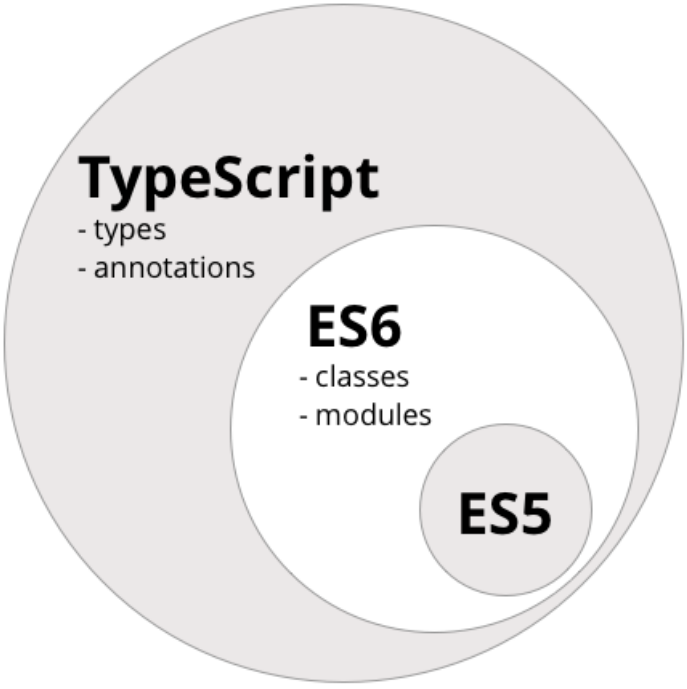
TypeScript

Angular 4 is built in TypeScript

Angular 4 is built in a JavaScript-like language called TypeScript[32].

You might be skeptical of using a new language just for Angular, but it turns out, there are a lot of great reasons to use TypeScript instead of plain JavaScript.

TypeScript isn't a completely new language, it's a superset of ES6. If we write ES6 code, it's perfectly valid and compilable TypeScript code. Here's a diagram that shows the relationship between the languages:



ES5, ES6, and TypeScript

What is ES5? What is ES6? ES5 is short for "ECMAScript 5", otherwise known as "regular JavaScript". ES5 is the normal JavaScript we all know and love. It runs in more-or-less every browser. ES6 is the next version of JavaScript, which we talk more about below.

[32] http://www.typescriptlang.org/

At the publishing of this book, very few browsers will run ES6 out of the box, much less TypeScript. To solve this issue we have *transpilers* (or sometimes called *transcompiler*). The TypeScript transpiler takes our TypeScript code as input and outputs ES5 code that nearly all browsers understand.

For converting TypeScript to ES5 there is a single transpiler written by the core TypeScript team. However if we wanted to convert *ES6* code (not TypeScript) to *ES5* there are two major ES6-to-ES5 transpilers: **traceur**[33] by Google and **babel**[34] created by the JavaScript community. We're not going to be using either directly for this book, but they're both great projects that are worth knowing about.

We installed TypeScript in the last chapter, but in case you're just starting out in this chapter, you can install it like so:

```
npm install -g typescript
```

TypeScript is an official collaboration between Microsoft and Google. That's great news because with two tech heavyweights behind it we know that it will be supported for a long time. Both groups are committed to moving the web forward and as developers we win because of it.

One of the great things about transpilers is that they allow relatively small teams to make improvements to a language without requiring everyone on the internet upgrade their browser.

One thing to point out: we don't *have* to use TypeScript with Angular2. If you want to use ES5 (i.e. "regular" JavaScript), you definitely can. There is an ES5 API that provides access to all functionality of Angular2. Then why should we use TypeScript at all? Because there are some great features in TypeScript that make development a lot better.

What do we get with TypeScript?

There are five big improvements that TypeScript bring over ES5:

- types
- classes
- decorators
- imports
- language utilities (e.g. destructuring)

Let's deal with these one at a time.

[33] https://github.com/google/traceur-compiler
[34] https://babeljs.io/

Types

The major improvement of TypeScript over ES6, that gives the language its name, is the typing system.

For some people the lack of type checking is considered one of the benefits of using a language like JavaScript. You might be a little skeptical of type checking but I'd encourage you to give it a chance. One of the great things about type checking is that

1. it helps when *writing* code because it can prevent bugs at compile time and
2. it helps when *reading* code because it clarifies your intentions

It's also worth noting that types are optional in TypeScript. If we want to write some quick code or prototype a feature, we can omit types and gradually add them as the code becomes more mature.

TypeScript's basic types are the same ones we've been using implicitly when we write "normal" JavaScript code: strings, numbers, booleans, etc.

Up until ES5, we would define variables with the `var` keyword, like `var fullName;`.

The new TypeScript syntax is a natural evolution from ES5, we still use `var` but now we can optionally provide the variable type along with its name:

```
var fullName: string;
```

When declaring functions we can use types for arguments and return values:

```
function greetText(name: string): string {
  return "Hello " + name;
}
```

In the example above we are defining a new function called `greetText` which takes one argument: `name`. The syntax `name: string` says that this function expects `name` to be a `string`. Our code won't compile if we call this function with anything other than a `string` and that's a good thing because otherwise we'd introduce a bug.

Notice that the `greetText` function also has a new syntax after the parentheses: `: string {`. The colon indicates that we will specify the return type for this function, which in this case is a `string`. This is helpful because 1. if we accidentally return anything other than a `string` in our code, the compiler will tell us that we made a mistake and 2. any other developers who want to use this function know precisely what type of object they'll be getting.

Let's see what happens if we try to write code that doesn't conform to our declared typing:

```
function hello(name: string): string {
    return 12;
}
```

If we try to compile it, we'll see the following error:

```
$ tsc compile-error.ts
compile-error.ts(2,12): error TS2322: Type 'number' is not assignable to type 's\
tring'.
```

What happened here? We tried to return `12` which is a `number`, but we stated that `hello` would return a `string` (by putting the `): string {` after the argument declaration).

In order to correct this, we need to update the function declaration to return a `number`:

```
function hello(name: string): number {
    return 12;
}
```

This is one small example, but already we can see that by using types it can save us from a lot of bugs down the road.

So now that we know how to use types, how can we know what types are available to use? Let's look at the list of built-in types, and then we'll figure out how to create our own.

Trying it out with a REPL

To play with the examples on this chapter, let's install a nice little utility called **TSUN**[35] (TypeScript Upgraded Node):

```
$ npm install -g tsun
```

Now start tsun:

[35] https://github.com/HerringtonDarkholme/typescript-repl

```
$ tsun
TSUN : TypeScript Upgraded Node
type in TypeScript expression to evaluate
type :help for commands in repl

>
```

That little > is the prompt indicating that TSUN is ready to take in commands.

In most of the examples below, you can copy and paste into this terminal and follow along.

Built-in types

String

A string holds text and is declared using the `string` type:

```
var fullName: string = 'Nate Murray';
```

Number

A number is any type of numeric value. In TypeScript, all numbers are represented as floating point. The type for numbers is `number`:

```
var age: number = 36;
```

Boolean

The `boolean` holds either `true` or `false` as the value.

```
var married: boolean = true;
```

Array

Arrays are declared with the `Array` type. However, because an Array is a collection, we also need to specify the type of the objects *in* the Array.

We specify the type of the items in the array with either the `Array<type>` or `type[]` notations:

```
var jobs: Array<string> = ['IBM', 'Microsoft', 'Google'];
var jobs: string[] = ['Apple', 'Dell', 'HP'];
```

Or similarly with a `number`:

```
var chickens: Array<number> = [1, 2, 3];
var chickens: number[] = [4, 5, 6];
```

Enums

Enums work by naming numeric values. For instance, if we wanted to have a fixed list of roles a person may have we could write this:

```
enum Role {Employee, Manager, Admin};
var role: Role = Role.Employee;
```

The default initial value for an enum is `0`, though you can set the starting enum number like this:

```
enum Role {Employee = 3, Manager, Admin};
var role: Role = Role.Employee;
```

In the code above, instead of `Employee` being `0`, `Employee` is `3`. The value of the enum increments from there, which means `Manager` is `4` and Admin is `5`, and we can even set individual values:

```
enum Role {Employee = 3, Manager = 5, Admin = 7};
var role: Role = Role.Employee;
```

You can also look up the name of a given enum by using its value:

```
enum Role {Employee, Manager, Admin};
console.log('Roles: ', Role[0], ',', Role[1], 'and', Role[2]);
```

Any

`any` is the default type if we omit typing for a given variable. Having a variable of type `any` allows it to receive any kind of value:

```
var something: any = 'as string';
something = 1;
something = [1, 2, 3];
```

Void

Using `void` means there's no type expected. This is usually in functions with no return value:

```
function setName(name: string): void {
  this.fullName = name;
}
```

Classes

In JavaScript ES5 object oriented programming was accomplished by using prototype-based objects. This model doesn't use classes, but instead relies on *prototypes*.

A number of good practices have been adopted by the JavaScript community to compensate the lack of classes. A good summary of those good practices can be found in Mozilla Developer Network's JavaScript Guide[36], and you can find a good overview on the Introduction to Object-Oriented JavaScript[37] page.

However, in ES6 we finally have built-in classes in JavaScript.

To define a class we use the new `class` keyword and give our class a name and a body:

```
class Vehicle {
}
```

Classes may have *properties*, *methods*, and *constructors*.

Properties

Properties define data attached to an instance of a class. For example, a class named `Person` might have properties like `first_name`, `last_name` and `age`.

Each property in a class can optionally have a type. For example, we could say that the `first_name` and `last_name` properties are `strings` and the `age` property is a `number`.

The declaration for a `Person` class that looks like this:

[36] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

[37] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Introduction_to_Object-Oriented_JavaScript

```
class Person {
  first_name: string;
  last_name: string;
  age: number;
}
```

Methods

Methods are functions that run in context of an object. To call a method on an object, we first have to have an instance of that object.

To instantiate a class, we use the `new` keyword. Use `new Person()` to create a new instance of the `Person` class, for example.

If we wanted to add a way to greet a `Person` using the class above, we would write something like:

```
    class Person {
      first_name: string;
      last_name: string;
      age: number;

      greet() {
        console.log("Hello", this.first_name);
      }
    }
```

Notice that we're able to access the `first_name` for this `Person` by using the `this` keyword and calling `this.first_name`.

When methods don't declare an explicit returning type and return a value, it's assumed they can return anything (`any` type). However, in this case we are returning `void`, since there's no explicit return statement.

Note that a `void` value is also a valid `any` value.

In order to invoke the `greet` method, you would need to first have an instance of the `Person` class. Here's how we do that:

```
// declare a variable of type Person
var p: Person;

// instantiate a new Person instance
p = new Person();

// give it a first_name
p.first_name = 'Felipe';

// call the greet method
p.greet();
```

You can declare a variable and instantiate a class on the same line if you want:

```
var p: Person = new Person();
```

Say we want to have a method on the `Person` class that returns a value. For instance, to know the age of a `Person` in a number of years from now, we could write:

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}
```

```
    // instantiate a new Person instance
    var p: Person = new Person();

    // set initial age
    p.age = 6;

    // how old will he be in 12 years?
    p.ageInYears(12);

    // -> 18
```

Constructors

A *constructor* is a special method that is executed when a new instance of the class is being created. Usually, the constructor is where you perform any initial setup for new objects.

Constructor methods must be named `constructor`. They can optionally take parameters but they can't return any values, since they are called when the class is being instantiated (i.e. an instance of the class is being created, no other value can be returned).

In order to instantiate a class we call the class constructor method by using the class name: `new ClassName()`.

When a class has no constructor defined explicitly one will be created automatically:

```
class Vehicle {
}
var v = new Vehicle();
```

Is the same as:

```
class Vehicle {
  constructor() {
  }
}
var v = new Vehicle();
```

In TypeScript you can have only **one constructor per class**.

That is a departure from ES6 which allows one class to have more than one constructor as long as they have a different number of parameters.

Constructors can take parameters when we want to parameterize our new instance creation.

For example, we can change `Person` to have a constructor that initializes our data:

```
class Person {
  first_name: string;
  last_name: string;
  age: number;

  constructor(first_name: string, last_name: string, age: number) {
    this.first_name = first_name;
    this.last_name = last_name;
    this.age = age;
  }

  greet() {
    console.log("Hello", this.first_name);
  }

  ageInYears(years: number): number {
    return this.age + years;
  }
}
```

It makes our previous example a little easier to write:

```
var p: Person = new Person('Felipe', 'Coury', 36);
p.greet();
```

This way the person's names and age are set for us when the object is created.

Inheritance

Another important aspect of object oriented programming is inheritance. Inheritance is a way to indicate that a class receives behavior from a parent class. Then we can override, modify or augment those behaviors on the new class.

If you want to have a deeper understanding of how inheritance used to work in ES5, take a look at the Mozilla Developer Network article about it: Inheritance and the prototype chain[38].

TypeScript fully supports inheritance and, unlike ES5, it's built into the core language. Inheritance is achieved through the `extends` keyword.

To illustrate, let's say we've created a `Report` class:

[38] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

```
    class Report {
      data: Array<string>;

      constructor(data: Array<string>) {
        this.data = data;
      }

      run() {
        this.data.forEach(function(line) { console.log(line); });
      }
    }
```

This report has a property `data` which is an `Array` of `strings`. When we call `run` we loop over each element of `data` and print them out using `console.log`

`.forEach` is a method on `Array` that accepts a function as an argument and calls that function for each element in the `Array`.

This `Report` works by adding lines and then calling `run` to print out the lines:

```
var r: Report = new Report(['First line', 'Second line']);
r.run();
```

Running this should show:

```
First line
Second line
```

Now let's say we want to have a second report that takes some headers and some data but we still want to reuse how the `Report` class presents the data to the user.

To reuse that behavior from the `Report` class we can use inheritance with the `extends` keyword:

```
    class TabbedReport extends Report {
      headers: Array<string>;

      constructor(headers: string[], values: string[]) {
        super(values)
        this.headers = headers;
      }

      run() {
        console.log(this.headers);
        super.run();
      }
    }
```

```
var headers: string[] = ['Name'];
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
var r: TabbedReport = new TabbedReport(headers, data)
r.run();
```

Utilities

ES6, and by extension TypeScript provides a number of syntax features that make programming really enjoyable. Two important ones are:

- fat arrow function syntax
- template strings

Fat Arrow Functions

Fat arrow => functions are a shorthand notation for writing functions.

In ES5, whenever we want to use a function as an argument we have to use the `function` keyword along with `{}` braces like so:

```
// ES5-like example
var data = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach(function(line) { console.log(line); });
```

However with the => syntax we can instead rewrite it like so:

```
// Typescript example
var data: string[] = ['Alice Green', 'Paul Pfifer', 'Louis Blakenship'];
data.forEach( (line) => console.log(line) );
```

Parentheses are optional when there's only one parameter. The `=>` syntax can be used both as an expression:

```
var evens = [2,4,6,8];
var odds = evens.map(v => v + 1);
```

Or as a statement:

```
data.forEach( line => {
  console.log(line.toUpperCase())
});
```

One important feature of the `=>` syntax is that it shares the same `this` as the surrounding code. This is **important** and different than what happens when you normally create a `function` in JavaScript. Generally when you write a `function` in JavaScript that function is given its own `this`. Sometimes in JavaScript we see code like this:

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    var self = this;
    this.guitars.forEach(function(g) {
      // this.name is undefined so we have to use self.name
      console.log(self.name + " plays a " + g);
    });
  }
};
```

Because the fat arrow shares `this` with its surrounding code, we can instead write this:

```
var nate = {
  name: "Nate",
  guitars: ["Gibson", "Martin", "Taylor"],
  printGuitars: function() {
    this.guitars.forEach( (g) => {
      console.log(this.name + " plays a " + g);
    });
  }
};
```

Arrows are a great way to cleanup your inline functions. It makes it even easier to use higher-order functions in JavaScript.

Template Strings

In ES6 new template strings were introduced. The two great features of template strings are

1. Variables within strings (without being forced to concatenate with +) and
2. Multi-line strings

Variables in strings

This feature is also called "string interpolation." The idea is that you can put variables right in your strings. Here's how:

```
var firstName = "Nate";
var lastName = "Murray";

// interpolate a string
var greeting = `Hello ${firstName} ${lastName}`;

console.log(greeting);
```

Note that to use string interpolation you must enclose your string in **backticks** not single or double quotes.

Multiline strings

Another **great** feature of backtick strings is multi-line strings:

```
var template = `
<div>
  <h1>Hello</h1>
  <p>This is a great website</p>
</div>
`

// do something with `template`
```

Multiline strings are a huge help when we want to put strings in our code that are a little long, like templates.

Wrapping up

There are a variety of other features in TypeScript/ES6 such as:

- Interfaces
- Generics
- Importing and Exporting Modules
- Decorators
- Destructuring

We'll be touching on these concepts as we use them throughout the book, but for now these basics should get you started.

Let's get back to Angular!