écrire des APIs, et toutes les bibliothèques vont bientôt les utiliser. Même les bibliothèques standards : c'est le cas de la nouvelle Fetch API par exemple.

3.10. (*arrow functions*)

Un truc que j'adore dans ES6 est la nouvelle syntaxe *arrow function* ("fonction flèche"), utilisant l'opérateur *fat arrow* ("grosse flèche") : ⇒. C'est super utile pour les *callbacks* et les fonctions anonymes !

Prenons notre exemple précédent avec des *promises* :

```
getUser()
  .then(function(user) {
    return getRights(user); // getRights is returning a promise
  })
  .then(function(rights) {
    return updateMenu(rights);
  })
```

Il peut être réécrit avec des *arrow functions* comme ceci :

```
getUser()
  .then(user => getRights(user))
  .then(rights => updateMenu(rights))
```

N'est-ce pas super cool ?!

Note que le `return` est implicite s'il n'y a pas de bloc : pas besoin d'écrire `user ⇒ return getRights(user)`. Mais si nous avions un bloc, nous aurions besoin d'un `return` explicite :

```
getUser()
  .then(user => {
    console.log(user);
    return getRights(user);
  })
  .then(rights => updateMenu(rights))
```

Et les *arrow functions* ont une particularité bien agréable que n'ont pas les fonctions normales : le `this` reste attaché lexicalement, ce qui signifie que ces *arrow functions* n'ont pas un nouveau `this` comme les fonctions normales. Prenons un exemple où on itère sur un tableau avec la fonction `map` pour y trouver le maximum.

En ES5 :

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    // let's iterate
    numbers.forEach(
      function(element) {
        // if the element is greater, set it as the max
        if (element > this.max) {
          this.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Ca semble pas mal, non ? Mais en fait ça ne marche pas… Si tu as de bons yeux, tu as remarqué que le `forEach` dans la fonction `find` utilise `this`, mais ce `this` n'est lié à aucun objet. Donc `this.max` n'est en fait pas le `max` de l'objet `maxFinder`… On pourrait corriger ça facilement avec un alias :

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    var self = this;
    numbers.forEach(
      function(element) {
        if (element > self.max) {
          self.max = element;
        }
      });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

ou en *bindant* le `this` :

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(
      function(element) {
        if (element > this.max) {
          this.max = element;
        }
      }.bind(this));
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

ou en le passant en second paramètre de la fonction `forEach` (ce qui est justement sa raison d'être) :

```
var maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(
      function(element) {
        if (element > this.max) {
          this.max = element;
        }
      }, this);
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Mais il y a maintenant une solution bien plus élégante avec les *arrow functions* :

```
let maxFinder = {
  max: 0,
  find: function(numbers) {
    numbers.forEach(element => {
      if (element > this.max) {
        this.max = element;
      }
    });
  }
};

maxFinder.find([2, 3, 4]);
// log the result
console.log(maxFinder.max);
```

Les *arrow functions* sont donc idéales pour les fonctions anynomes en *callback* !

3.11. *Set* et *Map*

On va faire court : on a maintenant de vraies collections en ES6. Youpi \o/!

On utilisait jusque-là de simples objets JavaScript pour jouer le rôle de *map* ("dictionnaire"), c'est à dire un objet JS standard, dont les clés étaient nécessairement des chaînes de caractères. Mais nous pouvons maintenant utiliser la nouvelle classe *Map* :

```
let cedric = { id: 1, name: 'Cedric' };
let users = new Map();
users.set(cedric.id, cedric); // adds a user
console.log(users.has(cedric.id)); // true
console.log(users.size); // 1
users.delete(cedric.id); // removes the user
```

On a aussi une classe `Set` ("ensemble") :

```
let cedric = { id: 1, name: 'Cedric' };
let users = new Set();
users.add(cedric); // adds a user
console.log(users.has(cedric)); // true
console.log(users.size); // 1
users.delete(cedric); // removes the user
```

Tu peux aussi itérer sur une collection, avec la nouvelle syntaxe `for ⋯ of` :

```
for (let user of users) {
  console.log(user.name);
}
```

Tu verras que cette syntaxe `for ⋯ of` est celle choisie par l'équipe Angular pour itérer sur une collection dans un template.

3.12. Template de string

Construire des strings a toujours été pénible en JavaScript, où nous devions généralement utiliser des concaténations :

```
let fullname = 'Miss ' + firstname + ' ' + lastname;
```

Les templates de string sont une nouvelle fonctionnalité mineure mais bien pratique, où on doit utiliser des accents graves (*backticks* `) au lieu des habituelles apostrophes (*quote* ') ou apostrophes doubles (*double-quotes* "), fournissant un moteur de template basique avec support du multi-ligne :

```
let fullname = `Miss ${firstname} ${lastname}`;
```

Le support du multi-ligne est particulièrement adapté à l'écriture de morceaux d'HTML, comme nous le feront dans nos composants Angular :

```
let template = `<div>
  <h1>Hello</h1>
</div>`;
```

3.13. Modules

Il a toujours manqué en JavaScript une façon standard de ranger ses fonctions dans un espace de nommage, et de charger dynamiquement du code. NodeJS a été un leader sur le sujet, avec un écosystème très riche de modules utilisant la convention CommonJS. Côté navigateur, il y a aussi l'API AMD (Asynchronous Module Definition), utilisé par RequireJS. Mais aucun n'était un vrai standard, ce qui nous conduit à des débats incessants sur la meilleure solution.

ES6 a pour objectif de créer une syntaxe avec le meilleur des deux mondes, sans se préoccuper de l'implémentation utilisée. Le comité Ecma TC39 (qui est responsable des évolutions d'ES6 et auteur de la spécification du langage) voulait une syntaxe simple (c'est indéniablement l'atout de CommonJS), mais avec le support du chargement asynchrone (comme AMD), et avec quelques bonus comme la possibilité d'analyser statiquement le code par des outils et une gestion claire des dépendances

cycliques. Cette nouvelle syntaxe se charge de déclarer ce que tu exportes depuis tes modules, et ce que tu importes dans d'autres modules.

Cette gestion des modules est fondamentale dans Angular 2, parce que tout y est défini dans des modules, qu'il faut importer dès qu'on veut les utiliser. Supposons qu'on veuille exposer une fonction pour parier sur un poney donné dans une course, et une fonction pour lancer la course.

Dans `races_service.js`:

```
export function bet(race, pony) {
  // ...
}
export function start(race) {
  // ...
}
```

Comme tu le vois, c'est plutôt simple : le nouveau mot-clé `export` fait son travail et exporte les deux fonctions.

Maintenant, supposons qu'un composant de notre application veuille appeler ces deux fonctions :

Dans races_service.js

```
import { bet, start } from 'races_service';
```

```
// later
bet(race, pony1);
start(race);
```

C'est ce qu'on appelle un *named export* ("export nommé"). Ici on importe les deux fonctions, et on doit spécifier le nom du fichier contenant ces deux fonctions, ici 'races_service'. Evidemment, on peut importer une seule des deux fonctions, avec un alias même si besoin :

```
import { start as startRace } from 'races_service';
```

```
// later
startRace(race);
```

Et si tu veux importer toutes les fonctions du module, tu peux utiliser le caractère joker *.

Comme tu le ferais dans d'autres langages, il faut utiliser le caractère joker * avec modération,

seulement si tu as besoin de toutes les fonctions, ou la plupart. Et comme tout ceci sera prochainement géré par ton IDE préféré qui prendra en charge la gestion automatique des imports, on n'aura plus à se soucier d'importer les seules bonnes fonctions.

Avec un caractère joker, tu dois utiliser un alias, et j'aime plutôt ça, parce que ça rend le reste du code plus lisible :

```
import * as racesService from 'races_service';
```

```
// later
racesService.bet(race, pony1);
racesService.start(race);
```

Si ton module n'expose qu'une seule fonction, ou valeur, ou classe, tu n'as pas besoin d'utiliser un *named export*, et tu peux bénéficier de l'export par défaut, avec le mot-clé `default`. C'est pratique pour les classes notamment :

```
// pony.js
export default class Pony { }
// races_service.js
import Pony from 'pony';
```

Note l'absence d'accolade pour importer un export par défaut. Tu peux l'importer avec l'alias que tu veux, mais pour être cohérent, c'est mieux de l'importer avec le nom du module (sauf évidemment si tu importes plusieurs modules portant le même nom, auquel cas tu devras donner un alias pour les distinguer). Et bien sûr tu peux mélanger l'export par défaut et l'export nommé, mais un module ne pourra avoir qu'un seul export par défaut.

En Angular 2, tu utiliseras beaucoup de ces imports dans ton application. Chaque composant et service sera une classe, généralement isolée dans son propre fichier et exportée, et ensuite importée à la demande dans chaque autre composant.

3.14. Conclusion

Voilà qui conclue notre rapide introduction à ES6. On a zappé quelques parties, mais si tu as bien assimilé ce chapitre, tu n'auras aucun problème à coder ton application en ES6. Si tu veux approfondir, je te recommande chaudement Exploring JS par Axel Rauschmayer, ou Understanding ES6 par Nicholas C. Zakas. Ces deux ebooks peuvent être lus gratuitement en ligne, mais pense à soutenir ces auteurs qui ont fait un beau travail ! En l'occurrence, j'ai relu récemment Speaking JS, le précédent livre d'Axel, et j'ai encore appris quelques trucs, donc si tu veux raffraichir tes connaissances en JS, je te le conseille vivement !

Chapter 4. Un peu plus loin qu'ES6

4.1. Types dynamiques, statiques et optionnels

Tu sais probablement que les applications Angular 2 peuvent être écrites en ES5, ES6, ou TypeScript. Et tu te demandes peut-être qu'est-ce que TypeScript, et ce qu'il apporte de plus.

JavaScript est dynamiquement typé. Tu peux donc faire des trucs comme :

```
let pony = 'Rainbow Dash';
pony = 2;
```

Et ça fonctionne. Ça offre pleins de possibilités : tu peux ainsi passer n'importe quel objet à une fonction, tant que cet objet a les propriétés requises par la fonction :

```
let pony = { name: 'Rainbow Dash', color: 'blue' };
let horse = { speed: 40, color: 'black' };
let printColor = animal => console.log(animal.color);
// works as long as the object has a `color` property
```

Cette nature dynamique est formidable, mais elle est aussi un handicap dans certains cas, comparée à d'autres langages plus fortement typés. Le cas le plus évident est quand tu dois appeler une fonction inconnue d'une autre API en JS : tu dois lire la documentation (ou pire le code de la fonction) pour deviner à quoi doivent ressembler les paramètres. Dans notre exemple précédent, la méthode `printColor` attend un paramètre avec une propriété `color`. Ça peut être dur à deviner, et c'est encore plus difficile dans notre travail quotidien, où on multiplie les utilisations de bibliothèques et services développés par d'autres. Un des co-fondateurs de Ninja Squad se plaint souvent du manque de type en JS, et déclare qu'il n'est pas aussi productif, et qu'il ne produit pas du code aussi bon qu'il le ferait dans un environnement plus statiquement typé. Et il n'a pas entièrement tort, même s'il trolle aussi par plaisir ! Sans les informations de type, les IDEs n'ont aucun indice pour savoir si tu écris quelque chose de faux, et les outils ne peuvent pas t'aider à trouver des bugs dans ton code. Bien sûr, nos applications sont testées, et Angular a toujours facilité les tests, mais c'est pratiquement impossible d'avoir une parfaite couverture de tests.

Cela nous ammène sur le sujet de la maintenabilité. Le code JS peut être difficile à maintenir, malgré les tests et la documentation. Refactorer une grosse application JS n'est pas chose aisée, comparativement à ce qui peut être fait dans des langages statiquement typés. La maintenabilité est un sujet important, et les types aident les outils comme les développeurs à éviter les erreurs lors de l'écriture et la modification de code. Google a toujours été enclin à proposer des solutions dans cette direction : c'est compréhensible, étant donné qu'ils gèrent des applications parmi les plus grosses du monde, avec GMail, Google apps, Maps... Alors ils ont essayé plusieurs approches pour améliorer la maintenablité des applications *front-end* : GWT, Google Closure, Dart... Elles devaient toutes faciliter

l'écriture de grosses applications web.

Avec Angular 2, l'équipe Google voulait nous aider à écrire du meilleur JS, en ajoutant des informations de type à notre code. Ce n'est pas concept nouveau pour JS, c'était même le sujet de la spécification ECMASCRIPT 4, qui a été abandonnée. Au départ ils annoncèrent AtScript, un sur-ensemble d'ES6 avec des annotations (des annotations de type et d'autres, dont je parlerai plus tard). Ils annoncèrent ensuite le support de TypeScript, le langage de Microsoft, avec des annotations de type additionnelles. Et enfin, quelques mois plus tard, l'équipe TypeScript annonçait, après un travail étroit avec l'équipe de Google, que la nouvelle version du langage (1.5) aurait toutes les nouvelles fonctionnalités d'AtScript. L'équipe Angular déclara alors qu'AtScript était officiellement abandonné, et que TypeScript était désormais la meilleure façon d'écrire des applications Angular 2 !

4.2. Hello TypeScript

Je pense que c'était la meilleure chose à faire pour plusieurs raisons. D'abord, personne n'a vraiment envie d'apprendre une nouvelle extension de langage. Et TypeScript existait déjà, avec une communauté et un écosystème actifs. Je ne l'avais jamais vraiment utilisé avant Angular 2, mais j'en avais entendu du bien, de personnes différentes. TypeScript est un projet de Microsoft, mais ce n'est pas le Microsoft de l'ère Balmer et Gates. C'est le Microsoft de Nadella, celui qui s'ouvre à la communauté, et donc, à l'open-source. Google en a conscience, et c'est tout à leur avantage de contribuer à un projet existant, plutôt que de maintenir le leur. Le framework TypeScript gagnera de son côté en visibilité : *win-win* comme dirait ton manager.

Mais la raison principale de parier sur TypeScript est le système de types qu'il offre. C'est un système optionnel qui vient t'aider sans t'entraver. De fait, après avoir codé quelque temps avec, il s'est fait complètement oublier : tu peux faire des applications Angular 2 en utilisant les trucs de TypeScript les plus utiles (j'y reviendrai) et en ignorant tout le reste avec du pur JavaScript (ES6 dans mon cas). Mais j'aime ce qu'ils ont fait, et on jettera un coup d'oeil à TypeScript dans le chapitre suivant. Tu pourras ainsi lire et comprendre n'importe quel code Angular 2, et tu pourras décider de l'utiliser, ou pas, ou juste un peu, dans tes applications.

Si tu te demandes "mais pourquoi avoir du code fortement typé dans une application Angular 2 ?", prenons un exemple. Angular 1 et 2 ont été construits sur le puissant concept d'injection de dépendance. Tu le connais déjà peut-être, parce que c'est un *design pattern* classique, utilisé dans beaucoup de frameworks et langages, et notamment AngularJS 1.x comme je le disais.

4.3. Un exemple concret d'injection de dépendance

Pour synthétiser ce qu'est l'injection de dépendence, prenons un composant d'une application, disons `RaceList`, permettant d'accéder à la liste des courses que le service `RaceService` peut retourner. Tu peux écrire `RaceList` comme ça :

```
class RaceList {
  constructor() {
    this.raceService = new RaceService();
    // let's say that list() returns a promise
    this.raceService.list()
    // we store the races returned into a member of `RaceList`
      .then(races => this.races = races);
      // arrow functions, FTW!
  }
}
```

Mais ce code a plusieurs défauts. L'un d'eux est la testabilité : c'est compliqué de remplacer `raceService` par un faux service (un bouchon, un *mock*), pour tester notre composant.

Si nous utilisons le *pattern* d'injection de dépendance (*Dependency Injection*, DI), nous déléguons la création de `RaceService` à un framework, lui réclamant simplement une instance. Le framework est ainsi en charge de la création de la dépendance, et il peut nous "l'injecter", par exemple dans le constructeur :

```
class RaceList {
  constructor(raceService) {
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}
```

Désormais, quand on teste cette classe, on peut facilement passer un faux service au constructeur :

```
// in a test
let fakeService = {
  list: () => {
    // returns a fake list
  }
};
let raceList = new RaceList(fakeService);
// now we are sure that the race list
// is the one we want for the test
```

Mais comment le framework sait-il quel composant injecter dans le constructeur ? Bonne question ! AngularJS 1.x se basait sur le nom du paramètre, mais cela a une sérieuse limitation : la minification du code va changer le nom du paramètre. Pour contourner ce problème, tu pouvais utiliser la notation à base de tableau, ou ajouter des métadonnées à la classe :

```
RaceList.inject = ['RaceService'];
```

Il nous fallait donc ajouter des métadonnées pour que le framework comprenne ce qu'il fallait injecter dans nos classes. Et c'est exactement ce que proposent les annotations de type : une métadonnée donnant un indice nécessaire au framework pour réaliser la bonne injection. En Angular 2, avec TypeScript, voilà à quoi pourrait ressembler notre composant `RaceList` :

```
class RaceList {
  raceService: RaceService;
  races: Array<string>;

  constructor(raceService: RaceService) {
    // the interesting part is `: RaceService`
    this.raceService = raceService;
    this.raceService.list()
      .then(races => this.races = races);
  }
}
```

Maintenant l'injection peut se faire sans ambiguité ! Tu n'es pas obligé d'utiliser TypeScript en Angular 2, mais clairement ton code sera plus élégant avec. Tu peux toujours faire la même chose en pur ES6 ou ES5, mais tu devras ajouter manuellement des métadonnées d'une autre façon (on y reviendra en détail).

C'est pourquoi nous allons passer un peu de temps à apprendre TypeScript (TS). Angular 2 est clairement construit pour tirer parti d'ES6 et TS 1.5+, et rendre notre vie de développeur plus facile en l'utilisant. Et l'équipe Angular a envie de soumettre le système de type au comité de standardisation, donc peut-être qu'un jour il sera normal d'avoir de vrais types en JS.

Il est temps désormais de se lancer dans TypeScript !

Chapter 5. Découvrir TypeScript

TypeScript, qui existe depuis 2012, est un sur-ensemble de JavaScript, ajoutant quelques trucs à ES5. Le plus important étant le système de type, lui donnant même son nom. Depuis la version 1.5, sortie en 2015, cette bibliothèque essaie d'être un sur-ensemble d'ES6, incluant toutes les fonctionnalités vues précédemment, et quelques nouveautés, comme les décorateurs. Ecrire du TypeScript ressemble à écrire du JavaScript. Par convention les fichiers sources TypeScript ont l'extension `.ts`, et seront compilés en JavaScript standard, en général lors du build, avec le compilateur TypeScript. Le code généré reste très lisible.

```
npm install -g typescript
tsc test.ts
```

Mais commençons par le début.

5.1. Les types de TypeScript

La syntaxe pour ajouter des informations de type en TypeScript est basique :

```
let variable: type;
```

Les différents types sont simples à retenir :

```
let poneyNumber: number = 0;
let poneyName: string = 'Rainbow Dash';
```

Dans ces cas, les types sont facultatifs, car le compilateur TS peut les deviner depuis leur valeur (c'est ce qu'on appelle l'inférence de type).

Le type peut aussi être défini dans ton application, avec par exemple la classe suivante `Pony` :

```
let pony: Pony = new Pony();
```

TypeScript supporte aussi ce que certains langages appellent des types génériques, par exemple avec un `Array` :

```
let ponies: Array<Pony> = [new Pony()];
```

Cet `Array` ne peut contenir que des poneys, ce qu'indique la notation générique `<>`. On peut se

demander quel est l'intérêt d'imposer cela. Ajouter de telles informations de type aidera le compilateur à détecter des erreurs :

```
ponies.push('hello'); // error TS2345
// Argument of type 'string' is not assignable to parameter of type 'Pony'.
```

Et comment faire si tu as besoin d'une variable pouvant recevoir plusieurs types ? TS a un type spécial pour cela, nommé `any`.

```
let changing: any = 2;
changing = true; // no problem
```

C'est pratique si tu ne connais pas le type d'une valeur, soit parce qu'elle vient d'un bout de code dynamique, ou en sortie d'une bibliothèque obscure.

Si ta variable ne doit recevoir que des valeurs de type `number` ou `boolean`, tu peux utiliser l'union de types :

```
let changing: number|boolean = 2;
changing = true; // no problem
```

5.2. Valeurs énumérées (*enum*)

TypeScript propose aussi des valeurs énumérées : `enum`. Par exemple, une course de poneys dans ton application peut être soit `ready`, `started` ou `done`.

```
enum RaceStatus {Ready, Started, Done}
let race: Race = new Race();
race.status = RaceStatus.Ready;
```

Un `enum` est en fait une valeur numérique, commençant à 0. Tu peux cependant définir la valeur que tu veux :

```
enum Medal {Gold = 1, Silver, Bronze}
```

5.3. Return types

Tu peux aussi spécifier le type de retour d'une fonction :

```
function startRace(race: Race): Race {
  race.status = RaceStatus.Started;
  return race;
}
```

Si la fonction ne retourne rien, tu peux le déclarer avec `void` :

```
function startRace(race: Race): void {
  race.status = RaceStatus.Started;
}
```

5.4. Interfaces

C'est déjà une bonne première étape. Mais comme je le disais plus tôt, JavaScript est formidable par sa nature dynamique. Une fonction marchera si elle reçoit un objet possédant la bonne propriété :

```
function addPointsToScore(player, points) {
  player.score += points;
}
```

Cette fonction peut être appliquée à n'importe quel objet ayant une propriété `score`. Maintenant comment traduit-on cela en TypeScript ? Facile !, on définit une interface, un peu comme la "forme" de l'objet.

```
function addPointsToScore(player: { score: number; }, points: number): void {
  player.score += points;
}
```

Cela signifie que le paramètre doit avoir une propriété nommée `score` de type `number`. Tu peux évidement aussi nommer ces interfaces :

```
interface HasScore {
  score: number;
}
function addPointsToScore(player: HasScore, points: number): void {
  player.score += points;
}
```

5.5. Paramètre optionnel

Y'a un autre truc sympa en JavaScript : les paramètres optionnels. Si tu ne les passes pas à l'appel de la fonction, leur valeur sera `undefined`. Mais en TypeScript, si tu déclares une fonction avec des paramètres typés, le compilateur te gueulera dessus si tu les oublies :

```
addPointsToScore(player); // error TS2346
// Supplied parameters do not match any signature of call target.
```

Pour montrer qu'un paramètre est optionnel dans une fonction (ou une propriété dans une interface), tu ajoutes `?` après le paramètre. Ici, le paramètre `points` est optionnel :

```
function addPointsToScore(player: HasScore, points?: number): void {
  points = points || 0;
  player.score += points;
}
```

5.6. Des fonctions en propriété

Tu peux aussi décrire un paramètre comme devant posséder une fonction spécifique plutôt qu'une propriété :

```
function startRunning(pony) {
  pony.run(10);
}
```

La définition de cette interface serait :

```
interface CanRun {
  run(meters: number): void;
}
function startRunning(pony: CanRun): void {
  pony.run(10);
}

let pony = {
  run: (meters) => logger.log(`pony runs ${meters}m`)
};
startRunning(pony);
```

5.7. Classes

Une classe peut implémenter une interface. Pour nous, un poney peut courir, donc on pourrait écrire :

```
class Pony implements CanRun {
  run(meters) {
    logger.log(`pony runs ${meters}m`);
  }
}
```

Le compilateur nous obligera à implémenter la méthode `run` dans la classe. Si nous l'implémentons mal, par exemple en attendant une `string` au lieu d'un `number`, le compilateur va crier :

```
class IllegalPony implements CanRun {
  run(meters: string) {
    console.log(`pony runs ${meters}m`);
  }
}
// error TS2420: Class 'IllegalPony' incorrectly implements interface 'CanRun'.
// Types of property 'run' are incompatible.
```

Tu peux aussi implémenter plusieurs interfaces si ça te fait plaisir :

```
class HungryPony implements CanRun, CanEat {
  run(meters) {
    logger.log(`pony runs ${meters}m`);
  }
  eat() {
    logger.log(`pony eats`);
  }
}
```

Et une interface peut en étendre une ou plusieurs autres :

```
interface Animal extends CanRun, CanEat {}

class Pony implements Animal {
  // ...
}
```

Une classe en TypeScript peut avoir des propriétés et des méthodes. Avoir des propriétés dans une classe n'est pas une fonctionnalité standard d'ES6, c'est seulement possible en TypeScript.

```
class SpeedyPony {
  speed: number = 10;
  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

Tout est public par défaut. Mais tu peux utiliser le mot-clé `private` pour cacher une propriété ou une méthode. Ajouter `public` ou `private` à un paramètre de constructeur est un raccourci pour créer et initialiser un membre privé ou public :

```
class NamedPony {
  constructor(public name: string,
              private speed: number) {}

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
let pony = new NamedPony('Rainbow Dash', 10);
// defines a public property name with 'Rainbow Dash'
// and a private one speed with 10
```

Ces raccourcis sont très pratiques et nous allons beaucoup les utiliser en Angular 2 !

5.8. Utiliser d'autres bibliothèques

Mais si on travaille avec des bibliothèques externes écrites en JS, comment savoir les types des paramètres attendus par telle fonction de telle bibliothèque ? La communauté TypeScript est tellement cool que ses membres ont défini des interfaces pour les types et les fonctions exposés par les bibliothèques JavaScript les plus populaires.

Les fichiers contenant ces interfaces ont une extension spéciale : `.d.ts`. Ils contiennent une liste de toutes les fonctions publiques des bibliothèques. DefinitelyTyped est l'outil de référence pour récupérer ces fichiers. Par exemple, si tu veux utiliser TS dans ton application AngularJS 1.x, tu peux récupérer le fichier dédié depuis le repository :

```
tsd query angular --action install --save
```

Puis tu inclues ce fichier au début de ton code et hop!, tu profites du bonheur d'avoir une compilation *typesafe* :

```
/// <reference path="angular.d.ts" />
angular.module(10, []); // the module name should be a string
// so when I compile, I get:
// Argument of type 'number' is not assignable to parameter of type 'string'.
```

`/// <reference path="angular.d.ts" />` est un commentaire spécial reconnu par TS, qui indique au compilateur de vérifier l'interface `angular.d.ts`. Maintenant, si tu te trompes dans l'appel d'une méthode AngularJS, le compilateur te le dira, et tu peux corriger sans avoir à lancer manuellement ton application !

Encore plus cool, depuis TypeScript 1.6, le compilateur est capable de trouver par lui-même ces interfaces si elles sont packagées avec ta dépendance dans ton répertoire `node_modules`. De plus en plus de projets adoptent cette approche, et Angular 2 fait de même. Tu n'as donc même pas à t'occuper d'inclure ces interfaces dans ton projet Angular 2 : le compilateur TS va tout comprendre comme un grand si tu utilises NPM pour gérer tes dépendances !

5.9. Decorateurs

C'est une fonctionnalité toute nouvelle, ajoutée seulement en TypeScript 1.5, juste pour le support d'Angular. En effet, comme on le verra bientôt, les composants Angular 2 peuvent être décrits avec des décorateurs. Tu n'as peut-être jamais entendu parler de décorateurs, car tous les langages ne les proposent pas. Un décorateur est une façon de faire de la méta-programmation. Ils ressemblent beaucoup aux annotations, qui sont principalement utilisées en Java, C#, et Python, et peut-être d'autres langages que je ne connais pas. Suivant le langage, tu peux ajouter une annotation sur une méthode, un attribut, ou une classe. Généralement, les annotations ne sont pas vraiment utiles au langage lui-même, mais plutôt aux frameworks et aux bibliothèques.

Les décorateurs sont vraiment puissants: ils peuvent modifier leur cible (classes, méthodes, etc...) et par exemple modifier les paramètres ou le résultat retourné, appeler d'autres méthodes quand la cible est appelée, ou ajouter des métadonnées destiné à un framework à un bout de code (c'est ce que font les décorateurs d'Angular 2). Jusqu'à présent, cela n'existait pas en JavaScript. Mais le langage évolue, et il y a maintenant une proposition officielle pour le support des décorateurs, qui les rendra peut-être possibles un jour (possiblement avec ES7/ES2016).

En Angular 2, on utilisera les annotations fournies par le framework. Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette dépendance est optionnelle", "ceci est une propriété spéciale du composant", etc... Ce n'est pas obligatoire de les utiliser, car tu peux toujours ajouter les métadonnées manuellement si tu ne veux que du pur ES5, mais le code sera plus élégant avec des décorateurs, comme ceux proposés par TypeScript.

En TypeScript, les annotations sont préfixées par `@`, et peuvent être appliquées sur une classe, une propriété de classe, une fonction, ou un paramètre de fonction. Pas sur un constructeur en revanche, mais sur ses paramètres oui.

Pour mieux comprendre ces décorateurs, essayons d'en construire un très simple par nous-mêmes, `@Log()`, qui va écrire le nom de la méthode à chaque fois qu'elle sera appelée.

Il s'utilisera comme ça :

```
class RaceService {

  @Log()
  getRaces() {
    // call API
  }

  @Log()
  getRace(raceId) {
    // call API
  }
}
```

Pour le définir, nous devons écrire une méthode renvoyant une fonction comme celle-ci :

```
let Log = function () {
  return (target: any, name: string, descriptor: any) => {
    logger.log(`call to ${name}`);
    return descriptor;
  };
};
```

Selon ce sur quoi nous voulons appliquer notre décorateur, la fonction n'aura pas exactement les mêmes arguments. Ici nous avons un décorateur de méthode, qui prend 3 paramètres :

- `target` : la méthode ciblée par notre décorateur
- `name` : le nom de la méthode ciblée
- `descriptor` : le descripteur de la méthode ciblée, par exemple est-ce que la méthode est énumérable, etc...

Nous voulons simplement écrire le nom de la méthode, mais tu pourrais faire pratiquement ce que tu veux : modifier les paramètres, le résultat, appeler une autre fonction, etc...

Ici, dans notre exemple basique, chaque fois que les méthodes `getRace()` ou `getRaces()` sont executées, nous verrons une nouvelle trace dans la console du navigateur :

```
    raceService.getRaces();
    // logs: call to getRaces
    raceService.getRace(1);
    // logs: call to getRace
```

En tant qu'utilisateur d'Angular 2, jetons un œil à ces annotations :

```
@Component({selector: 'home'})
class Home {

  constructor(@Optional() hello: HelloService) {
    logger.log(hello);
  }

}
```

L'annotation `@Component` est ajoutée à la classe `Home`. Quand Angular 2 chargera notre application, il va trouver la classe `Home`, et va comprendre que c'est un composant grâce au décorateur. Cool, hein ?! Comme tu le vois, une annotation peut recevoir des paramètres, ici un objet de configuration.

Je voulais juste présenter le concept de décorateur, nous aurons l'occasion de revoir tous les décorateurs disponibles en Angular 2 tout au long de ce livre.

Je dois aussi indiquer que tu peux utiliser les décorateurs avec Babel comme transpileur à la place de TypeScript. Il y a même un plugin pour supporter tous les décorateurs Angular 2 : angular2-annotations. Babel supporte aussi les propriétés de classe, mais pas le système de type apporté par TypeScript. Tu peux utiliser Babel, et écrire du code "ES6+", mais tu ne pourras pas bénéficier des types, et ils sont sacrément utiles pour l'injection de dépendances. C'est possible, mais tu devras ajouter d'autres décorateurs pour remplacer les informations de type.

Ainsi mon conseil est d'essayer TypeScript. Tous mes exemples dans ce livre l'utiliseront. Il n'est pas intrusif, tu peux l'utiliser quand c'est utile et l'oublier le reste du temps. Si vraiment tu n'aimes pas, il ne sera pas très compliqué de repasser à ES6 avec Babel ou Traceur, ou même ES5 si tu es complètement fou (mais pour ma part, je trouve le code ES5 d'une application Angular 2 vraiment pas terrible !).

Chapter 6. Le monde merveilleux des Web Components

Avant d'aller plus loin, j'aimerais faire une petite pause pour parler des Web Components. Vous n'avez pas besoin de connaître les Web Components pour écrire du code Angular 2. Mais je pense que c'est une bonne chose d'en avoir un aperçu, car en Angular 2 certaines décisions ont été prises pour faciliter leur intégration, ou pour rendre les composants que l'on construit similaires à des Web Components. Tu es libre de sauter ce chapitre si tu ne t'intéresses pas du tout au sujet, mais je pense que tu apprendras deux-trois choses qui pourraient t'être utiles pour la suite.

6.1. Le nouveau Monde

Les composants sont un vieux rêve de développeur. Un truc que tu prendrais sur étagère et lâcherais dans ton application, et qui marcherait directement et apporterait la fonctionnalité à tes utilisateurs sans rien faire.

Mes amis, cette heure est venue.

Oui, bon, peut-être. En tout cas, on a le début d'un truc.

Ce n'est pas complètement neuf. On avait déjà la notion de composants dans le développement web depuis quelques temps, mais ils demandaient en général de lourdes dépendances comme jQuery, Dojo, Prototype, AngularJS, etc… Pas vraiment le genre de bibliothèques que tu veux absolument ajouter à ton application.

Les Web Components essaient de résoudre ce problème : avoir des composants réutilisables et encapsulés.

Ils reposent sur un ensemble de standards émergents, que les navigateurs ne supportent pas encore parfaitement. Mais quand même, c'est un sujet intéressant, même si on ne pourra pas en bénéficier pleinement avant quelques années, ou même jamais sur le concept ne décolle pas.

Ce standard émergent est défini dans quatre spécifications :

- Custom elements ("éléments personnalisés")
- Shadow DOM ("DOM de l'ombre")
- Template
- HTML imports

Note que les exemples présentés ont plus de chances de fonctionner dans un Chrome ou un Firefox récent.

6.2. Custom elements

Les éléments customs sont un nouveau standard qui permet au développeur de créer ses propres éléments du DOM, faisant de `<pony-cmp><pony-cmp>` un élément HTML parfaitement valide. La spécification définit comment déclarer de tels éléments, comment tu peux les faire étendre des éléments existants, comment tu peux définir ton API, etc…

Déclarer un élément custom se fait avec un simple `document.registerElement('pony-cmp')` :

```
// new element
var PonyCmp = document.registerElement('pony-cmp');
// insert in current body
document.body.appendChild(new PonyCmp());
```

Note que le nom doit contenir un tiret, pour indiquer au navigateur que c'est un élément custom.

Évidemment ton élément custom peut avoir des propriétés et des méthodes, et il aura aussi des callbacks liés au cycle de vie, pour exécuter du code quand le composant est inséré ou supprimé, ou quand l'un de ses attributs est modifié. Il peut aussi avoir son propre template. Par exemple, peut-être que ce `pony-cmp` affiche une image du poney, ou seulement son nom :

```
// let's extend HTMLElement
var PonyCmpProto = Object.create(HTMLElement.prototype);
// and add some template using a lifecycle
PonyCmpProto.createdCallback = function() {
  this.innerHTML = '<h1>General Soda</h1>';
};

// new element
var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
// insert in current body
document.body.appendChild(new PonyCmp());
```

Si tu jettes un coup d'œil au DOM, tu verras `<pony-cmp><h1>General Soda</h1></pony-cmp>`. Mais cela veut dire que le CSS ou la logique JavaScript de ton application peut avoir des effets indésirables sur ton composant. Donc, en général, le template est caché et encaspulé dans un truc appelé le Shadow DOM ("DOM de l'ombre"), et tu ne verras dans le DOM que `<pony-cmp><pony-cmp>`, bien que le navigateur affiche le nom du poney.

6.3. Shadow DOM

Avec un nom qui claque comme celui-là, on s'attend à un truc très puissant. Et il l'est. Le Shadow DOM est une façon d'encapsuler le DOM de ton composant. Cette encapsulation signifie que la feuille de style

et la logique JavaScript de ton application ne vont pas s'appliquer sur le composant et le ruiner accidentellement. Cela en fait l'outil idéal pour dissimuler le fonctionnement interne de ton composant, et s'assurer que rien n'en fuit à l'extérieur.

Si on retourne à notre exemple précédent :

```
var PonyCmpProto = Object.create(HTMLElement.prototype);

// add some template in the Shadow DOM
PonyCmpProto.createdCallback = function() {
  var shadow = this.createShadowRoot();
  shadow.innerHTML = '<h1>General Soda</h1>';
};

var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
document.body.appendChild(new PonyCmp());
```

Si tu essaies maintenant de l'observer, tu devrais voir :

```
<pony-cmp>
  #shadow-root (open)
    <h1>General Soda</h1>
</pony-cmp>
```

Désormais, même si tu ajoutes du style aux éléments `h1`, rien ne changera : le Shadow DOM agit comme une barrière.

Jusqu'à présent, nous avions utilisé une chaîne de caractères pour notre template. Mais ce n'est habituellement pas la façon de procéder. La bonne pratique est de plutôt utiliser l'élément `<template>`.

6.4. Template

Un template spécifié dans un élément `<template>` n'est pas affiché par le navigateur. Son but est d'être à terme cloné dans un autre élément. Ce que tu déclareras à l'intérieur sera inerte : les scripts ne s'exécuteront pas, les images ne se chargeront pas, etc... Son contenu peut être requêté par le reste de la page avec la méthode classique `getElementById()`, et il peut être placé sans risque n'importe où dans la page.

Pour utiliser un template, il doit être cloné :

```
<template id="pony-tpl">
  <style>
    h1 { color: orange; }
  </style>
  <h1>General Soda</h1>
</template>

var PonyCmpProto = Object.create(HTMLElement.prototype);

// add some template using the template tag
PonyCmpProto.createdCallback = function() {
  var template = document.querySelector('#pony-tpl');
  var clone = document.importNode(template.content, true);
  this.createShadowRoot().appendChild(clone);
};

var PonyCmp = document.registerElement('pony-cmp', {prototype: PonyCmpProto});
document.body.appendChild(new PonyCmp());
```

Et si on pouvait déclarer cela dans un seul fichier, cela nous ferait un composant parfaitement encapsulé… C'est ce que nous allons faire avec les imports HTML !

6.5. HTML imports

C'est la dernière spécification. Les imports HTML permettent d'importer du HTML dans du HTML. Quelque chose comme `<link rel="import" href="pony-cmp.html">`. Ce fichier, `pony-cmp.html`, contiendrait tout ce qui est requis : le template, les scripts, les styles, etc…

Si quelqu'un voulait ensuite utiliser notre merveilleux composant, il lui suffirait simplement d'utiliser un import HTML.

6.6. Polymer et X-tag

Toutes ces spécifications constituent les Web Components. Je suis loin d'en être expert, et ils présentent toute sorte de pièges.

Comme les Web Components ne sont pas complètement supportés par tous les navigateurs, il y a un *polyfill* à inclure dans ton application pour être sûr que ça fonctionne. Ce *polyfill* est appelé webcomponent.js, et il est bon de noter qu'il est le fruit d'un effort commun entre Google, Mozilla et Microsoft, entre autres.

Au dessus de ce *polyfill*, quelques bibliothèques ont vu le jour. Elles proposent toutes de faciliter le travail avec les Web Components, et viennent souvent avec un lot de composants tout prêts.

Parmi les initiatives notables, on peut citer :

- Polymer de Google ;
- X-tag de Mozilla et Microsoft.

Je ne vais pas rentrer dans les détails, mais tu peux facilement utiliser un composant Polymer existant. Supposons que tu veuilles embarquer une carte Google dans ton application :

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Import element -->
<link rel="import" href="google-map.html">

<!-- Use element -->
<body>
  <google-map latitude="45.780" longitude="4.842"></google-map>
</body>
```

Il y a une tonne de composants disponibles. Tu peux en avoir un aperçu sur https://customelements.io/.

Polymer permet aussi de créer tes propres composants :

```
<dom-module id="pony-cmp">
  <template>
    <h1>[[name]]</h1>
  </template>
  <script>
    Polymer({
      is: 'pony-cmp',
      properties: {
        name: String
      }
    });
  </script>
</dom-module>
```

et de les utiliser :

```
<!-- Polyfill Web Components support for older browsers -->
<script src="webcomponents.js"></script>

<!-- Polymer -->
<link rel="import" href="polymer.html">

<!-- Import element -->
<link rel="import" href="pony-cmp.html">

<!-- Use element -->
<body>
  <pony-cmp name="General Soda"></pony-cmp>
</body>
```

Tu peux faire plein de trucs cools avec Polymer, comme du binding bi-directionnel, donner des valeurs par défaut aux attributs, émettre des événements custom, réagir aux modifications d'attribut, répéter des éléments si tu fournis une collection à un composant, etc…

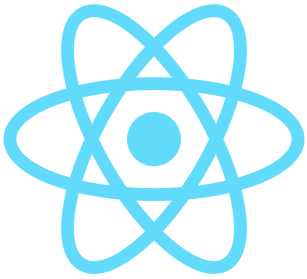
C'est un chapitre trop court pour te montrer sérieusement tout ce que l'on peut faire avec les Web Components, mais tu verras que certains de leurs concepts vont émerger dans la lecture à venir. Et tu verras sans aucun doute que l'équipe Google a conçu Angular 2 pour rendre facile l'utilisation des Web Components aux côtés de nos composants Angular 2.

Chapter 7. La philosophie d'Angular 2

Pour construire une application Angular 2, il te faut saisir quelques trucs sur la philosophie du framework.

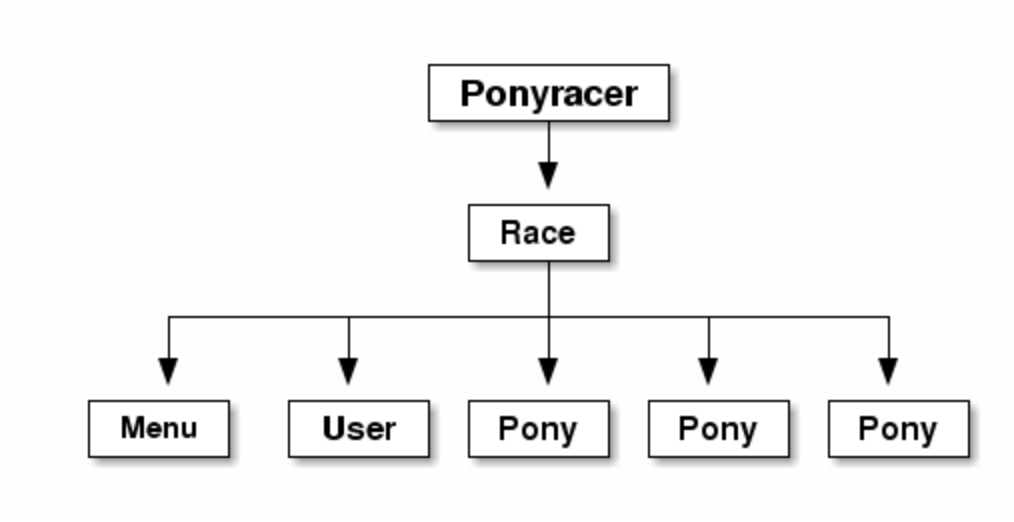
Avant tout, Angular 2 est un framework orienté composant. Tu vas écrire de petits composants, et assemblés, ils vont constituer une application complète. Un composant est un groupe d'éléments HTML, dans un template, dédiés à une tâche particulière. Pour cela, tu auras probablement besoin d'un peu de logique métier derrière ce template, pour peupler les données, et réagir aux événements par exemple. Pour les vétérans d'AngularJS 1.x, c'est un peu comme le fameux duo "template / controlleur", ou une directive.

Il faut aussi dire qu'un standard a été défini autour de ces composants : le standard *Web Component* ("composant web"). Même s'il n'est pas encore complètement supporté dans les navigateurs, tu peux déjà construire des petits composants isolés, réutilisables dans différentes applications (ce vieux rêve de programmeur). Cette orientation composant est largement partagée par de nombreux frameworks front-end : c'est le cas depuis le début de ReactJS, le framework tendance de Facebook ; EmberJS et AngularJS ont leur propre façon de faire quelque chose de similaire ; et les petits nouveaux Aurelia ou Vue.js parient aussi sur la construction de petits composants.



Angular 2 n'est donc pas le seul sur le sujet, mais ils sont parmi les premiers (ou le premier ?) à considérer sérieusement l'intégration des Web Components (ceux du standard officiel). Mais écartons ce sujet, trop avancé pour le moment.

Tes composants seront organisés de façon hiérarchique, comme le DOM : un composant racine aura des composants enfants, qui auront chacun des composants enfants, etc… Si tu veux afficher une course de poneys (qui ne voudrait pas ?), tu auras probablement une application (Poneyracer), avec une vue enfant (Race), affichant un menu (Menu), l'utilisateur connecté (User), et, évidemment, les poneys (Pony) en course :



Comme tu vas écrire des composants tous les jours (de la semaine au moins), regardons de plus près à quoi ça ressemble. L'équipe Angular voulait aussi bénéficier d'une autre pépite du développement web moderne : ES6 (ou ES2015, si tu préfères). Ainsi tu peux écrire tes composants en ES5 (pas cool !) ou en ES6 (super cool !). Mais cela ne leur suffisait pas, ils voulaient utiliser une fonctionnalité qui n'est pas

encore standard : les décorateurs. Alors ils ont travaillé étroitement avec les équipes de transpileurs (Traceur et Babel) et l'équipe Microsoft du projet TypeScript, pour nous permettre d'utiliser des décorateurs dans nos applications Angular 2. Quelques décorateurs sont disponibles, permettant de déclarer facilement un composant et sa vue. J'espère que tu es au courant, parce que je viens de consacrer deux chapitres à ces sujets !

Par exemple, en simplifiant, le composant Race pourrait ressembler à ça :

```
import {Component} from 'angular2/core';
import {Pony} from './components';
import {RacesService} from './services';

@Component({
  selector: 'race',
  templateUrl: 'race/race.html',
  directives: [Pony]
})
export class RaceCmp {

  race: any;

  constructor(racesService: RacesService) {
    racesService.get()
      .then(race => this.race = race);
  }
}
```

Et le template pourrait ressembler à ça :

```
<div>
  <h2>{{ race.name }}</h2>
  <div>{{ race.status }}</div>
  <div *ngFor="#pony of race.ponies">
    <pony [pony]="pony"></pony>
  </div>
</div>
```

Si tu connais déjà AngularJS 1.x, le template doit t'être familier, avec les même expressions entre accolades `{{ }}`, qui seront évaluées et remplacées par les valeurs correspondantes. Certains trucs ont cependant changé : plus de `ng-repeat` par exemple. Je ne veux pas aller trop loin pour le moment, juste te donner un aperçu du code.

Un composant est une partie complètement isolée de ton application. Ton application *est* un composant comme les autres. Dans un monde idéal, tu prendrais n'importe quel composant fourni par la communauté et en bénéficierais dans ton application, en l'ajoutant quelque part dans la hiérarchie.

Dans les chapitres suivants, on explorera quoi mettre en place, comment construire un petit composant, et la syntaxe des templates.

Il y a un autre concept au cœur d'Angular 2 : l'injection de dépendance (*Dependency Injection*, DI). C'est un pattern très puissant, et tu seras très rapidement séduit après la lecture du chapitre qui lui sera consacré. C'est particulièrement utile pour tester ton application, et j'adore faire des tests, et voir la barre de progression devenir entièrement verte dans mon IDE. Ça me donne l'impression de faire du bon boulot. Il y aura ainsi un chapitre entier consacré à tout tester : tes composants, tes services, ton interface…

Angular a toujours cette sensation magique de la v1, où les modifications sont automatiquement détectées par le framework et appliquées au modèle et à la vue. Mais c'est fait d'une façon très différente : la détection de changement utilise désormais un concept nommé `zones`. On étudiera évidemment tout ça.

Angular est aussi un framework complet, avec plein d'outils pour faciliter les tâches classiques du développement web. Construire des formulaires, appeler un serveur HTTP, du routage d'URL, interagir avec d'autres bibliothèques, des animations, tout ce que tu veux : c'est possible !

Voilà, ça fait pas mal de trucs à apprendre ! Alors commençons par le commencement : initialiser une application et construire notre premier composant.

Chapter 8. Commencer de zéro

8.1. Créer une application avec TypeScript

Commençons par créer notre première application Angular 2 et notre premier composant, avec un minimum d'outillage. Tu devras installer Node.js et NPM sur ton système. Comme la meilleure façon de le faire dépend de ton système d'exploitation, le mieux est d'aller voir le site officiel. Assure-toi d'avoir une version suffisamment récente de Node.js (en exécutant `node --version`), quelque-chose comme 4.2+. On va écrire notre application en TypeScript, donc tu devras l'installer aussi, via `npm` :

```
npm install -g typescript
```

Ensuite, tu vas créer un nouveau répertoire pour notre expérimentation, et utiliser `tsc` depuis ce nouveau répertoire vide pour y initialiser un projet. `tsc` sont les initiales de **TypeScript Compiler** ("compilateur TypeScript"). Il est fourni par le module NPM `typescript` qu'on vient d'installer globalement :

```
tsc --init --target es5 --sourceMap --experimentalDecorators --emitDecoratorMetadata
```

Cela va créer un fichier, `tsconfig.json`, qui stockera les options de compilation TypeScript. Comme on l'a vu dans les chapitres précédents, on va utiliser TypeScript avec des décorateurs (d'où les deux derniers flags), et on veut que notre code soit transpilé en ECMASCRIPT 5, lui permettant d'être exécuté par tout navigateur. L'option `sourceMap` permet de générer les *source maps* ("dictionnaires de code source"), c'est-à-dire des fichiers assurant le lien entre le code ES5 généré et le code TypeScript originel.

Ces *source maps* sont utilisés par le navigateur pour te permettre de débugger le code ES5 qu'il exécute en parcourant le code TypeScript originel que tu as écrit.

On va maintenant utiliser notre IDE préféré. Tu peux utiliser à peu près ce que tu veux, mais tu devrais y activer le support de TypeScript pour plus de confort (et t'assurer qu'il supporte TypeScript 1.5+) Choisis ton IDE préféré: Webstorm, Atom, VisualStudio Code… Ils ont tous un bon support de TypeScript.

Le compilateur TypeScript (et aussi souvent l'IDE) s'appuie sur le fichier `tsconfig.json` pour savoir quelles options il doit utiliser. Le fichier devrait ressembler à celui-ci :

```
{
  "compilerOptions": {
    "target": "es5",
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "module": "commonjs",
    "noImplicitAny": false,
    "outDir": "built",
    "rootDir": "."
  },
  "exclude": [
    "node_modules"
  ]
}
```

Tu peux voir que quelques options ont été ajoutées par défaut. Une intéressante est l'option `module`, qui indique que notre code va être packagé sous forme de modules CommonJS. Cela deviendra important dans un moment.

Maintenant on est prêt à lancer le compilateur TypeScript, en utilisant le *watch mode* ("mode de surveillance") pour compiler en tâche de fond dès la sauvegarde. Il se peut aussi que ton IDE puisse s'en charger.

```
tsc --watch
```

Cela devrait afficher quelque chose comme :

```
Compilation complete. Watching for file changes.
```

Tu peux désormais laisser ce compilateur tourner en fond et ouvrir une nouvelle ligne de commande pour la suite.

On doit maintenant ajouter la bibliothèque Angular 2 et notre code. Pour la bibliothèque Angular 2, on va la télécharger avec NPM, un chouette outil pour gérer ses dépendances.

Pour éviter quelques problèmes, nous allons utiliser NPM version 2. Vérifie quelle version tu as avec :

```
npm -v
```

Si tu as NPM version 3, tu peux facilement installer NPM version 2 :

```
npm install -g npm@2
```

Maintenant que c'est fait, commençons par créer le fichier `package.json`, qui contient toutes les informations dont NPM a besoin. Tu peux répondre par `Entrée` à toutes les questions.

```
npm init
```

Ensuite, installons `angular2` et ses dépendances :

```
npm install --save angular2
```

Tu peux jeter un coup d'œil au fichier `package.json`, il devrait désormais contenir les dépendances suivantes :

- `reflect-metadata`, parce que nous utilisons les décorateurs.
- `es6-shim` et `es6-promise` sont incluses, pour être sûr que le navigateur aura tout ce dont il a besoin.
- `rxjs`, une bibliothèque vraiment cool appelée RxJS pour la programmation réactive. On aura un chapitre entier consacré à ce sujet.
- et enfin, le module `zone.js`, qui assure la plomberie pour faire tourner notre code dans des zones isolées et y détecter les changements (on y reviendra aussi plus tard).

L'outillage est désormais en place, il est temps de créer notre premier composant !

8.2. Notre premier composant

Crée un nouveau fichier, nommé `ponyracer_app.ts`.

Dès que tu sauveras ce fichier, tu devrais voir apparaître un nouveau fichier `ponyracer_app.js` dans le répertoire `built` : c'est le compilateur TypeScript qui fait son travail. Sinon, c'est que tu as probablement arrêté ton compilateur TypeScript qui surveillait les changements, alors tu devrais le lancer à nouveau avec `tsc --watch`, et le laisser tourner en fond.

Comme on l'a vu dans le chapitre précédent, un composant est la combinaison d'une vue (le template) et de logique (notre classe TS). Créons une classe :

```
export class PonyRacerApp {

}
```

Notre application elle-même est un simple composant. Pour indiquer à Angular que c'en est un, on

utilise le décorateur `@Component`. Pour pouvoir l'utiliser, il nous faut l'importer :

```
import {Component} from 'angular2/core';

@Component()
export class PonyRacerApp {

}
```

Si ton IDE le supporte, la complétion de code devrait fonctionner car la dépendance Angular 2 a ses propres fichiers `d.ts` dans le répertoire `node_modules`, et TypeScript est capable de le détecter. Tu peux même naviguer vers les définitions de type si tu le souhaites.

TypeScript apporte sa vérification de types, donc tu verras les erreurs dès que tu les tapes. Mais les erreurs ne sont pas nécessairement bloquantes : si tu as oublié d'ajouter les informations de type sur ta variable, le code compilera quand même en JavaScript et s'exécutera correctement.

J'essaie de mon côté de garder le compte des erreurs TypeScript à zéro, mais tu peux faire comme tu veux. Comme nous utilisons des *source maps*, tu peux voir le code TypeScript directement dans ton navigateur, et même débugger ton application en positionnant des points d'arrêt directement dedans.

Le décorateur `@Component` attend un objet de configuration. On verra plus tard en détails ce qu'on peut y configurer, mais pour le moment une seule propriété est requise : `selector`. Elle indiquera à Angular ce qu'il faudra chercher dans nos pages HTML. A chaque fois que le sélecteur défini sera trouvé dans notre HTML, Angular remplacera l'élément sélectionné par notre composant :

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app'
})
export class PonyRacerApp {

}
```

Donc ici, chaque fois que notre HTML contiendra un élément `<ponyracer-app></ponyracer-app>`, Angular créera une nouvelle instance de notre classe `PonyRacerApp`.

Un composant doit aussi avoir un template. On pourrait externaliser le template dans un autre fichier, mais contentons-nous de faire simple la première fois, et mettons le en ligne :

```
import {Component} from 'angular2/core';

@Component({
  selector: 'ponyracer-app',
  template: '<h1>PonyRacer</h1>'
})
export class PonyRacerApp {

}
```

N'oublie pas d'importer le décorateur `Component` si tu veux l'utiliser. Tu l'oublieras régulièrement au début, mais tu t'y feras vite, parce que le compilateur ne va cesser de t'insulter ! ;)

Tu verras que l'essentiel de nos besoins se situe dans le module `angular2/core`, mais ce n'est pas toujours le cas. Par exemple, quand on fera du HTTP, on utilisera des imports de `angular2/http`, ou quand on utilisera le routeur, on importera depuis `angular2/router`, etc...

8.3. Démarrer l'application

Enfin, on doit démarrer l'application. Pour cela, il y a une méthode `bootstrap` à notre disposition. Il nous faut aussi l'importer, depuis `angular2/platform/browser`. C'est quoi ce module bizarre ?! Pourquoi n'est-ce pas `angular2/core` ? Bonne question : c'est parce que tu pourrais avoir envie de faire tourner ton application ailleurs que dans un navigateur, parce qu'Angular 2 permet le rendu côté serveur, ou peut tourner dans un Web Worker par exemple. Et dans ces cas, la logique de démarrage sera un peu différente. Mais on verra cela plus tard, on va se contenter d'un navigateur pour le moment.

Créons un nouveau fichier, par exemple `bootstrap.ts`, pour séparer la logique de démarrage :

```
import {bootstrap} from 'angular2/platform/browser';
import {PonyRacerApp} from './ponyracer_app';

bootstrap(PonyRacerApp)
 .catch(err => console.log(err)); // useful to catch the errors
```

Youpi ! Mais attends voir. On n'a pas encore de fichier HTML, si ? T'as raison !

Crée un autre fichier nommé `index.html` et ajoutes-y le contenu suivant :

```
<html>

<head></head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

Il nous faut maintenant ajouter nos scripts dans nos fichiers HTML. En AngularJS 1.x, c'était simple : il te suffisait d'ajouter un script pour angular.js, et un script pour chacun des fichiers JS que tu avais écrits, et c'était bon. Il y avait cependant un inconvénient : tout devait être chargé statiquement, dès le démarrage, ce qui pouvait allonger lourdement les temps de chargement pour les grosses applications.

Avec Angular 2, c'est plus complexe, mais aussi bien plus puissant. Angular est maintenant modulaire, et chaque module peut être chargé dynamiquement. Notre application est donc aussi modulaire, comme on l'a vu.

Il y a cependant quelques problèmes :

- la notion de module n'existe pas en ES5, et les navigateurs ne supportent que ES5 pour le moment ;
- les concepteurs d'ES6 ont décidé de spécifier comment les modules étaient définis et importés. Mais ils n'ont pas encore spécifié comment ils devaient être packagés et chargés par les navigateurs.

Pour charger nos modules, il nous faudra donc s'appuyer sur un outil : SystemJS. SystemJS est un petit chargeur de modules : tu l'ajoutes (statiquement) dans ta page HTML, tu lui indiques où sont situés les modules sur le serveur, et tu charges l'un d'eux. Il déterminera automatiquement les dépendances entre les modules, et téléchargera ceux utilisés par ton application.

Cela va entraîner des pelletés de téléchargements de fichiers JS. Si cela n'est pas un problème pendant le développement, ça le devient en production. Heureusement, SystemJS vient aussi avec un outil qui peut emballer plusieurs petits modules dans un plus gros paquet. Quand un module est requis, le paquet contenant ce module (et plusieurs autres) sera alors téléchargé.

Installons SystemJS :

```
npm install --save systemjs
```

On doit charger `SystemJS` statiquement, et lui indiquer où se situe notre module contenant la logique de démarrage (dans `built/bootstrap`). On doit aussi lui montrer où trouver les dépendances de notre application, comme `angular2`. Mais d'abord, il nous faut inclure `angular2-polyfills`, un script spécial

qui doit être chargé en premier, pour inclure `reflect-metadata` et `zone.js`:

```
<html>

<head>
  <script src="node_modules/angular2/bundles/angular2-polyfills.js"></script>
  <script src="node_modules/systemjs/dist/system.js"></script>
  <script>
  System.config({
    // we want to import modules without writing .js at the end
    defaultJSExtensions: true,
    // the app will need the following dependencies
    map: {
      'angular2': 'node_modules/angular2',
      'rxjs': 'node_modules/rxjs'
    }
  });
  // and to finish, let's boot the app!
  System.import('built/bootstrap');
  </script>
</head>

<body>
  <ponyracer-app>
    You will see me while Angular starts the app!
  </ponyracer-app>
</body>

</html>
```

OK ! Démarrons maintenant un serveur HTTP pour servir notre mini application. Je vais utiliser `http-server`, un outil Node.js qui fait ce que laisse deviner son nom. Mais tu peux évidemment utiliser ton serveur web préféré : Apache, Nginx, Tomcat, etc… Pour l'installer, on utilisera `npm`:

```
npm install -g http-server
```

Pour le démarrer, va dans ton répertoire, et entre :

```
http-server
```

Maintenant c'est la grande première ! Ouvre ton navigateur sur http://localhost:8080.

Tu devrais y voir brièvement "You will see this while Angular start the app!", puis ensuite "PonyRacer" devrait apparaître ! Ton premier composant est un succès !

Bon, OK, pour le moment ce n'est pas vraiment une application dynamique, on aurait pu faire la même chose en une seconde dans une page HTML statique. Alors jetons-nous sur les chapitres suivants, et apprenons tout de l'injection de dépendances et du système de templates.

8.4. Commencer de zéro avec angular-cli

Dans un vrai projet, il te faudra probablement mettre en place d'autres choses comme :

- des tests pour vérifier que tu n'as pas introduit de régressions ;
- un outil de construction, pour orchestrer différentes tâches (compiler, tester, packager, etc…)

Et c'est un peu laborieux de tout mettre en place tout seul, même si je pense que c'est utile de le faire au moins une fois pour comprendre tout ce qui se passe.

Ces dernières années, plusieurs petits projets ont vu le jour, tous basés sur le formidable Yeoman. C'était déjà le cas avec AngularJS 1.x, et il y a déjà plusieurs tentatives avec Angular 2.

Mais cette fois-ci, l'équipe Google a travaillé sur le sujet, et ils en ont sorti ceci : `angular-cli`.

`angular-cli` est un outil en ligne de commande pour démarrer rapidement un projet, déjà configuré avec un outil de construction (Broccoli), des tests, du packaging, etc…

Cette idée n'est pas nouvelle, et est d'ailleurs piquée d'un autre framework populaire : EmberJS et son `ember-cli` largement plébiscité.

L'outil est encore en développement, mais je pense qu'il va devenir le standard de fait pour créer une application Angular 2 dans le futur, alors tu peux l'essayer :

```
npm i -g angular-cli
ng new ponyracer
```

Cela va créer un squelette de projet. Tu peux démarrer l'application avec :

```
ng serve
```

Cela va démarrer un petit serveur HTTP localement, avec rechargement à chaud. Ainsi, à chaque modification de fichier, l'application sera rafraîchie dans le navigateur.

Tu peux aussi créer un squelette de composant :

```
ng component pony
```

Cela va créer un fichier de composant, avec son template associé, sa feuille de style, et un fichier de

test.

L'outil n'est pas seulement là pour nous aider à développer notre application : il vient avec un système de plugins qui vont faciliter d'autres tâches comme le déploiement. Par exemple, tu peux rapidement déployer sur Github Pages, avec le plugin `github-pages` :

```
ng github-pages:deploy
```

À terme, cela devrait être formidable ! On partagera la même organisation du code à travers les projets, une façon commune de construire et déployer les applications, et probablement un large écosystème de plugins pour simplifier certaines tâches.

Alors n'hésite pas à jeter un œil à `angular-cli` !

Chapter 9. Fin de l’extrait gratuit

Voilà, j’espère que ce que tu as lu t’aura comblé, et que tu rêves désormais de lire la suite ! :)