Angular 2 IN ACTION

Jeremy Wilken

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Angular 2 in Action
Version 4

For more information on this and other Manning titles go to
www.manning.com

welcome

Thank you for purchasing the MEAP for *Angular 2 in Action.* We've been working hard on this book and are excited that you've decided to join us as we create this comprehensive guide. This book is intended for intermediate level web developers who are already familiar with the essentials of JavaScript.

The book is designed in three parts, and aims to give you a complete picture of the Angular 2 ecosystem. In Part 1, we focus on the basics of Angular 2 and modern application development. Chapter 1 contains an overview of the technologies and techniques you'll use throughout the rest of the book, such as key features of TypeScript, newer ES6 syntax and concepts, and details about the tools used. In Chapter 2, you'll build an entire Angular 2 application from scratch. You'll learn about the primary pieces of an Angular 2 application and see how they work together. Then in Chapter 3, we'll step back in time and build the same application using Angular 1 as a way to learn about the differences between Angular 1 and 2 if you are an Angular 1 developer or are considering how to upgrade existing applications.

In Part 2, we're going to dive deep into the core parts of Angular 2, such as Components, the Router, and Services. These chapters are designed to give you a high level of proficiency in the core aspects, and explore the concepts with many examples.

Finally, Part 3 will provide insight into advanced and related skills that should be part of any professional Angular 2 developer's toolkit, such as testing, tooling options, and performance testing. As the Angular 2 ecosystem continues to evolve, we'll provide additional content about techniques such as server-side rendering and how to write Angular 2 applications using ES5 instead of TypeScript.

Angular 2 is still in development, and many useful tools and features are still in flux. We aim to incorporate as much information about the ecosystem as is possible or practical to support professional Angular application development. With the speed at which Angular 2 is changing, the content in this online version may be slightly dated from time to time. Have no fear, we are working hard to provide routine updates that will keep the content of the book in sync with the most recent versions of Angular 2, we just ask for some patience!

We're also very interested in your feedback. If you find something that is dated, that could be explained better or in more detail or is flat-out missing, let us know. We want this book to serve your needs and answer your questions. There are two places to provide input. The first is the Author Online forum, which is great for asking questions about content or making suggestions. If you have code-specific questions or suggestions, you can also submit issues on the GitHub projects. Your feedback and input will be vital to making this the best book it can be.

—Jeremy Wilken

brief contents

PART 1: GETTING ACQUAINTED WITH ANGULAR 2.X

1 Introducing Angular 2

2 Building your first Angular 2 App

3 AngularJS 2 for Angular 1 Developers

PART 2: DIGGING INTO ANGULAR 2

4 Views and templates

5 Creating services and using Http

6 Using directives and pipes in components

7 Pipes for transforming data on display

8 Using forms and events

9 Routing

PART 3: PROFESSIONAL ANGULAR 2

10 Various tooling options

11 Testing

12 Documentation

APPENDIXES:

A Basic Pre-Reqs

1

Introducing Angular 2

This chapter covers:

- Why any framework?
- Why Angular 2?
- Angular 2 prerequisites

There was a time many internet years ago when any kind of dynamic action connected with a web page was implemented on the server; requests were sent to the server for processing and re-rendered as an entirely new web page. The delay required to update the web page intrinsic in the "call and response" arrangement made for a disjointed user experience that was even worse when network latency was high or the task urgent.

That old paradigm was upended with the introduction of XMLHttpRequest that enabled the browser to make asynchronous server calls without having to refresh the page. This made it possible to deliver a more coherent user experience because user actions didn't have to be interrupted waiting for the server to return an entirely new page. It was that technology that kicked off the first wave of JavaScript tools that proved it was possible to do more in a browser than just display text and pretty pictures.

Most people would agree that jQuery won the initial round, partially because jQuery did such a good job of abstracting away all of the insanity surrounding browser variations, and allowed developers to use a single, simplified API to manage web pages. But jQuery was never intended to take the next step – to support large, complex websites that behave as if they are "real" applications. Taking steps towards larger, more robust applications ushered in an entirely new set of challenges. Although jQuery did an exceptional job of providing tools to manipulate the DOM, it offered no real guidance on how to organize large bodies of code into an application structure. And because it focused on DOM manipulation, with little support to implement application logic, it simply did not scale to build complex, event driven applications.

This desperate need for a better way to write large, maintainable JavaScript applications gave birth to an intense period of JavaScript creativity and the development of application-oriented JavaScript frameworks. In the last several years, a slew of frameworks have burst onto the scene. Some have thrived though many have quietly faded into oblivion. But a few frameworks have proven to be solid options for writing large-scale web applications that can be maintained, extended, and tested. One of the most popular, if not the most popular, is AngularJS from Google.

AngularJS is an open-source web application framework that offers unique and thorough tools for web application developers due to a well-supported code base, vibrant community, and rich ecosystem.

AngularJS 1.0 was released in June 2013. At the time it offered surprising approaches to common web application problems and appeared almost magical. It went on to command significant attention and thousands of developers adopted and used it to build a wide range of applications. AngularJS 1.x is introduced and explained in the first edition of this book although you do not need any knowledge of Angular 1.x in order to successfully use Angular 2.x. Because Angular 2 is, at least syntactically, very different from Angular 1, the primary focus will be to get up to speed on Angular 2 itself. Angular 2 is the critical path for the book whether or not you have previous experience with Angular.

However, we will include some optional sections, as needed, to help Angular 1 developers move to Angular 2 by bridging the gap and leveraging the knowledge they already have.

1.1 Why Angular 2?

There are many options today for which framework or set of tools you should use to build web applications. So, perhaps the first question we should ask ourselves is why choose Angular 2 in the first place? For those coming from Angular 1, the question might be phrased as "why should I go with Angular 2 which changes so many things about the Angular I'm familiar with?"

But to get basic on this, the question really starts with why we should use a framework at all – why not build applications with straight JavaScript so you don't run into the limitations of someone else's idea of how things should be done.

The simple answer is that once you move beyond building a trivial web site or even small applications, you encounter problems that pretty much everyone else has run into. Managing data starts to get complicated, you find yourself dealing with browser incompatibilities, you have to manage all sorts of possible competing user interactions (clicks, mouse movements, entering bad or incomplete data, etc.), you end up juggling the contributions of more than one developer work on the project and you need some kind of defined process to manage, test and deploy your code.

Everyone hits these issues, and other common problems, and every framework was created to address one or (hopefully more) of them. Angular is a feature-rich, framework that addresses many of these problems and has proven itself to be an excellent framework choice for a variety of application types.

For those coming from Angular 1, the answer to "should I adopt Angular 2" may have more than one answer. For existing, mature applications, it may make sense to stick with Angular 1. For other applications it will make sense to introduce and use Angular 2 along side Angular 1 when building new features or refactoring existing ones. For new applications, choosing Angular 2 generally makes sense because of the improvements and future-proofing it brings to the table. We will offer guidance on migration in this book, but whether you migrate an existing application to Angular 2 will require input from both the business and technology sides of the house.

For shops that are considering Angular for new application development or to replace legacy application, the decision to upgrade to Angular 2 is clearer and largely driven by what Angular 2 offers, the ease of adoption and access to competent technical resources

1.1.1 Angular 2 Benefits

At a high level, here's what Angular 2 promises and what makes it such a compelling choice for a JavaScript framework: [NOTE: for now the following is taken largely from the Angular website, we'll need to update it as we go and get experience with A2.]

- Mobile First. The Angular team has said from the beginning that Angular 2 will be "mobile first" which means it will not only support mobile features such as touch events but that it will be compact, perform well and be memory efficient.
- New Standards Support. Because Angular 2 will be used in a different world than Angular 1 in terms of browsers, the Angular team designed it to work on browsers as they will exist rather than as they existed when the project started. Instead of looking backwards to support last year's browsers (yes, we're looking at you Internet Explorer), they looked forward. This is important because new browsers support features our users will come to expect. It also means that development teams will be able to use modern, not outdated, tools and will have reasonable confidence their Angular 2 applications are not legacy the day they are released.
- Performance. Angular 1 applications of a certain size ran into performance problems. This was not necessarily specific to Angular, though it had its own set of problems; the entire industry is struggling with how to manage and write good, high-performance, large JavaScript applications. JavaScript was not originally intended to write full-fledged, sophisticated applications and browsers were not originally built to support them so Angular 2's emphasis on performance is important.
- Web components. While web component technology is not fully here yet, it is coming and Angular 2 leverages it to make applications more modular. We'll look at the features web components give us.
- Dependency injection (DI). If you are not already familiar with dependency injection, you will be by the time you finish this book. A quick summary of it is available in the Appendix. In brief, it is a way to manage sets of code that you can reuse throughout your application. For example, if you write a customized and elaborate function to manipulate date and time values you'll want to reuse it throughout your site. DI

provides a way to manage that function and make it available when and where needed. Conceptually, that sounds simple but it can get very involved when your application is large, you have many re-usable pieces and those pieces have interdependencies. Angular 2 features a sophisticated DI system that builds on and learns from the experience with Angular 1's DI.

- Routing. This is another feature that benefits from the experience of Angular 1 as it includes routing but its routing was somewhat limited. Routing is simply a system, method or convention for getting your user to the right place in your application based on the URL they request, query strings or form they have submitted. The Angular community came up with an alternative router and now Angular 2 builds on and extends that experienced to deliver a best-in-breed in routing solution.
- Typescript, ECMAScript 6, et al. Angular 2 is written in TypeScript, a super-script of JavaScript that adds optional sophisticated language features to JavaScript. The primary advantage of TypeScript is for you the developer. It helps to detect code issues early while you're writing the application rather than when you run it. Angular 2 also supports ECMAScript 6, the latest approved version of JavaScript. Using these newer, advancing languages helps improve developer productivity and can make your applications more reliable and easier to maintain. Because both TypeScript and ECMAScript 6 are transpiled, i.e. changed from one language into another. In this case, the "other" language into which TypeScript and ECMAScript 6 are changed is ECMAScript 5, the version of JavaScript that runs in most browsers.

1.1.2 Ease of Adoption

An often-overlooked factor when making technology choices is the ease of adoption that encompasses several factors:

- The relative difficulty of the technology itself
- The documentation available for it
- The community of developers that use it

When Angular 1 was released it was notoriously difficult to move beyond initial familiarity because parts of its API were complex and confusing, the documentation was poorly constructed and it didn't have a large community supporting it. Over time, two of the three of those problems were fixed: available documentation improved in quality and quantity and a dedicated and active community grew up around it.

Angular 2 is taking on, as one of the primary design goals, the first issue by simplifying and normalizing its APIs. The intention is to remove or revised the most confusing aspects and to reduce the number of concepts developers have to keep in their head to work. The core Angular team is also dedicating more time to provide basic reference documentation that is more accessible and avoids the pitfalls of the Angular 1 documentation. And, of course, the community remains intact and has been participating vigorously in Angular 2's development, direction and documentation.

The importance the Angular team puts on easing adoption bodes well for teams that want to adopt it. But this does not mean that learning and using Angular professionally is trivial. In fact, because Angular 2 also has a goal to take advantage of best practices and emerging standards for web application development, it employs technologies which may be new to most teams. In the next section, we examine those technologies.

1.2 Angular Overview

Angular 1 began as a way to help designers implement pages without having to know underlying technologies. It grew into a framework for developing sophisticated web applications and, with supporting tools and techniques that have sprouted up around it, has become an ecosystem for developing, managing and maintaining desktop and mobile mission-critical applications.

But whether you are prototyping an idea, building a small departmental application or creating an application that is your business's future you'll be working with and using a core set of Angular concepts.

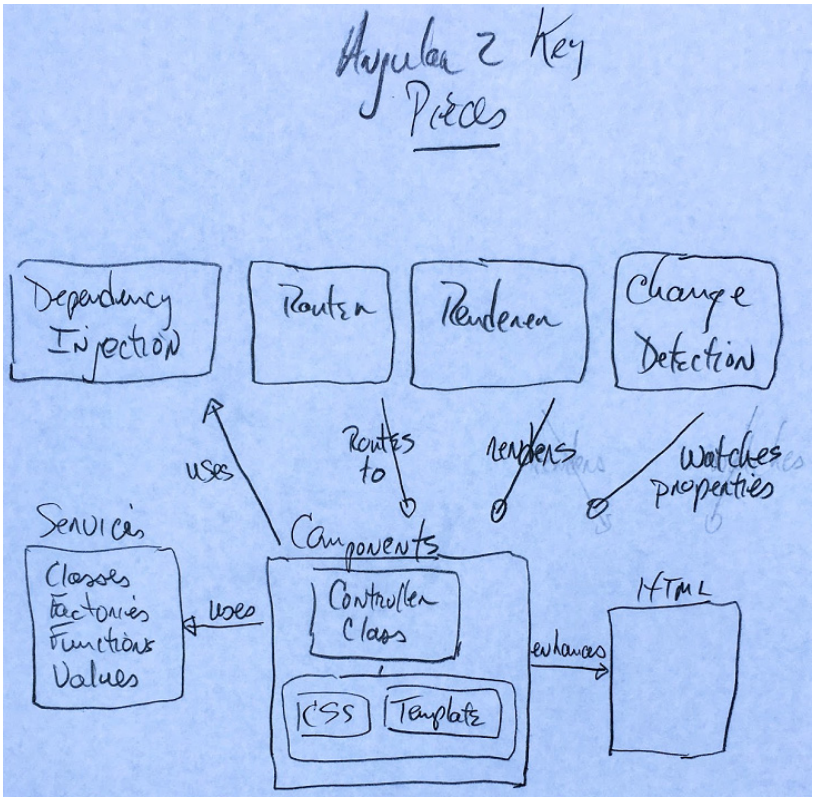

Figure 1.1: High-level view of Angular 2

"Components" are at the core of every Angular 2 application. They are a JavaScript class associated with and supporting a view, which is defined as a "portion of the screen that displays information and responds to user actions such as clicks, mouse moves, and keystrokes" (from the Angular official documentation). When building an Angular 2 application, you'll code views in HTML and CSS to display information to the user, present a form for entering data or render a control with which the user can interact. That view is associated with a controller class written in TypeScript or JavaScript that contains the logic needed by the view – it is the "code behind" the view. The controller class can create or update data (the "model") used by the view.

Together, this combination of a view and a controller form a unit that Angular 2 calls a "component." One of the properties of components is its "selector" which is the element name you use to add it to your pages. So, a component with a selector of "my-selector" is added to a page with "`<my-selector>`." Components are self-contained – the HTML and CSS you use to define their views only affects the component itself, it does not "leak" out to affect other parts of the page. Because they are self-contained you need to define what information can be "sent in" to a component from the outside and what information will be sent out when things happen inside it. All Angular 2 applications are made up of a hierarchy of components and you'll build your first components in the example application in chapter 2.

Figure 1.1 also shows that components can use "services" which are any kind of JavaScript object – classes, functions, values or objects – that you define once for re-use anywhere in your application.

In addition, Angular 2 includes several services of its own:

- Dependency Injection: This is a way to create and manage the services included in Angular or which you write. One you have defined a service, the dependency injection system is responsible for finding it when needed by a component and making it available where and when you need it.
- Router: This is responsible for associating URL paths with parts of your application. So, when the user requests `http://www.yourapp.com/home`, the application will display the home page view and when a user requests `http://www.yourapp.com/offers` the application will display the offers page.
- Renderer: As a developer you don't often deal directly with the renderer but it is responsible for rendering your component's view to the user while hooking up the code defined in the component's controller class.
- Change detection: This is responsible for watching your application for changes caused by the user entering data, information arriving from a database request or when some other event occurs. When it detects a change, it kicks of any needed updates or follow-on actions. This is what allows you to type information into an Angular 2 form and have it instantly update on another part of the page.

The above is a quick, high-level peak at the key parts of Angular 2. We'll see all of them in action in the example application in chapter 2 and dive into them in detail in Part 3. But now, let's look at the new web technologies Angular 2 leverages.

1.3 Angular 2: Leveraging New Web Technologies

Angular 2 is, of course, built on JavaScript but JavaScript itself is evolving and there are many emerging related technologies designed to help us to build more robust, maintainable JavaScript applications. We mentioned two above – TypeScript and ECMAScript – but there are others. The Angular team decided to adopt these specific technologies because they bring real value to the table and are either available today or will be soon across a variety of browsers and platforms. In this section, we'll look at these new technologies and what you need to know about them to get the most from Angular 2.

1.3.1 WebComponents

"Web components" is a term that refers to several technologies that together allow developers to create new elements that render in a browser. The idea is to put into developer's hands the ability to add to what can be done with basic HTML. When using web components technologies to create new elements, a key part of it is to make sure that the new element doesn't interact badly with existing elements – in other words to encapsulate the element's logic. This ability to encapsulate logic is a key part of web components as we'll see below.

This is a familiar idea for Angular 1 developers because it provided the ability to define "directives" which were a way to define new elements to extend HTML. But directives were one of the most complex parts of Angular 1: Angular 2 dispenses with that complexity by adopting the standards-based web component technology.

As mentioned, web components depend on several technologies including HTML, CSS and JavaScript. However, one of the set of technologies is the key to making it work: Shadow DOM. Shadow DOM is the ability of the browser to properly render a chunk of HTML and CSS that the end-user sees and can interact with, but its implementation details are hidden from the rest of the page. It is a self-contained unit of presentation.

One of the most interesting, and most difficult, aspects of Angular 1 was that it allowed the developer to create new elements and use them in mark-up just like any other valid HTML elements. To a large extent, web components do the same thing, at least in terms of the presentation part of a new element and so web components have been adopted by Angular 2. Let's look at an example, beginning with the mark-up part of a web component:

Listing 1.1 Mark-up that will define a simple component, '<first-comp>'

```
<div>
  <first-comp style="background: red;">Joe</first-comp> #A
</div>

<template id="bff"> #B
  <style> #C
    div {
      border-radius: 60px;
      background: yellow;
      font-size: 18pt;
      width: 8em;
      height: 2em;
      text-align: center;
    }
```

```
  </style
  <span>
    <div>
      My BFF: <content></content> #D
    </div>
  </span>
 </template>
```

#A HTML mark-up including the "first-comp" web component element
#B The new "template" tag which doesn't render but can be accessed by JavaScript
#C The start of the template content
#D The "content" tag that marks where out content will display

If the above appeared by itself within an HTML page and nothing else was done, only "Joe" would display because the contents of the `<template>` tag does not render but allows you to define a block of HTML that will not render or affect any other part of the page's rendering although its content can be reached with JavaScript.

The `<content>` tag inside the template is directly related to Shadow DOM so let's take a look at how to turn the above into a web component.

Listing 1.2 JavaScript that uses Shadow DOM

```
var shadow = document.querySelector('first-comp').createShadowRoot(); #A
var tmpl = document.querySelector('template'); #B
var clone = document.importNode(tmpl.content,true); #C

shadow.appendChild(clone); #D
```

#A Select our new element and creates a "shadow root" node under it
#B Get a reference to the <template> tag
#C Imports and creates a copy of the template content
#D Appends the cloned template into the shadow root

With a copy of the contents of the template tag added under the shadow root, the browser will render the contents of the shadow root as if it was part of the page's DOM. In full, here's what happens:

1. We create a shadow root node as a child of the `<first-comp>` element.
2. We append the contents of the `<template>` tag, including the CSS and mark-up, into the `<first-comp>` element's shadow root.
3. We insert the original contents of the `<first-element>` element ("Joe") into the template at the `<content>` element.
4. The browser renders the HTML enclosed by the shadow root as if it was located within the `<first-comp>` element.

However, if you try to access the contents of `<first-comp>` programmatically, such as by calling: `document.querySelector('first-comp').innerText`, you'll get "Joe", not what you see on-screen. The contents of the shadow root only displays to the user, but in every other

way is encapsulated and protected from direct outside interaction. The above allows us to define two new terms:

- Light DOM: This refers to the mark-up directly contained within the `<first-comp>` element ("Joe" in the above example).
- Shadow DOM: This is the set of DOM elements we added to the shadow root (which were originally defined within the <template> tag)

Perhaps these terms originated because the content in the `<first-comp>` element is like a light that projects into the shadow. Or perhaps that's just a bit too whimsical. The above code is available as a codepen at: http://codepen.io/daden/pen/pJYPEz but we've only scratched the surface of Shadow DOM, additional information is available on the web at "Shadow DOM 101" (http://www.html5rocks.com/en/tutorials/webcomponents/shadowdom/).

Definition: Shadow DOM

Shadow DOM is a new capability of browsers. The DOM is an API for interacting with HTML. It represents an HTML document as a hierarchy of nodes so each type of "thing" we're used to dealing with in HTML – elements (tags), attributes, text, comments – are a type of node within the DOM hierarchy. The DOM API lets us do things like adding a new element node that causes the browser to update what it has rendered.

Shadow DOM lets you add a new kind of node to the DOM – a "shadow root" – that can itself contain other elements, CSS and JavaScript. The feature that distinguishes the shadow root is that browsers will render it so users see it, but it is not accessible as part of the rest of the DOM which just means that things that happen within a shadow root stay within a shadow root. They do not affect anything outside it.

Web components are not yet fully supported in all browsers and the above covers only one aspect of using them, but this is the key data you'll need to work with Angular 2. The Angular team decided to use web component technology for Angular 2 as it provides both a technology and a set of existing and well-defined terms for describing how to build new elements for the web.

Angular 1 Note: Directives vs Web Components

One of the most common complaints about Angular 1 was the complexity of the directive API and generally dealing with directives. Web components in many respects resemble Angular 1 directives because they give developers tools for creating new HTML elements to encapsulate behavior. But the terminology used to describe web components stands on its own.

As you can see from the simple example above, making a web component work involves markup as well as code. Google Polymer (https://www.polymer-project.org) is often mentioned in connection with web components. As a project it is intended to:

- Polyfill existing browsers so those that do not yet support web components are able to.
- Abstract the creation of web components to make it more declarative and simpler.
- Provide a set of pre-built controls using web component technology

Because discussions of web components usually end up interspersed with discussions about Polymer, it may appear they are the same thing. They are not.

Web components are a set of features and capabilities browsers expose to allow developers to build their own control or widgets.

Polymer is a library based on the web component specifications that makes it easier for developers to leverage web component technology.

Angular 2 leverages web components.

1.3.2 ECMAScript 6

This and the following section will take a look at two languages that are used or related to Angular 2: ECMAScript 6 and TypeScript. If you followed the early days of Angular 2, the language story may have been a bit confusing. At one point, word was that Angular 2 would be written in ECMAScript 6, then it was announced that an ECMAScript 6 dialect created by the Angular team, named AtScript, would be used. Later, the Angular team began working with the TypeScript team at Microsoft and AtScript was retired. So, what is the story with regards to the language in which Angular 2 is written? And which language(s) can we, as developers, use to code our applications?

Angular 2 is written in TypesScript which was updated to adds features, most notably annotations, that were the original reasons behind creating AtScript. Annotations are targeted for ECMAScript 7 but that is a long way off and they are here now in TypeScript.

The above relates to Angular 2 itself because it is written in TypeScript. When you develop an Angular 2 application, you have a choice of languages. You can use ECMAScript 5 (the version currently most common in browsers), ECMAScript 6 and/or TypeScript. Because ECMAScript 6 and TypeScript are transpiled down to ECMAScript 5, using either of them will run in today's browsers. Even better, you can mix and match languages when building or upgrading applications in Angular 2.

In this section, we'll take a quick look at ECMAScript 6, in the next section we'll peak at TypeScript. Both are well covered elsewhere but these sections give an introduction to the characteristics of the languages most relevant to Angular 2 development. In ?? we cover how to set up your development environment to use ECMAScript 6 and TypeScript. [NOTE: Need to decide where that will be…might be in the Hello World application or could have specific directions on setting up an environment and tooling in an Appendix.]

ECMAScript 6 was approved in June 2015 and although browser manufacturers are working to support it, you will likely use a transpiler to turn it into ECMAScript 5 so it runs in any

browser. To see which browsers and which transpilers support ECMAScript 6 features see: https://kangax.github.io/compat-table/es6/.

ECMAScript 6, or ES6, adds many features to JavaScript which are tersely summarized at: https://github.com/lukehoban/es6features/blob/master/README.md. For thorough, in-depth coverage of ECMAScript 6, see the online resource *Exploring ES6 at*: http://exploringjs.com/es6/.

For now, let's look key ES6 features that apply to Angular 2 application development:

CLASSES

ECMAScript 6 adds a syntax for defining classes within JavaScript. In effect it is syntactic sugar on top of existing JavaScript capabilities but it makes code easier to read. In this section we assume you are familiar with JavaScript prototypical inheritance and related terminology such as "constructor functions", "instantiate", "prototype", "static" and "instance" member. If not, see Appendix ?? entitled "JavaScript prototypical inheritance".

ES6 classes are a way to group data and behavior together. Because ES6 classes become ES5 JavaScript constructor functions after transpiling, you're not required to use ES6 class syntax but doing so helps to future-proof your code.

So, let's take a quick look at the ES6 class syntax and then see what it transpiles to:

Listing 1.3 ES6 Class

```
class MotorVehicle{
  constructor(name) { #A
    this.name = name #B
  }
  static generateVin() { #C
    console.log("generate and return a new VIN")
  }
  start() { #D
    console.log("get going")
  }
  get gas() { #E
    console.log("return the gas level");
  }
  set gas(amount) {
    console.log("add " + amount + " gallons of gas to the tank")
  }
}
```

#A When transpiled this becomes a constructor function, "MotorVehicle"
#B Sets an instance property when the class is instantiated
#C Becomes a static method on the generated constructor function
#D Is added as a method to MotorVehicle.prototype
#E Defines a getter/setting property for the generated constructor

The syntax is pretty intuitive and pretty much does what you'd expect. Here's what the generated output looks like:

Listing 1.4 Generated ES5

```
var MotorVehicle = (function () {
  function MotorVehicle(name) { #A
    this.name = name;
  }
  MotorVehicle.generateVin = function generateVin() { #B
    console.log("generate and return a new VIN");
  };
  MotorVehicle.prototype.start = function start() { #C
    console.log("get going");
  };
  Object.defineProperty(MotorVehicle.prototype, "gas", { #D
    key: "gas",
    get: function get() {
      return this._amount;
    },
    set: function set(value) {
      this._amount = value;
    },
    enumerable:false,
    configurable: true
});
return MotorVehicle;
})();
```

#A This is the generated constructor function
#B The static method has been added to the constructor function directly
#C The instance method has been added to the constructor's prototype
#D The get/set properties are added to the constructor's proptotype

A couple of things to note about using ES6 classes:

- Instance properties. The syntax lets us define an instance function member that is shared across instances but there's no short-hand way to set an instance property other than setting it into `this` in the constructor. While this could seem like an omission, in fact it protects against potential errors. Because JavaScript classes inherit prototypically, adding a property to MotorVehicle.prototype would mean that all instances would share the same value, until an instance sets its own value. This ambiguity – am I getting a shared value or a local one? – is too dangerous to bake into the syntax.
- The only way to instantiate an ES6 class is by using the "new" operator such as: `let mv = new MotorVehicle("Mustang")`. This means you can't use Object.create() to instantiate an ES6 class.

Classes support inheritance using the "extends" clause:

Listing 1.5 ES6 Class inheritance

```
class MotorCycle extends MotorVehicle { #A
  constructor(name, style) {
    super(name); #B
    this.style = style;
  }
  doWheely() {
```

```
    console.log("Do a wheely!);
    console.log(this.name, this.style);
  }
}
```

#A Creates the MotorCycle class so it inherits from MotorVehicle
#B Required call to "super()" to invoke the parent's constructor function

The call to super()should be placed as the first line of the constructor because it must appear before any references to `this`. Other than that limitation, child classes work as expected – they inherit members from their parent and within the child `this` refers to the child instance.

ARROW FUNCTIONS

Arrow functions are an alternative, shorter, way to define anonymous functions while also handling some of the confusions that can surround what *"this"* keyword refers to in some contexts. The arrow syntax is syntactic sugar that ensures the function it defines runs using the same "this" context as where it was defined. This is probably easier demonstrated than described. The classic scenario to demonstrate the need for the arrow function is a DOM event callback. Let's start with a simple HTML page available at: http://codepen.io/daden/pen/WvWLZZ.

```
<html>
<head>
  <title>different 'this'</title>
</head>
<body>
  <div id="first">Click Me!</div>
</body>
</html>
```

We'll create a class which has a method to add an event listener to the div, instantiate the class and then call the addEvent() method.

Listing 1.6 Demonstrating potential confusion about "this"

```
class HandleDOM {
  constructor() {
    this.myVar = 'It changed!' #A
  }
  addEvent() {
    let elem = document.getElementById('first'); #B
    elem.addEventListener('click', function (evt) { #C
      console.log("original content", evt.target.textContent) #D
      evt.target.innerText = this.myVar || "'this.myVar' not defined"; #E
      console.log("after change", evt.target.textContent) #F
      console.log("does evt.target == 'this'", evt.target === this); #G
    })
  }
}
let o = new HandleDOM();
o.addEvent() #H
```

#A Define a property of the class
#B Get the "first" div
#C Add event listening using and pass in an anonymous function
#D Log out the original content, "Click Me!"
#E Set the content to either the local property or a message
#F Log out the text after the change which will be the message
#G Confirm "this" is set to the event target

If we change the call back to use the arrow function we get a different result

Listing 1.7 Arrow function

```
addEvent() {
  let elem = document.getElementById('first');
  elem.addEventListener('click', (evt) => { #A
    console.log("original content", evt.target.textContent)
    evt.target.innerText = this.myVar || "'this.myVar' is not defined"; #B
    console.log("after change", evt.target.textContent) #C
    console.log("does evt.target == 'this'", this === evt.target); #D
  })
}
```

#A Using the arrow syntax to define an anonymous function
#B In this case 'this.myVar' will be defined
#C The text will be "It changed!"
#D Returns false because "this" is not set to "evt.target"

In short, the arrow function guarantees we'll get the "this" where the function is defined rather.

Arrow functions also save some typing since they don't require the "function" keyword. Arrow functions can be written

```
() => { … } #A
x => { … } #B
(x,y) => { … } #C
```

#A Empty parens can be used if there are no parameters
#B The parens can be omitted if only one parameter
#C Parens required if multiple parameters

Likewise, there is more than one way to specify a return value from the function body:

```
   X => { return 2 * y } #A
   x => 2 * y #B
```

#A A statement block requires an explicit "return"
#B An expression will return implicitly

Arrow functions can significantly shorten the typing required for some common tasks – compare the first version written in ES5 with the second version that uses ES6's arrow syntax to return the same value:

```
let myVar = [1,2,3,4].filter(function(x) { return x>2 })
let myVar = [1,2,3,4].filter( x => x > 2 )
```

ES6's arrow syntax is a nice addition to the language that helps reduce typing, can make your intention clearer and avoids common errors resulting from the different ways JavaScript sets `this` in different contexts.

TEMPLATE STRINGS

Template strings are a new type of string literal with the following characteristics:

- They are enclosed in backticks ("`")
- They can span multiple lines
- They can include expressions that JavaScript will interpolate

For example, compare the ES6 using a template string and the ES5 version of the same code:

ES6

```
var name = "Chloe", gender = "girl";

`${name} is a

very big ${gender}!`
```

ES5

```
var name = "Chloe",

    gender = "female";



name + " is a\nvery big " + gender + "!";
```

This is not only helpful for composing strings with interpolated values, ES6 also lets you use template strings when defining object properties such as:

```
var obj = {};
obj[`${name}_${gender}`] = 7; #A   // obj.Chloe_female = 7
```

#A Becomes: obj.Chloe_female = 7

Using template strings can make your code easier to read and maintain – and help to eliminate annoying errors that can arise from trying to compose strings using the concatenation operator ("+") and having to correctly nest single and double quotes.

DESTRUCTURING

JavaScript has always supported various common data structures such as arrays and objects and we can put values into structures using object or array literals such as:

```
  let obj = { first: "Brianna", last: "Aden" };
  let ary = [1, 2, 3, 4];
```

ECMAScript 6 now allows us to get values out of these structures using a similar syntax:

```
  let {first: f, last: n} = obj; #A
  let [first, ,third] = ary; #B
```

#A Assigns “Brianna” to “f” and “Aden” to “n”
#B Assigns 1 to “first” and 3 to “third”

In the object example, the destructuring syntax names the properties to get and, for each, a target variable to which the property’s value is assigned. In the array destructuring, the target variables are just listed. In both cases, the destructuring format looks like structuring syntax but is distinguished by being on the assignment side of the equals sign.

Destructuring also supports a rest operator (three dots – “…”) that extracts the balance of items in an array into another array. For example using the array defined above:

```
let [a, …b] = ary;
```

#A Assigns 1 to “a”; assigns [2,3,4] to “b”

The three dots destructuring operator is overloaded as we’ll see in the next section when we look at the spread operator.

DEFAULT + SPREAD

With ECMAScript 6 it is now possible to define default values for function parameters inline:

```
let fn = function(x=0, str='foo') {
  console.log(x, str); #A
}
```

#A By default x=0 and str='foo'

The spread operator – which looks just like the rest operator described above – spreads out an array into a list of arguments:

```
let ary = [1, ‘bar’];
fn( …ary ) #A
```

#A Equivalent to fn(1, ‘bar’)

Or spreads the elements of an array into items in an array literal:

```
let ary2 = [2, 3, …ary, 4]; #A
```

#A Becomes [2,3,1,’bar’,4]

You may have noticed the action of the spread operator is almost the opposite of the rest operator described in the previous section: the spread operator turns array elements into function arguments or into elements in another array whereas the rest operator destructures array elements out of an array. You distinguish the two by context.

LET + CONST

`let` and `const` are new ES6 ways to declare variables that should be used instead of `var` declarations when writing new code or when refactoring. Neither `let` nor `const` provide one-for-one replacements of `var` so it is not possible to blindly replace `var` in a code base.

Both `let` and `const` provide block level scoping for variables whereas `var` provides function level scoping. What does that mean? Let's look at an example, first using `var`:

```
var x = 'foo';
var myFunc = function(num) {
  if(parseInt(num,10)) {
    var x = 'bar'; #A
    var y = 'whatever'; #A
  }
  console.log(x, y); #B
}
myFunc('a') #C
```

#A Variables are hoisted to beginning of MyFunc
#B Outputs "undefined undefined"
#C Invoking the function so the if-block does not run

The console.log outputs "undefined" for both x and y because x and y are hoisted to the beginning of the function definition so, in effect what we have:

```
var x = 'foo'; #A
var myFunc = function(num) {
  var x, y; #B
  if(parseInt(num,10)) {
    x = 'bar';
    y = 'whatever';
  }
  console.log(x, y);
}
myFunc('a')
```

#A This "x" is not visible to the console.log
#B JavaScript "hoists" the declarations

The effect is that when we log x and y to the console, they exist as variables but are undefined. Let's change the x declaration:

```
var x = 'foo'; #A
var myFunc = function(num) {
  if(parseInt(num,10)) {
    let x = 'bar'; #B
    var y = 'whatever';
  }
  console.log(x, y);
}
myFunc('a')
```

#A This "x" is visible to the console.log
#B Uses "let" rather than "var"

Now `x` is scoped to the block in which it exists – JavaScript does not hoist it – so the first `var x` is in effect when we log to the console and what comes displays is: "foo undefined". What do you think will happen if we change the `var y` to `let y`?

In that case, when we the console.log runs, y has not even been declared – because the "let" scopes it to the if-block – so instead of "undefined", we get an error: `ReferenceError: y is not defined`.

The takeaway is that using let limits the scope of a variable to the block in which it is defined, including any child blocks. That means that we can change the initial `var x = 'foo'` to let `x = 'foo'` and `x` would continue to be available inside the function, unless it is newly declared in the function because `myFunc` is defined within the same block as `x` even though the block is not explicit. This gives us greater control over variable scoping and prevents the automatic, but implicit hoisting of variable declarations that is potentially confusing.

One other difference – let declarations are not hoisted so if we make `y` a `let` declaration and add a console.log before it is declared, we'll also get a `ReferenceError`:

```
let x = 'foo';
let myFunc = function(num) {
  if(parseInt(num,10)) {
    console.log(y); #A
    let x = 'bar';
    let y = 'whatever'; #B
  }
  console.log(x);
}
myFunc(1) #C
```

#A Logging out "y" results in ReferenceError
#B Declaring "y" after it is used
#C Invoking the function so the if-block does run

`let` and `const` declarations are not hoisted so if the declaration hasn't occurred, the variable will not yet exist. This characteristic of let (and const) declarations – that they are not hoisted – is described by saying that they have a *temporal dead zone*, a very sci-fi sounding label that just means that where a let declaration appears in the code is important. In contrast, var declarations don't have a temporal dead zone because as we saw above, they are hoisted to the beginning of the function in which they appear. With let declarations, the JavaScript interpreter actually has to reach the declaration for the declaration to take effect.

The other new declaration type is a constant ("const") that behaves the same way as a let declaration except that when you use it you have to assign a value and that value cannot thereafter be changed:

```
const z; #A
const w = 'Liam';
w = 'Livia'; #B
```

#A Error because no value assigned
#B Error because a constant cannot be re-assigned

The new ECMAScript features we covered above are additions to the language – constructs you'll use when writing code. ECMAScript 6 also added support for a feature that is more about code organization than individual lines of code – modules – which we'll take up next.

MODULES

Modules are not brand new to the JavaScript world. Angular 1 had its own version of modules that it described as "a container for the different parts of your app." That is a pretty good generic definition for "module" and fits for ECMAScript 6 modules.

Typically, code is broken into modules to keep related things together, to enable code reuse and to break code into manageable chunks. JavaScript, until now, didn't have native support for modules but several module systems have been built for it.

On the server side – in Node.js – a type of module called CommonJS modules are used; in the browser a type of module called Asynchronous Module Definition (AMD) are often used. We won't cover those module systems here (and you don't need to know them to work with Angular 2) other than to comment that ECMAScript 6 modules attempt to combine features from both of them.

The basic idea of ES6 modules is that they allow you to package code such as variables, constants, functions or classes into a file so you can use them elsewhere. In an ECMAScript 6 module definition file, you `export` the variable, constant, function or class you want to use. In the code where you want to use the exported code, you `import` what you need from the first file. The simplest version is to define a "default" export in your module file:

```
export default class MotorVehicle() { … }
```

Then, when you want to use this class, you import it:

```
import MotorVehicle from './MyClass';
```

After the import statement the `MotorVehicle` class is available for use. A module file can contain only one default export, but can contain multiple "named" exports like so:

```
export const foo = 'bar';
export function hello(str) {
   return `Hello ${str}`;
}
export function goodbye(str) { … }
```

To import named exports you enclose the name in curly braces:

```
import { foo, hello, goodbye as bye } from './util';
```

This will make `foo`, `hello` and `goodbye` (aliased to `bye`) available. No matter how many files import a module, only one instance of it exists so memory does not bloat.

Angular 2 uses modules to keep code separate and to make it clear which code is used in a file – you'll see import statements throughout Angular 2 code.

There are other features in ES6 that you'll no doubt found useful but the above are the key ones for writing Angular 2 applications. This brings us to TypeScript, the language in which Angular 2 is being written which you can also use to write your Angular 2 applications.

1.3.3 TypeScript

TypeScript is a superset of JavaScript originally developed, and open-sourced, by Microsoft. Because it is a superset of JavaScript, valid JavaScript is valid TypeScript so any existing JavaScript will work without change as TypeScript.

TypeScript does not run natively in the browser – it is transpiled into JavaScript and the browser runs the JavaScript. In fact, TypeScript's primary purpose is not to add features to JavaScript – as is the purpose of ECMAScript 6 – but to make it easier and safer for developers to write and maintain large JavaScript applications.

JavaScript is an "untyped" language. This means that although there are different kinds of data – strings, Booleans, numbers, objects, etc. – JavaScript doesn't enforce those types which can make it more difficult to reliably write large applications. For example, in JavaScript it is perfectly legal to assign completely different kinds of data to a single variable without JavaScript complaining:

```
x = 123;
x = 'foo';
x = [1,2,3];
```

For small programs this seems convenient – you don't have to worry about what gets assigned where. But, now let's assume that you have 100,000 lines of code to maintain and your code has used the variable "name" in multiple places in a variety of ways:

```
var name = "Ashley";
var person = { name: "Liam", age: 4 };
var Employee = (function () {
    function Employee(name) {
        this.name = name;
    }
    return Employee;
})();
function showName(emp) {
    console.log(emp.name);
}
```

All the above are valid uses for a property or variable called "name." But, let's assume that for some reason in the Employee class you need to change "name" to "lastName." You obviously can't do a simple search and replace to make the change because "name" is used in many different ways.

TypeScript adds to the above example the concept of "types" – in other words specifying the kind or type of data that a variable or parameter holds. In TypeScript, we can specify the types for each of the above:

```
var name: string = "Ashley";
var person: {name:string, age: number} = { name: "Liam", age: 4 };
class Employee {
    name:string;
      constructor(name: string) {
      this.name = name;
    }
}
function showName(emp:Employee) {
```

```
  console.log(emp.name);
}
```

One immediate benefit is that now your IDE will tell you if you attempt to assign or use an incorrect value such as the following:

```
name = 123;     #A
person.name = 123; #A
var e = new Employee(123); #A
showName(123); #B
```

#A Generate errors because they are not strings
#B Function expects an Employee

A longer-term benefit comes if you need to refactor your code. Assuming you have specified types throughout your code and you need to change "name" within the Employee class and all its instances, your IDE can be smart about finding and changing the right uses of "name." Adding type information to an existing application might seem like a lot of work and it can be! That is why TypeScript does not require types, it only uses them where they exist so type information can be added incrementally to existing applications.

In contrast, for new applications development, adding type information can become routine and when writing Angular 2 applications the additional effort to add type information pays off almost immediately. This is because type information enables your IDE to give you a additional help such as in-depth code insight and write-time checking to identify common errors (such as trying to assign a string to a number) that would otherwise not show up until you hit them as run-time bugs.

So, the primary benefit of TypeScript shows up before your application runs – when you're writing or maintaining code – in addition to the following:

- TypeScript supports transpiling to ECMAScript 3, 5 and 6. This means you can write in TypeScript and generate JavaScript for today's browsers and, in the future, generate JavaScript for browsers that support ECMAScript 6.
- TypeScript lets you use ECMAScript 6 conveniences today such as classes.
- TypeScript now supports annotations, which are used in Angular 2.

The last point is interesting because annotations, which we cover below, were added to TypeScript because the Angular and TypeScript teams worked together. This close collaboration between the two teams bodes well for TypeScript continuing to support features needed by Angular 2 developers.

Note that you are not required to use TypeScript with Angular 2 even though it is written in TypeScript. If you don't want to use TypeScript, the Angular team has committed to supporting ECMAScript 5 and ECMAScript 6 for your applications. That being said, it helps to be familiar with TypeScript when studying Angular so let's look at aspects of TypeScript that apply most directly to Angular 2.

USING TYPESCRIPT

When you write in TypeScript you need to use a TypeScript transpiler to convert TypeScript files that typically end in "*.ts" into standard JavaScript files. Many IDE's already support automatic transpiling so as you edit a file, the IDE transpiles it automatically. [[TODO: Add an appendix or section that tells how to set up the common IDEs to work with the TypeScript transpiler.]] You can also include the TypeScript transpiler in your build process so that when you deploy, your application arrives as standard "*.js" files.

TypeScript supports standard data types such as strings, numbers, Booleans, etc. but there's no way for it to automatically understand types defined in third-party libraries. For example, Angular 2 defines has many functions and classes – TypeScript can't know on its own how they are defined. This is addressed with "type definition files" that tell TypeScript how a class or object is structured. You can create the type definition files yourself, which use a "*.d.ts" extension, but in most cases you don't need to because type definition files have been created for common libraries and are available at: https://github.com/borisyankov/DefinitelyTyped. A Node application, "TypeScript definition manager for DefinitelyTyped" or "tsd", is used to manage the type definition files needed by your application. tsd is available at: https://www.npmjs.com/package/tsd.

These type definition files are used by TypeScript-enabled IDE's to provide code insight and refactoring assistance for common libraries including Angular 2. The Angular type definition file is updated regularly as Angular 2 evolves.

CLASS/TYPES

TypeScript class syntax is slightly different from ECMAScript 6 syntax. Here's a TypeScript class definition:

Listing 1.7 TypeScript Class

```
class Employee {
  private state:string = 'FL'; #A
     name: string; #B
  size: number; #B
  constructor(name:string, size:number) {
      this.name = name;
      this.zize = size;
    }
      hire():string { #C
        console.log("do the hiring");
       return "Hired!";
      }
      get city():string {
        return this.city
      }
      set city(value:string) {
        this.city = value;
      }
 }
```

#A Properties can be private
#B Properties defined on the class
#C Specifying function return type

In the above "state" is marked private but, as with many TypeScript features, this only has an impact at development-time. The generated JavaScript does not enforce the "privateness" at run-time.

The above transpiles to standard JavaScript – notice there are no remnants of the type information or state's privatenes.

Listing 1.8 Generated JavaScript

```
var Employee = (function () {
    function Employee(name) {
      this.state = 'FL'; #A
      this.name = name; #B
      this.size = size; #B
    }
    Employee.prototype.hire = function () { #C
      console.log("do the hiring");
      return "Hired!";
    };
    Object.defineProperty(Employee.prototype, "city", {
      get: function () {
        return this.city;
      },
      set: function (value) {
        this.city = value;
      },
      enumerable: true,
      configurable: true
    });
    return Employee;
})();
```

#A No privateness
#B No type information
#C No type information

TypeScript supports type inheritance using the "extends" keyword to define child classes:

```
class NewHire extends Employee {
  probationPeriod: number;
  constructor(name:string, size:number, prob:number) {
    super(name,size);
    this.probationPeriod = prob;
  }
}
```

As with ECMAScript 6, the call to "super()" is required in the constructor.

DECORATORS/ANNOTATIONS

Decorators are a proposed feature for ECMAScript 7. They give developers the ability to add metadata to classes and properties and to customize them at compile-time. For example, let's assume you are defining a controller and want to associate a block of HTML with the controller. The HTML doesn't belong in the controller itself as it really is a separate thing, but for convenience you want both to be grouped together. You can use TypeScript to create a controller class and use a decorator to associate HTML with that controller.

As of version 1.5, TypeScript supports decorators. The Angular team has used the decorator functionality to create several specific annotations to implement Angular 2 features. In this case by "annotation" we just mean a decorator defined as part of the core Angular 2 features for building Angular 2 applications. An example from the Angular 2 site makes this clearer:

```
@Component({
  selector: 'greet'
})
@View({
  template: '<div>Hello {{name}}!</div>'
})
class Greet {
  name: string;
  constructor() {
    this.name = 'World';
  }
}
```

The above defines a `Greet` class which has "Component" and "View" annotations, identified with the "@" symbol. The Component annotation first tells Angular that the class is part of a component definition and specifies that it can be added to an application using the `<greet>` tag specified by the "selector" property. The View annotation tells Angular to associate the simple in-line HTML with the component definition. Under the covers when TypeScript transpiles this to runnable JavaScript, the annotation data is added to the class definition as metadata.

We'll see a lot more of these annotations but the above introduces the idea that TypeScript lets us use annotations to add metadata to class definition. You can see the full description of the ECMAScript 7 decorators proposal at: https://github.com/wycats/javascript-decorators.

As with ECMAScript 6, the serves to introduce a few TypeScript features. As we dive into Angular 2 and build applications in it, we will see these features in use. Other features will be explained when they are used.

1.4 Material Design

TODO: Not sure a section is needed on this so will hold off on it until we determine it is needed and there is more information available on Material Design.

1.5 Summary

In this chapter we've looked at why frameworks are needed, general prerequisites for studying this book and why Angular 2 is a compelling option. From there we dove into topics that relate to Angular 2 and which might be new material even for Angular 1 developers:

- Web Components: A new coming standard for building customized components for the web. It is on the way to adoption by all major browsers. A key feature of web components is the "shadow DOM" – a way for developers to hide user interface implementation details. In browsers that support shadow DOM, Angular 2 uses it. In those that don't Angular 2 polyfills.

- ECMAScript 6: The latest approved version of JavaScript that adds new features and fills longstanding gaps in JavaScript. You can write Angular 2 applications in ECMAScript 6, though you are not forced to use it. We covered several ECMAScript 6 features: classes, arrow functions, template strings, destructuring, the default and spread operators, let and constant declarations, and modules. These apply most directly to working with Angular 2 example applications and when writing your own applications.
- TypeScript: A super-set of JavaScript that is being used to write the Angular 2 itself. Developers do not have to use TypeScript for their applications but it brings significant help to the table when building large applications. Familiarity with helps understand Angular 2 even if you don't use it for your applications. We covered TypeScript classes, decorators and the tsd tool used to manage type definition files.

Now that we've looked at Angular 2's background, let's get our hands dirty – in the next chapter we build our first Angular 2 application.