Become a

NINJA

with

ANGULAR 2

ninja squad

Become a ninja with Angular2 (free sample)

Ninja Squad

Table of Contents

1. Free sample 1
2. Introduction 2
3. A gentle introduction to ECMASCRIPT 6 4
   3.1. Transpilers 4
   3.2. let 5
   3.3. Constants 6
   3.4. Creating objects 7
   3.5. Destructuring assignment 7
   3.6. Default parameters and values 9
   3.7. Rest operator 11
   3.8. Classes 12
   3.9. Promises 15
   3.10. Arrow functions 18
   3.11. Sets and Maps 22
   3.12. Template literals 23
   3.13. Modules 23
   3.14. Conclusion 25
4. Going further than ES6 26
   4.1. Dynamic, static and optional types 26
   4.2. Enters TypeScript 27
   4.3. A practical example with DI 27
5. Diving into TypeScript 30
   5.1. Types as in TypeScript 30
   5.2. Enums 31
   5.3. Return types 31
   5.4. Interfaces 32
   5.5. Optional arguments 32
   5.6. Functions as property 33
   5.7. Classes 33
   5.8. Working with other libraries 35
   5.9. Decorators 36
6. The wonderful land of Web Components 39
   6.1. A brave new world 39
   6.2. Custom elements 39
   6.3. Shadow DOM 40
   6.4. Template 41

6.5. HTML imports 42
6.6. Polymer and X-tag 42
7. Grasping Angular's philosophy 45
8. From zero to something 49
8.1. Developing and building a TypeScript app 49
8.2. Our first component 51
8.3. Our first Angular Module 53
8.4. Bootstrapping the app 55
8.5. From zero to something better with angular-cli 59
9. End of the free sample 61
Appendix A: Changelog 62
A.1. Changes since last release - 2016-10-17 62
A.2. v1.2 - 2016-08-25 62
A.3. v1.1 - 2016-05-11 64

Chapter 1. Free sample

What you're going to read is a free sample of our Angular 2 ebook: the starting part of the full book, which explains its purpose and contents, gives an overview of ECMAScript 6, TypeScript, and Web Components, describes Angular 2 philosophy, and finally lets you write your first application.

This sample extract does not require any preliminary knowledge.

Chapter 2. Introduction

So you want to be a ninja, huh? Well, you're in good hands!

But we have a long road, you and me, with lots of things to learn :).

We're living exciting times in Web development. There is a new Angular. A complete rewrite of the good old AngularJS. Why a complete rewrite? Was AngularJS 1.x not enough?

I like the old AngularJS very much. In our small company, we have built several projects with it, contributed code to the core framework, trained hundreds of developers (yes, really), and even written a book about it (in French, but that still counts).

AngularJS is incredibly productive once you have mastered it. Despite all of this, it doesn't prevent us from seeing its weaknesses. AngularJS is not perfect, with some very difficult concepts to grasp, and traps hard to avoid.

Most of all, the Web has changed since AngularJS was conceived. JavaScript has changed. New frameworks have emerged, with great ideas, or better implementation. We are not the kind of developers to tell you that you should use this tool instead of that one. We just happen to know some tools very well, and know what fits the project. AngularJS was one of those tools, allowing us to build well-tested web applications, and to build them fast. We also tried to bend it where it didn't fit. Don't blame us, it happens to the best of us.

Will Angular 2 be the tool we will use without hesitation in our future projects? It's hard to say right now, because the framework is really young and the ecosystem only just blooming.

But Angular 2 has a lot of interesting points, and a vision that few other frameworks have. It has been designed for the Web of tomorrow, with ECMAScript 6, Web Components and Mobile in mind. When it was first announced, I was, like many, sad at first that the 2.0 version would not be a simple update (I'm sorry if you're just learning about it).

But I was also eager to see what solution the talented Google team would come up with.

So I started to write this ebook, pretty much after the first commits, reading the design docs, watching the conference videos, reviewing every commit since the beginning. When I wrote my first ebook, about AngularJS 1.x, it was already a stable and known beast. This one is very different, it started when Angular 2 was not even clear in the minds of its designers. Because I knew I would learn a lot, not only about Angular but also about the concepts that would shape the future of Web development, some of which have nothing to do with Angular. And I did. I had to dig a lot about some of these concepts, and I hope that you will enjoy the journey of learning about them, and how they relate to Angular, as much as I did.

The ambition of this ebook is to evolve with Angular. If it turns out that Angular is the great framework we hope, you will receive updates with the best practices and some new features as they emerge (and with less typos, because, despite our countless reviews, there are probably some left…).

And I would love to hear back from you - if some chapters aren't clear enough, if you spot a mistake or if you have a better way for some parts.

I'm fairly confident about the code samples, though, as they are all in a real project, with several hundred unit tests. It was the only way to write an ebook with a newborn framework, and to be able to catch all the problems that inevitably arose with each release.

Even if you are not convinced by Angular in the end, I'm pretty sure you will have learnt a thing or two along your read.

If you have bought the "Pro package" (thank you!), you'll build a small application piece by piece along the book. This application is called **PonyRacer**, and it is a website where you can bet on pony races. You can even test the application here! Go on, I'll wait for you.

Fun, isn't it?

But it's not just a fun application, it's a complete one. You'll have to write components, forms, tests, use the router, call an HTTP API (that we have already built) and even do Web Sockets. It has all the pieces you'll need for writing a real app. Each exercise will come with a skeleton, a set of instructions and a few tests. Once you have all the tests in success, you have completed the exercise!

If you did not buy the "Pro package" (but really you should), don't worry: you'll learn everything that's needed. But you will not build this awesome application with beautiful ponies in pixel art. Your loss :)!

You will quickly see that, beyond Angular itself, we have tried to explain the core concepts the framework uses. The first chapters don't even talk about Angular: they are what I call the "Concept Chapters", here to help you level up with the new and exciting things happening in our field.

Then we will slowly build our knowledge of the framework, with components, templates, pipes, forms, http, routing, tests...

And finally we will learn about the advanced topics. But that's another story.

Enough with the introduction, let's start with one of the things that will definitely change the way we code: ECMAScript 6.

NOTE	The ebook is using Angular 2 version 2.1.0 for the examples.

Chapter 3. A gentle introduction to ECMASCRIPT 6

If you're reading this, we can be pretty sure you have heard of JavaScript. What we call JS is one implementation of a standard specification, called ECMAScript. The spec version you know the most about is the version 5, that has been used these last years.

But recently, a new version of the spec has been in the works, called ECMASCRIPT 6, ES6, or ECMASCRIPT 2015. From now on, I'll mainly say ES6, as it is the most popular way to reference it. It adds A LOT of things to JavaScript, like classes, constants, arrow functions, generators... It has so much stuff that we can't go through all of it, as it would take the whole book. But Angular 2 has been designed to take advantage of the brand new version of JavaScript. And, even if you can still use your old JavaScript, things will be more awesome if you use ES6. So we're going to spend some time in this chapter to get a grip on what ES6 is, and what will be useful to us when building an Angular app.

That means we're going to leave a lot of stuff aside, and we won't be exhaustive on the rest, but it will be a great starting point. If you already know ES6, you can skip these pages. And if you don't, you will learn some pretty amazing things that will be useful to you even if you end up not using Angular in the future!

3.1. Transpilers

ES6 has just reached its final state, so it's not yet fully supported by every browser. And, of course, some browsers will always be late to this game (even if, for once, Microsoft is doing a good job with Edge). You might be thinking: what's the use of all this, if I need to be careful on what I can use? And you'd be right, because there aren't that many apps that can afford to ignore older browsers. But, since virtually every JS developer who has tried ES6 wants to write ES6 apps, the community has found a solution: a transpiler.

A transpiler takes ES6 source code and generates ES5 code that can run in every browser. It even generates the source map files, which allows to debug directly the ES6 source code from the browser. At the time of writing, there are two main alternatives to transpile ES6 code:

- Traceur, a Google project
- Babeljs, a project started by a young developer, Sebastian McKenzie (17 years old at the time, yeah, that hurts me too), with a lot of diverse contributions.

Each has its own pros and cons. For example, Babeljs produces a more readable source code than Traceur. But Traceur is a Google project, so, of course, Angular and Traceur play well together. The source code of Angular 2 itself was at first transpiled with Traceur, before switching to TypeScript. TypeScript is an open source language developed by Microsoft. It's a typed superset of JavaScript that compiles to plain JavaScript, but we'll dive into it very soon.

Let's be honest Babel has waaaay more steam than Traceur, so I would advice you to use it. It is quickly becoming the de-facto standard in this area.

So if you want to play with ES6, or set it up in one of your projects, take a look at these transpilers, and add a build step to your process. It will take your ES6 source files and generate the equivalent ES5 code. It works very well but, of course, some of the new features are quite hard or impossible to transform in ES5, as they just did not exist. However, the current state is largely good enough for us to use without worrying, so let's have a look at all these shiny new things we can do in JavaScript!

3.2. let

If you have been writing JS for some time, you know that the `var` declaration is tricky. In pretty much any other languages, a variable is declared where the declaration is done. But in JS, there is a concept, called "hoisting", which actually declares a variable at the top of the function, even if you declared it later.

So declaring a variable like `name` in the `if` block:

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

is equivalent to declaring it at the top of the function:

```
function getPonyFullName(pony) {
  var name;
  if (pony.isChampion) {
    name = 'Champion ' + pony.name;
    return name;
  }
  // name is still accessible here
  return pony.name;
}
```

ES6 introduces a new keyword for variable declaration, `let`, behaving much more like what you would expect:

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

The variable `name` is now restricted to its block. `let` has been introduced to replace `var` in the long run, so you can pretty much drop the good old `var` keyword and start using `let` instead. The cool thing is, it should be painless to use `let`, and if you can't, you have probably spotted something wrong with your code!

3.3. Constants

Since we are on the topic of new keywords and variables, there is another one that can be of interest. ES6 introduces `const` to declare... constants! When you declare a variable with `const`, it has to be initialized and you can't assign another value later.

```
const poniesInRace = 6;
```

```
poniesInRace = 7; // SyntaxError
```

As for variables declared with `let`, constants are not hoisted and are only declared at the block level.

One small thing might surprise you: you can initialize a constant with an object and later modify the object content.

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

But you can't assign another object:

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

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

Same thing with arrays:

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

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

3.4. Creating objects

Not a new keyword, but it can also catch your attention when reading ES6 code. There is now a shortcut for creating objects, when the object property you want to create has the same name as the variable used as the value.

Example:

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

can be simplified to

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

3.5. Destructuring assignment

This new feature can also catch your attention when reading ES6 code. There is now a shortcut for assigning variables from objects or arrays.

In ES5:

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

Now, in ES6, you can do:

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

And you will have the same result. It can be a little disturbing, as the key is the property to look for into the object and the value is the variable to assign. But it works great! Even better: if the variable you want to assign has the same name as the property, you can simply write:

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

The cool thing is that it also works with nested objects:

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

And the same is possible with arrays:

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

Of course it also works for arrays in arrays, or arrays in objects, etc...

One interesting use of this can be for multiple return values. Imagine a function `randomPonyInRace` that returns a pony and its position in a race.

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

const { position, pony } = randomPonyInRace();
```

The new destructuring feature is assigning the position returned by the method to the position variable, and the pony to the pony variable! And if you don't care about the position, you can write:

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

const { pony } = randomPonyInRace();
```

And you will only have the pony!

3.6. Default parameters and values

One of the characteristics of JavaScript is that it allows developers to call a function with any number of arguments:

- if you pass more arguments than the number of the parameters, the extra arguments are ignored (well, you can still use them with the special `arguments` variable, to be accurate).
- if you pass less arguments than the number of the parameters, the missing parameter will be set to `undefined`.

The last case is the one that is the most relevant to us. Usually, we pass less arguments when the parameters are optional, like in the following example:

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

The optional parameters usually have a default value. The OR operator will return the right operand if the left one is `undefined`, as will be the case if the parameter was not provided (to be completely accurate, if it is *falsy*, i.e `0`, `false`, `""`, etc.). Using this trick, the function `getPonies` can then be called:

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

This worked alright, but it was not really obvious that the parameters were optional ones with default values, without reading the function body. ES6 introduces a more precise way to have default parameters, directly in the function definition:

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

Now it is perfectly clear that the `size` parameter will be `10` and the `page` parameter will be `1` if not provided.

NOTE There is a small difference though, as now `0` or `""` are valid values and will not be replaced by the default one, as `size = size || 10` would have done. It will be more like `size = size === undefined ? 10: size;`.

The default value can also be a function call:

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

or even other variables, either global variables, or other parameters of the function:

```
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 that if you try to access parameters on the right, their value is always `undefined`:

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

This mechanism for parameters can also be applied to values, for example when using a destructuring assignment:

```
const { 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 introduces a new syntax to define variable parameters in functions. As said in the previous part, you could always pass extra arguments to a function and get them with the special `arguments` variable. So you could have done something like that:

```
function addPonies(ponies) {
  for (var i = 0; i < arguments.length; i++) {
    poniesInRace.push(arguments[i]);
  }
}

addPonies('Rainbow Dash', 'Pinkie Pie');
```

But I think we can agree that it's neither pretty nor obvious: since the `ponies` parameter is never used, how do we know that we can pass several ponies?

ES6 gives us a way better syntax, using the rest operator `...`:

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

`ponies` is now a true array on which we can iterate. The `for ... of` loop used for iteration is also a new feature in ES6. It allows to be sure to iterate over the collection values, and not also on its properties as `for ... in` would do. Don't you think our code is prettier and more obvious now?

The rest operator can also work when destructuring data:

```
const [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
```

The rest operator is not to be confused with the spread operator which, I'll give you that, looks awfully similar! But the spread operator is the opposite: it takes an array and spreads it in variable arguments. The only examples I have in mind are functions like min or max, that receive variable arguments, and that you might want to call on an array:

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

3.8. Classes

One of the most emblematic new features, and one that we will vastly use when writing an Angular app: ES6 introduces classes to JavaScript! You can now easily use classes and inheritance in JavaScript. You always could, using prototypal inheritance, but that it was not an easy task, especially for beginners.

Now it's very easy, take a look:

```
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!
  }
}
const bluePony = new Pony('blue');
console.log(bluePony.toString()); // blue pony
```

Class declarations, unlike function declarations, are not hoisted, so you need to declare a class before using it. You may have noticed the special function `constructor`. It is the function being called when we create a new pony, with the `new` operator. Here it needs a color, and we create a new Pony instance with the color set to "blue". A class can also have methods, callable on an instance, as the method `toString()` here.

It can also have static attributes and methods:

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

Static methods can be called only on the class directly:

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

A class can have getters and setters, if you want to hook on these operations:

```
class Pony {
  get color() {
    console.log('get color');
    return this._color;
  }

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

And, of course, if you have classes, you also have inheritance out of the box in ES6.

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

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

Animal is called the base class, and Pony the derived class. As you can see, the derived class has the methods of the base class. It can also override them:

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

As you can see, the keyword `super` allows calling the method of the base class, with `super.speed()` for example.

The `super` keyword can also be used in constructors, to call the base class constructor:

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

3.9. Promises

Promises are not so new, and you might know them or use them already, as they were a big part of AngularJS 1.x. But since you will use them a lot in Angular 2, and even if you're just using JS, I think it's important to make a stop.

Promises aim to simplify asynchronous programming. Our JS code is full of async stuff, like AJAX requests, and usually we use callbacks to handle the result and the error. But it can get messy, with callbacks inside callbacks, and it makes the code hard to read and to maintain. Promises are much nicer than callbacks, as they flatten the code, and thus make it easier to understand. Let's consider a simple use case, where we need to fetch a user, then their rights, then update a menu when we have these.

With callbacks:

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

Now, let's compare it with promises:

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

I like this version, because it executes as you read it: I want to fetch a user, then get their rights, then update the menu.

As you can see, a promise is a 'thenable' object, which simply means it has a `then` method. This method takes two arguments: one success callback and one reject callback. The promise has three states:

- pending: while the promise is not done, for example, our server call is not completed yet.
- fulfilled: when the promise is completed with success, for example, the server call returns an OK HTTP status.
- rejected: when the promise has failed, for example, the server returns a 404 status.

When the promise is fulfilled, then the success callback is called, with the result as an argument. If the promise is rejected, then the reject callback is called, with a rejected value or an error as the argument.

So, how do you create a promise? Pretty simple, there is a new class called `Promise`, whose constructor expects a function with two parameters, `resolve` and `reject`.

```
const getUser = function (login) {
  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');
    }
  });
};
```

Once you have created the promise, you can register callbacks, using the `then` method. This method can receive two parameters, the two callbacks you want to call in case of success or in case of failure. Here we only pass a success callback, ignoring the potential error:

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

Once the promise is resolved, the success callback (here simply logging the user on the console) will be called.

The cool part is that it flattens the code. For example, if your resolve callback is also returning a promise, you can write:

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

but more beautifully:

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

Another interesting thing is the error handling, as you can use one handler per promise, or one for all the chain.

One per promise:

```
getUser(login)
  .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);
  })
```

One for the chain:

```
getUser(login)
  .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
  })
```

You should seriously look into Promises, because they are going to be the new way to write APIs, and every library will use them. Even the standard ones: the new Fetch API does for example.

3.10. Arrow functions

One thing I like a lot in ES6 is the new arrow function syntax, using the 'fat arrow' operator (⇒). It is SO useful for callbacks and anonymous functions!

Let's take our previous example with promises:

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

can be written with arrow functions like this:

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

How cool is it? THAT cool!

Note that the return is also implicit if there is no block: no need to write `user ⇒ return getRights(user)`. But if we did have a block, we would need the explicit return:

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

And it has a special trick, a great power over normal functions: the `this` stays lexically bounded, which means that these functions don't have a new `this` as other functions do. Let's take an example, where you are iterating over an array with the `map` function to find the max.

In 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);
```

You would expect this to work, but it doesn't. If you have a good eye, you may have noticed that the `forEach` in the `find` function uses `this`, but the `this` is not bound to an object. So `this.max` is not the `max` of the `maxFinder` object... Of course you can fix it easily, using an 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);
```

or binding the `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);
```

or even passing it as a second parameter of the `forEach` function (as it was designed for):

```
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);
```

But there is now an even more elegant solution with the arrow function syntax:

```
const 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);
```

That makes the arrow functions the perfect candidates for anonymous functions in callbacks!

3.11. Sets and Maps

This is a short one: you now have proper collections in ES6. Yay \o/! We used to have dictionaries filling the role of a map, but we can now use the class `Map`:

```
const cedric = { id: 1, name: 'Cedric' };
const 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
```

We also have a class `Set`:

```
const cedric = { id: 1, name: 'Cedric' };
const 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
```

You can iterate over a collection, with the new syntax `for ⋯ of`:

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

You'll see that the `for ... of` syntax is the one the Angular team chose to iterate over a collection in a template.

3.12. Template literals

Composing strings has always been painful in JavaScript, as we usually have to use concatenation:

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

Template literals are a new small feature, where you have to use backticks (`) instead of quotes or simple quotes, and you have a basic templating system, with multiline support:

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

The multiline support is especially great when your are writing HTML strings, as we will do for our Angular components:

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

3.13. Modules

A standard way to organize functions in namespaces and to dynamically load code in JS has always been lacking. NodeJS has been one of the leaders in this, with a thriving ecosystem of modules using the CommonJS convention. On the browser side, there is also the AMD (Asynchronous Module Definition) API, used by RequireJS. But none of these were a real standard, thus leading to endless discussions on what's best.

ES6 aims to create a syntax using the best from both worlds, without caring about the actual implementation. The Ecma TC39 committee (which is responsible for evolving ES6 and authoring the specification of the language) wanted to have a nice and easy syntax (that's arguably CommonJS's strong suit), but to support asynchronous loading (like AMD), and a few goodies like the possibility to statically analyze the code by tools and support cyclic dependencies nicely. The new syntax handles how you export and import things to and from modules.

This module thing is really important in Angular 2, as pretty much everything is defined in modules,

that you have to import when you want to use them. Let's say I want to expose a function to bet on a specific pony in a race and a function to start the race.

In races_service.js:

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

As you can see, this is fairly easy: the new keyword `export` does a straightforward job and exports the two functions.

Now, let's say one of our application components needs to call these functions.

In another file:

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

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

That's what is called a *named export*. Here we are importing the two functions, and we have to specify the filename containing these functions - here 'races_service'. Of course, you can import only one method if you need, you can even give it an alias:

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

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

And if you need to import all the methods from the module, you can use a wildcard '*'.

As you would do with other languages, use the wildcard with care, only when you really want all the functions, or most of them. As this will be analyzed by our IDEs, we will see auto-import soon and that will free us from the bother to import the right things.

With a wildcard, you have to use an alias, and I kind of like it, because it makes the rest of the code

clearer:

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

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

If your module exposes only one function or value or class, you don't have to use named export, and you can leverage the default keyword. It works great for classes for example:

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

Notice the lack of curly braces to import a default. You can import it with the alias you want, but to be consistent, it's better to call the import with the module name (except if you have multiple modules with the same name of course, then you can chose an alias that allows to distinguish them). And of course, you can mix default export with named ones, but obviously with only one default per module.

In Angular 2, you're going to use a lot of these imports in your app. Each component and service will be a class, generally isolated in their own file and exported, and then imported when needed in other components.

3.14. Conclusion

That ends our gentle introduction to ES6. We skipped some other parts, but if you're comfortable with this chapter, you will have no problem writing your apps in ES6. If you want to have a deeper understanding of this, I highly recommend Exploring JS by Axel Rauschmayer or Understanding ES6 from Nicholas C. Zakas ... Both ebooks can be read online for free, but don't forget to buy it to support their authors, they have done a great work! Actually I've re-read Speaking JS, Axel's previous book, and I again learned a few things, so if you want to refresh your JS skills, I definitely recommend it!

Chapter 4. Going further than ES6

4.1. Dynamic, static and optional types

You may have heard that Angular 2 apps can be written in ES5, ES6 or TypeScript. And you may be wondering what TypeScript is, or what it brings to the table.

JavaScript is dynamically typed. That means you can do things like:

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

And it works. That's great for all sort of things, as you can pass pretty much any object to a function and it works, as long as the object has the properties the function needs:

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

This dynamic nature allows wonderful things but it is also a pain for a few others compared to more statically-typed languages. The most obvious might be when you call an unknown function in JS from another API, you pretty much have to read the doc (or, worse, the function code) to know what the parameter should look like. Take a look at our previous example: the method `printColor` needs a parameter with a `color` property. That can be hard to guess, and of course it is much worse in day-to-day work, where we use various libraries and services developed by fellow developers. One of Ninja Squad's co-founders is often complaining about the lack of types in JS, and finds it regrettable he can't be as productive and write as good code as he would in a more statically-typed environment. And he is not entirely wrong, even if he is sometimes ranting for the sake of it too! Without type information, IDEs have no real clue if you're doing something wrong, and tools can't help you find bugs in your code. Of course, we have tests in our apps, and Angular has always been keen on making testing easy, but it's nearly impossible to have a perfect test coverage.

That leads to the maintainability topic. JS code can become hard to maintain, despite tests and documentation. Refactoring a huge JS app is no easy task, compared to what could be done in other statically-typed languages. Maintainability is a very important topic, and types help humans and tools to avoid mistakes when writing and maintaining code. Google has always been keen to push new solutions in that direction: it's easy to understand as they have some of the biggest web apps of the world, with GMail, Google apps, Maps... So they have tried several approaches to front-end maintainability: GWT, Google Closure, Dart... All trying to help writing big webapps.

For Angular 2, the Google team wanted to help us writing better JS, by adding some type information to

our code. It's not a very new concept in JS, it was even the subject of the ECMASCRIPT 4 specification, which was later abandoned. At first they announced AtScript, as a superset of ES6 with annotations (types annotations and another kind I'll discuss later). They also announced the support of TypeScript, the Microsoft language, with additional type annotations. And then, a few months later, the TypeScript team announced that they had worked closely with the Google team, and the new version of the language (1.5) would have all the shiny new things AtScript had. And the Google team announced that AtScript was officially dropped, and that TypeScript was the new top-notch way to write Angular 2 apps!

4.2. Enters TypeScript

I think this was a smart move for several reasons. For one, no one really wants to learn another language extension. And TypeScript was already there, with an active community and ecosystem. I never really used it before Angular 2, but I heard good things on it, from various people. TypeScript is a Microsoft project. But it's not the Microsoft you have in mind, from the Ballmer and Gates years. It's the Microsoft of the Nadella era, the one opening to its community, and, well, open-source. Google knows this, and it's far better for them to contribute to an existing project, rather than to have to bear the burden to maintain their own. And the TypeScript framework will gain a huge popularity boost: win-win, as your manager would say.

But the main reason to bet on TypeScript is the type system it offers. It's an optional type system that helps without getting in the way. In fact, after coding some time with it, I forgot about it: you can do Angular 2 apps using TypeScript just for the parts where it really helps (more on that in a second) and simply ignore it everywhere else and write plain JS (ES6 in my case). But I do like what they have done, and we will have a look at what TypeScript offers in the next section. At the end, you'll have enough understanding to read any Angular 2 code, and you'll be able to choose whether you want to use it or not (or just a little), in your apps.

You may be wondering: why use typed code in Angular 2 apps? Let's take an example. Angular 1 and 2 have been built around a powerful concept named "dependency injection". You might already be familiar with it, as it is a common design pattern used in several frameworks for different languages and, as I said, already used in AngularJS 1.x.

4.3. A practical example with DI

To sum up what dependency injection is, think about a component of the app, let's say `RaceList`, needing to access the races list that the service `RaceService` can give. You would write `RaceList` like this:

```
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!
  }
}
```

But it has several flaws. One of them is the testability: it is now very hard to replace the `raceService` by a fake (mock) one, to test our component.

If we use the Dependency Injection (DI) pattern, we delegate the creation of the `RaceService` to the framework, and we simply ask for an instance. The framework is now in charge of the creation of the dependency, and, well, injects it:

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

Now, when we test this class, we can easily pass a fake service to the constructor:

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

But how does the framework know what to inject in the constructor? Good question! AngularJS 1.x relied on the parameter's names, but it had a severe limitation, because minification of your code would have changed the param name… You could use the array syntax to fix this, or add a metadata to the class:

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

We had to add some metadata for the framework to understand what classes needed to be injected with. And that's exactly what type annotations give: a metadata giving the framework a hint it needs to do the right injection. In Angular 2, using TypeScript, we can write our `RaceList` component like:

```
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);
  }
}
```

Now the injection can be done! You don't have to use TypeScript in Angular 2, but clearly part of your code will be more elegant if you do. You can always do the same thing in plain ES6 or ES5, but you will have to manually add the metadata in another way (we'll come back on this in more details).

That's why we're going to spend some time learning TypeScript (TS). Angular 2 is clearly built to leverage ES6 and TS 1.5+, so we will have the easiest time writing our apps using it. And the Angular team really hopes to submit the type system to the standard committee, so maybe one day we'll have types in JS, and all this will be usual.

Let's dive in!

Chapter 5. Diving into TypeScript

TypeScript has been around since 2012. It's a superset of JavaScript, adding a few things to ES5. The most important one is the type system, giving TypeScript its name. From version 1.5, released in 2015, the library is trying to be a superset of ES6, including all the shiny features we saw in the previous chapter, and a few new things as well, like decorators. Writing TypeScript feels very much like writing JavaScript. By convention, TypeScript files are named with a `.ts` extension, and they will need to be compiled to standard JavaScript, usually at build time, using the TypeScript compiler. The generated code is very readable.

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

But let's start with the beginning.

5.1. Types as in TypeScript

The general syntax to add type info in TypeScript is rather straightforward:

```
let variable: type;
```

The types are easy to remember:

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

In such cases, the types are optional because the TS compiler can guess them (it's called "type inference") from the values.

The type can also be coming from your app, as with the following class `Pony`:

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

TypeScript also supports what some languages call "generics", for example for an array:

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

The array can only contain ponies, and the generic notation, using `<>`, indicates this. You may be wondering what the point of doing this is. Adding types information will help the compiler catch

possible mistakes:

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

So, if you need a variable to have multiple types, does it mean you're screwed? No, because TS has a special type, called `any`.

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

It's really useful when you don't know the type of a value, either because it's from a dynamic content or from a library you're using.

If your variable can only be of type `number` or `boolean`, you can use a union type:

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

5.2. Enums

TypeScript also offers `enum`. For example, a race in our app can be either `ready`, `started` or `done`.

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

The enum is in fact a numeric value, starting at 0. You can set the value you want, though:

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

5.3. Return types

You can also set the return type of a function:

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

If the function returns nothing, you can show it using `void`:

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

5.4. Interfaces

That's a good first step. But as I said earlier, JavaScript is great for its dynamic nature. A function will work if it receives an object with the correct property:

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

This function can be applied to any object with a `score` property. How do you translate this in TypeScript? It's easy: you define an interface, like the "shape" of the object.

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

It means that the parameter must have a property called `score` of the type `number`. You can name these interfaces, of course:

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

5.5. Optional arguments

Another treat of JavaScript is that arguments are optional. You can omit them, and they will become `undefined`. But if you define a function with typed parameter in TypeScript, the compiler will shout at you if you forget them:

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

To show that a parameter is optional in a function (or a property in an interface), you can add `?` after the parameter. Here, the `points` parameter could be optional:

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

5.6. Functions as property

You may also be interested in describing a parameter that must have a specific function instead of a property:

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

The interface definition will be:

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

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

5.7. Classes

A class can implement an interface. For us, the `Pony` class should be able to run, so we can write:

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

The compiler will force us to implement a `run` method in the class. If we implement it badly, by expecting a `string` instead of a `number` for example, the compiler will yell:

```
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.
```

You can also implement several interfaces if you want:

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

  eat() {
    logger.log(`pony eats`);
  }
}
```

And an interface can extend one or several others:

```
interface Animal extends CanRun, CanEat {}

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

When you're defining a class in TypeScript, you can have properties and methods in your class. You may realize that properties in classes are not a standard ES6 feature, it is only possible in TypeScript.

```
class SpeedyPony {
  speed: number = 10;

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

Everything is public by default, but you can use the `private` keyword to hide a property or a method. If

you add `private` or `public` to a constructor parameter, it is a shortcut to create and initialize a private or public member:

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

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

Which is the same as the more verbose:

```
class NamedPonyWithoutShortcut {
  public name: string;
  private speed: number;

  constructor(name: string, speed: number) {
    this.name = name;
    this.speed = speed;
  }

  run() {
    logger.log(`pony runs at ${this.speed}m/s`);
  }
}
```

These shortcuts are really useful and we'll rely on them a lot in Angular 2!

5.8. Working with other libraries

When working with external libraries written in JS, you may think we are doomed because we don't know what types of parameter the function in that library will expect. That's one of the cool things with the TypeScript community: its members have defined interfaces for the types and functions exposed by the popular JavaScript libraries!

The files containing these interfaces have a special `.d.ts` extension. They contain a list of the library's public functions. A good place to look for these files is DefinitelyTyped. For example, if you want to use TS in your AngularJS 1.x apps, you can download the proper file from the repo directly with NPM:

```
npm install --save-dev @types/angular
```

or download it manually. Then include the file at the top of your code, and enjoy the compilation checks:

```
/// <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" />` is a special comment recognized by TS, telling the compiler to look for the interface `angular.d.ts`. Now, if you misuse an AngularJS method, the compiler will complain, and you can fix it on the spot, without having to manually run your app!

Even cooler, since TypeScript 1.6, the compiler will auto-discover the interfaces if they are packaged in your `node_modules` directory in the dependency. More and more projects are adopting this approach, and so is Angular 2. So you don't even have to worry about including the interfaces in your Angular 2 project: the TS compiler will figure it out by itself if you are using NPM to manage your dependencies!

5.9. Decorators

This is a fairly new feature, added only in TypeScript 1.5, to help supporting Angular. Indeed, as we will shortly see, Angular 2 components can be described using decorators. You may not have heard about decorators, as not every language has them. A decorator is a way to do some meta-programming. They are fairly similar to annotations which are mainly used in Java, C# and Python, and maybe other languages I don't know. Depending on the language, you add an annotation to a method, an attribute, or a class. Generally, annotations are not really used by the language itself, but mainly by frameworks and libraries.

Decorators are really powerful: they can modify their target (method, classes, etc...), and for example alter the parameters of the call, tamper with the result, call other methods when the target is called or add metadata for a framework (which is what Angular 2 decorators do). Until now, it was not something possible in JavaScript. But the language is evolving and there is now an official proposal for `decorators`, which may be standardized one day in the future (possibly in ES7/ES2016). Note that the TypeScript implementation goes slightly further than the proposed standard.

In Angular 2, we will use the decorators provided by the framework. Their role is fairly basic: they add some metadata to our classes, attributes or parameters to say things like "this class is a component", "this is an optional dependency", "this is a custom property", etc... It's not required to use them, as you can add the metadata manually (if you want to stick to ES5 for example), but the code will definitely be more elegant using decorators, as provided by TypeScript.

In TypeScript, decorators start with an `@`, and can be applied to a class, a class property, a function or a

function parameter. They can't be applied to a constructor, but can be applied to its parameters.

To have a better grasp on this, let's try to build a simple decorator, `@Log()`, that will log something every time a method is called.

It will be used like this:

```
class RaceService {

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

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

To define it, we have to write a method returning a function like this:

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

Depending on what you want to apply your decorator to, the function will not have exactly the same arguments. Here we have a method decorator that takes 3 parameters:

- `target`: the method targeted by our decorator
- `name`: the name of the targeted method
- `descriptor`: a descriptor of the targeted method (is the method enumerable, writable, etc...)

Here we simply log the method name, but you could do pretty much whatever you want: interfere with the parameters, the result, calling another function, etc...

So, in our simple example, every time the `getRace()` or `getRaces()` methods are called, we'll see a trace in the browser logs:

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

As a user, let's look at what a decorator in Angular 2 looks like:

```
@Component({ selector: 'ns-home' })
class HomeComponent {

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

}
```

The `@Component` decorator is added to the class `Home`. When Angular 2 loads our app, it will find the class `Home` and will understand that it is a component, based on the metadata the decorator will add. Cool, huh? As you can see, a decorator can also receive parameters, here a configuration object.

I just wanted to introduce the raw concept of decorators; we'll look into every decorator available in Angular all along the book.

I have to point out that you can use decorators with Babel as a transpiler instead of TypeScript. There is even a plugin to support all the Angular 2 decorators: angular2-annotations. Babel also supports class properties, but not the type system offered by TypeScript. You can use Babel, and write "ES6+" code, but you will not be able to use the types, and they are very useful for the dependency injection for example. It's completely possible, but you'll have to add more decorators to replace the types.

So my advice would be to give TypeScript a try! All my examples from here will use it. It's not very intrusive, as you can use it just where it's useful and forget about it for the rest. If you really don't like it, it will not be very difficult to switch to ES6 with Babel or Traceur, or even ES5, if you are slightly crazy (but honestly, an Angular 2 app in ES5 has pretty ugly code).

Chapter 6. The wonderful land of Web Components

Before going further, I'd like to make a brief stop to talk about Web Components. You don't have to know about Web Components to write Angular 2 code. But I think it's a good thing to have an overview of what they are, because some choices in Angular 2 have been made to facilitate the integration with Web Components, or to make the components we will build similar to Web Components. Feel free to skip this part if you have no interest in this topic; however, I do believe you'll learn a thing or two that will be useful for the rest of the road.

6.1. A brave new world

Components are an old fantasy in development. Something you can grab off the shelves and drop into your app, something that would work right away and bring a needed functionality to your users.

My friends, this time has come.

Well, maybe. At least, there is the start of something.

That's not completely new. We have had components in web development for quite some time, but they usually require some kind of dependency, like jQuery, Dojo, Prototype, AngularJS, etc… Not necessarily libraries you wanted to add to your app.

Web Components attempt to solve this problem: let's have reusable and encapsulated components.

They rely on a set of emerging standards that browsers don't perfectly support yet. But, still, it's an interesting topic, even if there's a chance that we'll have to wait a few years to use them fully, or even that the concept never takes off.

This emerging standard is defined in 4 specifications:

- Custom elements
- Shadow DOM
- Template
- HTML imports

Note that the samples are most likely to work in a recent Chrome or Firefox browser.

6.2. Custom elements

Custom elements are a new standard allowing developers to create their own DOM elements, making something like `<ns-pony></ns-pony>` a perfectly valid HTML element. The specification defines how to declare such elements, how to make them extend existing elements, how to define your API, etc…

Declaring a custom element is done with a simple `document.registerElement('ns-pony')`:

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

Note that the name must contain a dash, so that the browser knows it is a custom element. Of course, your custom element can have properties and methods, and it also has lifecycle callbacks, to be able to execute code when the component is inserted or removed, or when one of its attributes changes. It can also have a template of its own. Maybe the `ns-pony` displays an image of the pony or just its name:

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

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

If you try to look at the DOM, you'll see `<ns-pony><h1>General Soda</h1></ns-pony>`. But that means the CSS and JavaScript logic of your app can have undesired effects on your component. So, usually, the template is hidden and encapsulated in something called Shadow DOM, and you'll only see `<ns-pony></ns-pony>` if you inspect the DOM, despite the fact that the browser displays the pony's name.

6.3. Shadow DOM

With a mysterious name like this, you expect something with great powers. And surely it is. The Shadow DOM is a way to encapsulate the DOM of our component. This encapsulation means that the stylesheet and JavaScript logic of your app will not apply on the component and ruin it inadvertently. It gives us the perfect tool to hide the internals of a component, and be sure that nothing leaks from the component to the app, or vice-versa.

Going back to our previous example:

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

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

var PonyComponent = document.registerElement('ns-pony', {prototype: PonyComponentProto});
document.body.appendChild(new PonyComponent());
```

If you try to inspect it now you should see:

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

Now, even if you try to add some style to the `h1` elements, the visual aspect of the component won't change at all: that's because the Shadow DOM acts like a barrier.

Until now, we just used a string as a template of our web component. But that's usually not the way you do that. Instead, the best practice is to use the <template> element.

6.4. Template

A template specified in a `<template>` element is not displayed in your browser. Its main goal is to be cloned in an element at some point. What you declare inside will be inert: scripts don't run, images don't load, etc... Its content can't be queried by the rest of the page using usual method like `getElementById()` and it can be safely placed anywhere in your page.

To use a template, it needs to be cloned:

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

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

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

var PonyComponent = document.registerElement('ns-pony', {prototype: PonyComponentProto});
document.body.appendChild(new PonyComponent());
```

Maybe we could declare this in a single file, and we would have a perfectly encapsulated component… Let's do this with HTML imports!

6.5. HTML imports

This is the last specification. HTML imports allow to import HTML into HTML. Something like `<link rel="import" href="ns-pony.html">`. This file, `ns-pony.html`, would contain everything needed: the template, the scripts, the styles, etc…

If someone wants to use our wonderful component, they just have to use an HTML import and they are good to go!

6.6. Polymer and X-tag

All these things put together make the Web Components. I'm far from being an expert on this topic, and there are all sorts of twisted traps on this road.

As Web Components are not fully supported by every browser, there is a polyfill you can include in your app to make sure it will work. The polyfill is called web-component.js, and it's worth noting that it is a joint effort from Google, Mozilla and Microsoft among others.

On top of this polyfill, a few libraries have seen the light. All aim to facilitate working with Web Components, and often come with some ready-to-use Web Components.

Among the most notable initiatives, you find:

- Polymer from Google
- X-tag from Mozilla and Microsoft

I won't go into the details, but you can easily use an already existing Polymer Component. Let's say you want a Google Map in your app:

```
<!-- 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>
```

There are a LOT of components out there. You can have an overview on https://customelements.io/.

Polymer also helps you build your own components:

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

and use them:

```
<!-- 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="ns-pony.html">

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

You can do a lot of cool things with Polymer, like two-way data binding, default values for attributes, emit custom events, react on attribute changes, repeat elements if we give a collection to a component, etc...

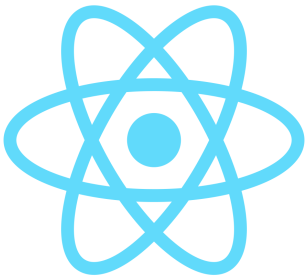
That's obviously far too short a chapter to tell you everything there is to say on Web Components, but you'll see that some of the concepts are going to pop out along your read. And you'll definitely see that the Google team designed Angular 2 to make it easy to use Web Components along our Angular 2 components.

Chapter 7. Grasping Angular’s philosophy

To write an Angular 2 application, you have to grasp a few things on the framework’s philosophy.

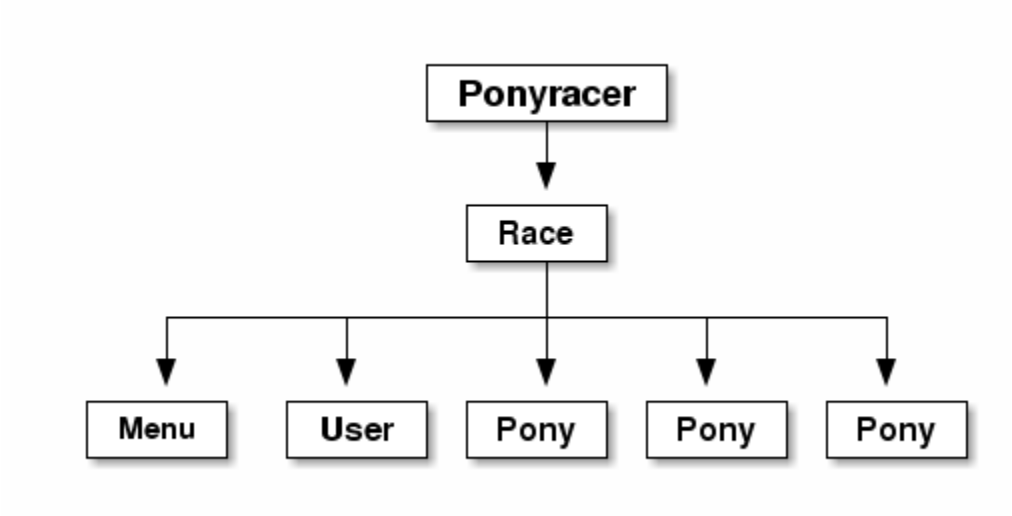
First and foremost, Angular 2 is component-oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, to populate data, and react to events for example. For the veterans of AngularJS 1.x, it’s a bit like a 'template/controller' duo, or a directive.

It has to be said that a standard has been established around this component thing: the Web Component standard. Even if it’s not completely supported by browsers yet, you can build small and isolated components, reusable in different applications - an old dream of computer programming. This component orientation is something that is becoming widely shared across front-end frameworks: ReactJS, the latest cool kid from Facebook, has been doing it that way from the beginning; EmberJS and AngularJS have their way of doing something similar; and newcomers like Aurelia or Vue.js are betting on building small components too.



Angular 2 is not alone in this, but it is among the first (it might actually be the first?) to really care about the integration of Web Components (the standard ones). But let's forget about this for now, as it is a more advanced topic.

Your components will be arranged in a hierarchical way, like the DOM is. A root component will have child components, each of them will also have children, etc… If you want to display a pony race (who wouldn't?), you'll have something like an app (Ponyracer), with a child view (Race), displaying a menu (Menu), the logged in user (User), and, of course, the ponies (Pony) in the races:



Writing components will be your everyday work, so let's see what it looks like. The Angular team wanted to harness another goodness of today's web development: ES6 (or ES2015, whatever you like to call it). So you can write your components in ES5 (but that's not very cool) or in ES6 (way cooler!). But that was not enough for them, they wanted to use a feature that is not a standard (yet): decorators. So they worked closely with the transpiler teams (Traceur and Babel) and the TypeScript team at Microsoft, to enable us to use decorators in our Angular 2 apps. A few decorators are available, allowing to easily declare a component for example. I hope you already know all of that, as I just spent two chapters on these things!

For example, if we simplify, the Race component could look like this:

```
import { Component } from '@angular/core';
import { RacesService } from './services';

@Component({
  selector: 'ns-race',
  templateUrl: 'race/race.html'
})
export class RaceComponent {

  race: any;

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

And the template looks like this:

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

If you already know AngularJS 1.x, the template should look familiar, with the same expression in curly braces `{{ }}`, that will be evaluated and replaced by the according value. Some things have changed though: no more `ng-repeat` for example. I don't want to go too deep for now, merely just give you a feel of what the code looks like.

A component is a very isolated piece of your app. Your app *is* a component like the others.

You will group components in one or several coherent entities, called modules (Angular Modules, not ES6 Modules).

In a perfect world, you will also take available modules from the community and just put them in your app, and be able to enjoy their features.

Such modules can offer UI components, or drag and drop capability, or validation for your forms, or whatever you can think of.

In the next chapters, we are going to explore how to get started, how to build a small component, your first module and the templating syntax.

There is another concept that is at the core, and that is Dependency injection (often called by its little name, DI). It is a very powerful pattern, and you will quickly get used to it after reading the dedicated chapter. It is especially useful to test your application, and I love doing tests, watching the progress bar go all green in my IDE. It makes me feel I'm doing a good job. So there will be an entire chapter on testing everything: your components, your services, your UI…

Angular still has the magic feeling it had in v1, where changes were automatically detected by the framework and applied to the model and the views. But it is done in a very different way than it was then: the change detection now uses a concept called `zones`. We will look into this, of course.

Angular is also a complete framework which provides a lot of help for performing common tasks in web development. Writing forms, calling an HTTP backend, routing, interacting with other libraries, animations, you name it: you're covered.

Well, that's a lot of things to learn! We should start with the beginning: bootstrap an app and write our first component.

Chapter 8. From zero to something

8.1. Developing and building a TypeScript app

Let's start by creating our first Angular 2 app and our first component, with a minimum of tooling. You'll have to install Node.js and NPM on your system. The best way to do that depends on your operating system - you can find more information on the official website. Make sure you have a recent enough version of Node.js (by executing `node --version`), something like 4.4+. We'll write our app in TypeScript, so you'll have to install it via `npm`:

```
npm install -g typescript
```

Then, create a new, empty folder for our experiment, and use `tsc` from that new empty folder to initialize a project. `tsc` stands for **TypeScript Compiler**. It's provided by the `typescript` NPM module we just installed globally:

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

This will create a file, `tsconfig.json`, which stores the TypeScript compilation options. As we saw in the previous chapters, we are using TypeScript with decorators (hence the last two flags), and we want our code to transpile to ECMASCRIPT 5, allowing it to run in every browser. The `sourceMap` option allows generating source maps, i.e. files that contain a mapping between the generated ES5 code and the original TypeScript code. Those source maps are used by the browser to let you debug the ES5 code it executes by stepping through the original TypeScript code that you have written.

We now want to start using our preferred IDE. You can use pretty much anything you want, but you should activate the TypeScript support for maximum comfort (and make sure you are using TypeScript 1.5+). Pick your favorite IDE: Webstorm, Atom, VisualStudio Code… All of them have great support for TypeScript.

The TypeScript compiler (and usually the IDE) relies on the `tsconfig.json` file to know what options it should use. The file should look like the following:

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

You can see that a few options have been added by default. An interesting one is the `module` option, telling us that our code will be packaged in CommonJS modules. It will be important in a moment.

Now we are ready to launch the TypeScript compiler, using the watch mode to compile the files when we save. Sometimes your IDE will do that for you.

```
tsc --watch
```

You should see something like:

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

You can let this run in the background and open a new terminal for what comes next.

We now need to add the Angular 2 library and our code. For the Angular 2 library, we are going to download it using NPM, a great tool to manage dependencies.

To avoid a few problems, we'll use NPM version 3. Check which version you have with:

```
npm -v
```

If you don't have NPM version 3, you can easily update NPM:

```
npm install -g npm
```

Now that's done, let's start by creating the `package.json` file, containing all the information that NPM needs. You can answer `Enter` to every question.

```
npm init
```

Then, let's install Angular 2 and its dependencies.

NOTE	The ebook is using Angular 2 version 2.1.0 for the examples. The following command will install the most recent version, which might not be the same. If you want to use the same version as us, add `@2.1.0` to each Angular package, like `@angular/core@2.1.0`. That might save you a few headaches! Angular 2 is really modular, so we have to install a few packages for the framework itself, and for its dependencies.

```
npm install --save @angular/core @angular/compiler @angular/common @angular/platform-
browser @angular/platform-browser-dynamic rxjs reflect-metadata zone.js
```

You can have a look at your `package.json` file, it should now contain the following dependencies:

- the different `@angular` packages.
- `reflect-metadata`, as we are using decorators.
- `rxjs`, a really cool library called RxJS for reactive programming. We have a dedicated chapter on this topic.
- and finally, the `zone.js` module, doing the heavy lifting of running our code in isolated zones for detecting the changes (we'll dive into this later also).

Last thing to make the compiler happy, you have to install the typings for the things related to ES6. The easiest way is to install the typings for `core-js`:

```
npm install --save-dev @types/core-js
```

The tooling is now in place, let's create our first component!

8.2. Our first component

Create a new file, called `app.component.ts`.

When you save your file, you should see a new file `app.component.js` popping in the directory: it's the TypeScript compiler doing its job. You should see the source map file as well. If not, you probably killed your TypeScript compiler watching for changes, so you might run it again with `tsc --watch`, and leave it running in the background.

As we saw in the previous section, a component is a combination of a view (the template) and some logic (our TS class). Let's create a class:

```
export class PonyRacerAppComponent {

}
```

Our application itself is a simple component. To tell Angular that it is a component, we use the `@Component` decorator. To be able to use it, we have to import it:

```
import { Component } from '@angular/core';

@Component()
export class PonyRacerAppComponent {

}
```

If your IDE supports it, code completion should work as the Angular 2 dependency has its own `d.ts` files in the `node_modules` directory, and TypeScript is able to detect it. You can even navigate to the type definitions if you want to.

TypeScript will bring its type-checking to the table, so you'll see what mistakes you make as you type. But the errors are not necessarily blocking: if you forget to add the type information to your variable, the code will still compile to JavaScript and run properly.

I try to keep the TypeScript errors count to 0, but you can do as you want. As we are using source maps, you can see the TS code directly from your browser, and even debug your app by setting breakpoints in the TypeScript code.

The `@Component` decorator is expecting a configuration object. We'll see later in details what you can configure here, but for now only one property is expected: the `selector` one. It will tell Angular what to look for in our HTML pages. Every time the selector we have defined is found in our HTML, Angular is going to replace the element selected by our component:

```
import { Component } from '@angular/core';

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

}
```

So, here, every time our HTML will contain an element like `<ponyracer-app></ponyracer-app>`, Angular will instantiate a new instance of our `PonyRacerAppComponent` class.

NOTE	There is not a clear naming convention established yet. I tend to suffix my component classes with `Component`. A component's selector should have a dash, like `ns-pony`, even if that's not mandatory. But, if you want to let other developers use your component and avoid potential name clashes, you should adopt a convention like "namespace-component". The namespace should be a short one, like "ns" for Ninja Squad for example. That would give a reusable component with a selector `ns-pony`. Finally, you can add a suffix to your filenames to see their role at first glance, `pony.component.ts` or `race.service.ts` for example.

A component must also have a template. We could externalize the template in another file, but for our first time, let's keep it simple, and inline it:

```
import { Component } from '@angular/core';

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

}
```

Don't forget to import the `Component` decorator. You may forget to do so at the beginning, but it won't last, as the compiler will yell at you! ;)

You'll see that most of the things we need are in the `@angular/core` module, but that's not always the case. For example, when dealing with HTTP, we'll use imports from `@angular/http`; or, if we use the router, we'll import from `@angular/router`, etc...

8.3. Our first Angular Module

Like we briefly said in the previous chapter, we are going to group our components and other pieces we'll see later in coherent entities: Angular Modules.

An Angular Module is different from the ES6 Modules we crossed earlier: here we are talking about **application** modules.

Your application will always have at least one module, the **root module**. Maybe, later, when your application grows, you'll add other modules, by feature. For example, you could add a module dedicated to the Admin part of your application, containing all the components and the logic for this part. But we'll come back to this later. We'll also see that third party libraries and Angular itself expose modules, that we can use in our app.

To define an Angular Module for our little app, we have to create a class. Usually, this is done in a

separate file, called `app.module.ts` for the root module.

The class has to be decorated with `@NgModule`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
@NgModule({
  imports: [BrowserModule],
})
export class AppModule {

}
```

Like the `@Component` decorator, it takes a configuration object.

As we are building an app for the browser, the root module should import `BrowserModule`. This is not the only target possible for Angular 2, you could choose to render the app on the server for example, and therefore import another module. `BrowserModule` contains all kinds of useful stuff we will use later. A module can choose to *export* components, directives and pipes. When you import a module, you will make all the directives, components and pipes exported by the imported module usable in your module. Our root module won't be imported in another module, so we don't have `exports`, but we will have several `imports` in the end.

The terminology is not beginner friendly here. We were talking about ES6 and TS modules in the first chapters, which define imports and exports. And now we are talking about Angular modules, which also have imports and exports… I'm not a fan of having the same terms for different things, so let me explain a little bit more.

You can see an ES6 or TS import purely as a language feature, like an import statement in Java: it allows using the imported classes/functions in your source code. It also declares a dependency for the bundler or module loader (Webpack or SystemJS, for example), which knows that if `a.ts` is loaded, then `b.ts` must also be loaded since `a` imports `b`. You have to use imports and exports with ES6 and TypeScript, whether or not you're using Angular or any other framework.

On the other hand, importing an Angular module (for example `BrowserModule`) in your own Angular module (`AppModule`), has a functional meaning. It tells Angular: all the components, directives and pipes that are exported by `BrowserModule` should be made available to my Angular components/templates. It has no special meaning for the TypeScript compiler.

Back to `NgModule`: in its configuration object, we must declare the components that belong to our root module, with the `declarations` field. Let's add the component we have carefully crafted: `PonyracerAppComponent`.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
})
export class AppModule {

}
```

Since this is the root module, we need to tell Angular which component is the root component, i.e. the component that will be started when we bootstrap the app. That's what the `bootstrap` field of the configuration object is for:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { PonyRacerAppComponent } from './app.component';

@NgModule({
  imports: [BrowserModule],
  declarations: [PonyRacerAppComponent],
  bootstrap: [PonyRacerAppComponent]
})
export class AppModule {

}
```

Module ready, let's bootstrap the app!

8.4. Bootstrapping the app

Finally, we need to start our app, using the `bootstrapModule` method. This method is exposed on an object returned by a method called `platformBrowserDynamic`. You have to import it too, from `@angular/platform-browser-dynamic`. Now, that's a strange module! Why is it not `@angular/core`? Good question: it's because you might want to run your app somewhere else than in a browser, as Angular 2 supports server-side rendering or running in a Web Worker for example. And in these cases, the bootstrap logic would be a bit different. But we'll see this later, as we are just focusing on the browser right now.

Let's create another file, for example `main.ts`, to separate the bootstrap logic:

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

platformBrowserDynamic().bootstrapModule(AppModule);
```

Yay! But wait a second. We don't have any HTML file, do we? You're right about that!

Create another file named `index.html` and add the following content:

```
<html>

<head></head>

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

</html>
```

We now need to add scripts to our HTML files. In AngularJS 1.x, it was simple: you just needed to add a script for angular.js, and a script for every JS file you wrote, and you were ready to go. There was a downside though: everything had to be loaded statically, at startup, which could lead to long startup times for big applications.

With Angular 2, things are more complicated, but complexity comes with additional power. Angular is now bundled in modules (ES6 modules), and these modules can be loaded dynamically. Our app is also bundled in modules, as we saw earlier.

There are a few problems though:

- modules don't exist in ES5, and browsers only support ES5 at the moment;
- the ES6 designers have decided to specify how modules are defined, imported, etc. But they have not yet specified how they're supposed to be packaged and loaded by the browsers.

To load our modules, we will thus need to rely on a tool: SystemJS. SystemJS is a small module loader: you add it (statically) into your HTML page, you tell it where modules are located on your server, and you load one of them. It automatically figures out the dependencies between modules, and downloads the ones used by your application.

This will lead to a bazillion of JS file downloads. That is fine during development, but it is a problem for production. Fortunately, SystemJS also comes with a tool that can pack several small modules into bigger bundles. When a module is needed, the bundle containing that module (and several other ones)

will then be downloaded.

Note that this is not the only tool that can do this job, you could use another one like Webpack if you wanted to.

Let's install SystemJS:

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

We need to load `SystemJS` statically, and to tell it where our bootstrap module is (in `main`). We also need to tell it where to find the dependencies of our application, like `@angular`. But first, we need to include `reflect-metadata` and `zone.js`:

```
<html>

<head>
  <script src="node_modules/zone.js/dist/zone.js"></script>
  <script src="node_modules/reflect-metadata/Reflect.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: {
        '@angular/core': 'node_modules/@angular/core/bundles/core.umd.js',
        '@angular/common': 'node_modules/@angular/common/bundles/common.umd.js',
        '@angular/compiler': 'node_modules/@angular/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': 'node_modules/@angular/platform-
browser/bundles/platform-browser.umd.js',
        '@angular/platform-browser-dynamic': 'node_modules/@angular/platform-browser-
dynamic/bundles/platform-browser-dynamic.umd.js',
        'rxjs': 'node_modules/rxjs'
      }
    });
    // and to finish, let's boot the app!
    System.import('main');
  </script>
</head>

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

</html>
```

OK! Let's start an HTTP server to serve this mini app. I'm going to use `http-server`, a node tool that does pretty much what its name says. But you may of course use whatever web server you prefer: Apache, Nginx, Tomcat, etc. To install it, use `npm`:

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

To start it, go to your directory, and enter:

```
http-server
```

Now it's time for the show! Open your browser to http://localhost:8080.

You should now briefly see "You will see this while Angular start the app!", and then "PonyRacer" should appear! Your first component is a success!

It's not really a dynamic app, and we could have done the same in one second in a static HTML page, I'll give you that. So let's jump to the next sections, and learn all about dependency injection and templating.

8.5. From zero to something better with angular-cli

In a real project, you'll probably have to set up several other things like:

- some tests to check if we're not breaking things
- a build tool, to orchestrate the various tasks (compile, test, package, etc...)

And it's a bit cumbersome to setup everything yourself, even if I think it's necessary to do it once to understand what's going on.

These past few years, a lot of small project generators have seen the light, pretty much all using the great Yeoman. It used to be the case for AngularJS 1.x, and there are already a few attempts for Angular 2.

But this time, the Google team has been working on this issue, and they have come up with something: `angular-cli`.

`angular-cli` is a command line utility to easily quick start a project, already configured with Webpack as a build tool, tests, packaging, etc...

The idea is not new, and is in fact borrowed from another popular framework: EmberJS and its popularly acclaimed `ember-cli`.

The tool is still under development, but I think it will be the *de facto* standard to create Angular 2 apps in the future, so you can give it a try.

In fact I think you should give it a try: you will have the equivalent of what we just did manually, plus a ton of cool stuff.

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

This will create a project skeleton. You can start your app with:

```
ng serve
```

This will start a small HTTP server locally, with a hot reload configuration. That means every time you are going to modify and save a file, the app will refresh in your browser.

A few other possibilities are available, like creating a component skeleton:

```
ng generate component pony
```

This will create a component file, with its associated template, stylesheet and test file.

The tool is not only here to help us develop the application, it also comes with a plugin system that will simplify a few tasks like deployment. For example, you can quickly deploy on Github Pages, using the `github-pages` plugin:

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

In the long term, this is going to be great! We'll have the same code organization across projects, a common way to build and deploy apps, and probably a huge eco-system of plugins for simplifying some tasks.

So go have a look at `angular-cli`!

Chapter 9. End of the free sample

That’s it! I hope that you enjoy this reading. If you want more of it (and you should), go buy it on the ebook website! :)

Appendix A: Changelog

Here are all the major changes since the first version. It should help you to see what changed since your last read!

By buying this ebook, you'll get all the following updates for free. Go to https://books.ninja-squad.com/claim to obtain the latest version of this ebook.

A.1. Changes since last release - 2016-10-17

Global

- Bump to `2.1.0` (2016-10-17)
- Remove typings and use `npm install @types/⋯` (2016-10-17)
- Use `const` instead of `let` and TypeScript type inference whenever possible (2016-10-01)
- Bump to `2.0.1` (2016-09-24)
- Bump to stable release `2.0.0` (2016-09-15)
- Bump to `rc.7` (2016-09-14)
- Bump to `rc.6` (2016-09-05)

From zero to something

- Update the SystemJS config for `rc.6` and bump the RxJS version (2016-09-05)

Pipes

- Remove the section about the replace pipe, removed in rc.6 (2016-09-05)

Testing your app

- Use `TestBed.get` instead of `inject` in tests (2016-09-30)

Router

- We don't need to unsubscribe from the router params in the `ngOnDestroy` method. (2016-10-07)

A.2. v1.2 - 2016-08-25

Global

- Bump to `rc.5` (2016-08-23)
- Bump to `rc.4` (2016-07-08)
- Bump to `rc.3` (2016-06-28)

- Bump to `rc.2` (2016-06-16)
- Bump to `rc.1` (2016-06-08)
- Code examples now follow the official style guide (2016-06-08)

From zero to something

- Small introduction to NgModule when you start your app from scratch (2016-08-12)

The templating syntax

- Replace the deprecated `ngSwitchWhen` with `ngSwitchCase` (2016-06-16)

Dependency Injection

- Introduce modules and their role in DI. Changed the example to use a custom service instead of Http. (2016-08-15)
- Remove deprecated `provide()` method and use `{provide: ⋯}` instead (2016-06-09)

Pipes

- Date pipe is now fixed in `rc.2`, no more problem with Intl API (2016-06-16)

Styling components and encapsulation

- New chapter on styling components and the different encapsulation strategies! (2016-06-08)

Services

- Add the service to the module's providers (2016-08-21)

Testing your app

- Tests now use the TestBed API instead of the deprecated TestComponentBuilder one. (2016-08-15)
- Angular 2 does not provide Jasmine wrappers and custom matchers for unit tests in `rc.4` anymore (2016-07-08)

Forms

- Forms now use the new form API (FormsModule and ReactiveFormsModule). (2016-08-22)
- Warn about forms module being rewritten (and deprecated) (2016-06-16)

Send and receive data with Http

- Add the HttpModule import (2016-08-21)
- `http.post()` now autodetects the body type, removing the need of using `JSON.stringify` and setting the `ContentType` (2016-06-16)

Router

- Introduce RouterModule (2016-08-21)
- Update the router to the API v3! (2016-07-08)
- Warn about router module being rewritten (and deprecated) (2016-06-16)

Changelog

- Mention free updates and web page for obtaining latest version (2016-07-25)

A.3. v1.1 - 2016-05-11

Global

- Bump to `rc.0`. All packages have changed! (2016-05-03)
- Bump to `beta.17` (2016-05-03)
- Bump to `beta.15` (2016-04-16)
- Bump to `beta.14` (2016-04-11)
- Bump to `beta.11` (2016-03-20)
- Bump to `beta.9` (2016-03-11)
- Bump to `beta.8` (2016-03-10)
- Bump to `beta.7` (2016-03-04)
- Display the Angular 2 version used in the intro and in the chapter "Zero to something". (2016-03-04)
- Bump to `beta.6` (`beta.4` and `beta.5` were broken) (2016-03-04)
- Bump to `beta.3` (2016-03-04)
- Bump to `beta.2` (2016-03-04)

Diving into TypeScript

- Use `typings` instead of `tsd`. (2016-03-04)

The templating syntax

- `*ngFor` now uses `let` instead of to declare a variable `*ngFor="let pony of ponies"` [small](2016-05-03)#
- `*ngFor` now also exports a `first` variable (2016-04-16)

Dependency Injection

- Better explanation of hierarchical injectors (2016-03-04)

Pipes

- A `replace` pipe has been introduced (2016-04-16)

Reactive Programming

- Observables are not scheduled for ES7 anymore (2016-03-04)

Building components and directives

- Explain how to remove the compilation warning when using `@Input` and a setter at the same time (2016-03-04)
- Add an explanation on `isFirstChange` for `ngOnChanges` (2016-03-04)

Testing your app

- `injectAsync` is now deprecated and replaced by `async` (2016-05-03)
- Add an example on how to test an event emitter (2016-03-04)

Forms

- A pattern validator has been introduced to make sure that the input matches a regexp (2016-04-16)
- Add a mnemonic tip to rememeber the `[()]` syntax: the banana box! (2016-03-04)
- Examples use `module.id` to have a relative `templateUrl` (2016-03-04)
- Fix error `ng-no-form` → `ngNoForm` (2016-03-04)
- Fix errors `(ngModel)` → `(ngModelChange)`, `is-old-enough` → `isOldEnough` (2016-03-04)

Send and receive data with Http

- Use `JSON.stringify` before sending data with a POST (2016-03-04)
- Add a mention to `JSONP_PROVIDERS` (2016-03-04)

Router

- Introduce the new router (previous one is deprecated), and how to use parameters in URLs! (2016-05-06)
- `RouterOutlet` inserts the template of the component just after itself and not inside itself (2016-03-04)

Zones and the Angular magic

- New chapter! Let's talk about how Angular 2 works under the hood! First part is about how AngularJS 1.x used to work, and then we'll see how Angular 2 differs, and uses a new concept called zones. (2016-05-03)