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. A gentle introduction to ECMAScript 6 2
   - 2.1. Transpilers 2
   - 2.2. let 3
   - 2.3. Constants 4
   - 2.4. Creating objects 5
   - 2.5. Destructuring assignment 5
   - 2.6. Default parameters and values 7
   - 2.7. Rest operator 9
   - 2.8. Classes 10
   - 2.9. Promises 12
   - 2.10. Arrow functions 16
   - 2.11. Sets and Maps 19
   - 2.12. Template literals 20
   - 2.13. Modules 20
   - 2.14. Conclusion 22
3. Going further than ES6 23
   - 3.1. Dynamic, static and optional types 23
   - 3.2. Enters TypeScript 24
   - 3.3. A practical example with DI 24
4. Diving into TypeScript 27
   - 4.1. Types as in TypeScript 27
   - 4.2. Enums 28
   - 4.3. Return types 28
   - 4.4. Interfaces 29
   - 4.5. Optional arguments 29
   - 4.6. Functions as property 30
   - 4.7. Classes 30
   - 4.8. Working with other libraries 32
   - 4.9. Decorators 33
5. Grasping Angular’s philosophy 36
6. End of the free sample 40

Chapter 1. Free sample

What you’re going to read is a free sample : a part giving an overview of ES6 and Angular 2 philosophy, which does not require any preliminary knowledge.

Chapter 2. 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 Angular2 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!

2.1. 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 (do I really need to name it?). 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 Angular2 itself was at first transpiled with Traceur, before switching to TypeScript.

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!

2.2. 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,
  // and can have a value from the if block
  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!

2.3. 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 PONIES_IN_RACE = 6;
```

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

I used a snake_case, FULL CAPS, to name the constant, as it's done in Java. There is no obligation to do so, but it feels natural to have a convention for constants: find yours and stick to it!

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
```

2.4. 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() {
  let name = 'Rainbow Dash';
  let color = 'blue';
  return { name: name, color: color };
}
```

can be simplified to

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

2.5. 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:

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

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

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

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

And the same is possible with arrays:

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

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

And you will only have the pony!

2.6. 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:

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

2.7. 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:

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

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:

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

2.8. 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!
  }
}
let 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:

```
let 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;
  }
}
let pony = new Pony();
pony.color = 'red'; // set color red
console.log(pony.color); // get color
```

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 {

}
let 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;
  }
}
let 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;
  }
}
let pony = new Pony(20, 'blue');
console.log(pony.speed); // 20
```

2.9. 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 its 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()
  .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 its 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`.

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

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

but more beautifully:

```
getUser()
  .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()
  .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()
  .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.

2.10. 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()
  .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()
  .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()
  .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:

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

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

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

2.11. 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`:

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

We also have a class `Set`:

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

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.

2.12. Template literals

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

```
let 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:

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

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

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

2.13. 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 races_service.js

```
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.

2.14. 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 3. Going further than ES6

3.1. 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, and 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:

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

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 an other API, you pretty much have to read the doc (or worse the function code) to know what the parameter should look like. 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 says he can't be as productive and produce 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 sheer pleasure 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 not something done easily, 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 among 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 has been later abandoned. At first they announced AtScript, as a superset of ES6 with annotations (types annotations and another kind I'll talk about 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 TypeScript was the new top notch way to write Angular 2 apps!

3.2. 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 Balmer 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 have 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've forgotten about it: you can do Angular 2 apps using it just for some parts where it really helps (more on that in a second) and pretty much 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 if you want to use it, or not, or 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 know 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.

3.3. 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
let fakeService = {
  list: () => {
    // returns a fake list
  }
};
let raceList = new RaceList(fakeService);
// now we are sure that the race list
// is the one we want for the test
```

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 has a severe limitation, because minification of your code would change 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 a few 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 4. Diving into TypeScript

TypeScript has been around since 2012, and is a superset of JavaScript, adding a few things to ES5. The more 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 last 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.

4.1. Types as in TypeScript

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

```
let variable: type;
```

The types are easy to remember:

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

In these 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`:

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

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

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

The array can only contain ponies, and the generic notation, using `<>` indicates this. You may be wondering what is the point of doing this. 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, 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
```

4.2. Enums

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

```
enum RaceStatus {Ready, Started, Done}
let race: 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}
```

4.3. 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;
}
```

4.4. 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 of course name these interfaces:

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

4.5. 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;
}
```

4.6. 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);
}

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

4.7. Classes

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

```
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`);
  }
}
let pony = new NamedPony('Rainbow Dash', 10);
// defines a public property name with 'Rainbow Dash'
// and a private one speed with 10
```

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

4.8. 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:

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

and 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 adopt this approach, and Angular 2 is doing the same. 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!

4.9. 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 have not 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`, that 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 for example "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 be definitely 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. Not on a constructor though, but it can be applied to the constructor's 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 use 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:

```
let 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, 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, like 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: 'home'})
class Home {

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

}
```

The `@Component` decorator is added to the class `Home`. When Angular 2 will load 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.

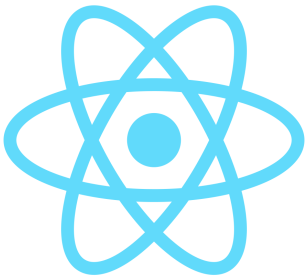
So my advice would be to give TypeScript a try! All my examples from here will use it. It's not very intrusive, as you can use it just where it's useful and forget about it for the rest. If you really don't like it, it will not be very difficult to switch to ES6 with Babel or Traceur, or even ES5, if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

Chapter 5. Grasping Angular’s philosophy

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

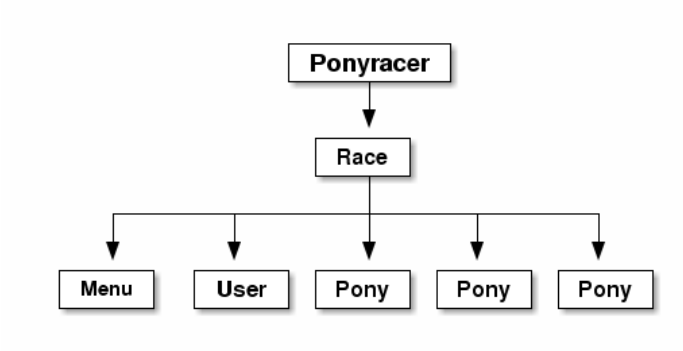
First and foremost, Angular 2 is component oriented. You will write tiny components and, together, they will constitute a whole application. A component is a group of HTML elements, in a template, dedicated to a particular task. For this, you will usually also need to have some logic linked to that template, 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 yet completely supported by browsers, 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, is doing it that way from the beginning; EmberJS and AngularJS have their way to do something similar; and newcomers like Aurelia or Vue.js are betting on building small components also.



Angular 2 is not alone in this, but they are among the first (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, and they wanted to use a feature that is not (yet) a standard: 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've just spent two chapters on these things!

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

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

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

  race: any;

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

And the template looks like this:

```
<div>
  <h2>{{ race.name }}</h2>
  <div>{{ race.status }}</div>
  <div *ng-for="#pony of race.ponies">
    <pony [pony]="pony"></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. In a perfect world, you will take available components from the community and just put them in your app, by adding them in the hierarchy somewhere.

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

There is another concept that is at the core, and that is Dependency injection (often call by its little name, DI). This 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, and seeing the progress bar going 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 has still the magic feeling it had in v1, where changes are automatically detected by the framework and applied on the model and the views. But it is done in a very different way than it was: the change detection now uses a concept called `zones`. We will look into this of course.

Angular is also a complete framework, with a lot of help for doing 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 by the beginning: bootstrapping an app and write our first component.

Chapter 6. End of the free sample

That’s it! I hope that you enjoy this reading, and that you want more of it! :)